



Introducing Hibernate Data Repositories

Version 7.0.10.Final

Table of Contents

Preface	1
1. Programming model	2
1.1. Repository interfaces	3
1.2. Organizing persistence operations	4
1.3. Default methods	5
1.4. Resource accessor methods	5
1.5. Lifecycle methods	5
1.6. Automatic query methods	6
1.7. Annotated query methods	7
1.8. @By and @Param	8
2. Configuration and integration	9
2.1. Project setup	9
2.2. Excluding classes from processing	10
2.3. Configuring Hibernate ORM	10
2.4. Obtaining a <code>StatelessSession</code>	10
2.5. Injecting a repository	10
2.6. Integration with Jakarta EE	11
3. Pagination and dynamic sorting	12
3.1. The static metamodel	12
3.2. Dynamic sorting	12
3.3. Limits	13
3.4. Offset-based pagination	13
3.5. Key-based pagination	14
3.6. Dynamic restrictions	15
3.7. Advanced control over querying	15
4. Reactive repositories	17
4.1. Defining a reactive repository	17
4.2. Obtaining a reactive repository	17
4.3. Calling a reactive repository	18

Preface

Jakarta Data is a new specification for *repositories*. A repository, in this context, means an interface exposing a typesafe API for interacting with a datastore. Jakarta Data is designed to accommodate a diverse range of database technologies, from relational databases, to document databases, to key-value stores and more.

Hibernate Data Repositories is an implementation of Jakarta Data targeting relational databases and backed by Hibernate ORM. Entity classes are mapped using the familiar annotations defined by Jakarta Persistence, and queries may be written in the Hibernate Query Language, a superset of the Jakarta Persistence Query Language (JPQL). On the other hand, the programming model for interacting with the database is quite different in Jakarta Data from the model you might be used to from Jakarta Persistence.

Therefore, this document will show you a different way to use Hibernate.

The coverage of Jakarta Data is intentionally inexhaustive. If exhaustion is sought, this document should be read in conjunction with the specification, which we've worked hard to keep readable.

If you are unfamiliar with Hibernate, this document should be read in conjunction with:

- the [Short Guide to Hibernate](#), and
- the [Guide to Hibernate Query Language](#).

Chapter 1. Programming model

Jakarta Data and Jakarta Persistence both represent data in a typesafe way, using *entity classes*. Since Hibernate’s implementation of Jakarta Data is backed by access to a relational database, these entity classes are mapped using the annotations defined by Jakarta Persistence.

For example:

```
@Entity
public class Book {
    @Id
    String isbn;

    @Basic(optional = false)
    String title;

    LocalDate publicationDate;

    @Basic(optional = false)
    String text;

    @Enumerated(String)
    @Basic(optional = false)
    Type type = Type.Book;

    @ManyToOne(optional = false, fetch = LAZY)
    Publisher publisher;

    @ManyToMany(mappedBy = Author_.BOOKS)
    Set<Author> authors;

    ...
}

@Entity
public class Author {
    @Id
    String ssn;

    @Basic(optional = false)
    String name;

    Address address;

    @ManyToMany
    Set<Book> books;
}
```

For more information about mapping entities, see the [Introduction to Hibernate 6](#).



Jakarta Data also works with entities defined using similarly-named annotations defined by Jakarta NoSQL. But in this document we’re using Hibernate Data Repositories, so all mapping annotations should be understood to be the ones defined in `jakarta.persistence` or `org.hibernate.annotations`. For more information about entities in Jakarta Data, please consult chapter 3 of the specification.

Furthermore, queries may be expressed in HQL, Hibernate’s superset of the Jakarta Persistence Query Language (JPQL).



The Jakarta Data specification defines a simple subset of JPQL called, appropriately, JDQL. JDQL is mostly relevant to non-relational datastores; an implementation of Jakarta Data backed by access to relational data is normally expected to support a much larger subset of JPQL. Indeed, Hibernate Data Repositories supports a *superset* of JPQL. So, even though we put rather a large amount of effort into advocating, designing, and specifying JDQL, we won’t talk much about it here. For information about JDQL, please consult chapter 5 of the Jakarta Data specification.

To learn more about HQL and JPQL, see the [Guide to Hibernate Query Language](#).

This is where the similarity between Jakarta Persistence and Jakarta Data ends. The following table contrasts the two programming models.

