JJUGナイトセミナー「Jakarta EEの始め方と新機能」

古き良きJPAから、モダンなJakarta Dataへ
～生産性を飛躍させる新たな選択肢～

2025/10/27　　20:05-20:55

NEC　景井教天

自己紹介

NEC
テクノロジーサービスソフトウェア統括部
景井 教天

2021年 NEC 入社

Javaアプリケーションサーバ(WebOTX Application Server) 開発・保守
Springサポート 企画・保守

2025年 技術書出版

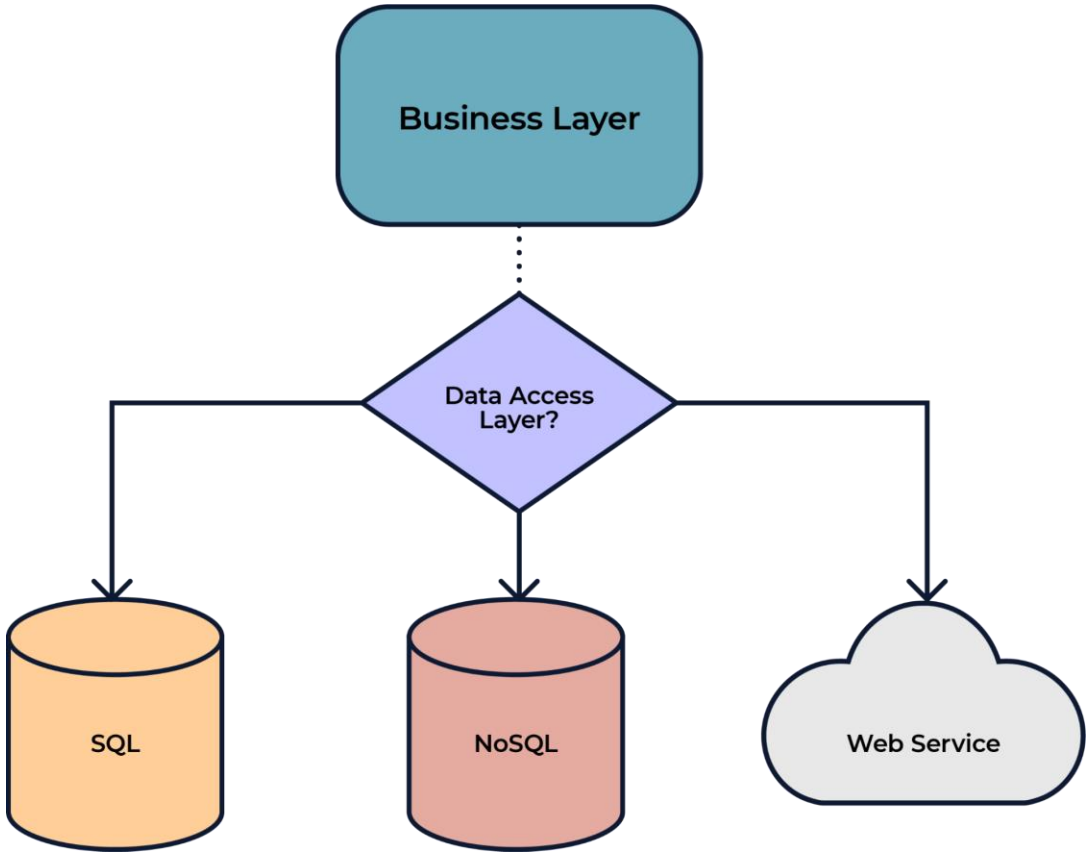
改訂3版　パーフェクトJava
https://gihyo.jp/book/2025/978-4-297-14680-1

Jakarta Data とは?

