How to Deploy Spring Boot Application on AWS EC2 (Complete Guide)

Written By: *Saikat Goswami*

Deploying a Spring Boot application to the cloud is a milestone for any backend developer. While containers and serverless architectures are gaining traction, the classic **AWS EC2 (Elastic Compute Cloud)** remains a powerful, flexible, and cost-effective way to host your Java applications. It gives you full control over the operating system, the Java version, and the deployment process.

This guide will walk you through every step—from packaging your Spring Boot JAR to running it as a service on a production-ready EC2 instance.

Prerequisites

Before we begin, ensure you have the following:

- An **AWS Account** (with billing enabled).
- **AWS CLI** installed and configured on your local machine (optional but helpful).
- **SSH Client** (Terminal on Mac/Linux, Git Bash or PuTTY on Windows).
- **Java 11 or 17** installed locally.
- **Maven** or **Gradle** installed locally.
- A basic **Spring Boot** application ready to deploy.

Step 1: Package Your Spring Boot Application

First, you need to create a runnable JAR file. Navigate to your project's root directory and run:

Using Maven:

```
mvn clean package
```

Using Gradle:

```
gradle build
```

After the build succeeds, you will find a JAR file in the `target/` (Maven) or `build/libs/` (Gradle) directory. The file will be named something like `myapp-0.0.1-SNAPSHOT.jar`.
Test that the JAR works locally:

```
java -jar target/myapp-0.0.1-SNAPSHOT.jar
```

Your application should start on `http://localhost:8080`.

Step 2: Launch an EC2 Instance

Now, let's create the server that will host your application.

1. Log in to the **AWS Management Console** and search for **EC2**.
2. Click **Launch Instance**.
3. **Name your instance** – e.g., `spring-boot-server`.
4. **Choose an Amazon Machine Image (AMI)** – Select **Amazon Linux 2023 AMI** (Free tier eligible). It comes with a good package manager (`dnf`) and is lightweight.
5. **Choose an Instance Type** – Select `t2.micro` (Free tier eligible).
6. **Key pair (login)** – Create a new key pair or use an existing one. Download the `.pem` file and store it securely. You will need this to SSH into your server.
7. **Network Settings** – Click "Edit". Ensure **Allow SSH traffic from** is set to "My IP". Then click **Add security group rule** to allow HTTP traffic from anywhere (0.0.0.0/0) on port 8080 (Spring Boot's default port).
8. **Configure storage** – Keep default (8 GB gp2 is fine).
9. Click **Launch instance**.

Your EC2 instance will be ready in a few minutes.

Step 3: Connect to Your EC2 Instance

Once the instance is running, connect via SSH. Open your terminal and set the correct permissions for your key file:

```
chmod 400 your-key.pem
```

Then connect using the public IP of your instance:

```
ssh -i your-key.pem ec2-user@<YOUR-EC2-PUBLIC-IP>
```

For Amazon Linux, the default username is `ec2-user`. For Ubuntu AMIs, it's `ubuntu`.

Step 4: Install Java on EC2

Your Spring Boot JAR needs Java to run. Install Amazon's Corretto (OpenJDK distribution):

```
sudo dnf update -y
sudo dnf install java-17-amazon-corretto -y
```

Verify the installation:

```
java -version
```

You should see output showing `openjdk version 17`.

Step 5: Upload Your JAR to EC2

You need to get your JAR file from your local machine to the EC2 instance. Open a **new terminal window** (don't close the SSH session) and use `scp` (secure copy):

```
scp -i your-key.pem target/myapp-0.0.1-SNAPSHOT.jar ec2-user@<EC2-PUBLIC-IP>:/home/ec2-user/
```

Alternatively, you can use tools like `rsync` or even set up a CI/CD pipeline, but `scp` is the simplest for a one-off deployment.

Step 6: Run the Spring Boot Application

Back in your SSH session, run the JAR manually to test:

```
java -jar myapp-0.0.1-SNAPSHOT.jar
```

If everything works, you should see Spring Boot startup logs. Now, open your web browser and visit:

```
http://<YOUR-EC2-PUBLIC-IP>:8080
```

You should see your application's response. However, as soon as you close the terminal, the application stops. This is not acceptable for production.

