1. **Create a place for the app**

sudo mkdir -p /opt/ProductionServer

**1. Why sudo for mkdir?**

* /opt is a **system directory** that belongs to the **root** user and is intended for installing optional or third-party software.
* Regular (non-root) users **do not have permission** to create folders directly under /opt.
* sudo temporarily gives you **root privileges**, allowing you to make system-level changes such as creating /opt/ProductionServer.

If you try without sudo:

mkdir /opt/ProductionServer

you’ll likely get:

mkdir: cannot create directory ‘/opt/ProductionServer’: Permission denied

**2. Why sudo without a password?**

This happens if:

* Your AWS EC2 Ubuntu account (e.g., ubuntu) is configured in /etc/sudoers with **passwordless sudo** privileges.
* In AWS, the default Ubuntu AMIs are set so that the ubuntu user can run sudo without entering a password — this is for convenience and automation in cloud environments.

You can confirm by checking:

sudo cat /etc/sudoers | grep ubuntu

You’ll see something like:

ubuntu ALL=(ALL) NOPASSWD:ALL

That NOPASSWD part means **no password is required** when running sudo commands.

1. Why -p in sudo?  
   It creates **all missing folders in the path** at once.

sudo chown ubuntu:ubuntu /opt/ProductionServer

**1. chown — Change Ownership**

* chown = **change owner** of a file or directory.
* The syntax is:

chown <user>:<group> <path>

* + **ubuntu** (first one) → the **user** that will own the directory.
  + **ubuntu** (second one) → the **group** that will own the directory.
  + /opt/ProductionServer → the folder whose ownership is being changed.

After this, the **ubuntu** user (your EC2 login user) will have full rights over /opt/ProductionServer.

**2. Why sudo is needed**

* /opt and its subfolders are **system-owned** by root by default.
* Changing ownership is a **privileged operation**, so you need sudo to do it.

**Example effect**

Before:

ls -ld /opt/ProductionServer

# drwxr-xr-x 2 root root 4096 Feb 14 10:00 /opt/ProductionServer

After:

ls -ld /opt/ProductionServer

# drwxr-xr-x 2 ubuntu ubuntu 4096 Feb 14 10:00 /opt/ProductionServer

Now you can create/edit files there **without** using sudo.

cd /opt/ProductionServer

git clone <https://github.com/JuliaF929/ProductionRepo.git>

This can be done by other ways if EC2 does not have enough permissions.  
For example:  
- By *scp*:  
scp = **secure copy after zipping at my local computer**.  
It’s a command to copy files from your computer **to** your EC2 instance (or the other way around) over the network, securely (like cp, but remote).  
- CI/CD

**Install runtime** (Node.js)

cd ProductionRepo/server

curl -fsSL https://deb.nodesource.com/setup\_18.x | sudo -E bash -

* Install Node.js setup script  
  **curl -fsSL ...**  
  Downloads a small script from the NodeSource website.
  + -f = fail if there’s an error.
  + -s = silent mode (don’t show download progress).
  + -S = show errors if any occur.
  + -L = follow redirects (in case the URL redirects somewhere else).
* **| sudo -E bash -**The pipe (|) sends the script directly into bash (Linux shell) to run it.  
  sudo = run with admin rights.  
  -E = keep your current environment variables.
* **What it does:**  
  It adds the NodeSource repository to your Ubuntu so you can install Node.js version **18.x** with the official packages.

sudo apt -y install nodejs

* **sudo = admin rights.**
* **apt install nodejs** = tells Ubuntu to install Node.js from the sources you just set up.
* **-y** = automatically answer “yes” to “Are you sure?” questions so it doesn’t stop and ask you.

**What it does:**  
Installs **Node.js** (JavaScript runtime) and **npm** (Node Package Manager) so you can run JavaScript server code on your EC2.

npm ci --omit=dev

* **npm ci** = clean install of all the dependencies listed in your package-lock.json.
* Removes node\_modules first.
* Ensures *exact* versions from package-lock.json are installed.
* **--omit=dev** = skip installing devDependencies (things needed only for development, not for production).
* **What it does:**  
  Downloads and installs all the libraries your app needs to run, but only the production ones.

