

Holon
PLATFORM

Holon Platform JDBC support

Version 5.0.4

Table of Contents

1. Introduction . . . 1
2. Obtaining the artifacts . . . 1
   2.1. Using the Platform BOM . . . 2
3. JDBC DataSource configuration . . . 2
   3.1. DataSource builder . . . 2
      3.1.1. Default DataSource type selection strategy . . . 3
      3.1.2. DataSourceFactory . . . 4
      3.1.3. DataSourcePostProcessor . . . 4
   3.2. DataSource configuration properties . . . 5
   3.3. Multiple DataSource configuration . . . 6
   3.4. BasicDataSource . . . 7
4. Multi-tenant DataSource . . . 8
5. Spring framework integration . . . 8
   5.1. DataSource auto-configuration . . . 8
   5.2. Additional configuration properties . . . 11
6. Spring Boot integration . . . 13
   6.1. Spring Boot starters . . . 14
7. Loggers . . . 14
8. System requirements . . . 14
   8.1. Java . . . 14

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 Platform **JDBC** module provides base JDBC support to the Holon platform, dealing with `javax.sql.DataSource` configuration and providing a *multi-tenant* DataSource implementation.

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-jdbc-bom</artifactId>
<version>5.0.4</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-jdbc-bom</artifactId>
      <version>5.0.4</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. JDBC DataSource configuration

The Holon platform provides an API to create and configure JDBC `DataSource` instances using a set of configuration properties.

Maven coordinates:

```
<groupId>com.holon-platform.jdbc</groupId>
<artifactId>holon-jdbc</artifactId>
<version>5.0.4</version>
```

3.1. DataSource builder

The DataSourceBuilder interface can be used to build `DataSource` instances using a `DataSourceConfigProperties` instance as configuration property source, using the `build(DataSourceConfigProperties configurationProperties)` method.

The default `DataSourceBuilder` implementation, which can be obtained through the `create()` interface static methods, relies on the `type` configuration property to define the concrete `DataSource` instance to create and configure.

The following types are supported by default:

- `com.holonplatform.jdbc.BasicDataSource`: Create `BasicDataSource` instances, to be used typically for testing purposes. It is a simple `DataSource` implementation, using the `java.sql.DriverManager` class and returning a new `java.sql.Connection` from every `getConnection` call.
- `com.zaxxer.hikari.HikariDataSource`: Create HikariCP connection pooling `DataSource` instances. The *HikariCP* library dependency must be available in classpath. All default configuration properties are supported, and additional Hikari-specific configuration properties can be specified using the `hikari` prefix before the actual property name, for example: `holon.datasource.hikari.connectionTimeout`.
- `org.apache.commons.dbcp2.BasicDataSource`: Create Apache Commons DBCP 2 connection pooling `DataSource` instances. The *DBCP 2* library dependency must be available in classpath. All default configuration properties are supported, and additional DBCP-specific configuration properties can be specified using the `dbcp` prefix before the actual property name, for example: `holon.datasource.dbcp.maxWaitMillis`.
- `org.apache.tomcat.jdbc.pool.DataSource`: Create Tomcat JDBC connection pooling `DataSource` instances. The *tomcat-jdbc* library dependency must be available in classpath. All default

configuration properties are supported, and additional Tomcat-specific configuration properties can be specified using the `tomcat` prefix before the actual property name, for example: `holon.datasource.tomcat.maxAge`.

- `JNDI`: Obtain a `DataSource` using **JNDI**. The `jndi-name` configuration property is required to specify the JNDI name to which the `DataSource` is bound in the JNDI context.

3.1.1. Default DataSource type selection strategy

If the `type` configuration property is not specified, the default DataSource `type` selection strategy adopted by the `DataSourceBuilder` is defined as follows:

1. If the HikariCP dependecy is present in classpath, the `com.zaxxer.hikari.HikariDataSource` type will be used;
2. If the Apache Commons DBCP 2 dependecy is present in classpath, the `org.apache.commons.dbcp2.BasicDataSource` type will be used;
3. If the Tomcat JDBC dependecy is present in classpath, the `org.apache.tomcat.jdbc.pool.DataSource` type will be used;
4. Otherwise, the `com.holonplatform.jdbc.BasicDataSource` type is used as fallback.

