**✅ Complete Example: Static Projection in Spring Data JPA**

### ✅ What is Projection?

* Normally, when you fetch data using a repository method, it returns the **full entity** (all columns of the table).
* But sometimes, you don’t want all columns — you only need **specific columns**.  
  👉 This is where **Projection** comes in.

# 🔹 What is Static Projection?

👉 Normally, when you call a repository method, it returns the **full Entity** (all columns).  
But sometimes, we don’t need all the columns → maybe just name and salary.  
**Projection** is used to fetch **only the required columns** instead of the whole entity.

This improves **performance** and reduces **unnecessary data transfer**.

# 🔹 Static Projection (Compile-Time, Fixed Structure)

* In **Static Projection**, we define an **Interface** (or class) in advance → with **getter methods** matching the required columns.
* Spring Data JPA will generate a **proxy class** behind the scenes that **implements this interface** and holds only those selected values.
* Because the interface is **predefined and fixed**, it’s called **static projection**.

**🔹 How It Works Internally**

In case of **Static Projection**, Spring Data JPA creates **2 Proxy classes**:

1. **Proxy for the Projection Interface (EmployeeView)** → holds only the selected columns (like name, company).
2. **Proxy for the Repository (EmployeeRepository)** → the usual Spring-generated repository proxy.

Together, they make the projection work **without writing extra code**.

**✅ Key Points to Remember**

* **Static Projection** → You predefine an interface with fixed columns.
* **Spring JPA** creates a **proxy class** for that interface to hold results.
* It helps in **performance optimization** by reducing unnecessary data transfer.
* Mostly used in **real-time dashboards, reports, or API responses** where full entity data is not needed.

Eg: SpringBootDataJpaStaticProjection

### ✅ Static vs Dynamic Projection

1. **Static Projection**
   * You already decide in advance which columns you want.
   * Example: Always return only name and salary.
2. **Dynamic Projection**
   * You don’t fix in advance which columns to fetch.
   * You can **dynamically decide** (at runtime) which columns you want.
   * Example: Sometimes you need only name, sometimes you need name and company.

### ✅ How Dynamic Projection Works?

* You write a **finder method** in your repository (like findByCountry(...)).
* But instead of returning a fixed entity or fixed DTO, you return a **generic type (T)**.
* Based on the interface you pass while calling, Spring Data JPA will automatically give you only those columns.

Eg: SpringDataJpaDynamicProjection

# 🔹 Static vs Dynamic Projection (with Code Snippets)

| **Aspect** | **Static Projection** | **Dynamic Projection** |
| --- | --- | --- |
| **Repository Method** | java List<NameView> findByCountry(String country);  👉 Return type is fixed (NameView). | java <T> List<T> findByCountry(String country, Class<T> type);  👉 Generic return type (<T>), flexible. |
| **Usage in Service** | java List<NameView> names = repo.findByCountry("India");  👉 Always returns NameView. | java List<NameView> names = repo.findByCountry("India", NameView.class); List<NameCompanyView> data = repo.findByCountry("India", NameCompanyView.class);  👉 Can return different projections using same method. |
| **Flexibility** | Only **one fixed projection per method**. | Same method can support **multiple projections**. |
| **Number of Methods Needed** | One method for each projection type.  Example:  java List<NameView> findByCountry(String country); List<NameCompanyView> findByCountry(String country); | Only **one method** is enough for all projections. |
| **Runtime Choice** | Choice is decided at compile-time (fixed). | Choice can be decided at runtime (NameView.class or NameCompanyView.class). |
| **Example Interfaces** | java public interface NameView { String getName(); } | java public interface NameView { String getName(); } public interface NameCompanyView { String getName(); String getCompany(); } |
| **When to Use?** | When requirement is **always fixed** (e.g., always need only name). | When requirement is **changing/flexible** (e.g., sometimes only name, sometimes name + company, sometimes more). |

# Stored Procedures in Spring Data JPA

1. **What is a Stored Procedure?**
   * A stored procedure is a pre-written SQL code stored in the database.
   * Instead of writing raw SQL in your Java code, you call the procedure by name.
   * Useful when business logic or complex queries are handled inside DB.
2. **Why Use Stored Procedures in JPA?**
   * For performance (database-optimized code).
   * For reusability (one procedure, many callers).
   * For complex logic that is better in SQL than Java.
3. **Ways to Call Stored Procedures in Spring Data JPA**
   * **Option 1:** Using @Procedure annotation on a repository method.
   * **Option 2:** Using @NamedStoredProcedureQuery on Entity class.

👉 Most projects use @Procedure because it is simple.

