Holon
PLATFORM

Holon Platform JPA Datastore Module - Reference manual

Version 5.1.0

Table of Contents

1. Introduction . . . 1
   1.1. Sources and contributions . . . 1
2. Obtaining the artifacts . . . 1
   2.1. Using the Platform BOM . . . 2
3. What's new in version 5.1.x . . . 2
4. JPA Datastore . . . 2
   4.1. Setup and configuration . . . 3
      4.1.1. Common Datastore configuration options . . . 4
      4.1.2. `EntityManagerFactory` configuration . . . 5
      4.1.3. ORM platform configuration . . . 5
      4.1.4. ORM Dialect configuration . . . 6
      4.1.5. Builtin ORM dialects . . . 7
      4.1.6. Auto-flush mode . . . 7
      4.1.7. EntityManager lifecyle . . . 8
   4.2. Data model attributes naming conventions . . . 8
   4.3. JpaTarget . . . 11
      4.3.1. Using a JPA entity class as Query projection . . . 11
   4.4. Using the Bean introspection APIs to obtain a JPA entity Property model . . . 12
   4.5. JPA *Write options* . . . 13
   4.6. Relational expressions . . . 13
   4.7. Auto-generated ids . . . 14
   4.8. Transactions management . . . 14
   4.9. `JpaDatastore` API . . . 15
   4.10. Extending the JPA Datastore API . . . 16
   4.11. Expression resolvers . . . 16
      4.11.1. JPA Expression resolvers registration . . . 16
      4.11.2. Specific expression resolvers registration . . . 17
      4.11.3. Expression resolvers priority . . . 17
      4.11.4. Expression validation . . . 17
      4.11.5. JPA Datastore expressions . . . 17
      4.11.6. JPA Expression resolution context . . . 22
   4.12. Commodity factories . . . 22
5. Spring ecosystem integration . . . 25
   5.1. Integration with the Spring JPA infrastructure . . . 25
   5.2. JPA Datastore configuration . . . 26
      5.2.1. EntityManagerFactory . . . 26
      5.2.2. Multiple JPA Datastores configuration . . . 27
      5.2.3. Auto-flush mode . . . 28

5.2.4. Transactional JPA Datastore operations . . . 29
5.2.5. Primary mode . . . 30
5.2.6. JPA Datastore configuration properties . . . 30
5.2.7. Datastore extension and configuration using the Spring context . . . 31
5.3. Full JPA bean stack auto-configuration . . . 33
5.3.1. `DataSource` configuration . . . 34
5.3.2. `EntityManagerFactory` configuration . . . 34
5.3.3. Persistence provider (ORM) configuration . . . 35
5.3.4. Transaction manager configuration . . . 35
5.3.5. JPA Datastore configuration . . . 36
5.3.6. JPA configuration properties . . . 36
5.3.7. Multiple JPA bean stacks configuration . . . 37
6. Spring Boot integration . . . 38
6.1. JPA Datastore auto-configuration . . . 38
6.2. Full JPA beans stack auto-configuration . . . 39
6.3. Multiple JPA beans stack auto configuration . . . 41
6.3.1. JPA entities scan . . . 42
6.4. Spring Boot starters . . . 43
7. Loggers . . . 44
8. System requirements . . . 44
8.1. Java . . . 44
8.2. JPA . . . 44
8.3. Persistence providers . . . 44

Copyright © 2016-2018

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

1. Introduction

The Holon **JPA Datastore** is the *Java Persistence API* reference implementation of the Datastore API.

See the Datastore documentation for further information about the Datastore `API`.

The JPA `Datastore` uses the EntityManagerFactory API as a factory to obtain EntityManager API instances to perform data access and management operations through the standard JPA `JPQL` language and the `EntityManager` API operations.

The Holon Platform JPA modules and components require **JPA version 2.0 or higher**, so a JPA 2.0+ compliant persistence provider (ORM) is needed at runtime.

1.1. Sources and contributions

The Holon Platform **JPA Datastore** module source code is available from the GitHub repository https://github.com/holon-platform/holon-datastore-jpa.

See the repository `README` file for information about:

- The source code structure.
- How to build the module artifacts from sources.
- Where to find the code examples.
- How to contribute to the module development.

2. Obtaining the artifacts

The Holon Platform uses Maven for projects build and configuration. All the platform artifacts are published in the **Maven Central Repository**, so there is no need to explicitly declare additional repositories in your project `pom` file.

At the top of each *section* of this documentation you will find the Maven *coordinates* (group id, artifact id and version) to obtain the artifact(s) as a dependency for your project.

A **BOM (Bill Of Materials)** `pom` is provided to import the available dependencies for a specific version in your projects. The Maven coordinates for the core BOM are the following:

Maven coordinates:

```
<groupId>com.holon-platform.jpa</groupId>
<artifactId>holon-datastore-jpa-bom</artifactId>
<version>5.1.0</version>
```

The BOM can be imported in a Maven project in the following way:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.holon-platform.jpa</groupId>
      <artifactId>holon-datastore-jpa-bom</artifactId>
      <version>5.1.0</version>
      <strong><type>pom</type></strong>
      <strong><scope>import</scope></strong>
    </dependency>
  </dependencies>
</dependencyManagement>
```

2.1. Using the Platform BOM

The Holon Platform provides an **overall Maven BOM (Bill of Materials)** to easily obtain all the available platform artifacts.

See Obtain the platform artifacts for details.

3. What's new in version 5.1.x

- The new JPA support module can be used to integrate the JPA data model definition with the Holon Platform Property model. See JPA entity bean post processors.
- Full support for date and time **functions** and for `java.time.*` temporal types. See Datastore API temporal functions support.
- Technology-independent **transactions** support through the `Transactional` API. See Transactions management.
- Complete and deep revision and rewriting of the internal **JPQL composer engine**. This ensures more consistent operation resolution strategies, remarkable performance improvements and extensibility by design. See Extending the JPA Datastore API.
- Improved support for Apache OpenJPA and Datanucleus ORMs, along with the standard support for **Hibernate** and **Eclipselink**.

4. JPA Datastore

The `holon-datastore-jpa` artifact is the main entry point to use the JPA `Datastore` API implementation.

Maven coordinates:

```
<groupId>com.holon-platform.jpa</groupId>
<artifactId>holon-datastore-jpa</artifactId>
<version>5.1.0</version>
```

The JpaDatastore interface represents the **JPA Datastore** API implementation, extending the core `Datastore` API.

The `JpaDatastore` API, besides the standard `Datastore` API operations, provides methods to:

- Create and configure a `JpaDatastore` API instance, using the provided *builder*.
- Directly working with `JpaDatastore` managed *EntityManagers*, with a consistent and integrated `EntityManager` instance and related *persistence context* lifecycle management. See [JpaDatastoreAPI].

If you want to reach the goal of a ***complete abstraction*** from the persistence store technology and the persistence model, the core `Datastore` API interface should be used instead of the specific `JpaDatastore` API by your application code. This way, the concrete `Datastore` API implementation may be replaced by a different one at any time, without any change to the codebase.

4.1. Setup and configuration

To create a **JPA Datastore** instance, the `builder()` static method of the `JpaDatastore` API can be used to obtain the JPA Datastore *builder* API.

The JPA Datastore builder provides a `JpaDatastore` instance:

```
JpaDatastore datastore = JpaDatastore.builder() ①
    // Datastore configuration omitted
    .build();
```

① Obtain the JPA `Datastore` builder to configure and create a new JPA Datastore instance

But, as stated in the previous section, to reach the goal of a **complete abstraction** from the persistence store technology and the persistence model, the core `Datastore` API interface should be used by your application code, instead of the specific `JpaDatastore` API.

So you can simply obtain the JPA Datastore implementation as a core `Datastore` API implementation:

```
Datastore datastore = JpaDatastore.builder() ①
    // Datastore configuration omitted
    .build();
```

① Obtain the JPA Datastore builder to configure and create a new JPA Datastore instance and expose it as core `Datastore` API type

4.1.1. Common Datastore configuration options

The JPA Datastore builder API extends the core `Datastore` builder API, which provides common Datastore configuration settings as listed below.

Builder method	Arguments	Description
`dataContextId`	The *data context id* `String` value	Set the *data context id* to which the Datastore is bound. Can be used, for example, to declare configuration properties for multiple Datastore instances. See the Multiple Datastores configuration documentation section.
`traceEnabled`	`true` or `false`	Whether to enable Datastore operations *tracing*. When enabled, the JPA Datastore will log any JPQL operation performed using the Datastore API.
`configuration`	A `DatastoreConfigProperties` instance	Set the `DatastoreConfigProperties` type configuration property set instance to use in order to read the Datastore configuration properties. See the Datastore configuration documentation section for details. This configuration properties can be used as an alternative for the programmatic configuration performed with the previous builder methods.

Example of base JPA Datastore configuration:

```
Datastore datastore = JpaDatastore.builder()
    // DataSource configuration omitted
    .dataContextId("mydataContextId") ①
    .traceEnabled(true) ②
    .build();
```

① Set a *data context id* for the Datastore

② Activate operations *tracing* in log

The configuration properties can also be provided through an external configuration property

source, using the properties provided by the DatastoreConfigProperties property set.

For example, supposing to have a properties file named `datastore.properties` like this:

```
holon.datastore.trace=true
```

We can use it as configuration property source to enable the Datastore *tracing* mode:

```
Datastore datastore = JpaDatastore.builder()
    // DataSource configuration omitted
    .configuration(DatastoreConfigProperties.builder().withPropertySource(
"datastore.properties").build()) ①
    .build();
```

① Use the `datastore.properties` file as configuration property source

4.1.2. `EntityManagerFactory` configuration

The JPA Datastore implementation relies on the standard JPA `EntityManagerFactory` API to obtain and manage the `EntityManager` instances to be used to perform the Datastore operations.

The `EntityManagerFactory` reference is the only **required** JPA Datastore configuration attribute and can be provided using the JPA `Datastore` *builder* API.

