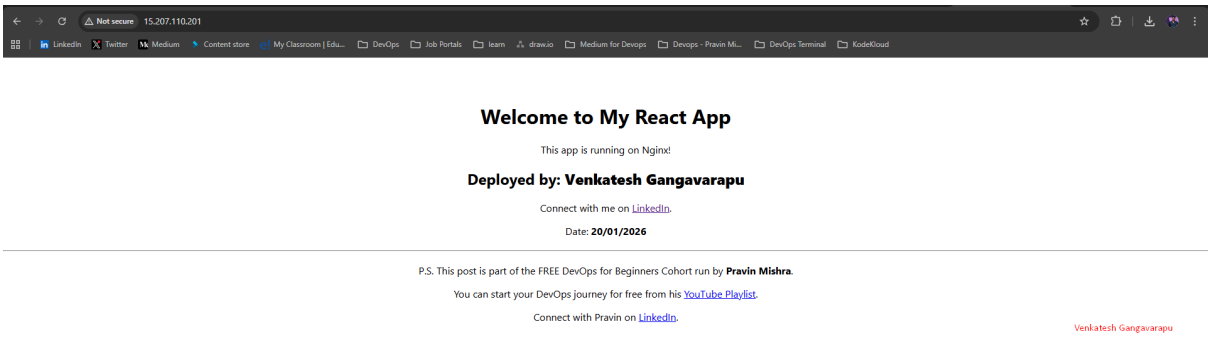
Production Maintenance Drill (OPS Checklist)

1. **Server Access - Traffic Verification:**



- Is it a serving?
  - Yes

- What proves it?
  - The React application is accessible via the public IP (http://15.207.110.201/) in the browser, and the UI loads successfully displaying my Full Name as Updated in "App.js". This confirms that the EC2 instance is reachable, Nginx is running, and traffic is being served correctly.

- If not, what would be the first suspicion?
  - My first suspicion would be:
    1. Nginx service not running or misconfigured.
    2. Security Group inbound rule missing port 80 (HTTP) access.

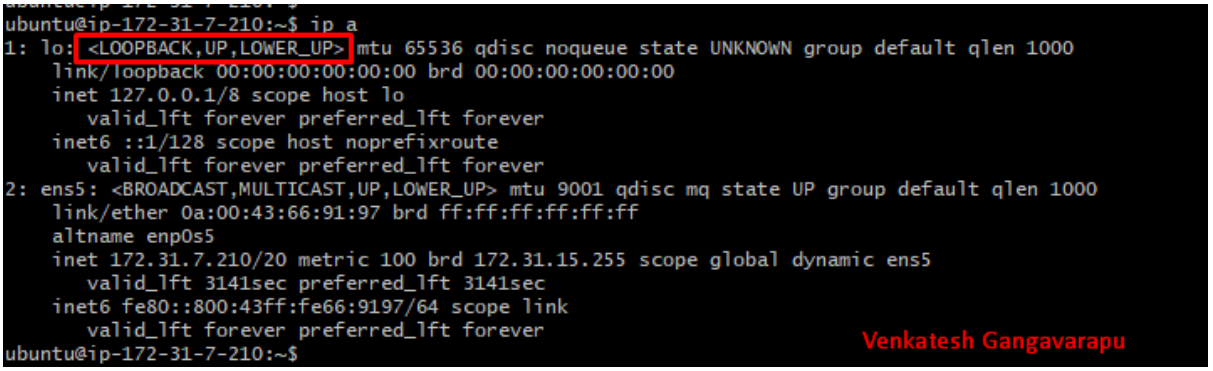
Assignment Tasks

PHASE 1: NETWORKING & ACCESS CHECKS (PRODUCTION BASICS):

This phase validates that the server is reachable, networked correctly, and exposing only intended services.

1. **IP and Network Interfaces:**

```
$ ip a
```

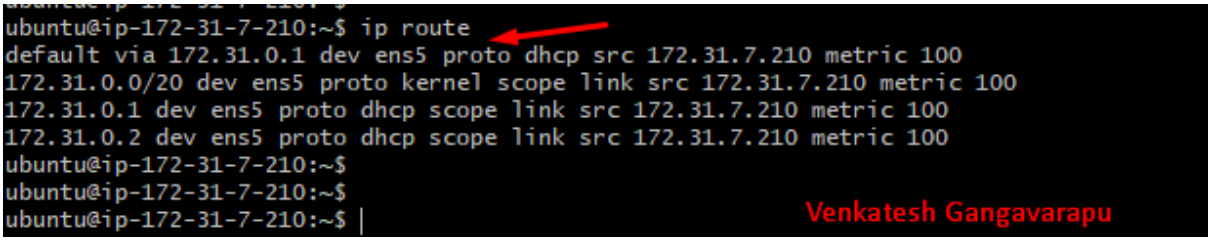


- What it means:
  - This command shows all network interfaces and their assigned IP addresses. From the output, I confirmed that the primary network interface is UP and has valid private IP assigned.

- Why it matters in production:
  - If the interface is down or misconfigured, the server cannot send or receive traffic, making the application completely unreachable.

2. Default Route (Gateway Check):

```
$ ip route
```



- What it means:
  - This output shows the default gateway used to reach external networks. The presence of a default route confirms that the server knows where to send traffic destined for the internet.

- Why it matters in production:
  - Without a correct default route, outbound traffic (updates, APIs, DNS, package installs) will fail, breaking deployments and monitoring.

3. DNS Resolution Check:

```
$ dig pravinmishra.com +short
```

```
ubuntu@ip-172-31-7-210:~$ dig pravinmishra.com +short
18.66.41.52
18.66.41.78
18.66.41.14
18.66.41.97
ubuntu@ip-172-31-7-210:~$
ubuntu@ip-172-31-7-210:~$
```


(or)

```
$ host pravinmishra.com
```

```
ubuntu@ip-172-31-7-210:~$ host pravinmishra.com
pravinmishra.com has address 18.66.41.78
pravinmishra.com has address 18.66.41.52
pravinmishra.com has address 18.66.41.97
pravinmishra.com has address 18.66.41.14
pravinmishra.com mail is handled by 1 smtp.google.com.
ubuntu@ip-172-31-7-210:~$
ubuntu@ip-172-31-7-210:~$
```


- What it means:
  - This confirms that DNS resolution is working and domain names can be resolved to IP addresses successfully.

- Why it matters in production:
  - If DNS fails, applications cannot reach external services, APIs, or dependencies, even if the network itself is up.

