# HOWTO Create Python GUIs using HTML

...now supporting Python GtkMozEmbed and PyWebKitGtk!

**Version:** r38 2013-03-21 15:01:53 -0600
**Author:** David Baird
**Contact:** dhbaird@gmail.com

Table of Contents



# History

2013-03-21 21:02:33 UTC David Baird <dhbaird@gmail.com>

> This article is translated to Serbo-Croatian language by Anja Skrba from Webhostinggeeks.com. (Thanks, Anja.)

2009-10-25 22:29:24 UTC David Baird <dhbaird@gmail.com>

Refactoring code to auto-detect for WebKit or GtkMozEmbed and use whichever one it finds. Also, now making use of standard library functions `os.path.abspath` and `urllib.pathname2url` instead of manually trying to generate proper paths and URLs.

2009-06-21 13:30:57 UTC David Baird <dhbaird@gmail.com>

Fixed "kill_gtk_thread()" and "my_quit_wrapper()" by switching from synchronous_gtk_message to asynchronous_gtk_message. (Thanks Tim Kersten for pointing this out).

2009-05-16 07:36:37 UTC David Baird <dhbaird@gmail.com>

I updated this document to describe GtkMozEmbed. I rewrote the example code to support both GtkMozEmbed and PyWebKitGtk.

2009-05-16 06:27:53 UTC David Baird <dhbaird@gmail.com>

Fixed some typos (thanks to John Tantalo at Emend (Twitter) - a great tool for grammar nuts). I am also implementing a suggestion (from Y Combinator News) to use condition variables. I also renamed the "execute" functions to the superior name of "worker".

2009-05-16 02:22:09 UTC David Baird <dhbaird@gmail.com>

I'm updating this document by moving the JavaScript "send()" function into a separate file named "ipc.js"; Prior to this, "send()" was being injected using WebKit's ".execute_script()" method. By refactoring "send()", it will be easier to use gtkmozembed (Mozilla), whose JavaScript injection is slightly different, as an alternative to WebKit.

2009-04-22 David Baird <dhbaird@gmail.com>

First release, explains how to embed WebKit and exchange messages with Python.

# Overview

In this howto, I explain the following concepts:

1. How to launch a GUI toolkit in a separate thread, and how to communicate with this thread.
2. How to embed a web browser inside the GUI, and how to communicate with this embedded web browser without using sockets. By not using sockets, we don't have to worry about socket security, and we don't have to worry about allocating a socket. (Note: sockets are still available, but you don't have to use them unless you want to.) Absolutely no web server is required (unless you specifically want to use one for whatever reason). In other words, we will have AJAX-like and AJAX-Push/Comet-like functionality to communicate with Python but without actually having to use AJAX.

This howto is about using HTML and associated web technologies (JavaScript, CSS, JSON, ...) to create a GUI for a standalone application in Python.

What this howto is not about:

- This howto is not about running a web server or about how to use a web framework. You do not need to run a web server or even *know* how to run a web server to use this howto.

# Motivation

Over a period of a few years, I have attempted to create GUIs. I have left disappointed each time. There are two main discouraging factors that I encountered:

1. Toolkit lock-in. The various toolkits have their own *main loops* and event processing frameworks. For example, if you want to write a GUI that integrates with sockets, it is hard to use the "select" function. Usage of select requires the programmer to write their own main loop, and this is at conflict with toolkits that provide a prepackaged main loop.

   > Solution: Spawn two threads. One thread is for just the GUI toolkit. The other thread runs your own personalized main loop. The two threads communicate by sending messages to each other (e.g. by using thread-safe FIFOs or queues).
   >
   > Note about threads and GUIs: It is important to strictly follow a message passing model: you generally CANNOT have multiple threads accessing GUI functions lest X11 libraries shall segfault; Thus, exactly one thread should be blessed to manage the GUI directly. All other threads can manage the GUI indirectly by sending messages to this one blessed thread. I will explain how to do this further in this document.

2. Expression. I found it hard to express ideas in toolkits because I found it hard to visualize and understand the structures which underlie GUI toolkits. Granted, some toolkits have integrated XML support for constructing their GUIs, but I felt this still required a lot of additional learning and was missing the flexibility that HTML specifically offers. After working with and enjoying HTML for some time, I started thinking: what if I could use HTML to create GUIs?

