TD4 – Déploiement d'un backend Spring Boot avec Ansible et GitHub Actions

Objectifs

- Déployer une application Spring Boot sur une instance EC2
- Utiliser Ansible pour automatiser la configuration
- Connecter l'application à une base RDS PostgreSQL existante
- Récupérer un secret depuis AWS Secrets Manager
- Mettre en place un pipeline CI/CD avec GitHub Actions
- Comprendre les limites de SSH et introduire AWS Systems Manager

Contexte

Vous travaillez dans une startup en cours de croissance.

L'infrastructure AWS a été sécurisée lors des TD précédents :

- VPC avec sous-réseaux publics et privés (TD2)
- Application Load Balancer (TD2)
- RDS PostgreSQL en sous-réseau privé (TD3)
- Secrets Manager pour les mots de passe (TD3)
- CloudTrail actif pour la traçabilité (TD3)

Un backend Spring Boot doit maintenant être déployé automatiquement sur cette infrastructure.

Objectifs métier :

- déploiement fiable et reproductible
- aucun secret exposé dans le code
- traçabilité complète des déploiements

Pré-requis

- Infrastructure TD2 et TD3 opérationnelle
- Instance EC2 accessible via SSH
- Dépôt GitHub disponible avec les droits d'administration
- Java 17 et Maven installés sur le poste de travail

Rappel sur les environnements d'exécution :

- Le poste de travail et le runner CI/CD ont besoin de Java et Maven pour **compiler** l'application
- L'instance EC2 a besoin de Java uniquement pour **exécuter** le jar
- Maven n'a pas sa place sur l'EC2 : on y dépose uniquement le jar déjà compilé

1. Compréhension de l'application

L'application Spring Boot utilise un fichier de configuration pour se connecter à la base de données.

Fichier : `app/src/main/resources/application.properties`

```
server.port=8080

spring.datasource.url=jdbc:postgresql://DB_HOST:5432/app
spring.datasource.username=app_user
spring.datasource.password=CHANGE_ME

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=false
```

Questions de compréhension :

- Pourquoi le mot de passe en clair dans ce fichier pose-t-il un problème de sécurité ?
- Que se passe-t-il si ce fichier est commité dans Git ?
- Quel service AWS permet de corriger cette situation ?

2. Build de l'application

Le build se fait sur le poste de travail. L'EC2 ne reçoit que le jar final.

Fichier : `app/pom.xml`

```
<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.example</groupId>
  <artifactId>demo</artifactId>
  <version>1.0</version>
  <packaging>jar</packaging>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.2.0</version>
  </parent>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
```

```
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
    </dependency>
  </dependencies>

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

</project>
```

Compiler l'application depuis le répertoire `app/` sur le poste de travail :

```
cd app
mvn clean package -DskipTests
```

Vérifier que le fichier `app/target/demo-1.0.jar` est bien généré :

```
ls -lh app/target/demo-1.0.jar
```

Questions de compréhension :

- Pourquoi utilise-t-on `-DskipTests` ici ?
- Dans quel cas ne faudrait-il pas ignorer les tests ?
- Où se trouve le jar produit par Maven ? Pourquoi ce répertoire est-il généralement dans `.gitignore` ?

3. Déploiement manuel

Avant d'automatiser, on effectue un déploiement manuel pour valider que l'application fonctionne sur l'EC2.

Cette section introduit volontairement un problème que vous devrez identifier.

Étape 1 : Vérifier que Java est disponible sur l'instance EC2.

