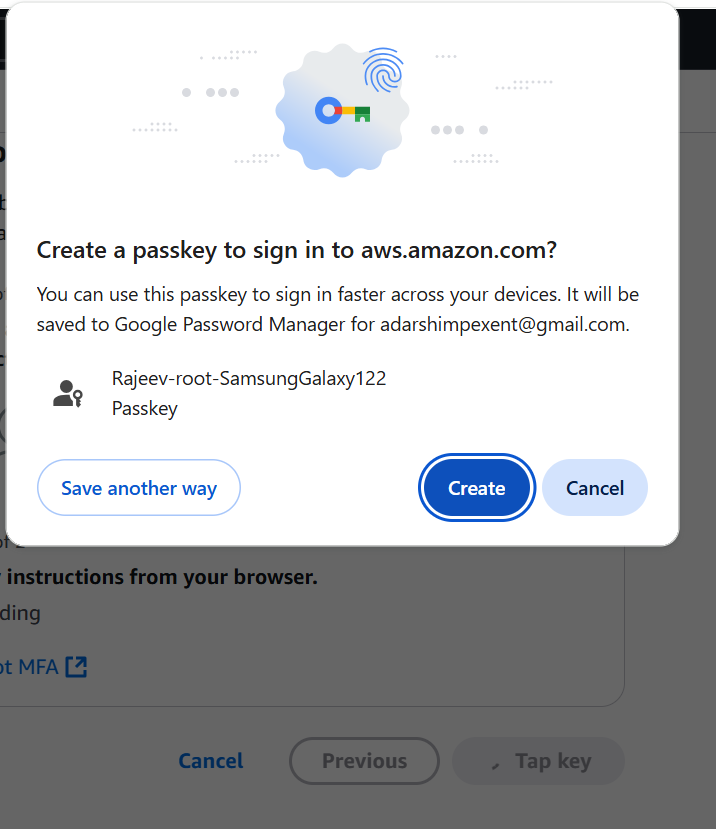
**Passkey** Rajeev-root-SamsungGalaxy122



A screenshot of a password manager

AI-generated content may be incorrect.

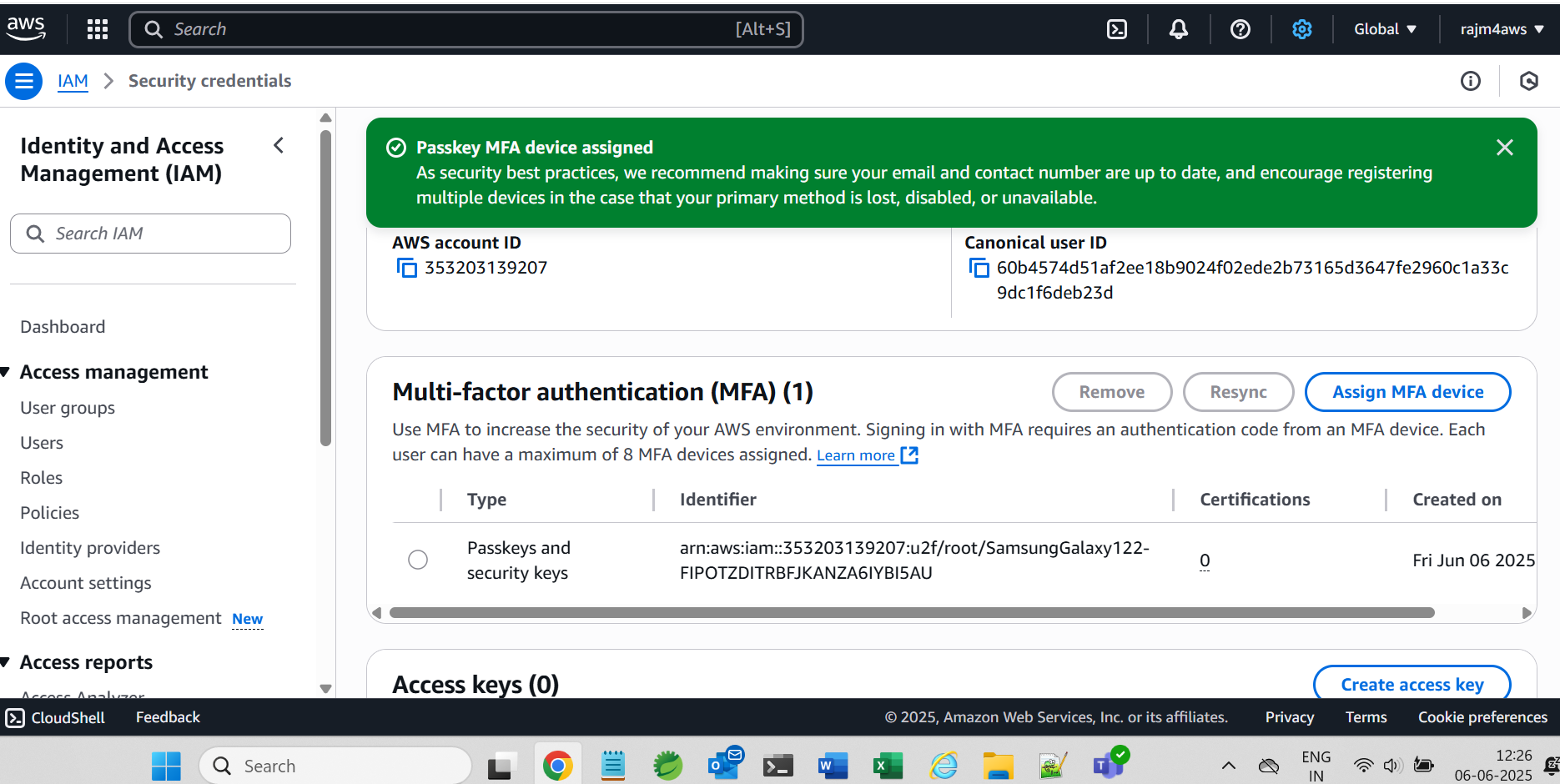
A screenshot of a computer

AI-generated content may be incorrect.

**Recovery** **Pin** 121218

**Canonical ID** 60b4574d51af2ee18b9024f02ede2b73165d3647fe2960c1a33c9dc1f6deb23

**AWS account ID** 353203139207



*In AWS how to Manually set up and configure the packages, for deploying my local Java Spring Boot web app using Tomcat ? Please elaborately enlist the steps. My environment includes Spring Boot 3.4.x, JDK 21, Spring Security 6.x. I want now to deploy this to my free tier account in AWS.*

To manually set up and configure the necessary packages for deploying your local Java Spring Boot web app using Tomcat on an AWS Free Tier EC2 instance, you'll generally follow these steps. This process involves:

**setting** up a **Linux** server,

**installing Java, Tomcat**, and then,

**deploying** your application.

Here's an elaborative enlistment of the steps: Deploying a Java Spring Boot web application (packaged as a WAR file) manually on an AWS EC2 Free Tier instance involves several steps, from setting up the instance to configuring the environment and deploying your application.

**Prerequisites:**

**AWS Account:** A free tier eligible AWS account.

**Spring Boot Application (WAR):** Your Spring Boot application should be built as a .war file.

**Crucial for WAR Deployment:** Ensure your pom.xml has <packaging>war</packaging>.

Also, your main Spring Boot application class should extend SpringBootServletInitializer and override the configure method.

Java

// Example:

@SpringBootApplication

public class MySpringApplication **extends** SpringBootServletInitializer {

public static void **main**(String[] args) {

SpringApplication.run(MySpringApplication.class, args);

}

@Override

protected SpringApplicationBuilder **configure**(SpringApplicationBuilder application) {

return application.sources(MySpringApplication.class);

}

}

Make sure spring-boot-starter-tomcat has a **<scope>provided</scope>** dependency in your pom.xml **if you're using an external Tomcat**.

XML

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-tomcat</artifactId>

<scope>provided</scope>

</dependency>

Build your WAR file: mvn clean package (or gradle clean build). The .war file will be in your target/ (Maven) or build/libs/ (Gradle) directory.

**Step-by-Step Deployment Guide**

**1. Launch an EC2 Instance (Free Tier Eligible)**

**Log in to AWS Management Console:** Go to the EC2 Dashboard.

**Launch Instance:** Click "Launch instances".

**Name and Tags:** Give your instance a descriptive name (e.g., SpringBootTomcatServer).

**Application and OS Images (Amazon Machine Image - AMI):**

Choose an **Amazon Linux AMI** (e.g., Amazon Linux 2023 AMI). These are generally free tier eligible and optimized for AWS. Ubuntu or Red Hat also work, but Amazon Linux comes with some AWS integrations out-of-the-box.

