

How To Set Up a Node.js Application for Production on Debian 9

Posted September 6, 2018  7.9k

NODE.JS

NGINX

DEBIAN 9

By: Kathleen Juell By: Brennen Bearnes

Not using **Debian 9**? Choose a different version:

Introduction

Node.js is an open-source JavaScript runtime environment for building server-side and networking applications. The platform runs on Linux, macOS, FreeBSD, and Windows. Though you can run Node.js applications at the command line, this tutorial will focus on running them as a service. This means that the applications will restart on reboot or failure and are safe for use in a production environment.

In this tutorial, you will set up a production-ready Node.js environment on a single Debian 9 server. This server will run a Node.js application managed by PM2, and provide users with secure access to the application through an Nginx reverse proxy. The Nginx server will offer HTTPS, using a free certificate provided by Let's Encrypt.

Prerequisites

This guide assumes that you have the following:

- A Debian 9 server setup, as described in the initial server setup guide for Debian 9. You should have a non-root user with sudo privileges and an active firewall.
- A domain name pointed at your server's public IP. This tutorial will use the domain name **example.com** throughout.
- Nginx installed, as covered in How To Install Nginx on Debian 9.
- Nginx configured with SSL using Let's Encrypt certificates. How To Secure Nginx with Let's Encrypt on Debian 9 will walk you through the process.

When you've completed the prerequisites, you will have a server serving your domain's default placeholder page at `https://example.com/`.

Step 1 — Installing Node.js

Let's begin by installing the latest LTS release of Node.js, using the NodeSource package archives.

To install the NodeSource PPA and access its contents, you will first need to update your package index and install `curl`:

```
$ sudo apt update
$ sudo apt install curl
```

Make sure you're in your home directory, and then use `curl` to retrieve the installation script for the Node.js 8.x archives:

```
$ cd ~
$ curl -sL https://deb.nodesource.com/setup_8.x -o nodesource_setup.sh
```

You can inspect the contents of this script with `nano` or your preferred text editor:

```
$ nano nodesource_setup.sh
```

When you're done inspecting the script, run it under `sudo`:

```
$ sudo bash nodesource_setup.sh
```

The PPA will be added to your configuration and your local package cache will be updated automatically. After running the setup script from Nodesource, you can install the Node.js package:

```
$ sudo apt install nodejs
```

To check which version of Node.js you have installed after these initial steps, type:

```
$ nodejs -v
```

Output

v8.11.4

Note: When installing from the NodeSource PPA, the Node.js executable is called `nodejs`, rather than `node`.

The `nodejs` package contains the `nodejs` binary as well as `npm`, a package manager for Node modules, so you don't need to install `npm` separately.

`npm` uses a configuration file in your home directory to keep track of updates. It will be created the first time you run `npm`. Execute this command to verify that `npm` is installed and to create the configuration file:

```
$ npm -v
```

Output

```
5.6.0
```

In order for some `npm` packages to work (those that require compiling code from source, for example), you will need to install the `build-essential` package:

```
$ sudo apt install build-essential
```

You now have the necessary tools to work with `npm` packages that require compiling code from source.

With the Node.js runtime installed, let's move on to writing a Node.js application.

Step 2 — Creating a Node.js Application

Let's write a *Hello World* application that returns "Hello World" to any HTTP requests. This sample application will help you get Node.js set up. You can replace it with your own application — just make sure that you modify your application to listen on the appropriate IP addresses and ports.

First, let's create a sample application called `hello.js`:

```
$ cd ~
$ nano hello.js
```

Insert the following code into the file:

```
~/hello.js

const http = require('http');

const hostname = 'localhost';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World!\n');
```

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

Save the file and exit the editor.

This Node.js application listens on the specified address (`localhost`) and port (`3000`), and returns "Hello World!" with a `200` HTTP success code. Since we're listening on `localhost` , remote clients won't be able to connect to our application.

To test your application, type:

```
$ node hello.js
```

You will see the following output:

Output

```
Server running at http://localhost:3000/
```

Note: Running a Node.js application in this manner will block additional commands until the application is killed by pressing `CTRL+C` .

To test the application, open another terminal session on your server, and connect to `localhost` with `curl` :

```
$ curl http://localhost:3000
```

If you see the following output, the application is working properly and listening on the correct address and port:

Output

```
Hello World!
```

If you do not see the expected output, make sure that your Node.js application is running and configured to listen on the proper address and port.

Once you're sure it's working, kill the application (if you haven't already) by pressing `CTRL+C` .

Step 3 — Installing PM2

Next let's install PM2, a process manager for Node.js applications. PM2 makes it possible to daemonize applications so that they will run in the background as a service.

Use `npm` to install the latest version of PM2 on your server:

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

The `-g` option tells `npm` to install the module globally, so it's available system-wide.

Let's first use the `pm2 start` command to run your application, `hello.js`, in the background:

```
$ pm2 start hello.js
```

This also adds your application to PM2's process list, which is outputted every time you start an application:

Output

```
[PM2] Spawning PM2 daemon with pm2_home=/home/sammy/.pm2
[PM2] PM2 Successfully daemonized
[PM2] Starting /home/sammy/hello.js in fork_mode (1 instance)
[PM2] Done.
```

App name	id	mode	pid	status	restart	uptime	cpu	mem	user	watching
hello	0	fork	1338	online	0	0s	0%	23.0 MB	sammy	disabled

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

As you can see, PM2 automatically assigns an `App name` (based on the filename, without the `.js` extension) and a `PM2 id`. PM2 also maintains other information, such as the `PID` of the process, its current status, and memory usage.