Jakarta EE 11

Jakarta Data とは

現代開発において、扱うデータの多様化により管理が複雑化したデータアクセスのボイラーコードが排除され、開発者がビジネスロジックに集中しやすい仕様が求められた

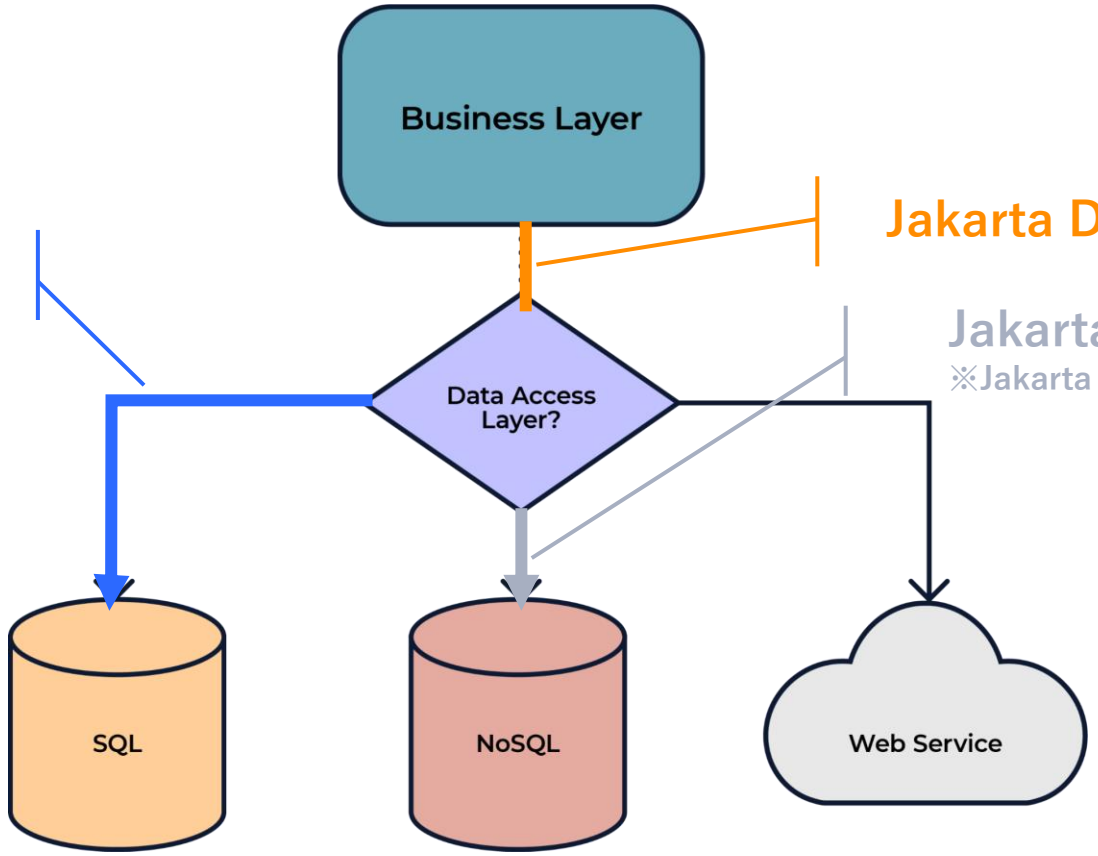


※ボイラーコードとは、ほぼ変わらないまま繰り返し使われる定型的なコードのことを指します

Jakarta Data とは

現代開発において、扱うデータの多様化により管理が複雑化したデータアクセスのボイラーコードが排除され、
開発者がビジネスロジックに集中しやすい仕様が求められた

Jakarta Persistence 3.2



Jakarta Data 1.0

Jakarta NoSQL 1.0
※Jakarta EE 11 のエコシステムとしては提供されない

ボイラーコードをどのくらい削減できる?

データアクセス処理 - SQL

Jakarta Persistence による実装例

```
public class ProductRepository {
    @PersistenceContext                                    EntityManagerを注入
    private EntityManager em;

    public void save(Product product) {                    CRUD操作用
        em.persist(product);                               リポジトリメソッドを実装
    }
    public Product findById(String id) {
        return em.find(Product.class, id);
    }
    public List<Product> findAll() {
        return em.createQuery("SELECT p FROM Product p", Product.class)
                 .getResultList();
    }
    public Product update(Product product) {
        return em.merge(product);
    }
    public void delete(Product product) {
        Product managed = em.merge(product);
        em.remove(managed);
    }
    …
```

```
    …

    public List<Product> findByCategory(String category) {
        return em.createQuery(    JPQLを定義⇒パラメータ置換⇒クエリ実行
            "SELECT p FROM Product p WHERE p.category = :category",
            Product.class)
            .setParameter("category", category)
            .getResultList();
    }

    public List<Product> findByPriceBetween(int min, int max) {
        return em.createQuery(
            "SELECT p FROM Product p WHERE p.price BETWEEN :min AND :max",
            Product.class)
            .setParameter("min", min)
            .setParameter("max", max)
            .getResultList();
    }
}
```

データアクセス処理 - SQL

Jakarta Data による実装例

```java
@Repository
public interface ProductRepository extends CrudRepository<Product, String> {

    @Find
    List<Product> findByCategory(String category);

    @Query("where price between :min and :max")
    List<Product> findByPriceBetween(int min, int max);
}
```

CRUD操作用
リポジトリメソッドを定義

JDQLを定義

データアクセス処理 - NoSQL

Jakarta NoSQL による実装例

```java
@ApplicationScoped
public class ProductRepository {
    @Inject                                          Templateを注入
    private Template template;

    public Product save(Product product) {                    CRUD操作用
        return template.insert(product);          リポジトリメソッドを実装
    }

    public Optional<Product> findById(String id) {
        return template.find(Product.class, id);
    }

    public void deleteById(String id) {
        DocumentDeleteQuery deleteQuery = delete()
            .from("Product")
            .where("_id")
            .eq(id)
            .build();
        template.delete(deleteQuery);
    }
    …
```

```java
    …

    public List<Product> findByCategory(String category) {
        DocumentQuery query = select()        クエリビルダー定義⇒クエリ実行
            .from("Product")
            .where("category")
            .eq(category)
            .build();
        return template.select(query);
    }

    public List<Product> findByPriceBetween(int min, int max) {
        DocumentQuery query = select()
            .from("Product")
            .where("price")
            .gte(min)
            .and("price")
            .lte(max)
            .build();
        return template.select(query);
    }
}
```

データアクセス処理 - NoSQL

Jakarta Data による実装例

CRUD操作用
リポジトリメソッドを定義

JDQLを定義

```java
@Repository
public interface ProductRepository extends CrudRepository<Product, String> {

    @Find
    List<Product> findByCategory(String category);

    @Query("where price between :min and :max")
    List<Product> findByPriceBetween(int min, int max);
}
```

どのような機能で構成されてる?

