Spring Boot Starter Data JPA

Contexte et objectif

- **Accéder** à une base de données relationnelle simplement
- **Simplifier** la gestion des entités et des requêtes
- **Accélérer** le développement grâce à des abstractions haut niveau

Spring Data JPA automatise 80% du travail classique d'accès aux bases de données.

JPA vs Hibernate : qui fait quoi ?

- **JPA** : spécification Java pour la gestion des données relationnelles (norme)
- **Hibernate** : implémentation de JPA la plus utilisée
- On programme contre **JPA**, Hibernate exécute réellement

Exemple :

```
@PersistenceContext
private EntityManager em;
```

`EntityManager` est défini par JPA, mais c'est Hibernate qui fournit son comportement.

Pourquoi séparer JPA et Hibernate ?

- **Portabilité** : votre code reste compatible avec d'autres implémentations (EclipseLink, OpenJPA…)
- **Standardisation** : une API unique pour différents moteurs
- **Optimisations spécifiques** : Hibernate propose des extensions puissantes au-delà de JPA standard

Exemples d'extensions Hibernate : - `@BatchSize`, `@DynamicInsert`, `@DynamicUpdate`, `@CreationTimestamp`…

Comment Hibernate étend JPA ?

Hibernate respecte l'API JPA… mais propose : - Des annotations supplémentaires - Des stratégies de cache avancées - Une gestion fine du lazy-loading - Des outils comme Hibernate Validator (validation des entités)

Attention : Utiliser des extensions Hibernate peut **casser** la portabilité si vous changez d'implémentation.

Il est conseillé de rester aussi proche que possible de JPA, et de n'utiliser les extensions Hibernate que lorsque c'est vraiment nécessaire.

Ajouter Spring Data JPA

Dans votre `pom.xml` :

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
  <groupId>org.postgresql</groupId> <!-- ou autre SGBD -->
  <artifactId>postgresql</artifactId>
</dependency>
```

- Starter complet pour JPA
- Pilote JDBC spécifique pour votre base de données

Configurer la base de données

Exemple `application.yml` :

```
spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/mydb
    username: user
    password: pass
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
```

- `ddl-auto` contrôle la génération du schéma (`update`, `create`, `validate`, `none`)
- `show-sql` affiche les requêtes générées en console

Définir une entité JPA

Exemple simple :

```
@Entity
public class Book {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String title;

    private String author;

    private int pageCount;

    private LocalDate publicationDate;

    // getters et setters
}
```

- `@Entity` signale une classe persistante
- `@Id` et `@GeneratedValue` pour la clé primaire

Créer un Repository

Un simple `interface` suffit :

```
public interface BookRepository extends JpaRepository<Book, Long> {
}
```

- Hérite de `JpaRepository<Entity, IdType>`
- CRUD complet automatiquement disponible

Utiliser le Repository

Dans un `@Service` ou directement en `@RestController` :

```
@Service
public class BookService {

    private final BookRepository bookRepository;

    public BookService(BookRepository bookRepository) {
        this.bookRepository = bookRepository;
    }

    public Book createBook(Book book) {
        return bookRepository.save(book);
    }
```

```
    public List<Book> getAllBooks() {
        return bookRepository.findAll();
    }

    public Optional<Book> getBookById(Long id) {
        return bookRepository.findById(id);
    }
}
```

Personnaliser les requêtes

Créer des méthodes avec des **noms intelligents** :

```
List<Book> findByAuthor(String author);

List<Book> findByTitleContainingIgnoreCase(String keyword);
```

- Pas besoin d'implémentation manuelle
- Traduction automatique en requêtes SQL

Utiliser les requêtes JPQL ou SQL natif

```
@Query("SELECT b FROM Book b WHERE b.pageCount > :minPages")
List<Book> findBooksLongerThan(@Param("minPages") int minPages);

@Query(value = "SELECT * FROM book WHERE page_count > :minPages", nativeQuery = true)
List<Book> findBooksLongerThanNative(@Param("minPages") int minPages);
```

- `@Query` pour du JPQL ou SQL natif
- Flexibilité maximale si besoin

Pagination avec Spring Data JPA

Exposer des données paginées dans un service :

```
public Page<Book> getBooksPaginated(int page, int size) {
    Pageable pageable = PageRequest.of(page, size);
    return bookRepository.findAll(pageable);
}
```

- `Pageable` gère page, taille, tri
- Retourne un `Page<Book>` contenant les résultats + métadonnées

Exemple d'appel API :

```
GET /books?page=0&size=10
```

Tri dynamique avec Spring Data JPA

Ajouter du tri en plus de la pagination :

```
public Page<Book> getBooksSortedByTitle(int page, int size) {
    Pageable pageable = PageRequest.of(page, size, Sort.by("title").ascending());
    return bookRepository.findAll(pageable);
}
```

- `Sort` permet de trier par un ou plusieurs champs
- Ascendant (`ascending()`) ou descendant (`descending()`)

Exemple d'appel API :

```
GET /books?page=0&size=5&sort=title,asc
```

Ces fonctionnalités sont essentielles pour les API modernes exposant des grandes quantités de données. Pagination et tri permettent des performances optimales et une meilleure expérience utilisateur.

Best practices avec Spring Data JPA

- **DTO vs Entités** : exposez des DTO aux API REST, pas vos entités
- **Transactions** : par défaut, les méthodes de `JpaRepository` sont transactionnelles
- **Lazy Loading** : attention aux accès hors transaction
- **Pagination** : utilisez `Pageable` pour éviter des retours trop volumineux

Résumé

- Spring Data JPA **automatise** le CRUD et simplifie l'accès à la base
- **Moins de code** = plus de productivité
- **Extensible** par des requêtes complexes si nécessaire

Slide final pour montrer pourquoi utiliser Spring Data JPA est un game changer dans un projet Spring moderne.