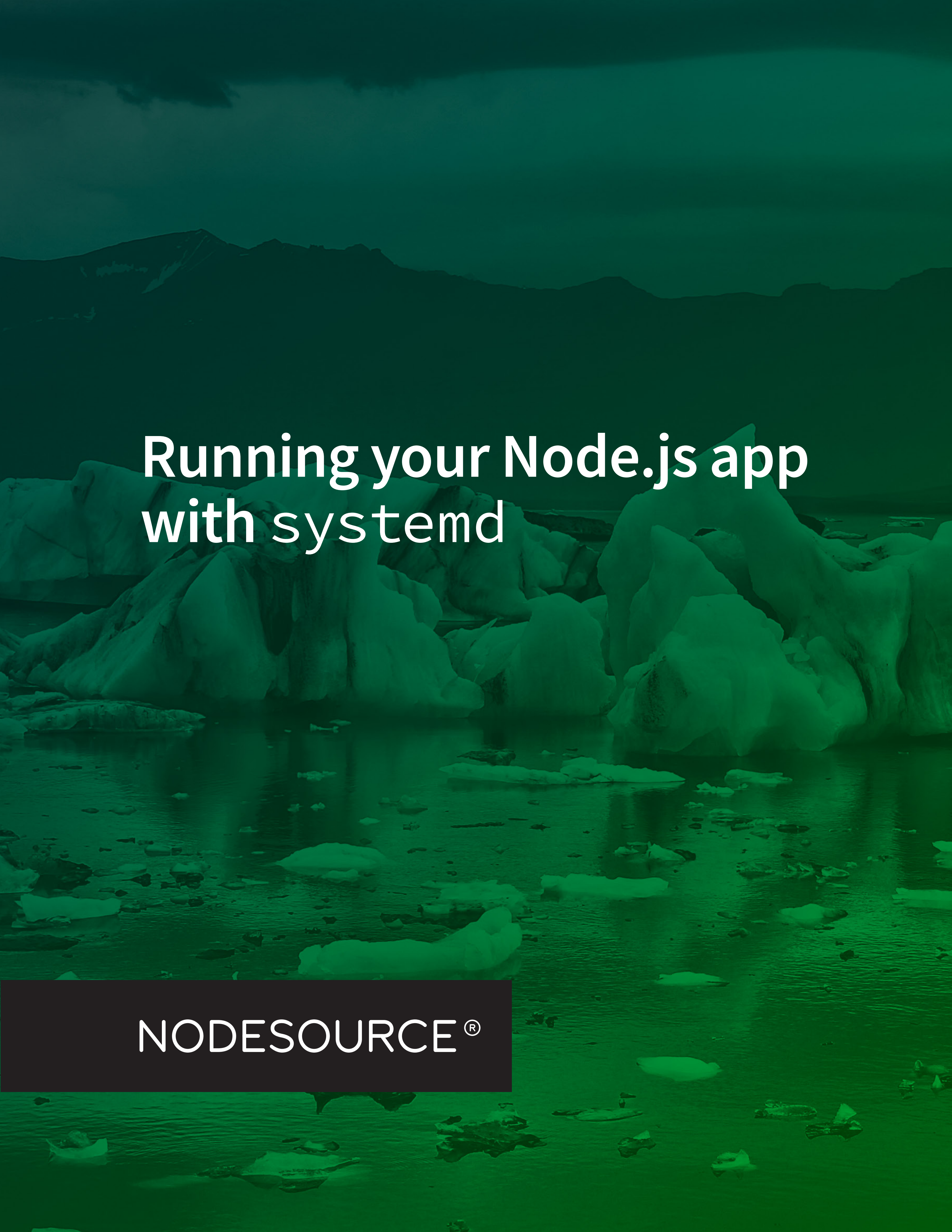

Running your Node.js app with `systemd`

NODESOURCE®

Contents

You've written the next great application, in Node, and you are ready to unleash it upon the world. Which means you can no longer run it on your laptop, you're going to actually have to put it up on some server somewhere and connect it to the real Internet. Eek.

