



Alan Sellers

a record of my tech and gadget making adventures



[Home](#) [Projects](#) [Resources](#) [Blog](#) [Contact](#)

The Button – Websocket and Raspberry Pi

Background

In the first iteration of this project, [The Button](#), I implemented a simple web server to display the status of the sales goal set for our company, [Overgroup](#). It used [AJAX](#) calls to communicate between the server and client browser. But because the client browser [polls](#) every second, to make updates close to real time, it proved [unscalable](#) since the load on the Raspberry Pi increase [substantially](#) with every browser that connected.

What would be most ideal is some form of communication that's instant but only transmits when needed. This made me think of when I used Sockets in college to implement an instant message program.

A socket is kind of like a direct telephone wire to someone or something you want to communicate with. The line is always open once established and you talk only when you need to. Whatever is at the other end doesn't have to do anything until it hears you say something. That makes things much more efficient on both ends. But in this case I'm dealing with a web page, not the lower levels of the network layer.

The answer to this is something called a [WebSocket](#), something that's been

around a while but only recently standardized. It's the same idea as a Socket, but it's implemented within the HTTP protocol, which means I can use it on nearly any page if it's served to an HTML5 compliant browser. But it will require some recoding of the web server I implemented.

Goal

Refactor the web server so that it handles websocket connectivity.

Implementation

Since I'm only refactoring the web server portion I won't repeat the hardware specifications. If you want to see that, refer to my first article: [The Button](#). And you can find all the code for this new version [here](#).

The first version of the web server used the `web.py` library for the web server. Unfortunately, `it doesn't support WebSockets`, so I searched for something that does. I found [Tornado](#), another web framework, but this one `is geared for asynchronous` (non-blocking I/O) networking, like WebSockets. It works a lot like `web.py` as far as the directory setup and a lot of the code you use, but there are some small differences.

While I'm implementing a new web framework I'll go ahead and modularize my program into separate files for the objects I'm using. So here's an outline of the file structure:

Repository.py

Libraries Used:

- sqlite3

Houses the Repository object I used before except I renamed Increment_Count function to Set_Count. This object handles reading and writing to the database that stores the current number of sales we have left. That way if the device or application is rebooted we maintain the correct count.

Button.py

Libraries Used:

- threading

- time
- RPi.GPIO

Houses the Button object I used in the first iteration of this application. This object produces a thread that monitors for button presses.

websocketserver.py

Libraries Used:

- tornado.web
- tornado.websocket
- tornado.httpserver
- tornado.ioloop
- tornado.gen
- Button
- Repository
- threading
- time

This is where I implement the server itself and is where the most significant

changes lie. The major sections of this part of the app now are:



- **MainHandler** – Obj – handles HTTP requests, in other words the web server part of the app.
- **WebSocketHandler** – Obj – handles web socket requests and maintains connection.
- **ButtonMonitor** – Function – Polls Button object for a button press event, when a press happens it updates the database and then messages all web socket connections with the new count #.
- **__main__** – Main Routine – Runs on startup, configures objects and spins up their threads.

The MainHandler is almost **identical** to the salesstatus method in the first version of my app. It consists of a Get and Post method that each fire for the corresponding HTTP request method. Tornado, however, makes things a little easier and lets you just tell it what file to **render**, and it doesn't matter where it is so long as you tell it in the render method call. I just have one HTML file, like before, and I've put it in the root directory.

```
class MainHandler(tornado.web.RequestHandler):
```

```
@tornado.web.asynchronous
def get(self):
    self.render("default.html")

def post(self):
    self.render("default.html")
```

A **subtle** difference with this version, though, is the inclusion of a "decorator", the `@tornado.web.asynchronous` line of code. This instructs Tornado to create **asynchronous connections**. Basically, it just allows the handler to manage more than one connection at the same time, otherwise it can only handle one at a time. Also, I probably shouldn't be rendering the HTML file again on POST, I'll likely change that but nothing POSTs to the server at the moment so it's not hurting anything being there.