   > Solution: Embed a web browser into your GUI. Thus, you have the flexibility of using a toolkit for creating a few basic things (such as menus and menu items), but you can use the web browser to do all the heavy lifting. This raises an additional problem which will be discussed in this document: how to transfer messages between the web browser and your personalized main loop. The conceptual underlying structure of HTML is the DOM which has a virtually one-to-one mapping back to HTML. I feel that this is an easy model to understand and manipulate, and it is also powerful.

I eventually came to these conclusions after working with a variety of event

processing loops involving sockets, serial ports, audio data, MIDI data, and GUIs. I tinkered with some MVC (Model View Controller) approaches to doing GUI design and felt they helped, but still didn't satisfy me. I finally found something that seemed to make sense when I started to work on simple web-based applications. Web based applications have a very strict separation of GUI (running on a remote browser) from your code (running on a server). Thus, your main code does not have to be locked into the GUI toolkit, per se. Or at least, it does not feel as such.

HTML is a good technology. It integrates with several other technologies including:

- CSS
- JavaScript (ECMAScript)
- XML DOM
- SVG
- MathML
- HTML5 Canvases
- Several JavaScript extensions, such as jQuery
- ...and the list keeps growing...

HTML is very composable, meaning it is easy to copy-and-paste HTML designs together. I would argue that it is harder to copy-and-paste GUI toolkit designs together, thus making HTML superior to GUI toolkits in this regard. There are also many templates and online resources for constructing high quality HTML documents.

It is possible to tinker with an HTML-based user interface, even if the underlying application logic is missing. This means that the process of designing the GUI does not have to be tightly coupled to writing the main application logic.

Another benefit of HTML is that it is open to non-programmers. There are many web designers who can now contribute to the design of application user interfaces if those interfaces are based on HTML.

# Launching the Toolkit in a Separate Thread

## Launching GTK

This is probably the simplest part of this article: launching GTK in its own thread, thus leaving our main thread available for implementing our own personalized main loop:

```
import gtk
import thread

gtk.gdk.threads_init()
thread.start_new_thread(gtk.main, ())

# Yes, it really is that easy.  GTK is now running in its own thread,
# waiting for us to send messages to it.
```

Continue by reading the "Message Passing" section to see what to do next.

# Message Passing

## Introduction

Message passing is the means by which processes communicate. Much like humans communicate by talking to each other or writing notes to each other, so do computer programs communicate with each other. Sockets and pipes are prime examples of message passing technologies. Shared memory can also be used for message passing.

Programming with functions or objects is also a form of message passing. In this case, calling a function or method is equivalent to sending a message to your program or to an object in your program. Essentially, your program is communicating with itself. But that's still message passing. Just like how people sometimes like to talk to themselves (well, at least I do).

Some programming language libraries have special queues that can be used to pass messages between threads in a multithreaded application. Some languages, like Erlang or Termite Scheme, have message passing so ingrained that you'd be crazy not to use those features.

## Message Passing with GTK

Here I am going to explain message passing between GTK and your custom main loop. *I am not explaining message passing inside GTK* (e.g. between GTK widgets - you'll have to read the GTK docs for more info about that).

GTK has a so-called "idle loop." This idle loop is how you can interface with GTK's main loop. GTK has a special function called "idle_add" which lets us add messages to be processed in the idle loop. I am going to describe two types of messages here:

1. Asynchronous messages

   These are messages that you insert into GTK's idle loop, but you *do not care* about getting a response.

2. Synchronous messages:

   These are messages that you insert into GTK's idle loop, but you *do care* about getting a response.

In the Python programming language, functions are first-class citizens that can be passed around just like normal data. Thus, a message really is just a function reference (or a functor) that we inject into GTK's idle loop. GTK's idle loop will merely call/execute the function. In C, this is similar to putting a function pointer into a queue which another loop dequeues and then calls the function associated with that function pointer.

