

Holon
PLATFORM

# Holon Platform JDBC Datastore Module - Reference manual

Version 5.1.3

# Table of Contents


1. Introduction . . . 1
   1.1. Sources and contributions . . . 1
2. Security related considerations . . . 1
3. Obtaining the artifacts . . . 1
   3.1. Using the Platform BOM . . . 2
4. What's new in version 5.1.x . . . 2
   4.1. Migrating from version 5.0.x . . . 3
      4.1.1. Deprecations . . . 3
      4.1.2. Dialects . . . 3
      4.1.3. SQL filter and sort . . . 3
5. JDBC Datastore . . . 3
   5.1. Setup and configuration . . . 4
      5.1.1. Common Datastore configuration options . . . 4
      5.1.2. `DataSource` configuration . . . 6
      5.1.3. Database platform configuration . . . 7
      5.1.4. SQL Dialect configuration . . . 7
      5.1.5. Builtin SQL dialects . . . 8
      5.1.6. Identifier resolution strategy . . . 9
      5.1.7. JDBC connection handler . . . 10
   5.2. Data model attributes naming conventions . . . 11
      5.2.1. Data path mappings . . . 12
   5.3. SQL data types mapping . . . 14
   5.4. Relational expressions . . . 15
   5.5. Auto-generated ids . . . 15
   5.6. Custom SQL filters and sorts . . . 16
      5.6.1. WhereFilter . . . 16
      5.6.2. OrderBySort . . . 17
   5.7. Transactions management . . . 17
   5.8. `JdbcDatastore` API . . . 18
   5.9. Extending the JDBC Datastore API . . . 19
   5.10. Expression resolvers . . . 19
      5.10.1. JDBC Expression resolvers registration . . . 19
      5.10.2. Specific expression resolvers registration . . . 20
      5.10.3. Expression resolvers priority . . . 20
      5.10.4. Expression validation . . . 20
      5.10.5. JDBC Datastore expressions . . . 20
      5.10.6. JDBC Expression resolution context . . . 25
   5.11. Commodity factories . . . 25

6. Spring ecosystem integration . . . 28
6.1. Integration with the Spring JDBC infrastructure . . . 28
6.2. JDBC Datastore configuration . . . 29
6.2.1. DataSource . . . 29
6.2.2. Multiple JDBC Datastores configuration . . . 30
6.2.3. Database platform . . . 33
6.2.4. Identifier resolution strategy . . . 34
6.2.5. Transactional JDBC Datastore operations . . . 34
6.2.6. Primary mode . . . 35
6.2.7. JDBC Datastore configuration properties . . . 36
6.2.8. Datastore extension and configuration using the Spring context . . . 37
7. Spring Boot integration . . . 39
7.1. Multiple JDBC Datastore auto configuration . . . 40
7.2. Disabling the JDBC Datastore auto-configuration feature . . . 41
7.3. Spring Boot starters . . . 42
8. Loggers . . . 42
9. System requirements . . . 43
9.1. Java . . . 43
9.2. JDBC Drivers . . . 43

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 **JDBC Datastore** is the *Java DataBase Connectivity* reference implementation of the Datastore API.



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

The JDBC `Datastore` implementation uses the Java `javax.sql.DataSource` API to access the the physical data source and the *SQL* language to perform data access and management operations.

## 1.1. Sources and contributions

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

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. Security related considerations

The SQL composition engine of JDBC Datastore API is designed to **avoid and prevent SQL injection type attacks**, since a meta-language is used for the Datastore operations definition a any provided *parameter* value is managed using JDBC `PreparedStatement` statements types.

# 3. 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.jdbc</groupId>
<artifactId>holon-datastore-jdbc-bom</artifactId>
<version>5.1.3</version>
```

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

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.holon-platform.jdbc</groupId>
      <artifactId>holon-datastore-jdbc-bom</artifactId>
      <version>5.1.3</version>
      <strong><type>pom</type></strong>
      <strong><scope>import</scope></strong>
    </dependency>
  </dependencies>
</dependencyManagement>
```

## 3.1. 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.

# 4. What's new in version 5.1.x

- Support of the the `PropertySet` **identifier properties** to detect the persistent entities primary keys avoiding additional database accesses. See Identifier resolution strategy.
- 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 **SQL composer engine**, which is now separated as an independent artifact (`holon-datastore-jdbc-composer`). This ensures more consistent operation resolution strategies, remarkable performance improvements and extensibility by design. See Extending the JDBC Datastore API.
- Improved support for industry-standard vendors RDBMS, such as Oracle Database, Microsoft SQLServer, IBM DB2 and SAP HANA.

## 4.1. Migrating from version 5.0.x

### 4.1.1. Deprecations

- **JdbcDatastore Builder API**: autoCommit. The default auto-commit mode setup for JDBC connections is deprecated in favor of the new *transactional* API support. Transactional operations should be used to manage connections auto-commit. See Transactions management.

### 4.1.2. Dialects

Because of the complete and deep revision and rewriting of the internal SQL composer engine, the package structure and class name is deeply changed. This not affects the public API in any way, only the concrete *sql dialect* implementations were relocated and the dialect class name is changed from `JdbcDialect` to `SQLDialect`.

The new dialect implementations can be found in the `holon-datastore-jdbc-composer` artifact, under the `com.holonplatform.datastore.jdbc.composer.dialect` package.

So if a dialect fully qualified class name was used to specify the JDBC Datastore dialect, it has to be replaced with the new `SQLDialect` implementation class.

### 4.1.3. SQL filter and sort

The `com.holonplatform.datastore.jdbc.JdbcWhereFilter` and `com.holonplatform.datastore.jdbc.JdbcOrderBySort`, which allow custom `QueryFilter` and `QuerySort` declarations directly providing the SQL `WHERE` and `ORDER BY` parts, are replaced with the WhereFilter and OrderBySort classes, located in the `holon-datastore-jdbc-composer` artifact.