**Instance Type:** Select a **t2.micro** or t3.micro (if t2.micro is not available in your region) – these are Free Tier eligible.

**Key Pair (Login):**

Choose an existing key pair or "Create a new key pair".

**Download the .pem file** immediately if you create a new one. This is crucial for connecting to your instance via SSH. Keep it secure.

**Network Settings:**

**Firewall (Security Groups):** This is vital. Click "Edit" next to Security Group.

**SSH (Port 22):** Already configured, keep it to "My IP" or "Anywhere" (less secure but easier for testing).

**HTTP (Port 80):** Add a new rule.

Type: HTTP

Source: Anywhere (0.0.0.0/0) or "My IP" for development.

**Custom TCP (Port 8080):** Add another new rule (Tomcat's default port).

Type: Custom TCP

Port Range: 8080

Source: Anywhere (0.0.0.0/0) or "My IP".

*(Optional: If your Spring Boot app uses a different port, open that port instead.)*

Leave other settings as default for Free Tier eligibility unless you know what you're doing.

**Configure Storage:**

Default 8 GiB gp2/gp3 is usually sufficient and Free Tier eligible. You can go up to 30 GiB for the Free Tier.

**Launch Instance:** Review your settings and click "Launch instance".

**View Instances:** Click "View all instances" to see your new instance. Wait for its "Instance State" to become Running and "Status Checks" to pass (2/2 checks passed).

**2. Connect to Your EC2 Instance**

**Get Public IP/DNS:** Select your running instance in the EC2 Dashboard. Note down its "Public IPv4 address" or "Public IPv4 DNS".

**Connect via SSH:**

**Linux/macOS:**

Bash

**chmod** 400 /path/to/your-key-pair.pem

ssh -i /path/to/your-key-pair.pem ec2-user@**<**Your-Instance-Public-IP-or-DNS**>**

(**Replace** /path/to/your-key-pair.pem **with your key file** path and ec2-user **with the default user** for your AMI, e.g., ec2-user for Amazon Linux, ubuntu for Ubuntu, admin for RHEL).

**Windows (using PuTTY/Git Bash/WSL):**

**PuTTY:** Use PuTTYgen to convert your .pem file to .ppk format. Then, configure PuTTY with the .ppk key under Connection -> SSH -> Auth and use ec2-user@<Your-Instance-Public-IP-or-DNS> as the hostname.

**Git Bash/WSL:** Follow the Linux/macOS instructions.

**AWS EC2 Instance Connect (Web-based):** Select your instance, click "Connect", choose "EC2 Instance Connect", and click "Connect". This is the easiest for quick access.

**3. Install JDK 21**

Once connected to your EC2 instance via SSH:

**Update System Packages:**

Bash

sudo yum update -y

(For Amazon Linux 2023, yum is still used, but it's dnf-based under the hood.)

**Install Amazon Corretto 21 (Amazon's OpenJDK distribution):** Amazon Linux 2023 (AL2023) comes with Corretto 17 as default. You can install Corretto 21 using yum. For server workloads, the headless version is usually sufficient.

Bash

sudo yum install java-21-amazon-corretto-devel -y

This installs the JDK development tools for Corretto 21.

**Verify Java Installation:**

Bash

java -version

You should see output similar to: openjdk version "21.0.x" ... Corretto-21.0.x...

**4. Install Apache Tomcat**

**Download Tomcat:** Go to the official Apache Tomcat website (e.g., tomcat.apache.org) and find the latest stable version of Tomcat 10 (or 9 if preferred for broader compatibility). Copy the direct download link for the .tar.gz file (e.g., apache-tomcat-10.x.x.tar.gz).

Bash

cd /opt # Recommended directory for optional software

sudo wget https://dlcdn.apache.org/tomcat/tomcat-10/v10.1.20/bin/apache-tomcat-10.1.20.tar.gz # Replace with latest version

**Extract Tomcat:**

Bash

sudo tar -xvzf apache-tomcat-10.1.20.tar.gz

**Rename for Convenience (Optional):**

Bash

sudo mv apache-tomcat-10.1.20 tomcat

Now Tomcat is in /opt/tomcat.

**Set Permissions:**

Bash

sudo chown -R ec2-user:ec2-user /opt/tomcat # Change ownership to your EC2 user

sudo chmod -R u+x /opt/tomcat/bin

**Configure Tomcat Users (Optional but Recommended for Manager UI):** To access the Tomcat Manager web interface (useful for deploying/undeploying WARs via UI), you need to configure users.

Bash

nano /opt/tomcat/conf/tomcat-users.xml

Add roles and users within the <tomcat-users> tags, *before* the closing </tomcat-users> tag.

XML

<role rolename="manager-gui"/>

<role rolename="manager-script"/>

<role rolename="admin-gui"/>

<user username="admin" password="your\_strong\_password" roles="manager-gui,manager-script,admin-gui"/>

**Important:** Change your\_strong\_password to a secure password. Save and exit (Ctrl+O, Enter, Ctrl+X for nano).

**Adjust context.xml for Manager (If Accessing from Anywhere):** By default, the Tomcat Manager (/manager/html) is restricted to localhost. To access it from your own IP, you might need to comment out or modify the RemoteAddrValve.

Bash

nano /opt/tomcat/webapps/manager/META-INF/context.xml

Find the <Valve ... RemoteAddrValve ... /> line and comment it out:

Save and exit. Do the same for webapps/host-manager/META-INF/context.xml if you want to access Host Manager.

**Start Tomcat:**

Bash

/opt/tomcat/bin/startup.sh

You should see output like Tomcat started.

**Verify Tomcat:** Open your web browser and navigate to: http://<Your-Instance-Public-IP-or-DNS>:8080/ You should see the Apache Tomcat welcome page. If not, check your security group rules and ensure Tomcat is running.

**5. Transfer Your Spring Boot WAR File**

**Build your WAR:** On your local machine, run mvn clean package (or gradle clean build) to generate the .war file. It will be in target/your-app-name.war or build/libs/your-app-name.war.

**Transfer using SCP (Secure Copy):** From your local machine's terminal (where your .pem key is):

Bash

scp -i /path/to/your-key-pair.pem /path/to/your-app-name.war ec2-user@<Your-Instance-Public-IP-or-DNS>:/tmp/

(This copies the WAR file to the /tmp/ directory on your EC2 instance.)

**Windows (WinSCP):** Use WinSCP (a free SFTP/FTP client) to connect to your EC2 instance using your .ppk key. Drag and drop the WAR file to a directory like /tmp/.

**6. Deploy Your Spring Boot WAR Application**

**Stop Tomcat:**

Bash

/opt/tomcat/bin/shutdown.sh

**Clear existing applications (Optional but Recommended):** If this is a fresh deploy or you want to remove previous apps:

Bash

sudo rm -rf /opt/tomcat/webapps/\* # WARNING: This deletes everything in webapps

**Move the WAR file to Tomcat's webapps directory:**

Bash

sudo mv /tmp/your-app-name.war /opt/tomcat/webapps/

**Start Tomcat:**

Bash

/opt/tomcat/bin/startup.sh

Tomcat will automatically detect the .war file, unpack it, and deploy it. This might take a few seconds depending on your application size.

**7. Access Your Deployed Application**

**Determine Context Path:**

If your WAR file is named ROOT.war, your application will be accessible directly at http://<Your-Instance-Public-IP-or-DNS>:8080/.

If your WAR file is named your-app-name.war, your application's context path will be your-app-name. You would access it at: http://<Your-Instance-Public-IP-or-DNS>:8080/your-app-name/

Replace /your-app-name/ with any specific endpoint of your Spring Boot application (e.g., /your-app-name/api/hello).

**8. (Optional) Configure Tomcat as a Systemd Service**

For better management and automatic startup on reboot, it's recommended to configure Tomcat as a systemd service.

**Create a systemd service file:**

Bash

sudo nano /etc/systemd/system/tomcat.service

**Paste the following content:**

Ini, TOML

[Unit]

Description=Apache Tomcat Web Application Container

After=network.target

[Service]

Type=forking

User=ec2-user

Group=ec2-user

Environment="JAVA\_HOME=/usr/lib/jvm/java-21-amazon-corretto" # Verify this path with `readlink -f $(which java)`

Environment="CATALINA\_HOME=/opt/tomcat"

Environment="CATALINA\_BASE=/opt/tomcat"

ExecStart=/opt/tomcat/bin/startup.sh

ExecStop=/opt/tomcat/bin/shutdown.sh

ExecReload=/bin/kill -HUP $MAINPID

SuccessExitStatus=143

[Install]

WantedBy=multi-user.target

**Verify JAVA\_HOME:** Run readlink -f $(which java) on your EC2 instance to get the exact path to your Java installation. Adjust Environment="JAVA\_HOME=..." accordingly.

