BUT 3
TP Dev Avancé #4
Migration vers Spring Boot – Sécurité JWT, Documentation, Tests & Industrialisation

Contexte pédagogique :

Après :

- **TP #1** : Servlet / JSP / JDBC
- **TP #2** : JPA / Hibernate / Architecture en couches
- **TP #3** : API REST JAX-RS / JAAS / Sécurité / Tests

Vous allez désormais migrer votre application **MasterAnnonce** vers une architecture moderne basée sur **Spring Boot**. L'objectif n'est pas "d'apprendre Spring par magie", mais de comprendre :

- ce que Spring automatisé pour vous,
- ce qu'il simplifie,
- ce qu'il masque,
- et comment garder la maîtrise de l'architecture.

Objectifs pédagogiques :

À l'issue de ce TP, l'étudiant doit être capable de :

- Migrer une API JAX-RS vers Spring Boot
- Utiliser Spring MVC pour exposer des endpoints REST
- Utiliser Spring Data JPA correctement
- Introspection + API Specification (Spring Data JPA Specifications) sur DAO
- Arrêter les conversions "à la main", comprendre la génération de code, et éviter les pièges JPA (lazy, cycles, etc.).
- Mettre en place une authentification JWT sécurisée
- Gérer les rôles et autorisations avec Spring Security
- Logging via Spring AOP => cross-cutting concerns, traçabilité, temps d'exécution, corrélation.
- Écrire des tests unitaires et d'intégration Spring
- Documenter une API avec OpenAPI
- Dockeriser une application backend
- Mettre en place une pipeline CI simple

Prérequis :

- TP AIR #3 validé

Contraintes générales :

- Interdiction d'utiliser des générateurs automatiques (copier-coller ChatGPT interdit aux étudiants □)
- Architecture en couches obligatoire :
  - Controller
  - Service
  - Repository
- DTO obligatoires (interdiction d'exposer les entités JPA directement)
  - Introduction à **MapStruct**
- Gestion propre des erreurs HTTP
- Code propre, structuré, documenté

Partie I – Migration vers Spring Boot

Exercice 1 – Migration du projet Spring Boot

1. Modifier votre fichier pom.xml pour basculer vers un projet Spring Boot Maven.
2. Dépendances minimales :
    a. Spring Boot Parent 3.5.5
    b. Spring Web
    c. Spring Data JPA
    d. Spring Security
    e. Spring AOP
    f. Validation
    g. Actuator
    h. PostgreSQL
    i. Spring Boot Test
3. Configurer :
    a. `application.yml`
    b. connexion PostgreSQL
    c. configuration JPA

 Livrable : application démarre sur `http://localhost:8080`

Exercice 2 – Migration des endpoints REST

1. Migrer les endpoints du TP#3 => Utiliser les annotations `@RestController`, `@RequestMapping`, `@Valid`, `@GetMapping`, `@PostMapping`, etc.

Verbe	URI	Description du résultat attendu
GET	• /api/annonces	• Liste paginée
GET	• /api/annonces/{id}	• Détail
POST	• /api/annonces	• Création
PUT	• /api/annonces/{id}	• Mise à jour
DELETE	• /api/annonces/{id}	• Suppression

2. Mise en place MapStruct (obligatoire)
    a. Ajouter MapStruct au `pom.xml` + config `maven-compiler-plugin` (annotation processing).
    b. Créer au minimum :
        i. `AnnonceMapper`
        ii. `UserMapper` (si DTO utilisateur)
        iii. `CategoryMapper`
    c. Mettre en place :
        i. `AnnonceDTO` (lecture)
    d. Interdictions :
        i. pas de `new DTO()` dans les controllers/services (sauf cas exceptionnel justifié)
        ii. pas de mapping “manuel” champ par champ

Points d'attention :

- Gestion des relations : `authorId`, `categoryId` dans DTO → reconstitution côté service (ou via `@Mapping` + méthodes utilitaires)
- Mise à jour partielle : utiliser un `@MappingTarget` (patch) et ignorer certains champs

Livrable

- Les controllers manipulent **uniquement des DTO**
- Les services manipulent entités + DTO, mais conversions via **mappers MapStruct**

Contraintes :

- **DTO obligatoires**, on étudiera l'opportunité d'utiliser Mapstruct pour remplacer le **Pattern Builder** pour faciliter le **mapping DTO to Entity & Entity to DTO**
- **Retour JSON uniquement**
- Codes HTTP corrects
- Pas de logique métier dans les ressources REST
- Bonus => implémenter un endpoint reposant sur le verbe HTTP PATCH
    - Proposer une description du résultat attendu