The classes semantic and API is the same as before. See Custom SQL filters and sorts.

# 5. JDBC Datastore

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

*Maven coordinates*:

```
<groupId>com.holon-platform.jdbc</groupId>
<artifactId>holon-datastore-jdbc</artifactId>
<version>5.1.3</version>
```

The JdbcDatastore interface represents the **JDBC Datastore** API implementation, extending the core `Datastore` API.

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

- Create and configure a `JdbcDatastore` API instance, using the provided *builder*.

- Directly working with `JdbcDatastore` managed *connections*, represented by the standard JDBC connection interface (`java.sql.Connection`). See `JdbcDatastore` API.



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 `JdbcDatastore` 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.

## 5.1. Setup and configuration

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

The JDBC Datastore builder provides a `JdbcDatastore` instance:

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

① Obtain the JDBC Datastore builder to configure and create a new JDBC 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 `JdbcDatastore` API.

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

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

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

### 5.1.1. Common Datastore configuration options

The JDBC 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 JDBC Datastore will log any SQL 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 JDBC Datastore configuration:

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

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

### 5.1.2. `DataSource` configuration

The JDBC Datastore implementation relies on the `javax.sql.DataSource` API to access the concrete data store using JDBC, and the `DataSource` instance to be used must be provided at JDBC Datastore configuration time.

The `DataSource` reference is the only **required** JDBC Datastore configuration attribute and can be provided in two ways:

**1. Direct:**

The concrete `DataSource` implementation can be provided using the JDBC Datastore builder, through the `dataSource(DataSource dataSource)` method.

```
DataSource datasource = createOrObtainDatasource();

Datastore datastore = JdbcDatastore.builder() //
    .dataSource(datasource) ①
    .build();
```

① Set the `DataSource` instance to use

**2. Indirect:**

A `DataSource` instance can be otained using the Holon Platform JDBC module `DataSource` configuration facilities through a set of configuration properties, represented by the DataSourceConfigProperties property set.



See the JDBC module DataSource configuration properties documentation to learn about the available `DataSource` configuration properties.

A `DataSourceConfigProperties` instance can be loaded using the configuration property providers API and provided to the JDBC Datastore builder.

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

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

The JDBC Datastore instance can be created in the following way:

```
Datastore datastore = JdbcDatastore.builder() //
    .dataSource(DataSourceConfigProperties.builder().withPropertySource(
"datasource.properties").build()) ①
    .build();
```

① A `DataSource` instance will be created and configured for the JDBC Datastore using the DataSource configuration properties loaded from the `datasource.properties` file



See the DataSource configuration documentation section of the Holon Platform JDBC module for detailed information about the `DataSource` creation and configuration APIs and strategies.

### 5.1.3. Database platform configuration

For some internal operations, the JDBC Datastore needs to know the concrete *Database platform* to which the `DataSource` instance is bound (for example *H2*, *MySQL* and so on). The *Database platform* is used, for example, to auto-detect the SQL dialect to use if not directly specified at JDBC Datastore configuration time.

The *Database platform* can be explicitly specified either using:

- The JDBC Datastore `database` builder method, using the DatabasePlatform enumeration.

```
Datastore datastore = JdbcDatastore.builder() //
    .dataSource(createOrObtainDatasource()) //
    .database(DatabasePlatform.H2) ①
    .build();
```

① Database platform specification

- Or the `holon.datasource.platform` configuration property, if provided in the `DataSource` configuration property set when using the *indirect* `DataSource` instance configuration, as described in the previous section.

**Database platform auto detection:**

When the *Database platform* is not explicitly specified, the JDBC Datastore will try to **auto-detect** it, inspecting the JDBC connection URL of the provided `DataSource` instance.

### 5.1.4. SQL Dialect configuration

To ensure operations consistency and efficiency, the JDBC Datastore uses the SQLDialect API abstraction in order to resolve each database platform specificity and SQL language difference.

Normally, the `SQLDialect` implementation to use is **auto-detected** by the JDBC Datastore, relying on the *Database platform* to which the `DataSource` instance is bound, as described in the previous section.

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

**1. `SQLDialect` configuration using the JDBC Datastore builder:**

The SQL dialect to use can be directly configured using the JDBC Datastore builder, using the `dialect(SQLDialect dialect)` or the `dialect(String dialectClassName)` method.

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

```
Datastore datastore = JdbcDatastore.builder() //
    .dataSource(createOrObtainDatasource()) //
    .dialect(SQLDialect.h2()) ①
    .dialect("com.holonplatform.datastore.jdbc.composer.dialect.H2Dialect") ②
    .build();
```

① Configure the dialect using a specific dialect implementation

② Configure the dialect using the dialect class name

**1. `SQLDialect` configuration using a configuration property:**

The SQL 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.jdbc.composer.dialect.H2Dialect
holon.datastore.trace=true
```

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