```
EntityManagerFactory entityManagerFactory = createOrObtainEntityManagerFactory();

Datastore datastore = JpaDatastore.builder() //
    .entityManagerFactory(entityManagerFactory) ①
    .build();
```

① Set the `EntityManagerFactory` instance to use

4.1.3. ORM platform configuration

For some internal operations, the JPA Datastore needs to know the *ORM platform* which manages the concrete `EntityManagerFactory` (and accorrdingly the `EntityManager`) instances (for example *Hibernate*, *Eclipselink* and so on). The *ORM platform* is used, for example, to auto-detect the ORM dialect to use if not directly specified at JPA Datastore configuration time.

The *ORM platform* can be explicitly specified using the JPA Datastore `platform` builder method, using the ORMPlatform enumeration.

```
EntityManagerFactory entityManagerFactory = createOrObtainEntityManagerFactory();

Datastore datastore = JpaDatastore.builder() //
    .entityManagerFactory(entityManagerFactory) //
    .platform(ORMPlatform.HIBERNATE) ①
    .build();
```

① Set `Hibernate` as ORM platform

ORM platform auto detection:

When the *ORM platform* is not explicitly specified, the JPA Datastore will **auto-detect** it using the configured `EntityManagerFactory`.

4.1.4. ORM Dialect configuration

To ensure operations consistency and efficiency, the JPA Datastore uses the ORMDialect API abstraction in order to resolve each ORM platform specificity and JPQL language difference.

Normally, the `ORMDialect` implementation to use is **auto-detected** by the JPA Datastore, relying on the *ORM platform* to which the `EntityManagerFactory` instance is bound, as described in the previous section.

The `ORMDialect` implementation to use can be also explicitly configured using the JPA Datastore builder. This can be done in two ways:

1. `ORMDialect` configuration using the JPA Datastore builder:

The ORM dialect to use can be directly configured using the JPA Datastore builder, either using the `dialect(ORMDialect dialect)` or the `dialect(String dialectClassName)` method.

The `ORMDialect` API provides a static methods to obtain the ORM dialect for a specific ORM platform or to directly obtain a specific dialect implementation within the available ones.

```
EntityManagerFactory entityManagerFactory = createOrObtainEntityManagerFactory();

Datastore datastore = JpaDatastore.builder() //
    .entityManagerFactory(entityManagerFactory) //
    .dialect(ORMDialect.hibernate()) ①
    .dialect("com.holonplatform.datastore.jpa.dialect.HibernateDialect") ②
    .build();
```

① Configure the ORM dialect using a specific dialect implementation

② Configure the ORM dialect using the dialect class name

1. `ORMDialect` configuration using a configuration property:

The ORM dialect to use can be also configured using the default `holon.datastore.dialect` Datastore configuration property, available from the DatastoreConfigProperties property set.

The **fully qualified dialect class name** must be provided as property value.

For example, supposing to have a properties file named `datastore.properties` like this:

```
holon.datastore.dialect=com.holonplatform.datastore.jpa.dialect.HibernateDialect
holon.datastore.trace=true
```

We can use it as configuration property source to configure the JPA Datastore dialect:

```
EntityManagerFactory entityManagerFactory = createOrObtainEntityManagerFactory();

Datastore datastore = JpaDatastore.builder() //
    .entityManagerFactory(entityManagerFactory) //
    .configuration(DatastoreConfigProperties.builder().withPropertySource(
"datastore.properties").build()) ①
    .build();
```

① Use the `datastore.properties` file as configuration property source

4.1.5. Builtin ORM dialects

The Holon JPA Datastore module provides a set of builtin ORM dialects for the most common ORM platforms. The currently available ORM dialect implementations are:

ORM platform	Dialect class
Hibernate	HibernateDialect
Eclipselink	EclipselinkDialect
Apache OpenJPA	OpenJPADialect
Datanucleus	DatanucleusDialect

4.1.6. Auto-flush mode

The JPA `Datastore` API can be configured to automatically *flush* the persistence context into the concrete persistence source when a data management operation of the `Datastore` API (such as *save* or *delete*) is performed.

When the auto-flush mode is enabled, the `EntityManager.flush()` operation is invoked just after the `Datastore` API operation execution, forcing the persistence context synchronization to the underlying database.

```
EntityManagerFactory entityManagerFactory = createOrObtainEntityManagerFactory();

Datastore datastore = JpaDatastore.builder() //
    .entityManagerFactory(entityManagerFactory) //
    .autoFlush(true) ①
    .build();
```

① Enable the auto-flush mode for the JPA `Datastore` API instance

4.1.7. EntityManager lifecyle

The default JPA `Datastore` strategy to handle the `EntityManager` lifecycle is defined as follows:

- For each `Datastore` API operation, a new `EntityManager` instance is obtained from the configured `EntityManagerFactory`, using the `createEntityManager()` method, before the operation execution. The `EntityManager` instance is then *closed*, using the `EntityManager.close()` method, right after the operation execution.
- If the `Datastore` API operation is executed within a Datastore managed *transaction* (see Transactions management), the `EntityManager` instance is created at transaction start, shared between any transaction operation and closed only at transaction end.

To customize the `EntityManager` lifecycle, the EntityManagerInitializer and EntityManagerFinalizer interfaces can be used to control the `EntityManager` instance creation and/or the `EntityManager` instance finalization for any JPA `Datastore` API operation.

The `EntityManagerInitializer` and `EntityManagerFinalizer` to use can be configured using the JPA `Datastore` *builder* API.

```
EntityManagerFactory entityManagerFactory = createOrObtainEntityManagerFactory();

Datastore datastore = JpaDatastore.builder() //
    .entityManagerFactory(entityManagerFactory) //
    .entityManagerInitializer(emf -> emf.createEntityManager()) ①
    .entityManagerFinalizer(em -> em.close()) ②
    .build();
```

① Configure a custom `EntityManagerInitializer`

② Configure a custom `EntityManagerFinalizer`

4.2. Data model attributes naming conventions

Since the JPA `Datastore` implementation relies on the JPA architecture and APIs to manage the concrete data model and persistence source, the **JPA data model definition**, i.e. the relational schema mapping through Java objects using *entity* beans representations, is used to resolve the `Datastore` API operations meta-language into valid JPA operation definitions, for example using the *JPQL* language.

This means that the JPA data model (the JPA *entity* beans definitions) has to be defined and made available to the `EntityManagerFactory` used by the JPA `Datastore` to allow the `Datastore` API operations execution, for example using a `persistence.xml` file. This is the convention used by JPA, and it does not directly concern the JPA `Datastore` implementation. See the JPA reference documentation for details.

For this reason, the JPA `Datastore` implementation relies on the following conventions regarding *DataTarget*s and *Path*s definitions:

- The **DataTarget** *name* is interpreted as the **JPA *entity* name** (i.e. the simple *entity* class name or the name specified using the `name` attribute of the `javax.persistence.Entity` annotation).
- The **Path** *name* is interpreted as a **JPA *entity* attribute name**, supporting nested classes through the conventional *dot* notation (e.g. `address.city`).

For example, given a *table* definition as follows:

```
create table test (
  code numeric(20) primary key,
  text varchar(100)
)
```

And the corresponding JPA *entity* mapping definition:

```
@Entity
public class Test {

  @Id
  @Column(name = "code")
  private Long id;

  @Column(name = "text")
  private String value;

  // getters and setters omitted

}
```

According to the naming conventions described above, a *property model* and a *DataTarget* for the example JPA *entity* can be defined as follows:

```
public static final NumericProperty<Long> ID = NumericProperty.longType("id"); ①
public static final StringProperty VALUE = StringProperty.create("value"); ②

public static final PropertySet<?> TEST = PropertySet.of(ID, VALUE);

public static final DataTarget<?> TARGET = DataTarget.named("Test"); ③
```

① Path property definition for the `id` JPA *entity* attribute name

② Path property definition for the `value` JPA *entity* attribute name

③ DataTarget definition for the `Test` JPA *entity* name

If the JPA *entity* name is specified using the `name` attribute of the `javax.persistence.Entity` annotation, that name has to be used as `DataTarget` name.

For example, if `myName` is specified as JPA *entity* name:

```
@Entity(name = "myName")
```

The `DataTarget` definition will be:

```
DataTarget<?> TARGET = DataTarget.named("myName");
```

See JpaTarget to learn how to use the JPA *entity* class definition rather than the JPA entity *name* to declare a `DataTarget`.

Now the *property model* definition can be used with the `Datastore` API operations in an abstract and implementation-independent way. For example:

```
Datastore datastore = getJpaDatastore();

Stream<Long> ids = datastore.query().target(TARGET).filter(ID.gt(0L)).sort(VALUE.asc(
)).stream(ID); ①
Stream<PropertyBox> results = datastore.query().target(TARGET).filter(ID.gt(0L)).sort
(VALUE.asc()).stream(TEST); ②

PropertyBox valueToSave = PropertyBox.builder(TEST).set(ID, 1L).set(VALUE, "test")
.build();
datastore.save(TARGET, valueToSave); ③
```

① Perform a query and obtain a stream of `ID` property values

② Perform a query and obtain a stream of `PropertyBox` values using the `TEST` property set, which corresponds to the `Test` JPA entity attributes definitions

③ Insert or update a value (which corresponds to a JPA `Test` instance) using the *save* operation

See Using the Bean introspection APIs to obtain a JPA entity Property model to learn how to create a *property model* definition for a JPA *entity* class using the Holon Platform Bean introspection APIs.

4.3. JpaTarget

The JpaTarget interface can be used to define a `DataTarget` using a **JPA *entity* class definition** rather than the entity *name*.

The entity name used in persistence operations will be the entity class simple name or the name specified using the `name` attribute of the `javax.persistence.Entity` annotation, if available.

Fo example, for the `Test` JPA entity class defined in the example of the previous section, a `JpaTarget` can be created in the following way:

```
JpaTarget<Test> TARGET = JpaTarget.of(Test.class);
```

The `JpaTarget` path type will be the JPA *entity* class. A `JpaTarget` definition can be used with `Datastore` API operations just like any other `DataTarget` definition.

4.3.1. Using a JPA entity class as Query projection

By default, the `Datastore` API relies on the `PropertyBox` type to represent and handle a set of data model attribute values. For this reason, the `PropertyBox` type is the default type used in *query projections* to obtain a set of *property* values.

The `JpaTarget` definition can be also used as *query projection*, to obtain results as JPA *entity* class instances, instead of `PropertyBox` type instances.

If you want to reach the goal of a ***complete abstraction*** from the persistence store technology and the persistence model, the `PropertyBox` type should be used to represent a set of data model attributes values. This way, the concrete `Datastore` API implementation may be replaced by a different one at any time, without any change to the codebase.

```
public static final JpaTarget<Test> JPA_TARGET = JpaTarget.of(Test.class); ①

void query() {
  Datastore datastore = getJpaDatastore();

  List<Test> resultEntities = datastore.query(JPA_TARGET).filter(ID.gt(0L)).list
(JPA_TARGET); ②
}
```

① `JpaTarget` definition using the `Test` JPA *entity* class

② Perform a query and obtain the results as `Test` JPA *entity* instances, using the `JpaTarget` as query

projection

4.4. Using the Bean introspection APIs to obtain a JPA entity Property model

The Holon Platform Bean introspection API can be used to obtain a *property model* definition from any Java Bean class, including a JPA *entity* class.

The `BeanPropertySet` obtained using the bean introspection API can be used to refer to the JPA *entity* attributes as *properties* for `Datastore` API operations definition and execution.

```
static //
@Entity public class Test {

  public static final BeanPropertySet<Test> PROPERTIES = BeanPropertySet.create(Test
.class); ①

  public static final DataTarget<Test> TARGET = JpaTarget.of(Test.class); ②

  @Id
  private Long id;

  private String value;

  // getters and setters omitted

}

void operations() {
  Datastore datastore = getJpaDatastore();

  PropertyBox value = PropertyBox.builder(Test.PROPERTIES).set(Test.PROPERTIES
.property("id"), 1L)
      .set(Test.PROPERTIES.property("value"), "test").build(); ③
  datastore.save(Test.TARGET, value);

  List<PropertyBox> values = datastore.query(Test.TARGET).filter(Test.PROPERTIES
.property("id").gt(0L))
      .sort(Test.PROPERTIES.property("value").desc()).list(Test.PROPERTIES); ④
}
```

① Create a `BeanPropertySet` for the `Test` JPA *entity* class

② Create a `DataTarget` for the `Test` JPA *entity* class

③ The `BeanPropertySet` can be used to obtain the *property* definitions which corresponds to the JPA *entity* class attributes

④ Perform a query using the property model provided by the `BeanPropertySet`

The JPA `Datastore` API implementation automatically registers the **JPA bean post processors** provided by the Holon Platform JPA Module, used to inspect standard JPA entity annotations (for example `@Id`, `@EmbeddedId` and `@Transient`) and configure the bean properties accordingly.

See the Holon Platform JPA Module documentation for details.

4.5. JPA *Write options*

The data manipulation `Datastore` API operations supports a set of **write options** to provide additional operation configuration attributes or behavior specifications.

Besides the standard write options, the JPA `Datastore` API implementation supports an additional FLUSH, available from the JpaWriteOption enumeration.

This *write option* can be provided a `Datastore` API data manipulation operation **to synchronize the persistence context to the underlying database** right after the operation execution, automatically invoking the `EntityManager.flush()` method on the current `EntityManager` instance.

```
PropertyBox valueToSave = PropertyBox.builder(TEST).set(ID, 1L).set(VALUE, "test")
.build();

getJpaDatastore().save(TARGET, valueToSave, JpaWriteOption.FLUSH); ①
```

① The `FLUSH` write option is provided to synchronize the persistence context to the underlying database after the operation execution

4.6. Relational expressions

As relational Datastore, the JPA `Datastore` API supports core *relational expressions* for data access and manipulation:

1. Sub-query:

The SubQuery interface can be used to represent a *sub-query*, which can be used in a query definition to express query restrictions (filters) that involve a sub-query as filter operand.

See the core sub query expression documentation for further information on sub query expressions.

2. Alias and Joins:

The RelationalTarget interface can be used to declare **alias** and **joins** for a `DataTarget` expression.

See the core alias and joins documentation for further information on alias and join expressions.

The supported *join types* are dependent from the JPA API specification version in use. For example, the **left** join type is only supported by the JPA specification version 2.1 or above.

4.7. Auto-generated ids

The JPA Datastore API supports the retrieving of auto-generated id column values, if supported by the underlying JDBC driver and ORM platform.

The auto-generated id values can be obtained from the `OperationResult` object, returned by Datastore data manipulation operations, through the `getInsertedKeys()` and related methods.

```
Datastore datastore = getJpaDatastore();

PropertyBox value = PropertyBox.builder(TEST).set(VALUE, "test").build();

OperationResult result = datastore.insert(TARGET, value, JpaWriteOption.FLUSH); ①

Optional<Long> idValue = result.getInsertedKey(ID); ②
```

① Perform an *insert* operation, using the `FLUSH` write option to *flush* the persistence context to the underlyng database after operation execution to ensure the key auto-generation is triggered

② Get the auto-generated key value which corresponds to the `ID` property definition, if available

The default **BRING_BACK_GENERATED_IDS** `WriteOption` can be provided to the `Datastore` API operation to bring back any auto-generated key value into the `PropertyBox` instance which was the subject of the operation, if a corresponding `PathProperty` (using the path name) is available in the `PropertyBox` property set.

```
Datastore datastore = getJpaDatastore();

PropertyBox value = PropertyBox.builder(TEST).set(VALUE, "test").build();

OperationResult result = datastore.insert(TARGET, value, JpaWriteOption.FLUSH,
    DefaultWriteOption.BRING_BACK_GENERATED_IDS); ①

Long idValue = value.getValue(ID); ②
```

① Perform an *insert* operation using the **BRING_BACK_GENERATED_IDS** write option and the **FLUSH** write option to ensure the id auto-generation is triggered

② The `ID` property value of the inserted `PropertyBox` value is updated with the auto-generated value, if available

4.8. Transactions management

The JPA `Datastore` API implementation is transactional, so it supports **transactions** management through the Transactional API, which can be used to manage transactions at a higher level, in an abstract and implementation-independent way.

See the Transactional Datastore documentation section for information on transactions management with the `Transactional` API.

```
final Datastore datastore = getJpaDatastore(); // build or obtain a JPA Datastore

datastore.requireTransactional().withTransaction(tx -> { ①
  PropertyBox value = buildPropertyBoxValue();
  datastore.save(TARGET, value);

  tx.commit(); ②
});

OperationResult result = datastore.requireTransactional().withTransaction(tx -> { ③

  PropertyBox value = buildPropertyBoxValue();
  return datastore.save(TARGET, value);

}, TransactionConfiguration.withAutoCommit()); ④
```

① Obtain the `Transactional` API to execute one or more `Datastore` operation within a transaction

② Commit the transaction

③ Obtain the `Transactional` API to execute the `Datastore` operation within a transaction and return a value

④ The transaction is configured with the *auto commit* mode, this way the transaction is automatically committed at the transactional operation end if no error occurred

4.9. `JpaDatastore` API

The specialized JpaDatastore API, which extends the standard `Datastore` API, makes available an additional methods through the EntityManagerHandler interface to **execute operations using a Datastore managed EntityManager**.

The JPA Datastore will take care of the `EntityManager` instance provision and finalization. See the EntityManager lifecyle section for information about the `EntityManager` lifecycle management in the JPA `Datastore` API implementation.

