Future-Proofing Your Data Access Layer: A Guide to Database-Agnostic Queries

This tutorial explains how to convert a native Oracle SQL query into a database-agnostic solution using Java and the Spring Boot framework.

By moving database-specific logic into the application layer, you can create a more portable and maintainable system.

This approach ensures that future database migrations—whether from Oracle to PostgreSQL, or from PostgreSQL to something else—require minimal changes to your application code.

The Goal: True Database Independence

When an application uses native SQL, it becomes tightly coupled to a specific database vendor. Migrating to a new database can trigger a costly and time-consuming effort to rewrite every query.

The solution is to use an abstraction layer. In the Java world, the **Java Persistence API (JPA)** is the standard. By writing queries in **JPQL (Java Persistence Query Language)** or by moving complex logic into your Java code, you let your JPA provider (like Hibernate) handle the translation to the specific SQL dialect of whatever database you are connected to.

This tutorial will demonstrate this process by refactoring the "Long Tenure Low Salary" dashboard.

Case Study: Refactoring the 'Long Tenure' Dashboard

Let’s start by analyzing the original Oracle query.

Step 1: Analyze the Original Oracle Query

The original query is highly efficient but uses two Oracle-specific functions that make it non-portable:

```
WITH JobMedian AS (
    SELECT
        job_id,
        MEDIAN(salary) AS median_salary --
    FROM employees
    GROUP BY job_id
)
```

```
SELECT
    e.employee_id,
    e.first_name || ' ' || e.last_name AS employee_name,
    j.job_title,
    e.salary,
    jm.median_salary,
    ROUND(MONTHS_BETWEEN(SYSDATE, e.hire_date) / 12, 2) AS years_of_service --
FROM employees e
JOIN jobs j ON e.job_id = j.job_id
JOIN JobMedian jm ON e.job_id = jm.job_id
WHERE MONTHS_BETWEEN(SYSDATE, e.hire_date) / 12 > 5 --
AND e.salary <= jm.median_salary
ORDER BY years_of_service DESC, salary;
```

`MEDIAN` is an Oracle-specific aggregate function. It is not part of the SQL standard and is not available in many other databases, including PostgreSQL, without a special function definition.

`MONTHS_BETWEEN` is also an Oracle-specific function. Other databases use different functions for date arithmetic (e.g., `AGE` in PostgreSQL).

A direct conversion of this query to JPQL is not possible because these functions are not part of the JPA specification. The solution is to move this business logic from the database into the application's service layer.

Step 2: Create a New Service Layer for Analytics

It's good practice to separate business logic from the controller. Let's create a new `AnalyticsService` to handle the logic for our dashboard.

