Jakarta EE&MicroProfileセミナー
「待望のJakarta EE 12最新動向！データアクセス新仕様＆ついに標準化されるJakarta MVCを学ぶ」

Jakarta EE 12 が変えるデータアクセスの新仕様: Data, NoSQL, Query 解説

2026/06/22(Mon)　19:10-20:00
NEC　景井教天

変更履歴

変更日	版数	内容
2026/06/26	1.0 (初版)	
2026/06/27	1.1	誤字修正

© NEC Corporation 2026

自己紹介

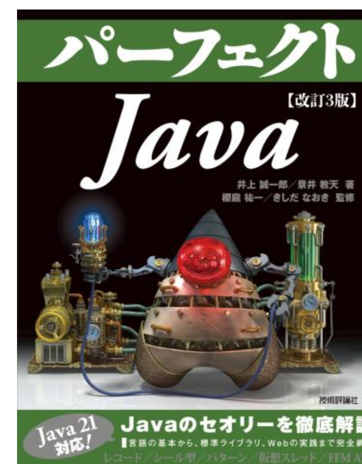
NEC
テクノロジーサービスソフトウェア統括部
景井 教天

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



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
https://jpn.nec.com/webotx/download/event/JakartaData.pdf

© NEC Corporation 2026

アジェンダ

1. Jakarta EE 12
2. Jakarta NoSQL
3. Jakarta Data
4. Jakarta Query
5. Jakarta Persistence

© NEC Corporation 2026

免責事項

本解説は Jakarta EE 12 マイルストーン版を基準としており、正式リリース時には内容が変更される場合があります。

掲載内容の正確性には配慮していますが、内容の誤り等によって生じた損害について、当方は一切の責任を負いかねます。

最終的な仕様は公式のリリース情報をご確認ください。

説明対象の各仕様:
- Jakarta NoSQL 1.1.0-M3
- Jakarta Query 1.0-M2
- Jakarta Data 1.1-M3
- Jakarta Persistence 4.0-M4

© NEC Corporation 2026

Jakarta EE 12

© NEC Corporation 2026

Jakarta EE 12 Platform

Updated Not Updated New

* Candidate for inclusion.

**May or may not be updated as a part of Security 5.0.

© NEC Corporation 2026

Jakarta EE 12 Platform

* Candidate for inclusion.

**May or may not be updated as a part of Security 5.0.

© NEC Corporation 2026

Jakarta EE 12 は Under Development
※2026/6/22 時点

© NEC Corporation 2026

Jakarta NoSQL

1.1

© NEC Corporation 2026

Jakarta NoSQL とは

NoSQL操作の標準化

Java開発者がスケーラブルで
特定のデータベースに依存しないアプリケーションを
NoSQL技術を用いて構築できるように設計された標準仕様およびAPI

© NEC Corporation 2026

仕様の立ち位置

Non-Goal

- ORMの完全再現でない
- 全NoSQLとの完全互換でない
- DB固有機能の置換でない

Goal

- 生産性の向上
- リッチなオブジェクトマッピング
- 柔軟性と適応性

© NEC Corporation 2026

API

package: **jakarta.nosql**

オブジェクトマッピング

@Entity @Column @Id などのアノテーションを使用し、Javaオブジェクトと NoSQLデータベース構造を対応付け

カスタム変換

@Convert アノテーション & AttributeConverter インタフェースを使用し、複雑なデータ型に対応

継承のサポート

@MappedSuperclass @DiscriminatorColumn @DiscriminatorValue を使用し、エンティティの継承をモデル化

シームレスな統合

Java EE/Jakarta EE 環境と最新のフレームワークと統合し、互換性と導入容易性を確保

© NEC Corporation 2026

概念・モデル

エンティティ

NoSQLに格納される構造化/半構造化データのスキーマ

Javaでは

エンティティクラスとして表現
⇒写像（field ⇔ DB属性）を定義

© NEC Corporation 2026

mutable / immutable

- アノテーション駆動で写像を明示的に定義
  - jakarta.nosql.Entity (必須)
    エンティティ定義アノテーション
  - jakarta.nosql.Id (必須)
    一意識別子アノテーション
- 可変・不変・両者混在のいずれもモデルとして許容

エンティティ定義

```
// mutable
@Entity
public class Book {
    @Id
    private UUID id;

    @Column
    private String title;

    @Column
    private String author;
}

// immutable
@Entity
public record Book(
    @Id UUID id,
    @Column String title,
    @Column String author) { }
```


© NEC Corporation 2026

コンストラクタ規約

1. public / protected かつ 無引数 または @Column / @Id 付き引数 を持つ
2. コンストラクタのアノテーションはDB読み込み(生成)、フィールドのアノテーションはDB書き込みに必要
3. 無引数 / アノテーション付きの両方が存在する場合、アノテーション付きが優先される
4. アノテーションのない引数は無視され、無引数が優先される
5. @Id / @Column を使用するコンストラクタを複数持ってはならない

© NEC Corporation 2026

永続フィールド

01

Basic Field

[対応必須]
プロバイダがネイティブ対応する基本型

02

Embedded Field(埋込)

[プロバイダ/DB依存]
より細粒度のJavaクラス状態を取り込む

03

Association Field(関連)

[プロバイダ/DB依存]
エンティティ間の関連を表す

© NEC Corporation 2026

永続フィールド｜Basic Field

基本型	説明
プリミティブ型 **(ラッパーを含む)**	int, double, boolean … (Integer, Double, Boolean …)
java.lang.String	テキストデータ
java.time	LocalDate, LocalDataTime, LocalTime, Instant
java.util.UUID	一意識別子
java.math	BigInteger, BigDecimal
byte[]	バイナリデータ
enum **(ユーザ定義)**	既定で name() を用い文字列として保存

エンティティ定義

```
public enum Category {
    FICTION, EDUCATION, SCIENCE, HISTORY
}
---
@Entity
public class Book {
    @Id
    private UUID id;


    @Column
    private String title;


    @Column
    private String author;


    @Column
    private LocalDate publishedDate;


    @Column
    private Category category;
}
```


© NEC Corporation 2026

永続フィールド｜Embedded Field

後述: @Embeddable(EmbeddableType.FLAT|GROUPING)で指定

FLAT

埋め込みクラスのフィールドを親の構造に直接展開

JSON

```
{
    "id": 1,
    "name": "Duke",
    "country": "Japan",
    "city": "Tokyo",
    "street": "1-1-1 Chiyoda",
    "postalCode": "100-0001"
}
```

