



Certified Tech Developer

The Ultimate Degree

Infraestrutura II

Configurando nosso servidor para Deploy - Parte 1

É hora de agregar complexidade ao nosso Pipeline, com o objetivo de colocar em prática os conceitos de Integração Contínua e criação de artefatos a partir do nosso código.

Objetivo final da prática

- Criando nossa Key na AWS
- Criar o nosso ambiente de deploy com terraform
- Configurar nosso servidor para receber uma aplicação java
- Ajustar nossa aplicação para ser um aplicação spring boot
- Linkar o nosso ambiente AWS com a nossa pipeline no GitLab

Mãos à obra!

Ao final da prática anterior, foi proposto adicionar uma etapa de empacotamento (package) ao nosso Pipeline. Se você não conseguiu, vamos te passar como deve ficar o seu Pipeline

```
image: maven:3.6.3
```

```
stages:
```

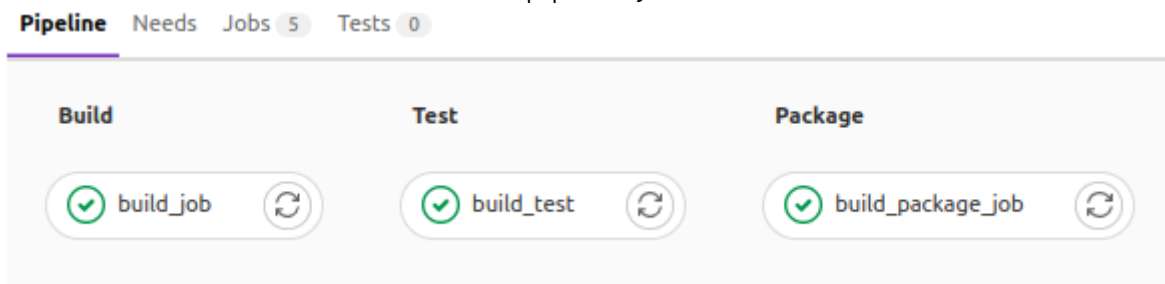
- build
- test
- package

```
variables:
  MAVEN_OPTS: -Dmaven.repo.local=.m2/repository
cache:
  paths:
    - .m2/repository
    - target
build_job:
  stage: build
  script:
    - echo "Maven compile started"
    - "mvn compile "

build_test:
  stage: test
  script:
    - echo "Maven test started"
    - "mvn test"

build_package_job:
  stage: package
  artifacts:
    paths:
      - target/*.jar
  script:
    - "mvn package"
```

Nosso Pipeline deveria ter sido executado satisfatoriamente, então quando vemos seus detalhes devemos vê-lo assim (vemos no menu CI/CD Pipelines #número do pipeline)



