



Robert Tod

[Follow](#)

Javascript Engineer working at Qubit

Jan 11, 2017 · 12 min read

Tutorial: Creating and managing a Node.js server on AWS, part 2

In [part 1](#) we started a server, responding to HTTP requests on port 3000. In this tutorial we will look at 4 more concepts

- Serving HTTP traffic on the standard port, `80`
- Keeping the Node.js process running
- Deploying code into the server
- Serving some HTML

Serving HTTP traffic on the standard port, 80

Paste this URL into a URL bar in a new tab.

```
http://imgur.com:80/
```

You will notice the `80` gets dropped in the URL bar. That's because port `80` is the default port for HTTP traffic.

HTTPS traffic uses port `443`. Try Hacker News.

```
https://news.ycombinator.com:443/
```

Because these ports are often public, you need special privileges to run processes using them. Also, it is not great to run Node.js on port `80` or `443` directly because you may want to open up a few different applications on these ports. With a router you will be able to send

traffic from port `80` or `443` to any program you wish, depending on the headers of the incoming HTTP request.

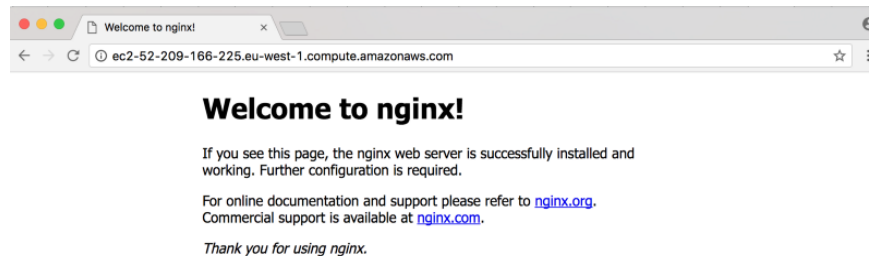
There are few great choices for a router, but I find that nginx is generally the best tool for most things. It's great for building your first ever app, or when you need to scale up to millions of visitors.

Before beginning, start your server if it is stopped and SSH into it (as shown in the last tutorial). Once logged in, we can install **nginx**.

Ubuntu comes with it's own package manager, **apt-get**. Using **apt-get**, we can install **nginx** in one command.

```
sudo apt-get install nginx
```

Most Linux distributions come with a package manager, so Google your version if **apt-get** doesn't work. **apt-get** runs nginx automatically after install so you should now have it running on port `80`, check by entering your public DNS URL into a browser.



Default nginx page

If this doesn't work, you might need to start it manually.

```
sudo /etc/init.d/nginx start
```

If you are still can't reach the server then check out Stack Overflow or post a comment here.

We need to configure **nginx** to route port `80` traffic to port `3000` .
nginx has config placed in the `/etc/nginx/sites-available` folder where there is already a default config which serves the **nginx** welcome page we saw earlier.

You can take a look at this config using `cat`.

```
cat /etc/nginx/sites-available/default
```

How are **nginx** configs set up? Configs are stored in plain text files in `sites-available` with any name. Linking them into the `sites-enabled` folder will cause them to be read and used when **nginx** starts. All of the configs are combined together by **nginx**.

Let's first remove the default config from `sites-enabled` , we will leave it in `sites-available` for reference.

```
sudo rm /etc/nginx/sites-enabled/default
```

Create a config file in `sites-available` and name it whatever you like.

```
sudo nano /etc/nginx/sites-available/tutorial
```

The following is the config we are going to use.

```
server {
    listen 80;
    server_name tutorial;
    location / {
        proxy_set_header    X-Real-IP    $remote_addr;
        proxy_set_header    Host        $http_host;
        proxy_pass            http://127.0.0.1:3000;
    }
}
```

This will forward all HTTP traffic from port `80` to port `3000` .

