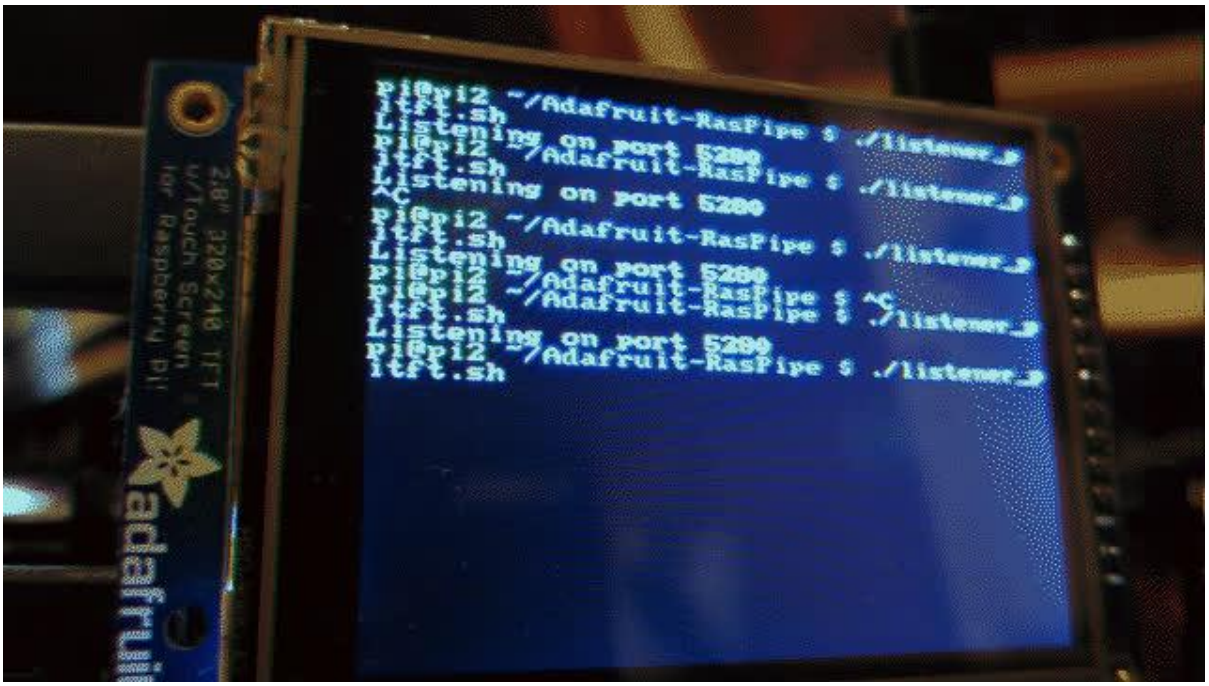
RasPipe: A Raspberry Pi Pipeline Viewer, Part 2

Created by Brennen Bearnes



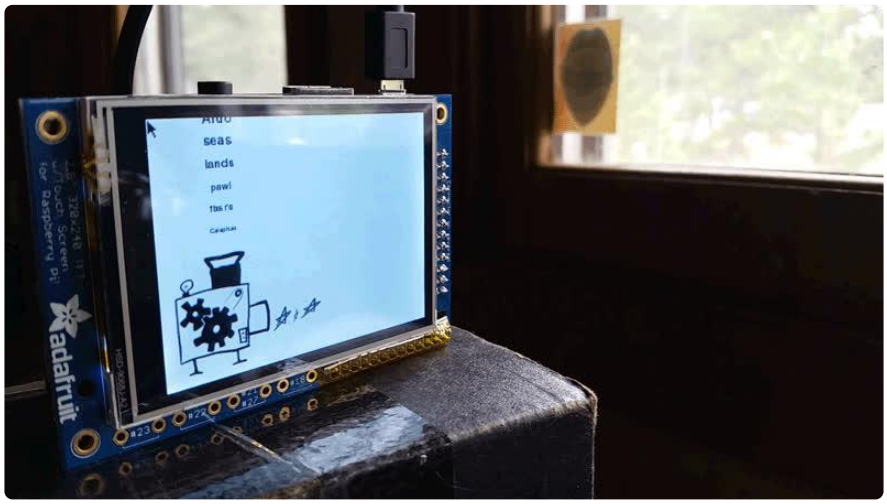
https://learn.adafruit.com/raspipe-a-raspberry-pi-pipeline-viewer-part-2

Last updated on 2023-08-29 02:46:37 PM EDT

Table of Contents

Overview

Previously, in RasPipe: A Raspberry Pi Pipeline Viewer, Part 1 (), we explored writing a visualization of standard streams using Python and the Pygame library to render simple graphics on a PiTFT.



