yet another uzbl tutorial

not authorized by the fine folks at uzbl.org

version 1.224, Mon Jan 19 20:39:47 PST 2015
*** currently being edited ***

Bill Evans, wje@acm.org

The canonical link to this tutorial is http://uzbl.mariposabill.com.
See what's new in this tutorial here.

The code and other suggestions in this tutorial come without warranty, express or implied. If you break anything, you get to keep the pieces. Although I welcome e-mail, I cannot guarantee to offer support for the code and other suggestions in this tutorial.

go to top go to list of contents

introduction

go to top go to list of contents

uzbl (pronounced like the word "usable") is a lightweight web browser. The core browser is devoted to actual web browsing. Everything else: cookies, URI transformation, browsing history, and the like, is handled outside the core browser, in external programs (typically Python scripts).

Before continuing with this tutorial, please read the wikipedia article about uzbl. Always remember the baseline reference site, the home for uzbl: uzbl.org.

There's been some discussion on whether uzbl is the best lightweight browser. Such discussion, in my opinion, misses the true advantage of uzbl: the ability to fine tune many aspects of its behavior through external programs. If that's not what you're looking for, if what you're looking for is simply a small-footprint browser, check out the list here.

This tutorial is based on the uzbl package provided for Debian 7.2.0 (Wheezy), whose commit string is "228bc38", and also the official release of uzbl as of 2015 Jan 15 (commit string "2012.05.14-1113-g69fa417"). The package provided for your system may do things a little differently, but there should be no major problem with that.

This tutorial is not a complete reference guide to uzbl. It merely encourages you to put your toes into the water, and then get your feet wet, and then splash around some as you modify uzbl to do your will. Some of the suggested changes are quite sloppy, with a clumsy Python coding style. Tighten them up if you wish.

This tutorial is divided into three parts. The first part shows how to use uzbl-tabbed almost straight out of the box, with very few modifications. The second part shows how to make uzbl do what you want, by either changing the config file or changing Python scripts (including uzbl-tabbed). The third part introduces compiling uzbl-core. Changing the C source for uzbl-core is certainly possible, but is beyond the scope of this tutorial.

There are gaps in this document which I will probably never fill, mostly because of time constraints. Those gaps are labeled *****. Some of these are because I haven't taken the time to investigate them. If you know how to fill any ***** in this document, e-mail me.

To continue reading past the list of contents, click here.

go to top go to list of contents

what's new

go to top go to list of contents

Thu Jan 15 03:21:49 PST 2015

Added a section in which we download, build, and install the latest official release of uzbl.

go to top go to list of contents

list of contents

go to top go to list of contents

go to top go to list of contents

basic prerequisites

go to top go to list of contents

uzbl runs on Unix-like operating systems. If you're running Microsoft Windows®(spit), I know of no way to run uzbl there. You can run uzbl on a Macintosh® system, but that's beyond the scope of this tutorial.

Some claim that you must know how to program if you want to use uzbl. Not so. Coding skills will be useful if you want to customize uzbl in fine detail, but otherwise no. The tricky part, though, is that if you end up wanting to customize uzbl, you'll end up learning to code a little, even though you don't want to. That's fine. It builds character.

You need to be able to know your way around a shell command prompt, and how to edit a text file. (This is not the same as using a word processing program.) It would help if you're used to the "vi" text editor, or workalikes such as vim or elvis, because out of the box, uzbl has "key bindings" (what happens when you press certain keys) similar to those in vi.

Use a uzbl package offered for your OS (e.g., Linux) distribution. You could compile the newest source from uzbl.org instead, but that is not for the faint of heart, and how to do so is beyond the scope of this tutorial.

This will be a bit of a journey. The rewards, in terms of both lightness of weight and degree of control over your browsing experience, are worth it.

go to top go to list of contents

this page comes in two editions

go to top go to list of contents

When this tutorial was first written, uzbl was in alpha. This means that features could change, and so they have. In particular, the syntax for some internal uzbl commands has changed. This tutorial will steer you around some of the syntax changes. But the "set" internal uzbl command is used enough in this tutorial that we'll be handling that one specially.

The earlier version of the syntax of the "set" command included "="; the later version does not. Here's an example of each:

```
set status_top = 1
set status_top 1
```

A tutorial written to accommodate both versions of the "set" syntax would be awkward to write and awkward to read, so there are two editions of this tutorial: the normal edition, and the old-set-syntax edition. Which one do you need? The answer can be found in the default configuration file that comes with your uzbl package. That file can usually be found in one of two places (that is, it has one of these two names):

```
/usr/share/uzbl/examples/config/config
/usr/local/share/uzbl/examples/config/config
```

So log in on your machine and enter this command at the shell prompt, adjusting as necessary for the proper config file location:

```
grep status_top /usr/share/uzbl/examples/config/config
```

You should see one of these two lines:

```
set status_top       = 0
set status_top       0
```

If you see the line with the equals sign, you'll be using the older syntax for the "set" internal uzbl command; if you see the one without, you'll be using the newer syntax.

You are currently reading the tutorial for the newer syntax. To switch to the tutorial for the older syntax, click here.

go to top go to list of contents

three flavors

go to top go to list of contents

There are three flavors to uzbl.

The first flavor is called uzbl-core. This is the bare-bones browser itself, with no configuration. It sits there and browses, nothing more. By itself, it's not usable.

The second flavor is called uzbl-browser. This flavor is actually usable, but it doesn't provide for tabbed browsing. As it runs, it calls on uzbl-core.

The third flavor is called uzbl-tabbed. As its name implies, it provides tabbed browsing. (As it runs, it calls on uzbl-browser.) This tutorial will assume that you want to use uzbl-tabbed.

go to top go to list of contents

bugs

go to top go to list of contents

I found a few bugs as I started to use uzbl; this tutorial describes for you my workarounds. You may not find it necessary to use all, or any, of the workarounds; your uzbl package may already have some or all of the bugs fixed. And you might find other bugs. I hope that reading about my workarounds will give you a useful perspective for solving your own uzbl situations. Actually, we're going to address one of those bugs before we even begin.

go to top go to list of contents

before you even begin

go to top go to list of contents

I hate to lay this on you, but there's a bug you need to address, if it's in your version of uzbl. May as well take care of it now. It's in uzbl-browser, a shell script.

If you can (and want to) act as root, it would be advisable to salt away an unmodified copy of uzbl-browser, and then make your changes.

If you want to act as a non-root user, copy uzbl-browser from its public location to a directory where your shell will find it before it finds the official copy. The PATH environment variable will help you here. For example, if you can change PATH like this (or similarly):

```
  export PATH=~/bin:/usr/local/bin:/usr/bin:/bin
```

then your shell will look in directory ~/bin before it looks in the public directories. So you need to copy the official uzbl-browser to the directory ~/bin and then make any changes to the new copy. (Be sure to use the chmod command to make this new copy executable.) I'll talk about the actual change in a moment, but first there's something else you need to know about the PATH environment variable.

If you changed the value of the PATH environment variable, you'll want to make that change happen every time you log in. You'll want to place into a shell startup file the command that sets the PATH environment variable. Where that startup file is located differs from one shell to the next; for bash, for example, you can read about startup files here.

Ok, let's get down to the potential bug. Part of uzbl is something called the event manager. There should be only one per user. By "user" I mean login name on your system. No matter how many windows you have separately running uzbl, if you're logged in as the same user on your system for all of those windows, you'll be using one event manager. When the event manager starts, one of the first things it does is to determine (imperfectly; go here for details) whether there's an event manager

already running for this user. To do so, it uses something called a pid file.

A pid file is a traditional way to keep track of whether something is already running, and if so, what its process ID (pid) is. The file simply contains the relevant pid. In theory, when the program stops running, it removes the pid file. But it's quite possible for the program to stop running without being able to remove that pid file.

So what's the bug? Take a look at the final few lines of uzbl-browser. They will probably look either something like this:

```
# uzbl-event-manager will exit if one is already running.
# we could also check if its pid file exists to avoid having to spawn it.
DAEMON_SOCKET="$XDG_CACHE_HOME/uzbl/event_daemon"
if ! [ -f "$DAEMON_SOCKET.pid" ]; then
        ${UZBL_EVENT_MANAGER:-uzbl-event-manager -va --server-socket "$DAEMON_SO
CKET" start} || \
                die_with_status 4 "Error: Could not start uzbl-event-manager"
fi
```

... or something like this:

```
# uzbl-event-manager will exit if one is already running.
# we could also check if its pid file exists to avoid having to spawn it.
DAEMON_SOCKET="$XDG_CACHE_HOME"/uzbl/event_daemon
#if [ ! -f "$DAEMON_SOCKET".pid ]
#then
        ${UZBL_EVENT_MANAGER:-uzbl-event-manager -va start}
#fi
```

Do you see the main difference? One version checks to see whether the pid file exists, and if it does, then uzbl-browser doesn't run the event manager. The other version comments out the check and calls the event manager unconditionally, letting the event manager figure out whether it's already running. The first version will cause uzbl to run very slightly faster, but will also cause uzbl to misbehave under some circumstances. Suppose the event manager stops running without being able to remove the pid file. The next time you run uzbl, uzbl-browser will see that a pid file is there, and therefore refrain from running the event manager. No event manager? Uzbl doesn't run well without one.

If your uzbl-browser makes the check, you should remove that check, so those lines look something like these:

```
# uzbl-event-manager will exit if one is already running.
# we could also check if its pid file exists to avoid having to spawn it.
DAEMON_SOCKET="$XDG_CACHE_HOME/uzbl/event_daemon"
#if ! [ -f "$DAEMON_SOCKET.pid" ]; then
        ${UZBL_EVENT_MANAGER:-uzbl-event-manager -va --server-socket "$DAEMON_SO
CKET" start} || \
                die_with_status 4 "Error: Could not start uzbl-event-manager"
#fi
```

You could, if you wanted, leave the check in but make it more thorough. That would work. See my thoughts on that here

go to top go to list of contents

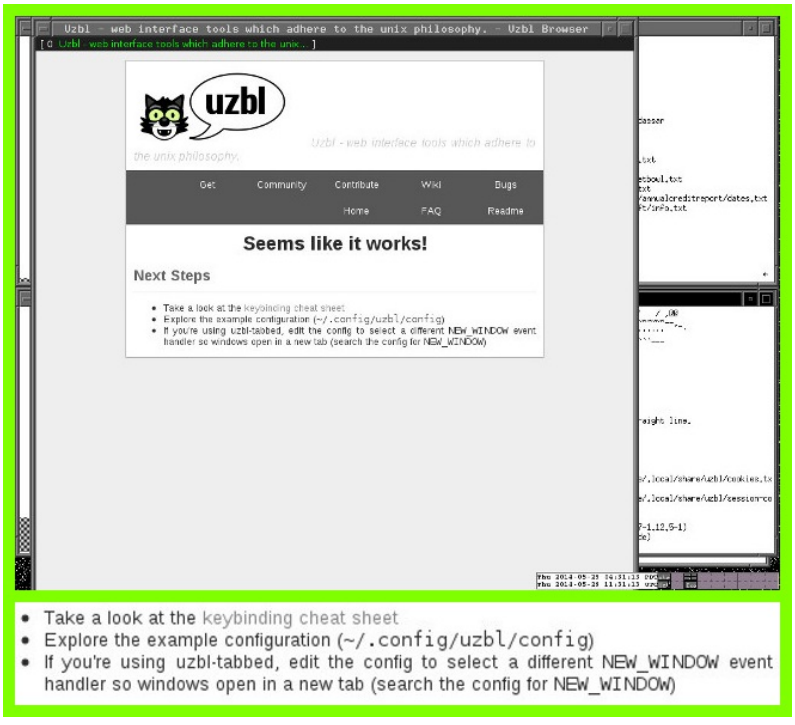
running uzbl-tabbed for the first time

go to top go to list of contents

So log in on your machine and enter this command at the shell prompt:

```
  uzbl-tabbed
```

When I did so, I got a window that wasn't wide enough for my taste, but was too tall to fit on the screen. My screen looked like what you see here. In the screen dump you see some "Next Steps". That fine print is magnified below the screen dump.