Create a new file named `AnalyticsService.java` in a new `service` package: `src/main/java/com/example/hr/backend_springboot/service/`

```
package com.example.hr.backend_springboot.service;

import com.example.hr.backend_springboot.dto.LongTenureLowSalaryDTO;
import com.example.hr.backend_springboot.model.Employee;
import com.example.hr.backend_springboot.model.Job;
import com.example.hr.backend_springboot.repository.EmployeeRepository;
import com.example.hr.backend_springboot.repository.JobRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.math.BigDecimal;
import java.time.LocalDate;
import java.time.Period;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Map;
```

```java
import java.util.stream.Collectors;

@Service
public class AnalyticsService {

    @Autowired
    private EmployeeRepository employeeRepository;

    @Autowired
    private JobRepository jobRepository;

    public List<LongTenureLowSalaryDTO> getLongTenureLowSalaryEmployees() {
        // 1. Calculate median salary for each job in Java
        Map<Job, Double> medianSalariesByJob = calculateMedianSalaries();

        // 2. Fetch all employees using a simple, portable query
        List<Employee> allEmployees = employeeRepository.findAll();

        // 3. Filter employees in the service layer
        List<LongTenureLowSalaryDTO> result = new ArrayList<>();
        for (Employee e : allEmployees) {
            if (e.getJob() == null) continue;

            // Calculate years of service in Java
            double yearsOfService = Period.between(e.getHireDate(), LocalDate.now
()).toTotalMonths() / 12.0;

            if (yearsOfService > 5) {
                double medianSalary = medianSalariesByJob.getOrDefault(e.getJob(),
0.0);
                if (e.getSalary().doubleValue() <= medianSalary) {
                    result.add(new LongTenureLowSalaryDTO(
                        e.getEmployeeId(),
                        e.getFirstName() + " " + e.getLastName(),
                        e.getJob().getJobTitle(),
                        e.getSalary(),
                        new BigDecimal(medianSalary),
                        new BigDecimal(String.format("%.2f", yearsOfService))
                    ));
                }
            }
        }

        result.sort((a, b) -> b.getYearsOfService().compareTo(a.getYearsOfService()));
        return result;
    }

    private Map<Job, Double> calculateMedianSalaries() {
        List<Employee> employees = employeeRepository.findAll();
        Map<Job, List<Employee>> employeesByJob = employees.stream()
                .filter(e -> e.getJob() != null)
```

```
                .collect(Collectors.groupingBy(Employee::getJob));

        return employeesByJob.entrySet().stream()
                .collect(Collectors.toMap(
                        Map.Entry::getKey,
                        entry -> calculateMedian(entry.getValue())
                ));
    }

    private double calculateMedian(List<Employee> employees) {
        if (employees.isEmpty()) return 0.0;

        List<BigDecimal> salaries = employees.stream()
                .map(Employee::getSalary)
                .sorted()
                .collect(Collectors.toList());

        int middle = salaries.size() / 2;
        if (salaries.size() % 2 == 1) {
            return salaries.get(middle).doubleValue();
        } else {
            return (salaries.get(middle - 1).doubleValue() + salaries.get(middle
).doubleValue()) / 2.0;
        }
    }
}
```

Step 3: Update the Analytics Controller

Now, update the `AnalyticsController` to use the new `AnalyticsService`. The controller's job is simplified to just handling the web request and passing data to the view.

```
package com.example.hr.backend_springboot.controller;

import com.example.hr.backend_springboot.service.AnalyticsService;
// ... other imports

@Controller
public class AnalyticsController {

    @Autowired
    private EmployeeRepository employeeRepository;

    @Autowired
    private AnalyticsService analyticsService; //

    // ... (other existing GET mappings)

    @GetMapping("/analytics/long-tenure-low-salary")
```

```
    public String longTenureLowSalary(Model model) {
        //
        model.addAttribute("tenureData", analyticsService
.getLongTenureLowSalaryEmployees());
        return "analytics/long-tenure-low-salary";
    }
}
```

Inject the new `AnalyticsService`. Call the service method to get the data. All the complex logic is now handled by the service.

Step 4: The Result - A Truly Portable Solution

With this refactoring:

- **No Native Queries**: The `EmployeeRepository` no longer needs a complex, native SQL query for this dashboard. It can rely on the standard `findAll()` method provided by `JpaRepository`.
- **Business Logic in Java**: The logic for calculating "years of service" and finding the "median salary" now resides in the `AnalyticsService`. This Java code is completely database-agnostic.
- **Easy Migration**: To migrate from Oracle to PostgreSQL, you only need to update your `pom.xml` with the PostgreSQL driver and your `application.properties` with the new connection string and Hibernate dialect. **No changes are needed in the repository, service, or controller for this dashboard.**

This approach of abstracting database-specific logic into the application layer is a powerful strategy for building robust, maintainable, and "future-proof" applications.

Q&A: Refactoring the Backend

Do I need to update the HTML view after refactoring the backend to be database-agnostic?

No. You do not need to update the HTML file at all.

The reason the view remains unchanged is due to the strong **separation of concerns** provided by the Model-View-Controller (MVC) architecture and the use of a Data Transfer Object (DTO).

1. **The Contract with the View**: The `AnalyticsController` has a "contract" with the `long-tenure-low-salary.html` view. This contract promises that the controller will provide a piece of data to the view named `"tenureData"`. It also guarantees that this data will be a list of objects, where each object has specific fields the view can access (e.g., `employeeName`, `jobTitle`, `salary`).
2. **A Consistent DTO**: In both the original native query implementation and the new database-agnostic service implementation, the final object passed to the view is a `List<LongTenureLowSalaryDTO>`. Because the structure of this DTO remains the same, the view has a consistent and predictable data structure to work with.
3. **Backend Changes are an Implementation Detail**: The Thymeleaf template in the HTML file only cares about the final `"tenureData"` object it receives. It has no knowledge of **how** that data

was created. Whether the list was generated from a native Oracle query or from complex logic in a Java service is completely irrelevant to the view.