	Persistence	Data
Persistence contexts	Stateful	Stateless
Gateway	EntityManager interface	User-written @Repository interface
Underlying implementation	Session	StatelessSession
Persistence operations	Generic methods like <code>find()</code> , <code>persist()</code> , <code>merge()</code> , <code>remove()</code>	Typesafe user-written methods annotated <code>@Find</code> , <code>@Insert</code> , <code>@Update</code> , <code>@Save</code> , <code>@Delete</code>
SQL execution	During flush	Immediate
Updates	Usually implicit (dirty checking during flush)	Always explicit (by calling <code>@Update</code> method)
Operation cascading	Depends on <code>CascadeType</code>	Never
Lazy fetching	Implicit	Explicit using <code>StatelessSession.fetch()</code>
Validation of JPQL	Runtime	Compile time

The fundamental difference here is that Jakarta Data does not feature stateful persistence contexts. Among other consequences:

- entity instances are always detached, and so
- updates require an explicit operation, and
- there's no transparent lazy association fetching.

It's important to understand that a repository in Hibernate Data Repositories is backed by a `StatelessSession`, not by a Jakarta Persistence `EntityManager`.



There's only one portable way to fetch an association in Jakarta Data, and that's by using a JPQL `join fetch` clause, in a `@Query` annotation. The specification does not provide a portable way to fetch an association lazily. To fetch an association, we need to [call the `StatelessSession` directly](#). This really isn't as bad as it sounds; overuse of lazy fetching is associated with poor performance due to many round trips to the database server.



A future release of Jakarta Data will feature repositories backed by Jakarta Persistence stateful persistence contexts, but this functionality did not make the cut for Jakarta Data 1.0.

The second big difference is that instead of providing a generic interface like `EntityManager` that's capable of performing persistence operations for any entity class, Jakarta Data requires that each interaction with the database go via a user-written method specific to just one entity type. The method is marked with annotations allowing Hibernate to fill in the method implementation.

For example, whereas Jakarta Persistence defines the methods `find()` and `persist()` of `EntityManager`, in Jakarta Data the application programmer is required to write an interface like the following:

```
@Repository
interface Library {
    @Find
    Book book(String isbn);

    @Insert
    void add(Book book);
}
```

This is our first example of a repository.

1.1. Repository interfaces

A *repository interface* is an interface written by you, the application programmer, and annotated `@Repository`. The implementation of the repository interface is provided by a Jakarta Data provider, in our case, by Hibernate Data Repositories.

The Jakarta Data specification does not say how this should work, but in Hibernate Data Repositories, the implementation is generated by an annotation processor. In fact, you might already be using this annotation processor: it's just `HibernateProcessor` from the module which used to be called `hibernate-jpamodelgen`, and has now been renamed `hibernate-processor` in Hibernate 7.



That's right, this fancy thing I'm calling Hibernate Data Repositories is really just a new feature of Hibernate's venerable static metamodel generator. If you're already using the JPA static metamodel in your project, you already have Jakarta Data at your fingertips. If you don't, we'll see how to set it up in the [next chapter](#).

Of course, a Jakarta Data provider can't generate an implementation of any arbitrary method. Therefore, the methods of a repository interface must fall into one of the following categories:

- default [methods](#),
- [lifecycle methods](#) annotated `@Insert`, `@Update`, `@Delete`, or `@Save`,
- [automatic query methods](#) annotated `@Find`,
- [annotated query methods](#) annotated `@Query` or `@SQL`, and
- [resource accessor methods](#).



For users migrating from Spring Data, Jakarta Data also provides a *Query by Method Name* facility. We don't recommend this approach for new code, since it leads to extremely verbose and unnatural method names for anything but the most trivial examples.

We'll discuss each of these kinds of method soon. But first we need to ask a more basic question: how are persistence operations organized into repositories, and how do repository interfaces relate to entity types? The—perhaps surprising—answer is: it's completely up to you.

1.2. Organizing persistence operations

Jakarta Data lets you freely assign persistence operations to repositories according to your own preference. In particular, Jakarta Data does not require that a repository interface inherit a built-in supertype declaring the basic "CRUD" operations, and so it's not necessary to have a separate repository interface for each entity. You're permitted, for example, to have a single `Library` interface instead of `BookRepository`, `AuthorRepository`, and `PublisherRepository`.

Thus, the whole programming model is much more flexible than older approaches such as Spring Data, which require a repository interface per entity class, or, at least, per so-called "aggregate".



The concept of an "aggregate" makes sense in something like a document database. But relational data does not have aggregates, and you should avoid attempting to shoehorn your relational tables into this inappropriate way of thinking about data.

As a convenience, especially for users migrating from older frameworks, Jakarta Data does define the `BasicRepository` and `CrudRepository` interfaces, and you can use them if you like. But in Jakarta Data there's not much special about these interfaces; their operations are declared using the same annotations you'll use to declare methods of your own repositories. This older, less-flexible approach is illustrated in the following example.

```
// old way

@Repository
interface BookRepository
    extends CrudRepository<Book,String> {
    // query methods
    ...
}

@Repository
interface AuthorRepository
    extends CrudRepository<Author,String> {
    // query methods
    ...
}
```

We won't see `BasicRepository` and `CrudRepository` again in this document, because they're not necessary, and because they implement the older, less-flexible way of doing things.