But don't do anything about those next steps just yet. The first thing to learn now is to get out of uzbl. At any point in uzbl, you can leave it by typing these four characters:

```
  <Esc><Esc>ZZ
```

Try it now. You'll get out of uzbl-tabbed. You didn't have to type <Esc> <Esc> this time, but get in the habit of doing that; it did no harm, and this simple recipe should work no matter where you are in uzbl, no matter what you're doing.

If you have more than one tab open, you'll have to type this command once for each tab that's open. Later in this tutorial, you'll learn how to add a command which you need type only once to leave uzbl-tabbed completely.

Now take a closer look at those next steps. Remember that the whole point of uzbl-tabbed is to work with multiple tabs? That won't work out of the box; you have to change a config file to make it work. Inconvenient, but true. We're going to take care of that third step almost right away. But first it's time to learn something about the uzbl config file.

go to top go to list of contents

the config file

go to top go to list of contents

There can be many files which affect how uzbl behaves. A quite important one is the config file. When a user runs uzbl, it checks for the existence of that user's config file; if the file doesn't exist, uzbl makes a fresh copy from elsewhere.

If we can refer to the user's home directory as "~", uzbl expects the user's config file to be at ~/.config/uzbl/config; if the file isn't there, then that's where uzbl will place a fresh config file. The user can then exit uzbl and change that file as desired.

Where does uzbl get a fresh copy of the config file? That depends on your uzbl installation, but it will typically be found at one of these two places:

```
  /usr/share/uzbl/examples/config
  /usr/local/share/uzbl/examples/config
```

And here I've just finishing describing the entire way uzbl config files work with each other. It could have been made more complex, but uzbl aims to be simple.

Let's give an imaginary counterexample. Let's suppose there's a software product which, unlike uzbl, works with both a system-wide config file and a user config file; the user config file can override system-wide values specified in the system-wide config file. It could be that the system-wide file sets, say, window_size to "800,600", and user config files, when first created, do not even mention window_size, and this means that the system-wide config file's value should be used. Then suppose a system administrator sets the system-wide window_size to "1600x1200". Then all users who have not overriden this value in their own config files will use the new value.

This is exactly what uzbl does not do. Once a user has a copy of the config file, that's what uzbl uses. If a system administrater changes the copy of window_size in the system-wide config file, that only affects a new uzbl user who copies the new system-wide config file to become his own config file.

But what if a uzbl user always wants to use the default config file, reflecting any changes the system administrator makes in it? That uzbl user should, after running uzbl-tabbed at least once, change file ~/.config/uzbl/config so it contains only one line:

  include /usr/share/uzbl/examples/config/config

(or wherever the default configuration file is for your system), and then make any desired changes after that one line. But if you do this, remember that many parts of this tutorial rely on the config file being modifiable, and being at ~/.config/uzbl/config, so you'll have to make appropriate adjustments as you're reading.

If you look inside a config file, you'll be asking, "What language is that config file using?" Config files are made up of internal uzbl commands; we'll discuss those later.

One final warning if you're the administrator of your system and you're tempted to change the main config file in directory /usr/share: salt away a copy of the original before you make any changes. This is a good idea for two reasons. The first is that if you mess things up, you have something to fall back on. The second is that if you ever update to a later version of uzbl, perhaps one that uses a different format of config file, you can do a diff between the original config file and your changed config file to get an idea of what you should change in the new config file.

go to top go to list of contents

changing the config file for running uzbl-tabbed

go to top go to list of contents

According to the screen shot, the config file is named:

  ~/.config/uzbl/config

So edit that file with your favorite text editor. You'll be making at least one change to this file, and possibly two or three changes.

For the first change, find the lines that look approximately like this (they may vary from one uzbl version to another):

```
# === Dynamic event handlers ==================================================

#   What to do when a website wants to open a new window:
# Open link in new window
@on_event   NEW_WINDOW     sh 'uzbl-browser ${1:+-u "$1"}' %r
# Open in current window (also see the REQ_NEW_WINDOW event handler below)
#@on_event   NEW_WINDOW     uri %s
# Open in new tab. Other options are NEW_TAB_NEXT, NEW_BG_TAB and NEW_BG_TAB_NEXT.
#@on_event   NEW_WINDOW     event NEW_TAB %s
```

Do not be alarmed if these lines don't look exactly like the ones in your config file; simply add # at the beginning of one line, and remove # from another line:

```
# === Dynamic event handlers ==================================================

#   What to do when a website wants to open a new window:
# Open link in new window
#@on_event   NEW_WINDOW     sh 'uzbl-browser ${1:+-u "$1"}' %r
# Open in current window (also see the REQ_NEW_WINDOW event handler below)
#@on_event   NEW_WINDOW     uri %s
# Open in new tab. Other options are NEW_TAB_NEXT, NEW_BG_TAB and NEW_BG_TAB_NEXT.
@on_event   NEW_WINDOW     event NEW_TAB %s
```

For the potential second change, if your config file contains a line that looks like this:

```
@on_event   REQ_NEW_WINDOW event NEW_WINDOW %s
```

then change it to this:

```
@on_event   REQ_NEW_WINDOW event NEW_TAB %s
```

But you can ignore this change if instead you see a line that looks somewhat like this:

```
@on_event   REQ_NEW_WINDOW event @- if (@embedded) "NEW_TAB"; else "NEW_WINDOW" -@ %s
```

For the potential third change, while you're still in your text editor, go to the very beginning of the config file. Then search for this string:

  cbind

If that line looks somewhat like this ...

```
set cbind     @mode_bind command
```

... then you don't need to make this potential third change. But if it looks roughly like this ...

```
@cbind  !ssl = chain 'toggle ssl_verify' 'reload'
```

then you should make a third change. The line that you found will be in the middle of a paragraph that looks (very roughly) like this:

```
# === SSL related configuration ===============================================

# Set it to certificates store of your distribution, or your own CAfile.
set ssl_ca_file /etc/ssl/certs/ca-certificates.crt
set ssl_verify 1
# Command to toggle ssl_verify value:
@cbind  !ssl = chain 'toggle ssl_verify' 'reload'
# Example SSL error handler:
@on_event   LOAD_ERROR    js var patt=new RegExp('SSL handshake failed'); if (patt.test('%3')) {alert ('%3');}
```

Move (not just copy) that whole paragraph to somewhere after the "set cbind" line. It almost doesn't matter where. A convenient place would be at the very end of the config file. A better place would be just before the final line that contains "====".

After making these changes, write out the file and get out of your text editor.

go to top go to list of contents

running uzbl-tabbed with some tabs

go to top go to list of contents

Enter this command at the shell prompt:

  uzbl-tabbed xkcd.com/1366

The page will load. In most systems, you should see the following in the window's title bar:

  xkcd: Train - Uzbl Browser

But you'll also see this in the upper lefthand corner of the window:

  [0 xkcd: Train]

That's the "tab" in "uzbl-tabbed". (Actually, most of it will be green, but I don't have a green typewriter ribbon, so there you are.) It kinda looks like a tab, but if you have more than one, you can't click on the one you want. You'll use the keyboard. (We'll discuss this later.) You'll get used to it eventually.

See the word "ABOUT" in the upper left corner of the page? Right-click it. A menu will appear. The second alternative is "Open Link in New Window". But since you changed the config file, it really means "Open Link in New Tab". Click on that choice. You'll notice that you're in a new tab, showing links to archived xkcd comics. You'll also notice that the title bar has changed to show the new page you're in. You'll also notice that there are two tabs, and they look like this:

```
[ 0 xkcd: Train ] [ 1 xkcd - A webcomic ]
```

You can guess that the tabs are numbered, starting at zero. You can also guess that the tab you're in right now is colored green (unless this tab's protocol is https:, in which case the color is gold); all others will be gray.

Now go back to the first tab by typing this user command:

```
gT
```

The letter "g" must be lower case, but for now in this demo, the letter "T" can be either upper case or lower case. As always when typing user commands to uzbl, make sure that "[Cmd]" appears in the lower left corner of the window. If, instead, you see "[Ins]" or "[Bnd]", press <Esc> until you see "[Cmd]", and then you can type your user command.

Anyway, since you've typed "gT", you'll notice that you're back in the first tab. Right-click on "ARCHIVE", choosing "Open Link In New Window". You'll notice that there are three tabs, and you're in tab 2 (the rightmost one). Each time you add a tab, the new tab goes to the end of the list, and you are transferred to that tab.

Now enter this user command, over and over, making sure that both letters are lower case:

```
gt
```

Now enter this user command, over and over:

```
gT
```

See the difference? The first user command moves you rightward in the set of tabs, wrapping around. The second user command moves you leftward, wrapping around.

For extra credit, do everything in this section of the tutorial again, but after changing this line in your config file:

```
@on_event   NEW_WINDOW     event NEW_TAB %s
```

to:

```
@on_event   NEW_WINDOW     event NEW_TAB_NEXT %s
```

and then play around with it. Each new tab now goes just to the right of the tab you were in, rather than always to the end. All tabs after the new one get renumbered so the new one will fit.

You can also add "_BG" before "_TAB". The effect will be that when you create a new tab, you'll stay in the tab you were in before, instead of going to the new tab. (This will work better after we make the fixes in the next section.)

go to top go to list of contents

the adventure begins: fixing with a squirt gun

go to top go to list of contents

Ok, it's time to fix an annoyance: the window isn't the size and shape we want. I want it to be 1000 pixels wide and 750 pixels tall. (Pick different numbers if you want.) The process of fixing this will take an unexpected turn.

So, first type <Esc> <Esc>ZZ to get out of uzbl-tabbed. Then enter this command at the shell prompt:

```
less $(which uzbl-tabbed)
```

You'll be looking at uzbl-tabbed itself, which is a Python script. Somewhere in there, you'll find this:

```
# All of these settings can be inherited from your uzbl config file.
config = {

...

  'window_size':           "800,800", # width,height in pixels.
```

Oh ho. Now we know what to do, don't we?

```
Look at our config file to discover how to set things;
set window_size to the value we want; and
run uzbl-tabbed again.
```

Looking at the config file, we discover that we can place something like this almost anywhere in it:

```
set window_size "1000,750"
```

So go ahead and do that on your machine. Then rerun uzbl-tabbed. I'll wait.

Did it work? It doesn't on older releases of uzbl, but does on newer ones. If it worked for you, skip almost to the end of this section. (You'll miss some choice droll comments about cats and squirt guns, but that's ok.) Otherwise, continue reading here.

To get the result I wanted, I treated the program uzbl-tabbed itself as an extension of the config file: I changed the default value of window_size in the program uzbl-tabbed itself; that is, I searched uzbl-tabbed for each occurrence of window_size, and changed the value I found there.

If you can (and want to) act as root, it would be advisable to salt away an unmodified copy of uzbl-tabbed, and then make your changes.

If you want to act as a non-root user, copy uzbl-tabbed from its public location to a directory where your shell will find it before it finds the official copy. The PATH environment variable will help you here. For example, if you can change PATH like this (or similarly):

```
export PATH=~/bin:/usr/local/bin:/usr/bin:/bin
```

then your shell will look in directory ~/bin before it looks in the public directories. So you need to copy the official uzbl-tabbed to the directory ~/bin and then make any changes to the new copy. (Be sure to use the chmod command to make this new copy executable.) In this case, you'll want to find the place where it sets window_size, and substitute the desired value. This is what worked for me.

If you changed the value of the PATH environment variable, you'll want to make that change happen every time you log in. You'll want to place into a shell startup file the command that sets the PATH environment variable. Where that startup file is located differs from one shell to the next; for bash, for example, you can read about startup files here.

What we've done here is to use an unconventional method to bend uzbl to our will. If one can consider uzbl to be a sort of cat, we fixed this with a squirt gun. It works for cats, and it works for uzbl. For best results with a cat, use the squirt gun when the cat cannot see you. And if you're going to use this method with uzbl, make sure the uzbl icon is not on the screen, so it can't see you applying the fix.

The question arises: when you create a new tab, do you switch to it right away, or do you stay in the tab you were in? Older versions of uzbl let you change the value of switch_to_new_tabs in the source for uzbl-tabbed, just as you set window_size in the above text; the newest version of uzbl-tabbed that I have lets you change this variable the same way, but you can also override its value in your config file. And other browsers allow you to change this in a configuration file. But wouldn't it be nice to have it either way during the same browser session, via a menu choice? This tutorial will show you how to configure uzbl-tabbed to do that.