```
JpaDatastore datastore = getJpaDatastore(); // build or obtain a JpaDatastore

datastore.withEntityManager(em -> { ①
  em.persist(new Test());
});

Test result = datastore.withEntityManager(em -> { ②
  return em.find(Test.class, 1);
});
```

① Execute an operation using a JPA Datastore managed `EntityManager` instance

② Execute an operation using a JPA Datastore managed `EntityManager` instance and return a result

If you want to reach the goal of a ***complete abstraction*** from the persistence store technology and the persistence model, the core `Datastore` API interface should be used instead of the specific `JpaDatastore` API by your application code. This way, the concrete `Datastore` API implementation may be replaced by a different one at any time, without any change to the codebase.

4.10. Extending the JPA Datastore API

4.11. Expression resolvers

The `Datastore` API can be extended using the `ExpressionResolver` API, to add new expression resolution strategies, modify existing ones and to handle new `Expression` types.

See the Datastore API extension documentation section for details.

4.11.1. JPA Expression resolvers registration

A new `ExpressionResolver` can be registered in the JPA `Datastore` API in two ways:

1. Using the JPA `Datastore` API instance:

An `ExpressionResolver` can be registered either using the `Datastore` *builder* API at Datastore configuration time:

```
Datastore datastore = JpaDatastore.builder() //
    .withExpressionResolver(new MyExpressionResolver()) ①
    .build();
```

① Register and new `ExpressionResolver`

Or using the `Datastore` API itself, which extends the `ExpressionResolverSupport` API:

```
datastore.addExpressionResolver(new MyExpressionResolver()); ①
```

① Register and new `ExpressionResolver`

2. Automatic registration using *Java service extensions*:

The JPA Datastore supports `ExpressionResolver` automatic registration using the JpaDatastoreExpressionResolver base type and the default *Java service extensions* modality.

To automatically register an `ExpressionResolver` this way, a class implementing `JpaDatastoreExpressionResolver` has to be created and its fully qualified name must be specified in a file named `com.holonplatform.datastore.jpa.config.JpaDatastoreExpressionResolver`, placed in the `META-INF/services` folder in classpath.

When this registration method is used, the expression resolvers defined this way will be registered

for **any JPA `Datastore` API instance**.

4.11.2. Specific expression resolvers registration

All the default `Datastore` API operation representations supports **operation specific** expression resolvers registration, through the `ExpressionResolverSupport` API.

An `ExpressionResolver` registered for a specific `Datastore` API operation execution will be available only for the execution of that operation, and will be ignored by any other `Datastore` API operation.

For example, to register an expression resolver only for a single `Query` execution, the `Query` builder API can be used:

```
long result = datastore.query().target(DataTarget.named("Test")) //
    .withExpressionResolver(new MyExpressionResolver()) ①
    .count();
```

① Register an expression resolver only for the specific `Query` operation definition

4.11.3. Expression resolvers priority

According to the standard convention, the `javax.annotation.Priority` annotation can be used on `ExpressionResolver` classes to indicate in what order the expression resolvers bound to the same *type resolution pair* (i.e. the expression type handled by a resolver and the target expression type into which it will be resolved) must be applied.

The less is the `javax.annotation.Priority` number assigned to a resolver, the higher will be it's priority order.

All the default JPA Datastore expression resolvers have the *minimum* priority order, allowing to override their behavior and resolution strategies with custom expression resolvers with a higher assigned priority order (i.e. a priority number less then `Integer.MAX_VALUE`).

4.11.4. Expression validation

The internal JPA Datastore *JPQL composer engine* will perform **validation** on any `Expression` instance to resolve and each corresponding resolved `Expression` instance, using the default expression `validate()` method.

So the `validate()` method can be used to implement custom expression validation logic and throw an `InvalidExpressionException` when validation fails.

4.11.5. JPA Datastore expressions

Besides the standard Datastore API expressions, such as `DataTarget`, `QueryFilter` and `QuerySort`, which can be used to extend the `Datastore` API with new expression implementations and new resolution strategies, the JPA `Datastore` API can be extended using a set of **specific JPQL resolution expressions**, used by the internal *JPQL composer engine* to implement the resolution and composition strategy to obtain JPQL statements from the `Datastore` API meta-language expressions.

These JPQL expression are available from the `com.holonplatform.datastore.jpa.jpql.expression` package of the `holon-datastore-jpa` artifact.

The JPQLExpression is the expression which represents an *JPQL statement part*, used to compose the actual JPQL statement which will be executed using the `EntityManager` API. So this is the *final* target expression used by the *JPQL composer engine* to obtain a JPQL statement part from other, more abstract, expression types.

The `JPQLExpression` type can be used to directly resolve an abstract `Datastore` API expression into a JPQL statement part.

For example, supposing to have a `KeyIs` class which represents a `QueryFilter` expression type to represent the expression *"the `key` JPA attribute value is equal to a given Long type value"*:

```
class KeyIs implements QueryFilter {

  private final Long value;

  public KeyIs(Long value) {
    this.value = value;
  }

  public Long getValue() {
    return value;
  }

  @Override
  public void validate() throws InvalidExpressionException {
    if (value == null) {
      throw new InvalidExpressionException("Kay value must be not null");
    }
  }

}
```

We want to create an `ExpressionResolver` class to resolve the `KeyIs` expression directly into a **JPQL WHERE statemet part**, using the `JPQLExpression` type. Using the convenience `create` method of the `ExpressionResolver` API, we can do it in the following way:

```
final NumericProperty<Long> KEY = NumericProperty.create("key", long.class);

final ExpressionResolver<KeyIs, JPQLExpression> keyIsResolver = ExpressionResolver
.create( //
    KeyIs.class, ①
    JPQLExpression.class, ②
    (keyIs, context) -> {
      String path = JPQLResolutionContext.isJPQLResolutionContext(context)
          .flatMap(ctx -> ctx.isStatementCompositionContext()).flatMap(ctx -> ctx
.getAliasOrRoot(KEY))
          .map(alias -> alias + ".key").orElse("key");
      return Optional.of(JPQLExpression.create(path + " = " + keyIs.getValue()));
    }); ③
```

① Expression type to resolve

② Target expression type

③ Expression resolution logic: since we resolve the `KeyIs` expression directly into `JPQLExpression` type, the JPQL WHERE clause part is provided

In this example, the `JPQLResolutionContext` resolution context extension is used to obtain th *alias* of property path: see JPA Expression resolution context for details.

After the `ExpressionResolver` is registered in the `Datastore` API, the new `KeyIs` expression can be used in the `Datastore` API operations which support the `QueryFilter` expression type just like any other filter expression. For example, in a `Query` expression:

```
Datastore datastore = JpaDatastore.builder().withExpressionResolver(keyIsResolver) ①
    .build();

Query query = datastore.query().filter(new KeyIs(1L)); ②
```

① Register the new expression resolver

② Use the `KeyIs` expression in a query definition

The *JPQL composer engine* will translate the given `KeyIs` expression in the JPQL WHERE statement part `(alias).key = 1`, using the previously defined expression resolver.

Other expression types are used to represent elements of a query or a Datastore operation, to be resolved into a *final* `JPQLExpression` type. These expression types often represent an *intermediate* expression type, between the highest abstract layer (i.e. an expression of the `Datastore` API meta-language) and the final JPQL statement part representation (i.e. the `JPQLExpression` type).

Some examples are:

- JPQLLiteral to represent a *literal* value.
- JPQLParameter to represent a *parameter* value.

- JPQLFunction to represent a *JPQL function*.
- JPQLProjection to represent a *JPQL SELECT* projection.
- JPQLStatement to represent a full *JPQL statement* with parameters support.
- JPQLQueryDefinition and JPQLQuery to represent the definition of a *JPQL query* and the query statement representation with the `JPQLResultConverter` to use to covert the query results in the expected type.

For example, let's see how the `JPQLFunction` expression can be used as an *intermediate* JPQL expression type to define a new JPQL *function*.

We want to define a **Trim** function to map the JPQL `TRIM(both from x)` function.

We will use the `QueryFunction` expression type to represent the function, since it is the default *function* expression representation of the `Datastore` API.