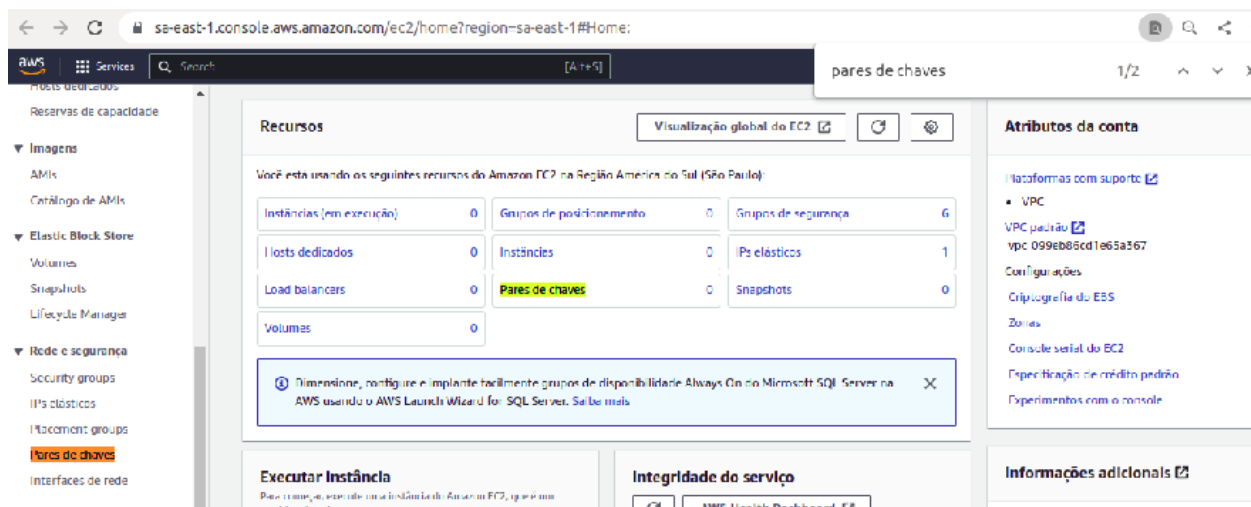
Criando nossa Key na AWS

Vamos começar a preparar nosso ambiente! Para isso vamos na console da AWS

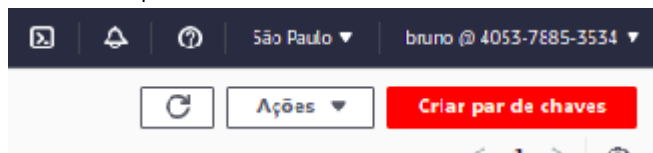
- **Selecione a região:** São Paulo



- Vá no console EC2 > Pares de chaves



- Vamos criar um par de chaves



- Com as seguintes informações:
 - **Nome:** <suas-iniciais>-mykey
 - **Tipo:** rsa
 - **Formato:** pem
- Aguarde o download e salve esta chave (vamos utilizá-la no decorrer do projeto)

Criar o nosso ambiente de deploy com terraform

Agora vamos provisionar nossa EC2 na AWS na VPC padrão, de forma bem simplificada. O repositório com o [código está aqui](#). Mas ele precisa de alguns ajustes de variáveis, para isso

- Clone o Repositório
- No console da aws (na região de são paulo, vamos em subnet) e vamos obter o **id de subnet publica**



	Name	ID da sub-rede	Estado	VPC	CIDR IPv4	CIDR IPv6
<input type="checkbox"/>	-vpc-private-sa-east-1	subnet-0491643203d8cc95	Available	vpc-0d32b0fd2c203a56 -vpc	10.0.2.0/24	-
<input type="checkbox"/>	-vpc-public-sa-east-1	subnet-084ef12a960b45ca9	Available	vpc-0d32b0fd2c203a56 -vpc	10.0.102.0/24	-
<input checked="" type="checkbox"/>	-	subnet-01a5976025a14708c	Available	vpc-099eb86cd1e65a367 sa-east-1	172.31.0.0/20	-
<input type="checkbox"/>	-vpc-private-sa-east-1	subnet-09b063b7ef7406a78	Available	vpc-0d32b0fd2c203a56 -vpc	10.0.1.0/24	-
<input type="checkbox"/>	-	subnet-0e6ec78cf927b548	Available	vpc-099eb86cd1e65a367 sa-east-1	172.31.16.0/20	-

- Após obter esse ID vamos no nosso repositório, infra > variables.tf e preencherlo com

```
# Região de execução: Vamos focar em são paulo
variable "my_region" {
    default = "sa-east-1"
}

# Subnet que acabamos de obter
variable "my_subnet_id" {
    default = "subnet-..."
}

# Nome da nossa chave gerada no passo 1 (vai ser perguntado no momento
# de execução do terraform)
variable "keypair_name" {
    description = "Informe o nome da sua chave cadastrada na AWS"
}

# Nome da nossa instancia (vai ser perguntado no momento de execução
# do terraform)
variable "instance_name" {
    description = "Informe o nome da seu servidor Backend"
}

# Para definirmos o ip publico da nossa EC2
variable "enable_public_ip" {
    default = true
}
```

Agora vamos executar nosso terraform.
Lembre-se:

- init

- plan
- apply

```
Only 'yes' will be accepted to approve.

Enter a value: yes

aws_instance.backend: Creating...
aws_instance.backend: Still creating... [10s elapsed]
aws_instance.backend: Still creating... [20s elapsed]
aws_instance.backend: Creation complete after 22s [id=i-0d1c9d253c8a984ad]

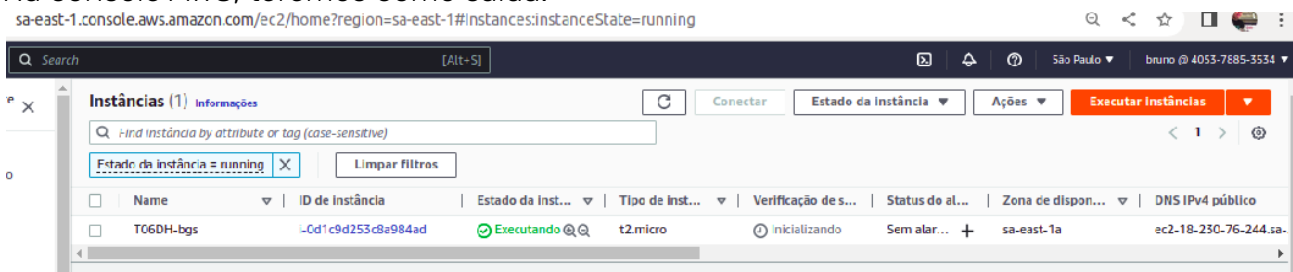
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Outputs:

ec2_ip_publico = |
  "18.230.76.244",
]
bgsouza@RDL724710 ~/Documentos/poc/dh/gitlab/aula16/infra
```

Importante: Copie o IP exibido no console do terraform para utilizarmos no proximo passo

Na console AWS, teremos como saída:



Configurar nosso servidor para receber uma aplicação java

Dentro do mesmo repositório recém clonado, há uma pasta chamada **configs-server-java** e nela tem um ansible que utilizaremos para configurar nosso servidor java na EC2 recém provisionada

Nessa EC2 iremos instalar:

- Java 11
- Um serviço systemd para controlar nossa aplicação java

- o não iremos configurar tomcat/jboss/jetty/..., vamos no mais simples, porém fica o desafio :)

Vamos ajustar nosso hosts com o IP da nossa EC2 e o path da nossa .pem no **configs-server-java > inventory > hosts.yml**

```
[all:vars]
ansible_python_interpreter=/usr/bin/python3
ansible_ssh_common_args='-o StrictHostKeyChecking=accept-new'
ansible_ssh_private_key_file=<path-para-sua-key>/<nome-da-sua-key>.pem #
diretório onde está sua .pem

[backends_servers]
18.230.76.244 # IP do passo do terraform aqui
```

Agora iremos analisar nosso playbook:

```
- hosts: backends_servers
  become: yes

  vars:
    service_name: dhapp.service

  tasks:
    - name: Executando um apt-get update
      apt:
        update_cache: true

    - name: instalando Java
      apt:
        name: openjdk-11-jdk
        state: latest
        install_recommends: no
        update_cache: yes

    - name: instalando APT Transport HTTPS
      apt:
        name: apt-transport-https
        state: present

    - name: Exibindo a versão do java
```



```
shell: java -version 2>&1 | grep version | awk '{print $3}' | sed
's/"//g'
changed_when: false
register: java_result

- debug:
    msg: "{{ java_result.stdout }}"

- name: Copiando o Service
  copy:
    src: "{{ service_name }}"
    dest: "/etc/systemd/system/{{ service_name }}"

- name: Reiniciando o daemon-service
  shell: sudo systemctl daemon-reload

- name: Registrando o service
  shell: sudo systemctl enable "{{ service_name }}"
```

