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 ou Session Manager
- Dépôt GitHub disponible avec les droits d'administration
- Java 17 et Maven installés sur le poste de travail

Rappel sur les environnements d'exécution :

- Java et Maven sont nécessaires sur le **poste de travail** pour compiler l'application
- Java est nécessaire sur **l'EC2** 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
```

Ce fichier sera commité dans Git avec le reste du code source.

- Quelles sont les conséquences concrètes si un dépôt contenant ce fichier devient public, même brièvement ?
- Un attaquant qui récupère ce fichier a-t-il accès à la base de données directement ? Quels autres éléments lui manquent-il ?
- Pourquoi externaliser ce secret dans Secrets Manager ne suffit pas si le code qui le récupère est lui-même mal sécurisé ?

2. Build de l'application

Le fichier Maven minimal pour construire le projet :

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/` :

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

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

Dans un pipeline CI/CD en production, ignorer les tests avec `-DskipTests` est une mauvaise pratique.

- Dans quelle situation précise serait-il acceptable de l'utiliser malgré tout ?
- Quels types de tests seraient les plus critiques à conserver pour une application qui manipule une base de données ?

3. Déploiement manuel

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

Copier le jar sur l'instance EC2 :

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

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
```

Sur un système Linux, les arguments passés à un processus sont visibles via `ps aux`.

- Quelle est la conséquence concrète de passer le mot de passe en argument de ligne de commande sur un serveur multi-utilisateurs ?
- Cette méthode de déploiement crée une dépendance forte entre le déploiement et la présence d'un opérateur humain. Quels risques cela introduit-il à l'échelle ?

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.
```

Avant de chercher la solution :

- L'erreur indique `refused` et non `timed out`. Quelle différence technique cela implique-t-il sur l'origine du problème ?
- Listez les couches réseau à vérifier dans l'ordre pour isoler la cause : application, Security Group, sous-réseau, routage.

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

```
nc -zv RDS_HOST 5432
```

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.

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
```

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 :

```
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/
```

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

Le jar est déployé 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 :

- Le playbook copie le jar mais ne fournit aucune configuration à l'application. Quels mécanismes Spring Boot permettent d'injecter cette configuration sans modifier le jar lui-même ?
- Ansible peut exécuter des commandes AWS CLI sur la machine cible. Quelles permissions IAM l'instance EC2 doit-elle posséder pour que cette approche fonctionne sans clé d'accès statique ?
- Quelle différence y a-t-il entre stocker le mot de passe dans un fichier de variables Ansible chiffré avec Ansible Vault et le stocker dans Secrets Manager ? Dans quel contexte chaque approche est-elle préférable ?

7. Correction – Récupération du secret depuis Secrets Manager

On ajoute la récupération du secret et le lancement de l'application via un service 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
```

- `no_log: true` masque la valeur dans les logs Ansible, mais le secret est tout de même écrit sur le disque dans `application.properties`. Quelles mesures complémentaires permettraient de réduire la durée d'exposition de ce secret sur le système de fichiers ?
- Le service systemd démarre l'application avec l'utilisateur `appuser` qui n'a pas de shell. Quel est l'intérêt de cette contrainte par rapport à un utilisateur standard ?
- Si Secrets Manager retourne le secret sous forme de JSON `{"password":"valeur"}` plutôt qu'une chaîne brute, que faut-il modifier dans la tâche de récupération ?

8. Pourquoi systemd plutôt que nohup

Comparez les deux approches du point de vue opérationnel :

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

- Un processus lancé avec `nohup` plante silencieusement à 3h du matin. Décrivez la chaîne d'événements jusqu'à la détection du problème sans systemd, puis avec systemd.
- systemd expose des métriques sur les redémarrages successifs d'un service. En quoi cette information est-elle utile pour distinguer un bug applicatif d'un problème d'infrastructure ?

9. Mise en place du pipeline CI/CD

On automatise maintenant le build et le déploiement via GitHub Actions.

Fichier : `.github/workflows/deploy.yml`

```
name: Build and Deploy

on:
  push:
    branches: [ "main" ]

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest

    steps:
      - name: Récupérer le code
        uses: actions/checkout@v4
```

```
      - name: Configurer Java 17
        uses: actions/setup-java@v4
        with:
          distribution: temurin
          java-version: 17

      - name: Compiler l'application
        run: |
          cd app
          mvn clean package -DskipTests

      - name: Configurer les credentials AWS
        uses: aws-actions/configure-aws-credentials@v4
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: eu-west-1

      - name: Préparer la clé SSH
        run: |
          mkdir -p ~/.ssh
          echo "${{ secrets.EC2_SSH_KEY }}" > ~/.ssh/ma-cle.pem
          chmod 600 ~/.ssh/ma-cle.pem