1. **Run as a service**

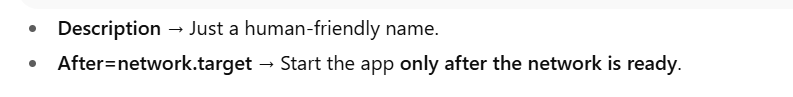
We want Node.js server app to **start automatically** when the EC2 instance turns on, and keep it running in the background even if I close the terminal.  
To do that, we make it a **Linux systemd service**.  
Telling Linux:  
“Here’s my app. Run it as the ubuntu user, keep it running forever, restart it if it stops, and start it automatically when the server boots.”

sudo tee /etc/systemd/system/ProductionServer.service >/dev/null <<'UNIT'

[Unit]

Description=ProductionServer Application

After=network.target



[Service]

User=ubuntu

WorkingDirectory=/opt/ProductionServer/ProductionRepo/server

ExecStart=/usr/bin/node /opt/ProductionServer/ ProductionRepo/server/server.js

Restart=always

Environment=NODE\_ENV=production

A screenshot of a chat

AI-generated content may be incorrect.

[Install]

WantedBy=multi-user.target



UNIT

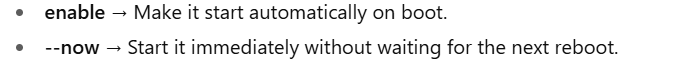
This creates the service file:

* **sudo** → You need admin rights because you are writing to a system folder (/etc/systemd/system).
* **tee** → This command writes whatever you type after it **into a file**.
* **/etc/systemd/system/myapp.service** → This is where systemd services are stored. You’re creating a file for your app’s service.
* **>/dev/null** → Hides the normal screen output (keeps your terminal clean).
* **<<'UNIT' ... UNIT** → This is a “here document” in bash. Everything you write between UNIT and UNIT will be saved into that file.

sudo systemctl daemon-reload

Tells systemd: “I’ve added/changed service files, please refresh.”

sudo systemctl enable --now ProductionServer



sudo systemctl status ProductionServer --no-pager



1. **Expose it safely**

This step is all about **making your app accessible on the web safely** — instead of exposing your app directly on port 3000, you put Nginx in front, open standard ports, and prepare for secure HTTPS in the future.  
We need to open the right port in your EC2 Security Group (like unlocking a door).  
Example:  
🡺Port 80 = normal web traffic (HTTP)  
🡺Port 443 = secure web traffic (HTTPS)  
🡺Or app’s own port (like 3000) if you want to test directly.  
So we can do one of the following:  
🡺Open the right port in the **Security Group** (e.g., 80/443 if using a proxy; or your app port temporarily).  
🡺Recommended: put **Nginx** in front as a reverse proxy (gets you clean ports 80/443 and easy TLS).

**Use Nginx as a “middleman”**

Instead of letting people connect directly to your app on port 3000, you put **Nginx** in front:

* Nginx listens on **port 80** (standard web)
* When someone visits your server, Nginx forwards that request to your app inside EC2 (e.g., on port 3000).
* This is called a **reverse proxy**.
* Benefit:
  + You get clean URLs without “:3000”
  + Easier to add HTTPS later
  + Can handle more traffic efficiently

sudo apt -y install nginx

sudo tee /etc/nginx/sites-available/ProductionServer >/dev/null <<'NGINX'

server {

listen 80;

server\_name \_; # replace with your domain later

location / {

proxy\_pass http://127.0.0.1:5000; # MY app port

proxy\_set\_header Host $host;

proxy\_set\_header X-Real-IP $remote\_addr;

}

}

NGINX

sudo ln -s /etc/nginx/sites-available/ProductionServer /etc/nginx/sites-enabled/

sudo nginx -t && sudo systemctl reload nginx