Instead, our repositories will often group together operations dealing with several related entities, even when the entities don't have a single "root". This situation is *extremely* common in relational data models. In our example, `Book` and `Author` are related by a `@ManyToMany` association, and are both "roots".

```
// new way

@Repository
```

```

interface Publishing {
    @Find
    Book book(String isbn);

    @Find
    Author author(String ssn);

    @Insert
    void publish(Book book);

    @Insert
    void create(Author author);

    // query methods
    ...
}

```

Now let's walk through the different kinds of method that a repository interface might declare, beginning with the easiest kind. If the following summary is insufficient, you'll find more detailed information about repositories in chapter 4 of the Jakarta Data specification, and in the Javadoc of the relevant annotations.

1.3. Default methods

A default method is one you implement yourself, and there's nothing special about it.

```

@Repository
interface Library {
    default void hello() {
        System.out.println("Hello, World!");
    }
}

```

This doesn't look very useful, at least not unless there's some way to interact with the database from a default method. For that, we'll need to add a resource accessor method.

1.4. Resource accessor methods

A resource accessor method is one which exposes access to an underlying implementation type. Currently, Hibernate Data Repositories only supports one such type: `StatelessSession`. So a resource accessor method is just any abstract method which returns `StatelessSession`. The name of the method doesn't matter.

```

StatelessSession session();

```

This method returns the `StatelessSession` backing the repository.



Usually, a resource accessor method is called from a default method of the same repository.

```

default void refresh(Book book) {
    session().refresh(book);
}

```

This is very useful when we need to gain direct access to the `StatelessSession` in order to take advantage of the full power of Hibernate.



A resource accessor method is also useful when we need to lazily fetch an association.

```

library.session().fetch(book.authors);

```

Usually, of course, we want Jakarta Data to take care of interacting with the `StatelessSession`.

1.5. Lifecycle methods

Jakarta Data 1.0 defines four built-in lifecycle annotations, which map perfectly to the basic operations of the Hibernate `StatelessSession`:

- @Insert maps to insert(),
- @Update maps to update(),
- @Delete maps to delete(), and
- @Save maps to upsert().



The basic operations of `StatelessSession`—`insert()`, `update()`, `delete()`, and `upsert()`—do not have matching `CascadeTypes`, and so these operations are never cascaded to associated entities.

A lifecycle method usually accepts an instance of an entity type, and is usually declared `void`.

```
@Insert
void add(Book book);
```

Alternatively, it may accept a list or array of entities. (A variadic parameter is considered an array.)

```
@Insert
void add(Book... books);
```



A future release of Jakarta Data might expand the list of built-in lifecycle annotations. In particular, we're hoping to add `@Persist`, `@Merge`, `@Refresh`, `@Lock`, and `@Remove`, mapping to the fundamental operations of `EntityManager`.

Repositories wouldn't be useful at all if this was all they could do. Jakarta Data really starts to shine when we start to use it to express queries.

1.6. Automatic query methods

An automatic query method is usually annotated `@Find`. The simplest automatic query method is one which retrieves an entity instance by its unique identifier.

```
@Find
Book book(String isbn);
```

The name of the parameter identifies that this is a lookup by primary key (the `isbn` field is annotated `@Id` in `Book`) and so this method will be implemented to call the `get()` method of `StatelessSession`.



If the parameter name does not match any field of the returned entity type, or if the type of the parameter does not match the type of the matching field, `HibernateProcessor` reports a helpful error at compilation time. This is our first glimpse of the advantages of using Jakarta Data repositories with Hibernate.

If there is no `Book` with the given `isbn` in the database, the method throws `EmptyResultException`. There's two ways around this if that's not what we want:

- declare the method to return `Optional`, or
- annotate the method `@jakarta.annotation.Nullable`.

The first option is blessed by the specification:

```
@Find
Optional<Book> book(String isbn);
```

The second option is an extension provided by Hibernate:

```
@Find @Nullable
Book book(String isbn);
```

An automatic query method might return multiple results. In this case, the return type must be an array or list of the entity type.

```
@Find
List<Book> book(String title);
```

Usually, arguments to a parameter of an automatic query method must match *exactly* with the field of an entity. However, Hibernate provides the `@Pattern` annotation to allow for "fuzzy" matching using `like`.

```
@Find
List<Book> books(@Pattern String title);
```

Furthermore, if the parameter type is a list or array of the entity field type, the resulting query has an `in` condition.

```
@Find
List<Book> books(String[] isbn);
```

Of course, an automatic query method might have multiple parameters.

```
@Find
List<Book> book(@Pattern String title, Year yearPublished);
```

In this case, every argument must match the corresponding field of the entity.

