BUT 3
TP Dev Avancé #3
Transformation de MasterAnnonce en Backend API sécurisé, testé et industrialisable

Contexte pédagogique :

À l'issue de ce TP, l'étudiant doit être capable de produire un **backend Java professionnel**, exposé sous forme d'API REST, sécurisé, testé et prêt à être intégré dans une architecture réelle.

Objectifs pédagogiques :

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

- Concevoir une **API REST propre et cohérente**
- Appliquer les principes REST (ressources, verbes, stateless)
- Sécuriser une application backend
- Gérer des règles métier complexes et des transactions
- Tester chaque couche de l'application
- Comprendre les compromis architecturaux
- Diagnostiquer et corriger des problèmes réels

Prérequis :

- TP AIR #2 validé

Contraintes générales :

- **Interdiction d'utiliser Spring (Spring Boot, Spring MVC, Spring Security, etc.)**
- Utilisation exclusive de :
    - Java EE / Jakarta EE
    - JAX-RS pour l'API REST
    - JPA / Hibernate pour la persistance
- L'application doit être **totalement utilisable sans JSP**
- Communication **JSON uniquement**
- Architecture en couches obligatoire

Architecture cible :

```
Client (Postman / Front JS)
        ↓ HTTP / JSON
API REST (JAX-RS)
        ↓
Service (transactions + règles métier)
        ↓
Repository (JPA)
        ↓
PostgreSQL
```

Partie I – Exposition REST (fondation)

Exercice 1 – Mise en place de JAX-RS

1. Configuration JAX-RS (Jersey ou RESTEasy)
2. Création du point d'entrée `/api`
3. Test avec un endpoint simple `/api/helloWorld`
4. Passage de paramètre avec un endpoint simple `/api/params`
    a. `QueryParams`
    b. `PathParams`

📌 **Livrable** : Justifier le choix de votre configuration

Exercice 2 – API REST Annonce

Implémenter les endpoints suivants :

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

📌 Contraintes :

- **DTO obligatoires**, on étudiera l'opportunité d'implémenter le **Pattern Builder** pour faciliter le **mapping DTO to Entity & Entity to DTO**
- 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

Partie II - Validation, erreurs et robustesse

Exercice 3 – Validation API

1. Bean Validation sur les DTO
2. Gestion centralisée des erreurs
3. Réponses JSON normalisées
   Exemple :

```
{
  "error": "VALIDATION_ERROR",
  "messages": ["title is required"]
}
```

Exercice 4 – Gestion des erreurs REST

- **400 : erreur client => requête invalid**
- **404 : ressource inexistante**
- **409 : conflit métier**
- **500 : erreur interne**
  💥 **Douleur volontaire :**
  - **exception non interceptée = API inutilisable**

Partie III – Sécurité

Exercice 5 – Authentification stateless

1. Endpoint `/api/login`
2. Génération d'un token simple
3. Stockage temporaire (mémoire)

Exercice 6 – Filtre de sécurité

1. Vérification du token sur endpoints protégés
2. Refus accès non authentifié

Exercice 7 – Règles métier avancées

1. Seul l'auteur peut modifier/supprimer une annonce
2. Une annonce PUBLISHED ne peut plus être modifiée
3. Archivage obligatoire avant suppression
4. Gestion concurrence via `@Version`

Partie IV – Tests & qualité logicielle

Exercice 8 – Tests Repository (intégration)

4. Pagination
5. Mettre en place une BDD de type **H2** en mode "In-Memory"
6. Implémenter le chargement d'un jeu de donnée avant l'exécution des scénarios de test

Exercice 9 – Tests API REST

3. Implémenter des Tests unitaires
4. Implémenter des Tests d'intégration REST
5. Vérification payloads + HTTP codes (cas d'erreurs aussi)

Exercice 10 – Industrialisation

6. Implémenter la possibilité de lancer séparément les tests unitaires & intégrations
    - Expliquer dans quelle mesure il est intéressant de séparer leurs exécutions
7. Logging structuré
8. Tests de charge simples
9. Documentation API (OpenAPI)
10. README

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

Livrables attendus

1. Projet Maven complet
2. README expliquant :
    - architecture
    - problèmes rencontrés
    - solutions apportées
3. Scripts SQL éventuels
4. Tests automatisés
5. 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 : JAAS sert à établir l'identité (Subject/Principals), et l'API REST utilise un token envoyé à chaque requête.**

Exercice 5 – Authentification stateless avec JAAS (version Bonus)

Objectif : Mettre en place une authentification sans session HTTP, basée sur :

- une phase de login (username/password)
- un **token** transmis à chaque requête
- une **reconstruction de l'identité via JAAS** (Subject/Principals) pour chaque appel protégé

5.1 – Mise en place JAAS (configuration)

1) Créer un fichier de configuration JAAS

Créer `jaas.conf` (placé dans `src/main/resources` ou à côté du serveur selon votre setup).