Next, we'll talk about making that visualizer (or other code) available over the network, so that you can easily send it traffic from other computers, including Linux, Windows, OS X, and mobile devices.

We'll use a grab bag of tools to accomplish this:

- netcat - a command-line utility for easily pushing traffic around a network.
- Node.js - a fast, capable server-side JavaScript implementation with a bunch of useful library code available.
- Flask - a fast and easy Python web application framework.

By the end of this guide, you should be equipped with a useful toolkit for bridging the gaps between networked machines for all sorts of projects.

Before we get started, make sure you have a copy of the Adafruit-RasPipe repository ( ) which contains all the code samples for this guide. You can update an existing copy from your terminal like so:

```
cd Adafruit-RasPipe
git pull
```

...or get a fresh one like so:

```
git clone https://github.com/adafruit/Adafruit-RasPipe.git
cd Adafruit-RasPipe
```

Networked Pipes with Netcat

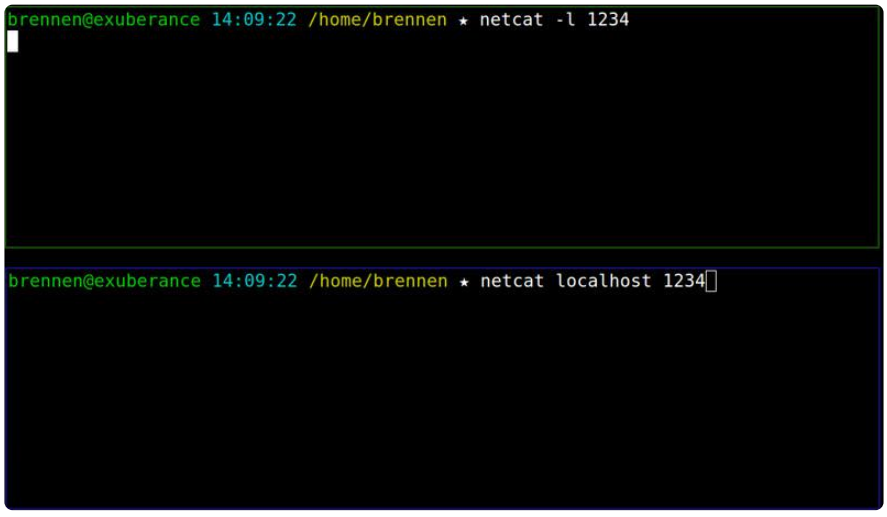
Netcat (), often invoked with the `nc` command, can be thought of as a little bit like a network version of the cat utility (). Its basic purpose is to open TCP/IP connections and pass stuff along them. It can:

- Listen for a network connection on a given port, and echo anything it receives to stdout.
- Connect to a given host and port, and send anything it receives on stdin.

...and much more, but these features are all we need to wire up commands in the shell on a desktop computer to a running instance of raspipe.py on a Raspberry Pi elsewhere on the network.

Transmitting and Receiving

Let's start with a simple example of connecting two `netcat` instances and sending some text between them. Here are two terminals running on my laptop:



In the first terminal, I tell `netcat` to listen on port 1234 by saying:

```
netcat -l 1234
```

In the second terminal, I tell netcat to connect to localhost (which is, as you might suspect, a special hostname that should resolve to the machine we're on right now), port 1234:

```
netcat localhost 1234
```

Now, in either window, I can type some text and hit enter, and it'll show up in the other window. Once I'm done typing, I can hit Ctrl-D, a conventional shortcut for end-of-file, and the connection will close.