This is a major benefit of this architectural pattern. The presentation layer (the HTML) is successfully decoupled from the data access layer. The extensive refactoring was performed entirely in the backend, and the frontend remained completely unaware and unaffected.

Q&A: Deeper Dive into Design Choices

Why use a DTO? Why not just return the `Employee` entity directly from the query?

That's a great design question. While you **could** return the `Employee` entity, using a Data Transfer Object (DTO) provides several key advantages:

- **Decoupling**: The DTO decouples your internal data model (the `Employee` entity) from the data structure your view needs. If you later add a field to the `Employee` entity that the view doesn't need (like an internal audit timestamp), you don't have to worry about it being accidentally exposed.
- **Efficiency**: Your queries can be written to select **only** the data needed for the DTO. This is more efficient than fetching the entire `Employee` entity, especially if the entity has many fields or lazy-loaded relationships that you don't need for a particular view.
- **Customization**: DTOs allow you to create custom, view-specific data structures. Your `LongTenureLowSalaryDTO` is a perfect example; it combines fields from the `Employee` and `Job` entities, along with a calculated value (`yearsOfService`), into a single clean object.

Isn't performing the median and date calculations in Java slower than in the database?

This is a critical trade-off to consider. The short answer is **yes, it can be slower**, but the practical impact depends on the scale of your data.

- **For Small to Medium Data Sets**: For hundreds or even thousands of employees, the performance difference will likely be negligible (milliseconds). The Java code will execute very quickly, and the benefit of having portable, maintainable code far outweighs the minor performance cost.
- **For Very Large Data Sets**: If you were processing hundreds of thousands or millions of records, performing these calculations in the application layer could become a bottleneck. In such a scenario, the database, which is highly optimized for set-based operations, would be significantly faster.
- **The Hybrid Approach**: For high-performance needs, you could adopt a hybrid strategy as discussed earlier: use database-agnostic JPQL for most of your application and isolate the few, performance-critical queries into database-specific implementations using Spring Profiles.

If native queries are not portable, is there ever a good reason to use them?

Absolutely. While you should default to JPQL or the Criteria API for portability, native queries are the right tool for specific jobs:

- **Performance-Critical Operations**: When you need to squeeze every last drop of performance out of a query and know that a specific database feature (like an Oracle index hint) is required.
- **Database-Specific Features**: For accessing features that are not supported by the JPA standard, such as Oracle's hierarchical queries (`CONNECT BY`) or specific data types.
- **Complex Reporting**: For very complex analytical or reporting queries that are difficult to express in JPQL and are easier to write and maintain in native SQL. The key is to use them judiciously and, when you do, to be aware that you are creating a tight coupling to your current database.

How would we handle a truly complex query, like Oracle's `CONNECT BY`, in a portable way?

This is where the abstraction strategy is truly tested. A direct JPQL equivalent for `CONNECT BY` does not exist because hierarchical queries are implemented very differently across databases (e.g., PostgreSQL uses `WITH RECURSIVE`).

You have two primary options:

1. **Query in the Application Layer**: Fetch the parent-child relationships with a simple, portable query (e.g., `SELECT e FROM Employee e`). Then, build the hierarchy in your Java `AnalyticsService`. This is feasible for moderately sized hierarchies but can be memory-intensive for very deep or wide organizational charts.
2. **Isolate with Spring Profiles**: This is often the best approach for complex cases. You would create two separate repository implementations for the hierarchical query—one for Oracle using `CONNECT BY` and one for PostgreSQL using `WITH RECURSIVE`. By annotating them with `@Profile("oracle")` and `@Profile("postgres")`, you can tell Spring to inject the correct implementation based on the active profile, keeping the database-specific code isolated and the rest of your application portable.

