

# Livrable du jour

## Couche de persistance abstraite (DAO/Repository + simulation de stockage)

### Objectif

Prouver que ton **métier fonctionne sans connaître le stockage**.

Autrement dit : ton code métier (domain / application) doit pouvoir tourner avec une persistance simulée (en mémoire), **sans SQL, sans ORM, sans base réelle**.

### Ce qui est attendu (résultat final)

À la fin du livrable, on doit retrouver dans ton projet :

1. **Une ou plusieurs interfaces** (DAO et/ou Repository) représentant le contrat d'accès aux données.
2. **Une implémentation simulée** de ce contrat (InMemory / Fake / Stub).
3. Une organisation claire dans l'arborescence (cohérente avec tes modules et tes couches).
4. **Un ou deux exemples de flux métier** qui utilisent uniquement l'abstraction (et pas le stockage).

Critère clé : on doit pouvoir remplacer `InMemoryTicketRepository` par `TicketRepositoryPostgres` plus tard **sans changer le métier**.

## Prérequis

### 1) Choisis 1 domaine métier “pilote”

Tu ne dois pas tout implémenter. Prends 1 domaine simple pour démontrer le découplage.

Exemples :

- Tickets (souvent le plus simple)

- Utilisateurs
- Facturation (si tu as déjà un modèle clair)

Recommandation : commence par **Tickets** si tu veux aller vite et faire propre.

## 2) Identifie 2 à 3 opérations métier

Choisis des opérations qui montrent une règle métier.

Exemples (Tickets) :

- créer un ticket
- clôturer un ticket (avec une règle : résolution obligatoire)
- lister les tickets ouverts

## Étape 1 – Définir le modèle métier minimal (domain)

Tu dois avoir un modèle métier minimal qui n'a pas connaissance du stockage.

### Exemple simple (Tickets)

- `Ticket` (entité)
- `TicketStatus` (enum)

Règle métier :

- un ticket clôturé doit avoir une résolution

**Important :**

- pas d'annotation ORM
- pas de SQL
- pas de dépendance framework dans le domaine

## Étape 2 – Définir le contrat : Repository (abstraction)

### Objectif

Ton service métier doit dépendre d'une **interface**.

Tu crées une interface (ex. `TicketRepository`) qui exprime les besoins du métier.

## Exemple d'interface attendue (niveau simple)

- `save(ticket)`
- `findById(id)`
- `findOpenTickets()`

### À respecter :

- le repository retourne des objets métier (Ticket), pas des objets ORM
- l'interface vit côté domaine (ou core stable), pas côté infrastructure

## Étape 3 – Implémenter une simulation : InMemory / Fake

### Objectif

Tu fournis une implémentation concrète du repository, mais **sans base**.

#### Option A (recommandée) : InMemoryRepository

- stockage interne : Map / List
- permet de sauvegarder et retrouver des objets

#### Option B : FakeRepository

- données pré-remplies
- utile si tu veux simuler rapidement des cas

#### Option C : Stub

- retours déterministes
- utile pour prouver un point, mais souvent trop pauvre

**Recommandation** : fais un **InMemoryRepository**, c'est le plus propre.

## Étape 4 – Écrire un service métier qui utilise uniquement le contrat

### Objectif

Ton service métier (ex. `TicketService`) ne doit connaître que :

- les règles métier
- l'interface `TicketRepository`

Il ne doit pas connaître :

- SQL
- ORM
- InMemory

### Exemple de règle à implémenter

“Un ticket ne peut pas être clôturé sans résolution.”

Le service :

1. récupère le ticket via `repository.findById()`
2. valide la règle métier
3. met à jour l'état du ticket
4. sauvegarde via `repository.save()`

## Étape 5 – Démontrer que ça fonctionne (preuve)

Tu dois fournir une preuve simple, au choix :

### Option : Test unitaire

- un test qui vérifie que la règle métier est appliquée
- sans base
- en utilisant `InMemoryTicketRepository`
- 

**Recommandation** : un test unitaire simple (même 2 tests) est idéal.

Cas à montrer :

- clôture sans résolution → erreur
- clôture avec résolution → OK

## Organisation attendue (arborescence)

Tu dois ranger ton code de manière cohérente.

### Exemple (monolithe modulaire + couches)

- `module-tickets/`
  - `domain/`
    - `Ticket`
    - `TicketStatus`
    - `TicketRepository` (interface)
  - `application/`
    - `TicketService`
  - `infrastructure/`
    - `InMemoryTicketRepository`

Tu peux aussi avoir :

- `core/` pour les éléments partagés

## Exemples simples (ce que tu peux livrer)

### Exemple 1 – TicketRepository + InMemory

- Interface : `TicketRepository`
- Implémentation : `InMemoryTicketRepository`
- Service : `TicketService.closeTicket(id, resolution)`
- Test : `closeTicket_requiresResolution()`

### Exemple 2 – UserRepository + InMemory

- Interface : `UserRepository`
- Implémentation : `InMemoryUserRepository`
- Service : `UserService.assignRole(userId, role)`
- Test : `assignRole_requiresAdmin()`

Tu choisis un seul exemple, mais il doit être propre.

## Points de vigilance (les erreurs qui font perdre des points)

1. **Le métier dépend de l'implémentation** (ex. `new InMemory...` partout dans le domaine)
2. **Entités métier annotées ORM**
3. Repository placé dans l'infrastructure (mauvais sens des dépendances)
4. Repository qui retourne des objets "DB" au lieu d'objets métier
5. Simulation trop pauvre : pas de stockage interne, pas de comportement
6. Logique métier dispersée (un peu dans l'entité, un peu dans l'infra, un peu dans le service)

## Livrables à rendre

1. **Arborescence du projet** (structure claire)
2. **Interface(s) Repository/DAO**
3. **Implémentation simulée** (InMemory/Fake)
4. **Service métier** utilisant l'abstraction
5. **Preuve de fonctionnement** (test ou runner)

## Définition du "succès" (critère simple)

Si je peux remplacer ton `InMemoryTicketRepository` par une future implémentation SQL **sans modifier ton service métier**, alors ton livrable est réussi.