

# JAVA 17 & 21

## Evoluzione da Java 8

Scopri le trasformazioni che hanno rivoluzionato il linguaggio Java negli ultimi anni, dalle funzionalità di pattern matching ai thread virtuali.



## CONTESTO STORICO

# Da Java 8 a Java 21

Un viaggio di innovazione continua. Dal rilascio rivoluzionario di Java 8 nel 2014, il linguaggio ha accelerato la sua evoluzione con un nuovo modello di rilascio semestrale.



**2014**

**Java 8**

Lambda Expressions, Stream API. L'inizio della programmazione funzionale in Java.

**2018**

**Java 10**

Introduzione del ciclo di rilascio semestrale per un'innovazione più rapida.

**2021**

**Java 17 (LTS)**

Consolidamento di Sealed Classes, Pattern Matching e rimozione di API obsolete.

**2023**

**Java 21 (LTS)**

Virtual Threads, Record Patterns, Sequenced Collections. Il futuro della concorrenza.

# Pattern Matching per Switch

Elimina il boilerplate code, migliora la leggibilità e la sicurezza dei tipi (JEP 441).

## Prima (Java 8)

```
if(obj instanceof String){  
    String s = (String) obj;  
    System.out.println(s);  
} else if (obj instanceof Integer){  
    Integer i = (Integer) obj;  
    System.out.println(i);  
}
```

## Dopo (Java 21)

```
switch (obj) {  
    case String s -> System.out.println(s);  
    case Integer i -> System.out.println(i);  
    case null -> System.out.println("Null" );  
    default -> System.out.println("Other" );  
}
```

```
'key' or not self.active_index == 0:  
self.active_index -= 1  
self.mags_dt = self.get_act_color_map()  
self.menu_labels[self.active_index].color = [255, 255, 255]  
elif symbol == key.DOWN and not self.active_index == 0:  
self.active_index += 1  
self.mags_dt = self.get_act_color_map()  
elif symbol == key.ENTER:  
    if self.active_index == 3:  
        pyglet.app.exit()  
    else:  
        self.context_index = self.active_index  
elif symbol == key.ESCAPE:  
    if self.context_index == -1:  
        pyglet.app.exit()  
    else:  
        self.context_index = -1  
if self.context_index == 1:  
    if symbol == key.ESCAPE:  
        self.context_index = -1
```

**Meno Boilerplate:** Niente più casting espliciti.

**Null Handling:** Gestione diretta nei case.

**Guard Clauses** : Uso di when per condizioni extra.

# Sealed Classes

---

## Controllo dell'Ereditarietà

I Sealed Classes permettono di definire esplicitamente quali classi o interfacce possono estenderle.

Il controllo della gerarchia passa all'autore della classe.

```
public sealed interface Shape
permits Circle, Rectangle, Square {
// ...
}
```

```
public final class Circle
implements Shape { ... }
```

# Record Patterns

## Decostruzione Dichiarativa

Java 21 estende il pattern matching per decostruire istanze di record. Accedi ai componenti direttamente, senza getter espliciti.

Decostruzione annidata potente

Eliminazione di casting e getter

Codice focalizzato sui dati

```
record Point(int x, int y) {}

record Rect(Point p1, Point p2) {}

// Prima (Verboso)
if (obj instanceof Rect r) {
    Point p1 = r.p1();
    int x = p1.x();
    int y = p1.y();
    process(x, y);
}

// Dopo (Record Patterns)
if (obj instanceof Rect(Point(var x, var y), var p2)) {
    // x e y sono estratti automaticamente!
    process(x, y);
}
```

# Virtual Threads

JEP 444 - Scalabilità Massiva per la Concorrenza

## Platform Threads

**1 : 1**

Ogni thread Java è vincolato a un thread del Sistema Operativo.

**Limite:** ~10.000 thread

**Costo:** ~2MB stack per thread

**Problema:** Collo di bottiglia I/O

## Virtual Threads

**M : N**

Migliaia di thread virtuali condividono pochi thread del Sistema Operativo.

