Holon Platform JDBC Module - Reference manual

Table of Contents

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.

Futhermore, the module provides integration with the **Spring** framework relatively to `DataSource` configuration and `DataSource` auto-configuration facilities using **Spring Boot**.

1.1. Sources and contributions

The Holon Platform **JDBC** module source code is available from the GitHub repository https://github.com/holon-platform/holon-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. 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>5.3.0</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>5.3.0</version>
      <strong><type>pom</type></strong>
      <strong><scope>import</scope></strong>
    </dependency>
  </dependencies>
</dependencyManagement>
```

2.1. Using the Platform BOM

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

See Obtain the platform artifacts for details.

3. What's new in version 5.2.x

- The JdbcTransactionOptions API was introduced to provide JDBC transaction configuration options.
- The transaction *isolation* level can be specified using the TransactionIsolation enumeration.
- Support for JDK 9+ module system using `Automatic-Module-Name`.

4. What's new in version 5.1.x

- A `DataSource` can now be built through the `DataSourceBuilder` API directly providing the `DataSource` configuration properties, besides using a configuration property source. See Provide DataSource configuration properties programmatically.
- The JdbcConnectionHandler API was introduced to provide `DataSource` connections lifecycle customization. See the Holon Platform JDBC Datastore Module for a use case.

5. JDBC DataSource creation and configuration

Maven coordinates:

```
<groupId>com.holon-platform.jdbc</groupId>
<artifactId>holon-jdbc</artifactId>
<version>5.3.0</version>
```

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

The DataSourceBuilder API can be used to build `DataSource` instances using a set of configuration properties, represented by the `DataSourceConfigProperties` property set.

See the next sections to learn how to use the DataSource builder API to create JDBC `DataSource` instances.

5.1. DataSource configuration properties

The `DataSource` configuration properties are available from the DataSourceConfigProperties interface, which is a standard ConfigPropertySet API and allows to obtain the configuration property set values from different sources.

The `DataSourceConfigProperties` property set name prefix is **holon.datasource**.

The available configuration properties are listed below:

Name	Type	Meaning
holon.datasource. **type**	String	DataSource type: see DataSource type
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`)	The default JDBC driver *auto-commit* mode
holon.datasource. **max-pool-size**	Integer number	For connection pooling `DataSource` types, configure the minimum connection pool size
holon.datasource. **min-pool-size**	Integer number	For connection pooling `DataSource` types, configure the maximum connection pool size

Name	Type	Meaning
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` property set can be loaded from a number a sources using the default `ConfigPropertySet` builder API:

```
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`

5.2. Multiple DataSource configuration

When multiple `DataSource` instances are to be configured and the configuration properties are read from the same property 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:

datasource.properties

```
holon.datasource.<strong>one</strong>.url=jdbc:h2:mem:testdb1
holon.datasource.<strong>one</strong>.username=sa

holon.datasource.<strong>two</strong>.url=jdbc:h2:mem:testdb2
holon.datasource.<strong>two</strong>.username=sa
```

In the example above, the `one` and `two` strings represents two different *data context ids*.

To build two `DataSource` instances, one bound to the `one` configuration property set and the other bound to the `two` configuration property set, the `DataSourceConfigProperties` implementation can be obtained as follows, specifying the *data context id* when obtaining the builder, even using the same property source:

```
DataSourceConfigProperties config1 = DataSourceConfigProperties.builder("one") ①
    .withPropertySource("datasource.properties").build();

DataSourceConfigProperties config2 = DataSourceConfigProperties.builder("two") ②
    .withPropertySource("datasource.properties").build();
```

① Obtain a `DataSourceConfigProperties` builder for the `one` *data context id*

② Obtain a `DataSourceConfigProperties` builder for the `two` *data context id*

5.3. Build a DataSource using the `DataSourceBuilder` API

The DataSourceBuilder API can be used to build `DataSource` instances using the DataSource configuration properties property set.

A `DataSourceBuilder` API implementation can be otained using the `create()` static method.

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` API can be used as follows to create a `DataSource` instance:

```
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` file as property source

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

5.3.1. Provide DataSource configuration properties programmatically

The `DataSourceBuilder` API can be also used directly providing the `DataSource` configuration properties. For this purpose, an appropriate *builder* API can be obtained using the `builder()` method.

This builder also supports `DataSource` *initialization scripts*, which will be executed at `DataSource` initialization time. The `DataSource` *initialization scripts* can be directly provided as a `String` of SQL statements or specifying classpath resource name (for example a file name).