### Asynchronous GTK Message

Here is the magic trick for asynchronous GTK messages in Python. By the way, if this code looks like giberish to you, then read up on *variable arguments* and *keyword arguments* in Python. You'll also want to read about *tuple unpacking* in Python. You might also want to read about *higher order functions* which is a concept from functional programming. (Sorry, I know that is a long list of things to read about.):

```
import gobject

# This is a function which takes a function as an argument and
# returns yet another function (this is what higher order functions
# are all about!).  (Side note: this can be invoked/applied with
# Python's decorator syntax, '@', if you so desire.):
def asynchronous_gtk_message(fun):

    def worker((function, args, kwargs)):
        # Apply is a function that takes a function as its first
        # argument and calls it with positional argument taken
        # from "args" and keyword arguments taken from "kwargs":
        apply(function, args, kwargs)

    # This is a special type of function known as a "closure"
    # (although it is not quite as advanced as a closure in Lisp
    # or Ruby.  Challenge question: do you know why?).  In C++,
    # a class-based functor which defines operator() (or Boost.Lambda)
    # is the closest you can get to a closure:
    def fun2(*args, **kwargs):
        # "worker" is none other than the function just defined
        # above (with "def worker"):
        gobject.idle_add(worker, (fun, args, kwargs))

    # Here, the closure is returned and must be called at some later
    # point in the program:
    return fun2
```

The code above has a lot of comments, so here is the same code again with all the comments stripped out (some people may find this easier to read):

```
import gobject

def asynchronous_gtk_message(fun):

    def worker((function, args, kwargs)):
        apply(function, args, kwargs)

    def fun2(*args, **kwargs):
        gobject.idle_add(worker, (fun, args, kwargs))

    return fun2
```

Despite being a few lines of code, there are some pretty deep programming concepts required to understand the code above. But it is concise and expressive and fairly hard to introduce bugs (because it is so simple).

Here is an example of using "asynchronous_gtk_message" to manipulate a web browser widget (specifically, WebKit) running in GTK:

```
browser = ... # read about synchronous_gtk_message below to see what goes here
...
```

```
asynchronous_gtk_message(browser.execute_script)('alert("oops")')

# or, alternatively:
async_execute_script = asynchronous_gtk_message(browser.execute_script)
async_execute_script('alert("oops")')
```

Note that "asynchronous_gtk_message" does not actually do anything. All it does is return a special function (remember the closure from above?). And it is that special function which we must call whenever we want to *actually* send an asynchronous message. Notice how we ignore the return value from the message? Well, that is what makes it asynchronous; And fast.

## Synchronous GTK Message

What if we need a return value? Then we need a synchronous message. Let's say, we want to send a message to GTK saying "please create a new GTK window and give me back a reference to that new window so that I can embellish it later on." This is what synchronous messages are good for. They take longer to execute since you have to sit around and wait for the return value. (There are tricks to get around this waiting... but that is for another article by another author):

```
import time
import gobject

def synchronous_gtk_message(fun):

    class NoResult: pass

    def worker((R, function, args, kwargs)):
        R.result = apply(function, args, kwargs)

    # WARNING: I know the busy/sleep polling loop is going to offend
    #          the sensibilities of some people.  I offer the following
    #          justifications:
    #          - Busy/sleep loops are a simple concept: easy to
    #            implement and they work in situations where you
    #            may not have condition variables (like checking your
    #            email with fetchmail).
    #          - If you do use a synchronous message, it will probably
    #            complete very rapidly, thus very few CPU cycles will
    #            by wasted by this busy loop (thanks to the sleep).
    #          - You probably shouldn't be using synchronous messages
    #            very often anyhow.  Async is cooler :-)
    #          - If this code is anything bad, it is probably that the
    #            sleep() adds a bit of undesired latency before the result
    #            can be returned.
    #          If this still doesn't appeal to you, then keep reading
    #          because I do this again with condition variables.
    def fun2(*args, **kwargs):
        class R: pass
        R.result = NoResult
        gobject.idle_add(callable=worker, user_data=(R, fun, args, kwargs))
        while R.result is NoResult:
            time.sleep(0.01)
        return R.result

    return fun2
```

Well, that was slightly more complicated than the asynchronous case; The primary difference is the addition of "R.result" and a loop that waits for "R.result" to reference

anything besides "NoResult". Here is a more compact form of the above code which some people may prefer:

```
import time
import gobject

# Slightly more compact version of the above code:
def synchronous_gtk_message(fun):

    class NoResult: pass

    def worker((R, function, args, kwargs)):
        R.result = apply(function, args, kwargs)

    def fun2(*args, **kwargs):
        class R: result = NoResult
        gobject.idle_add(callable=worker, user_data=(R, fun, args, kwargs))
        while R.result is NoResult: time.sleep(0.01)
        return R.result

    return fun2
```