The WebSocketHandler is a new beast. Like the MainHandler, there's a method for each type of request or event that happens in the life of a web socket.

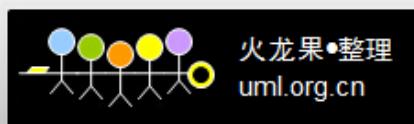
```
ler):  
    waiters = set()  
  
    def open(self):  
        self.set_nodelay(True)  
        print('Socket Connected: ' + str(self.  
request.remote_ip))  
        repo = Repository.Repository()  
        self.write_message(str(repo.get_curren  
t_count())))  
        WebSocketHandler.waiters.add(self)  
  
    def on_close(self):  
        WebSocketHandler.waiters.remove(self)  
  
    @classmethod  
    def send_updates(cls, index):  
        for waiter in cls.waiters:  
            try:  
                waiter.write_message(i
```

```
ndex)
```

```
except:
```

```
    print("Error sending m
```

```
essage")
```



The first thing I do here is setup a collection to store all the connection objects. I'll need this later when I want to send messages to all the clients connected to the service.

```
waiters = set()
```

Then I define what should be done when a client first connects. In this case it prints to console what IP just connected, then **it polls the DB for the current count and sends that to the client** so it can initialize the display. I also add the new connection to the waiters collection.

```
def open(self):  
    self.set_nodelay(True)  
    print('Socket Connected.' + str(self.request))
```

```
remote_ip))  
    repo = Repository.Repository()  
    self.write_message(str(repo.get_current_count()  
)) )  
  
WebSocketHandler.waiters.add(self)
```

Next I define what should be done when the client disconnects it's socket. All that needs to be done here is remove the connection from the waiters collection.

```
def on_close(self):  
    WebSocketHandler.waiters.remove(self)
```

Finally, I define a method to be called later that loops through the waiter collection and transmits the passed in message to each of them.

```
@classmethod
```

```
def send_updates(cls, index):
    for waiter in cls.waiters:
        try:
            waiter.write_message(index)
        except:
            print("Error sending message")
```

ButtonMonitor

This does the same job as the ButtonMonitor class from before, I just dumbed it down to a function. It will still get run in a separate thread. In this function I check for a button press and when it happens the index (current count down value) is decremented by one, the new value is broadcast to all socket connections, and the database is updated with the new value.

```
def ButtonMonitor():
    index = 20
    button = Button.Button(11)
```

```
repo = Repository.Repository()
index = int(repo.get_current_count())
WebSocketHandler.send_updates(str(index))

while 1:
    if button.pressed():
        index -= 1
        if index < 0:
            index = 20
    WebSocketHandler.send_updates(
        str(index))
        repo.set_count(index)
    time.sleep(0.05)
```

Main Function and Globals

Just like with web.py, I setup an application instance which represents the main thread of the Tornado web server. It takes in a collection of instructions, what classes handle what aspects of the server (MainHandler and WebSocketHandler), and where to find resource files on the hard drive. Unlike

web.py, I can define where I want the server to look for certain types of files like having the "static" files, images and such, in a separate folder so I'm going to keep the static directory and tell Tornado to look there. I also tell it where to look for various **favicon** files for desktop browsers and iOS browsers.

```
application = tornado.web.Application([
    (r"/", MainHandler),
    (r'/static/(.*)', tornado.web.StaticFileHandler,
     {'path': '~/yourdir/static'}),
    (r"/(favicon\.ico)", tornado.web.StaticFileHandler,
     {'path': '~/yourdir/static'}),
    (r"/(apple-touch-icon\.png)", tornado.web.StaticFileHandler,
     {'path': '~/yourdir/static'}),
    (r"/(apple-touch-icon-precomposed\.png)", tornado.web.StaticFileHandler,
     {'path': '~/yourdir/static'}),
    (r"/websocket", WebSocketHandler),
])
```