リポジトリパターン

主なメリットを享受

関心の分離

ビジネスロジックとDB処理を分離

テスト容易性

データベースに依存せずに
ビジネスロジックの単体テストを容易に実装

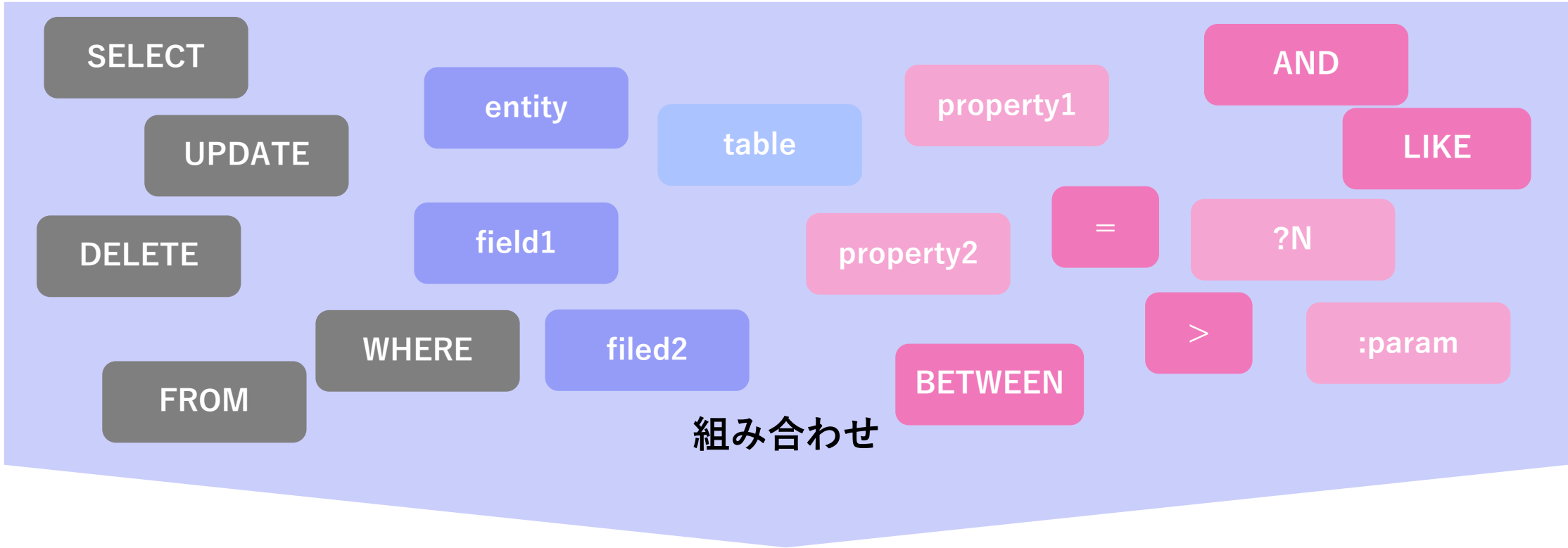
柔軟な切り替え

データソースの変更に対して
アプリケーションコードを軽微な修正で対応

保守性と拡張性

一貫したデータアクセス方法により、
規模が大きいほど、保守性や拡張性が向上

カスタムクエリ



どのように使用できる?

リポジトリ

基本構文

```
@Repository
修飾子 interface リポジトリ名 extends 基底インタフェース<T, K> {
}
```

@Repository
本アノテーションをインタフェースに付与することで、リポジトリパターンを簡単に導入できる

基底インタフェース
標準で3種類のインタフェースが提供されている　　*後述スライドを参照

T: 対象データのエンティティ
K: 対象データのID型

リポジトリ

■ 基本構文

```
@Repository
修飾子 interface リポジトリ名 extends 基底インタフェース<T, K> {
}
```

@Repository
本アノテーションをインタフェースに付与することで、リポジトリパターンを簡単に導入できる

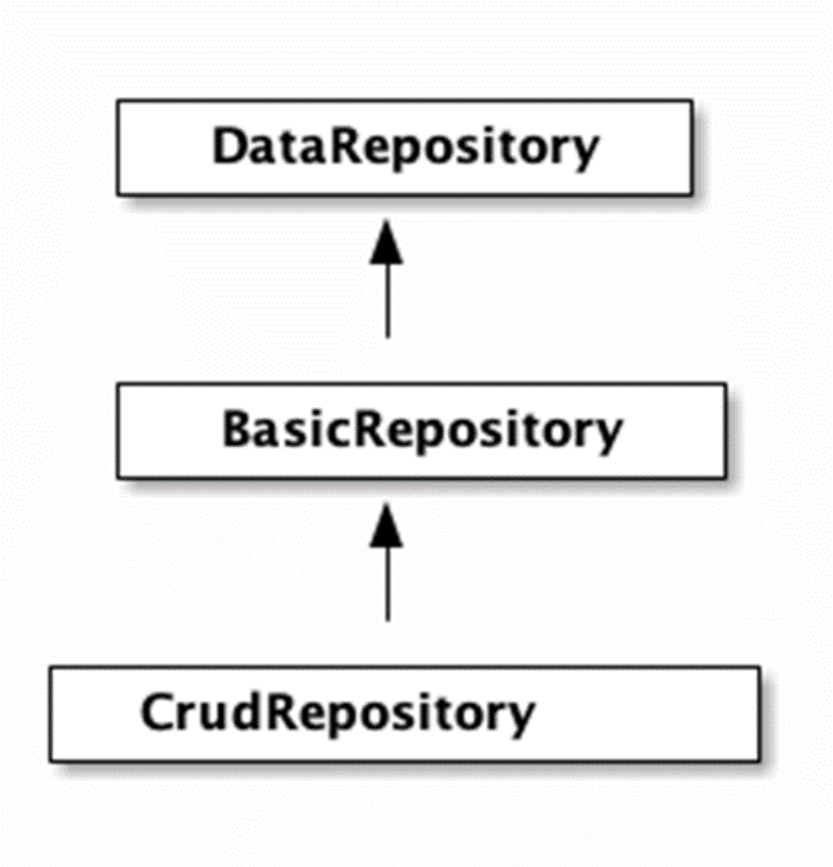
基底インタフェース
標準で3種類のインタフェースが提供されている　　*後述スライドを参照

