



TUTORIAL

How To Develop a Node.js TCP Server Application using PM2 and Nginx on Ubuntu 16.04

Nginx Node.js Ubuntu 16.04 JavaScript

By [Kunal Relan](#)

Posted July 23, 2018 49.6k

The author selected [OSMI](#) to receive a donation as part of the [Write for DONations](#) program.

Introduction

[Node.js](#) is a popular open-source JavaScript runtime environment built on Chrome's V8 Javascript engine. Node.js is used for building server-side and networking applications. *TCP (Transmission Control Protocol)* is a networking protocol that provides reliable, ordered and error-checked delivery of a stream of data between applications. A TCP server can accept a TCP connection request, and once the connection is established both sides can exchange data streams.

In this tutorial, you'll build a basic Node.js TCP server, along with a client to test the server. You'll run your server as a background process using a powerful Node.js process manager called [PM2](#). Then you'll configure [Nginx](#) as a reverse proxy for the TCP application and test the client-server connection from your local machine.

Prerequisites

To complete this tutorial, you will need:

- One Ubuntu 16.04 server set up by following [the Ubuntu 16.04 initial server setup guide](#), including a sudo non-root user and a firewall.
- Nginx installed on your server, as shown in [How To Install Nginx on Ubuntu 16.04](#). Nginx must be compiled with the `--with-` [SCROLL TO TOP](#) which is the default on a fresh

installation of Nginx through the `apt` package manager on Ubuntu 16.04.

- Node.js installed using the official PPA, as explained in [How To Install Node.js on Ubuntu 16.04.](#)

Step 1 – Creating a Node.js TCP Application

We will write a Node.js application using TCP Sockets. This is a sample application which will help you understand the [Net](#) library in Node.js which enables us to create raw TCP server and client applications.

To begin, create a directory on your server in which you would like to place your Node.js application. For this tutorial, we will create our application in the `~/tcp-nodejs-app` directory:

```
$ mkdir ~/tcp-nodejs-app
```

Then switch to the new directory:

```
$ cd ~/tcp-nodejs-app
```

Create a new file named `package.json` for your project. This file lists the packages that the application depends on. Creating this file will make the build reproducible as it will be easier to share this list of dependencies with other developers:

```
$ nano package.json
```

You can also generate the `package.json` using the `npm init` command, which will prompt you for the details of the application, but we'll still have to manually alter the file to add additional pieces, including a startup command. Therefore, we'll manually create the file in this tutorial.

Add the following JSON to the file, which specifies the application's name, version, the main file, the command to start the application, and the software license:

```
package.json
```

SCROLL TO TOP

```
{
  "name": "tcp-nodejs-app",
  "version": "1.0.0",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "license": "MIT"
}
```

The `scripts` field lets you define commands for your application. The setting you specified here lets you run the app by running `npm start` instead of running `node server.js`.

The `package.json` file can also contain a list of runtime and development dependencies, but we won't have any third party dependencies for this application.

Now that you have the project directory and `package.json` set up, let's create the server.

In your application directory, create a `server.js` file:

```
$ nano server.js
```

Node.js provides a module called `net` which enables TCP server and client communication. Load the `net` module with `require()`, then define variables to hold the port and host for the server:

server.js

```
const net = require('net');
const port = 7070;
const host = '127.0.0.1';
```

We'll use port `7070` for this app, but you can use any available port you'd like. We're using `127.0.0.1` for the `HOST` which ensures that our server is only listening on our local network interface. Later we will place Nginx in front of this app as a reverse proxy. Nginx is well-versed at handling multiple connections and horizontal scaling.

Then add this code to spawn a TCP server using the `createServer()` function from the `net` module. Then start listening for connections on the port and host you defined by using the `listen()` function of the `net` module. [SCROLL TO TOP](#)

server.js

```
...  
const server = net.createServer();  
server.listen(port, host, () => {  
  console.log('TCP Server is running on port ' + port + '.');  
});
```

Save `server.js` and start the server:

```
$ npm start
```

You'll see this output:

Output

```
TCP Server is running on port 7070
```

The TCP server is running on port `7070`. Press `CTRL+C` to stop the server.

Now that we know the server is listening, let's write the code to handle client connections.

When a client connects to the server, the server triggers a `connection` event, which we'll observe. We'll define an array of connected clients, which we'll call `sockets`, and add each client instance to this array when the client connects.