**Limite:** Milioni di thread

**Costo:** Pochi byte (heap)

**Soluzione:** Gestiti dalla JVM

# Virtual Threads

Esempi Pratici di Utilizzo

## Creazione Singola

```
// Creazione e avvio immediato
Thread. ofVirtual()
    .name("worker-1")
    .start(() -> {
        System.out.println("Hello Virtual!");
    });

// Oppure usando il builder
```

```
var builder = Thread.ofVirtual().name("v-thread", 1);
Thread t = builder.unstarted(task);
t.start();
```

L'API fluente `ofVirtual()` rende esplicita la creazione di thread leggeri, senza ambiguità rispetto ai thread di piattaforma.

## Executor Service (Scalabile)

```
// Un nuovo Executor per task virtuale
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {

    // Sottomissione di 10.000 task
    IntStream.range(0, 10_000).forEach(i -> {
        executor.submit(() -> {
            Thread.sleep(Duration.ofSeconds(1));
            return i;
        });
    });

} // L'executor si chiude e attende qui
```

Non serve più il pooling! Questo executor crea un **nuovo** virtual thread per **ogni** task sottomesso. Ideale per il paradigma "One thread per request".

# STRING TEMPLATES

## JEP 430 (Preview)

Un meccanismo moderno per comporre stringhe che combina testo letterale con espressioni incorporate, superando i limiti della concatenazione tradizionale.

### Interpolazione Sicura

Previene injection attacks validando le espressioni.

### Leggibilità Superiore

Elimina la confusione di + e StringBuilder .

### Template Processors

Usa STR per stringhe standard o crea processori custom (es. SQL, JSON).

Prima (Java 8+)

```
String msg = "Ciao " + name + ", oggi è " + day;  
  
// Oppure String.format()  
String msg = String.format("Ciao %s, oggi è %s" , name, day);
```

Java 21

```
String name = "Mario" ;  
var day = LocalDate.now();  
  
// Sintassi pulita e diretta  
String msg = STR. "Ciao \{name}, oggi è \{day}" ;
```

# VECTOR API

## Parallelismo SIMD

Sfrutta le istruzioni **SIMD** (Single Instruction, Multiple Data) delle moderne CPU per eseguire operazioni su array di dati in parallelo, superando i limiti del calcolo scalare.

### Casi d'uso ideali

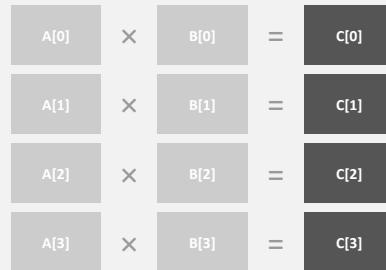
Elaborazione Immagini / Audio

Machine Learning & AI

Calcolo Scientifico

Crittografia

### Calcolo Scalare (Tradizionale)



4 Cicli CPU

### Calcolo Vettoriale (SIMD)



×



=



1 Ciclo CPU

# Vector API

SIMD in Azione: Scalare vs Vettoriale (JEP 414)

## Approccio Scalare

```
// Moltiplicazione elemento per elemento
for (int i = 0; i < a.length; i++) {
    // Una operazione per ciclo CPU
    c[i] = a[i] * b[i];
}
```

1x Speed

## Approccio Vector API

```
// Definizione della specie (es. 256-bit)
var SPECIES = IntVector.SPECIES_PREFERRED;

// Loop con passo "lunghezza vettore"
for (int i = 0; i < a.length; i += SPECIES.length()) {
    // Carica interi in parallelo
    var va = IntVector.fromArray(SPECIES, a, i);
    var vb = IntVector.fromArray(SPECIES, b, i);
    // Moltiplica in un ciclo
    var vc = va.mul(vb);
    // Salva il risultato
    vc.intoArray(c, i);
}
```

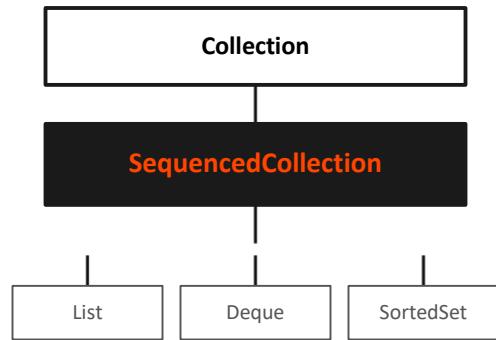
4x-16x Speed

Il processore esegue una singola moltiplicazione per ogni iterazione: semplice, senza parallelismo.