The `_` character in a parameter name may be used to navigate associations:

```
@Find
List<Book> booksPublishedBy(String publisher_name);
```

However, once our query starts to involve multiple entities, it's usually better to use an [annotated query method](#).

The `@OrderBy` annotation allows results to be sorted.

```
@Find
@OrderBy("title")
@OrderBy("publisher.name")
List<Book> book(@Pattern String title, Year yearPublished);
```

This might not look very typesafe at first glance, but—amazingly—the content of the `@OrderBy` annotation is completely validated at compile time, as we will see below.

Automatic query methods are great and convenient for very simple queries. For anything that's not extremely simple, we're much better off writing a query in JPQL.

1.7. Annotated query methods

An annotated query method is declared using:

- `@Query` from Jakarta Data, or
- `@HQL` or `@SQL` from `org.hibernate.annotations.processing`.

The `@Query` annotation is defined to accept JPQL, JDQL, or anything in between. In Hibernate Data Repositories, it accepts arbitrary HQL.



There's no strong reason to use `@HQL` in preference to `@Query`. This annotation exists because the functionality described here predates the existence of Jakarta Data.

Consider the following example:

```
@Query("where title like :pattern order by title, isbn")
List<Book> booksByTitle(String pattern);
```

You might notice that:

- The `from` clause is not required in JDQL, and is inferred from the return type of the repository method.
- Since Jakarta Persistence 3.2, neither the `select` clause nor entity aliases (identification variables) are required in JPQL, finally standardizing a very old feature of HQL.

This allows simple queries to be written in a very compact form.

Method parameters are automatically matched to ordinal or named parameters of the query. In the previous example, `pattern` matches `:pattern`. In the following variation, the first method parameter matches `?1`.

```
@Query("where title like ?1 order by title, isbn")
List<Book> booksByTitle(String pattern);
```

You might be imagining that the JPQL query specified within the `@Query` annotation cannot be validated at compile time, but this is not the case. `HibernateProcessor` is not only capable of validating the *syntax* of the query, but it even *typechecks* the query completely. This is much better than passing a string to the `createQuery()` method of `EntityManager`, and it's probably the top reason to use Jakarta Data with Hibernate.

When a query returns more than one object, the nicest thing to do is package each result as an instance of a Java record type. For example, we might define a record holding some fields of `Book` and `Author`.

```
record AuthorBookSummary(String isbn, String ssn, String authorName, String title) {}
```

We need to specify that the values in the `select` clause should be packaged as instances of `AuthorBookSummary`. The JPQL specification provides the `select new` construct for this.

```
@Query("select new AuthorBookSummary(b.isbn, a.ssn, a.name, b.title " +  
      "from Author a join books b " +  
      "where title like :pattern")  
List<AuthorBookSummary> summariesForTitle(String pattern);
```

Note that the `from` clause is required here, since it's impossible to infer the queried entity type from the return type of the repository method.



Since this is quite verbose, Hibernate doesn't require the use of `select new`, nor of aliases, and lets us write:

```
@Query("select isbn, ssn, name, title " +  
      "from Author join books " +  
      "where title like :pattern")  
List<AuthorBookSummary> summariesForTitle(String pattern);
```

An annotated query method may even perform an update, delete, or insert statement.

```
@Query("delete from Book " +  
      "where extract(year from publicationDate) < :year")  
int deleteOldBooks(int year);
```

The method must be declared `void`, or return `int` or `long`. The return value is the number of affected records.

Finally, a native SQL query may be specified using `@SQL`.

```
@SQL("select title from books where title like :pattern order by title, isbn")  
List<String> booksByTitle(String pattern);
```

Unfortunately, native SQL queries cannot be validated at compile time, so if there's anything wrong with our SQL, we won't find out until we run our program.

1.8. @By and @Param

Query methods match method parameters to entity fields or query parameters by name. Occasionally, this is inconvenient, resulting in less natural method parameter names. Let's reconsider an example we already saw above:

```
@Find  
List<Book> books(String[] ibsn);
```

Here, because the parameter name must match the field `isbn` of `Book`, we couldn't call it `isbn`s, plural.

The `@By` annotation lets us work around this problem:

```
@Find  
List<Book> books(@By("isbn") String[] ibsns);
```

Naturally, the name and type of the parameter are still checked at compile time; there's no loss of typesafety here, despite the string.

The `@Param` annotation is significantly less useful, since we can always rename our HQL query parameter to match the method parameter, or, at worst, use an ordinal parameter instead.

Chapter 2. Configuration and integration

Getting started with Hibernate Data Repositories involves the following steps:

1. set up a project with Hibernate ORM and HibernateProcessor,
2. configure a persistence unit,
3. make sure a `StatelessSession` for that persistence unit is available for injection, and then
4. inject a repository using CDI or some other implementation of `jakarta.inject`.