Further, it would be nice to be able to put a new tab either (a) next to the tab you're in right now; or (b) at the end of all tabs, not as a configuration choice, but as a menu choice. Again, this tutorial will show you how to configure uzbl-tabbed to do that.

As an alternative, it would be nice to put all new tabs (c) next to the tab you're in right now, but in left-to-right order. This tutorial will show you how to do that.

Other browsers already let you choose between opening a link in a new tab or a new window. This tutorial will show you how to do that.

But for now, we won't be making any more changes. The remainder of the first part of this tutorial will focus on how to use uzbl-tabbed as it is. What a relief, huh?

go to top go to list of contents

two kinds of commands

go to top go to list of contents

As you'll discover in the next section, you can tell uzbl to do things when uzbl is in "command mode"; that is, when "[Cmd]" is in the status bar. (The status bar is the single line with a black background, usually at the bottom of the window.) But there is another kind of command, another beast entirely. Consider the ZZ user command. Enter this command at the shell prompt:

```
grep ZZ ~/.config/uzbl/config
```

You'll see something like this:

```
@cbind  ZZ               = exit
```

Your config file translates, or "binds", from the ZZ keystrokes to an internal uzbl command, the "exit" command. This internal uzbl command causes uzbl-tabbed to eliminate the current tab, and to exit entirely if no tabs are left.

The set of translations from user commands to internal uzbl commands is referred to as "key bindings". You can change the ZZ to almost anything else; or, if you wish, you can change the "exit" to some other internal uzbl command. What you can't do (unless you change your config file) is to type "exit" in command mode and expect uzbl to do anything meaningful. (Yes, you can type ":exit<Enter>" in command mode, but we'll talk about that later).

The situation is in flux, but as of 2014 June 10 the thinking is that what the user can type are "user commands", and what uzbl uses internally are "uzbl commands".

Why is what you type called a "user command"? It's easy to think it's because it's a user that types it. But actually it's called a user command because the user can modify (reconfigure) it, typically in a config file. The other kind of command is called a "uzbl command" because its existence is an inherent part of uzbl; the user can't conveniently change it. To remind you that uzbl commands are these internal things, this tutorial will call them "internal uzbl commands".

Later in this tutorial you'll discover how internal uzbl commands can be used in wonderful ways to put uzbl under your control, and to make it work like your very own browser, not some big clunky elephant that behaves at the whim of some elephant trainer, where you must be content to observe it because you hold a ticket. (We've all used browsers which are like that elephant.) For more information on user commands, read the section right after this one.

go to top go to list of contents

what happens when I press this key

go to top go to list of contents

Before we begin, note that uzbl runs in one of two modes: Command mode, and Insert mode. (This is similar to the way text editors such as vi, vim, and elvis work.) In Insert mode, what you key in will be interpreted by the web page itself; a typical example of this is a form you might be filling out on the web. In Command mode, what you key in will be interpreted by uzbl as part of a user command.

While running uzbl, you can always find out what mode you're in by looking in the lower left corner of the window. You'll see either Ins (meaning Insert mode) or Cmd or Bnd (meaning Command mode). When you load a new page, you will end up in either Insert mode ("Ins") or Command mode ("Cmd"), depending on the page's content. You'll get into the habit of checking that lower left corner whenever you load a new page.

How do you switch from Command mode to Insert mode?

To switch from Command mode to Insert mode, you can press the "i" (lower case) key, as long as you're not in the middle of entering a user command already. (If you are, you can get out of that first by pressing <Esc> once or twice.) Or instead of pressing a key, you can click in a field on the page where you'd normally enter data, as in a search box or a form.

How do you switch from Insert mode to Command mode?

To switch from Insert mode to Command mode, press the <Esc> key once or twice.

You might also be in Command mode already, but be in the middle of typing a user command. If you're in the middle of typing a user command, you'll see either "Bnd" in the lower left corner, or one or more red characters in the lower left corner. To cancel the typing of that user command and start a new one, just press <Esc>.

Now, when we talk about key bindings, we refer to what happens when you press keys while in Command mode.

Ok, let's press onward. Look at the key bindings. See the part about uzbl-tabbed? As of when I wrote this tutorial, it said "Write me." What follows is a more complete list of key bindings, based on the config file. If some of these user commands don't work for you, you may have an older version of uzbl than I do. If you have a yet newer version, you may have additional user commands available, not listed here.

To see how I constructed this list, enter this command at the shell prompt:

```
  grep '^@cbind' ~/.config/uzbl/config | less
```

Since the table below shows the key bindings in the order in which they are in my config file, you can easily determine whether you have more, or fewer, key bindings in your version of uzbl. (The "!ssl" command might be at a different place in your config file than mine.)

In this table, when you see <Ctrl>, that means hold down the Ctrl key and then press the next key. Same for <Mod1> and the Alt key, and <Shift> and the Shift key. Remember that upper and lower case do different things; "G" is not the same as "g".

To skip this table and continue reading the tutorial, go here.

	=== enter internal uzbl commands ===
:xxx<Enter>	execute arbitrary internal uzbl command "xxx". Details on internal uzbl commands are here.
	=== open a window or tab ===
w	open a new window or new tab, depending on config; exactly the same as "gw"
	=== move around on this page ===
j	scroll downward 20 pixels
k	scroll upward 20 pixels
h	scroll leftward 20 pixels
l (ell)	scroll rightward 20 pixels
<Page Up>	scroll upward one page
<Page Down>	scroll downward one page
<Ctrl>f	scroll downward one page
<Ctrl>b	scroll upward one page
<<	scroll to top of page

>>	scroll to bottom of page
<Home>	scroll to top of page
<End>	scroll to bottom of page
^	scroll to left edge of page
$	scroll to right edge of page
<Space>	scroll to bottom of page
G2345<Enter>	scroll vertically so that row 2345 of the pixels will be at the top edge of the window
<Underscore>G345<Enter>	scroll horizontally so that column 345 of the pixels will be at the left edge of the window
	=== frozen ===
<Ctrl><Shift>f	toggle frozen: when frozen is set, inhibit networking for this tab
	=== navigate ===
b	go backwards one page in your browsing history
m (not f!)	go forwards one page in your browsing history
S (upper case)	stop loading the page you're in the middle of loading
r	reload the page you're viewing
R	reload the page you're viewing, ignoring what's currently in your page cache
	=== zoom ===
+	zoom in (increase the size of text, and maybe images)
-	zoom out (decrease the size of text, and maybe images)
T	toggle the zoom type (turn on or off the changing of image size)
1 (one)	set the size of text and images to the "normal" value
2	set the size of text, and maybe images, to double the "normal" value
	=== change the page appearance ===
t	toggle whether the status bar is displayed
	=== search in this page ===
/something<Enter>	search for something. It might be tempting to press <Esc> instead of <Enter> at the end, but that breaks the "n" and "N" commands below.
?something<Enter>	search backward for something. It might be tempting to press <Esc> instead of <Enter> at the end, but that breaks the "n" and "N" commands below.
n	search for the next occurrence of the previous search (moving forward, even if the previous initial search command was "?", which is not like vim or elvis)
N	search for the previous occurrence of the previous search (moving backward, even if the previous initial search command was "?", which is not like vim or elvis)
	=== print ===
<Ctrl>p	print the current page
	=== search the web ===
ggxxx<Enter>	search google.com for xxx
ddgxxx<Enter>	search duckduckgo.com for xxx
\awikixxx<Enter>	search wiki.archlinux.org for xxx
\wikixxx<Enter>	search en.wikipedia.org for xxx
	=== "handy binds", they call these ===
sxxx<Enter>yyy<Enter>	set uzbl variable xxx to yyy. This affects only what you're doing while running uzbl now; rerunning uzbl will set all values to what they are in your config file. For example, set status_top to 1, and watch your status bar switch from the bottom of the window to the top. Set status_top to 0, and watch it flip back. It's worth browsing through your config file to see what the status variables are. You could instead type: :set xxx yyy but this is faster for most people. If you do use the :set way, be sure to put a space before and after the =.
ZZ (upper case)	eliminate the current tab. If that was the final tab, exit uzbl-tabbed. Identical to gC.
!dump	display the variable names and their values (even those not mentioned in the config file) in the window in which you typed the shell "uzbl-tabbed" command
!reload	reset the values of all variables to what they are in the config file. This is useful if you have been playing with the "s" entry in this table. Use that feature to set status_top to 1; then !reload to watch status_top get set back to 0.
	=== see uzbl events and enter internal uzbl commands ===
<Ctrl><Mod1>t	pop up an xterm window, in which you can see events and into which you can type internal uzbl commands. Try this, and see what appears in the xterm window when you move the mouse around inside the uzbl window. Then move to the new xterm window and type these internal uzbl commands and watch the status bar move from the bottom of the uzbl window to the top and back again: set status_top 1 set status_top 0 If you need to scroll back to see old lines, you can go to this window, hold down the <Ctrl> key, and press the middle mouse button. You'll see the "Enable Scrollbar" choice at the top of the menu; select that. But if you want a larger view, if you want to log all events to disk, go here. You can close this window by going to it and typing <Ctrl>D.
	=== go to a specific web page ===
oxxx<Enter>	in the current tab, go to xxx on the web (like http://www.nps.gov/yose)
O (upper case oh)	for the current tab, display the current URI in the status bar. You can then use the arrow keys to go back and forth in this URI; use <Backspace> to delete characters from it; use <Enter> to finish editing and go to that URI in the current tab; or use <Esc> to cancel this operation.
	=== set Insert mode ===
i	switch from Command mode to Insert mode, so keystrokes now go to the current web page itself, not to user commands
	=== use a quick shortcut ===
gh	"go home": go to the default home page, which is hard-wired to http://www.uzbl.org. You may wish to change this setting in your config file.
	=== open a new window or tab ===
gw	open a new window or new tab, depending on config; exactly the same as "w"
	=== change to https: ===

zs	reload the current page the in current tab, changing the protocol from http: (if found) to https:. If it fails, you'll get an error message. To get out of the error message, press the "b" key (which is the user command for going back to the previous page). Or you can go ahead and press the "Try again" button, but it's not likely to do much good, and if it fails, you'll have to use the "b" user command twice (more generally, one time for each time you tried again, plus one). Note that any web page loaded via the https: protocol will have its title in the tab list be gold, instead of the usual green.
zS	load the current page in a new window or tab, changing the protocol from http: (if found) to https:. If it fails, you'll get an error message. To get out of the error message, type ZZ (which is the user command for closing the current tab or window). Or you can go ahead and press the "Try again" button, but it's not likely to do much good. Note that any web page loaded via the https: protocol will have its title in the tab list be gold, instead of the usual green.
	=== X selection handling ===
yu	copy the current URI to the primary selection
yU	copy the URI over which the cursor is hovering to the primary selection
yy	copy the title of the current document to the primary selection
ys	*****
	=== more selection user commands ===
<Ctrl>a	*****
<Ctrl>c	*****
	=== various commands ===
c	clone the current tab into a new tab
p (lower case)	go to the URI which is in the primary selection
P (upper case)	go to the URI which is in the clipboard
'p	open a new window or tab for the URI which is in the primary selection
<Shift><Insert>	press these keys while typing a user command. The content of the primary selection will be inserted at this point in the command you're typing.
	=== bookmarks and browser history ===
<Ctrl>msome tag<Enter>	bookmark the current URI with the tag you typed
M	bookmark the current URI with a tag you will type into a dialog box
U	go to some URI from your browsing history
u	go to some URI from your bookmarks
<Ctrl>d	temporarily bookmark the current URI
D	go to some URI from your temporary bookmarks
	=== link following ===
fl23 (lower case ell)	in the current tab, click on the item 23 shown in the window. The effect depends on what that item is. If it's a text field, that field will be highlighted. If it's a link, treat this as the "fL" command shown below. If it's a button, click it. (If you change your mind before you finish typing this command, press the <Esc> key.)
Fl23 (lower case ell)	same as in the "fL" command, shown next, but in a new tab
fL23	in the current tab, go to link 23 shown in the window. When you type "fL", a number will be displayed in a small circle to the left of each link. Enter the number corresponding to the link you want. The current tab will go to that link. If there are leading zeroes in the link number, be sure to type them. If you change your mind in midstream, press the <Esc> key.
FL23	same as the "fL" command, shown previously, but instead of following the link, just copy the URI to the primary selection
fi	*****
fs	*****
fS	*****
Fs	*****
FS	*****
ft	*****
fT	*****
Ft	*****
FT	*****
	=== filling in forms automatically (e.g., logging in) ===
ze	formfiller edit (I have not tested this)
zn	formfiller new (I have not tested this)
zl	formfiller load (I have not tested this)
zo	formfiller once (I have not tested this)
	=== tab opening ===
gn	"go new": open a new tab after all the others, loading the URI indicated in the config file's line beginning with "set uri ".
gN	open a new tab immediately to the right of the current one, loading the URI indicated in the config file's line beginning with "set uri ".
goxxx<Enter>	open a new tab after all the others and load URI xxx (like http://www.nps.gov/yose)
gOxxx<Enter>	open a new tab immediately to the right of the current one and load URI xxx (like http://www.nps.gov/yose)
	=== tab closing and resetting ===
gC	eliminate the current tab. If that was the final tab, exit uzbl-tabbed. Identical to ZZ.
gQ	eliminate all tabs, and load the default tab. Quite similar to getting completely out of uzbl-tabbed and then running it again.
	=== tab navigating ===
g<	switch to the first tab
g>	switch to the final tab
gt	switch to the next tab
gT	switch to the previous tab
gi3<Enter>	switch to tab 3
<Ctrl><Left>	move current tab to the left
<Ctrl><Right>	move current tab to the right