User=ec2-user and Group=ec2-user assume you've changed ownership of /opt/tomcat as advised earlier. If you use a different user/group, modify these. Save and exit.

**Reload systemd, enable and start the service:**

Bash

sudo systemctl daemon-reload

sudo systemctl enable tomcat

sudo systemctl start tomcat

**Check service status:**

Bash

sudo systemctl status tomcat

It should show "active (running)".

Now, **Tomcat will start automatically on instance reboot**, and you can manage it using **sudo systemctl start|stop|restart|status tomcat**.

This comprehensive guide should help you get your Spring Boot application running on AWS EC2 using Tomcat! **Remember to monitor your Free** Tier usage to avoid unexpected charges.

Sources

I **don't** have a **separate tomcat** configured so, can you enlist the application deployment and, finally, execution steps?

Okay, if your Spring Boot application does not have a separate Tomcat configured, it means you're building a standard Spring Boot executable JAR file, which includes an embedded Tomcat (or other web server like Jetty/Undertow). This is the default and usually simpler way to deploy Spring Boot applications.

Here are the step-by-step instructions for deploying and executing your Spring Boot JAR application on an AWS Free Tier EC2 instance, assuming you've built it with **JDK 21** and are using **Spring Boot 3.x**.

**Prerequisites:**

1. **AWS Account:** A free tier eligible AWS account.
2. **Spring Boot Application (JAR):**
   * Ensure your pom.xml has <packaging>jar</packaging> (or no <packaging> tag at all, as jar is the default).
   * Build your JAR file: mvn clean package (or gradle clean build). The executable JAR file will be in your target/ (Maven) or build/libs/ (Gradle) directory (e.g., your-app-name-0.0.1-SNAPSHOT.jar).

**Step-by-Step Deployment Guide for Spring Boot JAR**

**1. Launch an EC2 Instance (Free Tier Eligible)**

1. **Log in to AWS Management Console:** Go to the EC2 Dashboard.
2. **Launch Instance:** Click "Launch instances".
3. **Name and Tags:** Give your instance a descriptive name (e.g., SpringBootAppServer).
4. **Application and OS Images (Amazon Machine Image - AMI):**
   * Choose an **Amazon Linux AMI** (e.g., Amazon Linux 2023 AMI). These are generally free tier eligible and optimized for AWS.
5. **Instance Type:** Select a **t2.micro** or t3.micro (if t2.micro is not available in your region) – these are Free Tier eligible.
6. **Key Pair (Login):**
   * Choose an existing key pair or "Create a new key pair".
   * **Download the .pem file** immediately if you create a new one. This is crucial for connecting to your instance via SSH. Keep it secure.
7. **Network Settings:**
   * **Firewall (Security Groups):** This is vital. Click "Edit" next to Security Group.
     + **SSH (Port 22):** Already configured, keep it to "My IP" or "Anywhere" (less secure but easier for testing).
     + **Custom TCP (Port 9192):** Add a new rule. This is the port your Spring Boot app is configured to use (server.port=9192).
       - Type: Custom TCP
       - Port Range: 9192
       - Source: Anywhere (0.0.0.0/0) or "My IP" (for testing/limited access).
     + *(Optional: If your app also exposes HTTP on default port 80, add a rule for HTTP as well.)*
   * Leave other settings as default for Free Tier eligibility unless you know what you're doing.
8. **Configure Storage:**
   * Default 8 GiB gp2/gp3 is usually sufficient and Free Tier eligible. You can go up to 30 GiB for the Free Tier.
9. **Launch Instance:** Review your settings and click "Launch instance".
10. **View Instances:** Click "View all instances" to see your new instance. Wait for its "Instance State" to become Running and "Status Checks" to pass (2/2 checks passed).

**2. Connect to Your EC2 Instance**

1. **Get Public IP/DNS:** Select your running instance in the EC2 Dashboard. Note down its "Public IPv4 address" or "Public IPv4 DNS".
2. **Connect via SSH:**
   * **Linux/macOS:**

Bash

chmod 400 /path/to/your-key-pair.pem

ssh -i /path/to/your-key-pair.pem ec2-user@<Your-Instance-Public-IP-or-DNS>

* + **Windows (using Git Bash/WSL):** Follow the Linux/macOS instructions.
  + **AWS EC2 Instance Connect (Web-based):** Select your instance, click "Connect", choose "EC2 Instance Connect", and click "Connect". This is the easiest for quick access.

**3. Install JDK 21**

Once connected to your EC2 instance via SSH:

1. **Update System Packages:**

Bash

sudo yum update -y

1. **Install Amazon Corretto 21 (Amazon's OpenJDK distribution):**

Bash

sudo yum install java-21-amazon-corretto-devel -y

This installs the JDK development tools for Corretto 21.

1. **Verify Java Installation:**

Bash

java -version

You should see output similar to: openjdk version "21.0.x" ... Corretto-21.0.x...

**4. Build & Transfer Your Spring Boot JAR File**

1. **Build your JAR:** On your local machine, run mvn clean package (or gradle clean build). This will create the executable JAR (e.g., your-app-name-0.0.1-SNAPSHOT.jar) in your target/ or build/libs/ directory.
2. **Transfer using SCP (Secure Copy):** From your local machine's terminal (where your .pem key is, and you are in the directory containing your JAR):

Bash

scp -i /path/to/your-key-pair.pem target/your-app-name-0.0.1-SNAPSHOT.jar ec2-user@<Your-Instance-Public-IP-or-DNS>:/home/ec2-user/

(This copies the JAR file to the ec2-user's home directory on your EC2 instance. Adjust your-app-name-0.0.1-SNAPSHOT.jar to your actual JAR file name.)

* + **Windows (WinSCP):** Use WinSCP to connect to your EC2 instance. Drag and drop the JAR file to /home/ec2-user/.

**5. Deploy & Execute Your Spring Boot Application**

1. **Move the JAR (Optional but recommended):** You can create a dedicated directory for your application.

Bash

cd /home/ec2-user/

mkdir my-spring-app

mv your-app-name-0.0.1-SNAPSHOT.jar my-spring-app/

cd my-spring-app/

*(Rename your-app-name-0.0.1-SNAPSHOT.jar to something simpler like app.jar if you prefer: mv your-app-name-0.0.1-SNAPSHOT.jar app.jar)*

1. **Basic Execution (For Testing - runs in foreground):**

Bash

java -jar app.jar

Your application will start, and you'll see logs directly in your terminal. This is good for initial testing but **will block your SSH session**. If you close SSH, the app stops.

1. **Verify Application:** Open your web browser and navigate to: http://<Your-Instance-Public-IP-or-DNS>:9192/ (Or http://<Your-Instance-Public-IP-or-DNS>:9192/<your-context-path> if you have a context path configured). You should see your Spring Boot application responding.

**6. Configure as a Systemd Service (RECOMMENDED for Production/Reliability)**

For a robust deployment, you want your application to run in the background, start automatically on reboot, and be easily manageable. Systemd is the standard way to do this on Linux.

1. **Stop the foreground application:** If you ran java -jar in the previous step, press Ctrl+C in your SSH terminal to stop it.
2. **Create a systemd service file:**

Bash

sudo nano /etc/systemd/system/spring-boot-app.service

1. **Paste the following content:**

Ini, TOML

[Unit]

Description=My Spring Boot Application

After=network.target

[Service]

User=ec2-user

# Group=ec2-user # Uncomment if you want to assign a group, otherwise default is fine

WorkingDirectory=/home/ec2-user/my-spring-app # Directory where your JAR is located

ExecStart=/usr/bin/java -jar app.jar # Path to Java and your JAR file name

StandardOutput=journal

StandardError=journal

Restart=always

RestartSec=10

SuccessExitStatus=143 # Signifies graceful shutdown (e.g. from SIGTERM)

[Install]

WantedBy=multi-user.target

* + **User=ec2-user:** Runs the app as the ec2-user. Good for free tier.
  + **WorkingDirectory:** Make sure this points to the directory where you placed app.jar.
  + **ExecStart:** Make sure /usr/bin/java is the correct path to the java executable (you can verify with which java). Make sure app.jar matches the name of your JAR file.
  + **Restart=always:** Ensures your application restarts if it crashes or the server reboots.
  + Save and exit (Ctrl+O, Enter, Ctrl+X for nano).