2.1. Project setup

We definitely need the following dependencies in our project:

Table 1. Required dependencies

Dependency	Explanation
<code>jakarta.data:jakarta.data-api</code>	The Jakarta Data API
<code>org.hibernate.orm:hibernate-core</code>	Hibernate ORM
<code>org.hibernate.orm:hibernate-processor</code>	The annotation processor itself

And we'll need to pick a JDBC driver:

Table 2. JDBC driver dependencies

Database	Driver dependency
PostgreSQL or CockroachDB	<code>org.postgresql:postgresql</code>
MySQL or TiDB	<code>com.mysql:mysql-connector-j</code>
MariaDB	<code>org.mariadb.jdbc:mariadb-java-client</code>
DB2	<code>com.ibm.db2:jcc</code>
SQL Server	<code>com.microsoft.sqlserver:mssql-jdbc</code>
Oracle	<code>com.oracle.database.jdbc:ojdbc17</code>
H2	<code>com.h2database:h2</code>
HSQLDB	<code>org.hsqldb:hsqldb</code>

In addition, we might add some of the following to the mix.

Table 3. Optional dependencies

Optional dependency	Explanation
<code>org.hibernate.validator:hibernate-validator</code> and <code>org.glassfish:jakarta.el</code>	Hibernate Validator
<code>org.apache.logging.log4j:log4j-core</code>	log4j
<code>org.jboss.weld:weld-core-impl</code>	Weld CDI

You'll need to configure the annotation processor to run when your project is compiled. In Gradle, for example, you'll need to use `annotationProcessor`.

```
annotationProcessor 'org.hibernate.orm:hibernate-processor:7.0.0'
```

2.2. Excluding classes from processing

There's three ways to limit the annotation processor to certain classes:

1. A given repository may be excluded from processing simply by specifying `@Repository(provider="acme")` where "acme" is any string other than the empty string or a string equal, ignoring case, to "Hibernate". This is the preferred solution when there are multiple Jakarta Data Providers available.
2. A package or type may be excluded by annotating it with the `@Exclude` annotation from `org.hibernate.annotations.processing`.
3. The annotation processor may be limited to consider only certain types or certain packages using the `include` configuration option, for example, `-Ainclude=*.entity.*,*Repository`. Alternatively, types or packages may be excluded using the `exclude` option, for example, `-Aexclude=*Impl`.

2.3. Configuring Hibernate ORM

How you configure Hibernate depends on the environment you're running in, and on your preference:

- in Java SE, we often just use `hibernate.properties`, but some people prefer to use `persistence.xml`, especially in case of multiple persistence units,
- in Quarkus, we must use `application.properties`, and
- in a Jakarta EE container, we usually use `persistence.xml`.

Here's a simple `hibernate.properties` file for h2 database, just to get you started.

```
# Database connection settings
jakarta.persistence.jdbc.url=jdbc:h2:~/h2temp;DB_CLOSE_DELAY=-1
jakarta.persistence.jdbc.user=sa
jakarta.persistence.jdbc.pass=

# Echo all executed SQL to console
hibernate.show_sql=true
hibernate.format_sql=true
hibernate.highlight_sql=true

# Automatically export the schema
hibernate.hbm2ddl.auto=create
```

Please see the [Introduction to Hibernate 6](#) for more information about configuring Hibernate.

2.4. Obtaining a `StatelessSession`

Each repository implementation must somehow obtain a `StatelessSession` for its persistence unit. This usually happens via dependency injection, so you'll need to make sure that a `StatelessSession` is available for injection:

- in Quarkus, this problem is already taken care of for us—there's always an injectable `StatelessSession` bean for each persistence unit, and
- in a Jakarta EE environment, `HibernateProcessor` generates special code which takes care of creating and destroying the `StatelessSession`, but
- in other environments, this is something we need to take care of ourselves.



Depending on the libraries in your build path, `HibernateProcessor` generates different code. For example, if Quarkus is on the build path, the repository implementation is generated to obtain the `StatelessSession` directly from CDI in a way which works in Quarkus but not in WildFly.

If you have multiple persistence units, you'll need to disambiguate the persistence unit for a repository interface using `@Repository(dataStore="my-persistence-unit-name")`.

2.5. Injecting a repository

In principle, any implementation of `jakarta.inject` may be used to inject a repository implementation.

```
@Inject Library library;
```

However, this code will fail if the repository implementation is not able to obtain a `StatelessSession` from the bean container.

It's always possible to instantiate a repository implementation directly.

```
Library library = new Library_(statelessSession);
```

This is useful for testing, or for executing in an environment with no support for `jakarta.inject`.

2.6. Integration with Jakarta EE

Jakarta Data specifies that methods of a repository interface may be annotated with:

- Jakarta Bean Validation constraint annotations, and
- Jakarta Interceptors interceptor binding types, including,
- in particular, the `@Transactional` interceptor binding defined by Jakarta Transactions.