You can try this on your Raspberry Pi. If you're running the desktop, just start a couple instances of LXTerm. If you're on the console, try pressing Alt and the right arrow key until you see a new login prompt. You can use Alt+arrows to navigate back and forth between these virtual consoles.

Wiring Netcat to RasPipe

Now take a look at `listener_pitft.sh`:

```
nano listener_pitft.sh
```

```
#!/usr/bin/env bash
echo "Listening on port 5280"
netcat -l 5280 -k | ./raspipe_pitft.sh
```

This script just invokes netcat, listening on port 5280, with `-k`, which says to listen for a new connection once the current one closes. (Normally, the program will exit as soon as one connection finishes.)

It then pipes the output of this netcat instance to `raspipe_pitft.sh`.

In order to test this, you'll need an installed PiTFT display and a working network connection on your Pi. You can check your current network address with `ifconfig`, like so:

```
pi@pi2 ~/Adafruit-RasPipe $ ifconfig
eth0      Link encap:Ethernet  HWaddr b8:27:eb:73:15:91
          inet addr:192.168.1.4  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:148486 errors:0 dropped:0 overruns:0 frame:0
          TX packets:18536 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:27686012 (26.4 MiB)  TX bytes:3096689 (2.9 MiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:2821 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2821 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:381840 (372.8 KiB)  TX bytes:381840 (372.8 KiB)
```

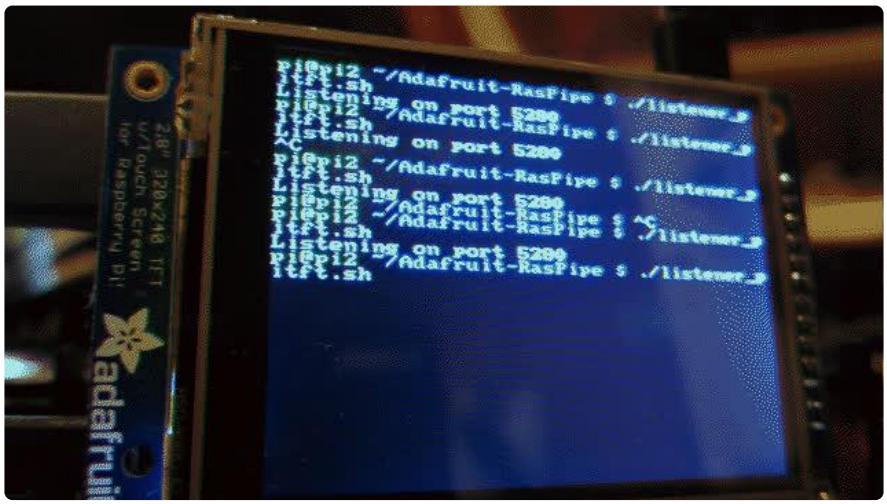
I looked for the `inet addr:192.168.1.4` entry in the `eth0` section. Yours might be under `wlan0`, if you're using a wireless adapter instead of an ethernet cable. Once you know your address, start the listener like so:

```
./listener_pitft.sh
```

Now, from a terminal elsewhere on your network, try sending some traffic to your Pi:

```
echo "hello\nworld\ni\nam\na\pi" | netcat 192.168.1.4 5280
```

If `netcat` isn't available, try typing `nc` instead. You should see something like the following:

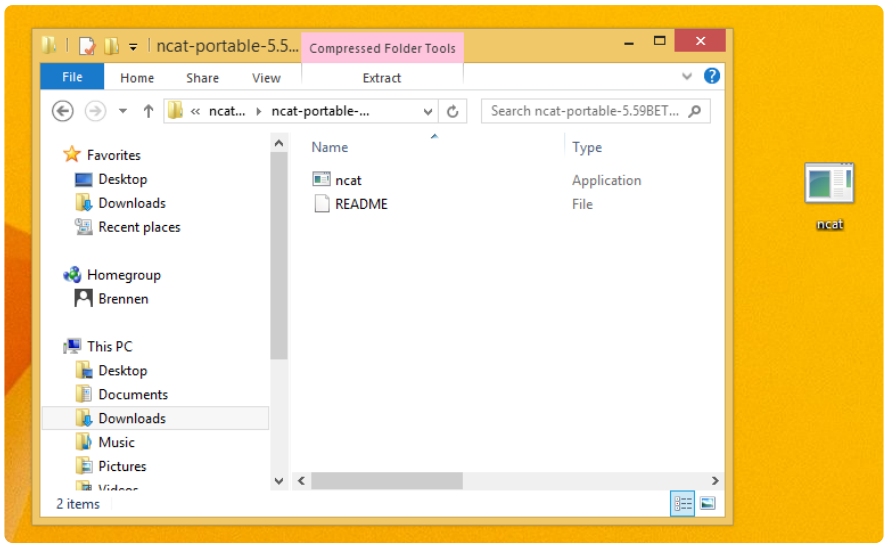


You can experiment with other input - just about anything should work.

Netcat on Windows

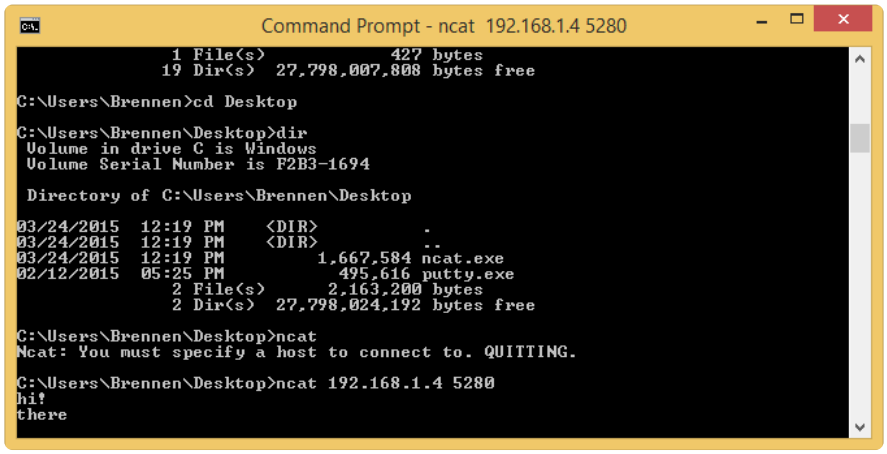
This all works fine if your desktop/laptop is a Linux or OS X machine, but what about Windows users?

Not to worry: The authors of the Nmap network scanning tool provide Ncat (), "much-improved reimplementation of the venerable Netcat", for Windows. To run, just visit the Ncat page () and find the zip file containing the standalone executable. I opened this and dragged a copy of `ncat` to my desktop:



...then just ran `cmd.exe`, followed by:

```
cd Desktop
ncat 192.168.1.4 5280
```



An Inline Utility for Piping to RasPipe

Ok, so my original goal for this project was to make a pipeline viewer that would let me drop a command into the middle of other shell one-liners and have output displayed on the Raspberry Pi.

Using a feature of Bash called process substitution, this turns out to be pretty easy.

```
nano raspipe_tee
```

```
#!/usr/bin/env bash

RASPIPE_ADDY=192.168.1.4
cat /dev/stdin | tee &gt;(netcat $RASPIPE_ADDY 5280)
```

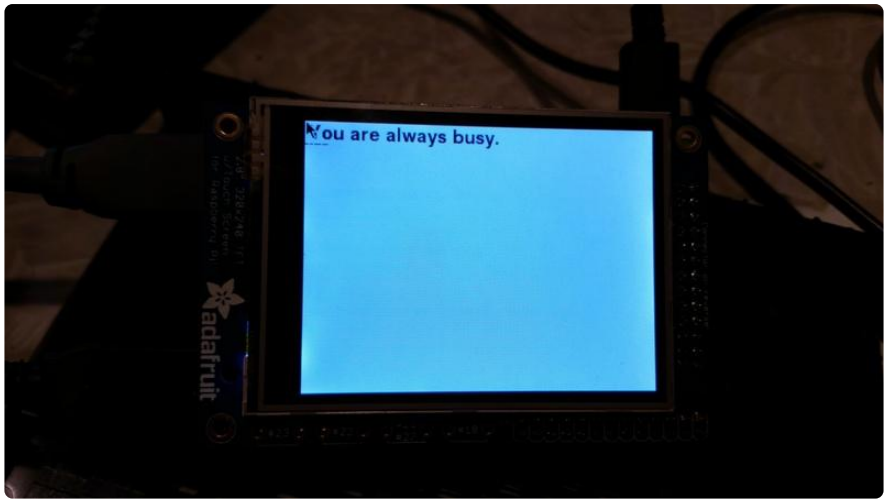
We use `cat /dev/stdin` to spit out the contents of standard input for this script, and pipe it to `tee`, which serves to redirect input to a file while also echoing it back to standard output ().