For example, using the following configuration properties file:

datasource.properties

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

The `DataSourceBuilder` can be used as follows, creating a DataSource of a type determined according to the default type selection strategy:

```
DataSourceConfigProperties config = DataSourceConfigProperties.builder()
    .withPropertySource("datasource.properties").build(); ①

DataSource dataSource = DataSourceBuilder.create().build(config); ②
```

① Create a configuration property set using the `datasource.properties` as property source

② Build a `DataSource` instance according to given configuration properties

To specify the `DataSource` type to create, the `type` property can be used. For example, to use HikariCP pooling DataSource:

datasource2.properties

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

holon.datasource.type=com.zaxxer.hikari.HikariDataSource
```

The `DataSourceBuilder` is used the same as before:

```
DataSourceConfigProperties config = DataSourceConfigProperties.builder()
    .withPropertySource("datasource2.properties").build(); ①

DataSource dataSource = DataSourceBuilder.create().build(config); ②
```

① Create a configuration property set using the `datasource2.properties` as property source

② Build a `DataSource` instance according to given configuration properties, which will be an `HikariDataSource` instance



In order to create `HikariDataSource` instances, the HikariCP library must be present in classpath. Similarly, to create Tomcat JDBC pooling DataSources, the Tomcat JDBC library must be present in classpath.

3.1.2. DataSourceFactory

The default `DataSourceBuilder` implementation delegates `DataSource` instances creation to DataSourceFactory interface implementation, each of them bound to a single `DataSource` **type** name, which must be registered in the `DataSourceBuilder`.

A `DataSourceFactory` can be used to provide additional `DataSource` types support or to replace a default type creation strategy with a new one. The `DataSource` type name to which the `DataSourceFactory` is bound is provided by the `getDataSourceType()` interface method.

The registration of a `DataSourceFactory` can be accomplished in two ways:

- **Direct registration**: A `DataSourceFactory` instance can directly registered in a `DataSourceBuilder` instance using the `registerFactory(DataSourceFactory factory)` method. Any previous binding with given type will be replaced by the given factory.
- **Extension services**: The default Java `ServiceLoader` extensions can be used, providing a `com.holonplatform.jdbc.DataSourceFactory` file under a `META-INF/services` folder in classpath, in which to specify the fully qualified name of the `DataSourceFactory` implementation. This way, the factory is automatically registered when the builder instance is initialized.

3.1.3. DataSourcePostProcessor

The DataSourcePostProcessor interface can be used to perform additional initialization and configuration on a {@link DataSource} instance created using the `DataSourceBuilder`. The

`postProcessDataSource(⋯)` method is called just after the creation of any DataSource instance. In order to activate a `DataSourcePostProcessor`, it must be registered in the `DataSourceBuilder` instance. The order with which the post processors are invoked reflect their registration order.

The registration of a `DataSourcePostProcessor` can be accomplished in two ways:

- **Direct registration**: A `DataSourcePostProcessor` instance can directly registered in a `DataSourceBuilder` instance using the `registerPostProcessor(DataSourcePostProcessor postProcessor)` method.
- **Extension services**: The default Java `ServiceLoader` extensions can be used, providing a `com.holonplatform.jdbc.DataSourcePostProcessor` file under a `META-INF/services` folder in classpath, in which to specify the fully qualified name of the `DataSourcePostProcessor` implementation. This way, the post processor is automatically registered when the builder instance is initialized.

3.2. DataSource configuration properties

The available `DataSource` configuration properties are collected and represented by the DataSourceConfigProperties interface, extending a default `ConfigPropertySet` bound to the property name prefix **holon.datasource**.

The available configuration properties are listed below:

Table 1. DataSource configuration properties

Name	Type	Meaning
holon.datasource. **type**	String	DataSource type name.
holon.datasource. **driver-class-name**	String	The JDBC Driver class name to use. If not specified, the default DataSource builder tries to auto-detect it form the connection URL.
holon.datasource. **url**	String	JDBC connection url
holon.datasource. **username**	String	JDBC connection username
holon.datasource. **password**	String	JDBC connection password
holon.datasource. **platform**	`DatabasePlatform` enumeration	Database platform to which the DataSource is connected. If not specified, the DataSource builder tries to auto-detect it form the connection URL.
holon.datasource. **auto-commit**	Boolean (`true` / `false`)	Enable/Disable auto-commit for the JDBC driver
holon.datasource. **max-pool-size**	Integer number	For connection pooling DataSource types, configure the minimum connection pool size

Name	Type	Meaning
holon.datasource. **min-pool-size**	Integer number	For connection pooling DataSource types, configure the maximum connection pool size
holon.datasource. **validation-query**	String	For connection pooling DataSource types the query to use to validate the connections in the pool
holon.datasource. **jndi-name**	String	*JNDI* lookup name for `JNDI` DataSource retrieval strategy

The `DataSourceConfigProperties` can be loaded from a number a sources using the default `ConfigPropertySet` builder interface:

```
DataSourceConfigProperties config = DataSourceConfigProperties.builder()
.withDefaultPropertySources().build(); ①

config = DataSourceConfigProperties.builder().withSystemPropertySource().build(); ②

Properties props = new Properties();
props.put("holon.datasource.url", "jdbc:h2:mem:testdb");
config = DataSourceConfigProperties.builder().withPropertySource(props).build(); ③

config = DataSourceConfigProperties.builder().withPropertySource(
"datasource.properties").build(); ④

config = DataSourceConfigProperties.builder()
    .withPropertySource(
        Thread.currentThread().getContextClassLoader().getResourceAsStream(
"datasource.properties"))
    .build(); ⑤
```

① Read the configuration properties from *default* property sources (i.e. the `holon.properties` file)

② Read the configuration properties from `System` properties

③ Read the configuration properties from a `Properties` instance

④ Read the configuration properties from the `datasource.properties` file

⑤ Read the configuration properties from the `datasource.properties InputStream`

3.3. Multiple DataSource configuration

When a multiple `DataSource` configuration is required and properties are read from the same source, a *data context id* can be used to discern one `DataSource` configuration property set form another.

From the property source point of view, the *data context id* is used as a **suffix** after the configuration property set name (`holon.datasource`) and before the specific property name.

For example, suppose we have a configuration property set for two different data sources as follows:

```
holon.datasource.one.url=jdbc:h2:mem:testdb1
holon.datasource.one.username=sa

holon.datasource.two.url=jdbc:h2:mem:testdb2
holon.datasource.two.username=sa
```

To build two DataSources, one bound to the `one` configuration property set and the other bound to the `two` configuration property set, the `DataSourceConfigProperties` can be obtained as follows, specifying the *data context id* when obtaining the builder:

```
DataSourceConfigProperties config1 = DataSourceConfigProperties.builder("one")
    .withPropertySource("datasource.properties").build();

DataSourceConfigProperties config2 = DataSourceConfigProperties.builder("two")
    .withPropertySource("datasource.properties").build();
```

3.4. BasicDataSource

A BasicDataSource is provided to directly use a simple, standard `javax.sql.DataSource` implementation.

The `BasicDataSource` uses the `java.sql.DriverManager` class and returns a new `java.sql.Connection` from every `getConnection()` call.

For this reason, can be useful for test or standalone environments outside J2EE containers.

A fluent builder is provided to create and configure a `BasicDataSource` instance:

```
DataSource dataSource = BasicDataSource.builder().url("jdbc:h2:mem:testdb").username(
"sa") //
    .driverClassName("org.h2.Driver") ①
    .build();

try (Connection connection = dataSource.getConnection()) {
  // ...
}

dataSource = BasicDataSource.builder().url("jdbc:h2:mem:testdb").username("sa") //
    .database(DatabasePlatform.H2) ②
    .build();
```

① Build a `BasicDataSource` providing driver class name

② Build a `BasicDataSource` using the `DatabasePlatform` enumeration to obtain the driver class name

4. Multi-tenant DataSource

The platform provides a DataSource with *multi-tenant* support, represented by the MultiTenantDataSource interface.

This DataSource acts as a **wrapper** for concrete `DataSource` implementations, and a new `DataSource` instance is created for each **tenant** which requires a connection. By default, `DataSource` instances are reused, so if an instance was already created for a specific tenant, this one is returned at next tenant connection request.

A `reset()` method is provided to clear the internal per-tenant `DataSource` instance cache. To clear only the cached instance for a specific **tenant id**, the `reset(String tenantId)` method is provided.

Two configured elements are required for the proper operation of a `MultiTenantDataSource`:

1. A `TenantResolver` instance, configured at DataSource build time or available in platform `Context` (See link:core.html#Multi-tenancy) to provide the current **tenant id**;
2. A TenantDataSourceProvider, which acts as a concrete `DataSource` instances provider, configured at DataSource build time or available in platform `Context`.