We'll use the `data` event to process the data stream from the connected clients, using the `sockets` array to broadcast data to all the connected clients.

Add this code to the `server.js` file to implement these features:

server.js

```
...  
  
let sockets = [];  
  
server.on('connection', function(sock) {  
  console.log('CONNECTED: ' + sock.remoteAddress + ':' + sock.remotePort);  
  sockets.push(sock);  
  
  sock.on('data', function(data) SCROLL TO TOP  
    console.log('DATA ' + sock.remoteAddress + ': ' + data);
```

```

        // Write the data back to all the connected, the client will receive it as data +
        sockets.forEach(function(sock, index, array) {
            sock.write(sock.remoteAddress + ':' + sock.remotePort + " said " + data + '\n'
        });
    });
});

```

This tells the server to listen to `data` events sent by connected clients. When the connected clients send any data to the server, we echo it back to all the connected clients by iterating through the `sockets` array.

Then add a handler for `close` events which will be triggered when a connected client terminates the connection. Whenever a client disconnects, we want to remove the client from the `sockets` array so we no longer broadcast to it. Add this code at the end of the connection block:

server.js

```

let sockets = [];
server.on('connection', function(sock) {

    ...

    // Add a 'close' event handler to this instance of socket
    sock.on('close', function(data) {
        let index = sockets.findIndex(function(o) {
            return o.remoteAddress === sock.remoteAddress && o.remotePort === sock.remotePort;
        });
        if (index !== -1) sockets.splice(index, 1);
        console.log('CLOSED: ' + sock.remoteAddress + ' ' + sock.remotePort);
    });
});

```

Here is the complete code for `server.js`:

server.js

```

const net = require('net');
const port = 7070;
const host = '127.0.0.1';

const server = net.createServer();
server.listen(port, host, () => {
    console.log('TCP Server is running on port ' + port + '.');
});

```

SCROLL TO TOP

```

let sockets = [];

server.on('connection', function(sock) {
  console.log('CONNECTED: ' + sock.remoteAddress + ':' + sock.remotePort);
  sockets.push(sock);

  sock.on('data', function(data) {
    console.log('DATA ' + sock.remoteAddress + ': ' + data);
    // Write the data back to all the connected, the client will receive it as data + 1
    sockets.forEach(function(sock, index, array) {
      sock.write(sock.remoteAddress + ':' + sock.remotePort + " said " + data + '\n');
    });
  });

  // Add a 'close' event handler to this instance of socket
  sock.on('close', function(data) {
    let index = sockets.findIndex(function(o) {
      return o.remoteAddress === sock.remoteAddress && o.remotePort === sock.remotePort;
    });
    if (index !== -1) sockets.splice(index, 1);
    console.log('CLOSED: ' + sock.remoteAddress + ' ' + sock.remotePort);
  });
});

```

Save the file and then start the server again:

```
$ npm start
```

We have a fully functional TCP Server running on our machine. Next we'll write a client to connect to our server.

Step 2 — Creating a Node.js TCP Client

Our Node.js TCP Server is running, so let's create a TCP Client to connect to the server and test the server out.

The Node.js server you just wrote is still running, blocking your current terminal session. We want to keep that running as we develop the client, so open a new Terminal window or tab. Then connect into the server again from the new tab.

```
$ ssh sammy@your_server_ip
```

SCROLL TO TOP

Once connected, navigate to the `tcp-nodejs-app` directory:

```
$ cd tcp-nodejs-app
```

In the same directory, create a new file called `client.js`:

```
$ nano client.js
```

The client will use the same `net` library used in the `server.js` file to connect to the TCP server. Add this code to the file to connect to the server using the IP address `127.0.0.1` on port `7070`:

client.js

```
const net = require('net');
const client = new net.Socket();
const port = 7070;
const host = '127.0.0.1';

client.connect(port, host, function() {
  console.log('Connected');
  client.write("Hello From Client " + client.address().address);
});
```

This code will first try to connect to the TCP server to ensure that the server we created is running. Once the connection is established, the client will send "Hello From Client " + `client.address().address` to the server using the `client.write` function. Our server will receive this data and echo it back to the client.