GROUPING

- フィールドを構造型にまとめる

JSON

```
{
    "id": 1,
    "name": "Duke",
    "address": {
        "country": "Japan",
        "city": "Tokyo",
        "street": "1-1-1 Chiyoda",
        "postalCode": "100-0001"
    }
}
```


© NEC Corporation 2026

永続フィールド | Embedded Field

エンティティ定義

```
@Entity
public class Driver {
    @Id
    private UUID id;

    @Column
    private String name;

    @Column
    private Iterable<Car> cars;
}

---

@Embeddable
public class Car {
    @Column
    private String plate;

    @Column
    private String category;
}
```

反復可能フィールドは GROUPING 扱い

JSON

```
{
    "id": "1234567889abcdef...",
    "name": "Duke",
    "cars": [
        {
            "plate": "あ:0123",
            "category": "SUV"
        },
        {
            "plate": "い:9876",
            "category": "Sedan"
        }
    ]
}
```


© NEC Corporation 2026

永続フィールド｜Association Field

- 関連フィールド:
  他の独立した（独自のIDを持つ）エンティティとの
  結びつきを表すもの

- NoSQLは結合を持たないため、GROUPING 埋め込み扱い

- Map<K,V>:
  K: 基本型
  V: 基本型、@Entity 、@Embeddable

すべてのDBが関連をサポートするわけではない

エンティティ定義

```
@Entity
public class Tag {
    @Id
    private String id;

    @Column
    private String name;
}
---
@Entity
public class Post {
    @Id
    private String id;
    @Column
    private String title;

    // 関連付け
    @Column
    private Tag tag;
    @Column
    private Tag[] tags;
    @Column
    private List<Tag> tags;
    @Column
    private Map<String, Tag> tags;
}
```


© NEC Corporation 2026

プロパティ

- エンティティのプロパティ名は
  大小無視かつ一意でなければならない

- 単純プロパティ:
  フィールド名/アクセサ名がそのままプロパティ名である

- 埋め込みクラス:
  各階層のフィールド名を連結し算出できる
  （区切り文字は任意）

階層構造

```
Person
  - address
    - city

⇒ address.city
```


© NEC Corporation 2026

アノテーション

@Entity 永続エンティティの宣言	**@Id** 一意識別子	**@Column** 列への写像
@Embeddable 埋め込み可能クラス	**@Convert** 変換器の明示適用	**@Converter** 変換器クラスの宣言
@MappedSuperclass 共通写像の親クラス	**@Inheritance** 継承	**@DiscriminatorColumn** **@DiscriminatorValue** 識別子列&識別値

© NEC Corporation 2026

@Entity

エンティティ定義

```java
@Entity
public class Driver {
    @Id
    private UUID id;

    @Column
    private String name;
}
```

JSON

```json
{
    "id": "1234567889abcdef...",
    "name": "Duke"
}
```


© NEC Corporation 2026

@Id

エンティティ定義

```
@Entity
public class Driver {
    @Id
    private UUID id;

    @Column
    private String name;
}
```

autoincrement 戦略は提供しない（DB依存）

例示
Key-Value モデルにおいて
ID=null → NullPointerException

JSON

```
{
    "id": "1234567889abcdef...",
    "name": "Duke"
}
```


© NEC Corporation 2026

@Column

エンティティ定義

```java
@Entity
public class Driver {
    @Id
    private UUID id;

    @Column
    private String name;

    // @Column なし → 写像対象外(無視)
    private Car car;

    // UDT(User Defined Type)が存在し、マッピング
    @Column(udt = "kuruma")
    private Car car;
}
```

JSON

```json
{
    "id": "1234567889abcdef...",
    "name": "Duke"
}
```


© NEC Corporation 2026

@Embeddable

エンティティ定義

```
@Entity
public class Driver {
    @Id
    private UUID id;

    @Column
    private String name;

    @Column
    private Car car;
}

---

@Embeddable(EmbeddableType.GROUPING)
public class Car {
    @Column
    private String plate;

    @Column
    private String category;
}
```

既定では FLAT

JSON

```json
{
    "id": "1234567889abcdef...",
    "name": "Duke",
    "car": {
        "plate": "あ:0123",
        "category": "SUV"
    }
}
```


© NEC Corporation 2026

@Convert & @Converter

エンティティ定義

```java
@Entity
public class Employee {
    @Id
    private String id;


    @Column
    private String name;


    @Column(“income")
    @Convert(MoneyConverter.class)
    private Salary salary;
}
```

変換器

```java
@Converter
public class MoneyConverter implements AttributeConverter<Double, String>
{
    @Override
    public String convertToDatabaseColumn(Salary attribute) {
        // logic to convert Double to String
        // e.g., 5,000,000 -> "5M:JPY"
        return ???;
    }


    @Override
    public Double convertToEntityAttribute(String dbData) {
        // logic to convert String to Double
        // e.g., "5M:JPY" -> 5,000,000
        return ???;
    }
}
```


© NEC Corporation 2026

@Converter

エンティティ定義

```
@Entity
public class Employee {
    @Id
    private String id;


    @Column
    private String name;


    @Column(“income")
    // @Convert(MoneyConverter.class) 不要
    private Salary salary;
}
```

変換器

```
@Converter(autoApply = true)
public class MoneyConverter implements AttributeConverter<Salary, String>
{
    @Override
    public String convertToDatabaseColumn(Salary attribute) {
        // logic to convert Salary to String
        // e.g., 5,000,000 and "JPY" -> "5M:JPY"
        return dbData;
    }


    @Override
    public Salary convertToEntityAttribute(String dbData) {
        // logic to convert String to Double
        // e.g., "5M:JPY" -> 5,000,000 and "JPY"
        return salary;
    }
}
```


© NEC Corporation 2026

@MappedSuperclass

エンティティ定義

```java
@MappedSuperclass
public class Employee {
    @Id
    protected Integer id;

    @Column
    protected Address address;
}

---

@Entity
public class Programmer extends Employee {
    @Column
    protected Salary salary;
}
```

JSON

```json
{
  "id": 101,
    "address": {
        "country": "Japan",
        "city": "Tokyo",
        "street": "1-1-1 Chiyoda",
        "postalCode": "100-0001"
    },
  "salary": {
    "amount": 5000000,
    "currency": "JPY"
  }
}
```