T: 対象データのエンティティ
K: 対象データのID型

開発者は
リポジトリインタフェースを定義するだけで
データアクセスレイヤーを実装できる

リポジトリ

一般的なデータアクセス処理が定義されたリポジトリインタフェースを提供



DataRepository

メソッドなし

BasicRepository

操作	メソッド
取得	findById(), findAll()
削除	delete(), deleteById(), deleteAll()
保存	save(), saveAll()
	データが存在する　：エンティティの更新 データが存在しない: エンティティの挿入

CrudRepository

操作	メソッド
挿入	insert(), insertAll()
更新	update(), updateAll()

リポジトリ

■ 実装例

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {
}
```

ビジネスロジックからの呼び出し例

```
@Inject
private ProductRepository productRepository;
…
productRepository.findById(id);
productRepository.save(product);
```

■ 解説

Product というエンティティ(データ)に対し、取得/削除/保存 を操作するリポジトリを定義
ビジネスロジックからは、@Injectを使用してリポジトリを注入し、リポジトリメソッドを呼び出してデータ操作

リポジトリ

リポジトリインタフェースを継承することでデータアクセス処理をJakarta Data側で自動実装

なぜ、実装していないリポジトリメソッドが使えるのか？

■ **@Repository アノテーションを付与することで...**

- Jakarta Data がリポジトリインタフェースの**実装クラスを自動で生成**
- CDI(Contexts and Dependency Injection)が有効な場合は、CDI Bean定義のアノテーション
  - @Injectにより、CDIコンテナがインスタンスを自動生成
  - スコープアノテーション(@ApplicationScoped 等)により、インスタンスを自動管理

リポジトリメソッドの独自実装

リポジトリインタフェースで提供される処理で不十分な場合は、リポジトリメソッドを独自に実装

推奨

1. Lifecycle Methods

Jakarta Data 独自のライフサイクルアノテーションを使用してリポジトリメソッドを実装

推奨

2. Parameter-based Automatic Query Methods

メソッドの引数情報を使用して動的にクエリを生成

推奨

3. JDQL Query Methods

複雑なクエリをJPQLに似た構文で記述

拡張機能

4. Query by Method Name

リポジトリメソッド名から自動的にクエリを生成

1. Lifecycle Methods

Jakarta Data 独自のライフサイクルアノテーションを使用してリポジトリメソッドを実装

■ **基本構文**

```
@Repository
修飾子 interface リポジトリ名 [extends 基底インタフェース<T, K>] {
    @Insert
    返り値型 メソッド名(引数);

    @Update
    返り値型 メソッド名(引数);

    @Save
    返り値型 メソッド名(引数);

    @Delete
    返り値型 メソッド名(引数);
}
```

1. Lifecycle Methods

■ Jakarta Data 独自のライフサイクルアノテーション

アノテーション	説明
@Insert	データの新規作成(追加)
@Update	既存データの更新
@Save	データが存在する　：データの更新 データが存在しない: データの新規作成(追加)
@Delete	データの削除

1. Lifecycle Methods

■ 実装例(@Insert,@Update,@Save)

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {
    @Insert
    Product add(Product product);

    @Insert
    List<Product> addAll(List<Product> products);

    @Update
    Product[] modifyAll(Product[] products);

    @Save
    void upsert(Product product);
}
```

■ 解説

Product というエンティティ(データ)に対し、追加/更新/保存 を操作する独自のリポジトリメソッドを定義

1. Lifecycle Methods

■ 実装例(@Insert,@Update,@Save)

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {
    @Insert
    Product add(Product product);
    @Insert
    List<Product> addAll(List<Product> products);
    @Update
    Product[] modifyAll(Product[] products);
    @Save
    void upsert(Product product);
}
```

E

E

List<E>

E[]

void

@Insert/Update/Saveを使用したリポジトリメソッドの返り値型は、
下記のいずれかに限定されます

- E
- E[]
- List<E>
- void

■ 解説

Product というエンティティ(データ)に対し、追加/更新/保存 を操作する独自のリポジトリメソッドを定義

1. Lifecycle Methods

■ **実装例(@Delete)**

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {

    @Delete
    void erase(Product product);
}
```

■ **解説**

Product というエンティティ(データ)に対し、削除 を操作する独自のリポジトリメソッドを定義

1. Lifecycle Methods

■ 実装例(@Delete)

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {

    @Delete
    void erase(Product product);
}
```

■ 解説

Product というエンティティ(データ)に対し、削除 を操作する独自のリポジトリメソッドを定義

リポジトリメソッドの返り値型は、
voidのみに限定されます

2. Parameter-based Automatic Query Methods

メソッドの引数情報を使用して動的にクエリを生成

■ **基本構文**

```
@Repository
修飾子 interface リポジトリ名 [extends 基底インタフェース<T, K>] {

    @アノテーション
    返り値型 メソッド名(フィールド名と同一の引数);
}
```

2. Parameter-based Automatic Query Methods

■ 使用可能なアノテーション

アノテーション	説明
@Delete	引数名とフィールド名が同じ場合、値が一致するデータを**削除** 引数名とフィールド名が異なる場合、 **@By**アノテーションを引数に付与し、引数名とフィールド名を対応付け 1. Lifecycle Methods を参照
@Find	引数名とフィールド名が同じ場合、値が一致するデータを**取得(検索)** 引数名とフィールド名が異なる場合、 **@By**アノテーションを引数に付与し、引数名とフィールド名を対応付け

*1つのリポジトリメソッドに対し併用はできない

2. Parameter-based Automatic Query Methods

■ **実装例**

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {

    @Find
    Product findByName(String name);

    @Find
    Optional<Product> findByName(String name);
}
```