Once the client receives the data back from the server, we want it to print the server's response. Add this code to catch the `data` event and print the server's response to the command line:

client.js

```
client.on('data', function(data) {
  console.log('Server Says : ' + data);
});
```

Finally, handle disconnections from the server gracefully by adding this code:

client.js

SCROLL TO TOP

```
client.on('close', function() {  
    console.log('Connection closed');  
});
```

Save the `client.js` file.

Run the following command to start the client:

```
$ node client.js
```

The connection will establish and the server will receive the data, echoing it back to the client:

`client.js` Output

Connected

Server Says : 127.0.0.1:34548 said Hello From Client 127.0.0.1

Switch back to the terminal where the server is running, and you'll see the following output:

`server.js` Output

CONNECTED: 127.0.0.1:34550

DATA 127.0.0.1: Hello From Client 127.0.0.1

You have verified that you can establish a TCP connection between your server and client apps.

Press `CTRL+C` to stop the server. Then switch to the other terminal session and press `CTRL+C` to stop the client. You can now disconnect this terminal session from your server and return to your original terminal session.

In the next step we'll launch the server with PM2 and run it in the background.

Step 3 – Running the Server with PM2

You have a working server that accepts client connections, but it runs in the foreground. Let's run the server using PM2 so it `SCROLL TO TOP` and can restart gracefully.

First, install PM2 on your server globally using `npm`:

```
$ sudo npm install pm2 -g
```

Once PM2 is installed, use it to run your server. Instead of running `npm start` to start the server, you'll use the `pm2` command. Start the server:

```
$ pm2 start server.js
```

You'll see output like this:

```
[secondary_label Output  
[PM2] Spawning PM2 daemon with pm2_home=/home/sammy/.pm2  
[PM2] PM2 Successfully daemonized  
[PM2] Starting /home/sammy/tcp-nodejs-app/server.js in fork_mode (1 instance)  
[PM2] Done.
```

Name	mode	status	U	cpu	memory
server	fork	online	0	5%	24.8 MB

Use ``pm2 show <id|name>`` to get more details about an app

The server is now running in the background. However, if we reboot the machine, it won't be running anymore, so let's create a `systemd` service for it.

Run the following command to generate and install PM2's `systemd` startup scripts. Be sure to run this with `sudo` so the `systemd` files install automatically.

```
$ sudo pm2 startup
```

You'll see this output:

```
Output  
[PM2] Init System found: systemd  
Platform systemd
```

```
[PM2] Writing init configuration in /etc/systemd/system/pm2-root.service
[PM2] Making script booting at startup...
[PM2] [-] Executing: systemctl enable pm2-root...
Created symlink from /etc/systemd/system/multi-user.target.wants/pm2-root.service to /etc/s
[PM2] [v] Command successfully executed.
+-----+
[PM2] Freeze a process list on reboot via:
$ pm2 save

[PM2] Remove init script via:
$ pm2 unstartup systemd
```

PM2 is now running as a systemd service.

You can list all the processes PM2 is managing with the `pm2 list` command:

```
$ pm2 list
```

You'll see your application in the list, with the ID of `0`:

Output

App name	id	mode	pid	status	restart	uptime	cpu	mem	user	watch
server	0	fork	9075	online	0	4m	0%	30.5 MB	sammy	disa

In the preceding output, you'll notice that `watching` is disabled. This is a feature that reloads the server when you make a change to any of the application files. It's useful in development, but we don't need that feature in production.

To get more info about any of the running processes, use the `pm2 show` command, followed by its ID. In this case, the ID is `0`:

```
$ pm2 show 0
```

This output shows the uptime, status, and other info about the running application:

SCROLL TO TOP

Output

Describing process with id 0 - name server

status	online
name	server
restarts	0
uptime	7m
script path	/home/sammy/tcp-nodejs-app/server.js
script args	N/A
error log path	/home/sammy/.pm2/logs/server-error-0.log
out log path	/home/sammy/.pm2/logs/server-out-0.log
pid path	/home/sammy/.pm2/pids/server-0.pid
interpreter	node
interpreter args	N/A
script id	0
exec cwd	/home/sammy/tcp-nodejs-app
exec mode	fork_mode
node.js version	8.11.2
watch & reload	X
unstable restarts	0
created at	2018-05-30T19:29:45.765Z

Code metrics value

Loop delay	1.12ms
Active requests	0
Active handles	3

Add your own code metrics: <http://bit.ly/code-metrics>

Use ``pm2 logs server [--lines 1000]`` to display logs

Use ``pm2 monit`` to monitor CPU and Memory usage server

If the application status shows an error, you can use the **error log path** to open and review the error log to debug the error:

```
$ cat /home/tcp/.pm2/logs/server-error-0.log
```

If you make changes to the server code, you'll need to restart the application's process to apply the changes, like this:

```
$ pm2 restart 0
```

SCROLL TO TOP

PM2 is now managing the application. Now we'll use Nginx to proxy requests to the server.