gm3<Enter>	move current tab to position 3
	=== tab presets ===
gsxxx<Enter>	save the current set of tabs, naming the set xxx
gloxxx<Enter>	load the set of tabs whose name is xxx, without removing any of the tabs you currently have open
gdxxx<Enter>	delete the set of tabs whose name is xxx
	=== manage Secure Sockets Layer ===
!ssl	toggle the ssl_verify value; if on, don't connect to an https: URI without first validating a certificate presented by the remote server against your local certificate authority file. In my case, that file is at /etc/ssl/certs/ca-certificates.crt. The default for this switch is "on".

go to top go to list of contents

key binding collisions

go to top go to list of contents

At some point, you may be inspired to add a new keyboard command; that is, a new binding between what the user types in Command mode and what happens. If so, please keep in mind that a command typed by a user takes effect usually without him having to press <Enter> afterwards. This will be a problem if you want to introduce say, a new user command "bar". When the user wants to type that user command, as soon as he types the "b", the "b" comand will take effect. He'll never get as far as the "ar". Arrrgh.

The easiest way to avoid this is to look at the @cbind statements in the config file in alphabetical order. Then you'll be able to see easily whether your proposed user command conflicts with a new shorter (or longer) user command. To do so, enter the following at the shell prompt:

```
grep '^ *@cbind' ~/.config/uzbl/config | sed -e 's/^ *//' -e 's/^@cbind */@cbind /' | sort | less
```

If you add new bindings, be careful the next time you update to a later version of uzbl. New key bindings may have been added to the example config file. Check that these new bindings do not conflict with your own.

go to top go to list of contents

adding a new user command: leaving uzbl-tabbed

go to top go to list of contents

We now swing into a part of the tutorial which requires a little more geekiness. You can leave the tutorial right now and find uzbl quite usable. Or you can continue and get more out of uzbl's flexibility. There be no dragons here.

We'll start with something relatively easy. We mentioned earlier that we'd add a user command that would get the user out of uzbl-tabbed completely, even if more than one tab was open. We'll do that now.

First we need to see whether uzbl-tabbed will support this new user command. Enter this command at the shell prompt:

```
  grep exit_all_tabs $(which uzbl-tabbed)
```

If the result is absolutely nothing except the next shell prompt, then your version of uzbl is older, and there is no super-simple way to add a full exit user command. (You can still do it, but it's left as an exercise for you.) But if the result is a line of code, you can continue.

Edit your config file. Look for the line that contains the two characters "gQ". It will look something like this:

```
@cbind  gQ            = event CLEAN_TABS
```

After that line, put a line that says this:

```
@cbind  gz            = event EXIT_ALL_TABS
```

Then write out the config file. Run uzbl-tabbed again, and the "gz" user command will exit uzbl-tabbed when entered once, no matter how many tabs are open.

go to top go to list of contents

internal uzbl commands

go to top go to list of contents

uzbl receives internal uzbl commands from various sources. Internal uzbl commands are not to be confused with user commands, as described in the previous section. But you can enter an internal uzbl command as a user command when you're in Command mode by typing a colon (":") followed by the internal uzbl command, followed by the <Enter> key.

The documentation on internal commands is quite detailed in the README file that should have come with your distribution of uzbl, or on the uzbl.org site itself; search for COMMAND SYNTAX.

go to top go to list of contents

more assigned reading

go to top go to list of contents

While you're looking at the README file that should have come with your distribution of uzbl, or on the uzbl.org site itself, browse through the whole thing. You'll find documentation on variables, events, and more. Don't be disheartened by all the detail; as we go through several examples of enhancing uzbl, you'll gradually become familiar with the details.

go to top go to list of contents

playing with variables

go to top go to list of contents

Among the many, many variables listed in the README file that should have come with your distribution of uzbl, or on the uzbl.org site itself, is enable_scripts. As of 2014 Jul 7, it was documented as "disable embedded scripting languages", with 0 as the default value. A more accurate description would be "enable embedded scripting languages", and its default is 1. (It's as though it had been originally named "disable_scripts" and gotten renamed and re-implemented, without a through rewriting of this line in the document.)

When your browser loaded this tutorial, it had JavaScript enabled. If you're reading the tutorial with uzbl, you can disable JavaScript with one of these two user commands:

```
  :toggle enable_scripts
  :set enable_scripts 0
```

Try one of those user commands now, and then reload the tutorial (positioning yourself at this section of it) by clicking here and then pressing the R key (upper case) to reload the page disregarding its cached copy. Then look at how the text has changed. Go ahead, I'll wait.

Like what you see? Then consider adding this user command to your config file (watching out for user command collisions, as usual):

```
  @cbind Ajs = toggle enable_scripts
```

Note, though, that there is a copy of this variable for each tab. If you set enable_scripts to 0, and then open a new tab, that tab will see enable_scripts as 1, the default value. If you want this variable to be set to 0 for each tab when you first open it, then simply place this line in your config file:

```
  set enable_scripts 0
```

By the way, you can also add the current state of the enable_scripts switch to the status bar. In your config file, look for the first occurrence of a line containing

```
status_format
```

Before that line, insert a line like this:

```
set script_section    scripts:<span foreground="#FFFF00">\@enable_scripts</span>
```

... and then change the line that might look like this:

```
set status_format_right <span font_family="monospace"><span foreground="#666">uri:</span> @uri_section</span>
```

... to a line that looks something like this:

```
set status_format_right <span font_family="monospace"><span foreground="#666">@script_section uri:</span> @uri_section</span>
```

You can set variable enable_scripts to 0 in one tab and 1 in another tab, and then flit from tab to tab, watching the value displayed in the status bar change.

Another interesting variable you may wish to display on the status line is is enable_plugins. When its value is 0, then for that tab, no plugins will work. This includes Java and Adobe® Flash®.

A quite different variable is request_handler. For this one, you need to have have experience coding simple scripts, in just about any commonly used language. A shell script is fine, or Python or Perl or something else. Straight out of the box, your config file should contain something like one of these two lines:

```
#set request_handler     sync_spawn @scripts_dir/request.py
#set request_handler       spawn_sync @scripts_dir/request.py
```

You can remove the "#" to change this line in your config file from a comment to a real command. Then, instead of "@scripts_dir/request.py", you can place the full /path/to/your/request/handler. (You won't want to use "@scripts_dir" if you're not root, because it's almost certain that you won't be able to change anything in that directory.)

The script that you write to act as your request handler should take a single command line argument. That argument is a URI that uzbl wants to access. If that URI is fine with you, then your script can simply exit after examining that argument, without writing anything to standard output. But if you wish to redirect to a different URI, have your script write that new URI to standard output before exiting.

go to top go to list of contents

tightening up the status bar

go to top go to list of contents

The status bar contains quite a few items. In my view, the two most important ones are the URI of the current page, and the URI of the link over which the mouse is hovering. There often isn't enough room to show both of these completely. So it would help to shorten everything else. Along the way, let's increase the contrast of the low-contrast items. But if you're satisfied with the status bar as it is, you should skip this section.

Let's start at the left end and shorten the mode indicator. In the config file, look for the only occurrence of "Cmd" and change it to "C". Similarly on the other two mode_indicator lines in the file, change "Ins" to "I" and "Bnd" to "B".

Then let's remove the square brackets which surround the mode indicator. On the line which defines the mode_section, remove the first left square bracket and the final right square bracket.

Let's say you've typed the "o" or "go" command, and you're being prompted for the new URI. That prompt is almost unreadable. Change its color to white. Locate the definition of prompt_style that's in the @stack statement, not in the set statement. Change the foreground color to white with

```
foreground="white"
```

While we're at it, change the progress section definition so the foreground is white. Change the foreground of other items in that paragraph of the configuration file to suit.

If you think that the progress display is too spacious and should simply show the percent complete, change the "set progress.format" definition to a simple %c.

One item in the status bar is the process ID of uzbl-tabbed, followed by a hyphen, followed by a unique numerical identifier. The end user almost never needs this info. Find the "set status_format" line in the config file, and remove the @name_section part, leaving only a single space there (not two spaces). While you're at it, if you're so inclined, similarly remove the @scroll_section, which shows in percentage terms how far you've scrolled through the current page. The scroll bar shold suffice for most users.

To make the URI of the link over which the mouse is hovering more readable, locate the "set selected_section" statement and change the foreground to white.

You already know that the URI in green at the right end of the status bar is a URI; you don't need it to be labeled. So in the line that sets status_format_right, remove the "uri:" and the space that precedes it.

go to top go to list of contents

logging events to disk

go to top go to list of contents

You may recall from here that you can pop up a window which shows all events as they happen. That's quite a few events; even moving the mouse around in the uzbl window while that window is active generates a zillion events. To find the events you're interested in, it would be nice to be able to log all events to disk.

To do so, find the line in the config file which defines the <Ctrl><Mod1>t user command. (That shouldn't be too difficult, because there are not many user commands which use Mod1. Keep in mind that this ends with the number "1", not the lower case "l". And remember that on most keyboards, Mod1 means Alt.) Then in that neighborhood of the config file, add one of the following, depending on whether the line you found contains "sh" or "spawn_sh":

```
@cbind  <Ctrl><Mod1>s  = sh 'xterm -e script -c \"socat unix-connect:\\\"$UZBL_SOCKET\\\" -\" -f ~/.local/share/uzbl/typescript.`date \"+%Y%m%d%H%M%S\"`'
@cbind  <Ctrl><Mod1>s  = spawn_sh 'xterm -e script -c \"socat unix-connect:\\\"$UZBL_SOCKET\\\" -\" -f ~/.local/share/uzbl/typescript.`date \"+%Y%m%d%H%M%S\"`'
```

You'll probably find it easier to make a copy of the line you searched for, and modify it. Don't forget to change the first t to s.

Then when you type <Ctrl><Mod1>s the window will pop up as before, but you'll also get a log file in directory ~/.local/share/uzbl, whose name includes the time of day that the file was created.

go to top go to list of contents

fixing two minor bugs in uzbl-tabbed

go to top go to list of contents

You might have a version of uzbl-tabbed which contains a typo, and this typo will need to be fixed before continuing with this tutorial. You already modified uzbl-tabbed here, so take a look at the copy of uzbl-tabbed that you modified. Enter this

command at the shell prompt:

```
  grep new_bg_tab $(which uzbl-tabbed)
```

If that command causes a line of code to be output, you're fine. Otherwise, you'll need to change your copy of uzbl-tabbed. Look for a line that contains the word "new_tab_bg", and change that word to "new_bg_tab". Be careful not to change the number of spaces at the beginning of the line, and don't accidentally remove the colon at the end of the line.