Usa registri vettoriali per elaborare più elementi in una singola istruzione CPU.

# Sequenced Collections

JEP 431



Una interfaccia che definisce l'ordine di incontro degli elementi e uniforma l'accesso a testa e coda.

## Unified API

<code>addFirst(E e)</code>	Aggiunge in testa
<code>addLast(E e)</code>	Aggiunge in coda
<code>getFirst()</code>	Legge la testa
<code>getLast()</code>	Legge la coda
<code>reversed()</code>	Vista inversa

# Gerarchia Sequenced

Integrazione nel Framework Collections

## Sequenced Collection

**SequencedCollection**

List

Deque

Implementazioni

**ArrayList**

**LinkedList**

**ArrayDeque**

## Sequenced Set

**SequencedSet**

SortedSet

NavigableSet

Implementazioni

**LinkedHashSet**

**TreeSet**

**ConcurrentSkipListSet**

## Sequenced Map

**SequencedMap**

SortedMap

NavigableMap

Implementazioni

**LinkedHashMap**

**TreeMap**

**ConcurrentSkipListMap**

# Sequenced Collections

Uniformità e Chiarezza (JEP 431)

## Java 8 (Inconsistente)

Legacy

Accesso Primo/Ultimo

```
// List  
var first = list.get(0);  
var last = list.get(list.size() - 1);  
  
// Deque/SortedSet  
deque.getFirst(); // similar  
set.first();
```

Iterazione Inversa

```
// Modifica la lista!  
Collections.reverse(list);  
  
// Oppure poco leggibile  
Iterator<E> it = list.listIterator(list.size());  
while (it.hasPrevious()) { ... }
```

## Java 21 (Uniforme)

Modern

Accesso Primo/Ultimo

```
// Unico API per varie collezioni  
var first = collection.getFirst();  
var last = collection.getLast();  
  
list.addFirst("Start"); list.addLast("End");
```

Iterazione Inversa

```
// Vista efficiente, originale intatto  
for (var item : list.reversed()) {  
    System.out.println(item);  
}  
  
// Funziona anche con stream  
list.reversed().stream()...
```

# Performance Migliorate

## Ottimizzazioni JVM HotSpot

### Class Hierarchy Analysis (CHA)

Nuova implementazione che gestisce meglio metodi astratti e default, ottimizzando il dispatching.

### Escape Analysis Aggressiva

Migliore capacità di allocare oggetti sullo stack invece che nell'heap, riducendo la pressione sul GC.

### Vectorization Migliorata

Il compilatore JIT sfrutta meglio le istruzioni SIMD delle CPU moderne per calcoli paralleli.

Il codice Java 8 esistente gira significativamente più veloce su Java 17/21 senza alcuna modifica.



```
i = modifier_ob.modifiers.new("object to mirror_ob
i.mirror_object = mirror_ob
== "MIRROR_X":
i.use_x = True
i.use_y = False
i.use_z = False
n == "MIRROR_Y":
i.use_x = False
i.use_y = True
i.use_z = False
n == "MIRROR_Z":
i.use_x = False
i.use_y = False
i.use_z = True

at the end -add back the deselected= 1
selected=1
seen.objects.active = modifier_ob + str(modifier_ob)) # modified selected = 0
cts[0]
ct = 1
two objects,
```

FREE SPEEDUP

# Pulizia del Linguaggio

Rimozione di API obsolete per mantenere Java moderno, sicuro e performante.

## RMI Activation

Rimosso

Meccanismo obsoleto per l'attivazione di oggetti remoti.

## Experimental AOT & JIT

Rimosso

Compilatori GraalVM rimossi dal JDK principale.

## Security Manager

Deprecato

Deprecato: sostituito da meccanismi più moderni.