Note that these annotations are usually applied to a CDI bean implementation class, not to an interface,^[1] but a special exception is made for repository interfaces.

Therefore, when running in a Jakarta EE environment, or in Quarkus, and when an instance of a repository interface is obtained via CDI, the semantics of such annotations is respected.

```
@Transactional @Repository
public interface Library {

    @Find
    Book book(@NotNull String isbn);

    @Find
    Book book(@NotBlank String title, @NotNull LocalDate publicationDate);

}
```

As an aside, it's rather satisfying to see all these things working so nicely together, since we members of the Hibernate team played pivotal roles in the creation of the Persistence, Bean Validation, CDI, Interceptors, and Data specifications.

[1] `@Inherited` annotations are inherited from superclass to subclass, but not from interface to implementing class.

Chapter 3. Pagination and dynamic sorting

An [automatic](#) or [annotated](#) query method may have additional parameters which specify:

- additional sorting criteria, and/or
- a limit and offset restricting the results which are actually returned to the client.

Before we see this, let's see how we can refer to a field of an entity in a completely typesafe way.

3.1. The static metamodel

You might already be familiar with the Jakarta Persistence static metamodel. For an entity class `Book`, the class `Book_` exposes objects representing the persistent fields of `Book`, for example, `Book_.title` represents the field `title`. This class is generated by `HibernateProcessor` at compilation time.

Jakarta Data has its own static metamodel, which is different to the Jakarta Persistence metamodel, but conceptually very similar. Instead of `Book_`, the Jakarta Data static metamodel for `Book` is exposed by the class `_Book`.



The Jakarta Persistence static metamodel is most commonly used together with the Criteria Query API or the EntityGraph facility. Even though these APIs aren't part of the programming model of Jakarta Data, you can still use them from a default method of a repository by [calling the StatelessSession](#) directly.

Let's see how the static metamodel is useful, by considering a simple example.

It's perfectly possible to obtain an instance of `Sort` by passing the name of a field:

```
var sort = Sort.asc("title");
```

Unfortunately, since this is in regular code, and not in an annotation, the field name `"title"` cannot be validated at compile time.

A much better solution is to use the static metamodel to obtain an instance of `Sort`.

```
var sort = _Book.title.asc();
```

The static metamodel also declares constants containing the names of persistent fields. For example, `_Book.TITLE` evaluates to the string `"title"`.



These constants are sometimes used as annotation values.

```
@Find
@OrderBy(_Book.TITLE)
@OrderBy(_Book.ISBN)
List<Book> books(@Pattern String title, Year yearPublished);
```

This example looks superficially more typesafe. But since Hibernate Data Repositories already validates the content of the `@OrderBy` annotation at compile time, it's not really better.

3.2. Dynamic sorting

Dynamic sorting criteria are expressed using the types `Sort` and `Order`:

- an instance of `Sort` represents a single criterion for sorting query results, and
- an instance of `Order` packages multiple Sorting criteria together.

A query method may accept an instance of `Sort`.

```
@Find
List<Book> books(@Pattern String title, Year yearPublished,
                Sort<Book> sort);
```

This method might be called as follows:

```
var books =
    library.books(pattern, year,
                  _Book.title.ascIgnoreCase());
```

Alternatively the method may accept an instance of `Order`.

```
@Find
List<Book> books(@Pattern String title, Year yearPublished,
                Order<Book> order);
```

The method might now be called like this:

```
var books =
    library.books(pattern, year,
                  Order.of(_Book.title.ascIgnoreCase(),
                           _Book.isbn.asc()));
```

Dynamic sorting criteria may be combined with static criteria.



If the entity has entity supertypes, use a lower bounded wildcard in the repository method declaration, for example:

```
@Find
List<Book> books(@Pattern String title, Year yearPublished,
                Order<? super Book> order);
```

This lets the caller sort by fields of the supertype.

```
@Find
@OrderBy("title")
List<Book> books(@Pattern String title, Year yearPublished,
                Sort<Book> sort);
```

We're not convinced this is very useful in practice.

3.3. Limits

A `Limit` is the simplest way to express a subrange of query results. It specifies:

- `maxResults`, the maximum number of results to be returned from the database server to the client, and,
- optionally, `startAt`, the position of the first result to be returned to the client.

These values map directly to the familiar `setMaxResults()` and `setFirstResult()` of the Jakarta Persistence Query interface.

```
@Find
@OrderBy(_Book.TITLE)
List<Book> books(@Pattern String title, Year yearPublished,
                Limit limit);
```

```
var books =
    library.books(pattern, year,
                  Limit.of(MAX_RESULTS));
```



Whereas `Query.setFirstResult()` is an *offset* of the first result to be returned, with `setFirstResult(0)` meaning "no offset", `Limit.startAt` numbers results from one, with `Limit.range(1, n)` meaning "at most *n* results, starting from the first result".