Key-Valueモデルにおいて、
スキーマレスにより継承がサポートされない可能性あり

© NEC Corporation 2026

@Inheritance

エンティティ定義

```
@Entity
@Inheritance
public abstract class Notification {
    @Id
    private String id;

    @Column
    private String recipient;
}

---

@Entity
public class SMSNotification extends Notification {
    @Column
    private String phoneNumber;
}

---

@Entity
public class EmailNotification extends Notification {
    @Column
    private String email;
}
```

JSON

```
{
  "id": "notif-001",
  "recipient": "Duke",
  "phoneNumber": "+819012345678",
  "dtype": "SMSNotification"
}

---

{
  "id": "notif-002",
  "recipient": "Duke",
  "email": "hogehoge@example.com",
  "dtype": "EmailNotification"
}
```

識別子フィールドには、
一般的に dtype または _type が使用される

Key-Valueモデルにおいて、
スキーマレスにより継承がサポートされない可能性あり

© NEC Corporation 2026

@DiscriminatorColumn & @DiscriminatorValue

エンティティ定義

```
@Entity
@Inheritance
@DiscriminatorColumn(“notification_type")
public abstract class Notification {
    @Id
    private String id;
    @Column
    private String recipient;
}

---
@Entity
@DiscriminatorValue(“SMS")
public class SMSNotification extends Notification {
    @Column
    private String phoneNumber;
}

---
@Entity
@DiscriminatorValue("Email")
public class EmailNotification extends Notification {
    @Column
    private String email;
}
```

JSON

```
{
  "id": "notif-001",
  "recipient": "Duke",
  "phoneNumber": "+819012345678",
  “notification_type": "SMS"
}

---

{
  "id": "notif-002",
  "recipient": "Duke",
  "email": "hogehoge@example.com",
  “notification_type": "Email"
}
```

識別子フィールドには、
一般的に dtype または _type が使用される

© NEC Corporation 2026

Template (CRUD)

- DAO パターン:
  永続化エンジンへの基本APIを提供し、
  業務ロジックとデータアクセスを分離

- Template クラス:
  下記操作の定義済みメソッドを持つ
  - CRUD
  - Fluent API
  - TTL

データアクセス(CRUD)

```java
@Inject
private Template template;

...

Book book = Book.builder()
    .id(id)
    .title("Great Book")
    .author("Duke")
    .publishedDate(LocalDate.of(1925, 4, 10))
    .build();


template.insert(book);

template.find(Book.class, id);

template.delete(Book.class, id);
```


© NEC Corporation 2026

Template (Fluent API & TTL)

データアクセス(Fluent API)

```java
@Inject
private Template template;

...

Optional<Book> optionalBook = template.select(Book.class)
    .where("title").eq("Great Book")
    .singleResult();


template.update(Book.class)
    .set("title").to("NotBad Book")
    .where("author").eq("Duke")
    .execute();


template.delete(Book.class)
    .where("publishedDate").lt(LocalDate.of(2000, 1, 1))
    .execute();
```

Key-Valueモデルにおいて、
スキーマレスにより継承がサポートされない可能性あり

データアクセス(TTL)

```java
@Inject
private Template template;

...

// 30 分後に削除
template.insert(book, Duration.ofMinutes(30));


// DB依存で丸め込みが発生する可能性がある
// 例: 3660 秒 → 3600 秒
template.insert(book, Duration.ofSeconds(3660));
```

TTL未対応データベースでは、サポートされない

© NEC Corporation 2026

サポートする主要なデータベースモデル

Key-Value

キーによる直接参照を基本としたシンプルなモデル

Wide-Column

列指向の特性を持つモデル
大量のデータに対して、高いスケーラビリティを発揮する

Document

JSONなどの構造化データを扱うモデル
高度な検索と柔軟なスキーマを両立する

Graph

データ間の複雑な関係性をノード(Node)とエッジ(Edge)で表現するモデル
高度な追跡を実現する

© NEC Corporation 2026

Key-Value

- キーバリューストアは、**最も限定的なクエリ機能しか提供しません。**

- データ操作

SELECT:

• = による比較のみ かつ 一意識別子（@Id）に対して**のみ**サポート

【例】FROM Entity WHERE id = :id

• 他のフィールドや属性に対するフィルタリングは許可されていない

DELETE:

• SELECTと同様

UPDATE:

• サポートされていない

© NEC Corporation 2026

Wide-Column

■ データ操作

SELECT:

- = による比較 かつ 一意識別子（@Id）に対してサポート
- （Option）>, <, IN 等の比較による @Id 以下のフィールドに対するフィルタリング
  多くの場合は、セカンダリインデックスが必要となるため、パフォーマンスに影響を与える可能性あり

DELETE:

- SELECT と同様の制約に従って @Id が必要

UPDATE:

- SELECT と同様の制約に従って @Id が必要

追加条件への対応度合いは、ベンダー依存

© NEC Corporation 2026

Document

- Documentデータベースは通常、完全なクエリ機能を提供します。

- データ操作

SELECT:
- =、>、<、IN、AND、ORなど、幅広い演算子をサポート
- @Idだけでなく、複数のフィールドでフィルタリング可能
- 順序付け（ORDER BY）機能やページネーション機能が含まれる場合あり

UPDATE:
- 完全にサポート

DELETE:
- 完全にサポート

Jakarta Common Query Language (JCQL) を使用した表現力豊かなクエリに適する

NEC \Orchestrating a brighter world
© NEC Corporation 2026

Graph

- 高度なクエリパターンとフィルタリングをサポート

- データ操作

  SELECT、UPDATE、DELETEに対するサポート状況は、プロバイダによって異なる

  SELECT, UPDATE, DELETE:

  - クエリは、リレーションシップ、プロパティ、および深い構造を対象にできる

  <u>柔軟性の点では、文書データベースに匹敵することが多い</u>

© NEC Corporation 2026

カスタムプロバイダ

- これまでのモデルに当てはまらないデータベースであっても、統合することが可能

- データ操作

SELECT:
- = による比較 かつ 一意識別子（@Id）に対してサポート

DELETE:
- SELECTと同様

Jakarta Queryのセマンティクスを尊重する限り、フィルタリングやソート、更新などの追加機能も許可

© NEC Corporation 2026

簡易比較表

	SELECT/フィルタリング	UPDATE/DELETE
Key-Value	△ @Id のみ	△ UPDATE 不可、DELETE @Id のみ
Wide-Column	○ @Id（+ 2次索引で任意条件は任意対応）	○ @Id前提（プロバイダ依存）
Document	◎ 複数フィールド =,>,<,IN,AND,OR,並替/ページング	◎ 完全サポート
Graph	◎ 関係/プロパティ/深い階層へ高度なクエリ	? プロバイダ依存
その他/独自	△ ～ 最低限 @Id の ＝	△ ～ JCQL準拠なら追加機能可

とどのつまり、使用するデータベースが対応しているかどうか

© NEC Corporation 2026

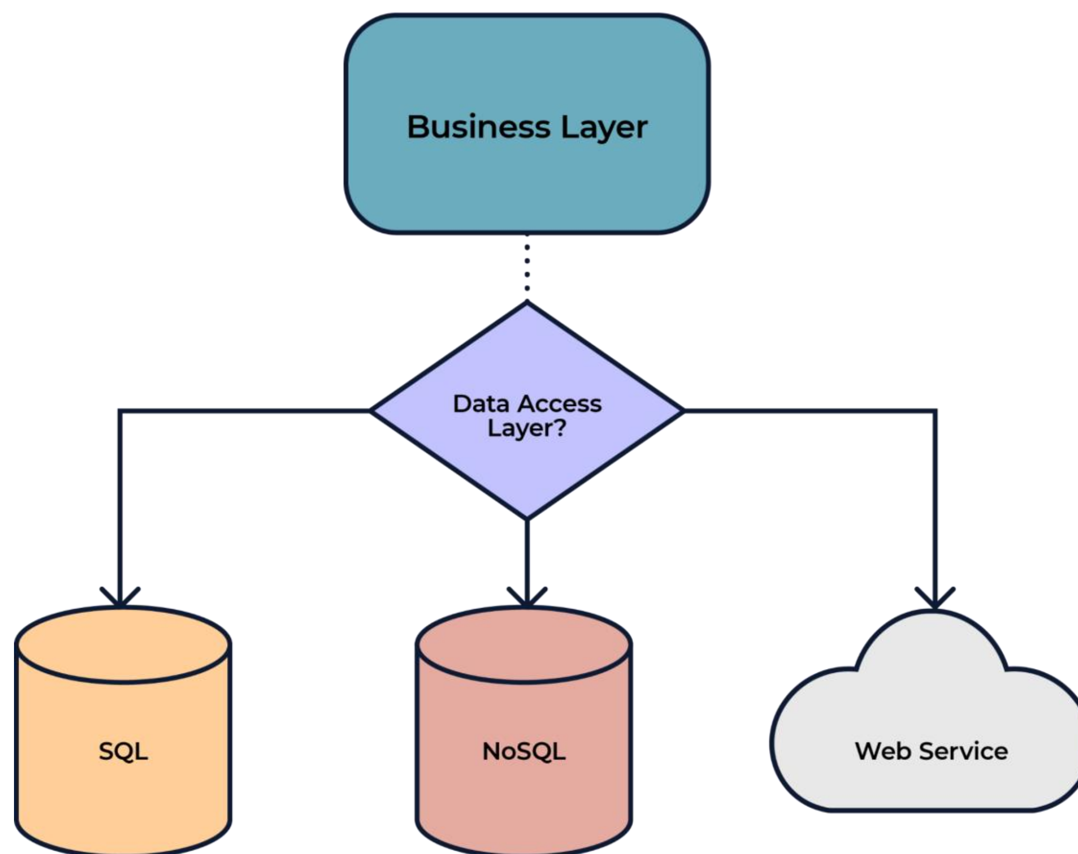
Jakarta Data

1.0 → 1.1

© NEC Corporation 2026

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

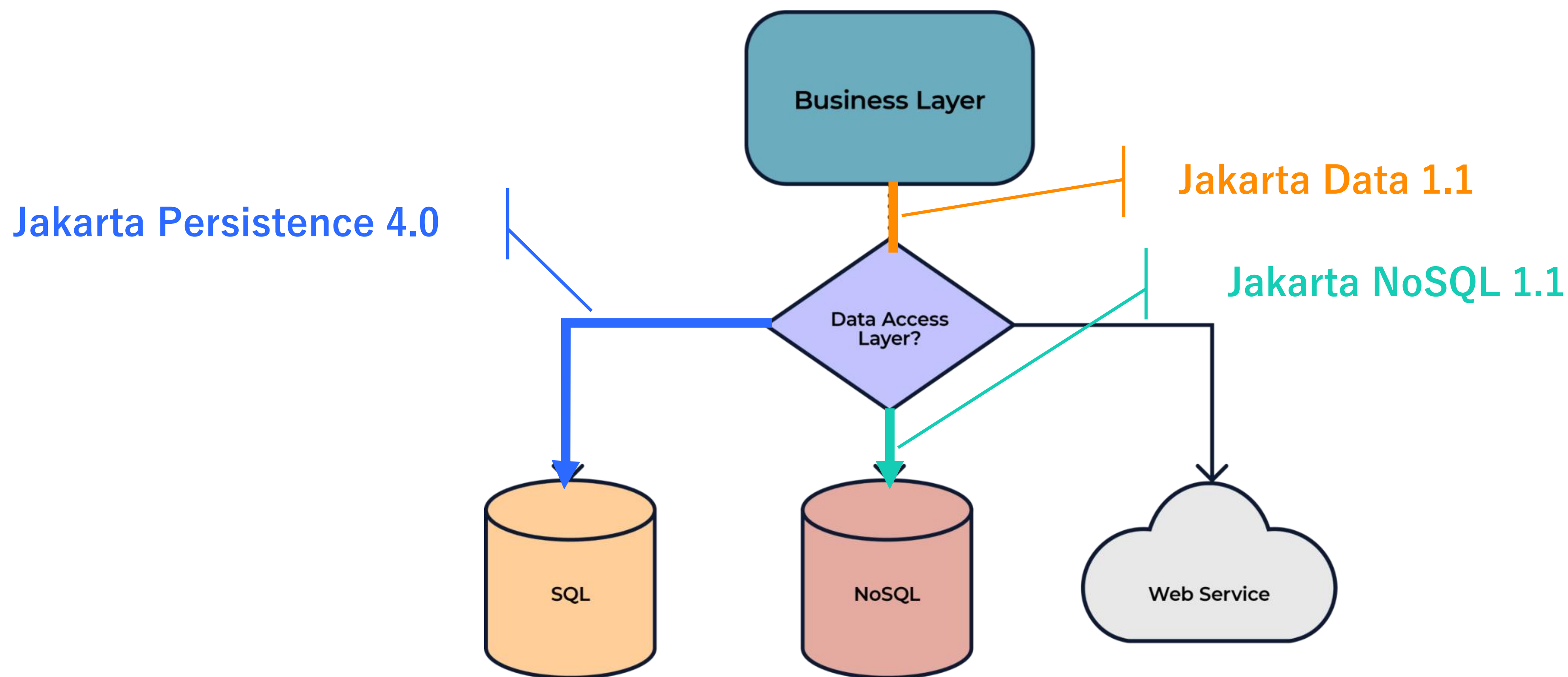


© NEC Corporation 2026

※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



© NEC Corporation 2026

リポジトリパターン

package: **jakarta.data.repository**

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

© NEC Corporation 2026

@Repository

リポジトリ定義

```
@Entity
public class Employee {
    @Id
    private String id;

    @Column
    private String name;

    @Column(“income")
    private Salary salary;
}

---

@Repository
public interface EmployeeRepository extends BasicRepository<Employee, Long>
{}
```

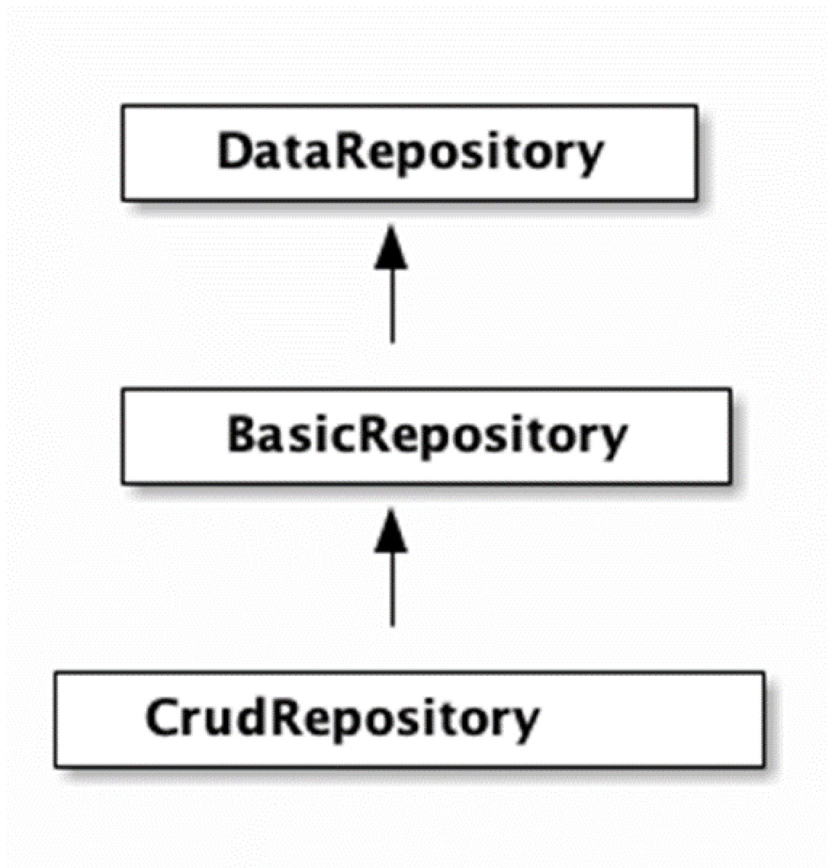
データアクセス(CRUD)

```
@Inject
private EmployeeRepository repository;

---

// すべてのEmployeeを取得
repository.findAll().forEach(System.out::println);

// id一致のEmployeeを取得
repository.findById(id).ifPresent(System.out::println);

// Employeeを更新
repository.save(employee);

// Employee一致のEmployeeを削除
repository.delete(employee);

// id一致のEmployeeを削除
repository.deleteById(id);
```


© NEC Corporation 2026

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()

© NEC Corporation 2026

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. Annotated Query Methods

複雑なクエリをJCQL/JPQL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

© NEC Corporation 2026

ステートレス