1. **Reload systemd, enable and start the service:**

Bash

sudo systemctl daemon-reload

sudo systemctl enable spring-boot-app # Enables auto-start on boot

sudo systemctl start spring-boot-app

1. **Check service status:**

Bash

sudo systemctl status spring-boot-app

It should show active (running).

1. **View logs (Optional):**

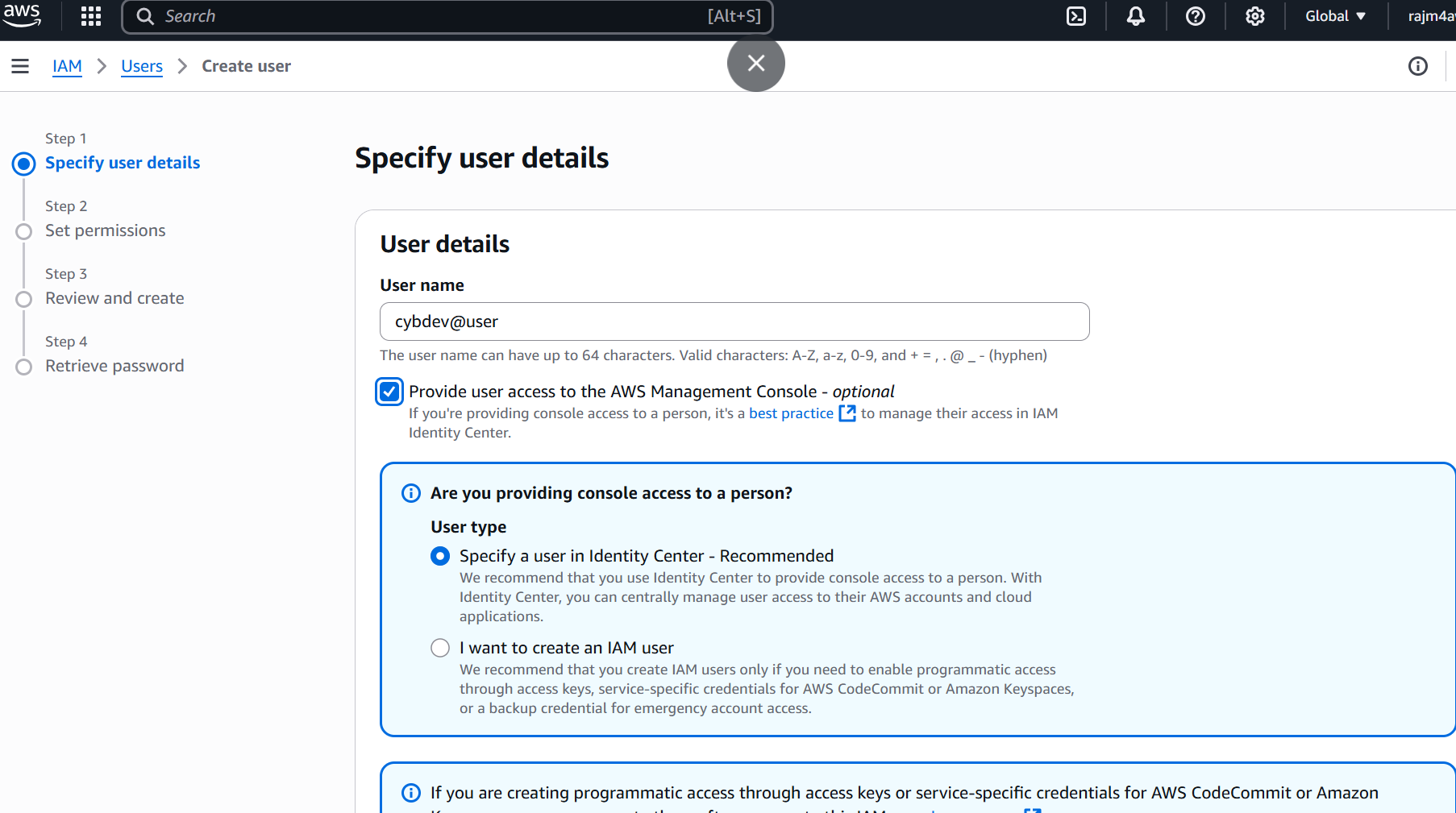
Bash

sudo journalctl -u spring-boot-app -f

This command will show you the real-time logs of your Spring Boot application.

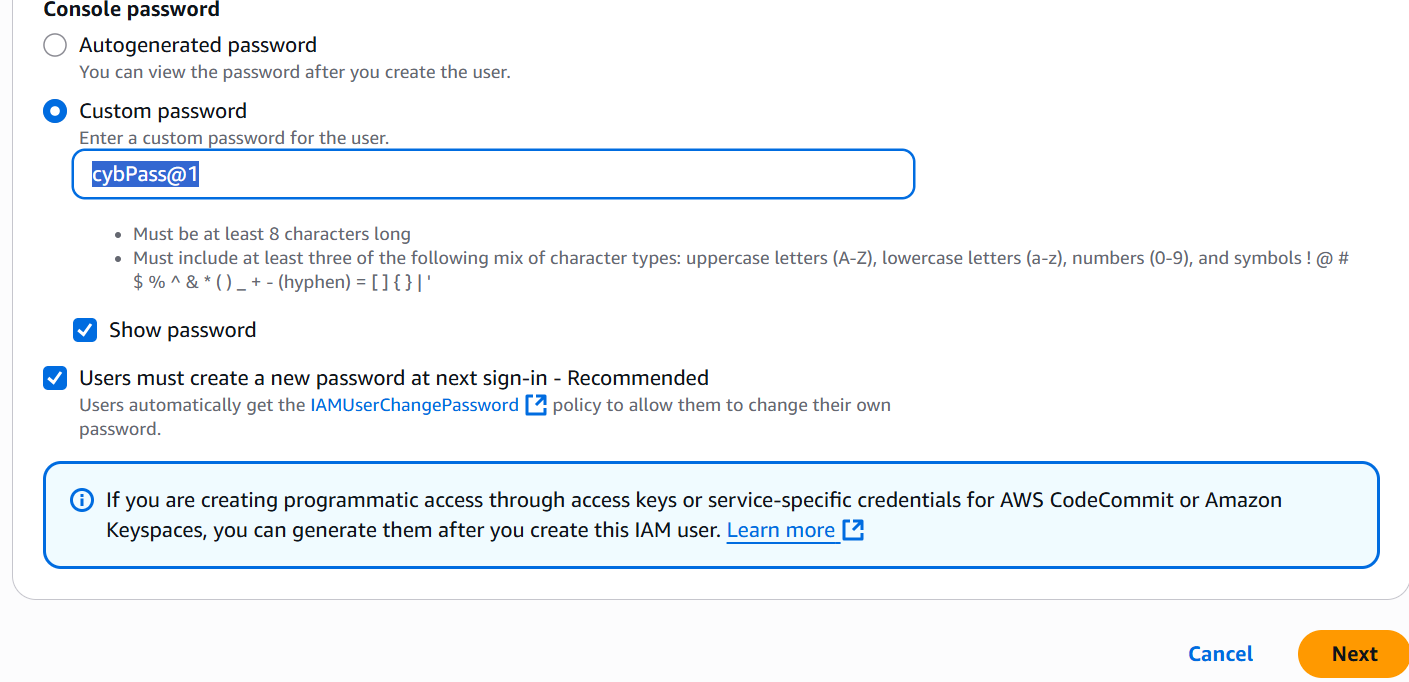
Now, your Spring Boot application is deployed, runs in the background, and will automatically restart if the instance reboots or the application crashes. Remember to monitor your Free Tier usage to avoid unexpected charges.

**AWS console Steps:**



A screenshot of a computer

AI-generated content may be incorrect.



A screenshot of a computer

AI-generated content may be incorrect.

context: AWS IAM user creation

With reference to the above context, I am creating a new IAM user, "cybdev@user" . on top of the root user I already have,

In this, I came across an optional step of creating a group. I feel I should create a developer group called "DevUserGroup", and make this IAM user, "cybdev@user", belong to this group.

The question is , I am not sure how many permissions should I assign to this group, so that mostly all developer related functions should be smoothly handled? Please mentions a few Permissions , I must give, in order to achieve this, and few nice to haves , kind of.

Creating an IAM group for your developers, like "DevUserGroup", is an excellent practice! It adheres to the principle of least privilege and simplifies user management, as you can assign permissions to the group once, and all users in that group inherit them.

The key is to give developers enough permissions to do their job efficiently in a development or test environment, without giving them excessive access that could pose a security risk.