```
MultiTenantDataSource dataSource = MultiTenantDataSource.builder().resolver(() ->
Optional.of("test")) ①
    .provider(tenantId -> new DefaultBasicDataSource()) ②
    .build();
```

① Set the `TenantResolver`

② Set the `TenantDataSourceProvider`

5. Spring framework integration

The `holon-jdbc-spring` artifact provides integration with the Spring framework for JDBC `DataSource` building and configuration.

Maven coordinates:

```
<groupId>com.holon-platform.jdbc</groupId>
<artifactId>holon-jdbc-spring</artifactId>
<version>5.0.4</version>
```

5.1. DataSource auto-configuration

The EnableDataSource annotation can be used on Spring configuration classes to enable automatic `DataSource` configuration, using Spring `Environment` property sources to obtain the `DataSource` configuration properties, which must be defined according to the DataSourceConfigProperties property set.

See DataSource configuration properties for details.

The *data context id* to which the `DataSource` configuration is bound can be configured using the `dataContextId()` annotation attribute, useful when is required to configure multiple `DataSource` instances. The *data context id* will be used as a **suffix** after the configuration property set name (`holon.datasource`) and before the specific property name. For example, if the *data context id* is `test`, the JDBC connection URL for the `DataSource` must be configured using a property named `holon.datasource.test.url`.

When a *data context id* is defined, a Spring **qualifier** named the same as the *data context id* will be associated to the auto-generated `DataSource` bean definitions, and such qualifier can be later used to obtain the right DataSource instance through dependency injection. Each bean definition will be named using the default DataSource bean name (`dataSource`) followed by an underscore and by the *data context id* name. For example: `dataSource_test`.

For example, given a `datasource.properties` file defined as follows:

datasource.properties

```
holon.datasource.one.url=jdbc:h2:mem:testdb1
holon.datasource.one.username=sa

holon.datasource.two.url=jdbc:h2:mem:testdb1
holon.datasource.two.username=sa
```

Two `DataSource` will be configured using the two `EnableDataSource` annotations, one bound to the *data context id* `one` and another bound to the *data context id* `two`, each qualified with the *data context id* name:

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

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

@PropertySource("datasource.properties")
@Import({ Config1.class, Config2.class })
@Configuration
class OverallConfig {

}

class MyBean {

  @Autowired
  @Qualifier("one")
  private DataSource dataSource1;

  @Autowired
  @Qualifier("two")
  private DataSource dataSource2;

}
```

When more than one `DataSource` bean is configured, one of these can be marked as *primary*, meaning that will be the one provided by the Spring context when no specific name or qualifier is specified, using the `primary` boolean configuration property.

For example, using the following configuration properties file:

datasource2.properties

```
holon.datasource.one.url=jdbc:h2:mem:testdb1
holon.datasource.one.username=sa
holon.datasource.one.primary=true

holon.datasource.two.url=jdbc:h2:mem:testdb1
holon.datasource.two.username=sa
```

The first `DataSource` can be injected omitting the qualifier:

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

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

@PropertySource("datasource2.properties")
@Import({ Config1.class, Config2.class })
@Configuration
class OverallConfig {

}

class MyBean {

  // primary DataSource
  @Autowired
  private DataSource dataSource1;

  @Autowired
  @Qualifier("two")
  private DataSource dataSource2;