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

	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

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
@Insert/Update/Save: void, E, E[], List<E>
@Delete: void

リポジトリ定義

```
@Entity
public class Employee {
    @Id
    private String id;
    @Column
    private String name;
    @Column(“income")
    private Salary salary;
}

---

@Repository
public interface EmployeeRepository {
    @Insert
    Employee hire(Employee employee);

    @Update
    Employee[] update(Employee[] employees);

    @Save
    List<Employee> save(List<Employee> employees);

    @Delete
    void fire(Employee employee);
}
```


© NEC Corporation 2026

ステートフル

1. Lifecycle Methods

Jakarta Persistence のエンティティ管理と同等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

	説明
@Persist	データの新規作成(追加)
@Merge	データのマージ
@Refresh	データベースの最新に同期（変更分は破棄）
@Remove	データの削除
@Detach	コンテキストから切り離す

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
void, E, E[], List<E>

リポジトリ定義

```
@Entity
public class Employee {
    @Id
    private String id;
    @Column
    private String name;
    @Column("income")
    private Salary salary;
}
---
@Repository
public interface EmployeeRepository {
    @Persist
    Employee persist(Employee employee);

    @Merge
    Employee[] merge(Employee[] employees);

    @Refresh
    List<Employee> refresh(List<Employee> employees);

    @Remove
    void remove(Employee employee);

    @Detach
    Employee detach(Employee employee);
}
```


© NEC Corporation 2026

1. Lifecycle Methods

ステートレス

	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

ステートフル

	説明
@Persist	データの新規作成(追加)
@Merge	データのマージ
@Refresh	データベースの最新に同期（変更分は破棄）
@Remove	データの削除
@Detach	コンテキストから切り離す

混在不可

© NEC Corporation 2026

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

	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
@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

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
@Find: E, 任意<E>, E[], List<E>, Stream<E>,
Page<E>, CursoredPage<E>