Se connecter à l'instance :

```
ssh -i ~/.ssh/ma-cle.pem ec2-user@EC2_IP
```

Depuis l'instance, vérifier Java :

```
java -version
```

Si Java n'est pas installé, l'installer manuellement :

```
sudo yum install -y java-17-amazon-corretto

# Vérifier l'installation
java -version
```

Quitter la session SSH :

```
exit
```

Étape 2 : Copier le jar sur l'instance EC2 depuis le poste de travail :

```
scp -i ~/.ssh/ma-cle.pem \
  app/target/demo-1.0.jar \
  ec2-user@EC2_IP:/home/ec2-user/demo.jar
```

Étape 3 : Se connecter à l'instance et lancer l'application :

```
ssh -i ~/.ssh/ma-cle.pem ec2-user@EC2_IP

# Sur l'instance EC2
java -jar /home/ec2-user/demo.jar \
  --spring.datasource.url=jdbc:postgresql://RDS_HOST:5432/app \
  --spring.datasource.username=app_user \
```

```
  --spring.datasource.password=CHANGE_ME
```

Questions d'analyse :

- Quels sont les problèmes concrets de cette méthode de déploiement ?
- Que se passe-t-il si l'instance EC2 redémarre ?
- Pourquoi passer le mot de passe en argument de ligne de commande est-il risqué ?
- Quelle commande Linux permet de voir les arguments passés à un processus en cours d'exécution ?

4. Problème volontaire – L'application ne démarre pas

En lançant l'application avec la configuration par défaut, vous obtenez une erreur de ce type :

```
org.postgresql.util.PSQLException: Connection to DB_HOST:5432 refused.
```

Analysez l'erreur avant de passer à la suite :

- Pourquoi l'application ne peut-elle pas se connecter à `DB_HOST` ?
- Quelle valeur devrait remplacer `DB_HOST` ?
- Comment vérifier que l'instance EC2 peut joindre le RDS ?

Commande utile pour tester la connectivité réseau depuis l'instance EC2 :

```
nc -zv RDS_HOST 5432
```

- Si cette commande échoue, quel composant AWS est probablement en cause ?
- Quel Security Group faut-il vérifier en priorité ?

Ne cherchez pas la solution immédiatement. Identifiez d'abord l'origine exacte du problème.

5. Déploiement avec Ansible

On automatise maintenant le déploiement pour qu'il soit reproductible.

Ansible se connecte à l'EC2 via SSH depuis le poste de travail. Il installe Java si nécessaire, dépose le jar, et configure le service.

Fichier : `ansible/inventory.ini`

```
[web]
EC2_IP ansible_user=ec2-user ansible_ssh_private_key_file=~/.ssh/ma-cle.pem
```

Fichier : `ansible/ansible.cfg`

```
[defaults]
host_key_checking = False
stdout_callback = yaml
```

Pourquoi `host_key_checking = False` est-il utile ici ?

Serait-ce acceptable dans un environnement de production ? Pourquoi ?

Fichier : `ansible/deploy.yml`

```
---
- hosts: web
  become: true

  vars:
    app_dir: /opt/demo
    app_jar: demo.jar
    app_user: appuser

  tasks:
    - name: Créer l'utilisateur applicatif
      user:
        name: "{{ app_user }}"
        system: true
        shell: /sbin/nologin
        create_home: false

    - name: Créer le répertoire de l'application
      file:
        path: "{{ app_dir }}"
        state: directory
        owner: "{{ app_user }}"
        mode: '0755'

    - name: Installer Java 17
      yum:
        name: java-17-amazon-corretto
        state: present

    - name: Copier le jar
      copy:
        src: ../app/target/demo-1.0.jar
        dest: "{{ app_dir }}/{{ app_jar }}"
        owner: "{{ app_user }}"
        mode: '0644'
```

Lancer le playbook depuis le poste de travail :

```
ansible-playbook -i ansible/inventory.ini ansible/deploy.yml
```

Vérifier que le jar est bien présent sur l'instance :

```
ssh -i ~/.ssh/ma-cle.pem ec2-user@EC2_IP ls -lh /opt/demo/
```

Questions de compréhension :

- Pourquoi crée-t-on un utilisateur dédié `appuser` plutôt que de lancer l'application en tant que `root` ou `ec2-user` ?
- Pourquoi Ansible installe-t-il Java sur l'EC2 alors qu'on l'a peut-être déjà installé manuellement à la section 3 ?
- Qu'est-ce que l'idempotence dans Ansible ? Pourquoi est-ce important ici ?