If you're not keen on the busy/sleep loop above, here's another version that uses condition variables instead:

```
# non-busy/sleep version of the above code:
def synchronous_gtk_message2(fun):
    import threading

    def worker((R, condition, function, args, kwargs)):
        R.result = apply(function, args, kwargs)
        condition.acquire()
        condition.notify()
        condition.release()

    def fun2(*args, **kwargs):
        condition = threading.Condition()
        condition.acquire()
        class R: pass
        gobject.idle_add(worker, (R, condition, fun, args, kwargs))
        condition.wait()
        condition.release()
        return R.result

    return fun2
```

Here's another option that doesn't work, so don't use it:

```
# non-working/broken version of the above code :-P
def synchronous_gtk_message3(fun):

    # This doesn't work for me.  Can anyone shed some light on this?
    #
    # Besides, http://library.gnome.org/devel/gdk/unstable/gdk-Threads.html
    # gives a warning that this may only work for X11 but not Win32:
    #
    #     GTK+ is "thread aware" but not thread safe - it provides a global
    #     lock controlled by gdk_threads_enter()/gdk_threads_leave() which
    #     protects all use of GTK+. That is, only one thread can use GTK+
    #     at any given time.
    #
    #     Unfortunately the above holds with the X11 backend only. With the
```

```
    #     Win32 backend, GDK calls should not be attempted from multiple
    #     threads at all.
    def fun2(*args, **kwargs):
        gtk.gdk.threads_enter()
        try:     x = apply(fun, args, kwargs)
        finally: gtk.gdk.threads_leave()
        return x

    return fun2
```

Anyways, here is an example of using "synchronous_gtk_message":

```
# Use synchronous messages here:
window = synchronous_gtk_message(gtk.Window)()
browser = synchronous_gtk_message(webkit.WebView)()

# Use asynchronous messages here:
asynchronous_gtk_message(window.set_default_size)(800, 600)
asynchronous_gtk_message(window.show_all)()
```

# Message Passing with WebKit

When communicating with WebKit, there will be two types of messages. Unlike with GTK though, these two WebKit messages are both asynchronous:

1. Asynchronous send (web_send / execute_script)

   To send a message from Python to WebKit, we use the "execute_script" method of a WebKit browser widget. There is a wrapper function called "web_send" which will invoke "execute_script".

2. Asynchronous receive (web_recv / title-changed)

   For WebKit to send a message to Python, a hack is required. The WebKit browser features a callback that is triggered whenever the "title" of the embedded web page is changed. We are not using the title for anything better, so why not hijack it and use it for message passing? We can connect the "title-changed" event notification of a WebKit browser to a function which enqueues the title's value into a queue. Then, the main loop can be woken up to check the queue, or it can poll the queue at its own leisure. There is a wrapper function called "web_recv" which interfaces to this queue.

Here is code for launching a browser and definining the web_send and web_recv functions:

```
import Queue

import gtk
import webkit

def launch_browser(uri, echo=True):
    # WARNING: You should call this function ONLY inside of GTK
    #          (i.e. use synchronous_gtk_message)

    window = gtk.Window()
    box = gtk.VBox(homogeneous=False, spacing=0)
```

```
    browser = webkit.WebView()

    window.set_default_size(800, 600)
    # Optional (you'll read about this later in the tutorial):
    window.connect('destroy', Global.set_quit)

    window.add(box)
    box.pack_start(browser, expand=True, fill=True, padding=0)

    window.show_all()

    # Note: All message passing stuff appears between these curly braces:
    # {
    message_queue = Queue.Queue()

    def title_changed(widget, frame, title):
        if title != 'null': message_queue.put(title)

    browser.connect('title-changed', title_changed)

    def web_recv():
        if message_queue.empty():
            return None
        else:
            msg = message_queue.get()
            if echo: print '>>>', msg
            return msg

    def web_send(msg):
        if echo: print '<<<', msg
        asynchronous_gtk_message(browser.execute_script)(msg)
    # }


    browser.open(uri)

    return browser, web_recv, web_send

uri = 'http://www.google.com/'
browser, web_recv, web_send = synchronous_gtk_message(launch_browser)(uri)
```