      - name: Installer Ansible
        run: |
          sudo apt-get update -q
          sudo apt-get install -y ansible

      - name: Déployer avec Ansible
        run: |
          ansible-playbook \
            -i ansible/inventory.ini \
            ansible/deploy.yml \
            -e "rds_host=${{ secrets.RDS_HOST }}"
```

10. Problème volontaire – Le pipeline échoue

En l'état, le pipeline GitHub Actions va échouer pour plusieurs raisons.

Identifiez les problèmes avant de passer à la section suivante :

- Le runner GitHub Actions est une machine éphémère hébergée chez GitHub. Quelles contraintes réseau cela implique-t-il pour joindre une instance EC2 dans un sous-réseau public ?
- L'inventaire Ansible contient `EC2_IP` en dur. Quelles sont les deux situations concrètes dans lesquelles cette valeur devient invalide sans que personne ne s'en aperçoive immédiatement ?
- Si la vérification de la clé SSH hôte est active, le pipeline se bloque en attente d'une confirmation interactive. Pourquoi ce comportement est-il particulièrement problématique dans un contexte CI/CD ?

11. Correction – Configuration des secrets GitHub

Dans le dépôt GitHub, aller dans Settings > Secrets and variables > Actions.

Créer les secrets suivants :

- `AWS_ACCESS_KEY_ID` : clé d'accès du compte AWS
- `AWS_SECRET_ACCESS_KEY` : clé secrète correspondante
- `EC2_SSH_KEY` : contenu complet du fichier `.pem` de la clé SSH
- `RDS_HOST` : endpoint du RDS récupéré depuis la console AWS

Mettre à jour l'inventaire Ansible avec l'IP réelle de l'instance :

Fichier : `ansible/inventory.ini`

```
[web]
EC2_IP_REELLE 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
remote_user = ec2-user
```

- Les secrets GitHub sont chiffrés au repos et masqués dans les logs. Malgré cela, un secret GitHub n'offre pas les mêmes garanties que Secrets Manager. Identifiez au moins deux différences en termes de contrôle d'accès et d'auditabilité.
- Ce pipeline utilise une clé d'accès AWS statique. Quelle alternative AWS permettrait d'authentifier GitHub Actions sans clé longue durée, et sur quel mécanisme repose-t-elle ?

12. Vérification du déploiement

Vérifier que l'application est bien démarrée sur l'instance EC2 :

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

# Vérifier le statut du service
sudo systemctl status demo

# Consulter les logs de l'application
sudo journalctl -u demo -n 50 --no-pager
```

Vérifier que l'application répond via l'ALB :

```
curl http://ALB_DNS/actuator/health
```

- L'accès direct à l'IP de l'instance EC2 fonctionne, mais l'accès via l'ALB retourne une erreur 502. Sans regarder les logs, listez les causes possibles dans l'ordre le plus probable.
- Dans une architecture avec plusieurs instances EC2 derrière l'ALB, pourquoi le fait de vérifier uniquement une instance ne suffit-il pas à valider le déploiement ?

13. Amélioration sécurité – Remplacer SSH par Session Manager

Objectif de cette section :

- supprimer le port 22 du Security Group
- éviter la gestion des clés SSH
- améliorer la traçabilité des connexions

13.1 Configuration IAM pour Session Manager

Fichier : `terraform/ec2_ssm.tf`

```
resource "aws_iam_role" "ec2_ssm_role" {
  name = "ec2-ssm-role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Effect = "Allow"
      Principal = {
        Service = "ec2.amazonaws.com"
      }
      Action = "sts:AssumeRole"
    }]
  })
}

resource "aws_iam_role_policy_attachment" "ssm_core" {
  role       = aws_iam_role.ec2_ssm_role.name
  policy_arn = "arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore"
}

resource "aws_iam_instance_profile" "ec2_ssm_profile" {
  name = "ec2-ssm-profile"
  role = aws_iam_role.ec2_ssm_role.name
}
```

Appliquer la configuration Terraform :

```
terraform apply
```

Associer le profil IAM à l'instance EC2 existante depuis la console AWS : EC2 > Instance > Actions > Security > Modify IAM role

13.2 Test de connexion via Session Manager

Tester la connexion sans clé SSH :

```
aws ssm start-session --target INSTANCE_ID
```

Vérifier que la session s'ouvre correctement, puis taper `exit`.

- SSH ouvre un port entrant sur l'instance. SSM n'en ouvre aucun. Expliquez le mécanisme qui permet à SSM de fonctionner malgré l'absence de port entrant ouvert.
- Un administrateur se connecte via SSM et exécute des commandes sensibles. Où ces actions sont-elles enregistrées, et quel service AWS permet de les consulter a posteriori ?
- SSM améliore la traçabilité, mais introduit une nouvelle dépendance. Laquelle, et quel impact cela a-t-il en cas de panne de connectivité vers les endpoints AWS ?