A more sophisticated approach is provided by `PageRequest`.

3.4. Offset-based pagination

A `PageRequest` is superficially similar to a `Limit`, except that it's specified in terms of:

- a page size, and
- a numbered page.

We can use a `PageRequest` just like a `Limit`.

```

@Find
@OrderBy("title")
@OrderBy("isbn")
List<Book> books(@Pattern String title, Year yearPublished,
                PageRequest pageRequest);

var books =
    library.books(pattern, year,
                  PageRequest.ofSize(PAGE_SIZE));

```



Query results should be totally ordered when a repository method is used for pagination. The easiest way to be sure that you have a well-defined total order is to specify the identifier of the entity as the last element of the order. For this reason, we specified `@OrderBy("isbn")` in the previous example.

However, a repository method which accepts a `PageRequest` may return a `Page` of results instead of a `List`, making it easier to implement pagination.

```

@Find
@OrderBy("title")
@OrderBy("isbn")
Page<Book> books(@Pattern String title, Year yearPublished,
                PageRequest pageRequest);

var page =
    library.books(pattern, year,
                  PageRequest.ofSize(PAGE_SIZE));
var books = page.content();
long totalPages = page.totalPages();
// ...
while (page.hasNext()) {
    page = library.books(pattern, year,
                        page.nextPageRequest().withoutTotal());
    books = page.content();
    // ...
}

```

Pagination may be combined with dynamic sorting.

```

@Find
Page<Book> books(@Pattern String title, Year yearPublished,
                PageRequest pageRequest, Order<Book> order);

```



It's important to pass the same arguments to query parameters, and the same sorting criteria, with each page request! The repository is stateless: it doesn't remember the values passed on the previous page request.

A repository method with return type `Page` uses SQL offset and limit to implement pagination. We'll refer to this as *offset-based pagination*. A problem with this approach is that it's quite vulnerable to missed or duplicate results when the database is modified between page requests. Therefore, Jakarta Data offers an alternative solution, which we'll call *key-based pagination*.

3.5. Key-based pagination

In key-based pagination, the query results must be totally ordered by a unique key of the result set. The SQL offset is replaced with a restriction on the unique key, appended to the `where` clause of the query:

- a request for the *next* page of query results uses the key value of the *last* result on the current page to restrict the results, or
- a request for the *previous* page of query results uses the key value of the *first* result on the current page to restrict the results.



For key-based pagination, it's *essential* that the query has a total order.

From our point of view as users of Jakarta Data, key-based pagination works almost exactly like offset-based pagination. The difference is that we must declare our repository method to return `CursoredPage`.

```

@Find
@OrderBy("title")
@OrderBy("isbn")

```

```
CorsoredPage<Book> books(@Pattern String title, Year yearPublished,
                          PageRequest pageRequest);
```

On the other hand, with key-based pagination, Hibernate must do some work under the covers rewriting our query.



Key-based pagination goes some way to protect us from skipped or duplicate results. The cost is that page numbers can lose synchronization with the query result set during navigation. This isn't usually a problem, but it's something to be aware of.

Direct API support for key-based pagination originated in the work of Hibernate team member Christian Beikov back in 2015 in the Blaze-Persistence framework. It was adopted from there by the Jakarta Data specification, and is now even available in Hibernate ORM via the `KeyedPage/KeyedResultList` API.

3.6. Dynamic restrictions

Jakarta Data 1.0 does not include an API for programmatically specifying restrictions, but for now we may use the native `Restriction` API in Hibernate 7.



Restrictions will be standardized by Jakarta Data 1.1.

Hibernate, an atomic `Restriction` is formed from:

- a reference to a JPA `SingularAttribute`, usually obtained via the *Jakarta Persistence* (not Jakarta Data) static metamodel, together with
- a Range of allowed values for that attribute.

A query method may have a parameter of type `Restriction`, for example:

```
@Find
List<Book> books(Restriction<Book> restriction,
                Order<Book> order);
```

This method would be called like this:

```
var books =
    library.books(Restriction.contains(Book_.title, "Hibernate"),
                  Order.of(_Book.title.ascIgnoreCase(),
                           _Book.isbn.asc()));
```

Notice the mix of metamodels here: `Book_` is the Persistence metamodel, and `_Book` is the Data metamodel.

