

Holon
PLATFORM

Holon JDBC Datastore

Version 5.0.5

Table of Contents

1. Introduction . . . 1
2. Obtaining the artifacts . . . 1
   2.1. Using the Platform BOM . . . 2
3. Datastore . . . 2
   3.1. Setup . . . 3
   3.2. Data targets and paths . . . 4
   3.3. Relational expressions . . . 4
   3.4. Dialects . . . 4
   3.5. Auto-generated ids . . . 5
   3.6. JDBC filters and sorts . . . 6
      3.6.1. JdbcWhereFilter . . . 6
      3.6.2. JdbcOrderBySort . . . 6
   3.7. Extensions . . . 6
   3.8. Expression resolvers . . . 6
   3.9. Commodity factories . . . 6
4. Spring framework integration . . . 7
   4.1. Datastore setup . . . 7
   4.2. Datastore auto-configuration . . . 7
   4.3. Commodity factories . . . 8
5. Spring Boot integration . . . 8
   5.1. Spring Boot starters . . . 9
6. Loggers . . . 10
7. System requirements . . . 10
   7.1. Java . . . 10
   7.2. JDBC Drivers . . . 10

Copyright © 2016-2017

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 abstraction.



See the Datastore documentation for further information about **Datastores**.

This Datastore uses the Java **DataSource** interface as a factory to obtain database connections and perform operations on the persistence store using the *SQL* language through the JDBC API.

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.jdbc</groupId>
<artifactId>holon-datastore-jdbc-bom</artifactId>
<version>5.0.5</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>5.0.5</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. Datastore

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

Maven coordinates:

```
<groupId>com.holon-platform.jdbc</groupId>
<artifactId>holon-datastore-jdbc</artifactId>
<version>5.0.5</version>
```

The JdbcDatastore interface represents the **JDBC Datastore**, which extends the core **Datastore** interface.

The `withConnection` method can be used to execute a JDBC operation using a Datastore managed `java.sql.Connection`, providing the operation execution code through the `ConnectionOperation` functional interface. The connection finalization operations are executed by the JDBC Datastore after the execution of the provided operation.

```
JdbcDatastore datastore = getJdbcDatastore(); // build or obtain a JdbcDatastore

String name = datastore.withConnection(connection -> {
  try (ResultSet rs = connection.createStatement().executeQuery("select name from test
where id=1")) {
    rs.next();
    return rs.getString(1);
  }
});
```



If you want to reach the goal of a ***complete abstraction*** from the persistence store technology and the persistence model, the core **Datastore** interface should be used as a reference for persistence operations, instead of the specific **JdbcDatastore** interface. This way, the concrete Datastore implementation may be replaced by a different one at any time, without any change to rest of the code.

3.1. Setup

The JDBC Datastore supports default Datastore configuration properties:

- `holon.datastore.trace`: To enable/disable Datastore operations tracing
- `holon.datastore.dialect`: To configure the JDBC dialect class to use (providing the fully qualified class name). See Dialects for the available default JDBC dialects.

To create a **JdbcDatastore** instance, the `builder()` static method of the interface can be used.

```
JdbcDatastore datastore = JdbcDatastore.builder() // obtain the builder
    .dataSource(getDataSource()) ①
    .database(DatabasePlatform.ORACLE) ②
    .build();

datastore = JdbcDatastore.builder() // obtain the builder
    .dataSource(DataSourceConfigProperties.builder().withPropertySource(
"jdbc.properties").build()) ③
    .dialect(new MyDialect()) ④
    .build();

datastore = JdbcDatastore.builder() // obtain the builder
    .dataContextId("db1") ⑤
    .dataSource(DataSourceConfigProperties.builder("db1").withPropertySource(
"jdbc.properties").build()) ⑥
    .configuration(
        DatastoreConfigProperties.builder("db1").withPropertySource(
"datastore.properties").build()) ⑦
    .build();
```