There are a lot of different ways to run an app in production. This guide covers the specific case of running something on a "standard" Linux server that uses `systemd`, which means that we are **not** going to be talking about using Docker, AWS Lambda, Heroku, or any other sort of managed environment. It's just going to be you, your code, and terminal with a `ssh` session my friend.

Before we get started though, let's talk for just a brief minute about what `systemd` actually is and why you should care.

What is `systemd` Anyway?

The full answer to this question is big, as in, "ginormous" sized big. So we're not going to try and answer it fully since we want to get on the the part where we can launch our app. What you need to know is that `systemd` is a thing that runs on "new-ish" Linux servers that is responsible for starting / stopping / restarting programs for you. If you install `mysql`, for example, and whenever you reboot the server you find that `mysql` is already running for you, that happens because `systemd` knows to turn `mysql` on when the machine boots up.

This `systemd` machinery has replaced older systems such as `init` and `upstart` on "new-ish" Linux systems. There is a lot of arguably justified angst in the world about exactly how `systemd` works and how intrusive it is to your system. We're not here to discuss that though. If your system is "new-ish", it's using `systemd`, and that's what we're all going to be working with for the forseeable future.

What does “new-ish” mean specifically? If you are using any of the following, you are using `systemd`:

- CentOS 7 / RHEL 7
- Fedora 15 or newer
- Debian Jessie or newer
- Ubuntu Xenial or newer

Running your App Manually

I’m going to assume you have a fresh installation of Ubuntu Xenial to work with, and that you have set up a default user named `ubuntu` that has `sudo` privileges. This is what the default will be if you spin up a Xenial instance in Amazon EC2. I’m using Xenial because it is currently the newest LTS (Long Term Support) version available from Canonical. Ubuntu Yakkety is available now, and is even *newer*, but Xenial is quite up-to-date at the time of this writing and will be getting security updates for many years to come because of its LTS status.

Use `ssh` with the `ubuntu` user to get into your server, and let’s install Node.

```
$ sudo apt-get -y install curl
$ curl -sL https://deb.nodesource.com/setup_6.x | sudo bash -
$ sudo apt-get -y install nodejs
```

Next let’s create an app and run it manually. Here’s a trivial app I’ve written that simply echoes out the user’s environment variables.

```
const http = require(‘http’);

const hostname = ‘0.0.0.0’;
const port = process.env.NODE_PORT || 3000;
const env = process.env;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader(‘Content-Type’, ‘text/plain’);
  for (var k in env) {
    res.write(k + “: “ + env[k] + “\n”);
  }
  res.end();
});

server.listen(port, hostname, () => {
  console.log(“Server running at http://” + hostname + “:” + port
+ “/”);
});
```

Using your text editor of choice, create a file called `hello_env.js` in the user’s home directory `/home/ubuntu` with the contents above. Next run it with

```
$ /usr/bin/node /home/ubuntu/hello_env.js
```

You should be able to go to

```
http://11.22.33.44:3000
```

in a web browser now, substituting `11.22.33.44` with whatever the actual IP address of your server is, and see a printout of the environment variables for the `ubuntu` user. If that is in fact what you see, great! We know the app runs, and we know the command needed to start it up. Go ahead and press `Ctrl-c` to close down the application. Now we’ll move on to the `systemd` parts.

Creating a `systemd` Service File

The “magic” that’s needed to make `systemd` start working for us is a text file called a `service` file. I say “magic” because for whatever reason, this seems to be the part that people block on when they are going through this process. Fortunately, it’s much less difficult and scary than you might think.

We will be creating a file in a “system area” where everything is owned by the root user, so we’ll be executing a bunch of commands using `sudo`. Again, don’t be nervous, it’s really very straightforward.