```
Datastore datastore = JdbcDatastore.builder() //
    .dataSource(createOrObtainDatasource()) //
    .configuration(DatastoreConfigProperties.builder().withPropertySource(
"datastore.properties").build()) ①
    .build();
```

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

### 5.1.5. Builtin SQL dialects

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

| Database platform | Dialect class | Supported versions |
|---|---|---|
| DB2 | DB2Dialect | 8 and higher |
| Derby | DerbyDialect | 10.5 and higher |
| H2 | H2Dialect | 1.4 and higher |
| HyperSQL (HSQLDB) | HSQLDialect | 2.0.0 and higher |
| Informix | InformixDialect | 11 and higher |
| MySQL | MySQLDialect | 4.1 and higher |
| MariaDB | MariaDBDialect | 5.5 and higher |
| Oracle Database | OracleDialect | 9i and higher |
| PostgreSQL | PostgreSQLDialect | 8.2.5 and higher |
| Microsoft SQL Server | SQLServerDialect | 2005 or higher |
| SAP HANA | HANADialect | 1.0 SPS12 and higher |
| SQLite | SQLiteDialect | 3.0.7 and higher |

### 5.1.6. Identifier resolution strategy

For some of its internal operations, the JDBC Datastore needs to know the actual table **primary key** to ensure consistency, for example for the `save` Datastore API operation, which performs an *insert* type operation if a row is not present or an *update* type operation otherwise.

Any standard `Datastore` API operation relies on the Holon Platform property model to represent the data model, using the `PropertyBox` API to receive and provide the data model attributes values.

To ensure consistency for operations which have to *identify* a `PropertyBox` instance against the concrete data model (in the JDBC world, the *row* of a table), the JDBC Datastore needs to know which properties of the `PropertyBox` property set have to be considered as **identifier properties**, if any. This way, the identifier property values can be used to implement a consistent strategy to map the `PropertyBox` instances to the proper table *rows*.

The JDBC Datastore **identifier resolution strategy** defines how this is accomplished, and it is represented by the IdentifierResolutionStrategy enumeration.

The available identifier resolution strategies are:

| Strategy | Description |
|---|---|
| `IDENTIFIER_PROPERTIES` | Use the `PropertySet` **identifier properties** as `PropertyBox` identifier properties. See PropertySet identifier properties for details. |
| `TABLE_PRIMARY_KEY` | Use the database table **primary key** to obtain the `PropertyBox` identifier properties. |
| `AUTO` | Use the `IDENTIFIER_PROPERTIES` strategy when the `PropertyBox` property set provides configured identifier properties, otherwise use the `TABLE_PRIMARY_KEY` strategy. |

The `AUTO` identifier resolution strategy is the **default** strategy adopted by the JDBC Datastore.

To configure a specific identifier resolution strategy, the JDBC Datastore builder API can be used, through the `identifierResolutionStrategy` method.

```
Datastore datastore = JdbcDatastore.builder() //
    .dataSource(createOrObtainDatasource()) //
    .identifierResolutionStrategy(IdentifierResolutionStrategy.TABLE_PRIMARY_KEY) ①
    .build();
```

① Set the identifier resolution strategy to `TABLE_PRIMARY_KEY`

### 5.1.7. JDBC connection handler

The JDBC Datastore builder allows to configure the JDBC connection lifecycle through the JdbcConnectionHandler API.

When a custom implementation is provided, the `getConnection` and `releaseConnection` methods of the `JdbcConnectionHandler` API will be use to obtain a JDBC connection and to release it, respectively.

The default JDBC connection handler used by the JDBC Datastore simply uses the `getConnection()` method of the `DataSource` to obtain a connection and the `Connection.close()` method to release it.

```
Datastore datastore = JdbcDatastore.builder() //
    .dataSource(createOrObtainDatasource()) //
    .connectionHandler(new JdbcConnectionHandler() { ①

      @Override
      public Connection getConnection(DataSource dataSource, ConnectionType
connectionType)
          throws SQLException {
        // provide the JDBC connection
        return dataSource.getConnection();
      }

      @Override
      public void releaseConnection(Connection connection, DataSource dataSource,
          ConnectionType connectionType) throws SQLException {
        // release the JDBC connection
        connection.close();
      }

    }).build();
```

① Configure a custom `JdbcConnectionHandler` for the JDBC Datastore

## 5.2. Data model attributes naming conventions

The JDBC Datastore relies on the following conventions regarding the mappings between the Holon Platform *property model* and the concrete JDBC data model (i.e. the RDMBS schema):

- A **DataTarget** **name** is mapped to a RDBMS **table (or view) name**.
- A `Path` **name** (and so a PathProperty path name) is mapped to a RDBMS table (or view) **column name**.



The `SQLDialect` in use could decide to apply some manipulation operations to the DataTarget or Path names in order to properly match the database schema elements names. For example, it could apply uppercase transformations if the undelying RDBMS only supports uppercase schema objects names.

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

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

The property model will be defined simply using the table and column names:

```
static final NumericProperty<Long> ID = NumericProperty.longType("code") ①
    .validator(Validator.notNull());
static final StringProperty VALUE = StringProperty.create("text") ②
    .validator(Validator.max(100));

static final PropertySet<?> TEST = PropertySet.builderOf(ID, VALUE).identifier(ID)
.build(); ③

static final DataTarget<?> TARGET = DataTarget.named("test"); ④
```

① Map the `ID` property to the `code` path name (i.e. the table column name)

② Map the `VALUE` property to the `text` path name (i.e. the table column name)

③ The `TEST` PropertySet is the `test` table schema representation. Furthermore, the `ID` property is configured as the property set identifier property

④ The `TARGET` DataTarget is defined using the `test` name, which corresponds to the database table name

Now the `Datastore` API can be used to manage the `test` table data:

```
Datastore datastore = JdbcDatastore.builder().dataSource(createOrObtainDatasource())
.build(); ①

PropertyBox value = PropertyBox.builder(TEST).set(ID, 1L).set(VALUE, "One").build();
datastore.save(TARGET, value); ②

Stream<PropertyBox> results = datastore.query().target(TARGET).filter(ID.goe(1L))
.stream(TEST); ③

List<String> values = datastore.query().target(TARGET).sort(ID.asc()).list(VALUE); ④

datastore.bulkDelete(TARGET).filter(ID.gt(0L)).execute(); ⑤
```

① Create a JDBC `Datastore` API instance

② Save a `test` table row (insert if not exists, update otherwise) with the given property values

③ Query the `test` table where the `ID` property value (i.e. the `code` table column value) is greater or equal to 1 and obtain the table rows as a Stream of `PropertyBox` values

④ Query the `test` table and obtain a list of the `VALUE` property values (i.e. the `text` table column value), ordering ascending by the `ID` property value

⑤ Execute a *bulk* delete operation, removing all the rows with the `ID` property value (i.e. the `code` table column value) greater than 0

### 5.2.1. Data path mappings

The JDBC Datastore fully supports the DataMappable API, which can be used to provide a *mapping* between a data related object path name and the actual path name to be used with the concrete data model schema.

When the `getDataPath()` method of a `DataMappable` type object returns a non empty value, such value will be used as actual **data path**, i.e. as RDBMS schema object name, instead of the default path name with which the object was configured.