Next, somewhere in your HTML/JavaScript, you'll need to define the "send()" function which sends a message to Python by changing the HTML title. Here's an example which I recommend putting in a file named "ipc.js":

```
function send(msg) {
    document.title = "null";
    document.title = msg;
}
```

# Message Passing with Mozilla (GtkMozEmbed)

The process for GtkMozEmbed is very similar to WebKit, so I recommend that you read the WebKit section. Here, I'm simply going to highlight the differences between WebKit and GtkMozEmbed.

```
import Queue

import gtk
#import webkit # <- webkit
import gtkmozembed # <- gtkmozembed
```

```
import urllib # <- gtkmozembed (for encoding JavaScript strings)

def launch_browser(uri, echo=True):
    # WARNING: You should call this function ONLY inside of GTK
    #          (i.e. use synchronous_gtk_message)

    window = gtk.Window()
    box = gtk.VBox(homogeneous=False, spacing=0)
    #browser = webkit.WebView() # <- webkit (obviously)
    browser = gtkmozembed.MozEmbed() # <- gtkmozembed

    # gtkmozembed only (for webkit, we use its .execute_script()):
    def inject_javascript(script):
        uri = 'javascript:%s' % urllib.quote(script + '\n;void(0);')
        browser.load_url(uri)

    window.set_default_size(800, 600)
    # Optional (you'll read about this later in the tutorial):
    window.connect('destroy', Global.set_quit)

    window.add(box)
    box.pack_start(browser, expand=True, fill=True, padding=0)

    window.show_all()

    # Note: All message passing stuff appears between these curly braces:
    # {
    message_queue = Queue.Queue()

    # webkit:
    #def title_changed(widget, frame, title):
    #    if title != 'null': message_queue.put(title)
    # gtkmozembed:
    def title_changed(*args):
        title = browser.get_title()
        if title != 'null': message_queue.put(title)

    #browser.connect('title-changed', title_changed) # <- webkit
    browser.connect('title', title_changed) # <- gtkmozembed

    def web_recv():
        if message_queue.empty():
            return None
        else:
            msg = message_queue.get()
            if echo: print '>>>', msg
            return msg

    def web_send(msg):
        if echo: print '<<<', msg
        #asynchronous_gtk_message(browser.execute_script)(msg) # <- webkit
        asynchronous_gtk_message(inject_javascript)(msg) # <- gtkmozembed
    # }

    #browser.open(uri) # <- webkit
    browser.load_url(uri) # <- gtkmozembed

    return browser, web_recv, web_send

uri = 'http://www.google.com/'
browser, web_recv, web_send = synchronous_gtk_message(launch_browser)(uri)
```

...and don't forget about "ipc.js" (described in the WebKit section):

```
function send(msg) {
```

```
    document.title = "null";
    document.title = msg;
}
```

# Polishing Things Up

## The Quit Wrapper

When exiting a multithreaded application, you need to be able to instruct *all* threads to terminate. Additionally, Python has an annoyance associated with signals and threads: you can never know which thread a signal, such as the Ctrl-C/SIGINT, will be sent to.

I solve both of these problems by declaring a shared, global "quit" variable. I then install a SIGINT (Ctrl-C) signal handler which sets the "quit" variable to True if trigerred. It won't matter which thread actually receives SIGINT, because the same global "quit" variable will be set in all cases.

Any process/thread in the program can instruct the program to terminate by setting this global "quit" variable. For example, if the user clicks the File/Quit menu item or pressed Ctrl-Q, the entire program can terminate by setting the "quit" variable.

Without further ado, here's the code:

```
import signal

class Global(object):
    quit = False
    @classmethod
    def set_quit(cls, *args, **kwargs):
        cls.quit = True

def my_quit_wrapper(fun):
    signal.signal(signal.SIGINT, Global.set_quit)
    def fun2(*args, **kwargs):
        try:
            x = fun(*args, **kwargs) # equivalent to "apply"
        finally:
            asynchronous_gtk_message(gtk.main_quit)()
            Global.set_quit()
        return x
    return fun2
```

And here's how to use it:

```
def main():
    while not Global.quit:
        # ... do some stuff ...
        time.sleep(0.1)

if __name__ == '__main__': # <-- this line is optional
    my_quit_wrapper(main)()
```

# A Complete Example

## Prerequisites

You need to install the following packages:

- PyWebKitGtk or Python GtkMozEmbed
- Python Simplejson

In Debian/Ubuntu, you can follow these steps to install those packages automatically:

```
apt-get install python-webkitgtk
apt-get install python-simplejson
# ummm... python-gtkmozembed...?
```

In Gentoo/Funtoo:

```
#emerge -va pywebkitgtk # <-- not yet working with Python 2.6 :-(
#emerge -va simplejson # <-- installs Python 2.5 :-(
emerge gtkmozembed-python # yay! finally something that works!
easy_install simplejson # this is how we bypass emerge
```

## JavaScript Code

This small JavaScript module will be used to allow sending messages from the browser to Python.

ipc.js:

```
function send(msg) {
    document.title = "null";
    document.title = msg;
}
```

## HTML Code

The following code uses standard JavaScript (the only uncertainty I have is whether or not "DOMContentLoaded" is a standard, but WebKit certainly supports it regardless). Even though none of the examples here require it, I *highly recommend* that you use jQuery (unless there is some other JavaScript library that you already like). jQuery allows you to navigate XHTML by using CSS Selectors. jQuery makes it very easy to configure event notifications and to modify the structure of an XHTML document. jQuery also has the ability to work with aggregates: making a single operation apply simultaneously to several XML elements.

demo.xhtml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
      PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
```

```
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:svg="http://www.w3.org/2000/svg"
      xmlns:xlink="http://www.w3.org/1999/xlink">

  <head>

    <title></title>

    <script src="ipc.js" type="text/javascript"></script>

    <!--
    Here are some jQuery plugins that I recommend:

    <script src="js/jquery-1.2.6.min.js" type="text/javascript"></script>
    <script src="js/jquery-plugins/json.js" type="text/javascript"></script>
    <script src="js/jquery-plugins/jquery.hotkeys-0.7.8.js" type="text/javascript"></script>
    <script src="js/jquery-plugins/jquery.intercept.js" type="text/javascript"></script>
    -->

    <link href="demo.css" rel="stylesheet" type="text/css"></link>

    <script type="text/javascript">
// <![CDATA[

// NOTE: jQuery would make this all much cleaner!  You need it!

function got_a_click() {
    // NOTE: send is a JavaScript function that is defined in "ipc.js".
    //       This send function is how messages are sent from HTML
    //       to Python:
    send('"got-a-click"');
}

function document_ready() {
    send('"document-ready"');
    document.getElementById('messages').addEventListener('click', got_a_click, false);
}

document.addEventListener('DOMContentLoaded', document_ready, false);

// If you were using jQuery (with the JSON plugin), you'd write the above
// code like this:
//
// Instead of addEventListener('DOMContentLoaded', ...), do this:
// $(document).ready(function() {
//     // Instead of getElementById('messages').addEventListener('click',...),
//     // do this:
//     // http://docs.jquery.com/Events/bind
//     $('#messages').bind('click', got_a_click);
//     // ...or try this shortcut version of bind:
//     $('#messages').click(got_a_click);
//     // http://jollytoad.googlepages.com/json.js provides $.toJSON(...):
//     send($.toJSON('document.ready'));
// })

// ]]>
</script>


  </head>

  <body>

    <h1>Python + Web GUI Demo</h1>
```

```
    <h2>Uptime</h2>

    <p class="uptime">
      Python uptime:
      <span id="uptime-value">?</span> seconds.
    </p>

    <h2>Messages</h2>

    <p id="messages">
      Click here (yes, anywhere here)...<br/>
    </p>

  </body>

</html>
```

# Python Code

The code is split up into two modules: webgui.py and demo.py. These are both shown below:

webgui.py:

```
import time
import Queue
import thread
import urllib

import gtk
import gobject

try:
    import webkit
    have_webkit = True
except:
    have_webkit = False

try:
    import gtkmozembed
    have_gtkmozembed = True
except:
    have_gtkmozembed = False

class UseWebKit: pass
class UseGtkMozEmbed: pass

if False: pass
elif have_webkit:
    use = UseWebKit
elif have_gtkmozembed:
    use = UseGtkMozEmbed
else:
    raise Exception('Failed to load any of webkit and gtkmozembed modules')

#use = UseGtkMozEmbed # <- choose your desired implementation here

class WebKitMethods(object):

    @staticmethod
    def create_browser():
        return webkit.WebView()
```

```
    @staticmethod
    def inject_javascript(browser, script):
        browser.execute_script(script)