## Applet API

Deprecato

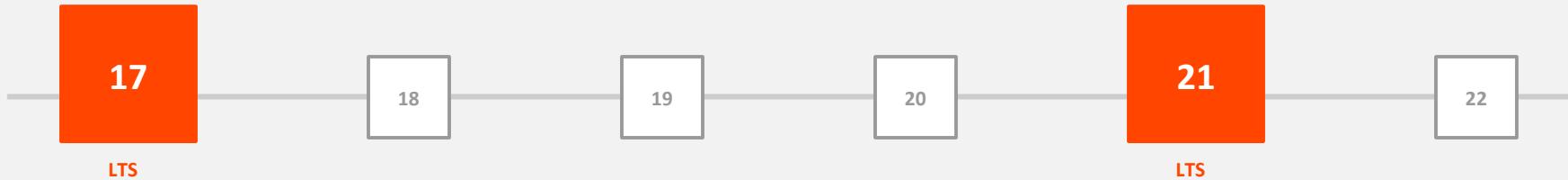
Tecnologia legacy non più supportata dai browser.

---

*"La rimozione di funzionalità obsolete riduce il debito tecnico e semplifica l'evoluzione futura della piattaforma."*

# Modello di Rilascio

Equilibrio tra innovazione rapida e stabilità a lungo termine.



## Feature Releases (6 Mesi)

Rilasciate ogni marzo e settembre. Introducono nuove funzionalità rapidamente, permettendo un feedback immediato tramite meccanismi di **Preview** e **Incubator**. Ideali per testare l'innovazione.

## LTS Releases (2-3 Anni)

Versioni consolidate (come Java 17 e 21) con supporto esteso (fino a 8 anni da Oracle). Garantiscono la **stabilità** necessaria per le applicazioni enterprise critiche, accumulando le feature maturate nei rilasci intermedi.

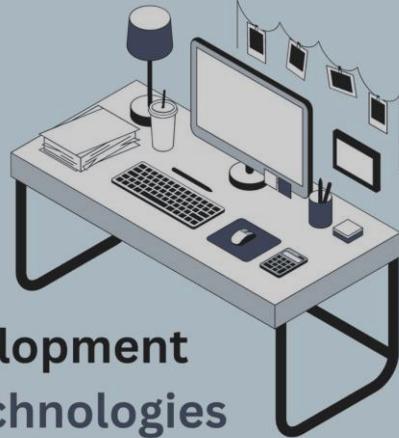
# IL FUTURO DI JAVA

## Innovazione & Stabilità

Da Java 8 a Java 21, il linguaggio ha completato una trasformazione radicale. Con il modello di rilascio semestrale e il supporto LTS, Java offre oggi il perfetto equilibrio tra funzionalità moderne e affidabilità enterprise.

- Sintassi Espressiva
- Concorrenza Scalabile
- Sicurezza By Design

The Future  
of Software Development  
and Emerging Technologies





# Criteria Query in Spring Data JPA

Costruire Query Type-Safe e Dinamiche

Esempio: Dynamic & Type-Safe Query

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<User> cq = cb.createQuery(User.class);
Root<User> root = cq.from(User.class);

List<Predicate> predicates = new ArrayList<>();

if (searchDto.getName() != null) {
    predicates.add(cb.like(root.get("name"),
        "%" + searchDto.getName() + "%"));
}
if (searchDto.getAge() != null) {
    predicates.add(cb.equal(root.get("age"), searchDto.getAge()));
}
if (searchDto.getStatus() != null) {
    predicates.add(cb.equal(root.get("status"),
        searchDto.getStatus()));
}

cq.where(predicates.toArray(new Predicate[0]));

TypedQuery<User> query = entityManager.createQuery(cq);
List<User> results = query.getResultList();
```

Query Equivalente in JPQL: SELECT U FROM User U WHERE U.name LIKE :name  
AND u.age = :age AND u.status = :status

# Cosa sono le Criteria Query?

Query Programmatiche Type-Safe per il Tuo Database

- **Approccio Programmatico:** Costruisci query usando oggetti Java invece di stringhe SQL/JPQL.
- **Type-Safety Garantita:** Il compilatore verifica i tipi e i nomi dei campi, prevenendo errori di sintassi a runtime.
- **Flessibilità Dinamica:** Ideale per costruire filtri complessi che cambiano in base all'input dell'utente.



# Criteria Query vs JPQL vs Query Methods

Tre Approcci a Confronto

Aspetto	Query Methods	JPQL	Criteria Query
Type-Safe	Parziale	No	Sì
Dinamico	No	Sì	Sì
Leggibilità	Ottima	Buona	Complessa
Boilerplate	Minimo	Minimo	Massiccio
Flessibilità	Bassa	Media	Alta



# Componenti Fondamentali

Gli Elementi Chiave di una Criteria Query



## CriteriaBuilder

Il costruttore principale (Factory) per creare query e prediciati.



## CriteriaQuery<T>

Definisce la struttura della query e il tipo di risultato.



## Root<T>

Il punto di partenza (FROM), rappresenta l'entità radice.



## Predicate

Le condizioni di filtro (WHERE) per restringere i risultati.



# Flusso di Esecuzione

Step-by-Step: Come Funziona una Criteria Query

**CriteriaBuilder**