Exercice 3 – Spring Data JPA

1. Remplacer les repositories manuels par :

```
public interface AnnonceRepository extends JpaRepository<Annonce, Long>
```

2. Mettre en place :
    - pagination (`Pageable`)
    - recherche personnalisée (`@Query`)
    - tri
3. Recherche dynamique via Specifications + introspection
    - Implémenter une recherche multi-critères sur `/api/annonces` via query params :
        - `q` (mot-clé sur title/description)
        - `status`
        - `categoryId`
        - `authorId`
        - `fromDate`, `toDate`
    - Interdiction de faire une requête JPQL "géante" unique.
    - Utiliser :
        - `JpaSpecificationExecutor<Annonce>`
        - une classe `AnnonceSpecifications`
4. **Introspection demandée (exemples possibles) :** implémenter **au moins 1 usage** d'introspection parmi :
    - Validation côté API : vérifier que `sort` ne contient que des champs autorisés, en s'appuyant sur la réflexion (ex: `Annonce.class.getDeclaredFields()`)
    - Générer dynamiquement la liste des champs filtrables / triables exposés dans un endpoint `/api/meta/annonces`
    - Détecter automatiquement les champs `String` pour appliquer une recherche `LIKE` sur un ensemble de champs (title, description, address) de façon configurable
    - Utiliser le **Metamodel JPA** (`Annonce_`) pour une validation plus avancé)

Livrable

- Endpoint liste supporte filtres + pagination + tri

- Recherche implémentée via Specifications (preuve : code + tests)

Partie II - Sécurité moderne avec JWT

Exercice 4 – Authentification JWT

1. Créer un endpoint :
    a. POST `/api/auth/login`
    b. Entrée :

```
{
  "username": "user",
  "password": "password"
}
```

Sortie :

```
{
  "token": "JWT_TOKEN"
}
```

    c. Le token doit :
        i. être signé
        ii. contenir l'ID utilisateur
        iii. contenir les rôles
        iv. avoir une expiration

Exercice 5 – Configuration Spring Security

1. Configurer :
    a. `SecurityFilterChain`
    b. filtre JWT personnalisé
    c. désactivation session (stateless)
2. Exigences :
    a. endpoints publics : `/api/auth/login`
    b. endpoints protégés : `/api/**`
    c. rôles :
        i. ROLE_USER
        ii. ROLE_ADMIN

Exercice 6 – Règles métier sécurisées

1. Implémenter :
    - seul l'auteur peut modifier son annonce
    - seul un ADMIN peut archiver
    - annonce PUBLISHED non modifiable
2. Utiliser :
    - `@PreAuthorize`
    - contrôles métier dans le service
3. Mise en place de **Logging transversal avec Spring AOP**
    - Ajouter Spring AOP.
    - Créer un `@Aspect` qui log :
        - entrée/sortie des méthodes de `service.* (@Around)`

        - durée d’exécution
        - exceptions (avec type + message)
    - Ajouter un **Correlation ID** :
        - généré si absent (Filter)
        - propagé dans les logs via MDC

Contraintes

- Pas de logs “bruit” en controller (ou minimal)
- Le logging doit être **centralisé** via AOP

Pièges pédagogiques

- log d’objets JPA lazy → peut déclencher lazy loading involontaire
- attention à ne pas logger des secrets (password, token)

Livrable

- Logs exploitables (niveau prod) : temps, erreurs, et traçage cohérent

Partie III – Tests & qualité

Exercice 7 – Tests unitaires Service

1. Mockito
2. Tests des règles métier
3. Cas d’erreur

Exercice 8 – Tests d’intégration REST

1. Utiliser :
    a. `@SpringBootTest`
    b. `MockMvc`
2. Tester :
    a. login
    b. endpoint protégé sans token → 401
    c. token invalide → 401
    d. rôle insuffisant → 403
    e. CRUD complet

Exercice 9 – Base de test

1. Utiliser :
    a. Testcontainers (recommandé)
    b. H2 (si justifié)
2. Les tests doivent être isolés.

Partie IV – Documentation & Observabilité

Exercice 10 – Documentation OpenAPI

1. Intégrer SpringDoc OpenAPI.
2. Exigences :
    a. documentation visible via `/swagger-ui`
    b. exemples de requêtes
    c. documentation des erreurs

Exercice 11 – Actuator

1. Activer :
    a. `/actuator/health`
    b. `/actuator/info`
2. Configurer un health check PostgreSQL.

Partie V – Industrialisation

Exercice 12 – Dockerisation

1. Créer :
    a. Dockerfile multi-stage
    b. docker-compose (app + postgres)
2. Commande unique pour lancer le projet.