Here's a breakdown of "must-have" and "nice-to-have" permissions, specifically tailored for a developer working on a Java Spring Boot web app, often interacting with EC2, S3, and CloudWatch.

**Core Principle: Least Privilege**

Always start with the minimum required permissions and add more as needed. For a "developer" role, the goal is often to enable them to:

* Launch, manage, and terminate their own development/test EC2 instances.
* View logs and metrics of their applications.
* Store and retrieve application artifacts (JARs, WARs) in S3.
* View the existing infrastructure to understand the environment.
* **Crucially: Prevent access to production environments or sensitive data unless absolutely necessary and audited.**

**How to Create/Attach Permissions**

You'll create a **custom managed policy** in AWS IAM and attach it to your "DevUserGroup". This allows you to tailor permissions very specifically.

1. **Navigate to IAM:** Go to the AWS Management Console, search for "IAM", and go to the dashboard.
2. **Create Group:** Go to "User groups" -> "Create group" -> Name it DevUserGroup. Add your cybdev@user to it.
3. **Create Policy:** Go to "Policies" -> "Create policy" -> Choose the "JSON" tab. Paste the JSON for the "Must-Have" policy below.
4. **Attach Policy:** After creating the policy, go back to your DevUserGroup and click "Add permissions" -> "Attach policies" -> Search for your newly created custom policy and attach it.

**Must-Have Permissions (Essential for Development Tasks)**

These permissions form the core set for a developer managing their own app on EC2. You'll likely combine these into a single custom managed policy named something like **DevUserGroupCorePolicy**.

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": [

// EC2 (Compute) - Basic Instance Management

"ec2:DescribeInstances", // View instances

"ec2:RunInstances", // Launch new instances

"ec2:StartInstances", // Start stopped instances

"ec2:StopInstances", // Stop running instances

"ec2:RebootInstances", // Reboot instances

"ec2:TerminateInstances", // Terminate instances (use with caution, maybe restrict by tags in production)

"ec2:CreateTags", // Tag resources (important for organization)

"ec2:DeleteTags", // Delete tags

"ec2:DescribeKeyPairs", // View SSH key pairs

"ec2:CreateKeyPair", // Create SSH key pairs

"ec2:DeleteKeyPair", // Delete SSH key pairs

"ec2:DescribeVolumes", // View EBS volumes

"ec2:DescribeImages", // View AMIs

"ec2:DescribeInstanceStatus", // View instance health

// EC2 (Networking - View Only for SG, Subnets, VPC)

"ec2:DescribeSecurityGroups",

"ec2:DescribeSubnets",

"ec2:DescribeVpcs",

"ec2:DescribeNetworkInterfaces",

// Security Groups (for creating/modifying access for their own dev/test instances)

"ec2:CreateSecurityGroup",

"ec2:DeleteSecurityGroup",

"ec2:AuthorizeSecurityGroupIngress", // Allow inbound rules

"ec2:RevokeSecurityGroupIngress", // Revoke inbound rules

// S3 (Storage for Application Artifacts, Logs)

"s3:ListAllMyBuckets", // List all buckets (general)

"s3:ListBucket", // List objects in a bucket (requires specific ARN)

"s3:GetObject", // Download objects

"s3:PutObject", // Upload objects (e.g., JARs/WARs)

"s3:DeleteObject", // Delete objects

"s3:PutBucketVersioning", // Enable versioning (good practice)

// IMPORTANT: Restrict S3 actions to specific development buckets for security

// "Resource": [

// "arn:aws:s3:::your-dev-bucket-name",

// "arn:aws:s3:::your-dev-bucket-name/\*"

// ]

// CloudWatch (Monitoring & Logging)

"cloudwatch:DescribeAlarms",

"cloudwatch:GetMetricData",

"cloudwatch:ListMetrics",

"logs:DescribeLogGroups", // View log groups

"logs:DescribeLogStreams", // View log streams

"logs:FilterLogEvents", // Search log events

"logs:GetLogEvents", // Retrieve log events

"logs:CreateLogGroup", // Create new log groups (if app creates)

"logs:CreateLogStream", // Create new log streams

"logs:PutLogEvents", // Put log events (if app logs directly to CloudWatch)

// IAM (Read-Only for context)

"iam:ListUsers",

"iam:ListRoles",

"iam:ListPolicies",

"iam:GetUser",

"iam:GetRole",

"iam:GetPolicy",

// AWS Console Basic Access

"aws-portal:ViewBilling", // Optional, but useful for developers to see service usage

"aws-portal:ViewUsage" // Optional, but useful for developers to see service usage

],

"Resource": "\*" // Apply broadly, then restrict where possible (e.g., S3 buckets)

}

]

}

Creating an **IAM group** for your developers, like "**DevUserGroup**", is an excellent practice! It adheres to the principle of least privilege and simplifies user management, as you can assign permissions to the group once, and all users in that group inherit them.

The key is to give developers enough permissions to do their job efficiently in a development or test environment, without giving them excessive access that could pose a security risk.

Here's a breakdown of "must-have" and "nice-to-have" permissions, specifically tailored for a developer working on a **Java Spring Boot** web app,often interactingwith **EC2, S3,** and **CloudWatch.**

**Core Principle: Least Privilege**

Always start with the minimum required permissions and add more as needed. For a "developer" role, the goal is often to enable them to:

* Launch, manage, and terminate their own development/test EC2 instances.
* View logs and metrics of their applications.
* Store and retrieve application artifacts (JARs, WARs) in S3.
* View the existing infrastructure to understand the environment.
* **Crucially: Prevent access to production environments or sensitive data unless absolutely necessary and audited.**

**How to Create/Attach Permissions**

You'll create a **custom managed policy** in AWS IAM and attach it to your "DevUserGroup". This allows you to tailor permissions very specifically.

1. **Navigate to IAM:** Go to the AWS Management Console, search for "IAM", and go to the dashboard.
2. **Create Group:** Go to "User groups" -> "Create group" -> Name it DevUserGroup. Add your cybdev@user to it.
3. **Create Policy:** Go to "Policies" -> "Create policy" -> Choose the "JSON" tab. Paste the JSON for the "Must-Have" policy below.
4. **Attach Policy:** After creating the policy, go back to your DevUserGroup and click "Add permissions" -> "Attach policies" -> Search for your newly created custom policy and attach it.

**Must-Have Permissions (Essential for Development Tasks)**

These permissions form the core set for a developer managing their own app on EC2. You'll likely combine these into a single custom managed policy named something like DevUserGroupCorePolicy.

JSON

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": [

// EC2 (Compute) - Basic Instance Management

"ec2:DescribeInstances", // View instances

"ec2:RunInstances", // Launch new instances

"ec2:StartInstances", // Start stopped instances

"ec2:StopInstances", // Stop running instances

"ec2:RebootInstances", // Reboot instances

"ec2:TerminateInstances", // Terminate instances (use with caution, maybe restrict by tags in production)

"ec2:CreateTags", // Tag resources (important for organization)

"ec2:DeleteTags", // Delete tags

"ec2:DescribeKeyPairs", // View SSH key pairs

"ec2:CreateKeyPair", // Create SSH key pairs

"ec2:DeleteKeyPair", // Delete SSH key pairs

"ec2:DescribeVolumes", // View EBS volumes

"ec2:DescribeImages", // View AMIs

"ec2:DescribeInstanceStatus", // View instance health

// EC2 (Networking - View Only for SG, Subnets, VPC)

"ec2:DescribeSecurityGroups",

"ec2:DescribeSubnets",

"ec2:DescribeVpcs",

"ec2:DescribeNetworkInterfaces",

// Security Groups (for creating/modifying access for their own dev/test instances)

"ec2:CreateSecurityGroup",

"ec2:DeleteSecurityGroup",

"ec2:AuthorizeSecurityGroupIngress", // Allow inbound rules

"ec2:RevokeSecurityGroupIngress", // Revoke inbound rules

// S3 (Storage for Application Artifacts, Logs)

"s3:ListAllMyBuckets", // List all buckets (general)

"s3:ListBucket", // List objects in a bucket (requires specific ARN)

"s3:GetObject", // Download objects

"s3:PutObject", // Upload objects (e.g., JARs/WARs)

"s3:DeleteObject", // Delete objects

"s3:PutBucketVersioning", // Enable versioning (good practice)

// IMPORTANT: Restrict S3 actions to specific development buckets for security

// "Resource": [

// "arn:aws:s3:::your-dev-bucket-name",

// "arn:aws:s3:::your-dev-bucket-name/\*"

// ]

// CloudWatch (Monitoring & Logging)