Link the config file in `sites enabled` (this will make it seem like the file is actually copied in `sites-enabled`).

```
sudo ln -s /etc/nginx/sites-available/tutorial
/etc/nginx/sites-enabled/tutorial
```

[Read more about symbolic links here if they are unfamiliar.](#)

Restart **nginx** for the new config to take effect.

```
sudo service nginx restart
```

If you did not stop/start the server since the last tutorial then you may still have your node application running.

```
# list background jobs
jobs
```

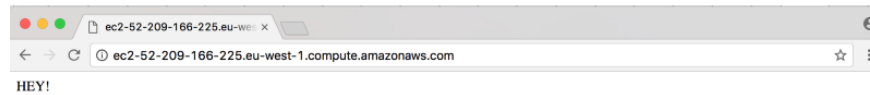
If your application shows up then no need to run it again. If it is not running then you need to start it.

```
node tutorial/index.js
```

Once the server is running, press `ctrl+z` , then resume it as a background task.

```
bg %1
```

Now visit your server's public DNS URL, using port `80` .



The server is running on port `80` !

Keeping the Node.js process running

It's quite tedious using `ctrl+z` to pause a process, and then running it in the background. Also, doing it this way will not allow the Node.js process to restart when you restart your server after an update or crash.

Before moving forward, stop your running node process

```
# Nukes all Node processes
killall -9 node
```

To keep these processes running we are going to use a great NPM package called PM2. While in an SSH session, install PM2 globally.

```
npm i -g pm2
```

To start your server, simply use `pm2` to execute `index.js` .

```
pm2 start tutorial/index.js
```

To make sure that your PM2 restarts when your server restarts

```
pm2 startup
```

This will print out a line of code you need to run depending on the server you are using. Run the code it outputs.

Finally, save the current running processes so they are run when PM2 restarts.

```
pm2 save
```

That's it! You can log out/in to SSH, even restart your server and it will continue to run on port 80.

To list all processes use

```
pm2 ls
```

Your process will have a pretty generic name, something like `index`, which would be hard to differentiate if you had a few other microservices running on the server too. Let's stop the process, remove it and start it back up with a nicer name.

```
# Use the number listed in pm2 ls
# to stop the daemon
pm2 stop index

# Remove it from the list
pm2 delete index

# Start it again, but give it a
# catchy name
pm2 start tutorial/index.js --name "Tutorial"
```

Great! Check out the [PM2 docs](#) to see what else you can do with process management.

Deploying code into the server

Instead of writing code in an SSH session, let's push the code to a git repo in Github, SSH into the server and pull in the new code.


Go to [Github](#) or your favourite source control website, login and create a new repository named what you like. If you aren't able to make it private, make it public (it's okay, no one's going to look anyway).

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner

Repository name

 roberttodd

 /

tutorial-pt-2

Great repository names are short and memorable. Need inspiration? How about [jubilant-eureka](#).

Description (optional)

Source code for an tutorial

☒ Public

Anyone can see this repository. You choose who can commit.

☐ Private

You choose who can see and commit to this repository.

☐ Initialize this repository with a README

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None

 |

Add a license: None

 ⓘ

Create repository

Make a new directory wherever you like to put your code projects **locally** (btw, to exit an SSH session just type `exit`). I named mine with gusto and imagination, "tutorial-pt-2".

```
cd ~/Code
mkdir tutorial-pt-2
cd tutorial-pt-2
```

Now set up your origin, make an empty commit and push it up, setting your upstream branch as master.

```
git init
git commit --allow-empty -m "Well this is my first commit,
*yay*"

# Use your repo's origin URL here
git remote add origin git@github.com:roberttod/tutorial-pt-
2.git
git push -u origin master
```

It's nice to start with an empty commit :).

Like we did on the server run `npm init` and then create an `index.js` file using the same code from the last tutorial. *P.S. I am missing out the `;` s on purpose, check out Standard Style.*

```
const express = require('express')
const app = express()
app.get('/', (req, res) => {
  res.send('HEY!')
})

app.listen(3000, () => console.log('Server running on port
3000'))
```

NPM install express

```
npm install express --save
```

Also, let's add a `.gitignore` file so that we don't check in the `node_modules` directory. `.DS_Store` files always get added to directories by OSX, they contain folder meta data. We want to ignore these too.


```
node_modules
.DS_Store
```

In general, it is best to only check in code that you actually need checked in, if you ever find yourself with chunks of copied or library code in your git repo then there is probably a better way to do it.