Vamos olhar o serviço systemd que será criado:

```
[Unit]
Description=REST Service
After=syslog.target

[Service]
User=ubuntu
ExecStart=/usr/bin/java -jar /home/ubuntu/dhinfra-1.0-SNAPSHOT.jar
SuccessExitStatus=143

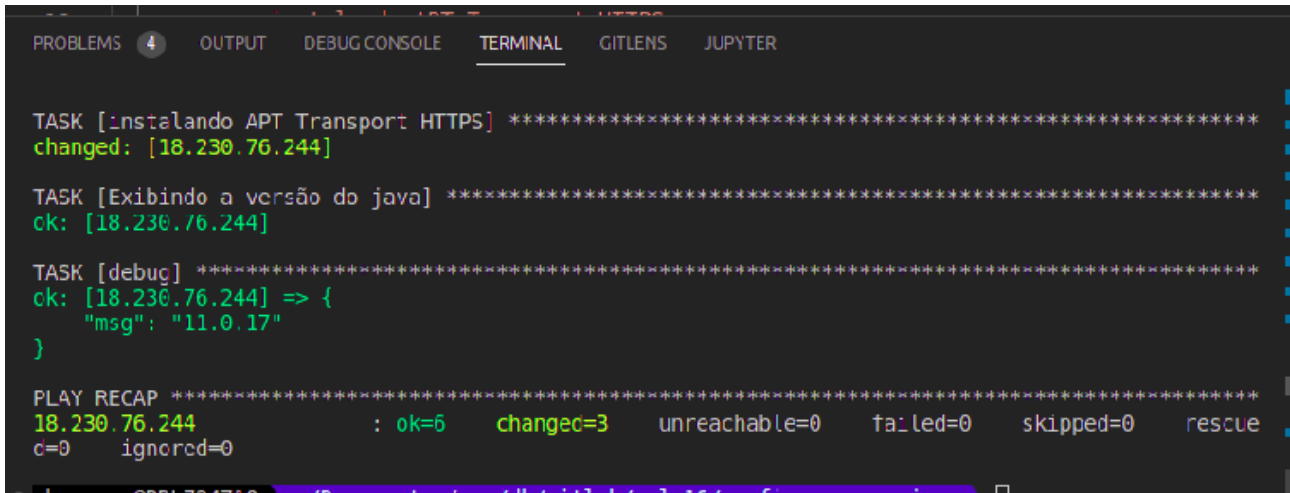
[Install]
WantedBy=multi-user.target
```

Systemd é um sistema init e gerenciador de sistema que se tornou o novo padrão para distribuições Linux. Devido à sua forte adoção, familiarizar-se com o systemd vale muito a pena, pois irá tornar a administração de servidores consideravelmente mais fácil. Aprender sobre as ferramentas e daemons que compõem o systemd e como usá-los irá ajudar a entender melhor o poder, flexibilidade e capacidades que ele oferece ou, pelo menos, facilitará o seu trabalho.

Mais sobre os systemd [aqui](#) e [aqui](#)

Vamos executar nosso ansible:

```
USER@DHINFRA: $ ansible-playbook -u ubuntu ec2.yml
```



```
PROBLEMS 4 OUTPUT DEBUG CONSOLE TERMINAL GITLENS JUPYTER

TASK [instalando APT Transport HTTPS] *****
changed: [18.230.76.244]

TASK [Exibindo a versão do java] *****
ok: [18.230.76.244]

TASK [debug] *****
ok: [18.230.76.244] => {
  "msg": "11.0.17"
}

PLAY RECAP *****
18.230.76.244 : ok=6  changed=3  unreachable=0  failed=0  skipped=0  rescue
c=0  ignored=0
```

Ajustar nossa aplicação para ser um aplicação springboot

Precisamos agora converter nossa aplicação java para uma aplicação [spring boot](#) para fazermos nosso deploy em produção.