6. Problème volontaire – L'application ne se connecte pas à la base

Le jar est déployé et Java est installé, mais l'application ne peut pas démarrer correctement car les variables de connexion à la base de données ne sont pas fournies.

Avant de regarder la section suivante :

- Quel élément manque dans le playbook actuel ?
- Comment Ansible peut-il récupérer un secret depuis AWS Secrets Manager ?
- Pourquoi ne doit-on pas mettre le mot de passe directement dans le playbook ?
- Pourquoi ne pas le mettre non plus dans un fichier de variables Ansible commité dans Git ?

7. Correction – Récupération du secret et service systemd

On complète le playbook avec la récupération du secret, la génération du fichier de configuration, et le démarrage via systemd.

Fichier : `ansible/deploy.yml` (version complète)

```
---
```

```
- hosts: web
  become: true

  vars:
    app_dir: /opt/demo
    app_jar: demo.jar
    app_user: appuser
    rds_host: RDS_HOST
    secret_id: td3-db-password

  tasks:
    - name: Créer l'utilisateur applicatif
      user:
        name: "{{ app_user }}"
        system: true
        shell: /sbin/nologin
        create_home: false

    - name: Créer le répertoire de l'application
      file:
        path: "{{ app_dir }}"
        state: directory
        owner: "{{ app_user }}"
        mode: '0755'

    - name: Installer Java 17
      yum:
        name: java-17-amazon-corretto
        state: present

    - name: Copier le jar
      copy:
        src: ../app/target/demo-1.0.jar
        dest: "{{ app_dir }}/{{ app_jar }}"
        owner: "{{ app_user }}"
        mode: '0644'

    - name: Récupérer le secret depuis Secrets Manager
      shell: >
        aws secretsmanager get-secret-value
        --secret-id {{ secret_id }}
        --query SecretString
        --output text
      register: db_password
      no_log: true

    - name: Créer le fichier de configuration de l'application
      copy:
        dest: "{{ app_dir }}/application.properties"
        owner: "{{ app_user }}"
        mode: '0600'
        content: |
          server.port=8080
          spring.datasource.url=jdbc:postgresql://{{ rds_host }}:5432/app
          spring.datasource.username=app_user
          spring.datasource.password={{ db_password.stdout }}
```

```
          spring.jpa.hibernate.ddl-auto=update
      no_log: true

    - name: Déployer le service systemd
      copy:
        dest: /etc/systemd/system/demo.service
        content: |
          [Unit]
          Description=Demo Spring Boot Application
          After=network.target

          [Service]
          User={{ app_user }}
          WorkingDirectory={{ app_dir }}
          ExecStart=/usr/bin/java -jar {{ app_dir }}/{{ app_jar }} \
            --spring.config.location={{ app_dir }}/application.properties
          SuccessExitStatus=143
          Restart=on-failure
          RestartSec=10

          [Install]
          WantedBy=multi-user.target

    - name: Recharger systemd
      systemd:
        daemon_reload: true

    - name: Activer et démarrer l'application
      systemd:
        name: demo
        state: restarted
        enabled: true
```

Questions de compréhension :

- Pourquoi utilise-t-on `no_log: true` sur la tâche de récupération du secret ?
- Pourquoi le fichier `application.properties` a-t-il les permissions `0600` ?
- Quel rôle IAM doit avoir l'instance EC2 pour que la commande `aws secretsmanager` fonctionne ?
- Pourquoi ne jamais stocker ce secret dans le dépôt Git, même dans un fichier de variables Ansible ?

8. Pourquoi systemd plutôt que nohup

<WRAP round help> Comparez les deux approches :

```
# Approche nohup — ne pas utiliser en production
nohup java -jar demo.jar &
```

- Que se passe-t-il avec nohup si l'instance EC2 redémarre ?
- Que se passe-t-il avec nohup si l'application plante ?
- Comment systemd rés