4. **Connectivity Test (Packet-Level):**

```
$ ping -c 4 thecloudadvisory.com
```

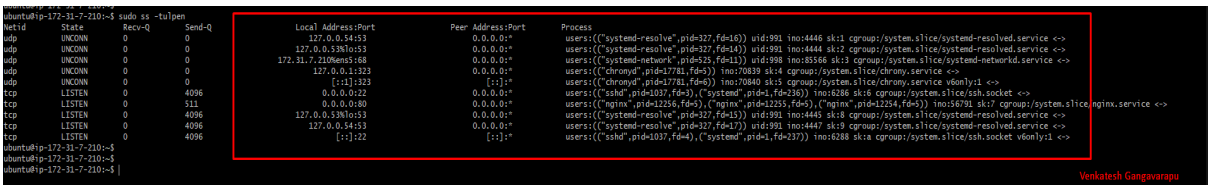
```
ubuntu@ip-172-31-7-210:~$ ping -c 4 thecloudadvisory.com
PING thecloudadvisory.com (3.169.71.43) 56(84) bytes of data.
64 bytes from server-3-169-71-43.tlv55.r.cloudfront.net (3.169.71.43): icmp_seq=1 ttl=246 time=130 ms
64 bytes from server-3-169-71-43.tlv55.r.cloudfront.net (3.169.71.43): icmp_seq=2 ttl=246 time=130 ms
64 bytes from server-3-169-71-43.tlv55.r.cloudfront.net (3.169.71.43): icmp_seq=3 ttl=246 time=130 ms
64 bytes from server-3-169-71-43.tlv55.r.cloudfront.net (3.169.71.43): icmp_seq=4 ttl=246 time=130 ms

--- thecloudadvisory.com ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3004ms
rtt min/avg/max/mdev = 130.035/130.039/130.043/0.003 ms
ubuntu@ip-172-31-7-210:~$
```


- What it means:
  - This verifies basic network connectivity by sending ICMP packets and receiving responses, confirming outbound reachability.

- Why it matters in production:
  - If ping fails, it may indicate network blocks, routing issues, or firewall problems that can prevent the app from communicating externally.

5. **Listening Ports and Owning Processes:**

```
$ sudo ss -tulpen
```



- What it means:
  - This command lists all listening ports along with the processes that own them. From the output, I verified which services are exposed on the server.

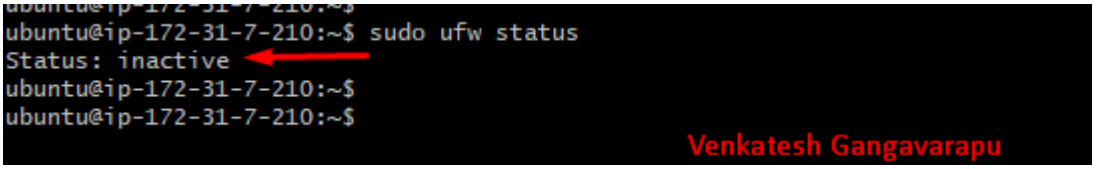
- Why it matters in production:
  - Unexpected open ports increase the attack surface.
  - Knowing exactly what is listening helps prevent security incidents.

- Key Observations:

  - Nginx listening on 0.0.0.0:80
  - SSH listening on port 22
  - No unexpected open ports

6. Firewall Status Check:

```
$ sudo ufw status
```



- What it means:
  - This shows whether a host-level firewall is active and enforcing rules.

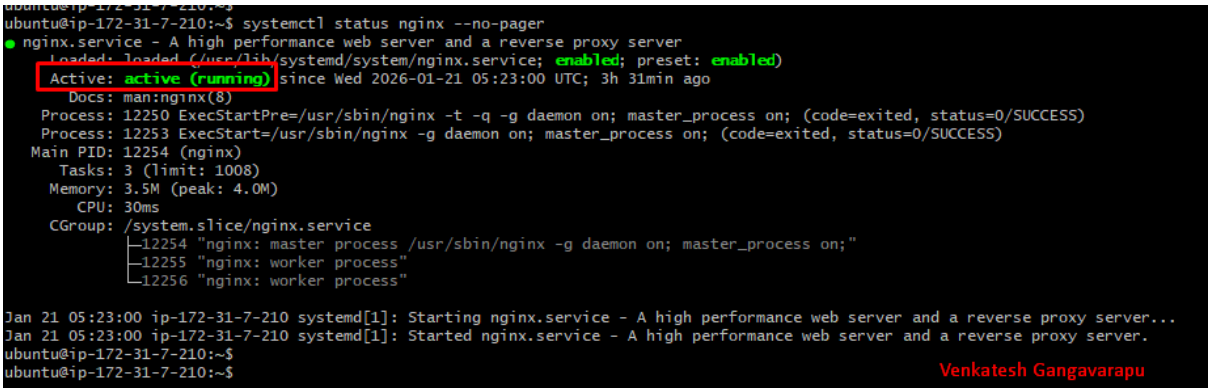
- Why it matters in production:
  - Even in cloud environments, host-level firewalls provide an additional security layer.
  - Knowing its status prevents false assumptions during incidents

PHASE 2: SERVICE HEALTH + PROCESS VALIDATION (SYSTEMD STYLE)

This phase validates whether the service is not only running, but also manageable, restartable, and production-safe using systemd-style operational checks.

1. Service Status Check:

```
$ systemctl status nginx --no-pager
```

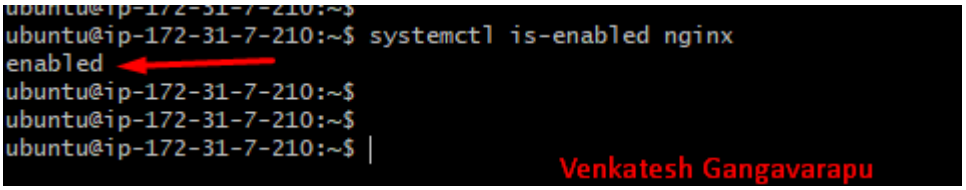


- What it means:
  - This command shows the current state of the Nginx service, including whether it is active, running and free of immediate errors.

- Why it matters in production:
  - If a service is not in an **“active (running)”** state, traffic will fail.
  - This is the first check during any incident or on-call alert.

2. Service Enabled on Boot (Reliability Check)

```
$ systemctl is-enabled nginx
```

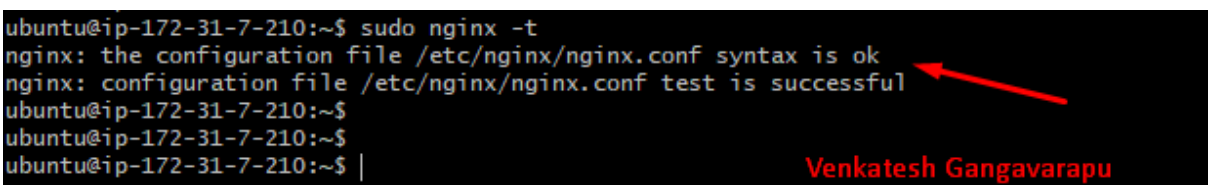


- What it means:
  - This confirms whether Nginx is configured to start automatically on system boot.