Now add all your code and push it up

```
git add .
git commit -m "Ze server."
git push
```

Now we need to pull the code into the server. This is where it can be kind of tricky.

We need to SSH into the server, generate a SSH private/public key pair and then add it as a deployment key in source control (i.e. Github). Only when the server is allowed access to the remote repo will it be able to clone the code and pull down changes.

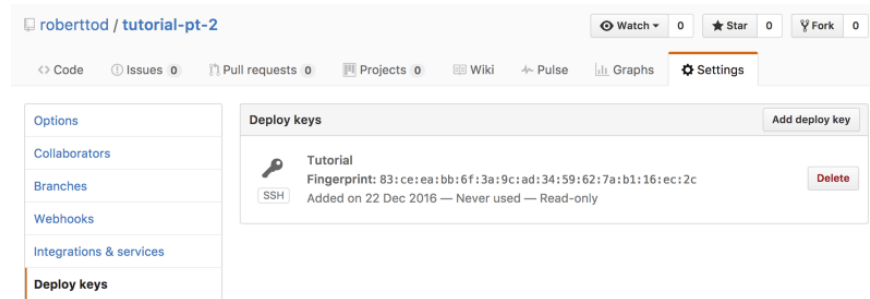
SSH into your server and generate the key pair.

```
# When prompted, use the default name.
# No need for a pass phrase.
ssh-keygen -t rsa
```

Show the contents of the file

```
cat ~/.ssh/id_rsa.pub
```

Select the key's contents and copy it into Github. Deploy keys are added in a section called **Deploy keys** in the settings for your repo.



Paste your key and call it something meaningful.

Whenever you are logged in over SSH, you want the keys to be added so that they are used to authenticate with Github. To do this, add these lines to the top of your `~/.bashrc` file.

```
# Start the SSH agent
eval `ssh-agent -s`

# Add the SSH key
ssh-add
```

This will make sure you use the keys whenever you log on to the server. To run the code without logging out, execute the `.bashrc` file

```
source ~/.bashrc
```

Now we can clone the repo! Remove any previous code on the server and in the user directory, clone the repo

```
# You should use your own git URL.
git clone git@github.com:roberttod/tutorial-pt-2.git
```

If it works, it will allow you to type “yes” to add github as a known host, then the repo will be downloaded.

In a nice world we like to completely avoid ever using SSH. For deployment, we are going to use PM2 in order for us to do the git cloning on the server for us, with some bonus features.

Before using PM2, remove the code you just pulled in from git into your server.

```
rm -rf ~/tutorial-pt-2
```

While you are still in the SSH session, ensure that there are no processes still running on PM2, if there are then remove them.

```
pm2 ls  
  
# Only do this if a task is still running  
pm2 delete tutorial
```

In your **local** version of the project, install PM2 globally

```
npm i -g pm2
```

Now we need to add a config file PM2 can read so that it knows how to deploy.

PM2 configs are fully explained in the [PM2 docs](#). The config file can be auto generated but I prefer to just create my own from scratch, avoiding any config I don't need.

The config file should be named `ecosystem.config.js` and should look like this

```
module.exports = {  
  apps: [{  
    name: 'tutorial-2',
```

```
    script: './index.js'
  }],
  deploy: {
    production: {
      user: 'ubuntu',
      host: 'ec2-52-209-166-225.eu-west-
1.compute.amazonaws.com',
      key: '~/ssh/tutorial-2.pem',
      ref: 'origin/master',
      repo: 'git@github.com:roberttod/tutorial-pt-2.git',
      path: '/home/ubuntu/tutorial-2',
      'post-deploy': 'npm install && pm2 startOrRestart
ecosystem.config.js'
    }
  }
}
```

You will need to add your own host in the config file.

How does PM2 use this config file? When you run `pm2 deploy ...`, PM2 SSHs into your server, clones your repo into the directory specified in `path`, then it runs the `post-deploy` on the server (so it starts your server using the PM2 installed globally on the server). After the deploy, you will be able to run `pm2 ls` on the server to see all the apps running, their names will be the same as specified in the config file.

