

TP 6 : Développer et déployer un Web service en mode DevOps

Prérequis pour le TP

- **Web services** : Savoir coder un Web service, avec une API REST, en utilisant Spring Boot
- **Tests et qualité** : Savoir écrire des tests unitaires sur une API REST
- **DevOps** : Savoir conteneuriser avec Docker, provisionner une ressource avec Terraform, et installer une application avec Ansible.
- **Reseau** : Notions d'IP, de NAT, de SSH, et du protocole HTTP.

Objectifs

Ce TP vise à mettre en oeuvre toutes les notions vues dans le module :

TDD → Docker → Terraform → Ansible.

Vous développerez une petite API REST en Java avec Spring Boot, la conteneuriserez puis la déploierez sur une **VM Linux locale** provisionnée avec Terraform et configurée par Ansible. **Ce TP peut être développé en binôme.**

Tâche	Temps conseillé
0. Posez plein de questions à vos enseignants	20 min
1. Développement du Web service en TDD	2h
2. Construction et test d'une image Docker	20 min
3. Provisionnement de la VM et redirection des ports	40 min
4. Configuration et déploiement avec Ansible	40 min
Total	4 h

Note. Le projet Spring est volontairement **sans authentification**.

1 Développer un Web service en TDD

1.1 Cahier des charges du service

Vous réalisez un service de suivi de *demandes de déploiement* d'applications. Le service ne lance pas de véritables déploiements : il enregistre les demandes et leur état, ce qui permettra de vérifier la chaîne DevOps de bout en bout.

Une demande possède : un identifiant généré par le serveur, un **service** (nom non vide), un **environnement** (DEV, STAGING ou PROD), un **status** et une date de création. Une nouvelle demande est toujours à l'état **PENDING**. Le cycle de vie autorisé est le suivant :

PENDING → RUNNING → SUCCEEDED ou FAILED

Une transition vers le même état ou depuis un état final est interdite. Les informations sont stockées en mémoire ; le choix des structures de données est libre.

Méthode et chemin	Comportement attendu	Succès
GET /api/health	Vérifier que l'application répond ; corps {"status":"UP"}	200
POST /api/deployments	Créer une demande initialisée à PENDING	201
GET /api/deployments	Lister toutes les demandes (liste vide au départ)	200
GET /api/deployments/{id}	Consulter une demande existante	200
PATCH /api/deployments/{id}/status	Changer son état selon les règles métier	200

Erreurs à traiter : 400 Bad Request pour un corps invalide (nom vide, environnement ou état inconnu), 404 Not Found si l'ID n'existe pas, 409 Conflict pour une transition d'état interdite. La création retourne également un en-tête Location: /api/deployments/{id}. Utilisez des réponses JSON ; n'exposez pas les détails d'exceptions Java.

Exemple de création :

```
POST /api/deployments
Content-Type: application/json

{"service":"catalogue", "environment":"STAGING"}
```

Exemple de changement d'état :

```
PATCH /api/deployments/{id}/status
Content-Type: application/json

{"status":"RUNNING"}
```

1.2 Préparer le projet et les tests

Créez un projet Spring Boot (Java 21, Maven), avec les dépendances Web MVC et tests. Organisez le code en packages : model, service, controller, dto, exception. La partie métier sera **indépendante des contrôleurs HTTP** pour faciliter les tests.

1.3 Développer la logique métier

Pour chaque fonctionnalité, suivez le cycle TDD comme vu en cours, et lors du TP2 : **failed** → **success** → **refactoring**. Autrement dit, écrivez d'abord un test qui échoue, implémentez le minimum, puis exécutez à nouveau tous les tests.

1. **Création.** Un test vérifie qu'une demande est créée avec un identifiant, l'environnement demandé et le statut PENDING. Un second vérifie que deux créations donnent deux identifiants distincts.
2. **Consultation.** Testez la recherche d'une demande, le cas absent et la liste initialement vide, puis non vide après deux créations.
3. **Validation.** Vérifiez qu'un nom de service vide n'est pas accepté. Choisissez où placer la validation, puis faites converger l'API vers une réponse HTTP 400.
4. **Transitions d'état.** Testez le passage PENDING → RUNNING, puis RUNNING → SUCCEEDED. Vérifiez enfin que PENDING → SUCCEEDED et SUCCEEDED → RUNNING sont refusés.