```
public class Trim implements QueryFunction<String, String> {

  private final TypedExpression<String> expression; ①

  public Trim(TypedExpression<String> expression) {
    super();
    this.expression = expression;
  }

  @Override
  public Class<? extends String> getType() {
    return String.class;
  }

  @Override
  public void validate() throws InvalidExpressionException { ②
    if (expression == null) {
      throw new InvalidExpressionException("Null function expression");
    }

  }

  @Override
  public List<TypedExpression<? extends String>> getExpressionArguments() { ③
    return Collections.singletonList(expression);
  }

}
```

① The expression to be *trimmed*

② Validate the argument

③ The unique function *argument* will be the expression to trim

Now we create an expression resolver to resolve the `QueryFunction` expression into a **`JPQLFunction` intermediate JPQL expression type**:

```
public class TrimResolver implements ExpressionResolver<Trim, JPQLFunction> {

  @Override
  public Optional<JPQLFunction> resolve(Trim expression, ResolutionContext context)
      throws InvalidExpressionException {
    return Optional.of(JPQLFunction.create(args -> { ①
      StringBuilder sb = new StringBuilder();
      sb.append("TRIM(both from ");
      sb.append(args.get(0));
      sb.append(")");
      return sb.toString();
    }));
  }

  @Override
  public Class<? extends Trim> getExpressionType() {
    return Trim.class;
  }

  @Override
  public Class<? extends JPQLFunction> getResolvedType() {
    return JPQLFunction.class;
  }

}
```

① Use the `JPQLFunction create` method to provide the JPQL function representation using the function arguments. The function arguments are already provided by the JPQL composition engine as serialized JPQL expression tokens, according to the actual arguments expression types

Now the function can be registered in the `Datastore` API and used the same way as any other `QueryFunction` expression implementation.

```
final StringProperty STR = StringProperty.create("stringAttribute");
final DataTarget<?> TARGET = DataTarget.named("Test");

Datastore datastore = JpaDatastore.builder() //
    .withExpressionResolver(new TrimResolver()) ①
    .build();

Stream<String> trimmedValues = datastore.query(TARGET).stream(new Trim(STR)); ②
```

① Register the `Trim` function expression resolver in Datastore

② The `Trim` function is used to provide the *trimmed* `STR` property value as query projection

4.11.6. JPA Expression resolution context

The JPA `Datastore` API makes available an extension of the standard expression `ResolutionContext`, to provide a set of configuration attributes and JPQL resolution context specific operations. This resolution context extension is represented by the JPQLResolutionContext API.

When used within the JPA `Datastore` API, the `ResolutionContext` provided to the registered expression resolvers is a `JPQLResolutionContext` for any JPA `Datastore` API standard operation. To obtain a `JPQLResolutionContext`, the `isJPQLResolutionContext` method can be used.

The JPQLStatementResolutionContext API is a further context extension which provides methods related to JPQL *statements* composition. It can be obtained from a `JPQLResolutionContext` through the `isStatementCompositionContext()` method.

The *statement* resolution context provides methods, for example, to obtain the **alias** for a property path used in a query.

```
@Override
public Optional<JPQLExpression> resolve(JPQLExpression expression, ResolutionContext
context) ①
    throws InvalidExpressionException {

  JPQLResolutionContext.isJPQLResolutionContext(context).ifPresent(ctx -> { ②
    ORMPlatform paltform = ctx.getORMPlatform().orElse(null); ③

    ctx.isStatementCompositionContext().ifPresent(sctx -> { ④
      Optional<String> alias = sctx.getAliasOrRoot(A_PROPERTY); ⑤
    });
  });

  return Optional.empty();
}
```

① Let's suppose we are implementing an expression resolver resolution method

② Check and obtain the current `ResolutionContext` as a `JPQLResolutionContext`

③ As an example, get the current `ORMPlatform` is requested

④ If the context is a JPQL *statement* resolution context, it can be obtained as a `JPQLStatementResolutionContext`

⑤ Get the *alias* assigned to the `A_PROPERTY` property in the current JPQL statement, if available

4.12. Commodity factories

The JPA `Datastore` API supports *Datastore commodities* registration using the JpaDatastoreCommodityFactory type.

See the Datastore commodities definition and registration documentation section to learn how the *Datastore commodity* architecture can be used to provide extensions to the default `Datastore` API.

The `JpaDatastoreCommodityFactory` type provides a specialized JpaDatastoreCommodityContext API as Datastore commodity context, to make available a set of JPA `Datastore` specific configuration attributes and references, for example:

- The `EntityManagerFactory` bound the JPA Datastore.
- The configured (or auto-detected) ORM platform.
- The available *expression resolvers*
- A set of APIs and configuration methods available from the *JPQL composer engine*.

Furthermore, it makes available some API methods to invoke some JPA Datastore operations, such as the `withEntityManager(EntityManagerOperation<R> operation)` method to execute an operation using a Datastore managed `EntityManager` instance or methods to inspect and create other Datastore *commodities*.

```
interface MyCommodity extends DatastoreCommodity { ①

  CriteriaBuilder getCriteriaBuilder();

}

class MyCommodityImpl implements MyCommodity { ②

  private final EntityManagerFactory entityManagerFactory;

  public MyCommodityImpl(EntityManagerFactory entityManagerFactory) {
    super();
    this.entityManagerFactory = entityManagerFactory;
  }

  @Override
  public CriteriaBuilder getCriteriaBuilder() {
    return entityManagerFactory.getCriteriaBuilder();
  }

}

class MyCommodityFactory implements JpaDatastoreCommodityFactory<MyCommodity> { ③

  @Override
  public Class<? extends MyCommodity> getCommodityType() {
    return MyCommodity.class;
  }

  @Override
  public MyCommodity createCommodity(JpaDatastoreCommodityContext context)
      throws CommodityConfigurationException {
    EntityManagerFactory entityManagerFactory = context.getEntityManagerFactory();
    return new MyCommodityImpl(entityManagerFactory);
  }

}
```

① Datastore commodity API

② Commodity implementation

③ Commodity factory implementation

A Datastore commodity factory class which extends the `JpaDatastoreCommodityFactory` interface can be registered in a JPA `Datastore` in two ways:

1. Direct registration using the JPA `Datastore` API builder:

The JPA `Datastore` API supports `DatastoreCommodityFactory` automatic registration using the JpaDatastoreCommodityFactory type using a *Java service extensions* based registration method.

```
Datastore datastore = JpaDatastore.builder() //
    .withCommodity(new MyCommodityFactory()) ①
    .build();
```

① Register the `MyCommodityFactory` commodity factory in given JPA `Datastore` implementation

2. Automatic registration using the *Java service extensions*:

To automatically register an commodity factory using the standard *Java service extensions* based method, a class implementing `JpaDatastoreCommodityFactory` has to be created and its qualified full name must be specified in a file named `com.holonplatform.datastore.jpa.config.JpaDatastoreCommodityFactory`, placed in the `META-INF/services` folder of the classpath.

When this registration method is used, the commodity factories defined this way will be registered for **any JPA `Datastore` API instance**.

5. Spring ecosystem integration

The `holon-datastore-jpa-spring` artifact provides integration with the Spring framework for the JPA `Datastore` API.

Maven coordinates:

```
<groupId>com.holon-platform.jpa</groupId>
<artifactId>holon-datastore-jpa-spring</artifactId>
<version>5.1.0</version>
```

5.1. Integration with the Spring JPA infrastructure

When a JPA `Datastore` API is configured as a Spring bean using the facilities described in this documentation section, a consistent integration with the Spring JPA infrastructure is automatically provided.

In particular, the `EntityManager` instance used for the JPA `Datastore` API operations is obtained as a Spring **transactional `EntityManager` proxy**, using the `EntityManagerFactory` configured for the JPA `Datastore` implementation.

This allows to seamlessy integrate with the Spring transaction synchronization architecture, since the *shared* `EntityManager` proxy provides the following behavior: it will delegate all calls to the current transactional `EntityManager`, if any; otherwise it will fall back to a newly created `EntityManager` per operation.

This way, the JPA `Datastore` API can be seamlessly used along with the Spring transactions management conventions, for example when using a Spring `PlatformTransactionManager` or the `@Transactional` annotation.

5.2. JPA Datastore configuration

The EnableJpaDatastore annotation can be used on Spring configuration classes to enable automatic JPA `Datastore` beans configuration.

By default, the JPA Datastore bean name configured with the `@EnableJpaDatastore` annotation will be `jpaDatastore`.

5.2.1. EntityManagerFactory

The `EntityManagerFactory` bean to be used to configure the JPA Datastore API is obtained as follows:

- If the `entityManagerFactoryReference` attribute of the `@EnableJpaDatastore` annotation is setted, the provided **bean definition name** will be used as the `EntityManagerFactory` bean definition name to use.

```
@EnableJpaDatastore(entityManagerFactoryReference = "myEntityManagerFactory") ①
@Configuration
class Config {

  @Bean(name = "myEntityManagerFactory")
  public FactoryBean<EntityManagerFactory> entityManagerFactory(DataSource dataSource)
{
    LocalContainerEntityManagerFactoryBean emf = new
LocalContainerEntityManagerFactoryBean();
    emf.setDataSource(dataSource);
    emf.setJpaVendorAdapter(new HibernateJpaVendorAdapter());
    emf.setPackagesToScan("com.example.entities");
    return emf;
  }

}

@Autowired
Datastore datastore; ②
```

① Provide the `EntityManagerFactory` bean definition name to use with the JPA Datastore

② The JPA Datastore is configured and made available, for example, using dependency injection

- Otherwise, the **default** `entityManagerFactory` bean definition name will be used to lookup for the `EntityManagerFactory` bean to use.