Para isso vamos no repositório **dhinfra** onde estão os códigos java e vamos fazer as seguintes modificações

- No pom.xml, vamos trocar ele todo por essa versão

```
<?xml version="1.0" encoding="UTF-8"?>

<project                                xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.dhinfra.app</groupId>
```




```
<artifactId>dhinfra</artifactId>
<version>1.0-SNAPSHOT</version>

<name>dhinfra</name>
<!-- FIXME change it to the project's website -->
<url>http://www.example.com</url>
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.7.0</version>
  <relativePath/>
  <!-- lookup parent from repository -->
</parent>
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <java.version>8</java.version>
</properties>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>

  <!-- novas dependencias -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
  <!-- novas dependencias -->

</dependencies>
```



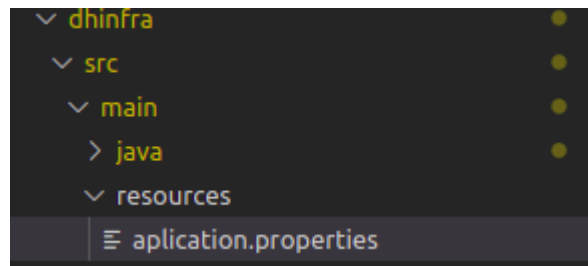
```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
</project>
```

- No src/main/java/com/dhinfra/app/App.java, vamos colocar:

```
package com.dhinfra.app;

import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
/**
 * Hello world!
 *
 */
@SpringBootApplication
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
        SpringApplication.run(App.class, args);
    }
}
```

- Na raiz do src/main/, vamos criar
 - um diretório chamado: **resources**
 - e dentro desse diretório, criar um arquivo vazio chamado: **application.properties** (vazio)



- No src/test/java/com/dhinfra/app/AppTest.java, vamos trocá-lo por

```
package com.dhinfra.app;
import static org.junit.Assert.assertTrue;
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;

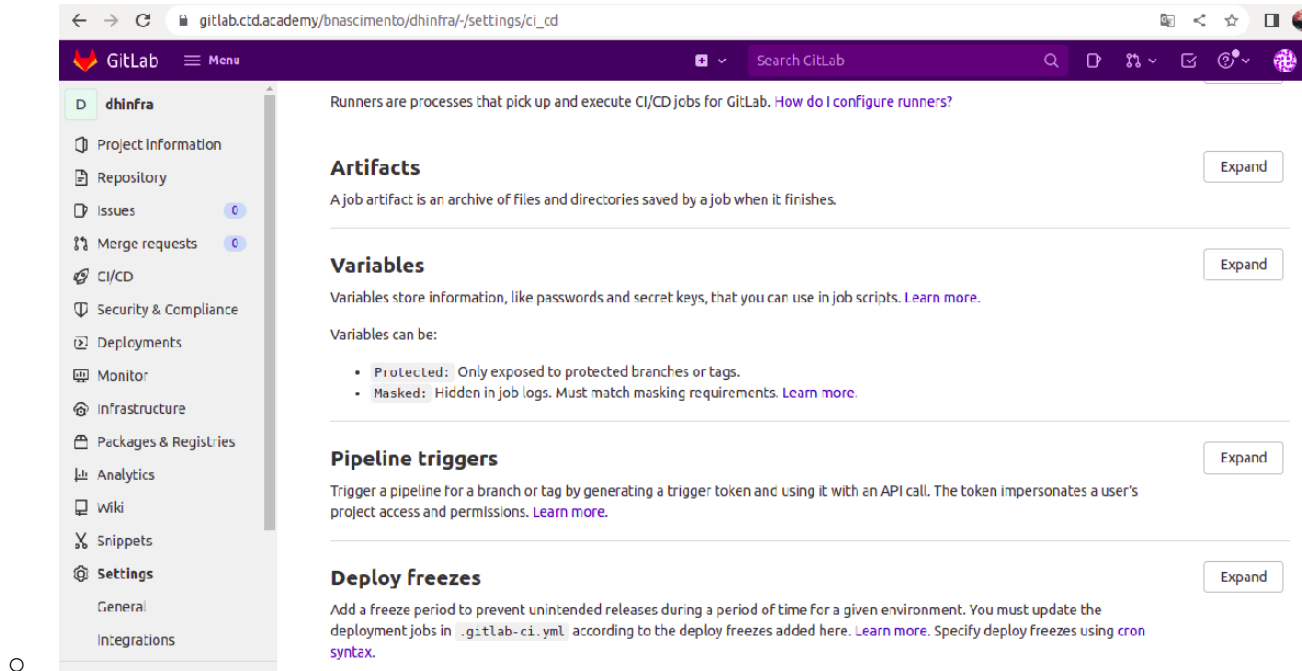
/**
 * Unit test for simple App.
 */
@SpringBootTest
public class AppTest
{
    /**
     * Rigorous Test :-)
     */
    @Test
    public void shouldAnswerWithTrue()
    {
        assertTrue( true );
    }
}
```