In `tee >(netcat $RASPIPE_ADDY 5280)`, the `>(...)` section creates a special temporary file of sorts, which `tee` writes to as if it were any other file. The difference is that when it writes to this file, it's actually writing to the stdin of `netcat`. Crazy, right?

So now we can say something like:

```
fortune | ./raspipe_tee
```

...and have the same output echoed on the PiTFT.



What's more, we can issue a command that uses the output of `raspipe_tee` and still see the output of `fortune` echoed on the PiTFT.

```
fortune | ./raspipe_tee | wc -w
```

In order to test this, you can either ssh to your Pi and run `raspipe_tee` there, or copy it to another machine and replace the value of `RASPIPE_ADDY` with the address of your Pi.

A Tiny Network Listener with Node.js

As an interesting alternative to `netcat`, we can write a few lines of node.js.

So what is node? Well, according to the official site:

> Node.js® is a platform built on Chrome's JavaScript runtime () for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

For more detail on what this means, check out our Node.js Embedded Development on the Raspberry Pi ().

Installing Node.js

In order to proceed, we'll need a working install of node. For that, you can follow the installation instructions () from our node.js guide. Once that's done, come back here and proceed.

Streaming From Network to Standard Output: listener.js

Next, have a look at some code in the Adafruit-RasPipe repository we cloned last time around ().

```
cd Adafruit-RasPipe
nano listener.js
```

```
#!/usr/bin/env node

var net = require('net');

var onConnection = function (socket) {
  socket.pipe(process.stdout);
};

net.createServer(onConnection).listen(5280);
console.log('Server running at 127.0.0.1:5280');
```

This creates a simple network server which listens for connections on port 5280. When a connection is established, the function in `onConnection` runs with a new stream contained in the `socket` variable. We then tell `socket` to pipe its input to `process.stdout`, which as you might guess is the standard output for the current process.

You can test this by running:

```
./listener.js | raspipe_pitft.sh
```

And sending output via netcat or

Why node.js? Again, check out the guide ():

> Why is it worth exploring node.js () development in an embedded environment? JavaScript is a widely known language that was designed to deal with user interaction in a browser. Thanks to node.js, we can now use JavaScript outside of the browser, and even interact with hardware on devices such as the Raspberry Pi.

Tools like `netcat` are sufficient for lots of simple cases, but if you want to build a full-fledged network application in a robust programming language with lots of libraries available, node can be a great starting point.

A Web Listener in Python with Flask

Flask is a "microframework" for writing web applications in Python. The example Flask application looks like this:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':
    app.run()
```

Run that, and you'll get a web server on your local machine which returns "Hello world!" when you visit `http://localhost:5000 ()` in your browser.

We'll do something a little more interesting: A simple web app that controls an instance of RasPipe.

Installing Flask

The Flask documentation on installation () is good, and you should probably follow it if you're planning to do a Flask project on a robust desktop system. Unfortunately, it's a bit heavy for the Raspberry Pi.

First, make sure you have pip, the recommended utility for installing Python packages.

```
sudo apt-get install python-pip
```

Next, do a system-wide install of Flask:

```
sudo pip install Flask
```

This should be all that's required, although you might see error messages to the following effect:

```
================================================================================
WARNING: The C extension could not be compiled, speedups are not enabled.
Plain-Python installation succeeded.
================================================================================
```

...don't worry about these.

flask_listener.py

Open up `flask_listener.py` and have a look. It's a pretty short program:

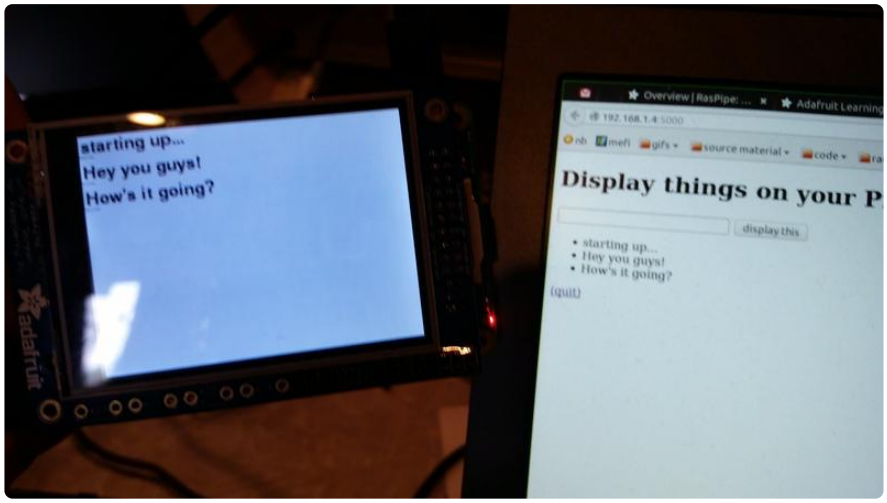
```
nano flask_listener.py
```

```
#!/usr/bin/env python

from flask import Flask
from flask import request
from flask import render_template
from flask import redirect, url_for

from raspipe import RasPipe

app = Flask(__name__)

rp = RasPipe(None)
rp.input_lines.append('starting up...')
rp.render_frame()

@app.route('/')
def index():
    return render_template('index.html', rp=rp)

@app.route('/display', methods=['POST'])
def display():
    rp.input_lines.append(request.form['line'])
    rp.render_frame()
    return redirect(url_for('index'))

@app.route('/quit')
def quit():
    func = request.environ.get('werkzeug.server.shutdown')
    func()
    return "Quitting..."

if __name__ == '__main__':
    # app.debug = True
    app.run(host='0.0.0.0')
```

You should now be able to run this from the console of your Pi with:

```
export SDL_FBDEV=/dev/fb1
./flask_listener.py
```

And visit `http://[YOUR PI'S ADDRESS HERE]:5000/` in a web browser. For example, I did `http://192.168.1.4:5000/`:



Let's look at what's going on here in more detail.

Line by Line

```
from flask import Flask
from flask import request
from flask import render_template
from flask import redirect, url_for

from raspipe import RasPipe
```

The first important thing to notice here is a set of `import`s for parts of the Flask framework. `request` is used to model different parts of the request sent by a browser (or other client). `render_template()` is a function we'll call to turn a

template file into some HTML (more about that in a bit). `redirect` and `url_for` will be used to have Flask send a client off to a different web page.

Next, we import the `RasPipe` class defined in `raspipe.py`. Python knows where to look for this because it's in the same directory as `flask_listener.py`.

```
app = Flask(__name__)

rp = RasPipe(None)
rp.input_lines.append('starting up...')
rp.render_frame()
```

This assigns an instance of the `Flask` class to `app`, and an instance of the `RasPipe` class to `rp`.

Next, we give `rp` a line of input ("starting up..."), and ask it to display a single frame of the animation with `rp.render_frame()`. By now, the PiTFT should be showing that line of text.

```
@app.route('/')
def index():
    return render_template('index.html', rp=rp)
```

This tells Flask that a URL, `/`, should return the result of calling the function `index()`.

Running `render_template('index.html', rp=rp)` actually does quite a bit of magic. It:

1. Looks in the `templates/` directory for a file called `index.html`
2. Loads this file as a Jinja template ()
3. Passes the `RasPipe` instance in `rp` into the template
4. Turns the template into HTML

To see what the home page of our application is made of, you can open `templates/index.html` with Nano:

```
nano templates/index.html
```

```
&lt;!DOCTYPE html&gt;
&lt;html lang="en"&gt;
&lt;head&gt;
  &lt;meta charset="utf-8"&gt;
  &lt;title&gt;RasPipe&lt;/title&gt;
&lt;/head&gt;

&lt;body&gt;