■ **解説**

Product というエンティティ(データ)に対し、取得(検索) を操作する独自のリポジトリメソッドを定義
各リポジトリメソッドは、name 引数 と Product.name プロパティの値が同じデータを検索・取得

2. Parameter-based Automatic Query Methods

■ 実装例

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {
    @Find
    Product[] findByManufacturer(String manufacturer);

    @Find
    List<Product> findByManufacturer(String manufacturer);

    @Find
    Stream<Product> findByCategory(String category);
}
```

■ 解説

Product というエンティティ(データ)に対し、取得(検索) を操作する独自のリポジトリメソッドを定義
findByManufacturerメソッドは、manufacturer 引数 と Product.manufacturer プロパティの値が同じデータを検索・取得
findByCategoryメソッドは、category 引数 と Product.category プロパティの値が同じデータを検索・取得

2. Parameter-based Automatic Query Methods

■ 実装例

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {
                                                           E
    @Find
    Product findByName(String name);
            E
    @Find
    Optional<Product> findByName(String name);
              Optional<E>
    @Find
    Product[] findByManufacturer(String manufacturer);
            E[]
    @Find
    List<Product> findByManufacturer(String manufacture
              List<E>
    @Find
    Stream<Product> findByCategory(String category);
              Stream<E>
}
```

@Findを使用したリポジトリメソッドの返り値型は、下記のいずれかに限定されます

- E
- Optional<E>
- E[]
- List<E>
- Stream<E>
- Page<E> *後述
- CursoredPage<E> *後述

2. Parameter-based Automatic Query Methods

■ **実装例**

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {

    @Find
    List<Product> findByName(@By("name") String productName);

    @Find
    List<Product> findByManufacturerAndCategory(String manufacturer, String category);
}
```

■ **解説**

Product というエンティティ(データ)に対し、取得(検索) を操作する独自のリポジトリメソッドを定義
findByName メソッドは、productName 引数 と Product.name プロパティの値が同じデータを検索
findByManufacturerAndCategoryメソッドは、下記の値が同じデータを検索

- manufacturer 引数 と Product.manufacturer プロパティ
- category 引数 と Product.category プロパティ

3. JDQL Query Methods

複雑なクエリをJPQLに似た構文で記述

■ **基本構文**

```
@Repository
修飾子 interface リポジトリ名 [extends 基底インタフェース<T, K>] {

    @Query("JDQL文")
    返り値型 メソッド名(引数);
}
```

@Query("JDQL文")

本アノテーションの引数にJDQLを記述し、より高度なデータ操作を実現

JDQL文におけるパラメータはメソッドの引数と紐づけできる

3. JDQL Query Methods

JDQL(Jakarta Data Query Language)

■ 識別子とキーワード

select, update, set, delete, from, where, order, by, asc,
desc, not, and, or, between, like, in, null, local, true, false

*大文字と小文字は区別しない

■ パラメータ化による値の指定

- 名前付きパラメータ(:name)
- 位置パラメータ(?1)

■ 演算子

- + - * / || = < > <> <= >=
- ( ) 句読点扱い

■ 数値リテラル

- 整数リテラル(INTEGER): int, long型相当
- 小数リテラル(DOUBLE): float, double型相当

■ 記述省略

- SELECT句: SELECT対象がエンティティ全体の場合
- FROM句: 対象がリポジトリの型から推測可能な場合

■ JPQL(Jakarta Persistence Query Language)

- プロバイダーが対応している場合は記述可能

3. JDQL Query Methods

■ 実装例

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {
    @Query("select Product from Product where name like :pattern")
    Product[] byNamePattern(String pattern);

    @Query("where name like :pattern")
    List<Product> byNamePattern(@Param("pattern") String productNamePattern);

    @Query("where category = ?1 and manufacturer = ?2")
    List<Product> byCategoryAndManufacturer(String category, String manufacturer);
}
```

■ 解説

Product というエンティティ(データ)に対し、取得(検索) を操作する独自のリポジトリメソッドを定義
byNamePatternメソッドは、Productテーブルのnameカラムと引数の値(パターン)に一致するデータを取得
byCategoryAndManufacturerメソッドは、Productテーブルのcategory, manufacturerカラムと引数の値が同じデータを取得

3. JDQL Query Methods

■ 実装例

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {
    @Query("select Product from Product where name like :pattern")
    Product[] byNamePattern(String pattern);

    @Query("where name like :pattern")
    List<Product> byNamePattern(@Param('pattern') String productNamePattern);

    @Query("where category = ?1 and manufacturer = ?2")
    List<Product> byCategoryAndManufacturer(String category, String manufacturer);
}
```

■ 解説

Product というエンティティ(データ)に対し、取得(検索) を操作する独自のリポジトリメソッドを定義
byNamePatternメソッドは、Productテーブルのnameカラムと引数の値(パターン)に一致するデータを取得
byCategoryAndManufacturerメソッドは、Productテーブルのcategory, manufacturerカラムと引数の値が同じデータを取得

3. JDQL Query Methods

■ 実装例

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {
    @Query('select Product from Product where name like :pattern")
    Product[] byNamePattern(String pattern);
            省略          省略

    @Query("where name like :pattern")
    List<Product> byNamePattern(@Param("pattern") String productNamePattern);

    @Query("where category = ?1 and manufacturer = ?2")
    List<Product> byCategoryAndManufacturer(String category, String manufacturer);
}
```