```
StringProperty PROPERTY = StringProperty.create("propertyName").dataPath("str"); ①
```

① In this case, the `str` table column name will be used instead of `propertyName` when the property is involved in persistence related JDBC Datastore operations

This can be useful, for example, when an existing data model representation wants to used with a JDBC Datastore and the path property names do not match the actual RDBMS data model schema names.

As an example, we suppose to have a JPA entity model definition that we want to use as property model definition with a JDBC Datastore implementation.

If we don't want to define a brand new property model, through a `PathProperty` set declaration, the Bean introspection API can be used to obtain a BeanPropertySet from the JPA entity definitions. But this way the `PathProperty` path names of the `BeanPropertySet` will be the *JPA entity Bean property*

*names*, and it is not assured that they match the RDBMS schema objects names. In particular, if the `@Column(name="xxx")` is used on the Bean properties.

The **data path** mapping can be used to provide the actual data attribute name for the properties of the bean property set, in this case using the `name` attribute of the JPA `@Column` annotation.

We can use the Holon Platform JPA module to automatically map the `name` attribute of the JPA `@Column` annotation in the corresponding **data path** value of the bean property set `PathProperty`. Just ensure that the `holon-jpa-bean-processors` artifact is present in classpath.

At this point, the bean introspection API will use the provided JPA bean post processors to automatically configure the data path mapping in the properties, and the JDBC Datastore API can be seamlessy used, without any additional configuration.

For example, given the following JPA entity definition:

```
@Entity @Table(name = "test") class MyEntity {

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

  public static final DataTarget<MyEntity> TARGET = BeanDataTarget.of(MyEntity.class);
②

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

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

  // getters and setters omitted

}
```

① Create a `BeanPropertySet` using the entity class

② Create a `DataTarget` for the entity class: the `name` attribute of the JPA `@Table` annotation will be used to configure a **data path** for the bean data target using the `test` table name

Since the `name` attribute of the JPA `@Column` annotations will be setted as **data path** property values at bean introspection time, the `BeanPropertySet` properties can be now used with a JDBC Datastore API just like in the previous example:

```
Datastore datastore = JdbcDatastore.builder().dataSource(createOrObtainDatasource())
.build(); ①

PropertyBox value = PropertyBox.builder(MyEntity.PROPERTIES) //
    .set(MyEntity.PROPERTIES.property("id"), 1L) //
    .set(MyEntity.PROPERTIES.property("value"), "One").build();
datastore.save(MyEntity.TARGET, value); ②

Stream<PropertyBox> results = datastore.query().target(MyEntity.TARGET)
    .filter(MyEntity.PROPERTIES.property("id").goe(1L)).stream(MyEntity.PROPERTIES);
③
```

① Create a JDBC `Datastore` API instance

② Save a `test` table row (insert if not exists, update otherwise) with the given property values

③ Query the `test` table where the `id` property value (i.e. the `code` table column value) is greater or equal to `1` and obtain the table rows as a Stream of `PropertyBox` values

## 5.3. SQL data types mapping

When the Java type is *serialized* as a SQL data type (for example when used as query parameter) or a SQL data type is *deserialized* as a Java type (for example to map it as a `Property` value), the JDBC conventions are used to perform the type mapping.

When a SQL type has not a direct Java type representation, the following conventions are used:

*Table 1. SQL types mapping*

| SQL type | Supported Java types |
|---|---|
| CLOB | `java.lang.String`, `java.io.Reader` |
| BLOB | `byte[]`, `java.io.InputStream` |

*Table 2. Java types mapping*

| Java type | SQL type | Notes |
|---|---|---|
| `java.lang.Enum` | INTEGER | By default, the ordinal enum value is used for the mapping |
| `java.time.LocalDate` | DATE | A specific SQL dialect could use a different data type and perform value manipulation if required by the specific SQL type semantic |
| `java.time.LocalTime` | TIME | A specific SQL dialect could use a different data type and perform value manipulation if required by the specific SQL type semantic |

| Java type | SQL type | Notes |
|---|---|---|
| `java.time.LocalDateTime` | TIMESTAMP | A specific SQL dialect could use a different data type and perform value manipulation if required by the specific SQL type semantic |

## 5.4. Relational expressions

As relational Datastore, the JDBC `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.

## 5.5. Auto-generated ids

The JDBC Datastore API supports the retrieving of auto-generated id column values, if supported by the JDBC driver in use.

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 = getDatastore(); // build or obtain a JDBC Datastore

PropertyBox value = buildPropertyBoxValue();

OperationResult result = datastore.insert(DataTarget.named("test"), value); ①

Map<Path<?>, Object> keys = result.getInsertedKeys(); ②
Optional<Long> keyValue = result.getInsertedKey(ID); ③
keyValue = result.getFirstInsertedKey(Long.class); ④
```

① Perform a *insert* type operation using the `Datastore` API

② Get the auto-generated keys as a Map of key `Path` representation and key value