The service files for the things that `systemd` controls all live under the directory path

```
/lib/systemd/system
```

so we’ll create a new file there. If you’re using Nano as your editor, open up a new file there with:

```
sudo nano /lib/systemd/system/hello_env.service
```

and put the following contents in it:

```
[Unit]
Description=hello_env.js - making your environment variables rad
Documentation=https://example.com
After=network.target

[Service]
Environment=NODE_PORT=3001
Type=simple
User=ubuntu
ExecStart=/usr/bin/node /home/ubuntu/hello_env.js
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

Let's go ahead and talk about what's in that file. In the `[Unit]` section, the `Description` and `Documentation` variables are obvious. What's less obvious is the part that says

```
After=network.target
```

That tells `systemd` that if it's supposed to start our app when the machine boots up, it should wait until after the main networking functionality of the server is online to do so. This is what we want, since our app can't bind to `NODE_PORT` until the network is up and running.

Moving on to the `[Service]` section we find the meat of today's project. We can specify environment variables here, so I've gone ahead and put in:

```
Environment=NODE_PORT=3001
```

so our app, when it starts, will be listening on port 3001. This is different than the default 3000 that we saw when we launched the app by hand. You can specify the `Environment` directive multiple times if you need multiple environment variables. Next is

```
Type=simple
```

which tells `systemd` how our app launches itself. Specifically, it lets `systemd` know that the app won't try and fork itself to drop user privileges or anything like that. It's just going to start up and run. After that we see

```
User=ubuntu
```

which tells `systemd` that our app should be run as the unprivileged `ubuntu` user. You definitely want to run your apps as unprivileged users to that attackers can't aim at something running as the `root` user.

The last two parts here are maybe the most interesting to us

```
ExecStart=/usr/bin/node /home/ubuntu/hello_env.js
Restart=on-failure
```

First, `ExecStart` tells `systemd` what command it should run to launch our app. Then, `Restart` tells `systemd` under what conditions it should restart the app if it sees that it has died. The `on-failure` value is likely what you will want. Using this, the app will _NOT_ restart if it goes away "cleanly". Going away "cleanly" means that it either exits by itself with an exit value of `0`, or it gets killed with a "clean" signal, such as the default signal sent by the `kill` command. Basically, if our app goes away because we want it to, then `systemd` will leave it turned off. However, if it goes away for any other reason (an unhandled exception crashes the app, for example), then `systemd` will immediately restart it for us. If you want it to restart no matter what, change the value from `on-failure` to `always`.

Last is the `[Install]` stanza. We're going to gloss over this part as it's not very interesting. It tells `systemd` how to handle things if we want to start our app on boot, and you will probably want to use the values shown for most things until you are a more advanced `systemd` user.

Using `systemctl` To Control Your App

The hard part is done! We will now learn how to use the system provided tools to control our app.

To begin with, enter the command

```
$ sudo systemctl daemon-reload
```

You have to do this whenever **any** of the service files change **at all** so that `systemd` picks up the new info.

Next, let's launch our app with

```
$ sudo systemctl start hello_env
```

After you do this, you should be able to go to

```
http://11.22.33.44:3001
```

in your web browser and see the output. If it's there, congratulations, you've launched your app using `systemd`! If the output looks very different than it did when you launched the app manually don't worry, that's normal. When `systemd` kicks off an application, it does so from a *much more minimal environment* than the one you have when you `ssh` into a machine. In particular, the `$HOME` environment variable may not be set by default, so be sure to pay attention to this if your app makes use of any environment variables. You may need to set them yourself when using `systemd`.

You may be interested in what state `systemd` thinks the app is in, and if so, you can find out with

```
$ sudo systemctl status hello_env
```

Now, if you want to stop your app, the command is simply

```
$ sudo systemctl stop hello_env
```

and unsurprisingly, the following will restart things for us

```
$ sudo systemctl restart hello_env
```

If you want to make the application start up when the machine boots, you accomplish that by *enabling* it