① Set an explicit `DataSource` instance to use

② Set the database platform: if no dialect is explicitly provided, the Datastore will try to auto-detect the dialect to use relying on the database platform type

③ Provide a `DataSourceConfigProperties` structure using the `jdbc.properties` file, to build and configure a DataSource according to the available properties. See JDBC DataSource

④ Set a custom dialect

⑤ Build a Datastore bound to a *data context id*, which will be used to distinguish different data sources

⑥ Build a `DataSource` using given *data context id* and associate it to the Datastore. In this case, the configuration properties are expected to be expressed with the *data context id* as prefix, for example `holon.datasource.db1.url=⋯`

3.2. Data targets and paths

The JDBC Datastore relies on the following conventions regarding *DataTarget*s and *Path*s:

- The **DataTarget** *name* is interpreted as the database *table* (or *view*) name
- The **Path** *name* is interpreted as a table *column* name



See the core Datastore documentation for more informations about **DataTarget*s and *Path*s, and about the use of extension hooks such as *DataTargetResolver** for target names resolution.

3.3. Relational expressions

As a *relational* Datastore, the JDBC Datastore 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 Sub query documentation for further information and code examples.

2. Alias and Joins:

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

See the Alias and Joins documentation for further information and code examples.

3.4. Dialects

The JDBC Datastore relies on **dialects** to address and resolve any SQL language dissimilarity between RDBMS platforms.

A dialect is represented and implemented by the JdbcDialect interface.

The currently available 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	11.5 and higher
MySQL	MySQLDialect	4.1 and higher
MariaDB	MariaDBDialect	5.5 and higher
SAP HANA	HANADialect	1.0 SPS12 and higher
Oracle Database	OracleDialect	9i and higher
PostgreSQL	PostgreSQLDialect	8.2.5 and higher
Microsoft SQL Server	SQLServerDialect	2005 or higher
SQLite	SQLiteDialect	3.0.7. and higher

3.5. Auto-generated ids

The JDBC datastore support 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()` method.

The default **BRING_BACK_GENERATED_IDS** `WriteOption` can be provided to Datastore data manipulation method to bring back any auto-generated id value into the `PropertyBox` which was subject of the operation, if a corresponding `Property` (using the property name) is available in the box 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 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` to insert, not providing the `key` value

③ Execute the insert operation using the **BRING_BACK_GENERATED_IDS** write option

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

3.6. JDBC filters and sorts

Two interfaces are available to create filter and sort expression using the SQL language, extending the `QueryFilter` and `QuerySort` expressions:

3.6.1. JdbcWhereFilter

The JdbcWhereFilter interface is a `QueryFilter` representing a SQL *where* clause expression.

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

```
QueryFilter filter = JdbcWhereFilter.create("name=? and id=?", "TestName", 1); ①
```

① Create a SQL filter expression with to parameters, providing `TestName` as first parameter value and `1` as second parameter value

3.6.2. JdbcOrderBySort

The JdbcOrderBySort interface is a `QuerySort` representing a SQL *order by* clause expression.

```
QuerySort sort = JdbcOrderBySort.create("id asc, name desc"); ①
```

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

3.7. Extensions

3.8. Expression resolvers

The JDBC Datastore supports `ExpressionResolver` automatic registration using the JdbcDatastoreExpressionResolver base type and default *Java service extensions*.

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

3.9. Commodity factories

The JDBC Datastore supports `DatastoreCommodityFactory` automatic registration using the JdbcDatastoreCommodityFactory base type and default *Java service extensions*.

To automatically register an `DatastoreCommodityFactory` this way, 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 in classpath.

The JdbcDatastoreCommodityContext interface represents the JDBC Datastore specific commodity context and it is provided at commodity creation time to factories.

The context extends the `JdbcDatastore` interface itself and provides the following additional resources:

- The `DataSource` bound the JDBC Datastore;
- The `DatabasePlatform` to which the DataSource is connected, if available;
- The `JdbcDialect` used by the JDBC Datastore;
- Whether the Datastore *trace* mode is enabled.

