Why EntityManager Instead of Repository @Query?

1. DTO Projections with Constructor Expressions

The queries use

```
SELECT new SomeDto(...)
```

syntax to directly construct DTOs:

```java
SELECT new com.employeeinsurancemanagement.dto.ClaimReportDto(
    e.enrollmentId, c.claimId, ...
)
```

While Spring Data JPA *can* do this with

```
@Query
```

, using

```
EntityManager
```

gives you more control and flexibility.

2. Dynamic Parameter Handling

Look at the
getClaimSummaryByEnrollment method:

```java
if (status != null && !status.equals("ALL")) {
    q.setParameter("status", statusEnum);
} else {
    q.setParameter("status", null);
}
```

This dynamic runtime logic (parsing strings to enums, handling "ALL" as null) is **harder to express cleanly** in a repository

```
@Query
```

.

3. Complex Aggregations

The queries use

```
GROUP BY
```

,

```
LEFT JOIN
```

,

```
COALESCE
```

, and aggregate functions (

`COUNT`

,

`SUM`

). While possible in

`@Query`

, these are often cleaner with programmatic JPQL via EntityManager.

When You SHOULD Use Repository @Query

- Simple CRUD operations
- Queries that don't need runtime logic
- When you don't need constructor expressions

When EntityManager Makes Sense

- Complex reporting queries with dynamic filters
- DTO projections with constructor expressions
- When you need programmatic control over query building

Bottom Line: Your current approach is **not wrong**—it's a valid design choice for reporting queries. If you wanted to standardize on repositories, you could move these queries to a dedicated

`ReportRepository`

with native or JPQL

`@Query`

methods, but the current implementation works and is maintainable.