13.3 Suppression du port SSH

Modifier le Security Group de l'instance EC2 pour supprimer la règle entrante sur le port 22.

Depuis la console AWS : EC2 > Security Groups > Sélectionner le SG de l'instance > Inbound rules > Supprimer la règle port 22

Vérifier qu'une connexion SSH directe est bien refusée :

```
ssh -i ~/.ssh/ma-cle.pem ec2-user@EC2_IP
# Attendu : Connection refused ou timeout
```

Vérifier que Session Manager fonctionne toujours :

```
aws ssm start-session --target INSTANCE_ID
```

13.4 Limite actuelle – Ansible et SSM

- Le playbook Ansible utilise encore SSH pour se connecter à l'instance alors que le port 22 vient d'être fermé. Quelle est la conséquence immédiate sur le pipeline CI/CD ?
- Dans une architecture sans SSH, deux alternatives à Ansible sont envisageables : le plugin de connexion SSM d'Ansible, ou le remplacement d'Ansible par AWS Systems Manager Run Command. Comparez ces deux approches sur les critères suivants : complexité de mise en oeuvre, dépendances, traçabilité.

- Plus largement, le modèle push d'Ansible et le modèle pull de certains outils (AWS CodeDeploy, Chef) répondent différemment à ce problème. Quelle architecture serait la plus adaptée si les instances EC2 sont dans un sous-réseau strictement privé sans accès entrant ?

Challenge final

L'objectif est d'améliorer l'architecture sur les points suivants.

Point 1 : Rotation automatique du secret

Configurer Secrets Manager pour effectuer une rotation automatique du mot de passe RDS tous les 30 jours.

Point 2 : Inventaire Ansible dynamique

Remplacer le fichier `inventory.ini` statique par un inventaire dynamique qui récupère automatiquement l'IP de l'instance EC2 via les tags AWS.

Point 3 : Notifications de déploiement

Ajouter une étape dans le pipeline GitHub Actions pour envoyer une notification (Slack ou email) en cas de succès ou d'échec du déploiement.

Questions de synthèse :

- La rotation automatique du secret en point 1 résout un problème de sécurité, mais crée un risque opérationnel. Lequel, et comment le playbook Ansible doit-il être adapté pour y faire face ?
- Si l'instance EC2 est remplacée par un conteneur ECS Fargate, identifiez les éléments du pipeline qui deviennent obsolètes et ceux qui restent pertinents.
- Ce pipeline déploie en continu sur la branche `main`. Un déploiement raté laisse l'application dans un état inconnu. Quelle stratégie de déploiement permet de garantir qu'une version précédente reste disponible pendant la mise à jour ?

Bonus – Déploiement Blue/Green simplifié

Mettre en place un déploiement sans interruption de service.

Modifier le playbook Ansible pour :

- déployer le nouveau jar dans un répertoire horodaté
- basculer un lien symbolique vers la nouvelle version
- redémarrer le service uniquement si le jar a changé

Fichier : `ansible/deploy_bluegreen.yml`

```
---
- hosts: web
```

```
  become: true

  vars:
    app_dir: /opt/demo
    releases_dir: /opt/demo/releases
    app_jar: demo.jar
    app_user: appuser
    timestamp: "{{ ansible_date_time.epoch }}"

  tasks:
    - name: Créer le répertoire de la release
      file:
        path: "{{ releases_dir }}/{{ timestamp }}"
        state: directory
        owner: "{{ app_user }}"
        mode: '0755'

    - name: Copier le jar dans le répertoire de la release
      copy:
        src: ../app/target/demo-1.0.jar
        dest: "{{ releases_dir }}/{{ timestamp }}/{{ app_jar }}"
        owner: "{{ app_user }}"
        mode: '0644'

    - name: Basculer le lien symbolique vers la nouvelle release
      file:
        src: "{{ releases_dir }}/{{ timestamp }}/{{ app_jar }}"
        dest: "{{ app_dir }}/current.jar"
        state: link
        owner: "{{ app_user }}"

    - name: Redémarrer le service
      systemd:
        name: demo
        state: restarted
```

- Ce déploiement réduit l'interruption de service à la durée du redémarrage JVM. Pourquoi un vrai Blue/Green sur AWS avec deux groupes cibles ALB va plus loin, et quel est le coût de cette approche ?
- Le lien symbolique est basculé avant le redémarrage du service. Si le redémarrage échoue, dans quel état se trouve l'application ? Comment modifier le playbook pour détecter cet échec et revenir automatiquement à la release précédente ?
- Sans politique de nettoyage, les répertoires de releases s'accumulent sur le disque. Rédigez la tâche Ansible qui conserve uniquement les 3 releases les plus récentes.