■ 解説

Product というエンティティ(データ)に対し、取得(検索) を操作する独自のリポジトリメソッドを定義
byNamePatternメソッドは、Productテーブルのnameカラムと引数の値(パターン)に一致するデータを取得
byCategoryAndManufacturerメソッドは、Productテーブルのcategory, manufacturerカラムと引数の値が同じデータを取得

4. Query by Method Name

リポジトリメソッド名から自動的にクエリを生成

■ 注意事項

同様の機能を持つ既存アプリケーションからの
移行容易性のために提供される機能

Jakarta Data 1.0 では必須要件の1つであるが、
将来的に本機能(Query by Method Name)は、
サポートされない/廃止される可能性が示唆されている

⇒前述の1~3で紹介した機能の使用を推奨します。

■ Jakarta Data 1.0 Specification Document

以下、抜粋
As an extension to the core specification, Jakarta Data 1.0 offers a Query by Method Name facility to provide a migration path for existing applications written for repository frameworks which offer similar functionality. Query by Method name is described in a companion document to this specification.

A Jakarta Data provider is required to support the Query by Method Name extension in Jakarta Data 1.0. This requirement will be removed in a future version of Jakarta Data.

4. Query by Method Name

■ **基本構文**

```
@Repository
修飾子 interface リポジトリ名 [extends 基底インタフェース<T, K>] {

    返り値型 メソッド名(引数);
}
```

返り値型 **メソッド名**(引数);
メソッド名は「動詞 + By + プロパティ名 + 条件キーワード」で構成
構成された要素から自動的にクエリを発行

4. Query by Method Name

■ **メソッド名の命名規則**

動詞 + By + プロパティ名 + **条件キーワード**

動詞
count
delete
exists
find

条件キーワード	
And	LessThan
Between	LessThanEqual
Contains	Like
EndsWith	Not
False	Null
GreaterThan	Or
GreaterThanEqual	StartsWith
IgnoreCase	True
In	

その他
Asc
Desc
OrderBy
First

4. Query by Method Name

■ 実装例(find動詞)

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {
    Product findByName(String name);
    Optional<Product> findByName(String name);

    Product[] findByCategory(String category);
    List<Product> findByCategory(String category);

    Stream<Product> findByPriceLessThanEqual(int price);
}
```

■ 解説

Product というエンティティ(データ)に対し、取得(検索)を操作する独自のリポジトリメソッドを定義
findByNameメソッドは、Product.name(プロパティ)が引数と同じデータを取得(find)
findByCategoryメソッドは、Product.category(プロパティ)が引数と同じデータを取得(find)
findByPriceLessThanEqualメソッドは、Product.price(プロパティ)が引数の値以下(LessThanEqual)のデータを取得(find)

4. Query by Method Name

■ 実装例(find動詞)

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {
    Product findByName(String name);
    Optional<Product> findByName(String name);
    Product[] findByCategory(String category);
    List<Product> findByCategory(String category);
    Stream<Product> findByPriceLessThanEqual(int price);
}
```

E
E
Optional<E>
E[]
List<E>
Stream<E>

■ 解説

Product というエンティティ(データ)に対し、取得(検索)を操作す
findByNameメソッドは、Product.name(プロパティ)が引数と同じ
findByCategoryメソッドは、Product.category(プロパティ)が引数
findByPriceLessThanEqualメソッドは、Product.price(プロパティ

動詞findを使用したリポジトリメソッドの返り値型は、下記のいずれかに限定されます

- E
- Optional<E>
- E[]
- List<E>
- Stream<E>
- Page<E> *後述
- CursoredPage<E> *後述

4. Query by Method Name

■ **実装例(count, exists動詞)**

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {

    long countByManufacturerNull();

    boolean existsByName(String name);
}
```

■ **解説**

Product というエンティティ(データ)に対し、件数と存在有無の取得を操作する独自のリポジトリメソッドを定義
countByManufacturerNullメソッドは、Product.manufacturer(プロパティ)がNULLであるデータの件数を取得(count)
existsByNameメソッドは、Product.name(プロパティ)が引数と同じデータが存在するかの結果を取得(exists)

4. Query by Method Name

■ 実装例(count, exists動詞)

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {

    long countByManufacturerNull();
    long

    boolean existsByName(String name);
    boolean
}
```

■ 解説

Product というエンティティ(データ)に対し、件数と存在有無の取得を操作する独自のリポジトリメソッドを定義
countByManufacturerNullメソッドは、Product.manufacturer(プロパティ)がNULLであるデータの件数を取得(count)
existsByNameメソッドは、Product.name(プロパティ)が引数と同じデータが存在するかの結果を取得(exists)

動詞countを使用したリポジトリメソッドの返り値型は、longのみに限定されます。

動詞existsを使用したリポジトリメソッドの返り値型は、booleanのみに限定されます。

4. Query by Method Name

■ **実装例(delete動詞)**

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {

    void deleteByActiveFalse();

    long deleteByManufacturer(String manufacturer);

    int deleteByNameContains(String keyword);
}
```

■ **解説**

Product というエンティティ(データ)に対し、削除を操作する独自のリポジトリメソッドを定義
deleteByActiveFalseメソッドは、Product.active(プロパティ)がFALSEであるデータを削除(delete)
deleteByManufacturerメソッドは、Product.manufacturer(プロパティ)が引数と同じデータを削除(delete)
deleteByNameContainsメソッドは、Product.name(プロパティ)が引数の値(文字列)を含むデータを削除(delete)

4. Query by Method Name

■ 実装例(delete動詞)

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {

    void deleteByActiveFalse();

    long deleteByManufacturer(String manufacturer);

    int deleteByNameContains(String keyword);
}
```