Once the file is saved, setup the directories on the remote

```
pm2 deploy ecosystem.config.js production setup
```

If you run into any auth issues, look back at setting up the SSH agent to make sure you didn't miss anything.

Once setup, commit and push your changes to Github so that when it clones it gets your `ecosystem.config.js` file, which is going to be used to start your app using PM2 on the server.

```
git add .
git commit -m "Setup PM2"
git push
```

Now you can run the deploy command

```
pm2 deploy ecosystem.config.js production
```

This should come up with an error, which will be that `npm` was not found.

The reason for this is because of some code in the `.bashrc` file of the server. This code stops the file from running if the shell is not **interactive**. Unlike using `ssh`, PM2 logs into the server in a **non-interactive** shell. NVM is set up in the `.bashrc` file so PM2 isn't running NVM, which adds the `npm` executable (thus the error from PM2). [Read more about interactive/non interactive shells.](#)

SSH into your server and open up the `~/.bashrc` file. The code that excludes **non-interactive** sessions is near the top.

```
# If not running interactively, don't do anything

case $- in
  *i*) ;;
  *) return;;
esac
```

This code is some magic bash shit. All we need to do is move the NVM code above this code, so it always executes. Find the following lines and move them above the case statement

```
export NVM_DIR="/home/ubuntu/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && . "$NVM_DIR/nvm.sh" # This
loads nvm
```

Save and exit.

Back on your local terminal, try running the PM2 deploy again

```
pm2 deploy ecosystem.config.js production
```

It should work this time! And your server should still be running when you check in a browser.

Using a global PM2 on the server and a global PM2 on the client is a bit messy. It would be better if our code used the local version of the PM2 package. To do this, add a `deploy` and `restart` script to your `package.json`

```
...
  "main": "index.js",
  "scripts": {
    "restart": "pm2 startOrRestart ecosystem.config.js",
    "deploy": "pm2 deploy ecosystem.config.js production",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": {
    ...
```

Install PM2 locally and save, using `--save-dev`

```
npm i pm2 --save-dev
```

For those not that familiar with NPM, adding `--save` or `--save-dev` adds the package, along with a version number to `package.json`. Any packages in the `package.json` file will be installed when running `npm install`, which happens in the PM2 `post-deploy`.

Before deploying, commit all your changes and push to git.

When you run `npm deploy`, it will now use the local version of PM2. Neat!

```
npm run-script deploy
```

In case you find that your app isn't running, check the PM2 logs on your server, located in `~/.pm2/logs/`.

To make sure the app restarts when the server restarts, SSH back into your server and run

```
pm2 save
```

The only PM2 process running should be your deployed server, so PM2 will ensure that is always kept running.

Serving HTML

Before concluding the tutorial, let's use our deployment setup to change the server so that it serves HTML, rather than just responding with "Hey".

Remove the route handler for `/` and set up a static directory that express will use to serve static files.

```
const express = require('express')
const app = express()

app.use(express.static('public'))
app.listen(3000, () => console.log('Server running on port 3000'))
```

By calling `express.static` with "public", static files will be served from the `public` directory. If we add an `index.html` in `/public`, then this will be automatically served at `/` by express (if we left a custom `/` handler, then that would override this functionality).

Save a simple HTML file in `public/index.html`

```
<!DOCTYPE html>
<html>
  <head>
```