```
$ sudo systemtl enable hello_env
```

and finally, if you previously enabled the app, but you change your mind and want to stop it from coming up when the machine starts, you correspondingly *disable* it

```
$ sudo systemctl disable hello_env
```

There is much, much more to learn and know about `systemd`, but this should help get you started with some basics.

Making your app production-ready

Now you're ready to learn how to launch multiple instances of your app, and load balance those behind Nginx to illustrate a more production ready example.

There are a few things we'd like to change about our setup to make it more production ready, which means we're going to have to dive a bit deeper into SysAdmin land.

In particular, the production machine you'll be running your application on likely has more than a single CPU core. Node.js is famously single-threaded, so in order to fully utilize our server's hardware, a good first pass is to run as many Node.js processes as we have cores. For the purposes of this tutorial I'll assume your server has a total of four. We can accomplish our goal then by running four copies of `hello_env.js` on our server, but making each one listen to a different TCP port so they can all coexist peacefully.

Of course, you don't want your clients to have to know anything about how many processes you are running, or about multiple ports. They should just see a single HTTP endpoint that they need to connect with. Therefore, we need to accept all the incoming connections in a single place, and then load balance the requests across our pool of processes from there. Fortunately, the freely available (and completely awesome) `Nginx` does an outstanding job as a load balancer, so we'll configure it for this purpose a bit later.

Configuring `systemd` to Run Multiple Instances

As it turns out, the `systemd` authors assumed you might want to run more than one copy of something on a given server. For a given service `foo`, you'll generally want to create a `foo.service` file to tell `systemd` how to manage it. This is exactly what we did earlier in this tutorial. However, if you instead create a file called `foo@.service`, you are telling `systemd` that you may want to run more than a single instance of `foo`. This sounds pretty much just like what we want, so let's rename our service file from before.

```
$ sudo mv /lib/systemd/system/hello_env.service /lib/systemd/
system/hello_env@.service
```

Next comes the "interesting" or "neat" part of this modified `systemd` configuration. When you have a service file such as this that can be used to start multiple copies of the same thing, you *additionally* get to pass the service file a variable based on how you invoke the service with `systemctl`. Modify the contents of

```
/lib/systemd/system/hello_env@.service
```

to contain the following:

```
[Unit]
Description=hello_env.js - making your environment variables rad
Documentation=https://example.com
After=network.target

[Service]
Environment=NODE_PORT=%i
Type=simple
User=chl
ExecStart=/usr/bin/node /home/chl/hello_env.js
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

The only difference from before is that now, we set:

```
Environment=NODE_PORT=%i
```

This lets us set the port that our application will listen on based on how we start it up. To start up four copies of `hello_env.js`, listening on ports ranging from 3001 to 3004, we can do the following:

```
$ sudo systemctl start hello_env@3001
$ sudo systemctl start hello_env@3002
$ sudo systemctl start hello_env@3003
$ sudo systemctl start hello_env@3004
```

Or, if you prefer a one-liner, the following should get the job done for you:

```
$ for port in $(seq 3001 3004); do sudo systemctl start
hello_env@$port; done
```

All of the `systemctl` commands we saw before (start / stop / restart / enable / disable) will still work in the same way they did previously, you just have to include the port number after the “@” symbol when we start things up.

This is not a point to be glossed over. You are now starting up multiple versions of *the exact same service* using `systemctl`. Each of these is a unique entity that can be controlled and monitored independently of the others, despite the fact that they share a single, common configuration file. Therefore, if you want to start all four processes when your server boots up, you need to use `systemctl enable` on *each* of them:

```
$ sudo systemctl enable hello_env@3001
$ sudo systemctl enable hello_env@3002
$ sudo systemctl enable hello_env@3003
$ sudo systemctl enable hello_env@3004
```

There is no included tooling that will automatically control all of the related processes, but it’s trivial to write a small script to do this if you need it. For example, here’s a `bash` script we could use to stop everything:

```
#!/bin/bash -e

PORTS=”3001 3002 3003 3004”

for port in ${PORTS}; do
  systemctl stop hello_env@${port}
done

exit 0
```