"cloudwatch:DescribeAlarms",

"cloudwatch:GetMetricData",

"cloudwatch:ListMetrics",

"logs:DescribeLogGroups", // View log groups

"logs:DescribeLogStreams", // View log streams

"logs:FilterLogEvents", // Search log events

"logs:GetLogEvents", // Retrieve log events

"logs:CreateLogGroup", // Create new log groups (if app creates)

"logs:CreateLogStream", // Create new log streams

"logs:PutLogEvents", // Put log events (if app logs directly to CloudWatch)

// IAM (Read-Only for context)

"iam:ListUsers",

"iam:ListRoles",

"iam:ListPolicies",

"iam:GetUser",

"iam:GetRole",

"iam:GetPolicy",

// AWS Console Basic Access

"aws-portal:ViewBilling", // Optional, but useful for developers to see service usage

"aws-portal:ViewUsage" // Optional, but useful for developers to see service usage

],

"Resource": "\*" // Apply broadly, then restrict where possible (e.g., S3 buckets)

}

]

}

**Important Considerations for the "Must-Have" Policy:**

* **Resource Level Permissions:** For s3:\* actions, it is **highly recommended** to replace "Resource": "\*" with specific ARNs of the development S3 buckets (e.g., arn:aws:s3:::your-dev-bucket-name and arn:aws:s3:::your-dev-bucket-name/\*). This is a crucial security hardening step.
* **ec2:TerminateInstances:** While included for developer convenience, in some organizations, even development instance termination is restricted or requires an approval process. Assess your team's needs.
* **ec2:CreateSecurityGroup, ec2:AuthorizeSecurityGroupIngress:** These are powerful. Developers can inadvertently open ports to the internet. Consider restricting Source to My IP in their security groups or having an operations team manage security groups.

**Nice-to-Have Permissions (Depending on Specific Developer Roles)**

These are additional permissions that you might attach to the DevUserGroup or even to individual users if their role requires it. You'd typically use AWS Managed Policies for these or create smaller custom policies.

1. **For Database Access (if they manage/access databases directly):**
   * **AmazonRDSReadOnlyAccess:** (AWS Managed Policy) Allows viewing RDS instances and databases.
   * **AmazonRDSFullAccess:** (AWS Managed Policy) Allows creating, modifying, and deleting RDS instances. **Use with extreme caution and only in dedicated development accounts.**
   * **AmazonDynamoDBReadOnlyAccess:** (AWS Managed Policy) For DynamoDB.
   * **AmazonDynamoDBFullAccess:** (AWS Managed Policy) For DynamoDB.
2. **For Serverless (AWS Lambda, API Gateway):**
   * **AWSLambda\_FullAccess:** (AWS Managed Policy) If they deploy/manage Lambda functions.
   * **AmazonAPIGatewayAdministrator:** (AWS Managed Policy) If they configure API Gateways.
3. **For Container Services (ECS, ECR):**
   * **AmazonEC2ContainerRegistryFullAccess:** (AWS Managed Policy) To push/pull Docker images from ECR.
   * **AmazonECS\_FullAccess:** (AWS Managed Policy) To manage ECS clusters, tasks, and services.
4. **For CI/CD (if they use AWS's Code services):**
   * **AWSCodeCommitPowerUser:** (AWS Managed Policy) To manage Git repositories in CodeCommit.
   * **AWSCodeBuildDeveloperAccess:** (AWS Managed Policy) To manage build projects in CodeBuild.
   * **AWSCodeDeployDeployerAccess:** (AWS Managed Policy) To manage deployments via CodeDeploy.
5. **For Infrastructure as Code (CloudFormation):**
   * **AWSCloudFormationFullAccess:** (AWS Managed Policy) To create/update/delete CloudFormation stacks. **Exercise extreme caution; developers can potentially alter critical infrastructure.** Often, a more restricted policy for specific stack prefixes is better.
6. **For Enhanced SSH Access (Session Manager):**
   * **AmazonSSMManagedInstanceCore:** (AWS Managed Policy) This policy needs to be attached to the **EC2 instance's IAM Role**, not the user.
   * **Session Manager specific permissions for the user/group:** Users need permissions like ssm:StartSession, ssm:TerminateSession to connect to instances via Session Manager. This is generally preferred over direct SSH key management for developers.

**Important Security Best Practices:**

* **Never give Root User credentials.** Always use IAM users.
* **Principle of Least Privilege:** Always grant only the permissions required to perform a task. If a developer needs fullAccess to a service, question why and if it can be narrowed down.
* **Environment Separation:** Ideally, production environments should be in separate AWS accounts or have extremely strict IAM policies, with developers having minimal (or no) direct access.
* **Tagging:** Encourage developers to tag their resources (e.g., Project: MyWebApp, Owner: cybdev@user) to help identify and manage resources. You can then write IAM policies that restrict actions based on these tags.
* **MFA (Multi-Factor Authentication):** Enforce MFA for all IAM users, especially for those with significant permissions.
* **Regular Audits:** Periodically review IAM policies and user activity to ensure permissions are still appropriate.

By following these guidelines, you can set up a secure and functional environment for your development team.

Can you strip off all the comments and give a cleaner JSON , so that I can paste it right away? Also, why don't you also provide a description, to be used in the description text area?

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": [

"ec2:DescribeInstances",

"ec2:RunInstances",

"ec2:StartInstances",

"ec2:StopInstances",

"ec2:RebootInstances",

"ec2:TerminateInstances",

"ec2:CreateTags",

"ec2:DeleteTags",

"ec2:DescribeKeyPairs",

"ec2:CreateKeyPair",

"ec2:DeleteKeyPair",

"ec2:DescribeVolumes",

"ec2:DescribeImages",

"ec2:DescribeInstanceStatus",

"ec2:DescribeSecurityGroups",

"ec2:DescribeSubnets",

"ec2:DescribeVpcs",

"ec2:DescribeNetworkInterfaces",

"ec2:CreateSecurityGroup",

"ec2:DeleteSecurityGroup",

"ec2:AuthorizeSecurityGroupIngress",

"ec2:RevokeSecurityGroupIngress",

"s3:ListAllMyBuckets",

"s3:ListBucket",

"s3:GetObject",

"s3:PutObject",

"s3:DeleteObject",

"s3:PutBucketVersioning",

"cloudwatch:DescribeAlarms",

"cloudwatch:GetMetricData",

"cloudwatch:ListMetrics",

"logs:DescribeLogGroups",

"logs:DescribeLogStreams",

"logs:FilterLogEvents",

"logs:GetLogEvents",

"logs:CreateLogGroup",

"logs:CreateLogStream",

"logs:PutLogEvents",

"iam:ListUsers",

"iam:ListRoles",

"iam:ListPolicies",

"iam:GetUser",

"iam:GetRole",

"iam:GetPolicy",

"aws-portal:ViewBilling",

"aws-portal:ViewUsage"

],

"Resource": "\*"

}

]

}

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**Let’s create the first IAM user!**

A screenshot of a computer

AI-generated content may be incorrect.

IAM User: **DevIamUser**

A screenshot of a computer

AI-generated content may be incorrect. A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A green rectangular box with white text

AI-generated content may be incorrect.

DevIam user info:  
A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer password

AI-generated content may be incorrect.

Upon logging in a s the DevIamUser, I see some console UI warning as”Cannot show list applications” , so AI explained:

* **If you DO need to use AWS Service Catalog (which is unlikely for a standard Spring Boot app deployment):**
  + You would need to modify your DevUserGroupPolicy to include the necessary servicecatalog permissions. For example, to just list applications, you'd add:

JSON

{

"Effect": "Allow",

"Action": [

"servicecatalog:ListApplications",

"servicecatalog:Describe\*" // If you need to see details about applications

],

"Resource": "\*"

}

(You'd add this new statement to the Statement array in your DevUserGroupPolicy in IAM).

* + However, for deploying a Spring Boot app on EC2, Service Catalog is typically used by larger organizations to standardize and govern the deployment of pre-approved products, not for direct developer interaction with raw EC2 instances.

**In conclusion: Don't worry about this particular warning unless your specific development workflow genuinely requires interacting with AWS Service Catalog.** Proceed with your EC2, S3, and CloudWatch tasks; those permissions should be fully functional.



Generate Audio Overview

**Tutorial:**

Instance creation cybdevinstance1

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Adding the Springboot port 9192 permission

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

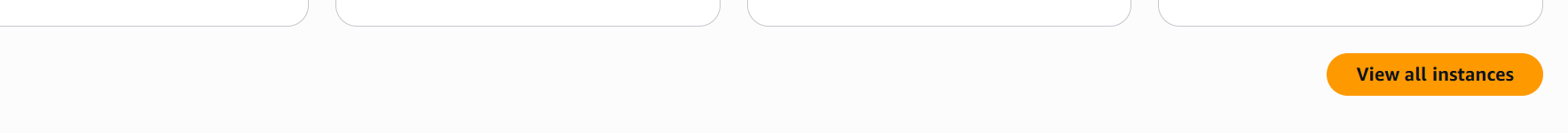
AI-generated content may be incorrect.