■ 解説

Product というエンティティ(データ)に対し、削除を操作する独
deleteByActiveFalseメソッドは、Product.active(プロパティ)が
deleteByManufacturerメソッドは、Product.manufacturer(プロ
deleteByNameContainsメソッドは、Product.name(プロパティ

動詞deleteを使用したリポジトリメソッドの返り値型は、下記のいずれかに限定されます

- **void**
- **long** *削除した件数を返す
- **int** *削除した件数を返す

より高度な活用方法

整列

取得結果を指定した条件で並び替え

■ **基本構文**

```
@Repository
修飾子 interface リポジトリ名 [extends 基底インタフェース<T, K>] {
    @Find
    @OrderBy(“属性名”) または @OrderBy(value = "属性名", オプション = <true|false>)
    返り値型 メソッド名(引数);

    @Find
    返り値型 メソッド名(Sort<T> sort[, 引数]);
}
```

@OrderBy
引数の属性名と一致するプロパティ名を基準に取得結果を整列。昇降順や小・大文字の区別有無はオプションで指定

Sort<T>
Sortオブジェクトに設定された条件を基準に取得結果を整列

整列

■ **実装例(@OrderBy)**

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {

    @Find
    @OrderBy("category")
    @OrderBy(value = "price", descending = true)
    @OrderBy(value = "name", ignoreCase = true)
    Stream<Product> findAll();
}
```

■ **解説**

Product というエンティティ(データ)に対し、取得を操作し、結果を整列する独自のリポジトリメソッドを定義
取得結果は、Product.category > Product.price > Product.name の優先順位で整列

整列

■ 実装例(@OrderBy)

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {

    @Find
    @OrderBy("category")
    @OrderBy(value = "price", descending = true)
    @OrderBy(value = "name", ignoreCase = true)
    Stream<Product> findAll();
}
```

降順の有効化

小・大文字の区別なし

■ 解説

Product というエンティティ(データ)に対し、取得を操作し、結果を整列する独自のリポジトリメソッドを定義

取得結果は、Product.category > Product.price > Product.nam

@OrderBy アノテーションを利用することで
リポジトリメソッド側で整列条件を指定可

呼び出し側で整列条件を指定不可

整列

■ **実装例(Sortオブジェクト)**

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {
    @Find
    Stream<Product> findAll(Sort<Product> sort);
}
```

ビジネスロジックからの呼び出し例

```
@Inject
private ProductRepository productRepository;
…
Sort<Product> sort = Sort.asc("category").desc("price").ascIgnoreCase("name");
productRepository.findAll(sort);
```

■ **解説**

Product というエンティティ(データ)に対し、取得を操作し、結果を整列する独自のリポジトリメソッドを定義
取得結果は、Product.category > Product.price > Product.name の優先順位で整列

整列

■ 実装例(Sortオブジェクト)

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {
    @Find
    Stream<Product> findAll(Sort<Product> sort);
}
```

ビジネスロジックからの呼び出し例

```
@Inject
private ProductRepository productRepository;
…
Sort<Product> sort = Sort.asc("category").desc("price").ascIgnoreCase("name");
productRepository.findAll(sort);
```

降順の有効化

小・大文字の区別なし

■ 解説

Product というエンティティ(データ)に対し、取得を操作し、結
取得結果は、Product.category > Product.price > Product.nam

Sort オブジェクトを利用することで
呼び出し側で整列条件を指定可

動的な整列を実現

件数制限

対象件数や範囲を指定して結果を取得

■ **基本構文**

```
@Repository
修飾子 interface リポジトリ名 [extends 基底インタフェース<T, K>] {

    @Find
    Page<T> メソッド名(PageRequest pageRequest[, 引数]);

    @Find
    返り値型 メソッド名(Limit limit[, 引数]);
}
```

`Page<T>`、`PageRequest`
ページネーションのリクエスト情報と取得結果を指定

`Limit`
シンプルな件数制限を指定

件数制限

■ 実装例(Page, PageRequest)

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {
    @Find
    Page<Product> findByCategory(String category, PageRequest pageRequest);
}
```

ビジネスロジックからの呼び出し例

```
@Inject
private ProductRepository productRepository;
…
PageRequest pageRequest = PageRequest.ofPage(2).size(10);
Page<Product> page = productRepository.findByCategory(category, pageRequest);
```

■ 解説

Product というエンティティ(データ)に対し、件数や範囲を設定し取得を操作する独自のリポジトリメソッドを定義
findByCategoryメソッドは Product.category(プロパティ)が同じ2ページ目10件分のデータを取得

件数制限

■ 実装例(Page, PageRequest)

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {
    @Find
    Page<Product> findByCategory(String category, PageRequest pageRequest);
}
```

ビジネスロジックからの呼び出し例

```
@Inject
private ProductRepository productRepository;
…
PageRequest pageRequest = PageRequest.ofPage(2).size(10);  2ページ目 10件分⇒11~20までのデータ
Page<Product> page = productRepository.findByCategory(category, pageRequest);
```

■ 解説

Product というエンティティ(データ)に対し、件数や範囲を設定
findByCategoryメソッドは Product.category(プロパティ)が同じ