```
DataSource dataSource = DataSourceBuilder.builder() ①
    .type(DataSourceType.HIKARICP) // type
    .url("jdbc:h2:mem:testdb") // jdbc url
    .username("sa") // jdbc username
    .minPoolSize(5) // max pool size
    .withInitScriptResource("init.sql") // init script resource
    .build();
```

① Obtain the `DataSource` builder

5.4. DataSource type

When using the `DataSourceBuilder` API, the concrete `DataSource` implementation to use can be selected using the `type` configuration property.

The following **type names** 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. See BasicDataSource.
- **`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.

To use a specific `DataSource` implementation, the corresponding classes must be available in classpath. So you have to ensure the required dependencies are declared for your project.

For example, to use a HikariCP pooling `DataSource` implementation, the `com.zaxxer.hikari.HikariDataSource` type can be specified:

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

<strong>holon.datasource.type</strong>=com.zaxxer.hikari.HikariDataSource
```

5.4.1. Default DataSource type selection strategy

If the `type` configuration property is not specified, the default `DataSource` type selection strategy 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.

5.4.2. DataSourceFactory

The default `DataSourceBuilder` API implementation delegates `DataSource` instances creation to a set of concrete DataSourceFactory implementations, each of them bound to a single `DataSource` **type** name.

A `DataSourceFactory` can be registered in the `DataSourceBuilder` API and 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()` method.

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

- **Direct registration**: A `DataSourceFactory` instance can directly registered in a `DataSourceBuilder` API using the `registerFactory` method. Any previous binding with given type will be replaced by the given factory.

```
class MyDataSourceFactory implements DataSourceFactory { ①

  @Override
  public String getDataSourceType() {
    return "my.type.name";
  }

  @Override
  public DataSource build(DataSourceConfigProperties configurationProperties) throws
ConfigurationException {
    // Build and return a DataSource istance using given configuration properties
    return buildTheDataSourceInstance();
  }

}

void usingTheFactory() {
  DataSourceBuilder builder = DataSourceBuilder.create();
  builder.registerFactory(new MyDataSourceFactory()); ②
}
```

① Create a `DataSourceFactory` implementation

② Register the factory in the `DataSourceBuilder` API instance

Using the direct `DataSourceFactory` registration, the registered factories will be available **only** for the specific `DataSourceBuilder` API instance.

- **Java ServiceLoader extensions**: The default Java `ServiceLoader` extensions can be used, providing a file named `com.holonplatform.jdbc.DataSourceFactory` under the `META-INF/services` folder, in which to specify the fully qualified name of the `DataSourceFactory` implementation/s to register. This way, the factory will be automatically registered at `DataSourceBuilder` API initialization time.

Using the `ServiceLoader` extensions method, the `DataSourceFactory` implementations will be available for **any** `DataSourceBuilder` API instance.

5.4.3. DataSourcePostProcessor

The DataSourcePostProcessor interface can be used to perform additional `DataSource` initialization and configuration when using the `DataSourceBuilder` API.

The `postProcessDataSource(…)` method is called just after the creation of any `DataSource` instance, providing the `DataSource` instance itself, the *type* name and the `DataSourceConfigProperties` used to create the `DataSource` instance.

In order to activate a `DataSourcePostProcessor`, it must be registered in the `DataSourceBuilder` API instance used to create the `DataSource`.

When more than one `DataSourcePostProcessor` is registered, the invocation order will be the same as the 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` method.

```
DataSourceBuilder builder = DataSourceBuilder.create();
builder.registerPostProcessor(new DataSourcePostProcessor() { ①

  @Override
  public void postProcessDataSource(DataSource dataSource, String typeName,
      DataSourceConfigProperties configurationProperties) throws
ConfigurationException {
    // perform DataSource post processing
  }
});
```

① Register a post processor in the `DataSourceBuilder` API instance

Using the direct `DataSourcePostProcessor` registration, the registered post processors will be available **only** for the specific `DataSourceBuilder` API instance.