1. **Steps (using @Procedure)**
   * Create a stored procedure in the database (e.g., Oracle, MySQL, etc.).
   * Write a repository method and annotate with @Procedure.
   * Call that method from Service → then from Main.
2. **Example Stored Procedure (MySQL/Oracle)**
3. CREATE PROCEDURE GET\_EMPLOYEE\_SALARY(IN emp\_id BIGINT, OUT emp\_salary DOUBLE)
4. BEGIN
5. SELECT salary INTO emp\_salary FROM employee\_list WHERE id = emp\_id;

END;

**1. @Procedure**

* **What it does**:  
  It tells Spring Data JPA that **this repository method is not a normal query**, but instead it should **call a stored procedure** in the database.
* **Why we need it**:  
  Without @Procedure, Spring will try to build a JPQL/SQL query for the method name (like findById, findByName, etc.).  
  But here we want to execute **already written logic in the database (stored procedure)**, not generate SQL.
* **How it works**:
* @Procedure(procedureName = "GET\_EMPLOYEE\_SALARY")
* Double getEmployeeSalary(@Param("emp\_id") Long empId);
  + procedureName = "GET\_EMPLOYEE\_SALARY" → matches the exact stored procedure name in Oracle.
  + When we call this method in Java, Spring internally runs:
  + CALL GET\_EMPLOYEE\_SALARY(?, ?);

(passing parameters automatically).

**🔹 2. @Param**

* **What it does**:  
  It tells Spring **which Java method parameter maps to which stored procedure parameter** in the database.
* **Why we need it**:
  + In our stored procedure, the parameter name is emp\_id.
  + In Java method, the parameter is Long empId.
  + Spring doesn’t automatically know they are the same.
  + So we use @Param("emp\_id") to map them correctly.
* **How it works**:
* Double getEmployeeSalary(@Param("emp\_id") Long empId);
  + @Param("emp\_id") means → pass the value of empId Java variable into the stored procedure parameter emp\_id.

**🔹 In Simple English**

* @Procedure → “Hey Spring, don’t generate a query, just call this stored procedure.”
* @Param → “This Java variable corresponds to this DB parameter name.”

Eg: SpringDataJpaStoredProcedure

# Date and Time Operations in Spring Data JPA

1. **Why we need Date/Time in DB?**
   * In real-world applications, most tables have date/time columns → created\_date, last\_updated, joining\_date, dob, etc.
   * We use them for filtering records, sorting, and auditing.
2. **Common Java Types Used**
   * **java.util.Date** → old style, still supported.
   * **java.sql.Date, java.sql.Timestamp** → JDBC-specific, used less in modern projects.
   * **java.time.LocalDate, java.time.LocalTime, java.time.LocalDateTime** → modern Java 8+ Date/Time API (industry preferred).
3. **Spring Data JPA Queries with Dates**  
   You can use:
   * **Exact match** → findByJoiningDate(LocalDate date)
   * **Range queries** → findByJoiningDateBetween(LocalDate start, LocalDate end)
   * **Comparison** → findByJoiningDateAfter(LocalDate date), findByJoiningDateBefore(LocalDate date)
4. **SQL Mapping**
   * DATE in DB ↔ LocalDate in Java
   * TIMESTAMP in DB ↔ LocalDateTime in Java
5. **Is this used in the industry?** ✅  
   Absolutely. Every real-time project uses Date/Time columns for:
   * **Auditing** (created\_at, updated\_at)
   * **Reports** (sales between dates, employees joined this month)
   * **Filtering** (active users, expired subscriptions, etc.)

**1. LocalDate (📅 Date only – no time)**

* **Most commonly used** when you only care about the **date part** (year, month, day).
* ✅ Industry Usage Examples:
  + Employee joining date / birth date
  + Expiry date (e.g., subscription, license)
  + Invoice / billing date
  + Holidays and leave management

👉 **Most used for business record keeping.**

**🔹 2. LocalTime (⏰ Time only – no date)**

* Used when you only care about the **time of the day**.
* ✅ Industry Usage Examples:
  + Store opening and closing hours
  + Reminder notifications at a certain time
  + Scheduling (e.g., meetings at 10:30 AM daily)
  + IoT devices logging time-only events

👉 **Less frequently used alone** — usually combined with a date.

**🔹 3. LocalDateTime (📅 + ⏰ Date + Time)**

* **Very common in enterprise apps** when you need **both date and time together**.
* ✅ Industry Usage Examples:
  + Created / Updated timestamp for records (audit fields)
  + Login / Logout timestamps
  + Transaction timestamps in banking/finance
  + Log entries in applications
  + Order placed / shipped timestamps

👉 **Very frequently used in audit trails, logs, and transaction systems**

Eg: SpringDataJpaLocalDateTime