



# Holon

PLATFORM

## Holon JPA Datastore

Version 5.0.1

# 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. JpaTarget .....	4
3.4. JPA Write options .....	4
3.5. Relational expressions .....	5
3.6. Auto-generated ids .....	6
3.7. Extensions .....	6
3.8. Expression resolvers .....	6
3.9. Commodity factories .....	7
4. Spring framework integration .....	7
4.1. Datastore setup .....	7
4.2. Datastore auto-configuration .....	8
4.2.1. Commodity factories .....	9
4.3. Full JPA auto-configuration .....	9
4.3.1. EnableJpa: DataSource configuration .....	10
4.3.2. EnableJpa: EntityManagerFactory configuration .....	10
4.3.3. EnableJpa: Persistence provider (ORM) configuration .....	11
4.3.4. EnableJpa: Transaction manager configuration .....	11
4.3.5. EnableJpa: JPA Datastore configuration .....	11
4.3.6. Using <i>data context id</i> for multiple data sources .....	11
5. Spring Boot integration .....	14
5.1. JPA Datastore auto-configuration .....	14
5.2. Full JPA stack auto-configuration .....	15
5.2.1. JPA entities scan .....	15
5.3. Spring Boot starters .....	15
6. Loggers .....	16
7. System requirements .....	16
7.1. Java .....	16
7.2. JPA .....	16
7.3. Persistence providers .....	16

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

# 1. Introduction

The Holon **JPA Datastore** is the *Java Persistence API* reference implementation of the [Datastore](#) abstraction.



See the [Datastore](#) documentation for further informations about Datastores.

This **Datastore** uses the JPA [EntityManagerFactory](#) interface as a factory to obtain [EntityManager](#) instances, used by the **Datastore** to perform persistence operations and queries through the [JPQL](#) language.



The Holon Platform JPA modules and components require **JPA 2.0 or higher**, so a JPA 2.0/2.1 compliant persistence provider is needed at runtime.



To use a JPA Datastore, an [EntityManagerFactory](#) instance must be configured and available. [EntityManagerFactory](#) setup (including any ORM platform configuration concerns) is beyond the scope of this documentation.

## 2. Obtaining the artifacts

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

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

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

*Maven coordinates:*

```
<groupId>com.holon-platform.jpa</groupId>
<artifactId>holon-datastore-jpa-bom</artifactId>
<version>5.0.1</version>
```

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

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.holon-platform.jpa</groupId>
      <artifactId>holon-datastore-jpa-bom</artifactId>
      <version>5.0.1</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-jpa` artifact is the main entry point to use the JPA Datastore.

*Maven coordinates:*

```
<groupId>com.holon-platform.jpa</groupId>
<artifactId>holon-datastore-jpa</artifactId>
<version>5.0.1</version>
```

The `JpaDatastore` interface represents the **JPA Datastore**, which extends the core **Datastore** interface.

The `withEntityManager` method can be used to execute a JPA operation using a Datastore managed `EntityManager`, providing the operation execution code through the `EntityManagerOperation` functional interface. Any `EntityManager` initialization and finalization operation is executed by the JPA Datastore before and after the execution of the provided operation.

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

Test result = datastore.withEntityManager(em -> {
    return em.find(Test.class, 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 **JpaDatastore** interface. This way, the concrete Datastore implementation may be replaced by a different one at any time, without any change to the rest of the code.