```
em.getCriteriaBuilder()
```

**CriteriaQuery**

```
cb.createQuery(Book.class)
```

**Root**

```
cq.from(Book.class)
```

**Predicate**

```
cb.equal(...)
```

**Where**

```
cq.where(predicate)
```

**TypedQuery**

```
em.createQuery(cq)
```

**Result**

```
query.getResultList()
```

# Esempio Pratico Base

## Trovare Tutti i Libri di un Autore

```
// Setup
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Libro> cq = cb.createQuery(Libro.class);
Root<Libro> libro = cq.from(Libro.class);

// Creare il predicato
Predicate p = cb.equal(
    libro.get("autore"),
    "J.R.R. Tolkien"
);

// Applicare e Eseguire
cq.where(p);
List<Libro> results = em.createQuery(cq).getResultList();
```

# Operatori di Confronto

Predicati Comuni per Filtrare i Dati

## Uguaglianza

```
cb.equal(path, value)
```

Trova valori esatti.

## Pattern Matching

```
cb.like(path, "%val%")
```

Ricerca parziale (SQL LIKE).

## Confronto

```
cb.greaterThan(path, val)
```

Maggiore di un valore specifico.

## Range

```
cb.between(path, min, max)
```

Valori compresi in un intervallo.

## Null Check

```
cb.isNull(path)
```

Verifica se il campo è nullo.

## Collezioni

```
cb.in(path)
```

Verifica appartenenza a una lista.

# Combinare Predicati Logici

## AND, OR e NOT per Query Complesse

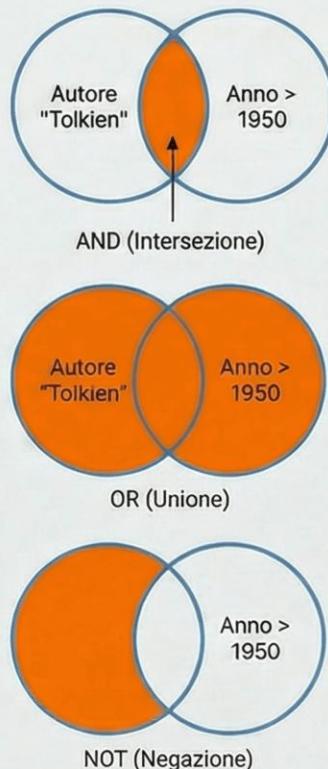
```
Predicate autore = cb.equal(libro.get("autore"), "Tolkien");
Predicate anno = cb.gt(libro.get("anno"), 1950);

// AND: Tutti veri
Predicate and = cb.and(autore, anno);

// OR: Almeno uno vero
Predicate or = cb.or(autore, anno);

// NOT: Negazione
Predicate not = cb.not(anno);

cq.where(and);
```



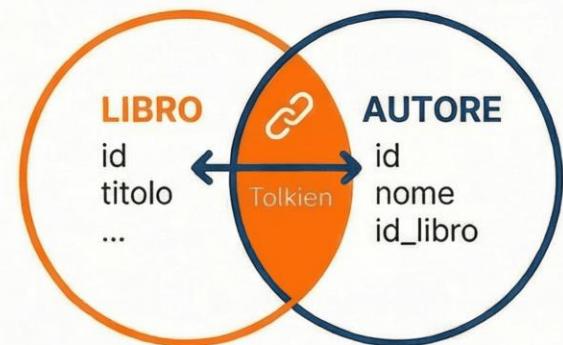
# Join e Relazioni

## Navigare le Relazioni tra Entità