- Why it matters in production:
  - If a server reboots and the service is not enabled, the application will remain down until manual intervention, causing prolonged downtime.

3. Nginx Configuration Validation:

```
$ sudo nginx -t
```



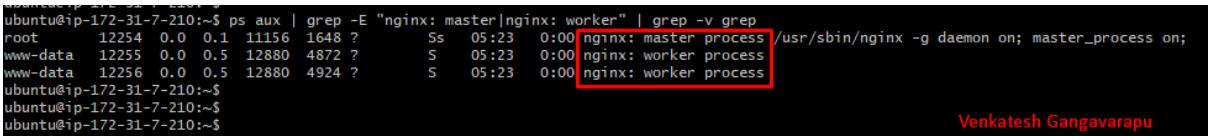
- What it means:

- This checks the syntax and validity of the Nginx configuration files without restarting the service.

- Why it matters in production:
  - Invalid configurations can prevent Nginx from starting after a restart, leading to a full production outage.

4. Master and Worker Process Identification:

```
$ ps aux | grep -E "nginx: master|nginx: worker" | grep -v grep
```

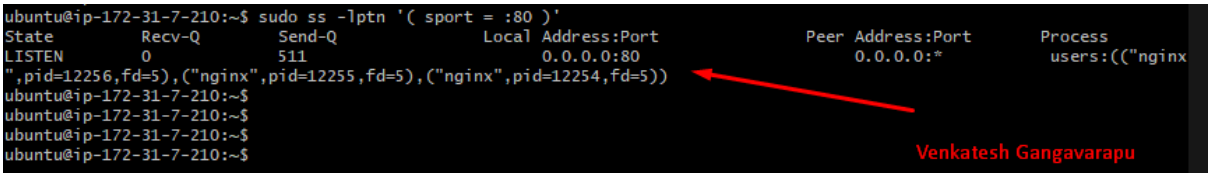


- What it means:
  - This confirms that the Nginx master process and worker processes are running as expected.

- Why it matters in production:
  - If worker processes are missing or unstable, Nginx may accept connections but fail to serve traffic properly.

5. Confirm Process Listening on Port 80:

```
$ sudo ss -lptn '( sport = :80 )'
```

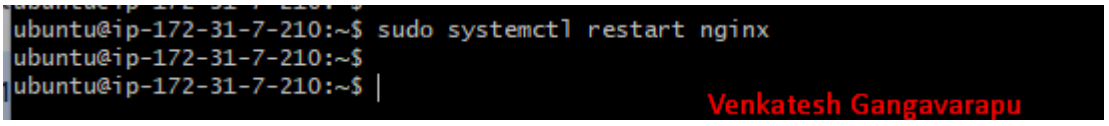


- What it means:
  - This identifies which process and PID are actively listening on port 80.

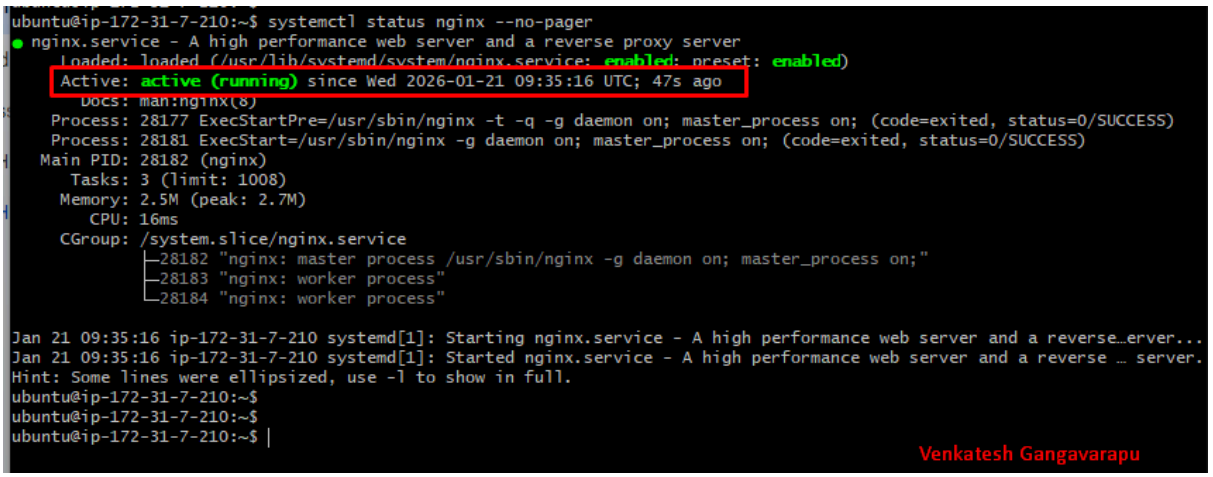
- Why it matters in production:
  - If the wrong process is bound to port 80 or no process is listening, users will not be able to access the application.

6. Safe Restart Drill (Routine Deploy Simulation)

```
$ sudo systemctl restart nginx
```




```
$ systemctl status nginx --no-pager
```



- What it means:
  - This simulates a routine restart such as during a deployment or configuration change and verifies that the service comes back cleanly.

- Why it matters in production
  - Services are restarted frequently during releases.
  - If restarts are unsafe, every deployment becomes a risk.

Production Risk Assessment

- **What could happen if nginx restarts and doesn't come back?**
  - If Nginx fails to restart, the application becomes completely unavailable, leading to immediate user impact, failed health checks, and potential SLA violations.

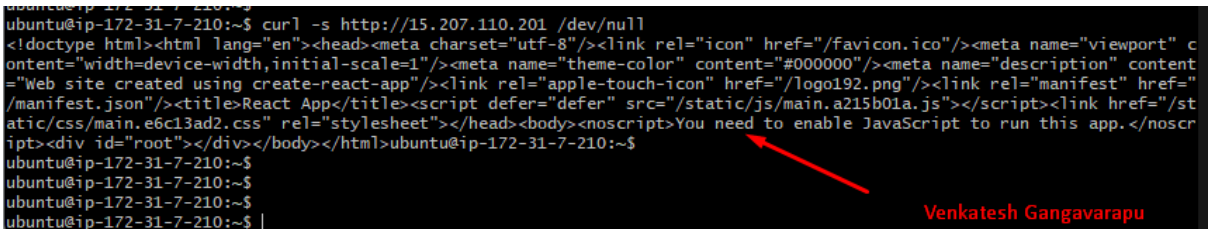
- **What is the rollback plan (basic)?**
  - Revert the last configuration change
  - Validate configuration using `nginx -t`
  - Restart Nginx again
  - If required, restore the last known working config backup

PHASE 3: LOGS & REQUEST TRACE (REAL DEVOPS WORK)

This phase validates whether the system is "observable". In real production, "it's working" is not enough - logs are the source of truth to confirm clean traffic and detect hidden issues.

