Application Server Setup Instructions

Installation of Pre-requisites

1. Update Apt Get

**$sudo apt-get update**

1. Install NPM

**$sudo apt-get install npm**

1. Install PM2

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

1. Install Node Js

Instructions from: h<ttps://nodejs.org/en/download/package-manager/#debian-and-u>buntu-based-linux-distributions

1. **$curl -sL https://deb.nodesource.com/setup\_6.x | sudo -E bash -**
2. **$sudo apt-get install -y nodejs**
3. Install Build Tools

**$sudo apt-get install -y build-essential**

1. **Install required tools for css, js minification**

**$sudo npm install -g uglify-es**

**$sudo npm install -g csso-cli**

**$sudo npm install -g less**

1. Install ImageMagick Library

**$sudo apt install imagemagick**

Set up Deployment User

Since the Application server will be running Node JS, we’ll use the ‘node’ user as the default deployment user. If ‘node’ user doesn’t already exist, create one as per command below.

**$sudo adduser --disabled-password --gecos "" node**

Set up Deployment Directories on Server

Production:

Get back to user 'ubuntu', then:

1. **$sudo su**
2. **$mkdir -p /var/www/appserver/production**
3. **$mkdir /var/www/appserver/production/data**
4. **$cd /var/www/appserver/production**
5. **$chown -R node /var/www/appserver/production** #change owner to new 'node' user

Staging:

Get back to user 'ubuntu', then:

1. **$sudo su**
2. **$mkdir –p /var/www/appserver/staging**
3. **$mkdir /var/www/appserver/staging/data**
4. **$cd /var/www/appserver/staging**
5. **$chown -R node /var/www/appserver/staging** #change owner to new 'node' user

Set up the Data Directory

Production:

On Production Environment we’d want to mount an AWS Elastic Block Storage Volume to the server for storing all files. Refer: <http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-using-volumes.html>

1. List available devices

**$lsblk**

Look for the device that is not already mounted, because that's the EBS volume that we want to mount

1. Check if we need to format or not

**$sudo file -s /dev/<device\_name>** # as identified in step 1 above

If the output is just "data" then we need to format using the following command

1. **sudo mkfs -t ext4 /dev/<device\_name>** #only if the format in 2 was 'data'
2. **sudo mkdir /var/www/appserver/production/data** #create mount point directory
3. **sudo mount /dev/<device\_name> /var/www/appserver/production/data**
4. **sudo chown node /var/www/appserver/production/data** #change ownership to node user

Staging:

On Staging environment, we’ll just use the local disk space on the server machine to store all the files. So, simply make sure the **/var/www/staging/data** folder is created and is owned by the user ‘**node’** (should have happened as part of the earlier steps)

1. **$sudo su**
2. **$mkdir /var/www/appserver/staging/data**
3. **$cd /var/www/appserver/staging**
4. **$chown -R node /var/www/appserver/staging** #change owner to new 'node' user

Set up SSH Keys to enable Silent Deployments

Production:

On the production server, we do not want automatic deployments from the CI machines (e.g., CircleCI) after every successful build. Rather, what we’d want is to allow manual deployments from the dev machine (once the developer has ensured the CI builds are working fine, and also Staging environment is working fine).

To allow for silent deployments from dev machine to the Production server, follow the steps below:

**On Dev machine:**

Generate SSH Key Pair on Dev Machine (if ~/.ssh/id\_rsa.pub does not already exist)

1. **$ssh-keygen -t rsa -b 4096 -C "Some Comment to identify this key"**

This will generate the private key at **~/.ssh/id\_rsa** and public key at **~/.ssh/id\_rsa.pub**

**On Production Server:**

Let the server know about the dev machine’s public key, so server can authenticate deployments automatically

1. **$su node** #Login to production server using the deployment user 'node'
2. **$ssh-keygen -t rsa** #create the **~/.ssh** folder (ignore the generated keys), use defaults and no paraphrase
3. Edit the **~/.ssh/authorized\_keys** file and append the contents of the public key that was generated on the Dev machine

You should now be able to deploy to the production server from this dev machine without any prompts

Staging:

On the Staging Server, we **do** want to allow silent deployments of successful builds directly from the CI server (CircleCI). For that to happen, follow the steps below:

**On Any Mac or Linux Machine:**

This is an optional step. If we already have Public/Private key pair for Circle CI generated earlier, we can just reuse them. Otherwise, proceed to generate a new pair as per steps below.

1. **$ssh-keygen -t rsa -b 4096 -C "Some Comment to identify this key"**

This will generate the private key at **~/.ssh/id\_rsa** and public key at **~/.ssh/id\_rsa.pub**

1. Store both the public and private keys in a secure location
2. Remove the generated keys from the machine on which they were generated (as they are no longer needed)

**On Circle CI portal:**

Add the private key to the Circle CI system to "identify" itself with. This will be used by the staging server to know that it’s talking to the Circle CI system and to approve silent deployments.

1. Login to the **Circle CI account > Project > Settings > Permissions > SSH Permissions**
2. Add the Private Key and Save (you can leave the host name empty if you want to use across multiple hosts)

**On Staging Server:**

Configure the Staging Server to know the public key of Circle CI so it can authenticate the incoming SSH requests from Circle CI for deployment.

1. Login to EC2 machine using the same user that Circle CI will be using to deploy (**'node'**)
2. Edit the **~/.ssh/authorized\_keys** file (create if not present)
3. Copy paste the Public Key (append to the file) and save. Make sure to copy the complete key including the leading **ssh-rsa** text

Set up SSH Keys to Allow Silent Cloning of Code from Github

Both Production and Staging servers need to be able to clone source code from Github without user prompts, and for that we need to set up SSH keys for Github.