4. Spring framework integration

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

Maven coordinates:

```
<groupId>com.holon-platform.jdbc</groupId>
<artifactId>holon-datastore-jdbc-spring</artifactId>
<version>5.0.5</version>
```

4.1. Datastore setup

To create a JDBC Datastore and register it as a Spring bean, the SpringJdbcDatastore interface is provided, with the convenience `builder()` method.

This interface creates and represents a JDBC Datastore implementation wich supports Spring JDBC and transaction management architecture, for example the `DataSource` connection and transaction synchronization support to provide a consistent integration with the Spring infrastructure.

4.2. Datastore auto-configuration

The EnableJdbcDatastore annotation can be used on Spring configuration classes to enable automatic JDBC Datastore configuration. An available `DataSource` bean must be present in context to enable the JDBC Datastore.

The *data context id* to which the JDBC Datastore is bound can be configured using the `dataContextId` annotation attribute, useful when multiple `DataSource` are available and it is required to configure multiple JDBC Datastore instances.

When a *data context id* is not specified, the JDBC Datastore is bound to the unique `DataSource` type bean registered in context. If the bean is not unique or is not present, a configuration error is

thrown. The `dataSourceReference` annotation attribute can be used to specify the explicit `DataSource` bean name to use for the JDBC Datastore.

When a *data context id* is specified, the registered Datastore is bound to the `DataSource` with a matching *data context id*, if available. During the registration phase, if a `dataSourceReference` is not specified, a `DataSource` bean is searched in context using the bean name pattern: `dataSource_[datacontextid]` where `[datacontextid]` is equal to the `dataContextId` annotation attribute.

A `DatabasePlatform` can be specified using the `platform` annotation attribute and to select a suitable dialect for the given platform. If the database platform is not specified, the JDBC Datastore tries to auto-detect it from the `DataSource` configuration.

The `transactional` annotation attribute (`true` by default) can be used to control the Spring transactions architecture integration, i.e. if a `Transactional` behaviour must be configured for the JDBC Datastore data manipulation methods, to automatically create or partecipate in a Spring transaction when these methods are invoked.

```
@EnableJdbcDatastore
@EnableDataSource
@PropertySource("datasource.properties")
@Configuration
class Config {

}

class MyBean {

  @Autowired
  private Datastore datastore;

}
```

4.3. Commodity factories

To use the `DatastoreCommodityFactory` annotation on beans to automatically register them into the `Datastore`, the `JdbcDatastoreCommodityFactory` base type must be used for such beans.

See Commodity factories for additional details about the `JdbcDatastoreCommodityFactory` type.

5. Spring Boot integration

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

To enable Spring Boot auto-configuration 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>5.0.5</version>
```

The JDBC datastore is auto-configured only when:

- A `JdbcDatastore` type bean is not already registered in Spring context
- A valid `DataSource` type bean is available in Spring context

When multiple `DataSource` type beans are registered in Spring context, and each of them is bound to a *data context id* (if they were configured using the Holon platform `DataSource` configuration modules), a JDBC Datastore is automatically configured and registered for each `DataSource` bean, using the *data context id* as Spring bean **qualifier** name.

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

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

5.1. 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 artifacts, in addition to default Holon *core* and *JDBC* Spring Boot starters (see the documentation for further information) and base Spring Boot starter (`spring-boot-starter`):

Maven coordinates:

```
<groupId>com.holon-platform.jdbc</groupId>
<artifactId>holon-starter-jdbc-datastore</artifactId>
<version>5.0.5</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.

Maven coordinates:

```
<groupId>com.holon-platform.jdbc</groupId>
<artifactId>holon-starter-jdbc-datastore-hikaricp</artifactId>
<version>5.0.5</version>
```

6. 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`.

7. System requirements

7.1. Java

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

7.2. JDBC Drivers

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