1. **Generate Traffic (Trigger Log Entries):**

```
$ curl -s http://<public-ip> /dev/null
```




```
$ curl -s http://<public-ip> /dev/null
```

- What it means:
  - These commands generate HTTP requests to ensure fresh entries appear in the Nginx access and error logs

- Why it matters in production:
  - Logs are only useful if traffic is actually reaching the service.
  - This step confirms real requests are being processed.

2. Access Logs (Traffic Proof)

```
$ sudo tail -n 30 /var/log/nginx/access.log
```

- What it means:
  - The access log records every incoming request, including client IP, HTTP method, response code and user agent.

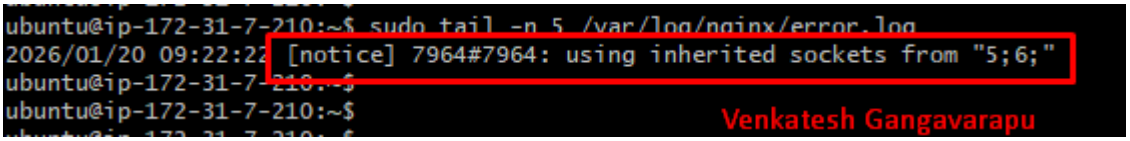
- Whey it matters in production:
  - Access logs confirm real traffic flow and help trace requests during incidents, performance issues, or security investigations.

- Key Observation:

- My `curl` requests are visible in the access log entries.

3. Error Logs (Failures, Misconfigurations and permission issues):

```
$ sudo tail -n 30 /var/log/nginx/error.log
```



- What it means:
  - The error log captures runtime issues such as permission problems, misconfigurations or failed upstream connections.
  - There is normal Nginx notice, indicating that Nginx restarted gracefully and reused existing sockets instead of dropping connections.

- Why it matters in production:
  - Many failures do not surface in the UI.
  - Error logs reveal silent problems that can later cause outages.
  - There is notice which is a clean, zero-downtime style restart. Notices are informational and not failures - no user traffic was impacted.

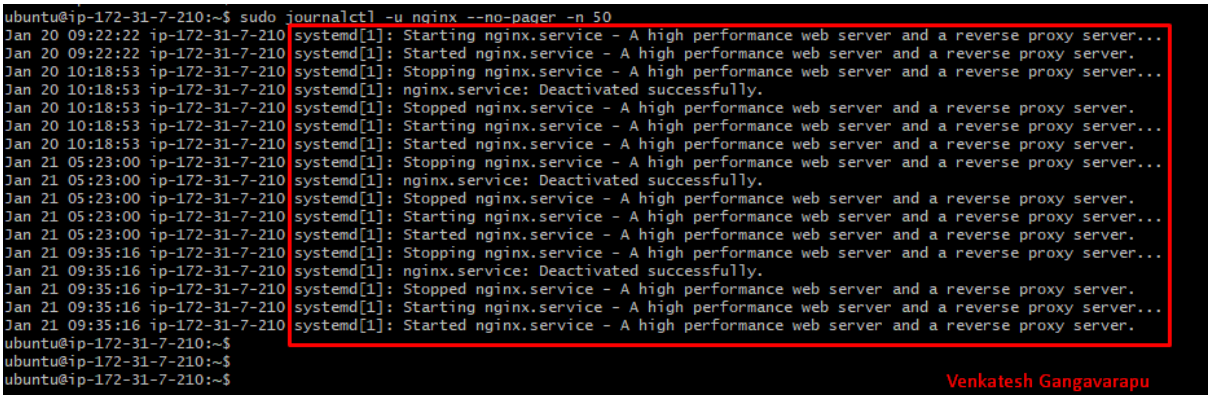
- Observation:
  - No errors were observed. Only a normal Nginx NOTICE related to socket inheritance during restart was present, which indicates a clean and graceful restart.

OPS rule of thumb:

- [error] - problem
- [warn] - potential risk
- [notice] - normal operational information

4. Systemd Journal (Service-Level logs):

```
$ sudo journalctl -u nginx --no-pager -n 50
```



- What it means:
  - This shows service-level logs managed by systemd, including restarts, failures and warnings.

- Why it matters in production:
  - Systemd logs help diagnose issues during service restarts, crashes or failed reloads.

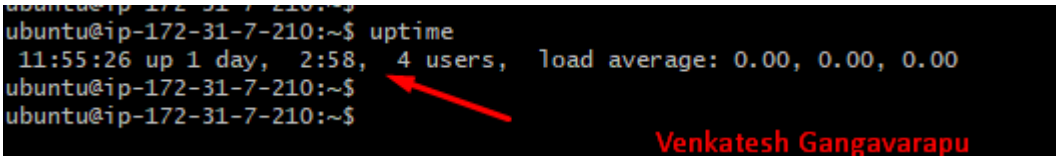
- Observation:
  - No recent failures or warnings related to Nginx were observed.

PHASE 4: SYSTEM RESOURCE HEALTH (CAPACITY RED FLAGS)

- This phase answers a critical production question:
  - Can this host safely handle traffic, or is it close to failure?
- The goal is to identify early warning signs before users are impacted.

1. Load Average & Uptime:

```
$ uptime
```

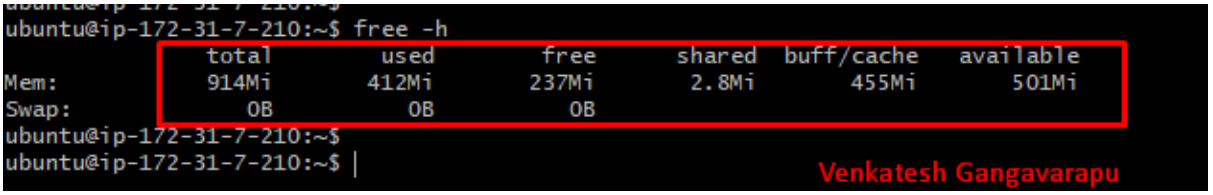


- What it means:
  - This shows system uptime and load averages over the last 1, 5, and 15 minutes, indicating how busy the system is.

- Why it matters in production:
  - High load relative to CPU cores indicates the system is overloaded, which can cause slow responses or service timeouts.