③ Get the auto-generated key value for the `ID` property, if available

④ Get the first auto-generated key value, if available, expecting it of `Long` type

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.

```
final PathProperty<Long> KEY = PathProperty.create("key", Long.class); ①
final PathProperty<String> TEXT = PathProperty.create("text", String.class);

Datastore datastore = getDatastore(); // build or obtain a JDBC Datastore

PropertyBox value = PropertyBox.builder(KEY, TEXT).set(TEXT, "test").build(); ②

datastore.insert(DataTarget.named("tableName"), value, DefaultWriteOption
.BRING_BACK_GENERATED_IDS); ③

Long keyValue = value.getValue(KEY); ④
```

① The `key` column is supposed to be auto-generated by the database

② Create the `PropertyBox` value to insert, not providing the `key` value

③ Perform the *insert* operation, providing the **BRING_BACK_GENERATED_IDS** write option

④ The `KEY` property value of the inserted `PropertyBox` is updated with the auto-generated key value, if available

## 5.6. Custom SQL filters and sorts

The JDBC `Datastore` API supports custom **SQL** filters and sorts expressions, i.e. `QueryFilter` and `QuerySort` type expressions for which the **SQL statement** that represents the `WHERE` and `ORDER BY` conditions is directly provided.



Since this kind of expressions accept SQL statements without performing any validation or parsing, you must pay maximum attention using them, since this way the code can be exposed to security concerns such as **SQL injection type attacks**.

### 5.6.1. WhereFilter

The WhereFilter interface is a `QueryFilter` representing a portion of the SQL `WHERE` clause expression.

```
QueryFilter whereFilter = WhereFilter.create("name='John'"); ①

Stream<Long> results = getDatastore().query().target(TARGET).filter(whereFilter)
.stream(ID); ②
```

① Create a `WhereFilter`, providing the SQL statement to be included as is in the query `WHERE` clause

② The `QueryFilter` can now be used in Datastore expressions just like any other filter expression

The `WhereFilter` expression supports **query parameters**, which must be expressed in the SQL statement using the default `?` placeholder. The parameters values can be setted using the `create` WhereFilter builder method:

```
QueryFilter whereFilter = WhereFilter.create("name=?", "John"); ①
```

① Create a `WhereFilter` providing the SQL statement (using the `?` parameter placeholder) and `John` as parameter value



The SQL `?` parameter placeholders are replaced with the parameters values in the same order they are provided in the `create` WhereFilter builder method

### 5.6.2. OrderBySort

The OrderBySort interface is a `QuerySort` representing a portion of SQL `ORDER BY` clause expression.

An `OrderBySort` expression can be created using the `create` builder method.

```
QuerySort orderBySort = OrderBySort.create("id asc, name desc"); ①

Stream<Long> results = getDatastore().query().target(TARGET).sort(orderBySort).stream
(ID); ②
```

① Create a SQL order by expression to order by the `id` value ascending and the `name` value descending

② The `QuerySort` can now be used in Datastore expressions just like any other sort expression

## 5.7. Transactions management

The JDBC `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 = getDatastore(); // build or obtain a JDBC 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

## 5.8. `JdbcDatastore` API

The specialized JdbcDatastore API, which extends the standard `Datastore` API, makes available an additional methods through the ConnectionHandler interface to **execute an operation using a Datastore managed JDBC connection**.

The JDBC Datastore will take care of connection providing and finalization (i.e. connection closing operations).

```
JdbcDatastore datastore = getJdbcDatastore();

datastore.withConnection(connection -> { ①
  // do something using the provided JDBC connection
  connection.createStatement();
});

String result = datastore.withConnection(connection -> { ②
  // do something using the provided JDBC connection and return a result
  return null;
});
```

① Execute an operation using a JDBC Datastore managed connection

② Execute an operation using a JDBC Datastore managed connection and return a `String` type

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 `JdbcDatastore` 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.

# 5.9. Extending the JDBC Datastore API

# 5.10. 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.

## 5.10.1. JDBC Expression resolvers registration

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

**1. Using the JDBC `Datastore` API instance:**

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

```
Datastore datastore = JdbcDatastore.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 JDBC Datastore supports `ExpressionResolver` automatic registration using the JdbcDatastoreExpressionResolver base type and the default *Java service extensions* modality.

To automatically register an `ExpressionResolver` this way, a class implementing `JdbcDatastoreExpressionResolver` has to be created and its fully qualified name must be specified in a file named `com.holonplatform.datastore.jdbc.config.JdbcDatastoreExpressionResolver`, 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 JDBC `Datastore` API instance**.

### 5.10.2. 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

### 5.10.3. 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 JDBC 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`).

### 5.10.4. Expression validation

The internal JDBC Datastore *SQL 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.

### 5.10.5. JDBC 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 JDBC `Datastore` API can be extended using a set of **specific SQL resolution expressions**, used by the internal *SQL composer engine* to implement the resolution and

composition strategy to obtain SQL statements from the `Datastore` API meta-language expressions.

These SQL expression are available from the `com.holonplatform.datastore.jdbc.composer.expression` package of the `holon-datastore-jdbc-composer` artifact.

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

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

For example, supposing to have a `KeyIs` class which represents a `QueryFilter` expression type to represent the expression *"the `key` column name 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 **SQL WHERE statemet part**, using the `SQLExpression` type. Using the convenience `create` method of the `ExpressionResolver` API, we can do it in the following way:

```
final ExpressionResolver<KeyIs, SQLExpression> keyIsResolver = ExpressionResolver
.create( //
    KeyIs.class, ①
    SQLExpression.class, ②
    (keyIs, ctx) -> Optional.of(SQLExpression.create("key = " + keyIs.getValue())));
③
```

① Expression type to resolve

② Target expression type

③ Expression resolution logic: since we resolve the `KeyIs` expression directly into `SQLExpression` type, the SQL WHERE clause part is provide

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 exmaple, in a `Query` expression:

```
Datastore datastore = JdbcDatastore.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 *SQL composer engine* will translate the given `KeyIs` expression in the SQL WHERE statement part `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* `SQLExpression` 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 SQL statement part representation (i.e. the `SQLExpression` type).

