Step 5: Configure Reverse Proxy with Nginx

Deploy Spring Boot Apps to AWS Lightsail

2026-04-08

Table of contents

1 Overview

What We'll Cover:

- Understanding reverse proxy architecture
- Installing and configuring Nginx
- Advanced configuration options
- Testing and monitoring

Learning Objectives:

- Set up Nginx as reverse proxy
- Implement security headers
- Configure rate limiting
- Optimize performance

Today we'll learn how to configure Nginx as a reverse proxy for your Spring Boot application. This is a crucial step for production deployments that provides security, performance, and scalability benefits.

---

2 Why Use a Reverse Proxy?

Performance Benefits: - Serve static content directly - Cache responses - Gzip compression - Load balancing capability

Security Benefits: - Hide application server details - SSL termination - Request filtering - Security headers

While Spring Boot has an embedded Tomcat server, using Nginx as a front-end proxy is a production best practice. It acts as an intermediary between clients and your application, providing multiple layers of optimization and security.

---

3 Reverse Proxy Architecture

```
graph LR
    A[Client] --> B[Nginx :80]
    B --> C[Spring Boot :8080]
    B --> D[Static Files]
    B --> E[Cache]
```

Flow: Client → Nginx (Port 80) → Spring Boot (Port 8080)

This diagram shows how requests flow through our architecture. Nginx receives all client requests on port 80, then either serves static content directly or forwards dynamic requests to Spring Boot running on port 8080.

---

4 Install Nginx

```
# Update package lists
sudo apt update

# Install Nginx
sudo apt install nginx -y

# Check if Nginx is running
sudo systemctl status nginx

# Enable Nginx to start on boot
sudo systemctl enable nginx
```

First, we need to install Nginx on our Lightsail instance. These commands will install Nginx, check its status, and ensure it starts automatically when the server reboots.

---

5 Configure Firewall

```
# Allow HTTP traffic (port 80)
sudo ufw allow 'Nginx HTTP'

# Allow HTTPS traffic (port 443)
sudo ufw allow 'Nginx HTTPS'

# Check firewall status
sudo ufw status
```

Don't forget: Also open ports 80 and 443 in Lightsail's Networking tab!

Security is crucial, so we need to properly configure the firewall to allow HTTP and HTTPS traffic. Remember to also check your Lightsail instance's networking settings in the AWS console.

---

6 Create Nginx Configuration

```
# Create configuration file
sudo nano /etc/nginx/sites-available/spring-boot-app
```

Key Configuration Sections: - Server block and domain settings - Security headers - Gzip compression - Proxy settings

Now we'll create a custom Nginx configuration specifically for our Spring Boot application. This configuration will include several important sections for security, performance, and proper request forwarding.

---

7 Basic Server Configuration

```
server {
    listen 80;
    server_name your-domain.com www.your-domain.com;

    # Security headers
    add_header X-Frame-Options "SAMEORIGIN" always;
    add_header X-Content-Type-Options "nosniff" always;
    add_header X-XSS-Protection "1; mode=block" always;

    # Main application proxy
    location / {
        proxy_pass http://localhost:8080;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

This is the basic server configuration that listens on port 80, includes essential security headers, and forwards all requests to our Spring Boot application running on port 8080. The proxy headers ensure your application receives proper client information.

---

8 Enable Gzip Compression

```
# Gzip compression
gzip on;
gzip_vary on;
gzip_min_length 1024;
gzip_proxied any;
gzip_comp_level 6;
gzip_types text/plain text/css text/xml
           text/javascript application/javascript
           application/xml+rss application/json;
```

Benefits: Reduces bandwidth usage by up to 70%

Gzip compression significantly reduces the size of responses sent to clients, improving load times and reducing bandwidth costs. This configuration compresses various file types while avoiding very small files where compression overhead isn't worth it.

---

9 Static Content Optimization

```
# Static content location
location /static/ {
    alias /home/ubuntu/app/static/;
    expires 1y;
    add_header Cache-Control "public, immutable";
}

# Optimize static assets
location ~* \.(js|css|png|jpg|jpeg|gif|ico|svg)$ {
    expires 1y;
    add_header Cache-Control "public, immutable";
    access_log off;
}
```

For better performance, we can configure Nginx to serve static content directly instead of forwarding these requests to Spring Boot. This includes setting appropriate cache headers to reduce server load.

---

10 Enable the Configuration

```
# Create symbolic link to enable the site
sudo ln -s /etc/nginx/sites-available/spring-boot-app \
           /etc/nginx/sites-enabled/