Q&A: Oracle to PostgreSQL Migration Details

What happens to Oracle sequences like `EMPLOYEES_SEQ` when migrating to PostgreSQL?

This is a great question, as it touches on a key difference in how auto-incrementing IDs are handled. Fortunately, JPA and Hibernate abstract this away almost entirely.

- **In Oracle**, IDs are typically generated by calling `NEXTVAL` on a sequence object (e.g., `EMPLOYEES_SEQ.NEXTVAL`).
- **In PostgreSQL**, the idiomatic way is to use `SERIAL` or `BIGSERIAL` data types for primary key columns, which automatically creates and manages a sequence behind the scenes.

Because your `Employee` entity is already configured with the standard JPA annotations, you are well-protected:

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "employees_seq")
@SequenceGenerator(name = "employees_seq", sequenceName = "EMPLOYEES_SEQ",
allocationSize = 1)
private Long employeeId;
```

When you migrate to PostgreSQL, **you don't have to change this code**. Hibernate's `OracleDialect` knows how to use `EMPLOYEES_SEQ.NEXTVAL`, and its `PostgreSQLDialect` knows how to correctly get the next value from the sequence that backs a `SERIAL` column. As long as your PostgreSQL schema is set up with `SERIAL` or `BIGSERIAL` columns for your primary keys, the transition is seamless from the application's perspective.

Do I need to worry about data type differences, like Oracle's `NUMBER` vs. PostgreSQL's `NUMERIC` or `INTEGER`?

For the most part, no. This is another major benefit of using an abstraction layer like JPA/Hibernate.

In your Java entities, you've mapped database numbers to Java's `Long` and `BigDecimal` types.

- `private Long employeeId;`
- `private BigDecimal salary;`

Hibernate's database dialect handles the mapping between these Java types and the appropriate native SQL types.

- When connected to **Oracle**, Hibernate knows to map `BigDecimal` to `NUMBER`.
- When you switch to **PostgreSQL**, the dialect will correctly map `BigDecimal` to `NUMERIC`, which is the appropriate type for arbitrary-precision decimals in PostgreSQL.

You would only run into issues if you were using a very obscure, vendor-specific data type in your database that doesn't have a standard mapping in Hibernate. For common types like numbers, strings, and dates, the abstraction works very well.

What if our application used Oracle-specific PL/SQL stored procedures?

This is one of the most challenging aspects of any database migration. Unlike the queries within your application, stored procedures written in PL/SQL are completely non-portable and would need to be rewritten.

The PostgreSQL equivalent of PL/SQL is **PL/pgSQL**. While they are both procedural languages for SQL, their syntax, error handling, and feature sets are quite different.

If your application relied on stored procedures, the migration strategy would involve:

1. **Identifying all PL/SQL procedures**: You would first need to inventory every procedure, function, and trigger used by the application.

2. **Rewriting in PL/pgSQL**: Each procedure would have to be manually translated from PL/SQL to PL/pgSQL. This is often a complex task.
3. **Considering a "Lift and Shift"**: An alternative is to move the logic from the stored procedures **out** of the database and into your Java-based `service` layer. This is often the preferred approach in modern application design, as it keeps the business logic within the application itself, making the application less dependent on the database and easier to test.

How is something like pagination handled during the migration?

Pagination is a perfect example of why using an abstraction layer is so beneficial. The SQL syntax for limiting results and skipping rows is famously different across databases.

- **Oracle (Older Versions)**: Used a `ROWNUM` pseudo-column in a subquery, which was complex.
- **Oracle (12c and later)**: Adopted the more standard `OFFSET ⋯ FETCH NEXT ⋯ ROWS ONLY` syntax.
- **PostgreSQL**: Uses the `LIMIT ⋯ OFFSET ⋯` syntax.

In your application, you are using Spring Data JPA's `Pageable` object in your `EmployeeController`. When you call `employeeRepository.findAll(pageable)`, you are telling Spring, "I want a specific page of data."

Spring Data JPA and Hibernate take care of the rest. The dialect will detect which database you are connected to and generate the correct, optimized SQL for pagination for that specific database. You don't have to write any pagination-specific SQL yourself, and your controller code remains completely portable across Oracle, PostgreSQL, and any other supported database.