リポジトリ定義

```
@Entity
public class Employee {
    @Id
    private String id;
    @Column
    private String name;
    @Column(“income")
    private Salary salary;
}
---
@Repository
public interface EmployeeRepository
                                 extends DataRepository<Employee, String> {
    @Find
    Employee whoRU(String name);

    @Delete
    void byebye(String id);

    @Delete
    void goodbye(@By("id") String identifier);

    // メタモデルで指定することで型安全を確保できる
    @Delete
    void graduate(@By("_Employee.id") String identifier);
}
```

3. Annotated 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

	説明
@Query	引数にJCQL/JPQLを記述し、より高度なデータ操作を実現 パラメータは下記の2種類を使用でき、メソッドの引数と紐づけできる 1. 位置パラメータ (?1) 2. 名前付きパラメータ (:name)

リポジトリ定義

```
@Entity
public class Employee {
    @Id
    private String id;
    @Column
    private String name;
    @Embeddable
    private Salary salary;
}

---

@Repository
public interface EmployeeRepository
                                 extends DataRepository<Employee, String> {

    @Query("SELECT Employee FROM Employee WHERE name = ?1")
    Employee findByName(String name);

    @Query("FROM Employee WHERE salary.base < ?1")
    Employee[] highBaseSalaryEmployees(double baseSalary);

    @Query("DELETE FROM Employee WHERE id = :id")
    void seeYou(@Param("id") String identifier);
}
```


© NEC Corporation 2026

特別なパラメータ

	説明
整列 Sort, Order *@OrderBy	取得結果の整列を指定が可能 **ソート条件の指定** 指定したエンティティの属性に基づいて、クエリ結果のソート順（昇順・降順など）を動的に指定 **複数ソートの順序付け** 複数の属性を組み合わせた一連のソート基準（リスト）をカプセル化し、複雑な順序付け（例: 名字の昇順、そのあと名前の降順）を動的に指定
範囲 Page, PageRequest	**ページネーション** 結果をページ単位に分割（ページング）するための要求を制御します。オフセットベースのページングや、カーソルベース（Cursor-based）のページングの要求に使用
範囲 Limit	**取得件数・範囲の制限** クエリ結果として取得する最大件数（maxResults）や、取得開始位置（offset）を動的に制限
条件 Restriction	**追加のフィルタリング / 動的条件** 呼び出し側でエンティティの属性に対する制約（条件）を動的に組み立て、クエリの検索条件を制限・拡張

© NEC Corporation 2026

特別なパラメータ｜整列

リポジトリ定義

```java
@Entity
public class Book {
    @Id
    private UUID id;
    @Column
    private String title;
    @Column
    private String author;
    @Column
    private Double price;
    @Column
    private LocalDate publishedDate;
}
---
public interface BookRepository {
    @Find
    @OrderBy(_Book.price)
    @OrderBy(value = _Book.title, descending = true, ignoreCase = true)
    List<Book> getBooks();

    @Find
    List<Book> getBooks(Order<Book> order);

    @Find
    List<Book> getBooks(Sort<Book> sort);
}
```

データアクセス

```java
@Inject
private EmployeeRepository repository;

...

// 下記の優先順位で整列されたエンティティ群が返される
// 1. 価格の昇順
// 2. タイトル名の降順 (大小文字の区別なし)

// @OrderBy
repository.getBooks();
```

リポジトリメソッド側で整列設定を制御

リポジトリメソッドの呼び出し側で整列設定を制御

```java
// Order
Order<Book> order = Order.by(
    _Book.price.asc(),
    _Book.title.desc().ignoreCase()
);
repository.getBooks(order);

// タイトル名の降順で整列されたエンティティ群が返される
Sort<Book> sort = Sort.desc(_Book.title).ignoreCase();
repository.getBooks(sort);
```


© NEC Corporation 2026

特別なパラメータ｜範囲①

リポジトリ定義

```java
@Entity
public class Book {
    @Id
    private UUID id;
    @Column
    private String title;
    @Column
    private String author;
    @Column
    private Double price;
    @Column
    private LocalDate publishedDate;
}
---

public interface BookRepository {
    @Find
    Page<Book> getBooks(PageRequest request);
}
```

データアクセス

```java
@Inject
private EmployeeRepository repository;

...

// 指定範囲: 2ページ目 20件分 のデータ
PageRequest request = PageRequest.ofPage(2).size(20);
Page<Book> page = repository.getBooks(request);

// 指定範囲のエンティティ群を返す
page.content();
```

リポジトリメソッドの呼び出し側で範囲設定を制御

© NEC Corporation 2026

特別なパラメータ｜範囲②

リポジトリ定義

```java
@Entity
public class Book {
    @Id
    private UUID id;
    @Column
    private String title;
    @Column
    private String author;
    @Column
    private Double price;
    @Column
    private LocalDate publishedDate;
}
---

public interface BookRepository {
    @Find
    List<Book> getBooks(Limit limit);
}
```

データアクセス

```java
@Inject
private EmployeeRepository repository;

...

// 指定範囲: 0件目 から 5件目 のデータ
Limit limit = Limit.range(0, 5);

// 指定範囲のエンティティ群を返す
repository.getBooks(limit);
```

リポジトリメソッドの呼び出し側で範囲設定を制御

© NEC Corporation 2026

特別なパラメータ｜条件

リポジトリ定義

```
@Entity
public class Book {
    @Id
    private UUID id;
    @Column
    private String title;
    @Column
    private String author;
    @Column
    private Double price;
    @Column
    private LocalDate publishedDate;
}
---

public interface BookRepository {
    @Find
    List<Book> getBooks(Restriction<Book> restriction);
}
```

データアクセス

```
@Inject
private EmployeeRepository repository;

...

// 全条件を満たす
Restriction<Book> restriction = Restrict.all(
    // 条件 1
    _Book.publishedDate.between(someday, today),

    // 条件 2
    _Book.price.minus(discount).greaterThanOrEqual(budget)
);

// いずれかの条件を満たす
Restriction<Book> restriction = Restrict.any(
    // 条件 1,2,3, ...
);


repository.getBooks(restriction);
```

リポジトリメソッドの呼び出し側で条件設定を制御

© NEC Corporation 2026

リソースアクセッサ

下層の永続化リソースを直接返却する抽象メソッドを定義
（例えば、Jakarta Persistenceの EntityManager など）

ユースケース
Jakarta Dataの標準APIだけでは、複雑なバルク処理や、
データベース・永続化プロバイダ固有の高度な最適化、
ネイティブ機能を表現しきれないケース