}
```

The `EnableDataSource` annotation provides also a `enableTransactionManager()` attribute, that, if set to `true`, automatically registers a JDBC `PlatformTransactionManager` to enable transactions management by using Spring's transaction infrastructure (for example to using `Transactional` annotations).

5.2. Additional configuration properties

The JDBC Spring integration supports a set of additional `DataSource` configuration properties, collected in the SpringDataSourceConfigProperties interface, which can be used to configure further `DataSource` initialization options.

The available additional configuration properties are listed below:

Table 2. Spring DataSource configuration properties

Name	Type	Meaning
holon.datasource. **primary**	Boolean (true/false)	Marks the DataSource bean as *primary*, meaning that will be the one provided by the Spring context when no specific name or qualifier is specified

Name	Type	Meaning
holon.datasource. **schema**	String	Specifies the the schema (DDL) script to execute when the DataSource is initialized
holon.datasource. **data**	String	Specifies the the data (DML) script to execute when the DataSource is initialized
holon.datasource. **continue-on-error**	Boolean (true/false)	Whether to stop schema/data scripts execution if an error occurs
holon.datasource. **separator**	String	Statement separator in SQL initialization scripts. Default is semicolon.
holon.datasource. **sql-script-encoding**	String	SQL scripts encoding
holon.datasource. **initialize**	Boolean (true/false)	Whether to populate the database after DataSource initialization using schema/data scripts (default is true)

Apart from the primary configuration `property`, all the other properties are related to database initialization through SQL script at `DataSource` configuration time.

If the `initialize` property is set to true (the default) and the script files `schema.sql` and `data.sql` are available from the standard locations (in the root of the classpath), the scripts are executed to initialize the database, in given order. The scripts location can be changed using the `schema` and `data` configuration properties.

In addition, the `schema-{platform}`.sql and `data-{platform}`.sql script (if present) are loaded if a database platform is specified using the `platform` configuration property and `{platform}` is the value of such property.

When a *data context id* is specified, the *data context id* name will be used as prefix for the default init scripts: `{datacontextid}-data-.sql` and `{datacontextid}-data-.sql`.

So, if you configured two DataSources using for example the following properties in the `application.properties` file:

```
holon.datasource.one.url=jdbc:h2:mem:testdb1
holon.datasource.one.username=sa
holon.datasource.one.password=

holon.datasource.two.url=jdbc:h2:mem:testdb2
holon.datasource.two.username=sa
holon.datasource.two.password=
```

You can provide different initialization scripts for each `DataSource`, i.e. for each *data context id*,

named in this example `one` and `two`. For the `one` DataSource you will provide the initialization files `one-schema.sql` and `one-data.sql`, while for the `two` DataSource you will provide the initialization files `two-schema.sql` and `two-data.sql`

6. Spring Boot integration

The `holon-jdbc-spring-boot` artifact provides integration with Spring Boot for JDBC `DataSource` 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-jdbc-spring-boot</artifactId>
<version>5.0.4</version>
```

Two auto-configuration features are provided:

1. JDBC `DataSource` auto-configuration. This auto-configuration feature is enabled when a Holon DataSource configuration property (`holon.datasource.*`) is detected in Spring context `Environment`, and provides automatic `DataSource` beans registration and configuration following the same strategy adopted by the DataSource auto-configuration annotation described above.

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

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

2. DataSource `PlatformTransactionManager` auto-configuration. This auto-configuration feature is enabled only if a `PlatformTransactionManager` bean is not already registered in Spring context and register a `DataSourceTransactionManager` bean for each `DataSource` registered using the Holon DataSource configuration properties (`holon.datasource.*`).

If a *data context id* is defined for a `DataSource`, the corresponding `PlatformTransactionManager` will be qualified with the *data context id* name, and such qualifier can be later used to obtain the right DataSource instance through dependency injection.

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

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

6.1. Spring Boot starters

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

1. Default JDBC starter provides the dependencies to the Holon JDBC Spring and Spring Boot integration artifacts, in addition to default Holon *core* 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</artifactId>
<version>5.0.4</version>
```

2. JDBC starter with HikariCP DataSource provides the same dependencies as the default JDBC 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 type is not explicitly specified using the corresponding configuration property.

Maven coordinates:

```
<groupId>com.holon-platform.jdbc</groupId>
<artifactId>holon-starter-jdbc-hikaricp</artifactId>
<version>5.0.4</version>
```

7. Loggers

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

The logger name for the **JDBC** module is `com.holonplatform.jdbc`.

8. System requirements

8.1. Java

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