1.4 Exposer et tester l'API REST

Créez les DTO nécessaires, les contrôleurs et une gestion centralisée des exceptions. Vous pouvez utiliser `@ExceptionHandler`. Implémentez les routes de la table précédente.

Pour chaque cas suivant, écrivez **au moins un test de contrôleur ou un test d'intégration HTTP** :

1. un POST valide retourne 201, un Location et un corps contenant PENDING ;
2. la liste contient la ressource créée ; une recherche d'ID inexistant retourne 404 ;
3. un POST invalide retourne 400 ;
4. un changement d'état valide retourne 200 ; une transition interdite retourne 409.

1.5 Avant la suite ...

```
./mvn clean verify
```

Les tests doivent être au vert. Si vous n'avez pas terminé toutes les routes, **conservez au minimum /api/test**, POST et GET pour pouvoir poursuivre le déploiement.

2 Conteneuriser le Web service

2.1 Construire l'application et l'image

- 1) Produisez un JAR exécutable avec Maven et repérez son nom dans `target/`.

```
./mvn clean package
```

2) À la racine du projet, créez un fichier `Dockerfile` inspiré du suivant. Adaptez le chemin du JAR si nécessaire :

```
FROM eclipse-temurin:21-jre
WORKDIR /app
COPY target/*.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

3) Construisez l'image et vérifiez-la localement. Le port 8081 de l'hôte est utilisé pour ce test afin de réserver le port 18080 au redirectionnement de la VM (voir Section 3.2) :

```
./mvnw clean verify
docker build -t tp6-api:1.0 .
docker run -d --name tp6-local -p 8081:8080 tp6-api:1.0
curl -i http://127.0.0.1:8081/api/test
docker logs tp6-local
docker rm -f tp6-local
```

3 Déployer une VM avec Terraform

3.1 Provider libvirt

Nous utilisons **libvirt/QEMU**, piloté par le provider Terraform **dmacvicar/libvirt**. Le provider gère notamment des *domains* (VM), des volumes et des réseaux. Son schéma a **changé entre les versions 0.8.x et 0.9.x** : choisissez une version et suivez sa documentation ; ne mélangez pas les deux syntaxes. Documentation : <https://registry.terraform.io/providers/dmacvicar/libvirt/latest/docs>.

Note. Le cours fournit un exemple de VM plus complet (disque, image cloud et réseau). Vous pouvez partir de cet exemple.

1) Vous placerez les scripts et autres contenus nécessaires à cette étape dans un dossier `infra/`. Vous identifierez et adapterez les éléments suivants :

- provider **dmacvicar/libvirt** et URI de connexion `qemu:///session` propre à votre installation) ;
- ressource de VM `libvirt_domain` nommée `tp6-websevice`, avec 2 vCPU, environ 2 Gio de RAM, disque cloud-init et interface réseau **QEMU en mode utilisateur** ;
- système et image correspondant à l'architecture de l'hôte : `x86_64/amd64` ou `aarch64/arm64`. **Attention : Le type de machine peut changer selon le matériel** ;
- accès SSH par clé publique et lancement du service SSH dans la VM.

2) Exécutez les commandes habituelles de Terraform et vérifiez le nom et l'état de la VM.

```
terraform -chdir=infra init
terraform -chdir=infra validate
terraform -chdir=infra plan
terraform -chdir=infra apply
```

3.2 Rediriger les ports SSH et HTTP

En mode réseau utilisateur QEMU, l'adresse privée de la VM (souvent `10.0.2.15`) n'est pas directement joignable depuis l'hôte. Nous ajoutons deux redirections **sur la machine hôte** :

```
Hôte:127.0.0.1:2222 → VM:22 (SSH)
Hôte:127.0.0.1:18080 → VM:8080 → conteneur:8080
```

Vérifiez d'abord que la VM tourne. Dans les commandes suivantes, `LIBVIRT_URI` doit contenir la même URI que celle du provider (utilisez `terraform state list`). Les règles sont **dynamiques** : après un redémarrage de la VM, il peut être nécessaire de les recréer.