And one more bug, if you please. I have occasionally noticed that when I first run uzbl-tabbed, it immediately crashes with a message like this:

```
Traceback (most recent call last):
  File "/usr/bin/uzbl-tabbed", line 367, in _socket_closed
    self.uzbl.close()
AttributeError: 'NoneType' object has no attribute 'close'
```

That doesn't mean that there's a bug at line 367. There might be a bug there, but the bug might be elsewhere. At any rate, removing the symptom seems harmless. To so, find this code in uzbl-tabbed:

```
    def _socket_closed(self, fd, condition):
        '''Remote client exited'''
        self.uzbl.close()
        return False
```

Change it so it looks like this:

```
    def _socket_closed(self, fd, condition):
        '''Remote client exited'''
        if self.uzbl:
            self.uzbl.close()
        return False
```

go to top go to list of contents

tab control

go to top go to list of contents

A uzbl-tabbed user might want to follow a link on a page by opening into a new tab, but might want to either: (a) open the new tab just to the right of the current tab, and go to it immediately; (b) the same, but stay in the current tab; (c) open the new tab at the end of all tabs, and go to it immediately; or (d) the same, but stay in the current tab.

You could configure uzbl-tabbed to do one of these four things, but suppose you want more than one of these choices available in the currently running uzbl-tabbed session. What if you want all of them available in a menu? We'll do that here.

In your config file, look for the line that contains these words:

```
Print Link
```

If the line looks like this:

```
menu_link_add Print Link = print \@SELECTED_URI
```

then you are working with the older menu syntax. After that line, add these five lines:

```
menu_link_separator sep_xx
menu_link_add Open Link to Right     = event NEW_TAB_NEXT    \@SELECTED_URI
menu_link_add Open Link at End       = event NEW_TAB         \@SELECTED_URI
menu_link_add Open Link to Right, bg = event NEW_BG_TAB_NEXT \@SELECTED_URI
menu_link_add Open Link at End, bg   = event NEW_BG_TAB      \@SELECTED_URI
```

If the line looks like this:

```
menu add link "Print Link" "print \@SELECTED_URI"
```

then you are working with the newer menu syntax. After that line, add these five lines:

```
menu add_separator link sep_xx
menu add link "Open Link to Right"     "event NEW_TAB_NEXT    \@SELECTED_URI"
menu add link "Open Link at End"       "event NEW_TAB         \@SELECTED_URI"
menu add link "Open Link to Right, bg" "event NEW_BG_TAB_NEXT \@SELECTED_URI"
menu add link "Open Link at End, bg"   "event NEW_BG_TAB      \@SELECTED_URI"
```

Then take uzbl-tabbed out for a spin. Right click on a link. See those familiar-looking new items at the bottom of the menu? Those are yours! The "bg" means "background"; that is, when opening the new tab, don't switch to it, but stay in the current tab.

go to top go to list of contents

new window, for real

go to top go to list of contents

Of course, using the menu as we've modified it above is not the only way we create new tabs. There are the "c" and "go" commands, for example. There are also times when JavaScript running in the current tab will attempt to create a new window, and end up creating a new tab (at least using the configuration as we've modified it). In any of these cases, where does this new tab go, and do we switch to it immediately? The content of the "@on_event NEW_WINDOW" line in the config file shows where the new tab goes, and whether it's in background mode. Incidentally, the "Open Link in New Window" menu item when you right click on a link also heeds this line in the config file.

But what if we want actually to create a new window? The easiest way is to clone a new instance of uzbl-tabbed. We'll do that here. We could change the behavior of the "Open Link in New Window" item in the right-click menu, but that's more involved than it looks. To keep things simple, let's just add a new menu item to the end. In your config file:

if you're using the old syntax, add this line, just after the final menu_link_add line you added:

```
menu_link_add New Window For Real = event CRY_WOLF \@SELECTED_URI
```

... but if you're using the new syntax, add this line, just after the final "menu add link" line you added:

```
menu add link "New Window For Real" "event CRY_WOLF \@SELECTED_URI"
```

We've invented a new event. Let's define its behavior. In your config file, just after the "@on_event NEW_WINDOW" line,

if you're using the old syntax, add this line:

```
@on_event CRY_WOLF sh 'uzbl-tabbed ${1:+ "$1"}' %r
```

... but if you're using the new syntax, add this line:

```
@on_event CRY_WOLF spawn_sh 'uzbl-tabbed ${0:+-u "$0"}' %r
```

Now try it out.

go to top go to list of contents

background tabs, left to right

go to top go to list of contents

Now we move into a place where you'll be doing some Python coding. There be no dragons here, but there are a few dragon eggs, waiting to hatch if you're not careful. Remember the following two items.

First, indentation counts. To avoid headaches later, don't use tabs; just use spaces. And how far a line is indented matters. Look around through uzbl-tabbed and you'll get an idea how it works. Note that if a line ends in a colon (":"), then the next line (or several lines) will be indented further. Lines that end in a colon announce a new block of code: what is to be executed if a condition is true, or what is to be executed as part of the function named on the line with the colon, and so on.

Second, punctuation counts. If you're going to insert a new line of code that ends in a colon (see previous paragraph), see that the colon isn't missing!

Ready to roll up your sleeves? Good!

Let's say you have five tabs open, and you're positioned at the third one. We can represent this situation thus:

  AAA BBB CCC DDD EEE

Ok, you're on page CCC. Let's say that on this page you're interested in three links: XXX, YYY, and ZZZ. You follow each of those links, in that order, using the "Open Link to Right, bg" item of the right-click menu. After the first click, you have:

  AAA BBB CCC XXX DDD EEE

After the second click, you have:

  AAA BBB CCC YYY XXX DDD EEE

After the third click, you have:

  AAA BBB CCC ZZZ YYY XXX DDD EEE

Well, that's fine, and the "Open Link to Right, bg" item is working as advertised. But let's say that what you really wanted was to have those links from left to right, thus:

  AAA BBB CCC XXX YYY ZZZ DDD EEE

That's the way some browsers actually work, and it's arguably more, um, usable for the ordinary user. Let's add another menu item to make it work that way.

The first step is to add a new menu item to the link right-click menu; the best place for it seems to be just before the New Window For Real item we just added. So before that item in your config file:

if you're using the old syntax, add this line:

```
menu_link_add Open Link LTOR, bg = event NEW_BG_TAB_LTOR \@SELECTED_URI
```

... but if you're using the new syntax, add this line:

```
menu add link "Open Link LTOR, bg" "event NEW_BG_TAB_LTOR \@SELECTED_URI"
```

The remaining part of this change is far more complex: changing uzbl-tabbed. Before you begin, make a copy of uzbl-tabbed and set it aside. If you're human, there's a chance you'll mess up the code, and you'll want to be able to start over.

The following changes to uzbl-tabbed are simple, and I made them that way to make the tutorial easier to understand. But from the viewpoint of long-term maintainability of the code, the changes are messy. There are better ways to do what we're about to do here. When you get more familiar with the internals of uzbl, go and implement one of those ways.

The changes center around a global variable we will be inventing which we'll call ltor_tab. At any given time, this variable's value will be either None, or some integer. If the value is None, then when you select the menu item "Open Link LTOR, bg", the new tab will be created immediately to the right of the current one. But if the value is an integer, then the new tab will be created just to the right of the tab numbered with that value: if ltor_tabbed is 3, for example, then the new tab will be numbered 4.

So ltor_tab contains the tab number of the tab most recently created with the menu item "Open Link LTOR, bg", provided that this tab creation was done recently. What does "recently" mean? The definition (and implementation) of "recently" is the most involved part of the changes we're about to make to uzbl-tabbed.

The first step is to define the global variable ltor_tab. Do so just before the "Default configuration section", with the following line, making sure you do not indent it:

```
ltor_tab = None
```

The second step is to change the definition of method create_tab. Search for "def create_tab". The content of that method seems fairly stable; it seems unlikely that it will be changed much in future versions of uzbl-tabbed. In my version, it looks like this, and in yours it's probably the same:

```
    def create_tab(self, beside = False):
        tab = gtk.Socket()
        tab.show()

        if beside:
            pos = self.notebook.get_current_page() + 1
            self.notebook.insert_page(tab, position=pos)
        else:
            self.notebook.append_page(tab)

        self.notebook.set_tab_reorderable(tab, True)
        return tab
```

Now change it so it looks like this:

```
    def create_tab(self, beside = False):

        '''beside is False, True, None (which implies True), or tab number '''

        global ltor_tab

        tab = gtk.Socket()
        tab.show()

        if beside == None:
            beside=True;

        if beside == True:
            beside=self.notebook.get_current_page() + 1

        if beside:
            ltor_tab = beside + 1
            self.notebook.insert_page(tab, position=beside)
        else:
            self.notebook.append_page(tab)

        self.notebook.set_tab_reorderable(tab, True)
        return tab
```

The third step is to add a new method which handles the event NEW_BG_TAB_LTOR. First, search for method new_bg_tab_next by searching for "def new_bg_tab_next". That method is short and sweet, and should look like this:

```
    def new_bg_tab_next(self, uri = ''):
        self.uzbl_tabbed.new_tab(uri, switch = False, next = True)
```

After that method, add this new one:

```
    def new_bg_tab_ltor(self, uri = ''):
        global ltor_tab
        self.uzbl_tabbed.new_tab(uri, switch = False, next = ltor_tab)
```

Recall that we reset ltor_tab to None when we haven't done an LTOR tab creation "recently". Now comes the tricky part: defining "recently". "Recently" means "since the most recent occurrence of any of these happenings". What happenings are those? Aye, there's the rub.

One occasion to reset ltor_tab is if we move from one tab to another. To do this, change method goto_tab. Don't change method goto_tab in class GlobalEventDispatcher:

```
    def goto_tab(self, index):
        self.uzbl_tabbed.goto_tab(int(index))
```

Leave that one alone. Instead, change method goto_tab in class UzblTabbed:

```
    def goto_tab(self, index):
        '''Goto tab n (supports negative indexing).'''

        global ltor_tab

        ltor_tab = None

        title_format = "%s - Uzbl Browser"

        tabs = list(self.notebook)
        [ ... and so on ... ]
```

Another occasion to reset ltor_tab is if we close any tab. To be safe (see how fragile this change is?), change three methods. First, change method exit:

```
    def exit(self):
        ''' Ask the Uzbl instance to close '''

        global ltor_tab
        ltor_tab = None

        if self._client:
            self._client.send('exit')
```

Then, change method close_tab:

```
    def close_tab(self, tabn=None):
        '''Closes current tab. Supports negative indexing.'''

        global ltor_tab
        ltor_tab = None

        if tabn is None:
            tabn = self.notebook.get_current_page()
        [ ... and so on ... ]
```

Then, change method tab_closed:

```
    def tab_closed(self, notebook, tab, index):
        '''Close the window if no tabs are left. Called by page-removed
        signal.'''

        global ltor_tab
        ltor_tab = None

        if tab in self.tabs.keys():
        [ ... and so on ... ]
```

Yet another occasion to reset ltor_tab is if we open a tab, not LTOR, without going to it. To do so, change this method:

```
    def new_bg_tab(self, uri = ''):
        global ltor_tab
        self.uzbl_tabbed.new_tab(uri, switch = False)
        ltor_tab = None
```

... and also this method:

```
    def new_bg_tab_next(self, uri = ''):
        global ltor_tab
        self.uzbl_tabbed.new_tab(uri, switch = False, next = True)
        ltor_tab = None
```

And that's it! Probably. There's a possibility that I missed a spot where ltor_tab should be reset. If you're inclined, now would be a good time to browse through uzbl-tabbed for yourself to see.

go to top go to list of contents

URI-dependent configuration

go to top go to list of contents

It would be nice to have more than one configuration, depending on the URI you're visiting. This has been actually attempted. It's instructive to look at the attempt described here (unless the fine folks at uzbl.org have reorganized their web pages so that this link no longer exists).

That page describes a contributed Python script which turns on JavaScript for certain URIs, and turns it off for all others. It sounds good; evidently this script works better than another solution, because that other solution doesn't run fast enough: the page is often completely loaded before the configuration change can take effect. But even the solution shown in this link only works (by the author's estimate) on 90% of pages, because even with this script some pages load too fast. This is just wrong. A web browser ought not have race coditions like this. This section of this tutorial seeks to eliminate the race condition completely by loading a custom configuration before informing uzbl of the desired URI.

There are two tradeoffs here. First, this feature will work only when new windows and tabs are created. If you just click on a link and go to a different URI in the current tab, this change will not affect your configuration in that tab.