```
@EnableJpaDatastore
@Configuration
class Config {

  @Bean
  public FactoryBean<EntityManagerFactory> entityManagerFactory(DataSource dataSource)
{ ①
    LocalContainerEntityManagerFactoryBean emf = new
LocalContainerEntityManagerFactoryBean();
    emf.setDataSource(dataSource);
    emf.setJpaVendorAdapter(new HibernateJpaVendorAdapter());
    emf.setPackagesToScan("com.example.entities");
    return emf;
  }

}

@Autowired
Datastore datastore; ②
```

① The default `entityManagerFactory` bean definition name is used for the `EntityManagerFactory` bean definition

② The JPA Datastore is configured and made available, for example, using dependency injection

If a `EntityManagerFactory` bean definition whith the required name is not present in Spring context, an initialization error is thrown.

See the Full JPA bean stack auto-configuration section to learn how to auto-configure a complete JPA bean stack in the Spring context.

5.2.2. Multiple JPA Datastores configuration

When more than one JPA Datastore bean has to be configured using the `@EnableJpaDatastore` annotation, the `dataContextId` attribute can be used to assign a different **data context id** to each JPA Datastore bean definition, in order to:

- Provide different sets of configuration properties using the same Spring environment.
- Provide a default *name pattern matching strategy* with the `EntityManagerFactory` bean definition to use for each JPA Datastore to configure: if not directly specified with the `entityManagerFactoryReference` attribute, the `EntityManagerFactory` bean definition to use for each JPA Datastore will be detected in Spring context using the bean name pattern: `entityManagerFactory_{datacontextid}` where `{datacontextid}` is equal to the `dataContextId` attribute of the `@EnableJpaDatastore` annotation.

When a *data context id* is defined, a Spring **qualifier** named the same as the *data context id* will be associated to the generated JPA `Datastore` bean definitions, and such qualifier can be later used to obtain the right `Datastore` instance through dependency injection.

Furthermore, the JPA Datastore bean definitions will be named using the *data context id* as suffix, according to the name pattern: `jpaDatastore_{datacontextid}`.

```
@Configuration class Config {

  @Configuration
  @EnableJpaDatastore(dataContextId = "one") ①
  static class Config1 {

    @Bean(name = "entityManagerFactory_one")
    public EntityManagerFactory entityManagerFactory() {
      return buildEntityManagerFactory();
    }

  }

  @Configuration
  @EnableJpaDatastore(dataContextId = "two") ②
  static class Config2 {

    @Bean(name = "entityManagerFactory_two")
    public EntityManagerFactory entityManagerFactory() {
      return buildEntityManagerFactory();
    }

  }

}

@Autowired
@Qualifier("one")
Datastore datastore1; ③

@Autowired
@Qualifier("two")
Datastore datastore2;
```

① Configure the first JPA Datastore using `one` as *data context id*: by default the bean named `entityManagerFactory_one` will be used as `EntityManagerFactory`

② Configure the first JPA Datastore using `two` as *data context id*: by default the bean named `entityManagerFactory_two` will be used as `EntityManagerFactory`

③ A specific `Datastore` type bean reference can be obtained using the *data context id* as **qualifier**

5.2.3. Auto-flush mode

The JPA `Datastore` Auto-flush mode can be configured using the `autoFlush` attribute of the `@EnableJpaDatastore` annotation.

```
@EnableJpa(autoFlush = true) ①
@Configuration
class Config {

  // ...

}
```

① Enable the JPA `Datastore` auto-flush mode

5.2.4. Transactional JPA Datastore operations

The `transactional` attribute of the `@EnableJpaDatastore` annotation can be used to control the *transactional* configuration of a set of the `Datastore` API operations.

When setted to `true`, the Spring `@Transactional` annotation behavior is automatically added to the following `Datastore` API operation methods:

- `refresh`
- `insert`
- `update`
- `save`
- `delete`

The default `REQUIRED` *propagation* behavior is used, thus allowing the method calls to participate in an existing transaction or to be executed in a new one when the Spring transactions infrastructure is used.

The `transactional` attribute is **`true` by default**

```
@EnableJpa
@PropertySource("jdbc.properties")
@EnableTransactionManagement ①
@Configuration
class Config {

}

@Autowired
Datastore datastore;

void doTransactionally() {
  datastore.insert(DataTarget.named("Test"), buildPropertyBoxValue()); ②
}
```

① Enables Spring's annotation-driven transaction management capability.

② The `insert` method is *transactional* by default, so the `@Transactional` annotation is not explicitly required here

The `transactional` attribute can be used to disable this default behavior:

```
@EnableJpaDatastore(transactional = false)
@Configuration
class Config2 {

}
```

5.2.5. Primary mode

The `@EnableJpaDatastore` annotation provides a `primary()` attribute which can be used to control the *primary mode* of the JPA `Datastore` bean registration.

If the *primary mode* is set to `PrimaryMode.TRUE`, the `Datastore` bean created with the corresponding annotation will be marked as **primary** in the Spring application context, meaning that will be the one provided by Spring in case of multiple available candidates, when no specific bean name or qualifier is specified in the dependency injection declaration.

This behaviour is similar to the one obtained with the Spring `@Primary` annotation at bean definition time.

By default, the *primary mode* is set to `PrimaryMode.AUTO`, meaning that the registred JPA Datastore bean will be marked as **primary** only when the `EntityManagerFactory` bean to which is bound is registered as primary candidate bean.

5.2.6. JPA Datastore configuration properties

When a JPA Datastore bean is configured using the `@EnableJpaDatastore` annotation, the Spring environment is automatically used as configuration properties source.

This way, many Datastore configuration settings can be provided using a configuration property with the proper name and value.

The supported configuration properties are:

1. The standard Datastore configuration properties, avaible from the DatastoreConfigProperties property set (See Datastore configuration).

The configuration property prefix is `holon.datastore` and the following properties are available:

Table 1. Datastore configuration properties

Name	Type	Meaning
holon.datastore. **trace**	Boolean (`true` / `false`)	Enable/disable Datastore operations *tracing*.
holon.datastore. **dialect**	String	The fully qualified class name of the *ORM Dialect* to use. See ORM Dialect configuration.

2. An additional set of properties, provided by the JpaDatastoreConfigProperties property set, which can be used as an alternative for the `@EnableJpaDatastore` annotation attributes described in the previous sections.

Table 2. JPA Datastore configuration properties

Name	Type	Meaning
holon.datastore.jpa. **primary**	Boolean (`true` / `false`)	Mark the JPA Datastore bean as *primary* candidate for dependency injection when more than one definition is available. IF not setted to `true`, the **AUTO** strategy will be used: the JPA Datastore bean will be marked as primary only when the `EntityManagerFactory` bean to which is bound is registered as primary candidate bean.
holon.datastore.jpa. **auto-flush**	Boolean (`true` / `false`)	Enable or disable the JPA Datastore auto-flush mode. See Auto-flush mode.
holon.datastore.jpa. **transactional**	Boolean (`true` / `false`)	Whether to add the Spring `@Transactional` behavior to the suitable Datastore API methods. See Transactional JPA Datastore operations.

Example of Datastore configuration properties:

```
holon.datastore.trace=true ①
holon.datastore.dialect=my.dialect.class.Name ②

holon.datastore.jpa.auto-flush=true ③
holon.datastore.jpa.transactional=false ④
```

① Enable tracing

② Set the ORM dialect class name

③ Enable the auto-flush mode

④ Disable the automatic *transactional* behavior of the Datastore operations

5.2.7. Datastore extension and configuration using the Spring context

The JPA Datastore implementation supports the standard Holon Platform Datastore Spring integration features for Datastore beans configuration and extension, which includes:

- Datastore **configuration post processing** using DatastorePostProcessor type beans.
- Datastore **extension** through ExpressionResolver registration using DatastoreResolver annotated beans.

- Datastore **commodity factory** registration using DatastoreCommodityFactory annotated beans.