Step 7: Run as a Background Service

To keep the app running after you log out, you have several options. The simplest is using `nohup`:

```
nohup java -jar myapp-0.0.1-SNAPSHOT.jar > app.log 2> &1 &
```

But for a real production setup, we'll use **systemd**—the Linux init system.

Create a Systemd Service

1. Create a new service file:

```
sudo nano /etc/systemd/system/springboot-app.service
```

2. Add the following content (adjust paths and usernames as needed):

```
[Unit]
Description=Spring Boot Application
After=network.target

[Service]
User=ec2-user
WorkingDirectory=/home/ec2-user
ExecStart=/usr/bin/java -jar /home/ec2-user/myapp-0.0.1-SNAPSHOT.jar
SuccessExitStatus=143
Restart=always
RestartSec=10

[Install]
WantedBy=multi-user.target
```

3. Save and exit (`Ctrl+O`, `Enter`, `Ctrl+X` in nano).

4. Reload systemd, enable the service to start on boot, and start it:

```
sudo systemctl daemon-reload
sudo systemctl enable springboot-app.service
sudo systemctl start springboot-app.service
```

5. Check the status:

```
sudo systemctl status springboot-app.service
```

Now your Spring Boot app will start automatically when the EC2 instance reboots and restart if it crashes.

Step 8: Configure AWS Security Group (Firewall)

If you cannot access your app on port 8080, it's likely a security group issue.

1. Go to **EC2 > Security Groups**.
2. Select the security group attached to your instance.
3. Click **Edit inbound rules**.
4. Add a rule:
    - **Type:** Custom TCP
    - **Port range:** 8080
    - **Source:** 0.0.0.0/0 (or your specific IP for better security)
5. Save rules.

No need to restart the instance—changes apply immediately.

Step 9: Optional – Map Port 80 to 8080

Users don't want to type `:8080`. To make your app accessible on the default HTTP port 80, you can use `iptables` to forward traffic:

```
sudo iptables -t nat -A PREROUTING -p tcp --dport 80 -j REDIRECT --
to-port 8080
```

To make this persistent across reboots:

```
sudo dnf install iptables-services -y
sudo service iptables save
```

Now your app is available at `http://<YOUR-EC2-PUBLIC-IP>` (no port needed).

Step 10: Monitoring Logs and Updates

View logs in real-time:

```
sudo journalctl -u springboot-app.service -f
```

Deploy a new version:

1. Upload the new JAR via `scp`.
2. Restart the service: `sudo systemctl restart springboot-app.service`

Troubleshooting Common Issues

Problem	Likely Fix
Connection refused on port 8080	Check security group inbound rules
`java: command not found`	Java not installed correctly
Permission denied (SSH)	Run `chmod 400` on your .pem file
Application crashes on startup	Check logs with `journalctl -u springboot-app`
Port 80 not working after iptables	Run `sudo iptables -L -t nat` to verify rules

Next Steps for Production

While running a Spring Boot app on a single EC2 instance is great for learning and small projects, a production environment would require:

- **Elastic IP** – To keep the same IP address after instance restarts.
- **Custom domain & HTTPS** – Using Route 53 and AWS Certificate Manager (via a load balancer or NGINX).
- **Database** – Use AWS RDS instead of embedded H2 or local MySQL.
- **Environment variables** – Store sensitive config (DB passwords, API keys) in AWS Systems Manager Parameter Store or use `.env` files.
- **CI/CD** – Automate deployment with GitHub Actions or AWS CodeDeploy.

Conclusion

You have just deployed a Spring Boot application on AWS EC2 from scratch. You learned how to:

- Package your Spring Boot app into a JAR.
- Launch and connect to an EC2 instance.
- Install Java and upload your application.
- Run it as a resilient systemd service.
- Open network ports and even map port 80 to 8080.

This setup forms the foundation of cloud deployment. Master it, and you can deploy anything from REST APIs to full-stack web applications on AWS infrastructure.

Now go ahead—deploy your next Spring Boot project to the cloud with confidence.