Linkar o nosso ambiente AWS com a nossa pipeline no GitLab

Agora, para finalizar nossa primeira etapa, vamos no <https://gitlab.ctd.academy/> > nosso repo **dhifra** e começar a configurar nosso repositório com o servidor.

Para isso vamos:

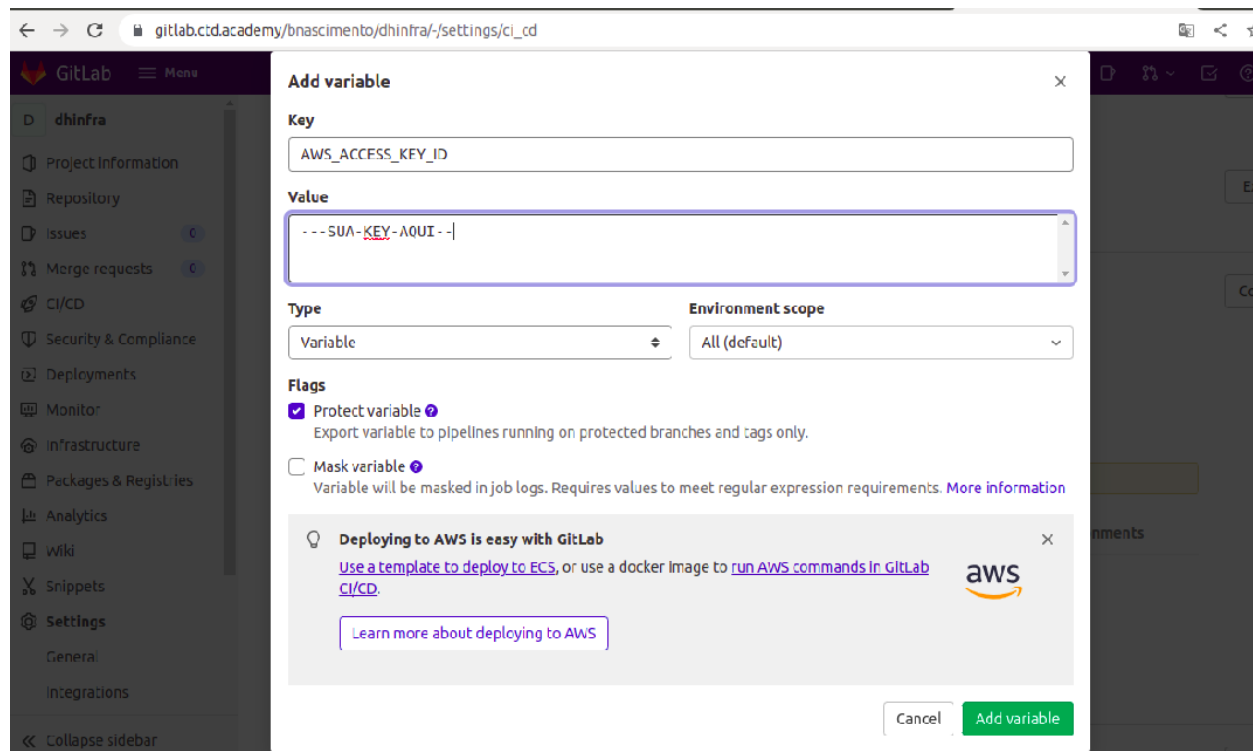
- No repositório > Settings > CI/CD Variables



