

## 4.4.2. Query Creation

The query builder mechanism built into Spring Data repository infrastructure is useful for building constraining queries over entities of the repository. The mechanism strips the prefixes `find...By`, `read...By`, `query...By`, `count...By`, and `get...By` from the method and starts parsing the rest of it. The introducing clause can contain further expressions, such as a `Distinct` to set a distinct flag on the query to be created. However, the first `By` acts as delimiter to indicate the start of the actual criteria. At a very basic level, you can define conditions on entity properties and concatenate them with `And` and `Or`. The following example shows how to create a number of queries:

### Example 13. Query creation from method names

```
interface PersonRepository extends Repository<Person, Long> {

    List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);

    // Enables the distinct flag for the query
    List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);
    List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);

    // Enabling ignoring case for an individual property
    List<Person> findByLastnameIgnoreCase(String lastname);
    // Enabling ignoring case for all suitable properties
    List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);

    // Enabling static ORDER BY for a query
    List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
    List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
}
```

The actual result of parsing the method depends on the persistence store for which you create the query. However, there are some general things to notice:

- The expressions are usually property traversals combined with operators that can be concatenated. You can combine property expressions with `AND` and `OR`. You also get support for operators such as `Between`, `LessThan`, `GreaterThan`, and `Like` for the property expressions. The supported operators can vary by datastore, so consult the appropriate part of your reference documentation.
- The method parser supports setting an `IgnoreCase` flag for individual properties (for example, `findByLastnameIgnoreCase(...)`) or for all properties of a type that supports ignoring case (usually `String` instances — for example, `findByLastnameAndFirstnameAllIgnoreCase(...)`). Whether ignoring cases is supported may vary by store, so consult the relevant sections in the reference documentation for the store-specific query method.
- You can apply static ordering by appending an `OrderBy` clause to the query method that references a property and by providing a sorting direction (`Asc` or `Desc`). To create a query method that supports dynamic sorting, see “[Special parameter handling](#)”.

### 4.4.3. Property Expressions

Property expressions can refer only to a direct property of the managed entity, as shown in the preceding example. At query creation time, you already make sure that the parsed property is a property of the managed domain class. However, you can also define constraints by traversing nested properties. Consider the following method signature:

```
List<Person> findByAddressZipCode(ZipCode zipCode);
```

Assume a `Person` has an `Address` with a `ZipCode`. In that case, the method creates the property traversal `x.address.zipCode`. The resolution algorithm starts by interpreting the entire part (`AddressZipCode`) as the property and checks the domain class for a property with that name (uncapitalized). If the algorithm succeeds, it uses that property. If not, the algorithm splits up the source at the camel case parts from the right side into a head and a tail and tries to find the corresponding property — in our example, `AddressZip` and `Code`. If the algorithm finds a property with that head, it takes the tail and continues building the tree down from there, splitting the tail up in the way just described. If the first split does not match, the algorithm moves the split point to the left (`Address`, `ZipCode`) and continues.

Although this should work for most cases, it is possible for the algorithm to select the wrong property. Suppose the `Person` class has an `addressZip` property as well. The algorithm would match in the first split round already, choose the wrong property, and fail (as the type

of `addressZip` probably has no `code` property).

To resolve this ambiguity you can use `_` inside your method name to manually define traversal points. So our method name would be as follows:

```
List<Person> findByAddress_ZipCode(ZipCode zipCode);
```

Because we treat the underscore character as a reserved character, we strongly advise following standard Java naming conventions (that is, not using underscores in property names but using camel case instead).

#### 4.4.4. Special parameter handling

To handle parameters in your query, define method parameters as already seen in the preceding examples. Besides that, the infrastructure recognizes certain specific types like `Pageable` and `Sort`, to apply pagination and sorting to your queries dynamically. The following example demonstrates these features:

##### Example 14. Using `Pageable`, `Slice`, and `Sort` in query methods

```
Page<User> findByLastname(String lastname, Pageable pageable);  
  
Slice<User> findByLastname(String lastname, Pageable pageable);  
  
List<User> findByLastname(String lastname, Sort sort);  
  
List<User> findByLastname(String lastname, Pageable pageable);
```

APIs taking `Sort` and `Pageable` expect non- null values to be handed into methods. If you don't want to apply any sorting or pagination use `Sort.unsorted()` and `Pageable.unpaged()`.

The first method lets you pass an `org.springframework.data.domain.Pageable` instance to the query method to dynamically add paging to your statically defined query. A `Page` knows about the total number of elements and pages available. It does so by the infrastructure triggering a count query to calculate the overall number. As this might be expensive (depending on the store used), you can instead return a `Slice`. A `Slice` only knows about whether a next `Slice` is available, which might be sufficient when walking through a larger result set