```
export LIBVIRT_URI="qemu:///session?socket=$HOME/.cache/libvirt/libvirt-sock"
virsh -c "$LIBVIRT_URI" list
virsh -c "$LIBVIRT_URI" qemu-monitor-command \
    tp6-websevice --hmp 'hostfwd_add tcp:127.0.0.1:2222-:22'
virsh -c "$LIBVIRT_URI" qemu-monitor-command \
    tp6-websevice --hmp 'hostfwd_add tcp:127.0.0.1:18080-:8080'
```

```
nc -vz 127.0.0.1 2222
ssh -p 2222 -i ~/.ssh/id_ed25519 ansible@127.0.0.1
```

Note. Cette partie du TP est la plus délicate, et est propre à la configuration matérielle de chacun. Le debug peut être long et difficile. Dans ce cas, vous pouvez vous tourner vers une IA générative pour fixer au plus vite vos bugs.

4 Configurer et déployer avec Ansible

4.1 Installer Docker sur la VM

Créez `inventory.ini`. L'inventaire doit employer le port SSH de l'hôte (2222), et non le port 22 de la VM ; dans le fichier INI, gardez tous les paramètres d'hôte sur une même ligne :

```
[api]
tp6-vm ansible_host=127.0.0.1 ansible_port=2222 ansible_user=ansible ansible_ssh_private_key_file=~/.ssh/
id_ed25519 ansible_python_interpreter=/usr/bin/python3
```

```
ansible -i inventory.ini api -m ping
```

Réutilisez et simplifiez votre playbook du TP 5 pour installer `docker.io` avec `apt`, puis activer/démarrer le service `docker`. Les tâches d'installation nécessitent `become: true`. Vérifiez ensuite :

```
ansible-playbook -i inventory.ini install-docker.yml
ansible -i inventory.ini api -b -m command -a 'docker --version'
```

4.2 Transférer et lancer l'application

Créez un playbook `deploy-api.yml` qui réalise les opérations suivantes **sur la VM** :

1. créer un répertoire `/opt/tp6-api` ;
2. copier le `Dockerfile` et le JAR construit sur l'hôte, avec une arborescence compatible avec la directive `COPY` ;
3. construire l'image Docker `tp6-api:1.0` sur la VM (ou y charger une image préalablement construite) ;
4. démarrer le conteneur `tp6-api`, en publiant le **port 8080 de la VM sur le port 8080 du conteneur** ;
5. vérifier que le conteneur est actif. Pour les tâches nécessitant Docker, exécutez les commandes avec les droits adaptés.

Note. Vous pouvez réutiliser `ansible.builtin.file`, `copy`, `command` et `service` vus dans les TP précédents.

```
ansible-playbook -i inventory.ini deploy-api.yml
ansible -i inventory.ini api -b -m command -a 'docker ps'
curl -i http://127.0.0.1:18080/api/test
```

Pour nettoyer à la fin : arrêtez l'application locale ou distante si nécessaire et exécutez `terraform -chdir=infra destroy` uniquement après sauvegarde de votre travail. **Attention** : cette commande détruit les ressources gérées par ce projet Terraform.

Pour aller plus loin

- Faire échouer un test métier avant toute modification, puis ajouter un statut `CANCELLED` pour annuler une demande en attente et sa couverture de test.
- Ajouter un système d'authentification avec JWT au web service (évidemment en mode TDD : on commence par écrire les tests).
- Rendre le playbook `deploy-api.yml` idempotent (seconde exécution sans modification inutile).
- Ajouter un pipeline CI déclenchant `mvn verify` et la construction de l'image Docker.