A screenshot of a computer

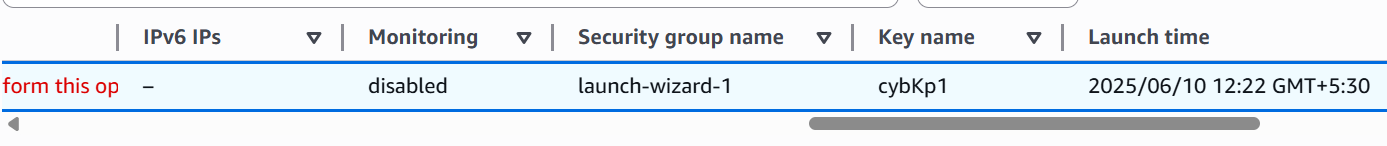
AI-generated content may be incorrect.

A green rectangular object with white text

AI-generated content may be incorrect.



A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Copied the Public IP and tried pinging…

A screenshot of a computer

AI-generated content may be incorrect.

copied Private IP4 address and tested again,

A computer screen with white text

AI-generated content may be incorrect.

Copied public DNS, ec2-13-200-222-82.ap-south-1.compute.amazonaws.com, and tried again…

Doesn’t work ☹

How can I test whether my instance is accessible publicly?

The public DNS provided is ec2-13-200-222-82.ap-south-1.compute.amazonaws.com

To test its public accessibility, you need to check if traffic can reach it on the specific ports your services are listening on. Given your Spring Boot app is on 9192 and SSH is on 22, these are the primary ports to verify.

Before you start testing, **the most critical step is to verify your EC2 Instance's Security Group settings.**

**Crucial Step 0: Verify Security Group (Firewall) Rules**

Even if your instance is running, if its Security Group doesn't allow inbound traffic on the ports you're testing, it will appear inaccessible.

1. **Go to EC2 Dashboard:** In the AWS Management Console, navigate to EC2.
2. **Select Your Instance:** Click on "Instances" and select cybdevinstance1 (or whatever you named it).
3. **Check Security Group:**
   * In the "Details" tab (or "Security" tab depending on console layout), find the "Security groups" link. Click on the associated security group (e.g., launch-wizard-1 or your custom one).
   * In the Security Group details page, go to the **"Inbound rules"** tab.
   * **Ensure you have rules that allow traffic from your source:**
     + **SSH (Port 22):** Type: SSH, Protocol: TCP, Port range: 22, Source: Your IP (most secure) or Anywhere IPv4 (0.0.0.0/0) (less secure, but confirms connectivity).
     + **Spring Boot App (Port 9192):** Type: Custom TCP, Protocol: TCP, Port range: 9192, Source: Your IP or Anywhere IPv4 (0.0.0.0/0).
     + **(Optional) HTTP (Port 80):** If your app eventually exposes something on standard HTTP port 80, open this too.
   * **If any rules are missing or incorrect, click "Edit inbound rules" to add/modify them.**

Let’s run some BASH script to check connectivity!

**Step00:** On Windows 11 machine, start **Gitbash**:

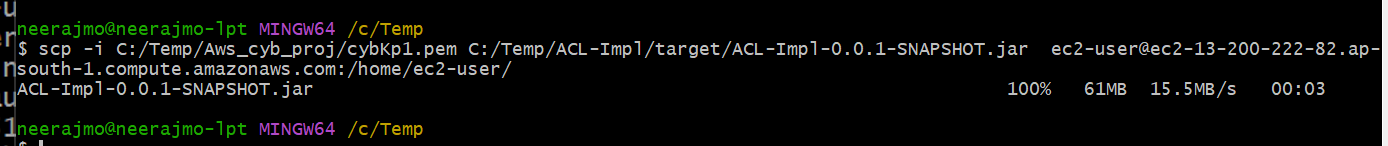
A black screen with a white text box

AI-generated content may be incorrect.

**Step01:** scp -i C:/Temp/Aws\_cyb\_proj/cybKp1.pem C:/Temp/ACL-Impl/target/ACL-Impl-0.0.1-SNAPSHOT.jar [ec2-user@ec2-13-200-222-82.ap-south-1.compute.amazonaws.com](mailto:ec2-user@ec2-13-200-222-82.ap-south-1.compute.amazonaws.com):/home/ec2-user/

Mistakenly, executed the above command on the instance itself 😊  
 A screenshot of a computer program

AI-generated content may be incorrect.

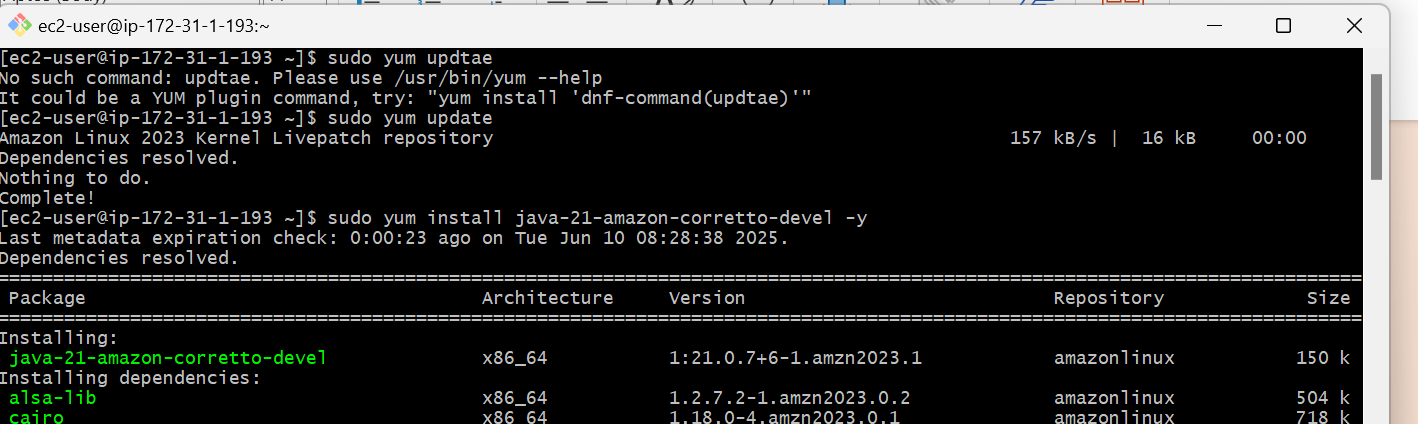
Run it from local Gitbash command,  
 

**Step 02:** Path to my key pair is **C:/Temp/Aws\_cyb\_proj**, so, executing, on the **local Gitbash** console, the following command,

$ ssh -i C:/Temp/Aws\_cyb\_proj/cybKp1.pem [ec2-user@ec2-13-200-222-82.ap-south-1.compute.amazonaws.com](mailto:ec2-user@ec2-13-200-222-82.ap-south-1.compute.amazonaws.com)  
 A computer screen with white text

AI-generated content may be incorrect.