```
@DatastoreResolver ①
class MyFilterExpressionResolver implements QueryFilterResolver<MyFilter> {

  @Override
  public Class<? extends MyFilter> getExpressionType() {
    return MyFilter.class;
  }

  @Override
  public Optional<QueryFilter> resolve(MyFilter expression, ResolutionContext context)
      throws InvalidExpressionException {
    // implement actual MyFilter expression resolution
    return Optional.empty();
  }

}

@Component
class MyDatastorePostProcessor implements DatastorePostProcessor { ②

  @Override
  public void postProcessDatastore(Datastore datastore, String datastoreBeanName) {
    // configure Datastore
  }

}

@DatastoreCommodityFactory ③
class MyCommodityFactory implements JpaDatastoreCommodityFactory<MyCommodity> {

  @Override
  public Class<? extends MyCommodity> getCommodityType() {
    return MyCommodity.class;
  }

  @Override
  public MyCommodity createCommodity(JpaDatastoreCommodityContext context)
      throws CommodityConfigurationException {
    // create commodity instance
    return new MyCommodity();
  }

}
```

① Automatically register a Datastore expression resolver using the `@DatastoreResolver` annotation

② Post process Datastore configuration using a `DatastorePostProcessor` type Spring bean

③ Automatically register a Datastore commodity factory using the `@DatastoreCommodityFactory`

annotation

5.3. Full JPA bean stack auto-configuration

The EnableJpa annotation can be used on Spring configuration classes to setup a full JPA enviroment bean stack:

- **DataSource:** If a `javax.sql.DataSource` type bean is not already registered in Spring context, a `DataSource` instance is created and configured using the Holon Platform DataSource configuration properties. See the Holon Platform JDBC Module documentation for details.
- **EntityManagerFactory:** A JPA `javax.persistence.EntityManagerFactory` bean is registered and configured, along with a suitable Spring `JpaVendorAdapter` instance for ORM configuration. A default Spring `LocalContainerEntityManagerFactoryBean` is used as `EntityManagerFactory` factory bean.
- **PlatformTransactionManager:** A `org.springframework.transaction.PlatformTransactionManager` bean is registered to enable the Spring transactions management APIs and conventions;
- **Datastore:** A JPA `Datastore` is configured and bound to the `EntityManagerFactory` bean of the JPA stack.

In a typical scenario, to enable a full JPA stack in Spring context, the `@EnableJpa` annotation can be used as foolows:

```
@EnableJpa(entityPackageClasses = TestEntity.class) ①
@PropertySource("jdbc.properties")
@Configuration
class Config {

}

@Autowired
DataSource dataSource;

@PersistenceUnit
EntityManagerFactory entityManagerFactory;

@Autowired
PlatformTransactionManager transactionManager;

@Autowired
Datastore datastore;
```

① Specify the base package to use to auto-detect the JPA *entity* definition classes to mapped into the `EntityManagerFactory` instance

The `DataSource` configuration properties in the example above are expected to be provided through the `jdbc.properties` file. For example:

```
holon.datasource.url=jdbc:h2:mem:test
holon.datasource.username=sa
```

See the next sections to learn about all the available configuration attributes to fine tune the JPA stack beans configuration.

5.3.1. `DataSource` configuration

A `DataSource` type bean is automatically configured if:

- A `javax.sql.DataSource` type bean is not already registered in the Spring context.
- The `dataSourceReference` attribute of the `@EnableJpa` annotation is not specified. If specified, it indicates the `DataSource` bean name to use, which must be available in the Spring context.

In order to auto-configure the `DataSource` bean, a suitable set of configuration properties must be available in the Spring Environment.

See the Holon Platform DataSource configuration properties section of the Holon Platform JDBC Module documentation for details.

5.3.2. `EntityManagerFactory` configuration

A Spring `LocalContainerEntityManagerFactoryBean` instance is used as `javax.persistence.EntityManagerFactory` implementation to ensure a full integration with Spring JPA architecture and provide functionalities such as automatic JPA persistence unit configuration without the need of a `persistence.xml` configuration file.

If a `persistence.xml` configuration file is not available, automatic persistence unit *entity* classes configuration can be performed using one of the following `@EnableJpa` annotation attributes:

- `entityPackages` : providing a list of package names to scan in order to map JPA *entity* classes into the `EntityManagerFactory`,
- `entityPackageClasses` : providing a list of classes from which to obtain the package names to scan in order to map JPA *entity* classes into the `EntityManagerFactory`. Represents a type-safe alternative to `entityPackages` attribute.

Two more annotation attributes are available for persistence unit configuration:

- `validationMode` : Specify the JPA 2.0 validation mode for the persistence unit;
- `sharedCacheMode` : Specify the JPA 2.0 shared cache mode for the persistence unit.

```
@EnableJpa(entityPackageClasses = TestEntity.class, validationMode = ValidationMode
.NONE)
@PropertySource("jdbc.properties")
@Configuration
class Config {

}
```

5.3.3. Persistence provider (ORM) configuration

By default, the JPA Persistence provider (ORM) to use is **auto-detected from classpath** and a suitable Spring `JpaVendorAdapter` instance is configured and bound to the `EntityManagerFactory` bean.

If more than one persistence provider implementation is present in classpath (or to explicitly specify a persistence provider implementation to use, anyway) the `orm` configuration property can be used. See JPA configuration properties for details.

5.3.4. Transaction manager configuration

A Spring JPA `PlatformTransactionManager` is auto-configured and bound to the `EntityManagerFactory` bean to enable the Spring's transactions support.

A set of attributes are made available by the `@EnableJpa` annotation to fine tune the transaction manager configuration:

Attribute name	Type	Meaning	Default
`transactionSynchronization`	int	Set when the transaction manager should activate the thread-bound transaction synchronization support.	-1 (always)
`defaultTimeout`	int	Specify the default timeout that the transaction manager should apply if there is no timeout specified at the transaction level, in seconds.	-1 (`TransactionDefinition.TIMEOUT_DEFAULT`)
`validateExistingTransaction`	boolean	Set whether existing transactions should be validated before participating in them.	`false`

Attribute name	Type	Meaning	Default
`failEarlyOnGlobalRollbackOnly`	boolean	Set whether to fail early in case of the transaction being globally marked as rollback-only.	`false`
`rollbackOnCommitFailure`	boolean	Set whether a *rollback* should be performed on failure of the transaction manager *commit* call.	`false`

5.3.5. JPA Datastore configuration

By default, a JPA `Datastore` implementation is automatically configured using the `EntityManagerFactory` bean and made available as Spring bean. The JPA `Datastore` implementation is configured for a seamless integration with the Spring JPA infrastructure, as described in Integration with the Spring JPA infrastructure.

To disable automatic JPA Datastore configuration, the `enableDatastore` attribute of the `@EnableJpa` annotation can be used.

```
@EnableJpa(entityPackageClasses = TestEntity.class, enableDatastore = false) ①
@PropertySource("jdbc.properties")
@Configuration
class Config {

}
```

① Disable the JPA `Datastore` bean auto configuration

The `@EnableJpa` annotation makes available two additional configuration attributes related to the JPA `Datastore` implementation setup:

- `autoFlush` : whether to enable the JPA `Datastore` *auto flush* mode. See Auto-flush mode for details.
- `transactionalDatastore` : whether to add *transactional* behaviour to data manipulation `Datastore` API operations. See Transactional JPA Datastore operations for details.

5.3.6. JPA configuration properties

The JpaConfigProperties interface provides a set of configuration properties to be used with JPA stack beans auto-configuration. It extends a default `ConfigPropertySet` bound to the property name prefix **holon.jpa**.

The available configuration properties are listed below:

Table 3. JPA configuration properties

Name	Type	Meaning
holon.jpa. **orm**	ORMPlatform enumeration	ORM platform to use as persistence provider.
holon.jpa. **dialect**	String	ORM dialect class name to use.
holon.jpa. **database**	`com.holonplatform.jdbc.DatabasePlatform` enumeration	Database platform to which the `DataSource` is connected (auto-detected by default).
holon.jpa. **generate-ddl**	Boolean (true/false)	Whether to initialize the database schema on startup.
holon.jpa. **show-sql**	Boolean (true/false)	Whether to instruct JPA ORM engine to show executed SQL statements, if supported by the ORM platform.

The `JpaConfigProperties` can be loaded from a number a sources using the default `ConfigPropertySet` builder interface.

Using the Spring integration, all `Environment` registered PropertySources will be enabled as a `JpaConfigProperties` source.

5.3.7. Multiple JPA bean stacks configuration

When more than one JPA stack has to be configured using the `@EnableJpa` annotation, the `dataContextId` attribute can be used to assign a different **data context id** to each JPA bean definitions stack, in order to:

- Provide different sets of configuration properties using the same Spring environment.
- Provide a default *name pattern strategy* for the JPA beans stack using the bean name pattern: `beanDefinitionName_{datacontextid}` where `{datacontextid}` is equal to the `dataContextId` attribute of the `@EnableJpa` annotation.

When a *data context id* is defined, a Spring **qualifier** named the same as the *data context id* will be associated to each JPA bean definition, and such qualifier can be later used to obtain the right bean instance through dependency injection.

For example, given the following `jdbc.properties` configuration property source file:

```
holon.datasource.one.url=jdbc:h2:mem:testdbm1
holon.datasource.one.username=sa

holon.datasource.two.url=jdbc:h2:mem:testdbm2
holon.datasource.two.username=sa
```

To configure two JPA bean stacks, one bound to the `one` *data context id* and the other to the `two` *data context id*, the `@EnableJpa` annotation can be used as follows:

```
@Configuration
@PropertySource("jdbc.properties")
static class Config {

  @Configuration
  @EnableJpa(dataContextId = "one", entityPackageClasses = TestEntity1.class)
  static class Config1 {
  }

  @Configuration
  @EnableJpa(dataContextId = "two", entityPackageClasses = TestEntity2.class)
  static class Config2 {
  }

}

@Autowired
@Qualifier("one")
Datastore datastore1;

@Autowired
@Qualifier("two")
Datastore datastore2;
```

6. Spring Boot integration

The `holon-datastore-jpa-spring-boot` artifact provides integration with Spring Boot for JPA stack and JPA `Datastore` beans **auto-configuration**.

To enable Spring Boot JPA auto-configuration features, the following artifact must be included in your project dependencies:

Maven coordinates:

```
<groupId>com.holon-platform.jpa</groupId>
<artifactId>holon-datastore-jpa-spring-boot</artifactId>
<version>5.1.0</version>
```

6.1. JPA Datastore auto-configuration

A JPA `Datastore` bean is auto-configured only when:

- A `JpaDatastore` type bean is not already available from the Spring application context.
- A valid `EntityManagerFactory` type bean is available from the Spring application context.

The JPA `Datastore` auto-configuration strategy is the same adopted by the `@EnableJpaDatastore`

annotation, including the Spring environment configuration properties support. See JPA Datastore configuration for details.

For example, given an `application.yml` configuration file as follows:

```
holon:
  datastore:
    trace: true
```

The Holon platform Spring Boot auto-configuration classes will auto configure a JPA `Datastore` bean bound to the `EntityManagerFactory` bean (which must be available in the Spring application context), enabling operations *tracing* according to the `holon.datastore.trace` property value.