Step 4 — Set Up Nginx as a Reverse Proxy Server

Your application is running and listening on `127.0.0.1`, which means it will only accept connections from the local machine. We will set up Nginx as a reverse proxy which will handle incoming traffic and direct it to our server.

To do this, we'll modify the Nginx configuration to use the `stream {}` and `stream proxy` features of Nginx to forward TCP connections to our Node.js server.

We have to edit the main Nginx configuration file as the `stream` block that configures TCP connection forwarding only works as a top-level block. The default Nginx configuration on Ubuntu loads server blocks within the `http` block of the file, and the `stream` block can't be placed within that block.

Open the file `/etc/nginx/nginx.conf` in your editor:

```
$ sudo nano /etc/nginx/nginx.conf
```

Add the following lines at the end of your configuration file:

`/etc/nginx/nginx.conf`

```
...

stream {
    server {
        listen 3000;
        proxy_pass 127.0.0.1:7070;
        proxy_protocol on;
    }
}
```

This listens for TCP connections on port `3000` and proxies the requests to your Node.js server running on port `7070`. If your application is set to listen on a different port, update the proxy pass URL port to the correct port number. The `proxy_protocol` directive tells Nginx to use the PROXY protocol to send client information to backend servers, which can then process that information a. [SCROLL TO TOP](#)

Save the file and exit the editor.

Check your Nginx configuration to ensure you didn't introduce any syntax errors:

```
$ sudo nginx -t
```

Next, restart Nginx to enable the TCP and UDP proxy functionality:

```
$ sudo systemctl restart nginx
```

Next, allow TCP connections to our server on that port. Use `ufw` to allow connections on port 3000:

```
$ sudo sudo ufw allow 3000
```

Assuming that your Node.js application is running, and your application and Nginx configurations are correct, you should now be able to access your application via the Nginx reverse proxy.

Step 5 — Testing the Client-Server Connection

Let's test the server out by connecting to the TCP server from our local machine using the `client.js` script. To do so, you'll need to download the `client.js` file you developed to your local machine and change the port and IP address in the script.

First, on your local machine, download the `client.js` file using `scp`:

```
$ [environment local  
$ scp sammy@your_server_ip:~/tcp-nodejs-app/client.js client.js
```

Open the `client.js` file in your editor:

```
$ [environment local  
$ nano client.js
```

Change the port to 3000 and change [SCROLL TO TOP](#) your server's IP address:

client.js

```
// A Client Example to connect to the Node.js TCP Server
const net = require('net');
const client = new net.Socket();
const port = 3000;
const host = 'your_server_ip';
...
```

Save the file, exit the editor, and test things out by running the client:

```
$ node client.js
```

You'll see the same output you saw when you ran it before, indicating that your client machine has connected through Nginx and reached your server:

client.js Output

Connected

```
Server Says : 127.0.0.1:34584 said PROXY TCP4 your_local_ip_address your_server_ip 52920 :
Hello From Client your_local_ip_address
```

Since Nginx is proxying client connections to your server, your Node.js server won't see the real IP addresses of the clients; it will only see Nginx's IP address. Nginx doesn't support sending the real IP address to the backend directly without making some changes to your system that could impact security, but since we enabled the PROXY protocol in Nginx, the Node.js server is now receiving an additional PROXY message that contains the real IP. If you need that IP address, you can adapt your server to process PROXY requests and parse out the data you need.

You now have your Node.js TCP application running behind an Nginx reverse proxy and can continue to develop your server further.

Conclusion

In this tutorial you created a TCP application with Node.js, ran it with PM2, and served it behind Nginx. You also created a client application to connect to it from other machines.

SCROLL TO TOP

You can use this application to handle large chunks of data streams or to build real-time messaging applications.