Page, PageRequest オブジェクトを利用することで
ページ情報*を含んだ形で件数を指定可

***データの総件数など**

件数制限

■ **実装例(Limit)**

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {
    @Find
    List<Product> findByCategory(String category, Limit limit);
}
```

ビジネスロジックからの呼び出し例

```
@Inject
private ProductRepository productRepository;
…
Limit limit = Limit.range(11, 20);
productRepository.findByCategory(category, limit);
```

■ **解説**

Product というエンティティ(データ)に対し、件数や範囲を設定し取得を操作する独自のリポジトリメソッドを定義
findByCategoryメソッドは Product.category(プロパティ)が同じ11~20件目のデータを取得

件数制限

■ 実装例(Limit)

```
@Repository
public interface ProductRepository extends BasicRepository<Product, String> {
    @Find
    List<Product> findByCategory(String category, Limit limit);
}
```

ビジネスロジックからの呼び出し例

```
@Inject
private ProductRepository productRepository;
…
Limit limit = Limit.range(11, 20);                    11~20までのデータ
productRepository.findByCategory(category, limit);
```

■ 解説

Product というエンティティ(データ)に対し、件数や範囲を設定
findByCategoryメソッドは Product.category(プロパティ)が同じ

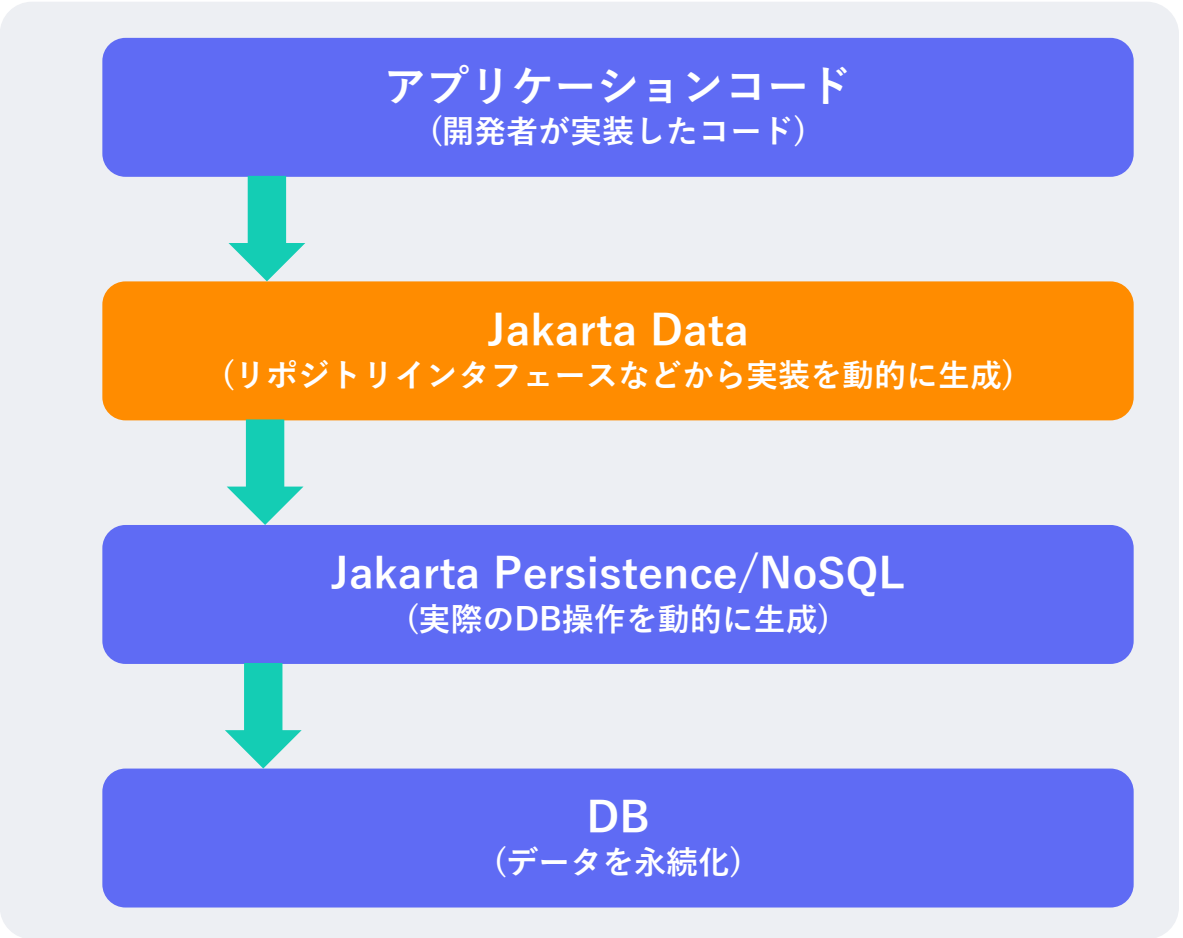
**Limit オブジェクトを利用することで
シンプルに件数を指定可**

振り返り

まとめ

Jakarta Data は Jakarta Persistence/NoSQL などの仕様の上で動作
Jakarta Data で実装できないアクセス処理は1つ下のレイヤーで実装

- ボイラーコードの削減による開発生産性の向上
- 最新仕様(機能)への追従が容易



最後に

Jakarta Data を活用することで
Javaエンタープライズアプリケーションの標準仕様(Jakarta EE)に準拠し
開発効率・堅牢性・信頼性などのエンタープライズ要件を満たすことができる

\Orchestrating a brighter world