- **Java ServiceLoader extensions**: The default Java `ServiceLoader` extensions can be used, providing a file named `com.holonplatform.jdbc.DataSourcePostProcessor` under the `META-INF/services` folder, in which to specify the fully qualified name of the `DataSourcePostProcessor` implementation/s to register. This way, the post processors will be automatically registered at `DataSourceBuilder` API initialization time.

Using the `ServiceLoader` extensions method, the `DataSourcePostProcessor` implementations will be registered for **any** `DataSourceBuilder` API instance.

5.5. BasicDataSource

The BasicDataSource API is made available to create simple, standard `javax.sql.DataSource` implementations.

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

This implementation is not designed for production and should be used only for testing purposes.

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

6. Multi-tenant DataSource

The Holon platform JDBC module provides a `DataSource` implementation with *multi-tenant* support, represented by the MultiTenantDataSource interface.

This `DataSource` implementation acts as a **wrapper** for concrete `DataSource` implementations, one for each *tenant* id. By default, `DataSource` instances are reused, so if an instance was already created for a specific tenant id, 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.

The `MultiTenantDataSource` implementation relies on the APIs to work properly:

1. A `com.holonplatform.core.tenancy.TenantResolver` API instance to obtain the **current tenant id**.
2. A TenantDataSourceProvider API implementation, which acts as concrete `DataSource` instances provider, used to obtain a `DataSource` for each **tenant id**.

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

① Set the `TenantResolver`

② Set the `TenantDataSourceProvider`

6.1. `TenantResolver` and `TenantDataSourceProvider` lookup strategy

If not directly configured, the `TenantResolver` and `TenantDataSourceProvider` implementation can be obtained by default using the Holon Platform Context resources architecture.

The `MultiTenantDataSource` will try to obtain a `TenantResolver` and a `TenantDataSourceProvider` implementation as context resources using the default *context keys* (i.e. the class names) when they are not directly configured using the appropriate builder methods.

7. Spring framework integration

Maven coordinates:

```
<groupId>com.holon-platform.jdbc</groupId>
<artifactId>holon-jdbc-spring</artifactId>
<version>5.3.0</version>
```

The `holon-jdbc-spring` artifact provides integration with the Spring framework for JDBC `DataSource` building and configuration, fully supporting multiple `DataSource` instances configuration and providing **Spring Boot** auto-configuration facilities.

7.1. 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 DataSource configuration properties property set.

For example, using the following configuration properties:

datasource.properties

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

A `DataSource` bean can be automatically configured using the `@EnableDataSource` annotation on a Spring configuration class:

```
@Configuration
@PropertySource("datasource.properties")
@EnableDataSource ①
class Config {

}

@Autowired
private DataSource dataSource1; ②
```

① Use the `@EnableDataSource` to create a `DataSource` bean instance, using the `datasource.properties` file as property source

② Obtain the configured `DataSource` bean instance

7.1.1. Multiple `DataSource` configuration

When **multiple `DataSource` instances** has to be configured, multiple `@EnableDataSource` annotations can be used, relying on the *data context id* specification to discern a configuration property set from another.

The *data context id* to which the `DataSource` configuration is bound can be configured using the `dataContextId()` attribute of the `@EnableDataSource` annotation.

As described in the Multiple DataSource configuration section, 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 a `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 generated `DataSource` bean definitions, and such qualifier can be later used to obtain the right DataSource instance through dependency injection.

Furthermore, 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.<strong>one</strong>.url=jdbc:h2:mem:testdb1
holon.datasource.<strong>one</strong>.username=sa

holon.datasource.<strong>two</strong>.url=jdbc:h2:mem:testdb1
holon.datasource.<strong>two</strong>.username=sa
```

To configure the the `DataSource` bean instances, one bound to the *data context id* `one` and another

bound to the *data context id* `two`, two `@EnableDataSource` annotations can be used this way:

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

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

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

}

@Autowired
@Qualifier("one") ③
private DataSource dataSource1;

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

① Enable a `DataSource` bean using the `one` *data context id*

② Enable a `DataSource` bean using the `two` *data context id*

③ The *data context id* can be used as *qualifier* to obtain the proper `DataSource` instance

7.1.2. Primary mode

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

If the *primary mode* is set to `PrimaryMode.TRUE`, the `DataSource` 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.

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

  @Configuration
  @EnableDataSource(dataContextId = "one", primary = PrimaryMode.TRUE) ①
  static class Config1 {
  }

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

}

@Autowired
private DataSource dataSource1; ②

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

① The `PrimaryMode.TRUE` is configured for the `one` *data context id* `@EnableDataSource` configuration

② The `DataSource` bean bound to the `one` *data context id* can be now obtained without specifyng a qualifier

7.1.3. Transaction management

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 transactions infrastructure (for example in order to use `@Transactional` annotations).

The registered transaction manager will be a standard Spring `DataSourceTransactionManager`.