```
// Setup
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Libro> cq = cb.createQuery(Libro.class);
Root<Libro> libro = cq.from(Libro.class);

// Creare un join con l'entità Author
Join<Libro, Autore> autore = libro.join("autore");

// Filtrare usando l'entità joinata
Predicate predicate = cb.equal(autore.get("nome"), "Tolkien");
cq.where(predicate);
```



# Ordinamento e Paginazione

## Ordinare i Risultati e Gestire Grandi Dataset

### Ordinamento:

- Usa `cb.asc()` e `cb.desc()` per definire l'ordine dei risultati.

### Paginazione:

- Gestita a livello di `TypedQuery`, non nel `CriteriaBuilder`.
  - `setFirstResult(int)`: Definisce l'offset (da dove iniziare).
  - `setMaxResults(int)`: Definisce il limit (quanti record recuperare).

```
// Ordinamento
cq.orderBy(
    cb.asc(libro.get("titolo")),
    cb.desc(libro.get("anno"))
);

// Paginazione
TypedQuery<Libro> query = em.createQuery(cq);
query.setFirstResult(0); // Offset
query.setMaxResults(10); // Limit
```

# Aggregazioni

## Conteggi, Somme e Statistiche

### Conteggio (Count)

```
// Contare i libri  
  
CriteriaQuery<Long> cq = cb.createQuery(Long.class);  
  
CriteriaQuery<Long> cq = cb.createQuery(Long.class);  
cq.select(cb.count(libro));
```

### Somma (Sum)

```
// Somma dei prezzi  
  
CriteriaQuery<Double> sq = cb.createQuery(Double.class);  
  
CriteriaQuery<Double> sq = cb.createQuery(Double.class);  
sq.select(cb.sum(libro.get("prezzo")));
```

**avg(), min(), max()**

# Query Dinamiche

Costruire Predicati Condizionali a Runtime

```
List<Predicate> predicati = new ArrayList<>();  
  
// Aggiungi solo se il parametro esiste  
if (autore != null) {  
    predicati.add(cb.equal(libro.get("autore"), autore));  
}  
if (titolo != null) {  
    predicati.add(cb.like(libro.get("titolo"), "%" + titolo + "%"));  
}  
  
// Applica tutti i predicati  
cq.where(predicati.toArray(new Predicate[0]));
```



# Vantaggi delle Criteria Query

Perché Usarle nel Tuo Progetto



## Type-Safety

Errori rilevati a compile-time, non a runtime.



## Dinamicità

Costruzione query a runtime basata su input.



## Riutilizzabilità

Componenti modulari con Specifications.



## Manutenibilità

Refactoring sicuro e immediato.



## Performance

Compilazione in SQL ottimizzato.

# Svantaggi e Limitazioni

## Quando Considerare Alternative



### Verbosità

Codice molto più lungo e complesso rispetto a JPQL.



### Curva di Apprendimento

Richiede comprensione profonda delle API e dei componenti.



### Leggibilità

Meno immediato da leggere rispetto a una semplice stringa SQL.



### Limitazioni Funzionali

Supporto limitato per grouping complessi e alcune subquery.



### Overhead di Sviluppo

Eccessivo per query semplici (es. findAll).

# Best Practices

## Consigli per Usare le Criteria Query Efficacemente

- ✓ **Usa Specifications:** Incapsula i predicati per massima modularità e riutilizzo.
- ✓ **Keep It Simple:** Usa i Query Methods per query semplici, non complicare inutilmente.
- ✓ **Documentazione:** Commenta sempre la logica delle query complesse per facilitare la manutenzione.
- ✓ **Testing:** Testa rigorosamente le query dinamiche con dati reali per coprire tutti i casi.

# Conclusion

## Criteria Query: Uno Strumento Potente nel Tuo Arsenal



- **Equilibrio:** Il giusto mix tra la semplicità dei Query Methods e la potenza di JPQL.
- **Use Case:** La scelta migliore per query dinamiche, type-safe e riutilizzabili.
- **Strategia:** Sfrutta le Specifications per un codice pulito e modulare.

Con la pratica, le Criteria Query diventeranno uno strumento indispensabile nel tuo toolkit Spring.