The main function then spins up a thread for the ButtonMonitor and starts the main thread of the Application object that was just instantiated, listening for requests on port 80.

```
if __name__ == "__main__":
    threading.Thread(target=ButtonMonitor).start()
    http_server = tornado.httpserver.HTTPServer(application)
    http_server.listen(80)
    tornado.ioloop.IOLoop.instance().start()
```

Client Side – HTML5, CSS3, & Javascript

The web page received a full overhaul. I enlisted the help of another coworker, Dennis Bamber, our UI designer – you can see his work [here](#). He redid the circular progress bar/pie chart in CSS3 and gave the rest of the page a little better color and style. To change the images displayed now, all I have to do is swap out the class on one of the div tags and change the number displayed in a

second div tag. I'll skip the details of how the CSS works for now, as I have taken the time to fully understand it yet. For now I'll explain the client side that handles communication over the web socket.



Here's the full HTML:

```
<!DOCTYPE HTML>

<head>
    <title>Countdown to 20</title>
    <link href="/static/reset.css" rel="stylesheet" type="text/css" />
    <link href="/static/style.css" rel="stylesheet" type="text/css" />
    <script type="text/javascript" src="/static/jquery-2.0.3.min.js"></script>
    <script type="text/javascript">
        jQuery(document).ready(function () {
            var audioON = 1
            if ("WebSocket" in window) {
```

```
var ws = new WebSocket("ws://10.200.0.  
websocket");  
  
    var messagecount = 0  
  
    ws.onmessage = function(evt) {  
  
        messagecount += 1;  
  
        jQuery(".hold").attr('id', 'pieSlice' +  
evt.data);  
  
        jQuery("#pieNumber").html(evt.data);  
  
        if (messagecount > 1 && audioON == 1) {  
  
            document.getElementById("horn").play  
();  
  
        }  
    }  
  
    ws.onopen = function() {  
  
        messagecount = 0;  
  
    }  
  
    ws.onclose = function() {  
  
        messagecount = 0;  
  
    }  
  
}
```

```
        else {
            alert("WebSocket NOT support by your Browser!");
        }
    
```

```
    jQuery("a#audiobutton").click(function () {
        if (audioON == 0) {
            jQuery(this).text('ON').removeClass('off');
            audioON = 1;
        } else {
            jQuery(this).text('OFF').addClass('off');
            audioON = 0;
        }
        return false;
    });
}) ;

```

```
</script>
</head>
```

```
<html>
  <body>
    <div id="wrapper">
      <h1>Deals till 20 deal goal</h1>
      <div class="pieContainer">
        <div class="pieBackground"></div>
        <div id="pieSlice20" class="hold">
          <div class="pie pie1"></div>
          <div class="pie pie2"></div>
        </div>
        <div id="pieNumber">:(</div>
      </div>
      <div class="avControls">
        <a id="audiobutton" class="on" href="">ON</a>
      </div>
      <p>Copyright © 2013 Overgroup Consulting, LLC. All Rights Reserved.</p>
    </div>
    <audio id="horn" preload="auto">
```

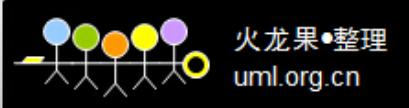
```
<source src="static/airhorn_long.m4a" type="audio/mpeg">  
    <source src="static/airhorn_long.ogg" type="audio/ogg">  
</audio>  
</body>  
</html>
```

Drawing the Pie Graph

This is all done in CSS3. We just have to setup some div tags to draw in:

```
<div class="pieContainer">  
    <div class="pieBackground"></div>  
    <div id="pieSlice20" class="hold">  
        <div class="pie pie1"></div>  
        <div class="pie pie2"></div>  
    </div>
```

```
<div id="pieNumber">:(</div>  
</div>
```



#pieBackground draws a white circle for the background

#pieSliceX does the real work. There's a version of this for every possible count from 0 to 20. It draws a green pie wedge over the white circle, one for each sale made. So #pieSlice20 draws nothing, but #pieSlice19 draws one green wedge, #pieSlice5 draws 15 wedges, and so on.