- Em variables, clique em “Expand” e depois em “Add Variable”

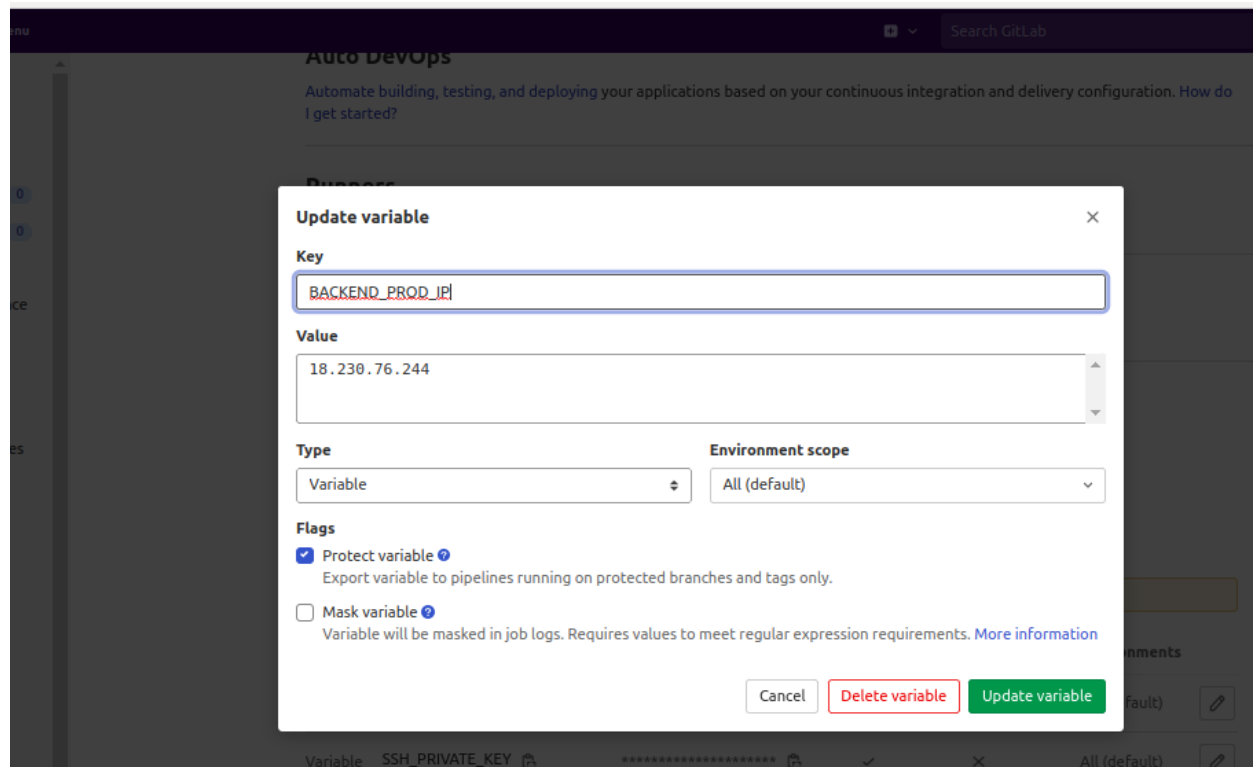


- Vamos criar uma variável chamada “**AWS_ACESS_KEY_ID**”, seu conteúdo é o que tem no **.pem** gerado no passo 1
 - `cat ~/<minhas-iniciais>-mykey.pem`
 - copia o conteúdo
 - cola na variável do gitlab



- Clique em “Add variable”
- Agora vamos adicionar uma nova variável chamada **BACKEND_PROD_IP**, contendo o IP público da nossa EC2

gitlab.ctd.academy/bnascimento/dhinfra/-/settings/ci_cd





Continua...

No próximo exercício iremos configurar nossa pipeline de Deploy e fazer nossa aplicação chegar na EC2 e ficar visível na porta 8080