Some examples are:

- SQLLiteral to represent a *literal* value.
- SQLParameter to represent a *parameter* value.
- SQLFunction to represent a *SQL function*.
- SQLProjection to represent a *SQL SELECT* projection.
- SQLStatement to represent a full *SQL statement* with parameters support.
- SQLQueryDefinition and SQLQuery to represent the definition of a *SQL query* and the query statement representation with the `SQLResultConverter` to use to covert the query results in the expected Java type.

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

We want to define a **IfNull** function, which returns a *fallback* value expression when a given *expression* is *null*, following the syntax of the MySQL IFNULL 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 IfNull<T> implements QueryFunction<T, T> {

  private final TypedExpression<T> nullableValue; ①
  private final TypedExpression<T> fallbackValue; ②

  public IfNull(TypedExpression<T> nullableValue, TypedExpression<T> fallbackValue) {
    super();
    this.nullableValue = nullableValue;
    this.fallbackValue = fallbackValue;
  }

  public IfNull(QueryExpression<T> nullableValue, T fallbackValue) { ③
    this(nullableValue, ConstantExpression.create(fallbackValue));
  }

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

  @Override
  public void validate() throws InvalidExpressionException { ④
    if (nullableValue == null) {
      throw new InvalidExpressionException("Missing nullable expression");
    }
    if (fallbackValue == null) {
      throw new InvalidExpressionException("Missing fallback expression");
    }
  }

  @Override
  public List<TypedExpression<? extends T>> getExpressionArguments() { ⑤
    return Arrays.asList(nullableValue, fallbackValue);
  }

}
```

① The *nullable* expression representation

② The *fallback* expression representation to be returned when the first expression value is *null*

③ Convenience constructor to accept a *constant* expression value as fallback expression

④ Validate the arguments

⑤ The two function *arguments* will be the *nullable* expression and the *fallback* expression

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

```
public class IfNullResolver implements ExpressionResolver<IfNull, SQLFunction> {

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

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

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

}
```

① Use the `SQLFunction create` method to provide the SQL function representation using the function arguments. The function arguments are already provided by the SQL composition engine as serialized SQL 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("str");
final DataTarget<?> TARGET = DataTarget.named("test");

Datastore datastore = JdbcDatastore.builder() //
    .withExpressionResolver(new IfNullResolver()) ①
    .build();

Stream<String> values = datastore.query(TARGET).stream(new IfNull<>(STR, "(fallback)"
)); ②
```

① Register the `IfNull` function expression resolver in Datastore

② The `IfNull` function is used to provide a fallback `(fallback)` value in a query projection for the `STR` String property value

### 5.10.6. JDBC Expression resolution context

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

When used within the JDBC `Datastore` API, the `ResolutionContext` provided to the registered expresion resolvers is `SQLCompositionContext` for any JDBC `Datastore` API standard operation. To obtain a `SQLCompositionContext`, the `isSQLCompositionContext` method can be used.

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

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

  SQLCompositionContext.isSQLCompositionContext(context).ifPresent(ctx -> { ②
    SQLDialect dialect = ctx.getDialect(); ③

    ctx.isStatementCompositionContext().ifPresent(sctx -> { ④
      sctx.addNamedParameter(SQLParameter.create("test", String.class)); 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 `SQLCompositionContext`

③ As an example, the current `SQLDialect` is requested

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

⑤ Add a *named* parameter to the current SQL statement context

## 5.11. Commodity factories

The JDBC `Datastore` API supports *Datastore commodities* registration using the JdbcDatastoreCommodityFactory 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 `JdbcDatastoreCommodityFactory` type provides a specialized JdbcDatastoreCommodityContext

API as Datastore commodity context, to make available a set of JDBC `Datastore` specific configuration attributes and references, for example:

- The `DataSource` instance used by the JDBC `Datastore`.
- The configured (or auto-detected) SQL dialect.
- The configured (or auto-detected) database platform.
- The configured identifier resolution strategy.
- The available *expression resolvers*
- A set of APIs and configuration methods available from the *SQL composer engine*.

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

```
interface MyCommodity extends DatastoreCommodity { ①

  DatabasePlatform getPlatform();

}

class MyCommodityImpl implements MyCommodity { ②

  private final DatabasePlatform platform;

  public MyCommodityImpl(DatabasePlatform platform) {
    super();
    this.platform = platform;
  }

  @Override
  public DatabasePlatform getPlatform() {
    return platform;
  }

}

class MyCommodityFactory implements JdbcDatastoreCommodityFactory<MyCommodity> { ③

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

  @Override
  public MyCommodity createCommodity(JdbcDatastoreCommodityContext context)
      throws CommodityConfigurationException {
    // examples of configuration attributes:
    DataSource dataSource = context.getDataSource();
    SQLDialect dialect = context.getDialect();
    return new MyCommodityImpl(context.getDatabase().orElse(DatabasePlatform.NONE));
  }

}
```

① Datastore commodity API

② Commodity implementation

③ Commodity factory implementation

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

**1. Direct registration using the JDBC `Datastore` API builder:**

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

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

① Register the `MyCommodityFactory` commodity factory in given JDBC `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 `JdbcDatastoreCommodityFactory` has to be created and its qualified full name must be specified in a file named `com.holonplatform.datastore.jdbc.config.JdbcDatastoreCommodityFactory`, 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 JDBC `Datastore` API instance**.