```
@Configuration
@EnableAutoConfiguration
class Config {

  @Bean
  public DataSource dataSource() {
    return buildDataSource();
  }

  @Bean
  public FactoryBean<EntityManagerFactory> entityManagerFactory(DataSource dataSource)
{
    LocalContainerEntityManagerFactoryBean emf = new
LocalContainerEntityManagerFactoryBean();
    emf.setDataSource(dataSource);
    emf.setJpaVendorAdapter(new HibernateJpaVendorAdapter());
    emf.setPackagesToScan("com.example.entities");
    return emf;
  }

}

@Autowired
Datastore datastore; ①
```

① A JPA `Datastore` type bean is auto-configured ad available from the Spring context

To disable this auto-configuration feature the `JpaDatastoreAutoConfiguration` class can be excluded:

```
@EnableAutoConfiguration(exclude={JpaDatastoreAutoConfiguration.class})
```

6.2. Full JPA beans stack auto-configuration

When the `holon-datastore-jpa-spring-boot` artifact is available in classpath, a **full JPA beans stack**

is auto-configured when an `EntityManagerFactory` type bean is not already registered in the Spring application context.

The JPA beans stack configuration strategy is the same of the one adopted by the `@EnableJpa` annotation, including the Spring environment configuration properties support. See Full JPA bean stack auto-configuration for details.

For example, given an `application.yml` configuration file as follows:

```
spring:
  datasource:
    url: "jdbc:h2:mem:test"
    username: "sa"

holon:
  datastore:
    trace: true
```

The Holon platform Spring Boot auto-configuration classes will auto configure a JPA beans stack using the `DataSource` auto-configured by the standard Spring Boot classes `spring.datasource.*` configuration properties. The Datastore operations *tracing* is enabled according to the `holon.datastore.trace` property value.

```
@Configuration
@EnableAutoConfiguration
class Config {

}

@Autowired
DataSource dataSource;

@PersistenceUnit
EntityManagerFactory entityManagerFactory;

@Autowired
PlatformTransactionManager transactionManager;

@Autowired
Datastore datastore;
```

The Holon Platform `DataSource` auto-configuration capabilities can also be used, replacing the `spring.datasource. configuration properties with the holon.datasource.` ones. For example:

```
holon:
  datasource:
    url: "jdbc:h2:mem:test"
    username: "sa"

  datastore:
    trace: true
```

To disable this auto-configuration feature the `JpaAutoConfiguration` class can be excluded:

```
@EnableAutoConfiguration(exclude={JpaAutoConfiguration.class})
```

6.3. Multiple JPA beans stack auto configuration

When the `DataSource` type beans are auto-configured using the Holon Platform JDBC Module features (see the Holon Platform JDBC Module Spring Boot integration documentation section), the auto-configuration of multiple JPA bean stacks is available out-of-the-box.

The ***data context id* convention** is used to provide multiple `DataSource` and JPA beans stack auto-configuration capabilities: when multiple `DataSource` type beans are registered, each of them bound to a *data context id*, the Spring Boot auto-configuration classes will provide to configure a JPA beans stack for each detected `DataSource` bean, binding the same *data context id* to the JPA stack bean definitions.

According to the *data context id* convention, each `DataSource` and JPA stack bean will be *qualified* with the corresponding *data context id*, so that the specific bean instance can be later obtained using the *data context id* as Spring bean **qualifier** name.

For example, given an `application.yml` configuration file as follows:

```
holon:
  datasource:
    one:
      url: "jdbc:h2:mem:test1"
      username: "sa"
    two:
      url: "jdbc:h2:mem:test2"
      username: "sa"
```

The auto-configuration feature will configure two `DataSource` beans:

- One `DataSource` bean instance using the `one` *data context id* configuration properties, qualified with the **one** qualifier.
- Another `DataSource` bean instance using the `two` *data context id* configuration properties, qualified with the **two** qualifier.

And two corresponding JPA beans stacks:

- One backed by the `DataSource` bound to the `one` *data context id*, qualified with the **one** qualifier.
- Another backed by the `DataSource` bound to the `two` *data context id*, qualified with the **two** qualifier.

So, for example, the `DataSource` and JPA `Datastore` beans can be obtained as follows:

```
// 'one' data context id:
@Autowired @Qualifier("one")
DataSource dataSource1;

@Autowired @Qualifier("one")
Datastore datastore1;

// 'two' data context id:
@Autowired @Qualifier("two")
DataSource dataSource2;

@Autowired @Qualifier("two")
Datastore datastore2;
```

6.3.1. JPA entities scan

When more than one JPA beans stack is auto-configured, the JpaEntityScan repeatable annotation can be used to provide the JPA *entity* set to map to each auto-configured `EntityManagerFactory`, using the *data context id* value to identify each `EntityManagerFactory` bean.

So, for example, given an `application.yml` configuration file as follows:

```
holon:
  datasource:
    one:
      url: "jdbc:h2:mem:test1"
      username: "sa"
    two:
      url: "jdbc:h2:mem:test2"
      username: "sa"
```

Two `DataSource` beans and two JPA beans stacks will be auto-configured using the `one` and `two` data context ids.

The `@JpaEntityScan` annotation can be used to specify the packages to scan for JPA entity classes mapping, one for the `one` data context id and another for the `two` data context id:

```
@Configuration
@EnableAutoConfiguration
@JpaEntityScan(value = "one", basePackageClasses = TestEntity1.class) ①
@JpaEntityScan(value = "two", basePackageClasses = TestEntity2.class) ②
class Config {

}
```

① Use the base package of the `TestEntity1` class to scan for the JPA entity classes to map to the `EntityManagerFactory` bean associated to the `one` data context id

② Use the base package of the `TestEntity2` class to scan for the JPA entity classes to map to the `EntityManagerFactory` bean associated to the `two` data context id

6.4. Spring Boot starters

The following *starter* artifacts are available to provide a quick project configuration setup using Maven dependency system:

1. JPA starter using Hibernate provides the dependencies to the Holon JPA Spring Boot integration artifact `holon-datastore-jpa-spring-boot` and the dependencies to use **Hibernate ORM** as persistence provider.

Furthermore, it provides dependencies for:

- The Holon Platform Core Module Spring Boot integration base starter (`holon-starter`).
- The base Spring Boot starter (`spring-boot-starter`), see the Spring Boot starters documentation for details.
- The HikariCP pooling `DataSource` implementation.

Maven coordinates:

```
<groupId>com.holon-platform.jpa</groupId>
<artifactId>holon-starter-jpa-hibernate</artifactId>
<version>5.1.0</version>
```

2. JPA starter using Eclipselink provides the same dependencies as the previous JPA starter, but using **Eclipselink** as persistence provider, instead of Hibernate.

Maven coordinates:

```
<groupId>com.holon-platform.jpa</groupId>
<artifactId>holon-starter-jpa-eclipselink</artifactId>
<version>5.1.0</version>
```

7. Loggers

By default, the Holon platform uses the SLF4J API for logging. The use of SLF4J is optional: it is enabled when the presence of SLF4J is detected in the classpath. Otherwise, logging will fall back to JUL (`java.util.logging`).

The logger name for the **JPA Datastore** module is `com.holonplatform.datastore.jpa`.

8. System requirements

8.1. Java

The Holon Platform JPA Datastore module requires Java **8** or higher.

8.2. JPA

The **Java Persistence API version 2.0 or higher** is required for the JPA Datastore module proper use. To use most recent JPA features, such as *left joins* and the *ON* clause, the **Java Persistence API version 2.1** is required.

8.3. Persistence providers

Altough any JPA 2.0 or higher compliant persistence provider (ORM) is supported, the Holon Platform JPA Datastore module is tested and certified with the followings:

- Hibernate ORM version **4.x** or **5.x**
- EclipseLink version **2.5 or higher**
- Apache OpenJPA version **2.4 or higher**
- Datanucleus version **5.1 or higher**