Exercice 13 – Mise en place d'une pipeline CI avec GitHub Actions

Objectif : Automatiser, à chaque push et pull request, les étapes suivantes :

- Compilation
- Exécution des tests (unitaires + intégration)
- Packaging
- Publication d'artefacts
- Build de l'image Docker

1. **Contraintes (imposées) :**
    a. GitHub Actions est obligatoire
    → fichier attendu : .github/workflows/ci.yml
    b. La pipeline doit s'exécuter automatiquement :
        i. sur push
        ii. sur pull_request
    c. Les tests ne doivent jamais être "skippés" dans la CI (-DskipTests interdit)
    d. Le projet doit être buildable sur une machine vierge (CI = source de vérité)
2. **Étapes minimales attendues (obligatoires) :**
    a. Étape A — Checkout
        i. Récupérer le code source avec actions/checkout
    b. Étape B — Setup Java + cache Maven
        i. Installer **Java 17 ou 21** (selon votre projet)
        ii. Activer le cache Maven (~/.m2) pour accélérer le pipeline
    c. Étape C — Build & tests
        i. Commande obligatoire : mvn -B clean verify

 verify est requis : il doit exécuter l'ensemble des tests, y compris les tests d'intégration.

    d. Étape D — Packaging et artefacts
        i. Le .jar généré doit être publié en **artifact GitHub Actions**
        ii. L'artifact doit être téléchargeable depuis l'onglet "Actions" du workflow
    e. Livrable attendu :
        i. Artifact nommé master-annonce-jar
3. Gestion de la base de données en CI (obligatoire), vous devez choisir **UNE** des stratégies suivantes (le choix doit être justifié dans le README).

Option 1 (recommandée) — Testcontainers

- Les tests démarrent PostgreSQL automatiquement

- Aucun service PostgreSQL n'est déclaré dans la CI
- Les tests doivent passer sans configuration DB spécifique

  ✅Avantages :
  - reproductible
  - fiable
  - conforme "industrialisable"

  ⚠ Point d'attention :
  - Les tests doivent être bien isolés, et le conteneur doit se démarrer correctement en CI.

Option 2 — PostgreSQL via service container GitHub Actions

Déclarer PostgreSQL comme **service** dans le workflow.

Attendus :

- configuration des variables d'environnement dans la CI
- attente de readiness (health check PostgreSQL ou script)

  ✅Avantages :
  - simple à comprendre

  ⚠ Risques :
  - instabilité si readiness mal gérée
  - dépendance forte au workflow

4. Règles de déclenchement (obligatoires), La pipeline doit tourner :
   a. A chaque push sur n'importe quelle branche
   b. A chaque pull_request vers main

Bonus :

- ajouter un job Docker uniquement sur main (ou sur tag)

5. Bonus (obligatoires)
   a. Bonus 1 — Matrice multi-Java
      i. Exécuter le workflow sur **Java 17 et Java 21**.
   b. Bonus 2 — Build Docker
      i. Construire l'image Docker avec docker build
      ii. Publier l'image en artifact (ou push dans GitHub Container Registry, bonus++)
   c. Bonus 3 — Qualité
      i. Activer JaCoCo maven plugin et publier le rapport en artifact
6. Livrables
   a. fichier github/workflows/ci.yml
   b. README :
      i. expliquer le choix DB (Testcontainers ou service PostgreSQL)
      ii. fournir une capture ou preuve du dernier workflow vert
      iii. préciser le nom de l'artifact produit
      iv. Artifact téléchargeable contenant le .jar
7. Critères de validation (checklist)

   ✅Workflow déclenché sur push + PR

   ✅Java installé + cache Maven

   ✅mvn clean verify exécuté

   ✅Tests exécutés (aucun skip)

   ✅Artifact .jar publié

   ✅Base de données fonctionnelle (Testcontainers ou service)

   ✅Pipeline reproductible et stable

Un court paragraphe expliquant chaque problème rencontré devra être fourni dans le README.