# 6. Spring ecosystem integration

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

*Maven coordinates*:

```
<groupId>com.holon-platform.jdbc</groupId>
<artifactId>holon-datastore-jdbc-spring</artifactId>
<version>5.1.3</version>
```

## 6.1. Integration with the Spring JDBC infrastructure

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

In particular, the JDBC connection handler used by the Spring JDBC `Datastore` API implementations in fully integrated with the Spring JDBC infrastructure, for example regarding the current `Connection` and transaction synchronization support.

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

## 6.2. JDBC Datastore configuration

The EnableJdbcDatastore annotation can be used on Spring configuration classes to enable automatic JDBC Datastore beans configuration.



By default, the JDBC Datastore bean name configured with the `@EnableJdbcDatastore` annotation will be `jdbcDatastore`.

### 6.2.1. DataSource

The `DataSource` bean to be used to configure the JDBC Datastore API is obtained as follows:

- If the `dataSourceReference` attribute of the `@EnableJdbcDatastore` is setted, the provided **bean definition name** will be used as the `DataSource` bean definition name to use.

```
@EnableJdbcDatastore(dataSourceReference = "myDataSource") ①
@Configuration
class Config {

  @Bean(name = "myDataSource")
  public DataSource myDataSource() {
    return buildDataSource();
  }

}

@Autowired
Datastore datastore; ②
```

① Provide the `DataSource` bean definition name to use with the JDBC Datastore

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

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

```
@EnableJdbcDatastore
@Configuration
class Config {

  @Bean
  public DataSource dataSource() { ①
    return buildDataSource();
  }

}

@Autowired
Datastore datastore; ②
```

① The default `dataSource` bean definition name is used for the `DataSource` bean definition

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

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



You can use the `@EnableDataSource` provided by the Holon Platform JDBC Module to auto configure the `DataSource` bean to be used with the JDBC Datastore.

```
@EnableDataSource ①
@EnableJdbcDatastore ②
@PropertySource("jdbc.properties") ③
@Configuration
class Config {

}

@Autowired
DataSource dataSource;

@Autowired
Datastore datastore;
```

① Use the `@EnableDataSource` annotation to configure a `DataSource` bean using the Spring environment configuration properties

② Use the `@EnableJdbcDatastore` annotation to configure a JDBC `Datastore` bean backed by the previously configured `DataSource`

③ Configuration properties source

### 6.2.2. Multiple JDBC Datastores configuration

When more than one JDBC Datastore bean has to be configured using the `@EnableJdbcDatastore` annotation, the `dataContextId` attribute can be used to assign a different **data context id** to each

JDBC 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 `DataSource` bean definition to use for each JDBC Datastore to configure: if not directly specified with the `dataSourceReference` attribute, the `DataSource` bean definition to use for each JDBC Datastore will be detected in Spring context using the bean name pattern: `dataSource_{datacontextid}` where `{datacontextid}` is equal to the `dataContextId` attribute of the `@EnableJdbcDatastore` 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 JDBC `Datastore` bean definitions, and such qualifier can be later used to obtain the right `Datastore` instance through dependency injection.

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

```
@Configuration class Config {

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

    @Bean(name = "dataSource_one")
    public DataSource dataSource() {
      return buildDataSource();
    }

  }

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

    @Bean(name = "dataSource_two")
    public DataSource dataSource() {
      return buildDataSource();
    }

  }

}

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

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

① Configure the first JDBC Datastore using `one` as *data context id*: by default the bean named `dataSource_one` will be used as `DataSource`

② Configure the first JDBC Datastore using `two` as *data context id*: by default the bean named `dataSource_two` will be used as `DataSource`

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

When using the `@EnableDataSource` provided by the Holon Platform JDBC Module to auto configure the `DataSource` bean, a matching *data context id* value can be provided to match the JDBC Datastore beans with the corresponding `DataSource` beans.

For example, given a `jdbc.properties` configuration property source file defined as follows:

```
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
```

The `@EnableDataSource` and `@EnableJdbcDatastore` annotations can be used to configure two DataSource/Datastore pairs, one using the `one` *data context id* and the other using the `two` *data context id*:

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

  @Configuration
  @EnableDataSource(dataContextId = "one")
  @EnableJdbcDatastore(dataContextId = "one")
  static class Config1 {
  }

  @Configuration
  @EnableDataSource(dataContextId = "two")
  @EnableJdbcDatastore(dataContextId = "two")
  static class Config2 {
  }

}

@Autowired
@Qualifier("one")
DataSource dataSource1;
@Autowired
@Qualifier("one")
Datastore datastore1;

@Autowired
@Qualifier("two")
DataSource dataSource2;
@Autowired
@Qualifier("two")
Datastore datastore2;
```

### 6.2.3. Database platform

By default, the *database platform* to which the `DataSource` is bound is **auto-detected** by the JDBC Datastore inspecting the JDBC connection URL (see Database platform configuration).

The *database platform* is used by the JDBC Datastore to select the most suitable SQL Dialect to use, if not otherwise specified.

To explicitly specify a *database platform*, the `platform` attribute of the `@EnableJdbcDatastore` annotation can be used.

```
@EnableJdbcDatastore(platform = DatabasePlatform.H2) ①
@Configuration
class Config {

  // ...

}
```

① Explicitly set `H2` as database platform

### 6.2.4. Identifier resolution strategy

The the `identifierResolutionStrategy` attribute of the `@EnableJdbcDatastore` annotation can be used to specify **identifier resolution strategy** to use with the configured JDBC Datastore.

See the Identifier resolution strategy section for information about the identifier resolution strategies.