The second tradeoff is that like the change we made in the previous section of this tutorial, this one involves changing not just a config file, but uzbl-tabbed itself. But, as with the previous change, you're dealing with open source here. There's nothing wrong with getting your fingernails dirty. This feature does require that you be familiar with regular expressions. If you are, but only (say) in Perl, that's fine; you'll catch on quickly.

The first step is to figure out what your usual configuration should be; by now you've probably done that. The second step is to figure out some alternate configuration files, and which URIs each one should affect. For example, you might decide that certain URIs should be loaded with scripting (for example, JavaScript) disabled.

In each of these alternate config files, it's a good idea to comment out any line that has a "uri" or "set uri" command. The change you're about to make will find the right alternative config file, create a temporary copy of that config file, and place the relevant URI in a uri statement at the end of that temporary copy.

The third step is to create a file and fill it with pairs of lines. The first line in each pair is a regular expression. The second line in each pair is the full pathname of the config file which is to be used if the URI of the new tab or window matches the regular expression in the first line. Here's what such a file might look like.

```
(https?\:\/\/)?(www\.)?fark\.com([\/\?].*)?$
/config/path/1
(https?\:\/\/)?(www\.)?google\.com([\/\?].*)?$
/config/path/2
```

```
(https?\:\/\/)?(www\.)?youtube\.com([\/\?].*)?$
/config/path/3
```

Where do you place this file? Just about anywhere, but our crude implementation of this idea puts this file at ~/.config/config_list.txt.

And now it's time to let you in on a dirty little secret. Some readers of this tutorial are dealing with both an old-syntax config file (for use with an older, root-installed uzbl) and a new-syntax config file (for a newer uzbl which they are installing as non-root). They may wish to switch between the two uzbl releases. They should have two files: ~/.config/config_list.txt, and ~/.config1/config_list.txt. The even-numbered lines in the first of those files should point to config files that are in the old format; the even-numbered lines in the second of those files should point to config files that are in the new format.

Ok, let's take a breather here and look at the first of those regular expressions. It accepts any string that contains "fark.com", optionally preceded by "www.", optionally preceded by "http://" or "https://". After the "fark.com" there can be any other characters, as long as those characters begin with "/" or "?".

Now, continuing with our project, the fourth (and final) step is to make changes to uzbl-tabbed. First, salt away a good copy of uzbl-tabbed; it's easy while editing to mess things up, and you want to have something to go back to.

Near the beginning of uzbl-tabbed, there are several "import" statements. Add this one somewhere:

```
import tempfile
```

Then look for a paragraph of code that looks like this:

```
# Ensure uzbl xdg paths exist
if not os.path.exists(DATA_DIR):
    os.makedirs(DATA_DIR)
```

After that code, insert this somewhat lengthy chunk of code. You'll notice that it sets environment variables XDG_CONFIG_HOME. You'll also notice that the setting of this environment variable is done in more than one part of this tutorial. It's done in more than one place so you can pick and choose which of this tutorial's suggested changes you want to make. All these settings of XDG_CONFIG_HOME are identical, except the one you make if you choose to reinstall uzbl as non-root; in that case, that different setting of XDG_CONFIG_HOME will actually override all the others. Anyway, here's the code to insert:

```
# This is the config list file location.

os.environ.setdefault("XDG_CONFIG_HOME",os.path.join(os.environ["HOME"],".config"))
CONFIG_LIST_FILE = os.path.join(os.environ['XDG_CONFIG_HOME'],'config_list.txt')

config_list = [];
line_number = 0;
first_line  = None;

try:
    for in_line in open(CONFIG_LIST_FILE):
        line_number += 1

        if first_line:
            config_list += [[ first_line, in_line.rstrip("\n") ]]
            first_line = None
        else:
            try:
                first_line = re.compile(in_line.rstrip("\n"))
            except:
                sys.stderr.write("file "
                                 +CONFIG_LIST_FILE
                                 +", line "
                                 +str(line_number)
                                 +" contains an invalid regular expression\n"
                                 )
                exit(1)
except IOError:
    sys.stderr.write("["
                     +CONFIG_LIST_FILE
                     +" not found; no dynamic configuration will be done]\n"
                     )

if first_line:
    sys.stderr.write("file "
                     +CONFIG_LIST_FILE
                     +" contains an odd number of lines\n"
                     );
    exit(1)

def config_search(uri):
  try:
    return [x[1] for x in config_list if x[0].match(uri)][0]
  except:
    return None
```

Now look for the "class UzblInstance" definition. Within that, the first method is called __init__, and the second is called set_tab. Place the following code at the end of method __init__, being careful to line up the left edge of the code with the left edge of the preceding statements.

```
        override_config = config_search(uri);

        self.named_config_file = None
        if override_config:
            with tempfile.NamedTemporaryFile("w+t",delete=False) as f:
                self.named_config_file = f.name
                with open(override_config,"rt") as g:
                    f.write(g.read())
                    f.write("uri ")
                    f.write(uri)
                    f.write("\n")
```

The following two changes will cause the temporary config files to be deleted after they're no longer needed.

Look for "def exit", which begins the definition of the "exit" method of class UzblInstance. Change it so it ends like this; the red lines are the ones you'll be adding.

```
        if self.named_config_file:
            try:
                os.remove(self.named_config_file)
            except:
                pass

        if self._client:
            self._client.send('exit')
```

Right after that is class UzblInstance's "close" method. (It is important not to confuse this with class SocketClient's "close" method, which looks similar.) Change it so it looks like this. The red lines are the ones you'll be adding.

```
    def close(self):
        '''The remote instance exited'''

        if self.named_config_file:
            try:
                os.remove(self.named_config_file)
            except:
```

```
                pass

        if self._client:
            self._client.close()
            self._client = None
```

The final changes are in method new_tab: not the method new_tab in class GlobalEventDispatcher, but the method new_tab in class UzblTabbed. (The method we're interested already contains a couple of dozen lines; the other one is a two-liner.)

First, delete the paragraph that looks approximately like this:

```
        if(uri):
          cmd = cmd + ['--uri', str(uri)]
```

Then, if you see a paragraph that looks like this, delete it. (Older verions of uzbl-tabbed will not have this paragraph.)

```
        if config['explicit_config_file'] is not None:
            cmd += ['-c', config['explicit_config_file']]
```

Where you just made your code deletion, add these lines:

```
        if uzbl.named_config_file:
            cmd = cmd + ['-c',uzbl.named_config_file]
        elif uri:
            cmd = cmd + ['--uri', str(uri)]
```

Finally, consider two sets of lines in this method. We'll call one set Fred, and the other set Barney. Here's Fred:

```
        uzbl = UzblInstance(self, name, uri, title, switch)
        uzbl.set_tab(tab)
```

... and here's Barney:

```
        cmd = ['uzbl-browser', '-n', name, '-s', str(sid),
               '--connect-socket', self.socket_path]
```

So what you do is take the code named Fred, and move (not copy) it up a bit in the source file, to just before Barney. That's it. You're done.

go to top go to list of contents

cellular browsing

go to top go to list of contents

Private browsing mode is useful if you don't want your computer to keep long-term track of where you've been on the web. In uzbl, private browsing mode is implemented in WebKit.

There are limitations to private browsing. First and possibly foremost, server-side tracking of your activities and tracking by your Internet service provider are not limited by private browsing. Second, users of private browsing are relying on the kindness of any browser extensions they may be using. Third, there may be bugs in the way your browsing engine (such as WebKit) implements private browsing.

If you want to enable private browsing, you can set uzbl's "enable_private" variable to 1. But another approach is to use what I call cellular browsing. Here's the general idea:

Suppose each time you opened a new tab or window, you actually ran uzbl with the HOME environment variable defined to point to a fresh, new directory, with certain data (like the .config subdirectory) copied into it. I refer to this as "cellular browsing". It's good for starting with a clean cookie sheet and browsing history, and no cruft from using anything from Adobe.

Let's review the limitations we discussed a couple of paragraphs above. Server-side and ISP tracking of your activities won't be reduced in the slightest with cellular browsing. But browser extensions won't be able to get away with simply ignoring your preference for private browsing; they'll have to be really malicious and sly to get around cellular browsing. And cellular browsing should transcend any bugs your browsing engine has in its implementation of private browsing.

Before continuing, note that this tutorial's implementation of this idea relies on your having already made some of the changes suggested in previous sections of this tutorial. If you grep for the string "uzbl-browser" (without the quotation marks) in your config file, and the only match is one which is commented out (that is, it has "#" at the beginning of the line), you'll be fine; otherwise, you haven't made the changes required.

Now, here's what happens with cellular browsing:

Every time you run uzbl-tabbed, either you use cellular browsing throughout that run, or you do not use cellular browsing at all.

You use cellular browsing if two conditions are met: (a) you've created a script which initializes a new home directory the way you want; and (b) you haven't disabled cellular browsing for this run of uzbl-tabbed.

So the first step is to create a script which initializes a new home directory, given the old home directory specified in environment variable ORIG_HOME and the new home directory specified in environment variable NEW_HOME. Here's a sample:

```
#!/bin/sh

set -e
umask 077
mkdir -p $NEW_HOME
cp -prd $ORIG_HOME/.config $NEW_HOME
if [ -e $ORIG_HOME/.config1 ]
then
  cp -prd $ORIG_HOME/.config1 $NEW_HOME
fi
mkdir -p $NEW_HOME/.local/share/uzbl
touch    $NEW_HOME/.local/share/uzbl/cookies.txt
touch    $NEW_HOME/.local/share/uzbl/session-cookies.txt
```

"Wait, you're thinking. "What's that config1 stuff doing in there?" As explained in the previous section, some readers of this tutorial are dealing with an "official" uzbl for their system, which uses an old-syntax config file, and a newer uzbl, which they are installing as non-root and which uses a new-syntax config file. That config1 stuff is for them. If this bothers you, take out the four lines prtaining to config1. But it won't hurt to leave them there.

And those final three lines suppress a silly error message that occurs in some versions of uzbl under some circumstances.

So make that script executable using the chmod command, and place it in your HOME directory, at ~/.config/uzbl-cell-init. And also at ~/.config1/uzbl-cell-init, if you're a dual-uzbl-config-syntax person.

Once you've created this script, when you run uzbl-tabbed with your current value of the HOME environment variable (which most users never change), you will be doing cellular browsing.

But the occasion may arise when you don't want to do cellular browsing. For example, if you go to a site and want to log in, and that particular site handles this by popping up a new window (or tab) to handle the logging in, then with cellular browsing the new tab would have nothing to do with your original tab; any cookies would be kept entirely separate, and the logging in will be ineffective. So how do you turn off cellular browsing?

If you want to turn off cellular browsing permanently for future runs of uzbl-tabbed, rename the script you created above to something else. If you wish to temporarily run uzbl-tabbed without cellular browsing, you don't need to rename this script; simply define the environment variable UZBL_CELL_DISABLE before running uzbl-tabbed. In most shells, enter this command at the shell prompt:

```
export UZBL_CELL_DISABLE 1
```

Spelling is important; if you misspell "UZBL_CELL_DISABLE", then the "export" command will not do what you want, you will get no warning, and you will be doing cellular browsing.

After entering that command, run uzbl-tabbed. When you exit from uzbl-tabbed, be sure to enter this command at the shell prompt to re-enable cellular browsing:

```
unset UZBL_CELL_DISABLE
```

If you neglect to do this, then the next time you run uzbl-tab, you will probably run without using cellular browsing, but mistakenly think that you are indeed using cellular browsing. This obviously could be a security issue for you. So each time you run uzbl-tabbed with a good "uzbl-cell-init" file, if uzbl-tabbed notices that cellular browsing has been disabled, it will ask you if that is your intention.

Asking this question actually serves another purpose, too. If you attempt to disable cellular browsing, but mis-spell "UZBL_CELL_DISABLE", and do not see this question when you run uzbl-tabbed, you know that you're still using cellular browsing.

When you leave a uzbl-tabbed session with cellular browsing, the temporary HOME directory is not actually deleted from disk; you may wish to inspect it later. But this can easily lead to proliferation of unwanted duff. So when you run uzbl-tabbed with cellular browsing, it first checks for HOME directories created by uzbl-tabbed which are no longer in use. If it finds any, it asks whether you want to delete them all. After you answer this question, uzbl-tabbed will then proceed to come up as usual.