2. Memory Usage (Quick View):

```
$ free -h
```

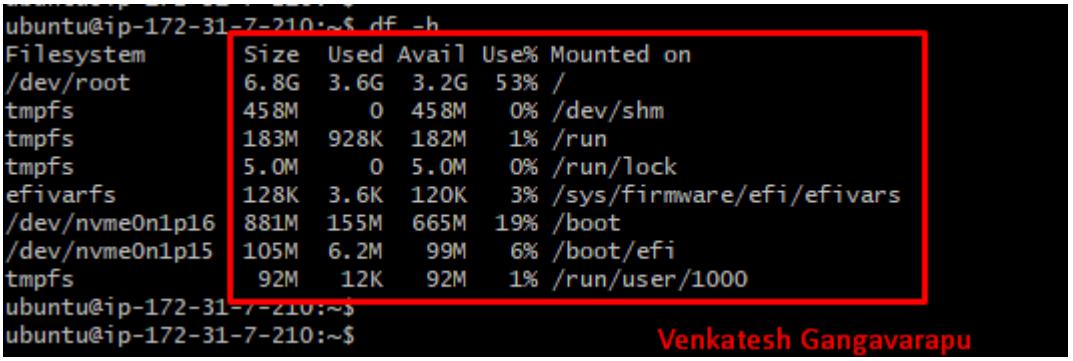


- What it means:
  - Displays total, used, free memory and swap usage in a human-readable format.

- Why it matters in production:
  - Low available memory or heavy swap usage can lead to application slowdowns or process termination by OOM killer.

3. **Disk Usage:**

```
$ df -h
```

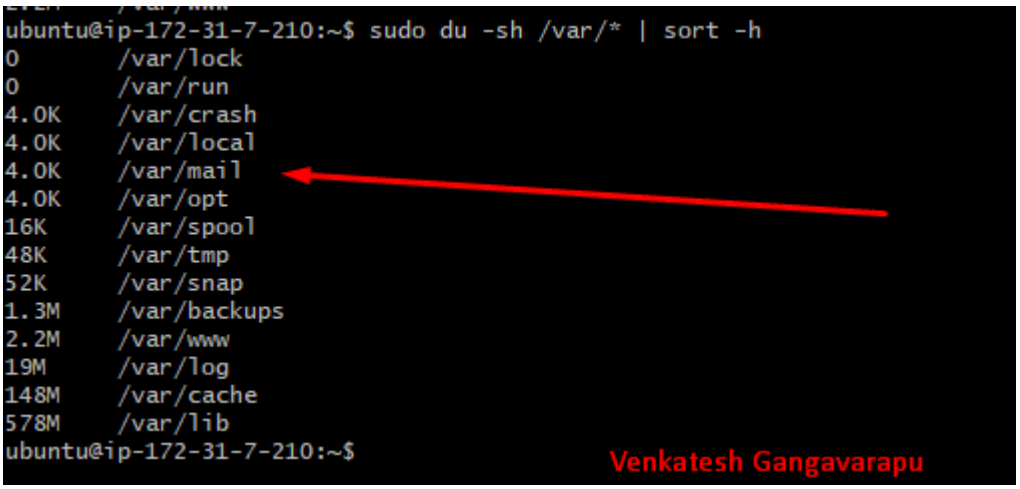


- What it means:
  - Shows disk space usage for all mounted filesystems.

- Why it matters in production:
  - Disks approaching full capacity (>80-90%) are a major risk.
  - Many services fail silently when disk space runs out.

4. **Largest Disk Consumers under `/var`:**

```
$ sudo du -sh /var/* | sort -h
```



- What it means:
  - Identifies which directories under `/var` are consuming the most disk space.
  - Logs are often the biggest contributors in production systems.

- Why it matters in production:
  - Unchecked log growth is one of the most common causes of production outages.

Capacity Risk Assessment:

- **Which resource looks most risky right now?**
  - Disk usage is the most critical resource to monitor because logs under `/var` can grow continuously and fill the filesystem if not rotated or monitored.

- What happens to production services when the disk fills up?
  - Applications fail to write logs or temp files
  - Nginx may stop serving requests
  - Databases can crash or enter read-only mode
  - Deployments and restarts fail

  This can lead to full production outages even if CPU and memory are healthy.

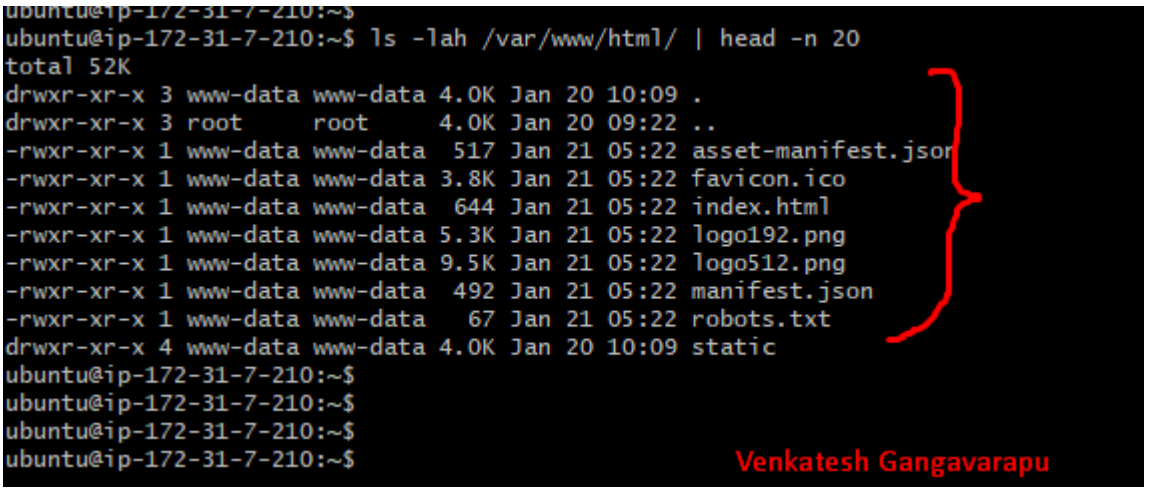
PHASE 5: CONFIGURATION & CONTENT INTEGRITY (RELEASE SAFETY)

This phase verifies that the “Correct build is deployed”, the “right content is being served”, and the “web server configuration matches” after a release.

Real production teams never assume a deployment succeeded, they only verify it.