Refer:

<https://help.github.com/articles/which-remote-url-should-i-use/> <https://help.github.com/articles/connecting-to-github-with-ssh/>

**On Any Mac or Linux Machine:**

This is an optional step. If we already have Public/Private key pair for Github generated earlier, we can just reuse them. Otherwise, proceed to generate a new pair as per steps below.

1. **$ssh-keygen -t rsa -b 4096 -C "Some Comment to identify this key"**

This will generate the private key at **~/.ssh/id\_rsa** and public key at **~/.ssh/id\_rsa.pub**

1. Store both the public and private keys in a secure location
2. Remove the generated keys from the machine on which they were generated (as they are no longer needed)

**On Github.com account (**[**ezeeideas@gmail.com)**](mailto:ezeeideas@gmail.com))

Add the Public key to the ezeeideas@gmail.com Github account. This is so that Github can authenticate the incoming clone requests from the server.

1. Go to (**Settings > SSH and GPG Keys > SSH Keys > New SSH Key**) and paste the public key data

**On Server (either Production or Staging)**

Now configure the Privacy SSH key for Github on the server that needs to perform the silent cloning, as per steps below:

1. Set the ‘**node’** deployment user (or the user that needs to clone)
2. Create a file called **$HOME/.ssh/id\_rsa\_github**, and copy the contents of the Private Key into this file

Note: Make sure to copy the entire key, including the BEGIN and END markers

1. **$chmod 400 $HOME/.ssh/id\_rsa\_github** #set reduced permissions on the file
2. Create a file called **$HOME/.ssh/config**, and add the below to the file:

**host github.com**

**HostName github.com**

**IdentityFile ~/.ssh/id\_rsa\_github**

**User git**

1. All Set! You should now be able to clone the repository using the below command:

**git clone git@github.com:ezeeideas/delphinus.git**

Triggering Deployment Setup (from Dev Machine)

Once all the previous steps have been done, it’s important to set up the deployment on the server. What this does is sets up the necessary directories on the server (by invoking setup\_dirs.sh command, and also clones the source code).

1. Ensure **ecosystem.config.js** is available and accurate
2. Ensure that the server doesn’t already have the ‘source’ folder **(/var/www/appserver/production/source or /var/www/appserver/staging/source**). Delete these if needed before proceeding.
3. Ensure you don’t have any code changes pending commit. Check-in any open changes and push to github.
4. Trigger deployment setup

**$pm2 deploy ecosystem.config.js production setup** #for production

or

**$pm2 deploy ecosystem.config.js staging setup** #for staging

Note: You can also just invoke **/deployment/scripts/setup\_prod.sh** or **/deployment/scripts/setup\_stage.sh** which does the same thing

1. Log into the server to check that the setup was successful. All relevant folders should have been created, and the **/var/www/appserver/production/source** or **/var/www/appserver/staging/source** folder should have been created and populated with the source from Github.

Triggering actual Deployment (from Dev Machine)

Once the deployment is all set up, a good way to test it would be to trigger it. Make a code change on the dev machine, check it in, and then trigger the deployment commands below:

1. Ensure that the server is already set up – ensure the source code exists, and the deployment space is set up with the right directory structure.
2. Make a code change that you want to test. Check-it in and push to Github.
3. Trigger deployment

**$pm2 deploy ecosystem.config.js production** #for production

or

**$ pm2 deploy ecosystem.config.js staging** #for staging

Note: You can also just invoke **/deployment/scripts/deploy\_prod.sh** or **/deployment/scripts/deploy\_stage.sh** which does the same thing

1. Log into the server and ensure the latest check-in code is synced correctly to the server, and that the node server is running with the latest changes incorporated.

IMPORTANT TROUBLESHOOTING TIP: PM2 logs to the ~/.pm2/logs/ folder on the server, so in case the deployment fails for some reason (e.g, you see a screen like the one below which happens when Node server is not starting up), make sure to look at the logs to find out what the issue is. You can configure the path in ecosystem.config.js, but it’s not possible to specifcy separate folders for production versus staging, so just using the default location in ~/.pm2/logs for now.

Set up Run-time Settings on Server

There are a bunch of run-time environment variables that are used by the Application to successfully run, and these need to be set by going explicitly into the application and setting these for that particular environment.. These variables different between Stage, Production and development environments, and are confidential in nature so should **not** be checked in.

The application expects a file called ‘**.env**’ to be present in the **/src/server** directory, with the below format:

**HOSTNAME="http://localhost:8080"**

**#NEO4J AUTHENTICATION INFORMATION**

**NEO4J\_HOSTNAME="localhost:7474"**

**NEO4J\_USERNAME=neo4j**

**NEO4J\_PASSWORD=mypassword**

**#SOCIAL AUTHENTICATION TOKENS**

**FACEBOOK\_CLIENT\_ID=140197703347931**

**FACEBOOK\_CLIENT\_SECRET=1203425195b711a1cf9b1a278a9a6d58**

**TWITTER\_CLIENT\_ID=U7d82P2yFhZ0dD47zsOpvAZI1**

**TWITTER\_CLIENT\_SECRET=0suazDxP70uPgTkdWJbzIHoT4Xjd82XLmksxsKktvlaHUHy4r5**

**GOOGLE\_CLIENT\_ID=583238880111-chvdagjqp00ncf4rsqca9jhltp1qk2id.apps.googleusercontent.com**

**GOOGLE\_CLIENT\_SECRET=zVdNs0O--Q2Hv0ZgLbd608Ci**

Note that the above are sample values – you need to make sure the correct values are entered for each.. The Social authentication tokens can be either recovered from the secure location, or they can be copied from individual developer pages for Facebook, Twitter and Google.

Note: The Node App will fail to start if these are not configured properly!