### 6.2.5. Transactional JDBC Datastore operations

The `transactional` attribute of the `@EnableJdbcDatastore` 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**

```
@EnableDataSource
@EnableJdbcDatastore
@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:

```
@EnableJdbcDatastore(transactional = false)
@Configuration
class Config {

}
```

### 6.2.6. Primary mode

The `@EnableJdbcDatastore` annotation provides a `primary()` attribute which can be used to control the *primary mode* of the JDBC `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 JDBC Datastore bean will be marked as **primary** only when the `DataSource` bean to which is bound is registered as primary candidate bean.

### 6.2.7. JDBC Datastore configuration properties

When a JDBC Datastore bean is configured using the `@EnableJdbcDatastore` 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 3. 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 *SQL Dialect* to use. See SQL Dialect configuration. |

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

*Table 4. JDBC Datastore configuration properties*

| Name | Type | Meaning |
|---|---|---|
| *holon.datastore.jdbc.* **platform** | A valid `String` which identifies one of the platform names listed in the `DatabasePlatform` enumeration. | Set the *database platform* to which the Datastore is bound. |
| *holon.datastore.jdbc.* **primary** | Boolean (`true` / `false`) | Mark the JDBC 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 JDBC Datastore bean will be marked as primary only when the `DataSource` bean to which is bound is registered as primary candidate bean. |
| *holon.datastore.jdbc.* **identifier-resolution-strategy** | A valid `String` which identifies one of the strategy names listed in the `IdentifierResolutionStrategy` enumeration. | Set the JDBC Datastore *identifier resolution strategy*. See Identifier resolution strategy. |

| Name | Type | Meaning |
| --- | --- | --- |
| *holon.datastore.jdbc.* **transactional** | Boolean (`true` / `false`) | Whether to add the Spring `@Transactional` behavior to th suitable Datastore API methods. See Transactional JDBC Datastore operations. |

Example of Datastore configuration properties:

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

holon.datastore.jdbc.platform=H2 ③
holon.datastore.jdbc.identifier-resolution-strategy=TABLE_PRIMARY_KEY ④
holon.datastore.jdbc.transactional=false ⑤
```

① Enable tracing

② Set the dialect class name

③ Set the database platform

④ Use the `TABLE_PRIMARY_KEY` identifier resolution strategy

⑤ Disable the automatic *transactional* behavior of the Datastore operations

### 6.2.8. Datastore extension and configuration using the Spring context

The JDBC 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 JdbcDatastoreCommodityFactory<MyCommodity> {

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

  @Override
  public MyCommodity createCommodity(JdbcDatastoreCommodityContext 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

# 7. Spring Boot integration

The `holon-datastore-jdbc-spring-boot` artifact provides integration with Spring Boot for JDBC Datastores **auto-configuration**.

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

*Maven coordinates*:

```
<groupId>com.holon-platform.jdbc</groupId>
<artifactId>holon-datastore-jdbc-spring-boot</artifactId>
<version>5.1.3</version>
```

A JDBC datastore is auto-configured only when:

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



The `holon-datastore-jdbc-spring-boot` declares as dependency the `holon-jdbc-spring-boot` artifact, which provides Spring Boot auto-configuration features for `DataSource` type beans made available from the Holon Platform JDBC Module.

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

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

  datastore:
    trace: true
```

The Holon platform Spring Boot auto-configuration classes will auto configure:

- A `DataSource` type bean using the provided `holon.datasource.*` configuration properties.
- A JDBC `Datastore` type bean backed by such `DataSource`, enabling operations *tracing* according to the `holon.datastore.trace` property value.

```
@Configuration
@EnableAutoConfiguration
class Config {

}

@Autowired
Datastore datastore; ①
```

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

Of course, you can use **default Spring Boot `DataSource` auto-configuration features** to create the `DataSource` bean:

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

holon:
  datastore:
    trace: true
```

## 7.1. Multiple JDBC Datastore 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 JDBC Datastores is available out-of-the-box.

The ***data context id* convention** is used to provide multiple `DataSource` and `Datastore` 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 JDBC `Datastore` bean for each detected `DataSource` bean, binding the same *data context id* to the JDBC Datastore bean definitions.

According to the *data context id* convention, each `DataSource` and JDBC `Datastore` pair 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 JDBC `Datastore` beans:

- 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 the `DataSource` and JDBC `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;
```

## 7.2. Disabling the JDBC Datastore auto-configuration feature

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

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

## 7.3. Spring Boot starters

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

**1. Default JDBC Datastore starter** provides the dependencies to the Holon JDBC Datastore Spring Boot integration artifact `holon-datastore-jdbc-spring-boot`, in addition to:

- The Holon Platform JDBC Module Spring Boot integration base starter (`holon-starter-jdbc`).
- 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.

*Maven coordinates*:

```
<groupId>com.holon-platform.jdbc</groupId>
<artifactId>holon-starter-jdbc-datastore</artifactId>
<version>5.1.3</version>
```

**2. JDBC Datastore starter with HikariCP DataSource** provides the same dependencies as the default JDBC Datastore starter, adding the HikariCP pooling DataSource dependency.

This way, the *HikariCP* `DataSource` will be selected by default by the `DataSource` auto-configuration strategy if the `DataSource` *type* is not explicitly specified using the corresponding configuration property.



See the Holon Platform JDBC Module - Datasource type documentation for details about `DataSource` types.

*Maven coordinates*:

```
<groupId>com.holon-platform.jdbc</groupId>
<artifactId>holon-starter-jdbc-datastore-hikaricp</artifactId>
<version>5.1.3</version>
```

# 8. 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 **JDBC Datastore** module is `com.holonplatform.datastore.jdbc`.

# 9. System requirements

## 9.1. Java

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

## 9.2. JDBC Drivers

To retrieve back the database generated keys, the JDBC driver in use must be compliant to the **JDBC API version 3 or higher**.