```
@Configuration
@PropertySource("datasource.properties")
@EnableDataSource(enableTransactionManager = true) ①
class Config {

}

@Transactional
void doSomethingTransactionally() { ②
  // ...
}
```

① Enable a the transaction manager using the configured `DataSource`

② When a `PlatformTransactionManager` is available, the `@Transactional` annotation can be used

7.1.4. Additional `DataSource` 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 when the `@EnableDataSource` annotation is used.

The available additional configuration properties are listed below:

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
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)

If the `initialize` property is set to `true` (the default) and the script files `schema.sql` and/or `data.sql` are available from the standard locations (in the root of the classpath), such scripts are executed to initialize the `DataSource`, in given order.

The scripts locations can be changed using the `schema` and `data` configuration properties.

Additionaly, if the `platform` configuration property is provided, the `schema-{platform}`.sql and `data-{platform}`.sql scripts are executed if available, where `{platform}` is the value of the `platform` configuration property.

7.1.5. Using the *data context id* for `DataSource` initialization

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

If one or more script with a matching name pattern is available, it will be executed using the `DataSource` bean instance which corresponds to the *data context id*.

For example, given the following configuration properties to configure two `DataSource` bean instances, one bound to the *data context id* `one` and the other bound to the *data context id* `two`:

```
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
```

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

- For the `one` *data context id* `DataSource` you will provide the initialization files `one-schema.sql` and `one-data.sql`;
- For the `two` *data context id* `DataSource` you will provide the initialization files `two-schema.sql` and `two-data.sql`.

8. Spring Boot integration

Maven coordinates:

```
<groupId>com.holon-platform.jdbc</groupId>
<artifactId>holon-jdbc-spring-boot</artifactId>
<version>5.3.0</version>
```

The `holon-jdbc-spring-boot` artifact provides integration with Spring Boot to enable JDBC `DataSource` auto-configuration facilities.

8.1. JDBC `DataSource` auto-configuration

This auto-configuration feature is enabled when one of the Holon `DataSource` configuration properties (`holon.datasource.*`) is detected in the Spring `Environment`. See the DataSource configuration properties section for information about the available configuration properties.

It provides automatic `DataSource` beans registration and configuration following the same strategy adopted by the DataSource auto-configuration annotation described above.

For example, using the given `yaml` configuration properties:

```
holon:
  datasource:
    one:
      url: "jdbc:h2:mem:testdb1"
      username: "sa"
    two:
      url: "jdbc:h2:mem:testdb2"
      username: "sa"
```

The auto-configuration feature will configure two `DataSource` bean instances:

- 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.

So the `DataSource` bean instances can be obtained using dependency injection this way:

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

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

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

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

8.2. DataSource `PlatformTransactionManager` auto-configuration

This auto-configuration feature is enabled only if a `PlatformTransactionManager` bean is not already registered in the Spring context.

It registers a `DataSourceTransactionManager` bean for each `DataSource` registered using the Holon DataSource configuration properties (`holon.datasource.*`), as described in the section above.

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

Furthermore, the `PlatformTransactionManager` bean name when *data context id* is specified will be

assigned using the pattern: `transactionManager_{dataContextId}`. So, for example, the `PlatformTransactionManager` bean created for the `test` *data context id* will be named `transactionManager_test`.

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

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

8.3. Spring Boot starters

The following *starter* artifacts are available to provide a quick project configuration setup using the 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 the *core* Spring Boot starter (`spring-boot-starter`):

Maven coordinates:

```
<groupId>com.holon-platform.jdbc</groupId>
<artifactId>holon-starter-jdbc</artifactId>
<version>5.3.0</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 `DataSource` **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>5.3.0</version>
```

See the Spring Boot starters documentation for details about the Spring Boot *starters* topic and the core Spring Boot starter features.

9. 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`.

10. System requirements

10.1. Java

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