Livrables attendus

1. Projet Maven complet (lien du repo github)
2. README expliquant :
    - architecture
    - problèmes rencontrés
    - solutions apportées
3. Docker Compose
4. CI fonctionnelle
5. Documentation Swagger
6. Tests automatisés
7. Postman collection

Critères d'évaluation

1. Respect des consignes
2. Qualité du code
3. Bonne gestion des transactions
4. Compréhension des pièges
5. Clarté du code et du README
6. Le nombre de café que vous m'apporterez durant la journée
    - Seront accepté tout accompagnement pour le café

⭐ Bonus

- Refresh token
- Rate limiting
- Couverture > 80 %
- Pipeline Docker automatique

⭐ SuperBonus — Déploiement Kubernetes en local avec Minikube (K8s)

Objectif

Déployer l'application **MasterAnnonce Spring Boot** sur un cluster **Minikube** en local, avec une base **PostgreSQL**, en appliquant les bonnes pratiques minimales :

- configuration externalisée (ConfigMap / env vars),
- secrets,
- healthchecks (readiness/liveness),
- déploiement reproductible via manifests YAML,
- exposition via Ingress ou NodePort.

Contraintes imposées :

- L'environnement Kubernetes est **Minikube en local**.
- Le déploiement se fait via des manifests YAML stockés dans le projet : dossier **`/k8s`**.
- Le déploiement doit être **reproductible** :
  - `minikube start`
  - `kubectl apply -f k8s/`
  - accès API OK
- Les secrets ne doivent **pas** être committés en clair.

Pré-requis techniques (Minikube)

Vous devrez fournir dans le README les commandes d'installation/activation suivantes (selon OS) :

1. Démarrer Minikube :
   - `minikube start`
2. Activer l'Ingress Minikube (obligatoire si vous utilisez un Ingress) :
   - `minikube addons enable ingress`
3. Utiliser le Docker daemon de Minikube (pour éviter un registry) :
   - `eval $(minikube docker-env)` *(Linux/macOS)*
   - ou commande équivalente Windows/PowerShell

 But : construire l'image Docker **directement dans Minikube**.

Exigences attendues

1) Image Docker utilisable par Minikube (obligatoire)

- Build de l'image applicative dans le Docker de Minikube :
  - `eval $(minikube docker-env)`
  - `docker build -t masterannonce:1.0 .`
- Le `Deployment` doit référencer `masterannonce:1.0` (ou tag équivalent)

 Interdiction de dépendre d'un registry externe pour ce bonus.

2) Manifests Kubernetes (obligatoires)

Créer un dossier **`/k8s`** contenant au minimum :

1. **PostgreSQL**
   a. `postgres-deployment.yaml`
   b. `postgres-service.yaml`
   c. *(optionnel)* PVC : `postgres-pvc.yaml`
2. **Application**
   a. `app-deployment.yaml`
   b. `app-service.yaml`
3. **Configuration & secrets**
   a. `app-configmap.yaml`
   b. `postgres-secret.yaml` (ou secret unique)
4. **Exposition**
   a. `ingress.yaml`

3) Configuration Spring via env vars (obligatoire)

L'application doit lire sa configuration depuis Kubernetes :

- `SPRING_DATASOURCE_URL`
- `SPRING_DATASOURCE_USERNAME`
- `SPRING_DATASOURCE_PASSWORD`
- `SPRING_PROFILES_ACTIVE`
- etc.

 Interdiction de coder les paramètres en dur dans le jar.

4) Health checks (obligatoire)

Dans `app-deployment.yaml`, configurer :

- `readinessProbe` → `/actuator/health/readiness`
- `livenessProbe` → `/actuator/health/liveness`

5) Scalabilité (obligatoire)

- L'application doit être déployée avec :
  - `replicas: 2`

Validation attendue (preuves obligatoires dans le README)

A) Commandes d'exécution

1. `minikube start`
2. `minikube addons enable ingress` *(si Ingress utilisé)*
3. `eval $(minikube docker-env)`
4. `docker build -t masterannonce:1.0 .`
5. `kubectl apply -f k8s/`
6. `kubectl get pods`
7. `kubectl get svc`
8. `kubectl get ingress` *(si applicable)*

B) Preuve que l’API fonctionne

- Un appel réel sur l’API :
  - `POST /api/auth/login`
  - `GET /api/annonces`
- Avec URL Minikube :
  - soit via Ingress + `minikube ip`
  - soit via `minikube service <service-name> --url`

C) Preuve des probes

- Les pods sont en état `READY 1/1`
- Une capture de `kubectl describe pod ...` (ou extrait lisible)

□ Points d’attention (Minikube)

- `ImagePullBackOff` (image non présente dans Minikube) → oubli `eval $(minikube docker-env)`
- Postgres “pas prêt” → readiness à ajuster / init time
- Mauvais mapping env vars Spring → crashloop
- Ingress non activé → route inaccessible
- Secrets committés en clair → pénalité de la mort qui tue !!!!!!