Where are these temporary home directories? Each is formed like this example:

```
/tmp/uzbl-cell-fred/20140804.075222.2345/000007
```

As you see, they're all stored in the /tmp directory. Usually in FreeBSD and in most Linux distributions, the /tmp directory is cleared when you reboot. You can change the value of the cell_base variable in the changes shown below to /var/tmp if you wish to have these old home directories persist across reboots.

The "fred" is whatever name you're logged in as when you run uzbl-tabbed; it's taken from the LOGNAME environment variable.

After the "fred" come the date and time that uzbl-tabbed started, and its process identification (PID). The final six-digit number is the serial number uzbl-tabbed assigned to this particular tab or window.

Ok, it's time to start changing uzbl-tabbed. The first change is the largest by far. To make it a little easier to spot the end of this change as you read it here, I've typed the whole change in red. I've sprinkled seven marks through the change to make it easier to copy and paste the change in smaller chunks. All this code should go just before the "Default configuration section".

```
# mark 1 ---------- >8 ---------- >8 ---------- >8 ---------- >8 ----------

def yesno(question):
  later = False;
  while True:
    if later:
      sys.stdout.write("Please answer yes or no.\n")
    sys.stdout.write(question+" ")
    sys.stdout.flush()
    answer=sys.stdin.readline().rstrip("\n")
    if re.search(r'^yes$',answer,re.IGNORECASE):
      return True
    if re.search(r'^no$',answer,re.IGNORECASE):
      return False
    later = True;

# mark 2 ---------- >8 ---------- >8 ---------- >8 ---------- >8 ----------

cell_base            = "/tmp"
cell_cellbase        = os.path.join(cell_base,
                                    'uzbl-cell-'+os.environ['LOGNAME']
                                   )
cell_cwd             = os.getcwd()
cell_proc            = "/proc"   # change to elsewhat to use test data
os.environ.setdefault("XDG_CONFIG_HOME",os.path.join(os.environ["HOME"],".config"))
cell_script_location = os.path.join(os.environ['XDG_CONFIG_HOME'],'uzbl-cell-init')

# mark 3 ---------- >8 ---------- >8 ---------- >8 ---------- >8 ----------

cell_switch          = False  # {{{----------------------+
                                                        #     |
if os.path.exists(cell_script_location):                #     |
    #                                                   #     |
    try:                                                #     |
        os.environ['UZBL_CELL_DISABLE']                 #     |
    except KeyError:                                    #     |
        cell_switch = True                              #     |
    else:                                               #     |
        cell_switch = False   # redundant, but hey! }}}-+
        if yesno("File "
                 +cell_script_location
                 +" exists,\n"
                 +"but you have set environment variable UZBL_CELL_DISABLE.\n"
                 +"Are you sure you want to disable cellular browsing?  >"
                ):
            pass
        else:
            sys.stdout.write(
                "Ok, so \"unset UZBL_CELL_DISABLE\" and rerun uzbl-tabbed\n")
            exit(0)

# mark 4 ---------- >8 ---------- >8 ---------- >8 ---------- >8 ----------

if cell_switch:
    if not(os.access(cell_script_location,os.X_OK)):
        sys.stdout.write("File "
                         +cell_script_location
                         +" is not executable.\n"
                         +"Use chmod to fix this,"
                         " and then re-run uzbl-tabbed.\n"
                         )
        exit(1)

    cell_time=time.localtime()
    cell_session=os.path.join(cell_cellbase,
                              "%04d%02d%02d.%02d%02d%02d.%d"
                              % ( cell_time.tm_year,
                                  cell_time.tm_mon,
                                  cell_time.tm_mday,
                                  cell_time.tm_hour,
                                  cell_time.tm_min,
                                  cell_time.tm_sec,
                                  os.getpid()
                                )
                             )

    cell_serial=-1

    os.environ['ORIG_HOME']=os.environ['HOME']

# mark 5 ---------- >8 ---------- >8 ---------- >8 ---------- >8 ----------

# We may want to delete all directories in cell_cellbase except those
# associated with currently running instances of uzbl-tabbed.  The name of
# each subdirectory of cell_cellbase includes the date and time that
```

```
# uzbl-tabbed started, and also the process identification (PID).  We use
# the PID part of the subdirectory name to see whether uzbl-tabbed is
# running with the same PID.  But if we might be cleaning up many, many
# subdirectories for many, many prior runs of uzbl-tabbed, we might have more
# than one directory indicating the same PID.  (Consider the birthday
# paradox.)  If there's more than one entry for a given PID, we should
# consider only the most recently created directory for that PID.  To do
# this, we create a dictionary whose keys are PIDs.  We place each
# directory name into that dictionary in chronological order, with the key
# being the PID, so for a given PID, only the most recent entry will
# remain.  When the smoke clears, the dictionary will show which entries
# should not be deleted because uzbl-tabbed is still running.

# mark 6 ---------- >8 ---------- >8 ---------- >8 ---------- >8 ----------

if cell_switch and os.path.exists(cell_cellbase):

    cell_pid_directory      ={}
    cell_deletion_candidates=[]
    cell_dirname_pattern    =re.compile("^\\d{8}\\.\\d{6}\.(\\d+)$")
    cell_uzbl_tabbed_pattern=re.compile("uzbl-tabbed")

    for name in sorted(os.listdir(cell_cellbase)):
        cell_found=cell_dirname_pattern.search(name)
        if cell_found:
            cell_deletion_candidates+=[name];
            cell_pid=cell_found.group(1)
            if os.path.exists(os.path.join(cell_proc,cell_pid)):
                if cell_uzbl_tabbed_pattern.search(
                    os.path.realpath(os.path.join(cell_proc,cell_pid,"exe"))):

                    cell_pid_directory[cell_pid]=name

    cell_deletion_candidates=[x for x in cell_deletion_candidates
        if x not in cell_pid_directory.values()]

        if os.environ.get('UZBL_REMOVE_OLD') == "no":
            pass;
        else:
          if os.environ.get('UZBL_REMOVE_OLD') == "yes" or \
          yesno(cell_q1+cell_q2+cell_q3):
            for name in cell_deletion_candidates:
                subprocess.call(["rm","-rf",os.path.join(cell_cellbase,name)])

    if len(cell_deletion_candidates) > 0:
        if len(cell_deletion_candidates) > 1:
            cell_q1="There are "
            cell_q2=str(len(cell_deletion_candidates))
            cell_q3=" old session directories.  Should I delete them?"
        else:
            cell_q1="There is "
            cell_q2=str(len(cell_deletion_candidates))
            cell_q3=" old session directory.  Should I delete it?"
        if yesno(cell_q1+cell_q2+cell_q3):
            for name in cell_deletion_candidates:
                subprocess.call(["rm","-rf",os.path.join(cell_cellbase,name)])

# mark 7 ---------- >8 ---------- >8 ---------- >8 ---------- >8 ----------
```

The other three changes are in method new_tab: not the method new_tab in class GlobalEventDispatcher, but the method new_tab in class UzblTabbed. (The method we're interested already contains a couple of dozen lines; the other one is a two-liner.)

First, add these three "global" statements near the beginning of method new-tab:

```
        when you need to load multiple tabs at a time (I.e. like when
        restoring a session from a file).'''

        global cell_switch
        global cell_session
        global cell_serial

        tab = self.create_tab(next)
        sid = tab.get_id()
```

Second, add this "if cell_switch:" chunk of code, shown in red, after the call to method uzbl.set_tab():

```
        uzbl = UzblInstance(self, name, uri, title, switch)
        uzbl.set_tab(tab)

        if cell_switch:
            cell_serial+=1
            cell_home=os.path.join(cell_session,
                                   "%06d" % cell_serial
                                  )

            os.environ['HOME']=cell_home
            os.environ['NEW_HOME']=cell_home

            if not(subprocess.call(["mkdir","-p",cell_home])==0):
                sys.stdout.write("failed to create temporary home directory %s"
                                 %
                                 cell_home
                                )
                exit(1)

            os.chdir(cell_home)

            if not(subprocess.call(cell_script_location,shell=True)==0):
                sys.stdout.write(
                  "temporary home directory initialization failed")
                exit(1)

        cmd = ['uzbl-browser', '-n', name, '-s', str(sid),
               '--connect-socket', self.socket_path]
```

Finally, add this "if cell_switch:" chunk of code, shown in red, at the end of method new_tab:

```
        gobject.spawn_async(cmd, flags=gobject.SPAWN_SEARCH_PATH)

        SocketClient.instances_queue[name] = uzbl

        if cell_switch:
          os.environ['HOME']=os.environ['ORIG_HOME']
          os.chdir(cell_cwd)

          os.environ.pop('NEW_HOME',None)

    def clean_slate(self):
        '''Close all open tabs and open a fresh brand new one.'''
```

That's it. You're done.

go to top go to list of contents

getting uzbl source code

go to top go to list of contents

Almost all uzbl users will have no reason to change and recompile the C source code for uzbl-core. But in case you need to, here's how.

The most valuable place to get the C source code, in my opinion, is wherever you got the uzbl package, because you'll want the source code which matches the behavior of your version of uzbl, and it's the easiest to actually change and recompile with minimized chances of descending into dependency hell. You may have to wander around your uzbl package's site of origin a little. I found the source code for the version of uzbl I'm running, for example, at

  https://packages.debian.org/wheezy/web/uzbl

On that page, I looked at the sidebar section entitled "Download Source Package uzbl", and downloaded:

  [uzbl_0.0.0~git.20120514.orig.tar.gz]
  [uzbl_0.0.0~git.20120514-1.1.diff.gz]

This left me with two files:

  uzbl_0.0.0~git.20120514.orig.tar.gz
  uzbl_0.0.0~git.20120514-1.1.diff.gz

I then transformed the "orig"inal code (the raw material for the Debian developers) into the Debianized source code by running the following shell commands. I didn't type them in and execute them one by one; I put them in a shell script in the same directory as the two files listed above, and then executed it.

```
rm -rf work
mkdir work
cd work
cp -prd ../*.gz .
tar -xovzf *.tar.gz
gunzip *.diff.gz
patch < *.diff
less control
```

That final "less" command showed me which other packages I needed to install before I could compile uzbl; in particular, these lines were of interest:

```
Build-Depends: debhelper (>= 7.0.50),
               quilt,
               libgtk2.0-dev (>= 2.14),
               libwebkitgtk-dev (>= 1.2.5-2.1),
               libsoup2.4-dev (>= 2.24),
               pkg-config,
               python-support (>= 0.5.3)
```

Of all the packages whose names were displayed, I installed the ones I hadn't installed already and updated any whose versions were too low, except that I didn't bother to install quilt, the one package that wasn't needed. I then made the "work" directory my current directory, and then:

```
cd Dieter*
cd src
```

At this point I was able to make any changes I wanted to any of the dozen or so .c source files. Then:

```
cd ..    # back to the Dieter* directory
make
find . -name uzbl-core
```

This compiled the code. I moved the uzbl-core file (if you know what a core file is, don't worry, this isn't one of those) into a directory that's mentioned in my PATH environment variable. In my case, the directory was ~/bin, which I was careful to make sure was in my PATH environment variable. (For more info on the PATH environment variable, look here.)

Well, that's the end of the discussion of downloading from Debian and compiling. Obviously, FreeBSD and Linux distributions other than Debian handle this differently.

Now that you have the source that corresponds to the uzbl provided by your OS distribution, do a dry run of compiling and running it. Make sure you're running the uzbl-core you've just compiled, not the stock one on your system.

go to top go to list of contents

getting uzbl from uzbl.org

go to top go to list of contents

This section describes how to download, build, and install the latest official release of uzbl. The description will have you do everything as non-root, including installation of the new uzbl (but not including the installation of any other packages on which uzbl depends), so that the new installation can coexist with the official one for your system.

Caution: I'm still testing this section. I ought to finish that in February 2015 or earlier. Stay tuned.

Before you start, be sure you know what you're getting yourself into. If you made the nodifications described in the earlier sections of this tutorial, you'll have to make them all over again for this new version of uzbl. You up for that? Good, I thought so. And changing the config file will be rather easy, because you originally read this warning and saved the original stock config file so you can diff it against your current config file to see all the changes you made, right? Good, I thought so. Let's begin.

The first step in building the latest release of uzbl from uzbl.org is to install all the software packages on which it depends. You can find a list of them by going here and searching for the strings "Dependencies" and "Make dependencies".