Applications that are running under PM2 will be restarted automatically if the application crashes or is killed, but we can take an additional step to get the application to launch on system startup using the `startup` subcommand. This subcommand generates and configures a startup script to launch PM2 and its managed processes on server boots:

```
$ pm2 startup systemd
```

The last line of the resulting output will include a command to run with superuser privileges to set PM2 to start on boot:

Output

[PM2] Init System found: systemd

[PM2] To setup the Startup Script, copy/paste the following command:

```
sudo env PATH=$PATH:/usr/bin /usr/lib/node_modules/pm2/bin/pm2 startup systemd -u sammy --hp /home/sammy
```

Run the command from the output, with your username in place of **sammy**:

```
$ sudo env PATH=$PATH:/usr/bin /usr/lib/node_modules/pm2/bin/pm2 startup systemd -u sammy --hp /home/sammy
```

As an additional step, we can save the PM2 process list and corresponding environments:

```
$ pm2 save
```

You have now created a systemd *unit* that runs **pm2** for your user on boot. This **pm2** instance, in turn, runs **hello.js**.

Start the service with **systemctl**:

```
$ sudo systemctl start pm2-sammy
```

Check the status of the systemd unit:

```
$ systemctl status pm2-sammy
```

For a detailed overview of systemd, see [Systemd Essentials: Working with Services, Units, and the Journal](#).

In addition to those we have covered, PM2 provides many subcommands that allow you to manage or look up information about your applications.

Stop an application with this command (specify the PM2 App name or id):

```
$ pm2 stop app_name_or_id
```

Restart an application:

```
$ pm2 restart app_name_or_id
```

List the applications currently managed by PM2:

```
$ pm2 list
```

Get information about a specific application using its `App name` :

```
$ pm2 info app_name
```

The PM2 process monitor can be pulled up with the `monit` subcommand. This displays the application status, CPU, and memory usage:

```
$ pm2 monit
```

Note that running `pm2` without any arguments will also display a help page with example usage.

Now that your Node.js application is running and managed by PM2, let's set up the reverse proxy.

Step 4 — Setting Up Nginx as a Reverse Proxy Server

Your application is running and listening on `localhost` , but you need to set up a way for your users to access it. We will set up the Nginx web server as a reverse proxy for this purpose.

In the prerequisite tutorial, you set up your Nginx configuration in the `/etc/nginx/sites-available/example.com` file. Open this file for editing:

```
$ sudo nano /etc/nginx/sites-available/example.com
```

Within the `server` block, you should have an existing `location /` block. Replace the contents of that block with the following configuration. If your application is set to listen on a different port, update the highlighted portion to the correct port number:

```
server {
    ...
    location / {
        proxy_pass http://localhost:3000;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }
    ...
}
```

This configures the server to respond to requests at its root. Assuming our server is available at `example.com` , accessing `https://example.com/` via a web browser would send the request to `hello.js` , listening on port `3000` at `localhost` .

You can add additional `location` blocks to the same server block to provide access to other applications on the same server. For example, if you were also running another Node.js application on port `3001`, you could add this location block to allow access to it via `https://example.com/app2`:

`/etc/nginx/sites-available/example.com` — Optional

```
server {  
    ...  
    location /app2 {  
        proxy_pass http://localhost:3001;  
        proxy_http_version 1.1;  
        proxy_set_header Upgrade $http_upgrade;  
        proxy_set_header Connection 'upgrade';  
        proxy_set_header Host $host;  
        proxy_cache_bypass $http_upgrade;  
    }  
    ...  
}
```

Once you are done adding the location blocks for your applications, save the file and exit your editor.

Make sure you didn't introduce any syntax errors by typing:

```
$ sudo nginx -t
```

Restart Nginx:

```
$ sudo systemctl restart nginx
```

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. Try it out by accessing your server's URL (its public IP address or domain name).

Conclusion

Congratulations! You now have your Node.js application running behind an Nginx reverse proxy on a Debian 9 server. This reverse proxy setup is flexible enough to provide your users access to other applications or static web content that you want to share.

By: Kathleen Juell By: Brennen Bearnes

♥ Upvote (4)

📌 Subscribe

🔗 Share



We just made it easier for you to deploy faster.

[TRY FREE](#)

Related Tutorials

[How To Deploy a PHP Application with Kubernetes on Ubuntu 16.04](#)

[How To Ensure Code Quality with SonarQube on Ubuntu 18.04](#)

[How To Set Up a Private Docker Registry on Ubuntu 18.04](#)



[How To Secure a Containerized Node.js Application with Nginx, Let's Encrypt, and Docker Compose](#)

[How to Set Up an Nginx Ingress with Cert-Manager on DigitalOcean Kubernetes](#)

2 Comments

Leave a comment...

[Log In to Comment](#)

-  [vlamoly](#) *September 24, 2018*
-  0 Hi ! Thank you very much for those tutorials on Debian 9. I've been through all and now I've got my https node js app + nginx :)
However I confirm that there is no need of the "sudo ufw allow 3000" because it's running on localhost (and

do not need any allow from ufw for this) and we redirect traffic from HTTPS and HTTP to it via nginx. So there is no need to open this port for outside incoming traffic.

 [katjuell](#) MOD September 26, 2018

o Hi [@vlamoly](#)! Thank you very much for the comment. I was initially experimenting with having readers expose the app publicly in Step 2, just to walk through how you would do it at that stage, but that didn't end up making sense in the scope of the article. And then I never removed the prerequisite! Thank you so much for pointing this out.

I'm glad that the Debian 9 guides have been of use to you!



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



Copyright © 2019 DigitalOcean™ Inc.

[Community](#) [Tutorials](#) [Questions](#) [Projects](#) [Tags](#) [Newsletter](#) [RSS](#) 

[Distros & One-Click Apps](#) [Terms, Privacy, & Copyright](#) [Security](#) [Report a Bug](#) [Write for DOnations](#) [Shop](#)