1. **Confirm Correct Web Root Contains Build Files:**

```
$ ls -lah /var/www/html | head -n 20
```

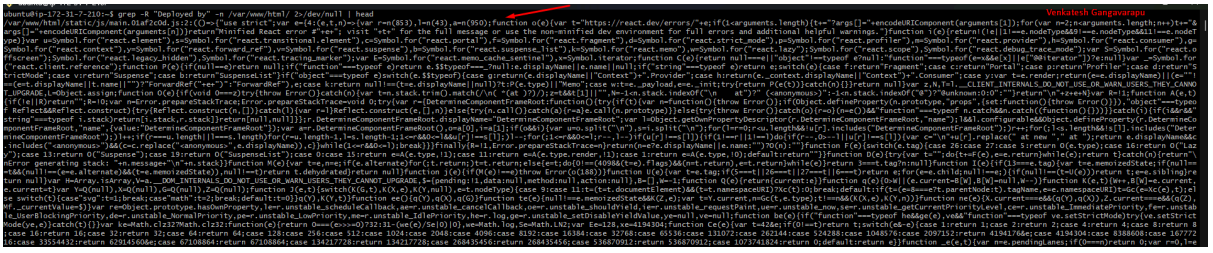


- What it means:
  - This lists the contents of the Nginx web root directory and confirms that the React build artifacts (such as `index.html` and `static`) are present.

- Why it matters in production:
  - If the wrong directory is serving content, users may see an old version, a default page, or broken assets after deployment.

2. Confirm Nginx Is Serving the Correct Build Content:

```
$ grep -R "Deployed by" -n /var/www/html 2>/dev/null | head
```

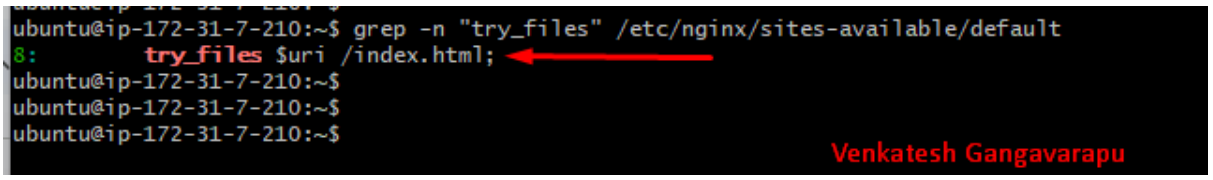


- Observation:
  - Confirms that the deployed files contain the expected application content.

- Why it matters in production:
  - Verifying content integrity prevents serving stale or incorrect versions after a release.

3. Confirm SPA Routing in Nginx Configuration:

```
$ grep -n "try_files" /etc/nginx/sites-available/default
```



- What it means:
  - This confirms that Nginx is configured to route all requests to “index.html”, which is required for React Single Page Applications.

- Why it matters in production:
    - Without proper SPA routing, page refreshes or deep links will return 404 errors, breaking user navigation.

Release Verification:

- **How do you know the correct version is deployed?**

  The correct version is confirmed by:
    - Presence of the latest build files in /var/www/html
    - Updated timestamps matching the recent deployment
    - Application UI displaying the correct deployment details
    - Nginx configuration correctly serving the SPA build

  This combination confirms the right code + right config + right content are live.

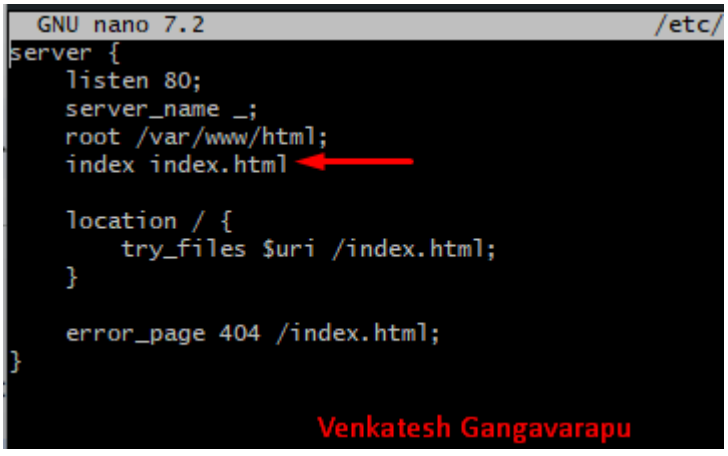
PHASE 6: MINI INCIDENT SIMULATION + RECOVERY (MANDATORY)

This phase simulates real production failures and validates safe recovery.
All actions were performed deliberately and restored fully.

Task A: Bad Nginx Configuration Deployed

Step 1: Introduce Failure

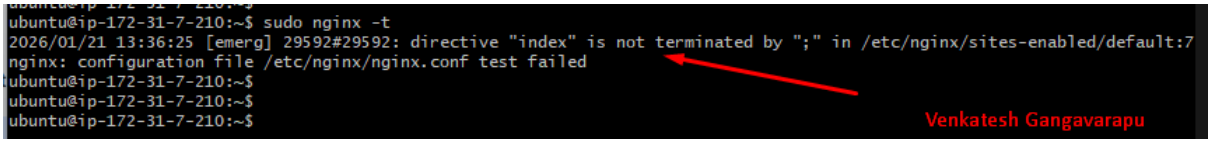
```
$ sudo nano /etc/nginx/sites-available/default
```



- A small syntax error was introduced (missing semicolon).

Step 2: Confirm config is broken by testing Nginx

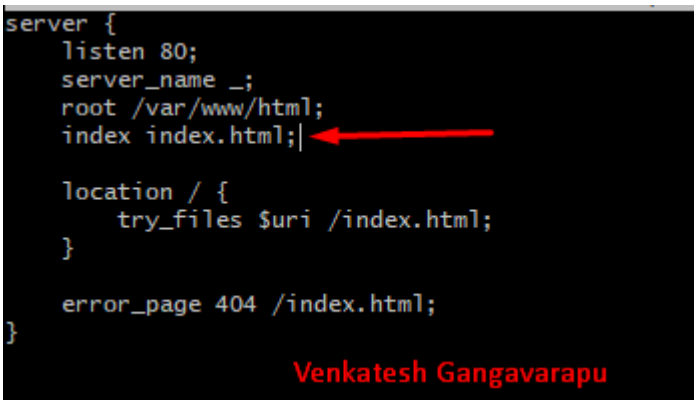
```
$ sudo nginx -t
```



- Configuration test failed.