You could save this to a file called `stop_hello_env`, then make it executable and invoke it with:

```
$ chmod 755 stop_hello_env
$ sudo ./stop_hello_env
```

Please Note: that there is no requirement on having an integer or numeric value after the "@" symbol. We are just doing this as a trick to designate the port number we want to listen to since that's how our app works. We could just as easily have used a string to specify different config files if *that* was how our app worked. For example, if `hello_env.js` accepted a `--config` command line option to specify a config file, we could have created a `hello_env@.service` file like this:

```
[Unit]
Description=hello_env.js - making your environment variables rad
Documentation=https://example.com
After=network.target

[Service]
Type=simple
User=chl
ExecStart=/usr/bin/node /home/chl/hello_env.js --config /home/
ubuntu/%i
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

and then started our instances doing something like:

```
$ sudo systemctl start hello_env@config1
$ sudo systemctl start hello_env@config2
$ sudo systemctl start hello_env@config3
$ sudo systemctl start hello_env@config4
```

Assuming that we did in fact have files under `/home/ubuntu` named `config1` through `config4`, we would achieve the same effect.

Go ahead and start your four processes up, and try visting the following URLs to make sure things are working:

```
http://11.22.33.44:3001
http://11.22.33.44:3002
http://11.22.33.44:3003
http://11.22.33.44:3004
```

again substituting the IP address of your server instead of `11.22.33.44`. You should see very similar output on each, but the value for `NODE_PORT` should correctly reflect the port you are connecting to. Assuming things look good, it's on to the final step!

Configuring Nginx as a Load Balancer

First, let's install `Nginx` and remove any default configuration that it ships with. On Debian style systems (Debian, Ubuntu, and Mint are popular examples), you can do this with the following commands:

```
$ sudo apt-get update
$ sudo apt-get -y install nginx-full
$ sudo rm -fv /etc/nginx/sites-enabled/default
```

Next we'll create a load balancing configuration file. We have to do this as the `root` user, so assuming you want to use `nano` as your text editor, you can create the needed file with:

```
$ sudo nano /etc/nginx/sites-enabled/hello_env.conf
```

and put the following into it:

```
upstream hello_env {
    server 127.0.0.1:3001;
    server 127.0.0.1:3002;
    server 127.0.0.1:3003;
    server 127.0.0.1:3004;
}
server {
    listen 80 default_server;
    server_name _;

    location / {
        proxy_pass http://hello_env;
        proxy_set_header Host $host;
    }
}
```

Luckily for us, that's really all there is to it. This will make `Nginx` use its default load balancing scheme which is round-robin. There are other schemes available if you need something different.

Go ahead and restart `Nginx` with:

```
$ sudo systemctl restart nginx
```

Yes, `systemd` handles starting / stopping / restarting `Nginx` as well, using the same tools and semantics.

You should now be able to run the following command repeatedly:

```
$ curl -s http://11.22.33.44
```

and see the same sort of output you saw in your browser, but the `NODE_PORT` value should walk through the possible options 3001 - 3004 incrementally. If that's what you see, congrats, you're all done! We have four copies of our application running now, load balanced behind `Nginx`, and `Nginx` itself is listening on the default port 80 so our clients don't have to know or care about the details of the backend setup.

Next Steps

There has probably never been a better or easier time to learn basic Linux system administration. Things such as Amazon's AWS EC2 service mean that you can fire up just about any kind of Linux you might want to, play around with it, and then just delete it when you are done. You can do this for very minimal costs, and you don't run the risk of breaking anything in production when you do.

Learning all there is to know about `systemd` is more than can reasonably covered in this tutorial, but there is ample documentation online if you want to know more. I personally have found the "systemd for Administrators Blog Series", linked to from that page, a very valuable resource.

I hope you're had fun getting this app up and running!