It's even possible to directly use a `Range` to restrict a given property or field of an entity:

```
@Find
List<Book> books(Range<String> title, Range<Year> yearPublished,
                Order<Book> order);
```

There are various kinds of `Range`, including lists, patterns, and intervals:

```
var books =
    library.books(Range.prefix("Hibernate"),
                  Range.closed(Year.of(2000), Year.of(2009)),
                  Order.of(_Book.title.ascIgnoreCase(),
                           _Book.isbn.asc()));
```

3.7. Advanced control over querying

For more advanced usage, an automatic or annotated query method may be declared to return `jakarta.persistence.Query`, `jakarta.persistence.TypedQuery`, `org.hibernate.query.Query`, or `org.hibernate.query.SelectionQuery`.

```
@Find
SelectionQuery<Book> booksQuery(@Pattern String title, Year yearPublished);

default List<Book> booksQuery(String title, Year yearPublished) {
```

```
    return books(title, yearPublished)
        .enableFetchProfile(_Book.PROFILE_WITH_AUTHORS)
        .setReadOnly(true)
        .setTimeout(QUERY_TIMEOUT)
        .getResultList();
}
```

This allows for direct control over query execution, without loss of type safety.

Chapter 4. Reactive repositories

Hibernate Data Repositories provides repositories backed by [Hibernate Reactive](#) for use in reactive programming. The methods of a reactive repository are non-blocking, and so every operation returns a reactive stream.



Jakarta Data 1.0 does not define a way to write repositories for use in reactive programming, but the spec was written to accommodate such extensions. This capability is being written into Data 1.1 under the more general category of *asynchronous repositories*.

In Hibernate Data Repositories we use [Mutiny](#) to work with reactive streams.



A future version of Hibernate Data Repositories will also allow the use of `CompletionStage` from `java.util.concurrent`. Asynchronous repositories based on Mutiny are unlikely to ever be portable to other implementations of Jakarta Data.

In our opinion, Mutiny is a *much* more comfortable API than `CompletionStage`.



The documentation for Hibernate Reactive contains a [comparison table](#) that's useful if you're more familiar with `CompletionStage`.

4.1. Defining a reactive repository

In the following code example we notice the two requirements for a reactive repository in Hibernate Data Repositories:

1. there must be a resource accessor method returning the underlying `Mutiny.StatelessSession` from Hibernate Reactive, and
2. the return type of every other operation is `Uni`, a reactive stream type defined by Mutiny.

For example, a `@Find` method which would return `Book` in a regular Jakarta Data repository must return `Uni<Book>` in a reactive repository. Similarly, lifecycle methods usually return `Uni<Void>` instead of `void`.

```
@Repository
interface Library {

    Mutiny.StatelessSession session();

    @Find
    Uni<Book> book(String isbn);

    @Insert
    Uni<Void> add(Book book);

    @Find
    Uni<List<Book>> books(@By("isbn") String[] ibsns);
}
```

It's *not* possible to mix blocking and non-blocking operations in the same repository interface.



Depending on how you're managing the stateless session, you might need to declare the resource accessor method with the type `Uni<Mutiny.StatelessSession>`.

4.2. Obtaining a reactive repository

To make use of our reactive repository, we'll need to bootstrap Hibernate Reactive and obtain a `Mutiny.SessionFactory`. For example, if we have a persistence unit named `example` in our `persistence.xml` file, we can obtain a `SessionFactory` like this:

```
Mutiny.SessionFactory factory =
    createEntityManagerFactory("example")
        .unwrap(Mutiny.SessionFactory.class);
```

Please refer to the [documentation for Hibernate Reactive](#) for more information on this topic.



In Quarkus, this step is unnecessary, and you can let Quarkus manage and inject the reactive `SessionFactory`.

Once we have the `SessionFactory`, we can easily obtain a `Mutiny.StatelessSession`, and use it to instantiate our repository:

```
factory.withStatelessTransaction(session -> {
    Library library = new Library_(session);
    ...
})
```



An even better approach is to make a `@RequestScoped` instance of `Mutiny.StatelessSession` or `Uni<Mutiny.StatelessSession>` available for injection by CDI. Then the `Library` repository itself may be directly injected, and you won't have to worry about managing the stateless session in application program code. This is a little bit tricky to get working perfectly, so hopefully by the time you're reading this, there will already be a built-in implementation in Quarkus.

4.3. Calling a reactive repository

To actually make use of a reactive repository, you'll need to be familiar with the programming model of reactive streams. For this, we refer you to the Mutiny documentation, and to the documentation for Hibernate Reactive, which goes over some gotchas.

The most important thing to understand is that a code fragment like the following does *not* result in any immediate interaction with the database:

```
Uni<Void> uni =
    factory.withStatelessTransaction(session -> {
        Library library = new Library_(session);
        return library.book("9781932394153");
    })
    .invoke(book -> out.println(book.title))
    .replaceWithVoid();
```

This code does no more than construct a reactive stream. We can execute the stream blockingly by calling `uni.await().indefinitely()`, but that's not something we would ever do in real code. Instead, what we usually do is simply return the stream, allowing it to be executed in a non-blocking way.