    @staticmethod
    def connect_title_changed(browser, callback):
        def callback_wrapper(widget, frame, title): callback(title)
        browser.connect('title-changed', callback_wrapper)

    @staticmethod
    def open_uri(browser, uri):
        browser.open(uri)




class GtkMozEmbedMethods(object):

    @staticmethod
    def create_browser():
        return gtkmozembed.MozEmbed()

    @staticmethod
    def inject_javascript(browser, script):
        uri = 'javascript:%s' % urllib.quote(script + '\n;void(0);')
        browser.load_url(uri)

    @staticmethod
    def connect_title_changed(browser, callback):
        # XXX: probably you should cross your fingers and hope browser
        #      isn't sending title messages too quickly...?
        def callback_wrapper(*args): callback(browser.get_title())
        browser.connect('title', callback_wrapper)

    @staticmethod
    def open_uri(browser, uri):
        browser.load_url(uri)



if use is UseWebKit:
    implementation = WebKitMethods

if use is UseGtkMozEmbed:
    implementation = GtkMozEmbedMethods






def asynchronous_gtk_message(fun):

    def worker((function, args, kwargs)):
        apply(function, args, kwargs)

    def fun2(*args, **kwargs):
        gobject.idle_add(worker, (fun, args, kwargs))

    return fun2


def synchronous_gtk_message(fun):

    class NoResult: pass
```

```
    def worker((R, function, args, kwargs)):
        R.result = apply(function, args, kwargs)

    def fun2(*args, **kwargs):
        class R: result = NoResult
        gobject.idle_add(worker, (R, fun, args, kwargs))
        while R.result is NoResult: time.sleep(0.01)
        return R.result

    return fun2

def launch_browser(uri, quit_function=None, echo=True):

    window = gtk.Window()
    browser = implementation.create_browser()

    box = gtk.VBox(homogeneous=False, spacing=0)
    window.add(box)

    if quit_function is not None:
        # Obligatory "File: Quit" menu
        # {
        file_menu = gtk.Menu()
        quit_item = gtk.MenuItem('Quit')
        accel_group = gtk.AccelGroup()
        quit_item.add_accelerator('activate',
                                  accel_group,
                                  ord('Q'),
                                  gtk.gdk.CONTROL_MASK,
                                  gtk.ACCEL_VISIBLE)
        window.add_accel_group(accel_group)
        file_menu.append(quit_item)
        quit_item.connect('activate', quit_function)
        quit_item.show()
        #
        menu_bar = gtk.MenuBar()
        menu_bar.show()
        file_item = gtk.MenuItem('File')
        file_item.show()
        file_item.set_submenu(file_menu)
        menu_bar.append(file_item)
        # }
        box.pack_start(menu_bar, expand=False, fill=True, padding=0)

    if quit_function is not None:
        window.connect('destroy', quit_function)

    box.pack_start(browser, expand=True, fill=True, padding=0)

    window.set_default_size(800, 600)
    window.show_all()

    message_queue = Queue.Queue()

    def title_changed(title):
        if title != 'null': message_queue.put(title)

    implementation.connect_title_changed(browser, title_changed)

    implementation.open_uri(browser, uri)

    def web_recv():
        if message_queue.empty():
            return None
        else:
```

```
            msg = message_queue.get()
            if echo: print '>>>', msg
            return msg

    def web_send(msg):
        if echo: print '<<<', msg
        asynchronous_gtk_message(implementation.inject_javascript)(browser, msg)

    return browser, web_recv, web_send


def start_gtk_thread():
    # Start GTK in its own thread:
    gtk.gdk.threads_init()
    thread.start_new_thread(gtk.main, ())

def kill_gtk_thread():
    asynchronous_gtk_message(gtk.main_quit)()
```

demo.py:

```
import signal
import os
import time
import urllib

from simplejson import dumps as to_json
from simplejson import loads as from_json

from webgui import start_gtk_thread
from webgui import launch_browser
from webgui import synchronous_gtk_message
from webgui import asynchronous_gtk_message
from webgui import kill_gtk_thread

class Global(object):
    quit = False
    @classmethod
    def set_quit(cls, *args, **kwargs):
        cls.quit = True


def main():
    start_gtk_thread()