制約
① 1リポジトリインターフェース内に、
基礎となる永続化リソースを公開するメソッドは**最大で1つ**

② 戻り値の型が jakarta.persistence.EntityManager である
リソースアクセサメソッドは、ステートフルなリポジトリのみ

リポジトリ定義

```
@Entity
public class Book {
    @Id
    private UUID id;
    @Column
    private String title;
    @Column
    private String author;
    @Column
    private Double price;
    @Column
    private LocalDate publishedDate;
}
---

public interface BookRepository {
    EntityManager getEntityManager();

    default void regisiterBooks(List<Book> books) {
        EntityManager em = getEntityManager();
        for (Book book : books) {
            em.persist(book);
        }
    }
}
```


© NEC Corporation 2026

Jakarta Query

1.0

© NEC Corporation 2026

Jakarta Query とは

<u>オブジェクト指向クエリ言語の標準</u>

Jakarta Persistence / Jakarta Data / Jakarta NoSQL その他類似の
永続化技術のために設計設計された標準仕様およびAPI

⇒関連(association)とサブタイプ多態を扱えるSQL方言

© NEC Corporation 2026

仕様の立ち位置

Non-Goal

- クエリ実行
- Javaプログラムへクエリの埋め込み方
- プログラムでのクエリ構築
- クエリで使用されるエンティティクラスの定義

Goal

- クエリ言語の構文と意味の標準化
- 共通部分集合を画定
- クエリ言語とJava言語の両要素における対応指針

© NEC Corporation 2026

概念

Persistence Language (永続言語)

Common Language (共通言語)

strict subset (厳密な部分集合)

共通言語

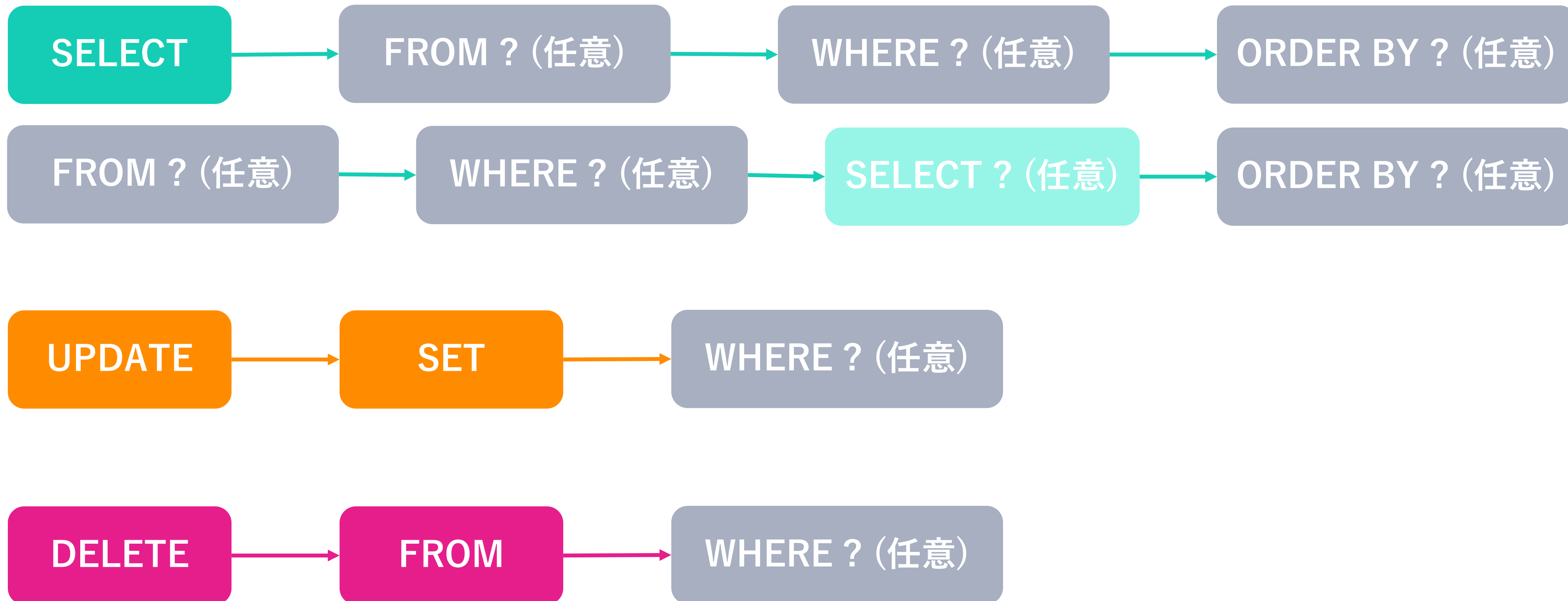
- SQL以外のデータストアでも実装可能なように設計
- Jakarta Data / Jakarta NoSQL が再利用

永続言語

- SQLデータベースで実装可能な範囲
- Jakarta Persistence が再利用

© NEC Corporation 2026

文法(JCQL)



© NEC Corporation 2026

文法(JCQL)

句の論理的な評価順序

FROM → WHERE → GROUPBY → HAVING → SELECT SET → ORDERBY

記述順 ≠ 論理順

先頭にselectを記述しても、上記の順番で評価される

共通言語

永続言語

© NEC Corporation 2026

識別子とキーワード

- 識別子

  以下のキーワードに該当しないJava識別子（大文字・小文字の区別あり）

- キーワード

  abs, all, and, any, as, asc, avg, between, both, by, case, ceiling, class, coalesce, concat, count, current_date, current_time, current_timestamp, delete, desc, distinct, else, empty, end, entry, escape, except, exists, exp, extract, false, fetch, first, floor, from, function, greatest, group, having, in, index, inner, intersect, is, join, key, last, leading, least, left, length, like, local, ln, locate, lower, max, member, min, mod, new, not, null, nulls, nullif, object, of, on, or, order, outer, position, power, replace, right, round, select, set, sign, size, some, sqrt, substring, sum, then, trailing, treat, trim, true, type, union, update, upper, value, when, where