Was this helpful?

Yes

No



[Report an issue](#)

About the authors



Kunal Relan

Full Stack Developer, Information Security Researcher and Author of iOS Penetration Testing. Currently based in New Delhi.



Brian Hogan

Editor

Related

TUTORIAL

How To Launch Child Processes in Node.js

Since Node.js instances create a single process with a single thread, JavaScript operations that take a long time to run can

TUTORIAL

Understanding Arrow Functions in JavaScript

Arrow functions are a new way to write anonymous function expressions in JavaScript, and are similar to lambda functions in

TUTORIAL

How To Build a Discord

SCROLL TO TOP

TUTORIAL

How To Share State

Bot with Node.js

Discord is a chat application that allows millions of users across the globe to message and voice chat online in

Across React Components with Context

In this tutorial, you'll share state across multiple components using React ...

Still looking for an answer?



Ask a question



Search for more help

Comments

13 Comments

Leave a comment...

Sign In to Comment



markjgsmith August 17, 2018

Very similar to a setup I've been building for the past few months on DO. Could you expand a little on how to read the PROXY details from the app? I never got that part to work.

[Reply](#) [Report](#)

 [kunalrelan](#) August 17, 2018

0 Hey, Can you explain more on the “read the PROXY details from the app”, didn’t understand that part much.

[Reply](#) [Report](#)

 [markjgsmith](#) August 17, 2018

0 I’m referring to the second to last paragraph in the article:

since we enabled the PROXY protocol in Nginx, the Node.js server is now receiving an additional PROXY message that contains the real IP. If you need that IP address, you can adapt your server to process PROXY requests and parse out the data you need

[Reply](#) [Report](#)

 [markjgsmith](#) September 16, 2018

0 Kunalrelan - It would be really great if you could say how to “adapt the server to process PROXY requests and parse out the data” (quote taken from the article). Without this data, all requests appear to originate from the same location, which is a severe limitation to the entire setup.

[Reply](#) [Report](#)

 [letian1997](#) March 14, 2019

0 Face the same issue too. After some research, I find the solution to get PROXY details (real ip address) is to use a library, [findhit-proxywrap](#).

Have written a blog post on the solution, feel free to check it out:

[How to get real ip address in Node.js TCP server behind Nginx proxy](#)

[Reply](#) [Report](#)

 [afern247](#) September 2, 2018

1 status | errored

I followed what you did.

```
Error: listen EADDRINUSE 127.0.0.1:7070
    at Object.exports._errnoEx (node:internal/streams/worker:120:11)
    at exports._exceptionWithHostPort (util.js:1043:20)
```

```
at Server._listen2 (net.js:1271:14)
at listen (net.js:1307:10)
at net.js:1417:9
at args.(anonymous function) (/usr/lib/node_modules/pm2/node_modules/event-loop
at _combinedTickCallback (internal/process/next_tick.js:83:11)
at process._tickDomainCallback (internal/process/next_tick.js:128:9)
at Module.runMain (module.js:613:11)
at run (bootstrap_node.js:394:7)
```

I don't have bootstrap installed...

[Reply](#) [Report](#)

 [kunalrelan](#) September 2, 2018

 Hi,

This was because of some other service running on port 7070.

[Reply](#) [Report](#)

 [55altaha4waterOe947b94aeae](#) September 16, 2018

 انا لاعلم


[Reply](#) [Report](#)

 [55altaha4waterOe947b94aeae](#) September 16, 2018

 الرجاء المساعد وشكرا

[Reply](#) [Report](#)

 [55altaha4waterOe947b94aeae](#) September 16, 2018

 عالم التحلية والضباب والرذاذ
ياريت تساعد نى على تصريحات الاخطاء
أنا موقعك زهنين
الرجاء المساعد وشكرا

[Reply](#) [Report](#)

 [zzjlamb](#) July 20, 2019

 Thanks for a very useful tutorial.

There is a minor bug in your code. You have used a variable name, "sock", in the forEach iterator of your callback function for data received, which is already in use in that scope. The handler is therefore broadcasting the address and port of each socket back to itself in the "said" message, rather than the details of the socket from which the message came.