Step 3: Fix the Config back

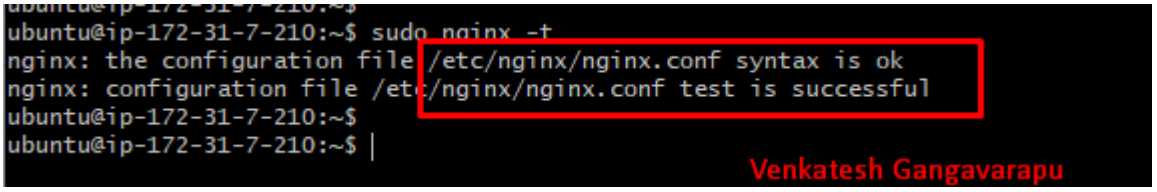
```
$ sudo nano /etc/nginx/sites-available/default
```



- Updated the semicolon again to save

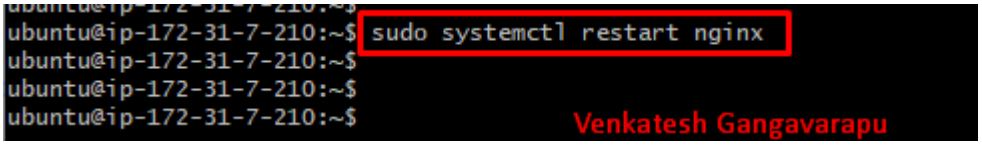
Step 4: Validate and Restart

```
$ sudo nginx -t
```



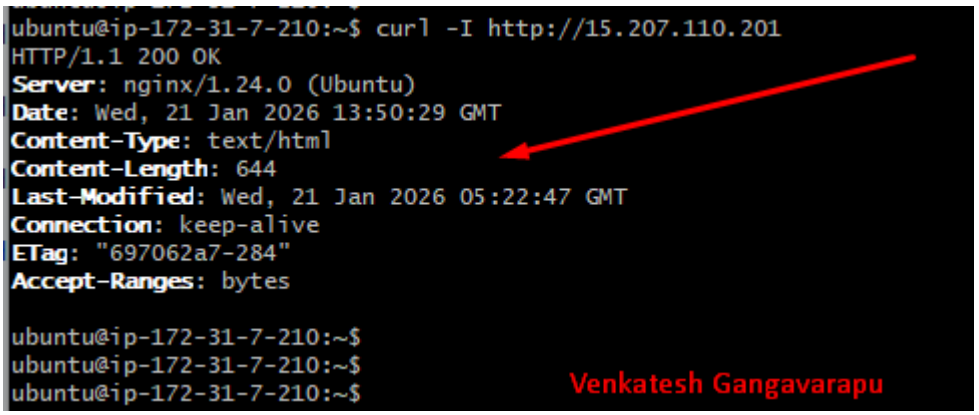
- Configuration test successful.

```
$ sudo systemctl restart nginx
```



- Nginx restarted.

```
$ curl -I http://15.207.110.201/
```



- HTTP response returned 200.

Root Cause:

- A Syntax error (missing semicolon) was introduced in Nginx configuration, causing nginx -t to fail.

Fix:

- Corrected configuration and validated it using nginx -t before restarting the service.

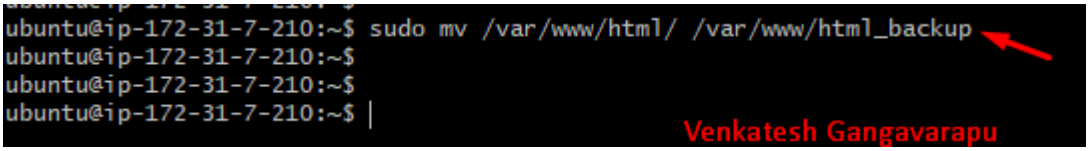
Prevention:

- Always validate configuration changes before reloads and keep a last-know-good configuration for quick rollback.

Task B: Web Content Missing After Deployment

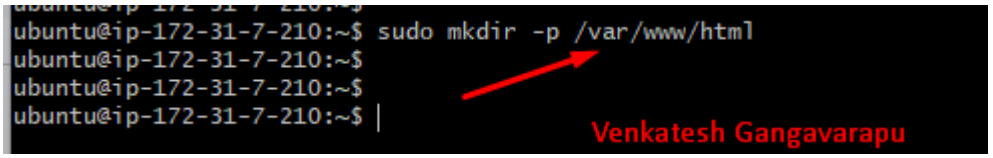
Step 1: Break Content (Safely)

```
$ sudo mv /var/www/html /var/www/html_backup
```



- Taking backup of the web working content for safety.

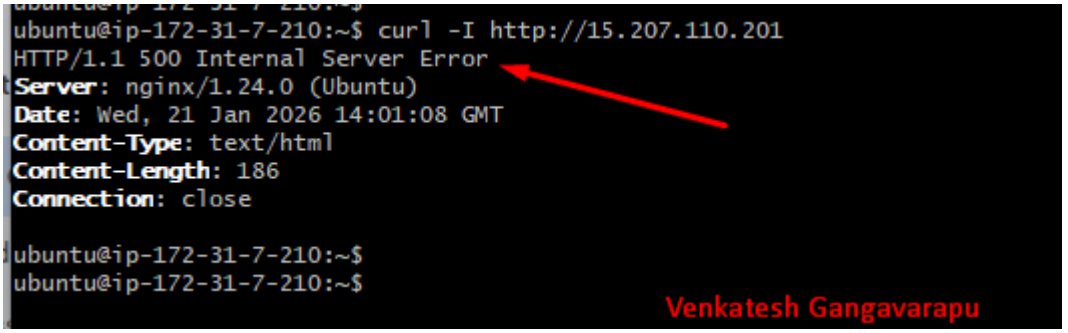
```
$ sudo mkdir -p /var/www/html
```



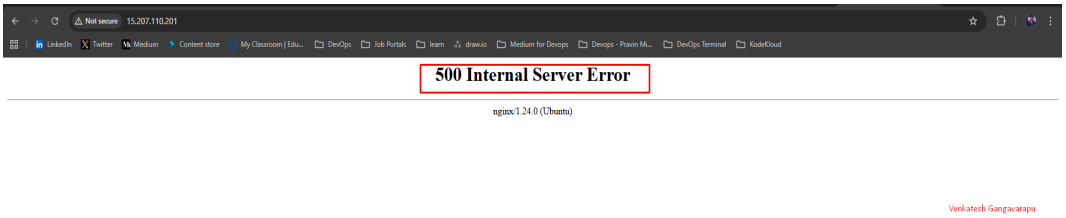
- Creating the new web directory

Step 2: Confirm failure

```
$ curl -I  http://15.207.110.201/
```



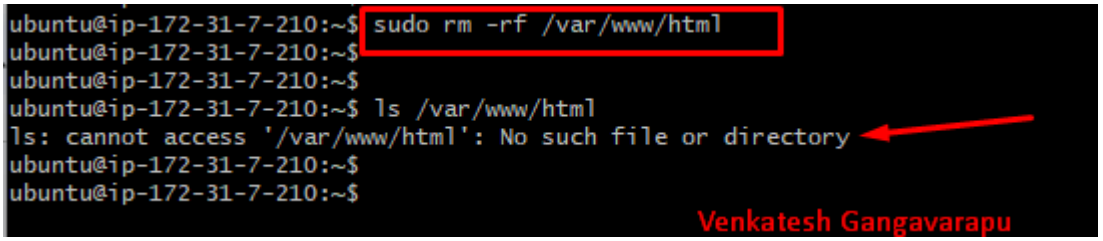
- Tested with curl and failed with 500 server error



- Browser returned 500 Internal Server Error.