#pieNumber draws a smaller blue circle in the middle of the green wedges – the layer effect of which causes the green wedges to look more like a circular bar graph reaching around the circumference of the white background circle. The innerHTML of this div tag holds the number of sales we have left to goal. That gets displayed in the center of the blue circle.

WebSocket Connection

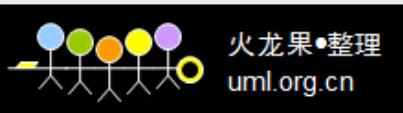
Not all browsers support HTML5, though most do. At the time I wrote this application only Chrome, Safari, and IE9 /10 support it and the new WebSocket

standard. Because of this I first need to confirm that the page is being viewed in a browser capable of a web socket connection. That's what the following code does:

```
if ("WebSocket" in window) {  
    ...  
}  
else {  
    alert("WebSocket NOT support by your Browser!");  
}
```

So if the browser doesn't support web sockets an alert is popped saying as much. Also, the default image in the progress bar is a **frown** face. If the socket connects successfully, the server sends the current count in a message, that replaces the frown face immediately with the current number. If the browser doesn't support websockets, or if there was a problem connecting, the frown face remains and is displayed – just a little humor I guess.

If the browser support sockets then a WebSocket object can be instantiated. we can bind some functions to the various event handlers we'll need to a for.



```
var ws = new WebSocket("ws://10.200.0.253/websocket");
var messagecount = 0

ws.onmessage = function(evt) {
    messagecount += 1;
    jQuery(".hold").attr('id','pieSlice' + evt.data);
    jQuery("#pieNumber").html(evt.data);
    if (messagecount > 1 && audioON == 1) {
        document.getElementById("horn").play();
    }
}

ws.onopen = function() {
    messagecount = 0;
}

ws.onclose = function() {
    messagecount = 0;
```

First I instantiate a WebSocket object, passing in where to access the web socket server. Then I setup a message count variable, this will be used to ensure the air horn sound effect doesn't fire when the page first loads – more on that later. Then I bind the onmessage even function, this is the most important and I'll discuss that in a little more detail next. Last thing is binding functions to the onopen and onclose events, where I just ensure that the messagecount variable is reset to 0.

When a message is received from the web socket server, the onmessage event fires. Thats when the function I bound to it is executed. The socket I setup is very simple and only passes a number, the current count. I take that number and use jQuery to change the class on the pieSlice div tag, there's one corresponding with ever number from 0 to 20 and each one shows a different amount of fill in the outer circle. The number also gets injected into the pieNumber div tag for display in the center of the progress bar. And of course, the air horn. Depending on the audioON variable (toggled by a mute button on the page) the audio tag's play function is triggered as well, sounding the air horn effect.

As I mentioned, there's a mute button. This is a simple 'a' tag – a hyper link tag. I bind a function to it's onclick event to handle changing the icon displayed and the text to indicate if the audio is ON or OFF. Its a pretty simple script that flips the class, kinda like the pieSlice:

```
jQuery("a#audiobutton").click(function () {  
    if (audioON == 0) {  
        jQuery(this).text('ON').removeClass('off');  
        audioON = 1;  
    }  
    else {  
        jQuery(this).text('OFF').addClass('off');  
        audioON = 0;  
    }  
    return false;  
}) ;
```

Once again, I set the websocketserver.py to be executable, then run it. Connect to the RPi via Chrome on another computer and vuala:





◀ ON

Copyright © 2013 Overgroup Consulting, LLC. All Rights Reserved.

Share this:

Like 0

Share 0

Share

Tweet 0

0 Comments

alansellers.net



Sort by Best ▾



Start the discussion...

Be the first to comment.

[Subscribe](#)

[Add Disqus to your site](#)