Spring Data JPA not only integrates JPA in your Spring stack, but it also makes using several features much easier. Here are 6 examples that are used in most applications.

1.Executing Basic JPQL Queries

JPQL is JPA's query language which enables you to define your query based on your domain instead of the table model. That requires 3 steps. You need to:

1. Define and instantiate the query
2. Set all bind parameter values
3. Execute the query

Depending on the complexity of your query, you can skip most or even all of these steps with Spring Data JPA.

If your query isn't too complicated and doesn't use more than 2 bind parameters, I recommend you use the derived query feature. Spring Data then generates the query based on the name of your repository method and executes it. You then don't need to write any JPA- or persistence-related code.

Here you can see two examples.

```
public interface AuthorRepository extends JpaRepository<Author, Long> {

    List<Author> findByFirstName(String firstName);

    List<Author> findByBooksTitle(String title);

}
```

When you call the *findByFirstName* method, Spring Data JPA generates a JPQL query that selects all *Author* entities with a given *firstName*. And the *findByBooksTitle* method returns all *Author* entities who've written a *Book* with a given *title*.

As you can see, defining and executing a basic query gets incredible easy. And you can also order your query results, use pagination, and create more complex expressions for your WHERE clause. I explained all of that in more details in my guide to derived queries with Spring Data JPA.

As comfortable as this feature is, sooner or later, your query gets too complex to express it in a method name. You can then annotate your repository method with a @Query annotation. Within the annotation, you can specify your JPQL query. You also need to provide a method parameter for each bind parameter used in the query.

```
public interface AuthorRepository extends JpaRepository<Author, Long> {

    @Query(
          "SELECT a FROM Author a WHERE firstName = ?1 AND lastName = ?2")
    List<Author> findByFirstNameAndLastName(String firstName,
                                            String lastName);

}
```

When you call that method, Spring Data JPA uses the provided statement to instantiate a query, sets the bind parameter values, and maps the result. This prevents you from writing lots of boilerplate code.

2. Using DTO projections

As I explained in an earlier article, DTO projections provide much better performance than entity projections. So, whenever you implement a read-only operation, you should prefer them over entities.

If you want to use DTO projections with plain JPA or Hibernate, you need to implement a DTO class with a constructor that expects all attributes as parameters.

In your JPQL or Criteria query, you can then use a constructor expression that references the constructor of your DTO class. This tells your persistence provider to call the constructor for each record in the result set and to return the created objects.

Spring Data JPA can handle most of these tasks for you. You just need to define an interface and use it as the return type of your repository method. Spring Data JPA then takes care of the rest.

```
public interface AuthorValueIntf {

    String getFirstName();
    void setFirstName(String firstName);

    String getLastName();
    void setLastName(String lastName);
}
```

```
public interface AuthorRepository extends JpaRepository<Author, Long> {

    List<AuthorValueIntf> findByFirstName(String firstName);

}
```

3. Paginate Your Query Results

Similar to SQL, you can decide to fetch only a subset of your query results with JPA. You can do that by calling the *setMaxResults* and *setFirstResult* methods on the *Query* interface.

When your user steps from page to page, you need to calculate the value you provide to the *setFirstResult* method for each page.

Spring Data JPA's *Pageable* interface makes that a little bit easier. You can add it as a parameter to your repository method, to activating pagination for your query.

```
public interface BookRepository extends JpaRepository&lt;Book, Long&gt; {

    Page<Book> findAll(Pageable pageable);

}
```

When you instantiate a new *Pageable* object, you just need to define which page you want to get and how many records are on a page. Spring Data JPA then calculate the correct LIMIT and OFFSET values for you.

```
Pageable pageable = PageRequest.of(0, 10);
Page<Book> b = bookRepository.findAll(pageable);
```

4. Using EntityGraphs

An EntityGraph provides an easy and reusable way to initialize required entity associations within your query. Instead of executing an additional query for each entity association, which is known as the n+1 select issue, Hibernate then gets all required information with just one query.

Defining and using such a graph with JPA isn't complicated, but it requires multiple steps. You first need to use a *@NamedEntityGraph* annotation or the *EntityGraph* API to define the graph.

```
@Entity
@Table(name = "purchaseOrder")
@NamedEntityGraph(
    name = "graph.Order.items",
    attributeNodes = @NamedAttributeNode(value = "items", subgraph = "items"),
    subgraphs = @NamedSubgraph(
                                    name = "items",
                                    attributeNodes = @NamedAttributeNode("product")))
public class Order { ... }
```

In the second step, you need to instantiate the graph and add it to your query.

```
EntityGraph graph = this.em.getEntityGraph("graph.Order.items");

Map hints = new HashMap();
hints.put("javax.persistence.fetchgraph", graph);

return this.em.find(Order.class, orderId, hints);
```

Spring Data JPA makes this a little bit easier. When you annotate your repository method with *@EntityGraph*, you can:

- Reference an existing *@NamedEntityGraph* by setting its name as the value attribute.
- Define an ad-hoc graph using the *attributePaths* attribute of the *@EntityGraph* annotation.

```
public interface OrderRepository extends JpaRepository<Order, Long> {

    @EntityGraph(value = "graph.Order.items", type = EntityGraphType.LOAD)
    List<Order> findByOrderNumber(String orderNumber);

}
```

5. Calling Stored Procedures

JPA provides 2 ways to call a stored procedure. You can use a *@NamedStoredProcedureQuery* annotation to define your stored procedure call at build time. And if you want to create an ad-hoc stored procedure call, you can use the *StoredProcedureQuery* API.

```
@NamedStoredProcedureQuery(
    name = "calculate",
    procedureName = "calculate",
    parameters = {
        @StoredProcedureParameter(mode = ParameterMode.IN, type =
Double.class, name = "x"),
        @StoredProcedureParameter(mode = ParameterMode.IN, type =
Double.class, name = "y"),
        @StoredProcedureParameter(mode = ParameterMode.OUT, type =
Double.class, name = "sum")
    }
)
```

Spring Data JPA makes calling a *@NamedStoredProcedureQuery* very easy.

With plain JPA, you need to instantiate the query, set the bind parameter values, and execute the query. Almost all of that is very repetitive boilerplate code.

Similar to the previously shown JPQL queries, Spring Data JPA takes care of the boilerplate code. You just need to annotate your repository method with *@Procedure* and provide method parameters with the same names as your bind parameters.

```
public interface OrderRepository extends JpaRepository<Order, Long> {

    @Procedure(name = "calculate")
    Double calculateOrderValue(Double x, Double y);

}
```

When you call the repository method, Spring Data JPA uses this information to instantiate the *@NamedStoredProcedureQuery*, set the bind parameter values, execute the query, and return the result.