Deux "domains" (2 LoginModules) :

- `MasterAnnonceLogin` : authentification username/password
- `MasterAnnonceToken` : authentification par token

Exemple (indicatif) :

```
MasterAnnonceLogin {
  com.masterannonce.security.login.DbLoginModule required;
};

MasterAnnonceToken {
  com.masterannonce.security.login.TokenLoginModule required;
};
```

2) Activer la config JAAS au lancement

Ajouter l'option JVM au démarrage de Tomcat :

- `-Djava.security.auth.login.config=/chemin/vers/jaas.conf`

📌 **Livrable :** la config JAAS est chargée (preuve : logs au démarrage ou test unitaire).

5.2 – Authentification Login (username/password) via JAAS

3) Implémenter `DbLoginModule`

Créer une classe `DbLoginModule` (qui implémente `javax.security.auth.spi.LoginModule` ou `jakarta…` selon stack JDK) :

- Récupère les credentials via `CallbackHandler` (`NameCallback`, `PasswordCallback`)
- Vérifie username/password en base (via `UserRepository`)
- En cas de succès :
    - ajoute au `Subject` :
        - un `UserPrincipal(username, userId)`
        - un ou plusieurs `RolePrincipal` (ex: `ROLE_USER`, `ROLE_ADMIN` si vous gérez des rôles)

📌 **Contraintes**

- Aucun stockage en session HTTP
- Les rôles sont matérialisés sous forme de `Principal` dans le `Subject`

5.3 – Endpoint REST `/api/login`

4) Créer le endpoint login

`POST /api/login`

Exemple d'entrée JSON

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

Traitement

- Créer un `LoginContext("MasterAnnonceLogin", callbackHandler)`
- `loginContext.login()`
- Récupérer le `Subject` authentifié
- Générer un token (UUID ou opaque)

Sortie JSON

```
{ "token": "xxxxx-xxxxx-xxxxx", "expiresIn": 3600 }
```

📌 **Note pédagogique (important)**
Même si JAAS authentifie l'utilisateur, **le token est nécessaire** pour respecter le côté "stateless" côté client : le client renvoie le token **à chaque requête**.

5.4 – Authentification par token via JAAS sur chaque requête

5) Implémenter `TokenLoginModule`

Créer `TokenLoginModule` qui :

- reçoit un token via `CallbackHandler` (callback custom `TokenCallback` ou `NameCallback` réutilisé)
- valide le token (stockage mémoire `token -> userId` + expiration)
- reconstitue l'identité :
    - charge `User` (ou a minima username/userId)
    - peuple le `Subject` avec `UserPrincipal` + `RolePrincipal`

📌 **Livrable :** Un token valide produit un `Subject` valide contenant l'identité et les rôles.

5.5 – Protection des endpoints (JAX-RS Filter + JAAS)

6) Filtre JAX-RS obligatoire

Mettre en place un `ContainerRequestFilter` (ou filtre Servlet si vous préférez) sur les endpoints protégés :

- Lit le header : `Authorization: Bearer <token>`
- Si absent → HTTP 401
- Sinon :
    - lance `LoginContext("MasterAnnonceToken", callbackHandlerWithToken)`
    - si succès : récupère le `Subject`
    - attache l'identité au contexte de requête :
        - soit via un `SecurityContext` JAX-RS custom
        - soit via `request.setAttribute("subject", subject)` si filtre Servlet

📌 **Résultat attendu :** Chaque requête protégée reconstruit l'utilisateur courant via JAAS **sans session**.

5.6 – Exploitation de l'identité dans le code métier

Dans la couche Service :

- récupérer `userId` depuis le `Subject` courant
- appliquer les règles métier :
    - seul l'auteur modifie/supprime
    - actions publish/archive limitées à certains rôles si vous l'imposez (option)

Livrables attendus (Exercice 5)

1. `jaas.conf`
2. `DbLoginModule` + `TokenLoginModule`
3. `UserPrincipal` / `RolePrincipal`
4. `POST /api/login`
5. Filtre REST de sécurité + gestion des codes HTTP 401/403
6. README : expliquer le flow d'auth (login → token → JAAS par requête)

Bonus du Bonus Tests (liés à JAAS)

- Test unitaire `DbLoginModule` (mocks repository + callbacks)
- Test unitaire `TokenLoginModule` (token valide/invalide/expiré)
- Test intégration REST :
    - login → récupérer token
    - appeler endpoint protégé sans token → 401
    - appeler endpoint protégé avec token → 200