Another (implied) dependency is the "git" program, which you should use to download the latest official uzbl release.

The next step in getting the latest official release of uzbl is to make a working subdirectory (we'll call it "work"), and enter these shell commands:

```
git clone git://github.com/uzbl/uzbl.git work
cd work    # <--- Don't skip this, or the tutorial won't mean anything to you.
```

If, instead, you want the source code for the leading edge "next" version, the one not released yet:

```
git clone git://github.com/uzbl/uzbl.git work
cd work
git checkout next
```

Now that you have the source, you get to decide where the installed files go. They will be in three directories: bin, lib, and share. If you're root, you may be happy to know that the default location for these directories is /usr/local; if you want to change that to simply /usr, for example, it's easy. And if you're not root, changing the destination directory is not only easy, it's mandatory! In this tutorial, the examples will assume you want the destination directory to be /u/home/uzbl3, but you'll almost certainly want to use a different location.

So if you want to change the directory location, create a file called local.mk containing the following line (you already did the cd into the work directory, didn't you?):

```
PREFIX=/u/home/uzbl3
```

In theory, you should now be able to just let 'er rip. But the official release I'm using, 2012.05.14-1113-g69fa417, has an installation problem. To see whether yours does, finish the installation with the following shell commands:

```
make
make install
```

Then inspect the installed files with this shell command:

```
ls -lR /u/home/uzbl3   # or wherever the installed files went
```

Are all the files world-readable? If not, and you want them to be, you'll need to create a patch for the Makefile. I like to put this patch not in the work directory, but in its parent, so that it won't be wiped out the next time I clear out the work directory and "git" the source. So go back to the parent directory and create a file called Makefile.patch that looks like this. You can copy this one verbatim if you like (but no guarantees, obviously).

```
*** Makefile    Mon May 26 02:07:28 2014
--- Makefile.new        Mon May 26 02:16:03 2014
***************
*** 262,267 ****
--- 262,268 ----
        #$(INSTALL) -m644 README.event-manager.md $(DOCDIR)/README.event-manager.md
        cp -rv examples $(INSTALLDIR)/share/uzbl/examples
        chmod 755 $(INSTALLDIR)/share/uzbl/examples/data/scripts/*.sh $(INSTALLDIR)/share/uzbl/examples/data/scripts/*.py
+       chmod -R u=rwX,go=rX $(INSTALLDIR)
        $(INSTALL) -m644 uzbl.desktop $(INSTALLDIR)/share/applications/uzbl.desktop
        $(INSTALL) -m644 uzbl-browser.1 $(MANDIR)/man1/uzbl-browser.1
```

Then enter these shell commands:

```
cd work
patch -p0 < ../Makefile.patch
make clean
make
make install
```

Then you'll need to let the event manager know where the newly built Python packages are. To do so, first enter this command at the shell prompt:

```
find /u/home/uzbl3/lib -mindepth 2 -maxdepth 2
```

You'll get something that looks like this:

```
/u/home/uzbl3/lib/python3.2/site-packages
```

That's where the event manager should look for uzbl packages. We'll add that directory name to the (very short) uzbl-manager script. So look at file /u/home/uzbl3/bin/uzbl-event-manager. It will look something like this:

```
#!/usr/bin/python3
from uzbl import event_manager
event_manager.main()
```

Change it so it looks like this:

```
#!/usr/bin/python3
import sys
sys.path.insert(0,"/u/home/uzbl3/lib/python3.2/site-packages")
from uzbl import event_manager
event_manager.main()
```

Then you need to make sure that the bin directory to which you've just installed uzbl (for example, /u/home/uzbl3/bin) is in your PATH environment variable; for more info, look here.

At this point, you'd be tempted to take the new uzbl out for a spin, right? That's ok if the old uzbl and the new uzbl have the same format for the config files. If they do, you can skip the rest of this section, and go right to working through the tutorial again to make any desired changes to the new uzbl or its config file.

But if the format of the config files has changed, you don't want to take the new uzbl out for a spin just yet. If you're using a newer uzbl but you have a config file that has an older syntax, things won't work so well.

But you went ahead and did it, right? If you did, you'll notice at least three things. The first thing is that you got a boatload of error messages on your xterm screen (or your local equivalent). The second thing is that uzbl didn't seem to pay enough attention to your config file, so that uzbl misbehaved. The third thing, which you might not have noticed until you exited this premature running of uzbl, is that you now probably have a new directory in your current working directory. It looks like it's named "=", but there's a space after the "=" in the directory name. If you see this new directory, you should remove it thus:

```
rm -r "= "
```

So now would be a good time to customize your new version of uzbl by working through this tutorial again and customizing the Python script uzbl-tabbed and your new-format config file, which you should copy from (for example) /u/home/uzbl3/share/uzbl/examples/config/config.

But wait, before you go: You might want to switch back and forth between the old and new versions of uzbl, with the old and new formats of the config file. How do you have both the old and new config files at ~/.config/uzbl/config? You don't, obviously. You have to put one of them somewhere else. How is this done? Take the two simple steps described here.

The first step is to modify uzbl-tabbed so that just after the first line (the shebang line, "#!") you insert these two lines:

```
import os
os.environ.setdefault("XDG_CONFIG_HOME",os.path.join(os.environ["HOME"],".config1"))
```

Note that we're hijacking an environment variable from freedesktop.org (formerly known as the X Desktop Group, hence "XDG"). When you go back and make other changes suggested by this tutorial, you'll notice that several other changes, later in the code, are similar to this, but use .config instead of .config1. You can make those other changes to be either .config or .config1; the above change you've just made will override those other changes.

The second step is to modify uzbl-browser (in case you ever run it) so that just after the first line (the shebang line, "#!") you insert these lines:

```
if [ "$XDG_CONFIG_HOME" = "" ]
then
  XDG_CONFIG_HOME=$HOME/.config1
fi
```

When you next run the new uzbl, you'll notice that the user config files are stored under not ~/.config, but ~/.config1. When you go back through this tutorial and modify the user config files, these are the ones you want to modify. So wherever you see .config, read .config1.

go to top go to list of contents

primary and clipboard selections

go to top go to list of contents

There are two places you can store selected text: as the primary selection, and as the clipboard selection.

You store text from a web page as the primary selection by dragging over the text. You drag over text by moving the mouse pointer to one end of the text, clicking and holding down the mouse button, moving the mouse pointer to the other end of that text, and releasing the mouse pointer. You'll discover that the background area around the selected text is now of a different color. Even if you now click elsewhere so that the background area has reverted to its original color, your selected text will be the primary selection. You can perform this operation on the currently displayed web page regardless of whether you're in Command mode or Insert mode.

You copy the primary selection to the clipboard selection by pressing <Ctrl>C. This must be done in Insert mode.

You can paste text into a web form from either the primary selection or the clipboard selection. This must be done in Insert

mode, as follows:

Paste text from the primary selection by clicking where in the web form you want to insert and clicking the middle button of your mouse.

Paste text from the clipboard selection by clicking where in the web form you want to insert and pressing <Ctrl>V.

go to top go to list of contents

pid files and you

go to top go to list of contents

A pid file is a file which exists in a conventional place (or doesn't exist at all). Some programs should have only one copy running at a time, or one copy per user, or some similar restriction. Such a program finds its own process ID (pid) and writes that into a pid file. When the program stops running, it usually removes the pid file. I say "usually" because sometimes the program stops so abruptly that it cannot remove the pid file. It could be killed with a SIGKILL signal (9), or the system could crash or lose electrical power.

So when a program starts running and wants to see whether that instance of the program is a duplicate and therefore not needed, the program should check whether the pid file exists. If it doesn't exist, then the program should indeed run. But if the pid file exists, the program should check whether there's an actual process with that pid. If not, then the program should indeed run. But if the pid file exists and a process with that pid exists, the program should check whether that other process is actually running this program. It's unlikely, but possible, since the system will recycle pids which are no longer in use. And even this third check is vulnerable, because there could be some other program with the same name. This objection, however, should be considered only by the most paranoid among you.

The event manager uses the first two of these three checks. That's usually enough. It could be argued, though, that the third check, whether the other process is actually running the event manager, should be made.

But the event manager is not the only program which checks its pid file. Some versions of usbl-browser also check the event manager's pid file, and if it exists, then uzbl-browser doesn't bother to run the event manager. Unfortunately, it only makes the first of the three checks, so when the system recovers from abrupt death, uzbl-browser will mistakenly fail to run the event manager. The simplest solution is to modify uzbl-browser so it doesn't check the pid file at all, as we recommend here. But if you don't disable that check, you should at least make uzbl-browser make the first two checks of the event manager's pid file described above.

Four final observations. The first is that if you want to extend either the event manager or uzbl-browser to make the third check, don't use the /proc filesystem to do so if you want your change to be universally usable. Although Linux has the /proc filesystem, other Unixen, such as FreeBSD, don't normally have that.

The second observation is that it is really wisest to remove the check entirely from uzbl-browser. The only advantage to have the check in uzbl-browser is to make the machine run faster. But the percentage gain in speed for your system is so negligible as to be not worth the additional code complexity. As just an example of why the additionaly complexity is bad, what if in some future version of uzbl the event-manager is changed so that the pid file is placed somewhere else? Then you have this code in uzbl-browser which misleads anyone wishing to maintain it, because the code now does nothing. So don't complicate the code just to avoid a negligible slowdown of the program. This calls to mind two rules for optimizing code. The first rule is: do not optimize. The second rule is for expert coders: do not optimize yet.

The third observation is that using pid files is not a bulletproof way to synchronize things so that only one copy of the event manager is running per user. There is a race condition. Two copies of the event manager could be started up almost simultaneously. Both of them can then check whether the pid file exists; both of them find that it does not exist; and both of them create it. It's better to use some sort of lock file to synchronize things. (The fcntl module could be used for this for event managers written in Python, for example.) No need to write the pid into that lock file or into any file.

The fourth observation is that most of this stuff is a tempest in a teapot. The only important thing is to yank the event manager pid file check out of uzbl-browser. The rest could be considered overkill. Nothing here is mission critical. It's only a browser, not a nuclear reactor.

go to top go to list of contents

referrer fun fact

go to top go to list of contents

When you're on a web page and click on a link to another one, the software running on the new web site has knowledge of which web site you have just come from. This previous web site is known as the referrer, and the relevant URI is passed to the new web site's software as environment variable HTTP_REFERER (yep, only three R's). This is also true, for most browsers, if you open the new URI in a new tab or new window.

But it isn't true for uzbl. At least the version that I've tested, opening in a new tab or window will cause the information about the referrer to be lost. I consider this to be a feature, not a bug.

go to top go to list of contents

thoughts on Python

go to top go to list of contents

One major advantage of Python is that it's a quite readable language. A Python script meant for use with uzbl is likely to be simple enough that if you know any programming language, you'll be able to understand that script, and will probably learn some Python along the way.

A detailed look at the transition from Python 2 to Python 3 is here. It all seems reasonable, but I'm still leery of investing much effort in serious new software development using Python. (Hobby development, yes. Serious development, no.) Don't be telling me that Python 4 will never cause Python 3 to freeze. It probably won't happen, but with Python you can't say "never" with a straight face.

A good, but not perfect, resource for learning Python is, um, Learning Python, by Mark Lutz. He's quite thorough, and the book is deliciously long, but you need to check his work as you go. For example, he says:

> Dictionaries aren't sequences like lists and strings, but if you need to step through the items in a dictionary, it's easy -- calling the dictionary keys method returns all stored keys, which you can iterate through with a for.

But in Python 3, the keys method returns not all stored keys, but an iterator over the keys. The difference is that if you get this iterator, add a new item to the dictionary, and use the iterator, it will reflect the presence of the new key. (Lutz does point this out at other places in the book.) So read the book (or as much of it as you need), but if something doesn't seem quite right, stop and test it.

go to top go to list of contents

the difference between URI and URL

go to top go to list of contents

What's the difference between URI and URL? It's this:

```
p(URI-URL)
```

where p ranges from 0 to 1, and is the degree to which you wish to be pedantic. In other words, if you have no desire to be pedantic, the value of the expression is zero.

For a less flippant discussion of this issue, go here.

go to top go to list of contents

of historical interest

go to top go to list of contents