Modifying the callback function like this works:

SCROLL TO TOP

```
sockets.forEach(function(aSock, index, array) {  
  aSock.write(sock.remoteAddress + ':' + sock.remotePort + " said " + data + '\n');  
}
```

[Reply](#) [Report](#)

^ [imdset](#) December 8, 2019

0 is there any specific reason that you are not listening on main IP of server, and you are listening on a local port and making a proxy for that?

I have some issues with my app, I wonder that my problem is for working directly to server IP :/

[Reply](#) [Report](#)

^ [arnaud.gaboury](#) January 22, 2020

0 First thank yo so much for this tuto. A small change:
at the end of STEP2, you write: *Then switch to the other terminal session and press CTRL+C to stop the client.* But his is not necessary, as right before we stopped the server, which mechanically makes an end to our ssh session. We can see on our command line:

```
Connection closed  
%
```

[Reply](#) [Report](#)



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

SCROLL TO TOP



BECOME A CONTRIBUTOR

You get paid; we donate to tech nonprofits.



GET OUR BIWEEKLY NEWSLETTER

Sign up for Infrastructure as a Newsletter.



HUB FOR GOOD

Working on improving health and education, reducing inequality, and spurring economic growth? We'd like to help.

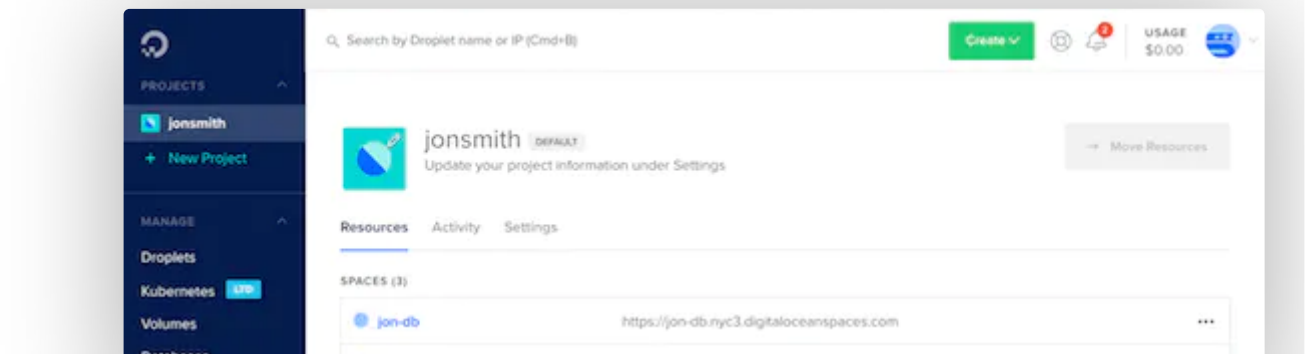
[Featured on Community](#) [Kubernetes Course](#) [Learn Python 3](#) [Machine Learning in Python](#)
[Getting started with Go](#) [Intro to Kubernetes](#)

[DigitalOcean Products](#) [Droplets](#) [Managed Kubernetes](#) [Spaces Object Storage](#) [Marketplace](#)

Welcome to the developer cloud

DigitalOcean makes it simple to launch in the cloud and scale up as you grow – whether you're running one virtual machine or ten thousand.

[Learn More](#)



© 2020 DigitalOcean, LLC. All rights reserved.

Company

- [About](#)
- [Leadership](#)
- [Blog](#)
- [Careers](#)
- [Partners](#)
- [Referral Program](#)
- [Press](#)
- [Legal](#)
- [Security & Trust Center](#)

Products

- [Pricing](#)
- [Products Overview](#)
- [Droplets](#)
- [Kubernetes](#)
- [Managed Databases](#)
- [Spaces](#)

Community

- [Tutorials](#)
- [Q&A](#)
- [Tools and Integrations](#)
- [Tags](#)
- [SCROLL TO TOP](#)
- [Write for DigitalOcean](#)

Contact

- [Get Support](#)
- [Trouble Signing In?](#)
- [Sales](#)
- [Report Abuse](#)
- [System Status](#)

[Marketplace](#)

[Load Balancers](#)

[Block Storage](#)

[API Documentation](#)

[Documentation](#)

[Release Notes](#)

[Presentation Grants](#)

[Hatch Startup Program](#)

[Shop Swag](#)

[Research Program](#)

[Open Source](#)

[Code of Conduct](#)

[SCROLL TO TOP](#)