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
```

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.

Pourquoi utilise-t-on `-DskipTests` ici ?

Dans quel cas ne faudrait-il pas ignorer les tests ?

3. Déploiement manuel

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

Le jar est compilé sur le poste de travail. Avant de le copier sur l'EC2, il faut s'assurer que Java est disponible sur l'instance.



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

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

# Sur l'instance EC2 — vérifier Java
java -version
```

Si Java n'est pas installé :

```
# Sur l'instance EC2
```



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

# Vérifier l'installation
java -version

# Quitter la session
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 ?

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 :

- 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
```

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 prépare 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
```

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/
```

Ansible installe Java automatiquement via la tâche dédiée. Ce comportement est idempotent.

- Que se passe-t-il si on rejoue le playbook alors que Java est déjà installé ?
- Pourquoi l'idempotence est-elle importante dans un contexte d'automatisation ?

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 :

- 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 ?

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
```

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` ?
- Pourquoi ne jamais stocker ce secret dans le dépôt Git, même dans un fichier de variables Ansible ?

8. Pourquoi systemd plutôt que nohup

Comparez les deux approches :

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

- Que se passe-t-il avec `nohup` si l'instance redémarre ?
- Que se passe-t-il avec `nohup` si l'application plante ?
- Comment systemd résout-il ces deux problèmes ?
- Quelle option du service systemd gère le redémarrage automatique ?

9. Mise en place du pipeline CI/CD

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

Le runner GitHub Actions joue le même rôle que le poste de travail : il compile le jar avec Maven et Java, puis lance Ansible pour déployer sur l'EC2.

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 :

- Quels secrets GitHub doivent être configurés pour que le pipeline fonctionne ?
- Pourquoi le runner GitHub Actions ne peut-il pas se connecter à EC2 par défaut ?
- Quel fichier Ansible permet d'éviter l'erreur de vérification de la clé SSH hôte en CI ?
- Pourquoi la valeur `EC2_IP` dans `inventory.ini` doit-elle être remplacée par une vraie adresse ?

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
```

Questions sur la sécurité des secrets CI/CD :

- Pourquoi ne faut-il jamais mettre une clé AWS dans un fichier commité dans Git ?
- Quelle est la différence entre un secret GitHub et une variable GitHub Actions ?
- Comment s'assurer que les logs du pipeline n'affichent pas les valeurs des secrets ?

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
```

- Que doit retourner la commande `systemctl status demo` si le déploiement est réussi ?
- Quelle différence y a-t-il entre accéder à l'application via l'IP EC2 et via l'ALB ?
- Pourquoi vaut-il mieux exposer uniquement l'ALB et non l'IP de l'instance ?

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`.

Comparez les deux méthodes de connexion :

- Quelle différence y a-t-il entre une session SSH et une session SSM du point de vue réseau ?
- Où sont enregistrées les sessions SSM ? Comment cela améliore-t-il la traçabilité ?
- Quel service AWS peut être utilisé pour stocker les logs de sessions SSM ?

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

Réflexion sur la limite de l'architecture actuelle :

- Le playbook Ansible utilise encore SSH pour se connecter à l'instance. Pourquoi ?
- Ansible dispose d'un plugin de connexion SSM. Quels seraient les avantages de l'utiliser ?
- Pourquoi le passage complet à SSM pour Ansible est-il plus complexe qu'un simple changement de configuration ?
- Dans une architecture sans SSH, quelle alternative à Ansible pourrait être envisagée pour la configuration des instances ?

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 :

- Quels sont les points faibles restants dans cette architecture ?
- Si l'instance EC2 est remplacée par un conteneur ECS, quels éléments du pipeline doivent changer ?
- Comment garantir qu'un déploiement raté ne met pas l'application hors service ?

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
```

- Pourquoi utiliser un lien symbolique facilite-t-il un retour arrière (rollback) ?
- Comment modifier ce playbook pour conserver uniquement les 3 dernières releases ?
- Quelle est la différence entre ce déploiement et un vrai Blue/Green sur AWS ?

From:
http://slamwiki2.kobject.net/ - **SlamWiki 2.1**

Permanent link:
http://slamwiki2.kobject.net/eadl/bloc4/fm4/td4?rev=1782347781

Last update: **2026/06/25 02:36**