  &lt;h1&gt;Display things on your PiTFT!&lt;/h1&gt;
```

```
  &lt;form action="display" method="post"&gt;
    &lt;input name="line"&gt;
    &lt;button&gt;display this&lt;/button&gt;
  &lt;/form&gt;

  &lt;ul&gt;
  {% for line in rp.input_lines %}
    &lt;li&gt;{{ line }}&lt;/li&gt;
  {% endfor %}
  &lt;/ul&gt;

  &lt;p&gt;&lt;a href="/quit"&gt;(quit)&lt;/a&gt;&lt;/p&gt;

&lt;/body&gt;
&lt;/html&gt;
```

Most of this is just a simple HTML document, which is a bit out of the scope of this guide to explain in detail. Right now, note that things inside `{% %}` brackets, like `{% for line in rp.input_lines %}`, are directives to the template engine, while things inside `{{ }}` brackets are the names of variables to include.

Back to `flask_listener.py`:

```
@app.route('/display', methods=['POST'])
def display():
    rp.input_lines.append(request.form['line'])
    rp.render_frame()
    return redirect(url_for('index'))
```

This creates a URL, `/display`, which we can use to send lines of text to the display. Remember the little input form on the front page? It's defined in the template like so:

```
  &lt;form action="display" method="post"&gt;
    &lt;input name="line"&gt;
    &lt;button&gt;display this&lt;/button&gt;
  &lt;/form&gt;
```

This tells the web browser that when "display this" is pressed, we want to send whatever's in the input to `display` as an HTTP POST request (). Inside our function, the contents of the form are available in `request.form`, so we append the line to the end of `rp.input_lines`, and render another frame of animation so that it'll show up right away.

Finally, we return the result of `redirect(url_for('index'))` to let Flask know that it should send the client back to the home page.

```
@app.route('/quit')
def quit():
    func = request.environ.get('werkzeug.server.shutdown')
    func()
    return "Quitting..."
```

Here, we define a URL, `/quit`, we can use to quit the program. You obviously wouldn't want to expose this in a public-facing web application, but here it's useful.

```
if __name__ == '__main__':
    # app.debug = True
    app.run(host='0.0.0.0')
```

With `app.run(host='0.0.0.0')`, we tell Flask it should start up and act like a web server. The `host='0.0.0.0'` bit is important because otherwise it'll only listen for local connections on 127.0.0.1, and won't be visible to the rest of the network.

If you're running into weird behavior, uncomment `app.debug(True)` to get nicely-formatted stack traces in the browser when something breaks. (Be careful with this - it can allow a remote user to execute arbitrary code, so you definitely don't want to leave it running in debug mode on an untrusted network.)

Testing with cURL

The web browser is all well and good, but what if you want to talk to your shiny new web application from the command line?

`curl` is a command-line client for talking to things that have URLs.

Normally, when you give `curl` a URL, it just fetches whatever's at that URL and prints it to stdout.

With the `--data="..."` option, however, it'll act like your browser does when you submit a form. Try something like the following with your Pi's address:

```
curl --data "line=hello there" 'http://192.168.1.4:5000/display'
```

On Security

It's worth taking a moment to note that nearly everything I've discussed so far in this guide is probably really insecure.

Of course, this might not actually matter a whole lot.

For my purposes, it's not too important whether anyone else on my network can see the things I'm sending to my Raspberry Pi's display. On the other hand, there are a lot of situations in which it just might matter.

Consider the following:

- The code in this guide is almost certainly vulnerable to attacks based on sending it weird input, and opening it up to arbitrary input from the network makes this much more likely to happen.
- The raw network sockets we use in the netcat and node examples here are not only unencrypted, they're unauthenticated - you don't know who's sending you traffic, and they aren't limited to clients with a specific set of credentials.
- The same is true of unencrypted HTTP requests.
- Text sent in the clear can probably be intercepted by other users of the network, or by whoever controls the hardware that runs the network. On your home router, this might not matter very much. Everywhere else, you should be careful.

So what can you do about this? Well, if you're considering using any of the techniques here for a production project that handles sensitive data, think about the following:

- Use HTTPS for web interfaces
- Use a robust data-logging protocol with built-in authentication and encryption mechanisms
- Tunnel traffic over SSH

Piping through SSH

The first two of these recommendations are out of scope for this guide, but the third is pretty straightforward for simple cases. Consider a version of `listener_pitft.sh` which restricts netcat to listening on localhost:

```
      #!/usr/bin/env bash
echo "Listening on port 5280"
netcat -l 127.0.0.1 5280 -k | ./raspipe_pitft.sh
```

If you drop this in a file called `listener_local_pitft.sh`, you can run it on the Pi with:

```
chmod +x listener_local_pitft.sh
./listener_local_pitft.sh
```

Now from a terminal on another machine, we'll use a couple of really handy SSH features you may not have seen before:

```
echo 'a secret message' | ssh pi@192.168.1.4 'netcat localhost 5280'
```

First, if you give `ssh` a command after the hostname, like `'netcat localhost 5280'`, it'll execute that command on the remote server.

Second, if you pipe the output of another command into `ssh`, it'll pass that output to the stdin of whatever command it runs. (Careful here - if you don't give it a command, it'll execute the input in Bash.)

Create an SSH Tunnel

As an alternative, you can use SSH to create an encrypted tunnel for traffic from utilities like `netcat`.

First, connect from your client machine via SSH, using the `-L` option to specify a port on the local host to be forwarded to a port on the remote host. (This can be a bit confusing: In `5280:localhost:5280` here, the `localhost:5280` part is localhost fr om the context of the Raspberry Pi.)

```
ssh -L 5280:localhost:5280 pi@192.168.1.4
```

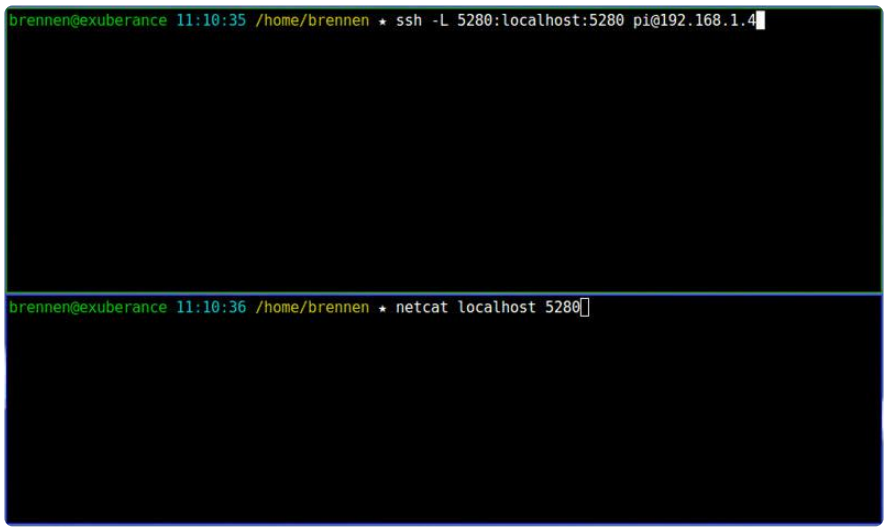
Next, on the Pi, start `netcat` to listen for local connections:

```
netcat -l 127.0.0.1 5280 -k
```

Finally, back on your client machine, point netcat at port 5280 and try typing something:

```
netcat localhost 5280
```

If all has gone well, the traffic should be forwarded by SSH from port 5280 on the client machine to port 5280 on the Raspberry Pi. Here's an example in two terminals:



This technique is widely useful for all sorts of network services, especially if you're dealing with more restricted networks that you don't, yourself, control.

Further Reading

Now that you've had a taste of ways to make your Pi project networked, here's how you can learn more about the technique's we've tried:

On Netcat, Network Sockets, and SSH Tunneling

- Netcat: the TCP/IP swiss army ()
- Wikipedia: Network socket ()
- What are SSH tunnels and how to use them ()

On Node.js

- Node.js Embedded Development on the Raspberry Pi ()
- Using OSC to Communicate with a Raspberry Pi ()
- Adafruit IO Client Library for Node.js ()

On Flask

- Installation ()
- Quickstart () - a solid intro to basic concepts and solutions
- Tutorial () - a walkthrough on creating a full-featured Flask application
- The Flask Mega-Tutorial ()