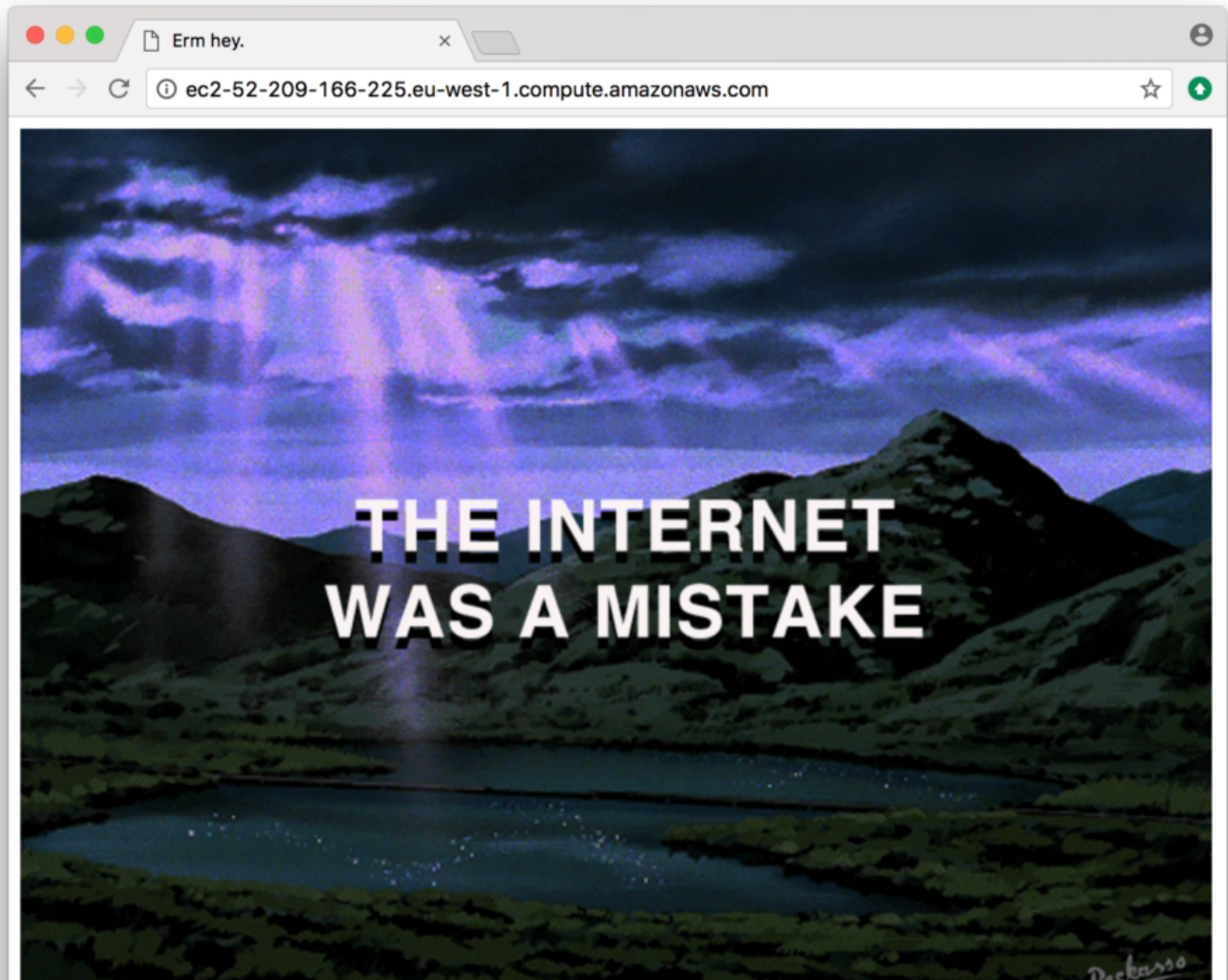
```
<title>Erm hey.</title>
</head>
<body>
  
  </body>
</html>
```

Commit these changes (perhaps you could check the server still works by running `node index.js` locally and visiting <http://localhost:3000>).

Deploy the changes

```
npm run-script deploy
```

Your server will now respond with your static HTML page.



Conclusion

You should now have a simple express app that is fairly easy to deploy on EC2, running on port `80`. I left my source code public in case you would like to take a look <https://github.com/roberttod/tutorial-pt-2>.

If you like this tutorial, I am doing a few more explaining [how to setup Node with MySQL](#).



like!



tweet!



about!

Hacker Noon is how hackers start their afternoons. We're a part of the @AMI family. We are now accepting submissions and happy to discuss advertising & sponsorship opportunities.

If you enjoyed this story, we recommend reading our latest tech stories and trending tech stories. Until next time, don't take the realities of the world for granted!