# Remove default Nginx site
sudo rm /etc/nginx/sites-enabled/default
```

```
# Test Nginx configuration
sudo nginx -t

# Reload Nginx to apply changes
sudo systemctl reload nginx
```

After creating our configuration, we need to enable it by creating a symbolic link, remove the default site, test our configuration for syntax errors, and reload Nginx to apply the changes.

---

11 Implement Rate Limiting

```
# Add to /etc/nginx/nginx.conf http block
limit_req_zone $binary_remote_addr zone=login:10m rate=5r/m;
limit_req_zone $binary_remote_addr zone=api:10m rate=10r/s;
limit_req_zone $binary_remote_addr zone=general:10m rate=1r/s;

# In server block
location /api/auth/login {
    limit_req zone=login burst=3 nodelay;
    proxy_pass http://localhost:8080;
}
```

Protection against: DDoS attacks, brute force attempts, API abuse

Rate limiting is crucial for protecting your application from abuse. We can set different limits for different endpoints - stricter limits for login attempts and more generous limits for general API usage.

---

12 Custom Error Pages

```
# Custom error pages
error_page 502 503 504 /50x.html;
location = /50x.html {
    root /var/www/html;
}

# Health check endpoint (bypass proxy for quick responses)
location /actuator/health {
    proxy_pass http://localhost:8080;
    proxy_connect_timeout 2s;
    proxy_read_timeout 2s;
}
```

Professional applications need proper error handling. We can configure custom error pages and set up dedicated health check endpoints that respond quickly for monitoring systems.

---

13 Testing Your Configuration

Server Tests:

```
# Check Nginx status
sudo systemctl status nginx

# Verify port listening
sudo netstat -tlnp | grep :80

# Test locally
curl -I http://localhost
```

Browser Tests: - Navigate to public IP - Check developer tools for headers - Verify application loads correctly - Test static content serving

Testing is crucial to ensure everything works correctly. We should test both from the server itself and from external browsers to verify that requests are being properly forwarded and responses include our security headers.

---

14 Troubleshooting Common Issues

- **502 Bad Gateway** → Spring Boot app not running on port 8080
- **Connection refused** → Firewall blocking ports 80/443
- **Configuration errors** → Run `sudo nginx -t` to validate
- **Performance issues** → Check logs and system resources

Log Locations: - `/var/log/nginx/access.log` - `/var/log/nginx/error.log`

When things go wrong, these are the most common issues you'll encounter. Always check the logs first, and remember that Nginx configuration validation can catch syntax errors before they cause problems.

---

15 Monitoring and Maintenance

```
# Set up log rotation
sudo nano /etc/logrotate.d/nginx-spring-boot

# Monitor performance
ps aux | grep nginx
curl http://localhost/nginx_status
htop
```

Best Practices: - Regular log rotation - Monitor system resources - Track response times - Set up alerting

Ongoing maintenance is important for production systems. Set up log rotation to prevent disk space issues, monitor system resources, and consider implementing alerting for critical issues.

---

16 Security Headers Explained

```
X-Frame-Options: "SAMEORIGIN"
X-Content-Type-Options: "nosniff"
X-XSS-Protection: "1; mode=block"
Referrer-Policy: "no-referrer-when-downgrade"
```

Protection Against: - Clickjacking attacks - MIME type sniffing - Cross-site scripting - Information leakage

Security headers are essential for protecting your application against common web vulnerabilities. Each header serves a specific purpose in hardening your application against attacks.

---

17 Performance Optimization Tips

{.incremental} - Enable gzip compression for text content - Set appropriate cache headers for static assets - Use separate location blocks for different content types - Implement connection keep-alive for better performance - Configure proper timeout values - Monitor and tune worker processes

These optimization techniques can significantly improve your application's performance. The key is to understand your traffic patterns and configure Nginx accordingly.

---

18 Summary

What We Accomplished: - Installed and configured Nginx as reverse proxy - Implemented security headers and rate limiting - Optimized performance with compression and caching - Set up proper error handling and monitoring

Key Benefits: - Enhanced security and performance - Professional error handling - Production-ready architecture - Better monitoring and maintenance

We've successfully set up a professional-grade reverse proxy configuration that provides security, performance, and monitoring capabilities. Your Spring Boot application is now running behind a robust web server that can handle production traffic effectively.

---

19 Next Steps

Coming Up in Step 6: - Add custom domain configuration - Implement SSL/TLS certificates - Configure HTTPS redirects - Final security hardening

Your application is now: - Accessible through Nginx on port 80 - Protected with security headers - Optimized for performance - Ready for domain and SSL setup

With Nginx configured as our reverse proxy, we're ready to move on to the final step of adding a custom domain and SSL certificate to complete our professional deployment setup.