    # Create a proper file:// URL pointing to demo.xhtml:
    file = os.path.abspath('demo.xhtml')
    uri = 'file://' + urllib.pathname2url(file)
    browser, web_recv, web_send = \
        synchronous_gtk_message(launch_browser)(uri,
                                                quit_function=Global.set_quit)

    # Finally, here is our personalized main loop, 100% friendly
    # with "select" (although I am not using select here)!:
    last_second = time.time()
    uptime_seconds = 1
    clicks = 0
    while not Global.quit:

        current_time = time.time()
        again = False
        msg = web_recv()
        if msg:
            msg = from_json(msg)
            again = True
```

```
        if msg == "got-a-click":
            clicks += 1
            web_send('document.getElementById("messages").innerHTML = %s' %
                     to_json('%d clicks so far' % clicks))
            # If you are using jQuery, you can do this instead:
            # web_send('$("#messages").text(%s)' %
            #          to_json('%d clicks so far' % clicks))

        if current_time - last_second >= 1.0:
            web_send('document.getElementById("uptime-value").innerHTML = %s' %
                     to_json('%d' % uptime_seconds))
            # If you are using jQuery, you can do this instead:
            # web_send('$("#uptime-value").text(%s)'
            #        % to_json('%d' % uptime_seconds))
            uptime_seconds += 1
            last_second += 1.0


        if again: pass
        else:     time.sleep(0.1)


def my_quit_wrapper(fun):
    signal.signal(signal.SIGINT, Global.set_quit)
    def fun2(*args, **kwargs):
        try:
            x = fun(*args, **kwargs) # equivalent to "apply"
        finally:
            kill_gtk_thread()
            Global.set_quit()
        return x
    return fun2


if __name__ == '__main__': # <-- this line is optional
    my_quit_wrapper(main)()
```

## Screenshot

# Resources

- Appcelerator Titanium - this is a tool for creating desktop applications using web technology. It appears to be a nicely integrated environment and can support the use of mixed languages, including Python, Ruby, and C++ as well as JavaScript:
    - http://titaniumapp.com/
    - http://www.appcelerant.com/
- Awesomium "Awesomium is a library that makes it easy for developers to embed web-content in their applications."
- Python and XULRunner (I haven't done this yet, but might be useful):
    - http://pyxpcomext.mozdev.org/no_wrap/tutorials/pyxulrunner/python_xulrunner_about.html Creating Python GUI Applications using XULRunner
    - http://michaelthornhill.blogspot.com/2005/07/giving-python-legs-with-xulrunner.html Giving Python legs with XULRunner
- CSS at W3Schools
- CSS Selectors (and here for CSS3)
- JavaScript (ECMAScript)
- XML DOM
    - http://www.w3schools.com/dom/default.asp XML DOM Tutorial
    - http://www.w3.org/TR/DOM-Level-2-Core/core.html
- Embedding SVG in XHTML
    - http://benjamin.smedbergs.us/blog/2008-12-22/using-svg-on-the-web/

    - https://developer.mozilla.org/en/Code_snippets/Embedding_SVG
    - http://www.w3.org/TR/SVG11/

- HTML5 Canvases

    - http://www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.html
    - https://developer.mozilla.org/En/Canvas_tutorial

- MathML

    - http://en.wikipedia.org/wiki/MathML
    - http://www.w3.org/TR/REC-MathML/
    - NOTE: To my knowledge, MathML is not supported in WebKit yet.

- jQuery

    - http://jquery.com/
    - http://www.youtube.com/watch?v=8mwKq7_JlS8 jQuery at Google TechTalks (yes indeed, a 12-year old is giving the talk) (note: fast forward about 3 minutes in to get to the core of the talk)

- JSON (a good compromise when XML is too much and S-expressions are too little ... assuming you're not obsessed with YAML ;-))

    - http://www.json.org/
    - http://en.wikipedia.org/wiki/JSON

- Creating quality design with HTML and CSS

    - http://www.alistapart.com/ One of my favorite design sites
    - Jeffrey Zeldman's book "Designing With Web Standards" (this book is what motivated me to take time to learn web standards)

- X3D (the XML version of VRML)...? I'm not really sure how to get this working yet. If you have any thoughts, drop me a line.

---

View document source. Generated on: 2013-03-22 10:49 UTC. Generated by Docutils from reStructuredText source.