© NEC Corporation 2026

式（1/2）

- パラメータ
  - 名前付き(:name)、位置(?1)
- 演算子・区切り
  - +, -, *, /, ||, =, <, >, <>, <=, >=
  - (,), and ,
- 文字列リテラル
  - ‘ (セルフエスケープあり)
- 単一文字リテラル
  - ‘ で囲む単一文字
- 数値リテラル
  - int, long, float, double
  - 接尾辞あり
    - L(64-bit integer), D(64-bit floating-point), F(32-bit floating-point)
  - 接尾辞なし
    - 指数/小数点なし: 32-bit整数
    - 指数/小数点あり: 64-bit整数
- 空白
  - スペース、水平タブ、改行、フォームフィード、キャリッジリターン
- 識別変数のスコープ
  - 宣言後の式・select句・サブクエリで参照可能

© NEC Corporation 2026

式（2/2）

- 特殊な値
  - bool(true, false), local_date, local_time, local_datetime
- Enum
  - 列挙型（Enum）の特定のメンバーを参照
- パス
  - エンティティの属性などをドット（.）で繋いでたどる式です（例: address.street）
- 関数
  - プロバイダが提供する組み込み関数の呼び出し
- サブクエリ式
  - 値が求められる場所において、
  - 結果として1つの要素（スカラー値）のみを返すサブクエリを記述可能
- 条件式
  - Null比較, IN, Between, Like, 等価・不等価演算し, 量化条件式, 論理演算子

© NEC Corporation 2026

Jakarta Persistence

3.2 → 4.0

© NEC Corporation 2026

主な新機能

- EntityAgent
  - 永続コンテキストを使用せずに
    データベースを直接更新するインタフェース
- 静的クエリ
  - jakarta.persistence.query クエリメソッドへのアノテーションで方安全なクエリを表現
- 結果セットマッピングのプログラマティックAPI
  - jakarta.persistence.sql SQL結果のマッピングをプログラムで定義
- 新ライフサイクルイベント
  - アノテーション群でデータベース操作の前後に
    よりきめ細かく処理をフックできる
  - 楽観的ロック（Optimistic Locking）からの除外
  - 読み取り専用モードでのロード
- EntityGraph 強化
  - データのフェッチ範囲を定義する「名前付きエンティティグラフ」をより直感的かつ簡潔に記述できる
  - また、refresh()メソッドの引数として直接エンティティティグラフを渡せるよう改善
- Jakarta Data 連携

© NEC Corporation 2026

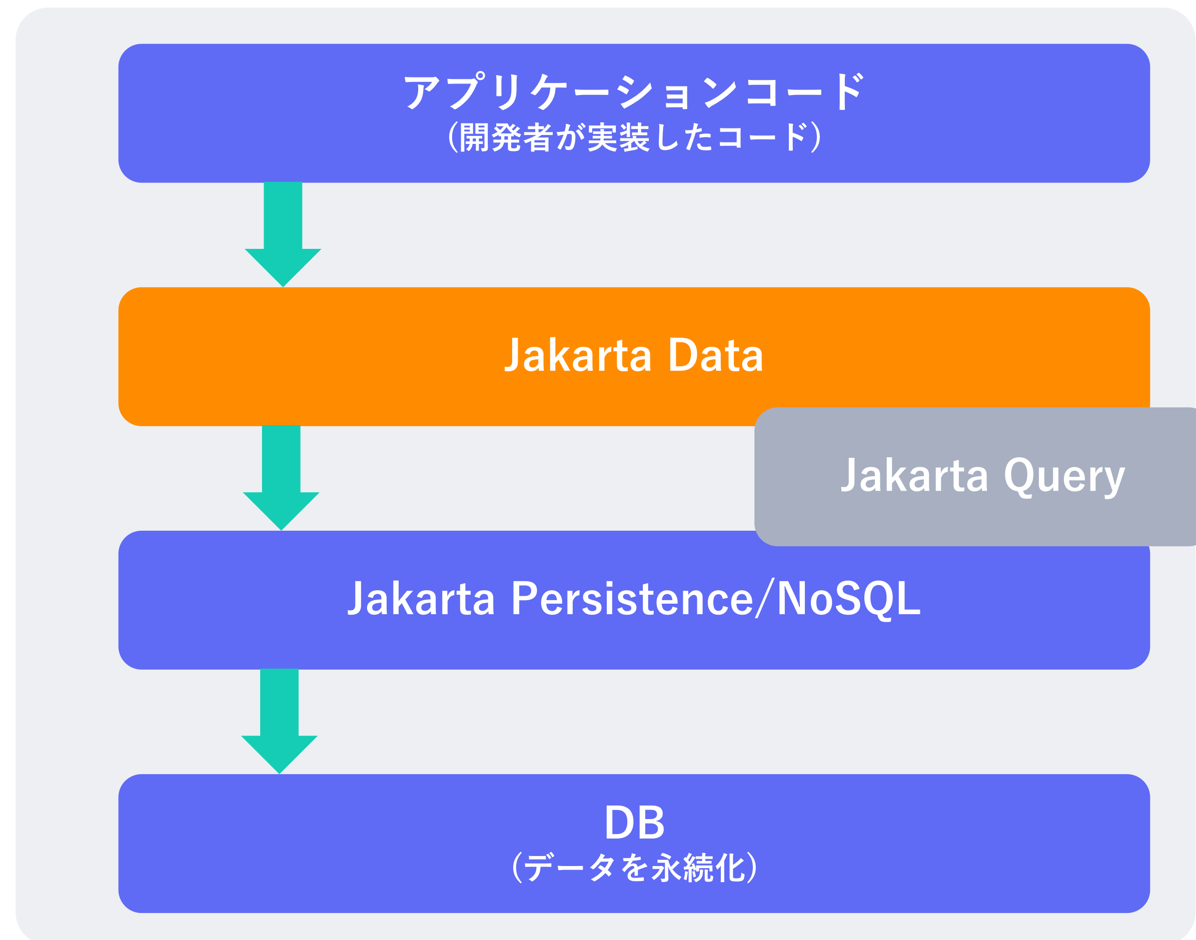
まとめ

© NEC Corporation 2026

まとめ

Jakarta Data は Jakarta Persistence/NoSQL などの仕様の上で動作
Jakarta Data / Query 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



© NEC Corporation 2026

ご清聴ありがとうございました。

© NEC Corporation 2026

NEC

\Orchestrating a brighter world