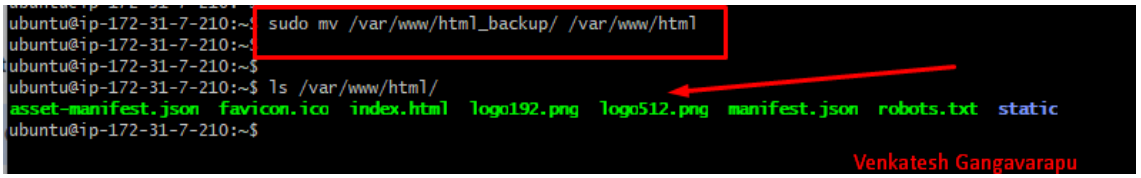
Step 3: Restore Content

```
$ sudo rm -rf /var/www/html
```



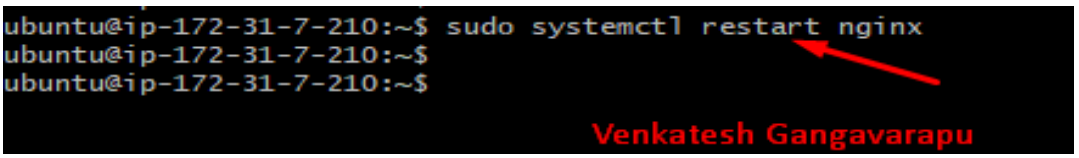
- Removed and confirmed the fault web directory.

```
$ sudo mv /var/www/html_backup /var/www/html
```



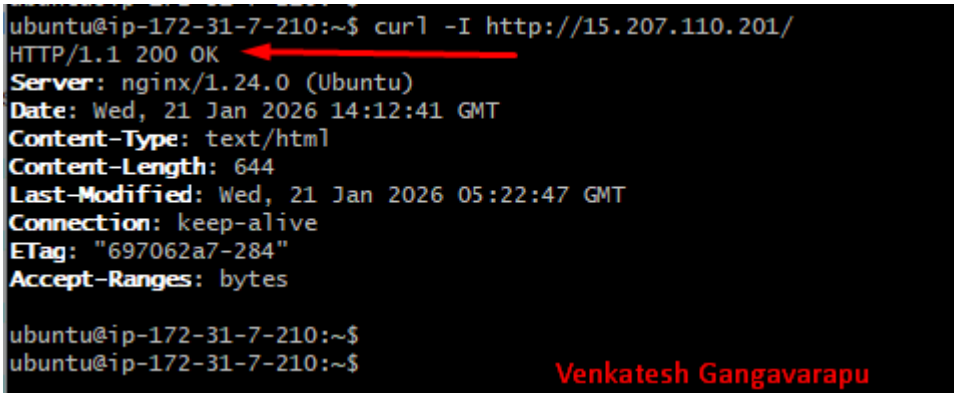
- Restored the old content and confirmed.

```
$ sudo systemctl restart nginx
```

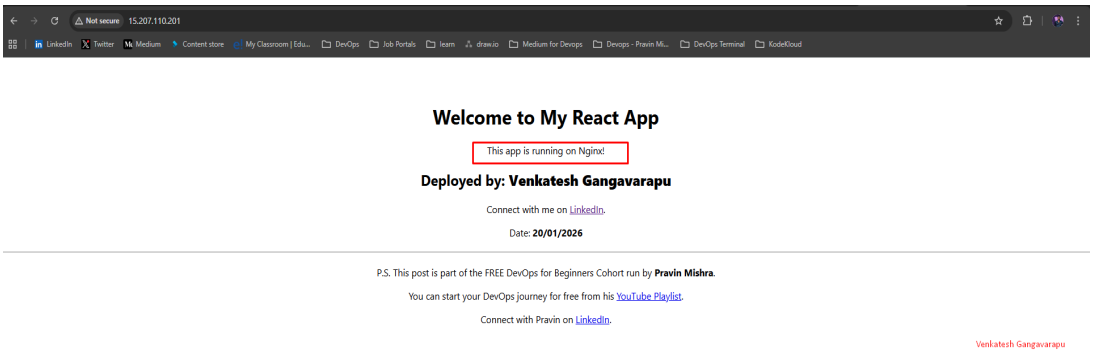


- Restarted the nginx server

```
$ curl -I http://15.207.110.201/
```



- Tested with curl and HTTP response returned 200



- Refreshed the browser and Nginx is running.

Root Cause:

- The web root directory (/var/www/html) was missing, so Nginx had no content to serve.

Fix:

- Restored the correct build files from backup and restarted Nginx to serve the application.

Prevention:

- Use atomic deployments with backups and verify deployment paths before exposing changes to users.

On-Call Notes (Thinking, Not Command Dumps)

- **What I checked first:**
  - I first checked the service status and configuration validity to determine whether the failure was service-related, configuration-related, or content-related before taking any corrective action.

- **What I concluded from the outputs:**
  - The failed `nginx -t` output confirmed a configuration error, not a service crash. In the content incident, the HTTP response indicated missing web content rather than a network or service failure. This allowed me to focus recovery on configuration and content restoration instead of unnecessary restarts or network changes.

Security & Reliability Notes

- SSH access to the server is secured using **key-based authentication**, and ensuring that no passwords are shared or stored insecurely.

- Only **required network ports** are exposed to the internet — **port 22 (SSH)** for access and **port 80 (HTTP)** for the application. No unnecessary ports are open.

- **Nginx is enabled on system boot**, ensuring the service automatically starts after a reboot and reducing the risk of downtime.

- No **secrets, credentials, or private keys** were committed to GitHub or shared publicly at any stage.

- Since this is a cloud-based VM, I will **stop or terminate resources when not in use** to avoid unexpected costs and maintain responsible resource usage.

REFLECTION

This week taught me that real production checks go far beyond “the app is running.” I learned how to validate systems layer by layer—network, service health, logs, resources, configuration, and recovery—just like an on-call engineer.

The biggest issue I faced was intentionally breaking Nginx configuration and web content to simulate incidents; the challenge was staying calm and diagnosing instead of blindly restarting services. I fixed it by validating configs (`nginx -t`), checking logs, and restoring known-good states step by step.

Next time, I will rely even more on logs and evidence before action, and I will validate every deployment with post-release checks instead of assuming success.