On my instance , installed Java 21: sudo yum install java-21-amazon-corretto-devel -y

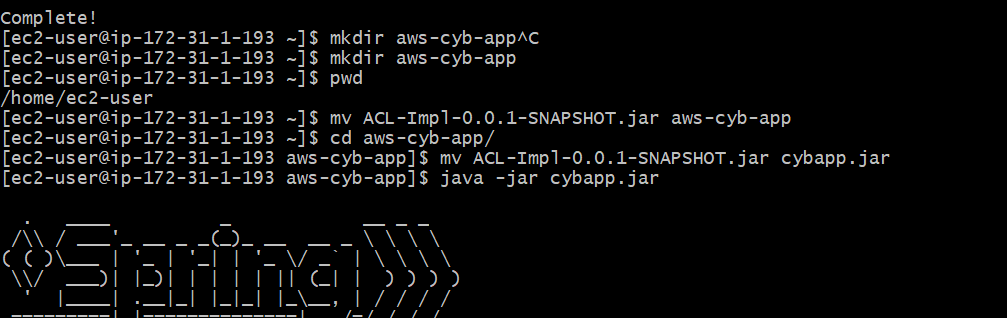


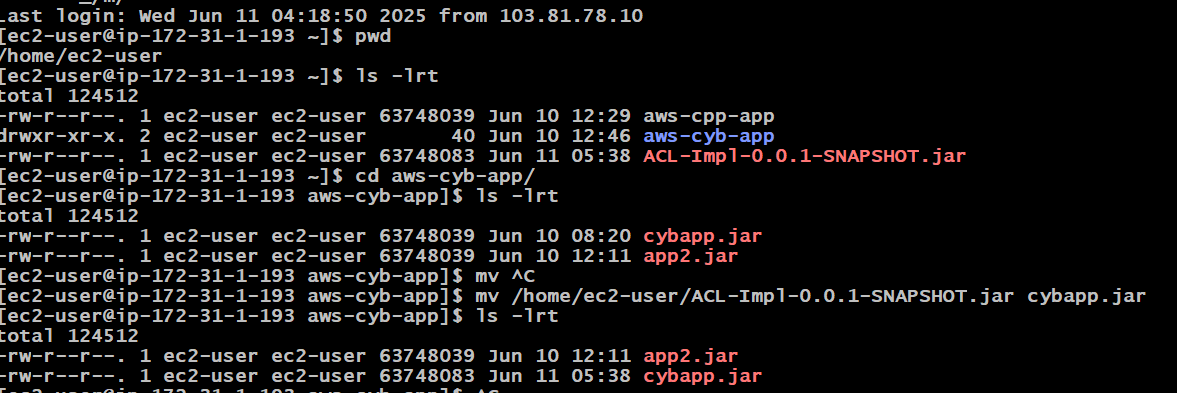
A screenshot of a computer

AI-generated content may be incorrect.

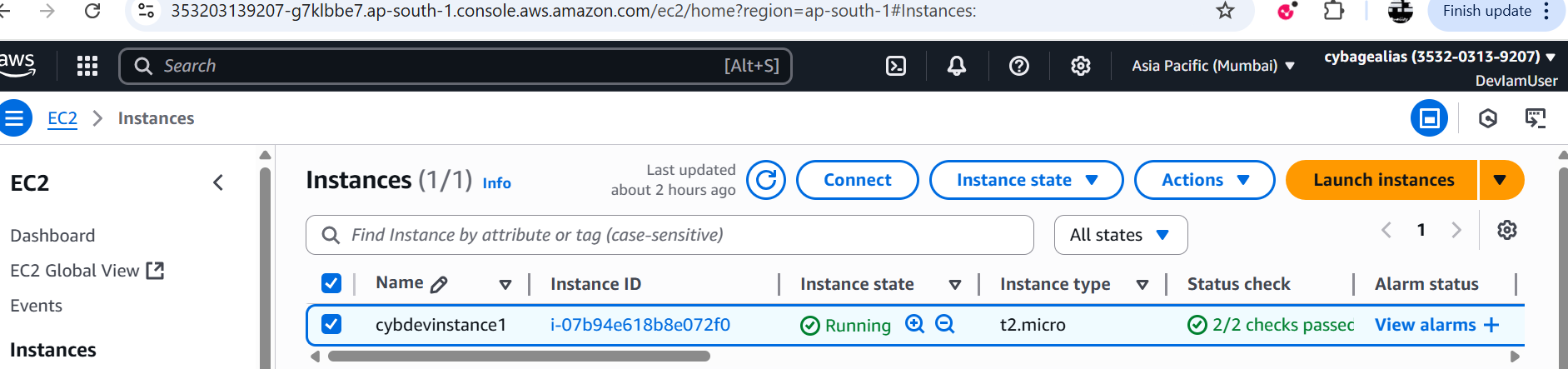
**Step 03: mkdir** aws-cyb-app  
Step 04: move the uploaded jar to this newly created dir. And name it to cybapp.jar  
 mv /home/ec2-user/ACL-Impl-0.0.1-SNAPSHOT.jar cybapp.jar

Step 05: Run other commands as seen :

* **mv** /home/ec2-user/ACL-Impl-0.0.1-SNAPSHOT.jar **aws-cyb-app**
* **cd** aws-cyb-app
* mv /home/ec2-user/ACL-Impl-0.0.1-SNAPSHOT.jar cybapp.jar  
    
  Confirm the **timestamp**!  
   **ls -lrt**

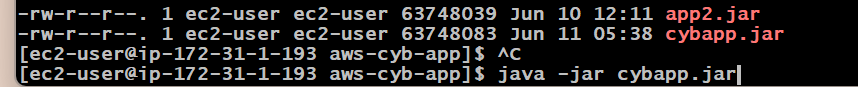


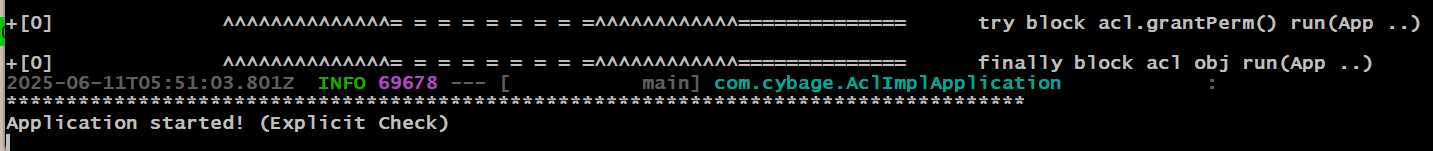
Step 05: Ensure that the instance is **Running** in **EC2** section (**Instance state**)



Step 06: Start the spring Boot application

**java -jar** cybapp.jar





Step 07: Testing in browser

public DNS of my app is: <http://ec2-13-200-222-82.ap-south-1.compute.amazonaws.com:9192>

A screenshot of a computer

AI-generated content may be incorrect.

Works!

My Google Application integration as of now, allows only the following Redirect URIs:

A screenshot of a computer

AI-generated content may be incorrect.

Experiment 2:

* + 1. From local github, execute:  
       scp -i C:/Temp/Aws\_cyb\_proj/cybKp1.pem C:/Temp/ACL-Impl/target/ACL-Impl-0.0.1-SNAPSHOT.jar [ec2-user@ec2-13-200-222-82.ap-south-1.compute.amazonaws.com:/home/ec2-user/aws-cpp-app](mailto:ec2-user@ec2-13-200-222-82.ap-south-1.compute.amazonaws.com:/home/ec2-user/aws-cpp-app)
    2. Log into Ec2 instance using: ssh -i C:/Temp/Aws\_cyb\_proj/cybKp1.pem [ec2-user@ec2-13-200-222-82.ap-south-1.compute.amazonaws.com](mailto:ec2-user@ec2-13-200-222-82.ap-south-1.compute.amazonaws.com)
    3. Go into directory cd **/aws-cpp-app**
    4. Find the files/directories in here **ls -lrt**

It shows : total 124512

-rw-r--r--. 1 ec2-user ec2-user 63748039 Jun 10 08:20 cybapp.jar

-rw-r--r--. 1 ec2-user ec2-user 63748039 Jun 10 12:11 aws-cyb-app

A computer screen with white text

AI-generated content may be incorrect.

Changed Google client Redirect URIs to:

A screenshot of a computer

AI-generated content may be incorrect.

But it still tries to redirect to goes to local, so, now, deleting this entry entirely,

A screenshot of a computer

AI-generated content may be incorrect.

It should go to amazon.in, but it doesn’t,  
A screenshot of a computer error message

AI-generated content may be incorrect.

Error 400: redirect\_uri\_mismatch

You can't sign in to this app because **it doesn't comply with Google's OAuth 2.0 policy**.  
  
**If you're the app developer, register the redirect URI in the Google Cloud Console.**

Request details: response\_type=code state=c12hyZ8IPodw0HAFlWMCzPb8m-tZDjKSBgPiwF3l-p8= redirect\_uri=http://localhost:9192/login/oauth2/code/google nonce=-gNyBLBax78oIhBdspQNAZqFRZVmaDnzifhp9sMVK-w client\_id=977120495583-lm4d46atdr4l061o7kjbjdp0uc6009fb.apps.googleusercontent.com access\_type=online scope=openid https://www.googleapis.com/auth/userinfo.email https://www.googleapis.com/auth/userinfo.profile flowName=GeneralOAuthFlow

[Related developer documentation](https://developers.google.com/identity/protocols/oauth2/web-server#authorization-errors-redirect-uri-mismatch)

So changing the Redirect\_uri’s again,

A screenshot of a computer screen

AI-generated content may be incorrect.

And, **finally,** again:

A screenshot of a computer

AI-generated content may be incorrect.

**This worked** on cell phone (essentially **out of company** network), as it was tested, with AWS-hosted, Spring Boot app.