## 3.1. Setup

The JPA Datastore supports the default Datastore configuration property:

- `holon.datastore.trace`: To enable/disable Datastore operations tracing

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

```
EntityManagerFactory entityManagerFactory = getEntityManagerFactory(); // get the
EntityManagerFactory to use
```

```
JpaDatastore datastore = JpaDatastore.builder() // obtain the builder
    .entityManagerFactory(entityManagerFactory) ①
    .autoFlush(true) ②
    .traceEnabled(true) ③
    .build();
```

```
datastore = JpaDatastore.builder() // obtain the builder
    .entityManagerFactory(entityManagerFactory) ④
    .entityManagerInitializer(emf -> emf.createEntityManager()) ⑤
    .entityManagerFinalizer(em -> em.close()) ⑥
    .build();
```

```
datastore = JpaDatastore.builder() // obtain the builder
    .entityManagerFactory(entityManagerFactory) ⑦
    .dataContextId("test") ⑧
    .configuration(
        DatastoreConfigProperties.builder("test").withPropertySource(
            "datastore.properties").build()) ⑨
    .build();
```

- ① Set the `EntityManagerFactory` to use
- ② Set the `autoFlush` mode to `true`, i.e. `EntityManager.flush()` is invoked after any Datastore data manipulation operation
- ③ Enable the `trace` mode to log the `JPQL` queries and operation performed by the Datastore
- ④ Set the `EntityManagerFactory` to use
- ⑤ Provide a custom `EntityManagerInitializer` to control how the `EntityManager` instances are obtained from the `EntityManagerFactory`
- ⑥ Provide a custom `EntityManagerFinalizer` to control how the `EntityManager` instances are finalized after Datastore operations executions

- ⑦ Set the `EntityManagerFactory` to use
- ⑧ Build a Datastore bound to a specific *data context id* named `test`
- ⑨ Set a Datastore configuration property source using the `datastore.properties` file

## 3.2. Data targets and paths

The JPA `Datastore` relies on the following conventions regarding *\*DataTarget\*s* and *\*Path\*s*:

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



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. JpaTarget

The `JpaTarget` interface can be used to define a **DataTarget** using a JPA *entity* class rather than an entity name. The entity name used in persistence operations will be the provided entity class simple name or the name specified using the `name` attribute of the `javax.persistence.Entity` annotation, if available.

```
JpaTarget<Test> TARGET = JpaTarget.of(Test.class); ①
```

- ① Create a **DataTarget** bound to given `Test` JPA entity class using the `JpaTarget` interface

## 3.4. JPA Write options

The additional `WriteOption` `JpaWriteOption.FLUSH` is supported by the JPA Datastore, and can be used to synchronize the persistence context to the underlying database after a data manipulation operation, automatically calling the `EntityManager.flush()` method.

```

@Entity
class Test {

    @Id
    private Integer id;
    private String name;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}

public void insertAndFlush() {

    final PathProperty<Integer> ID = PathProperty.create("id", Integer.class);
    final PathProperty<String> NAME = PathProperty.create("name", String.class);

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

    OperationResult result = getDatastore().insert(TARGET,
        PropertyBox.builder(ID, NAME).set(NAME, "Test name").build(), JpaWriteOption
        .FLUSH);

}

```

## 3.5. Relational expressions

As a *relational* Datastore, the JPA 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.

## 3.6. Auto-generated ids

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

The auto-generated id values can be obtained from the [OperationResult](#) object, returned by Datastore data manipulation operations, through the [getInsertedKeys\(\)](#) 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<Integer> ID = PathProperty.create("id", Integer.class); ①
final PathProperty<String> NAME = PathProperty.create("name", String.class);

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

PropertyBox value = PropertyBox.builder(ID, NAME).set(NAME, "Test name").build(); ②

datastore.insert(JpaTarget.of(Test.class), value, JpaWriteOption.FLUSH,
    DefaultWriteOption.BRING_BACK_GENERATED_IDS); ③

Integer idValue = value.getValue(ID); ④
```

- ① The **id** column is supposed to be auto-generated by the ORM/database
- ② Create the [PropertyBox](#) to insert, not providing the **id** value
- ③ Execute the insert operation using the **BRING\_BACK\_GENERATED\_IDS** write option and the **FLUSH** JPA write option to ensure the id auto-generation is triggered
- ④ The **ID** property of the inserted [PropertyBox](#) is updated with the auto-generated value

## 3.7. Extensions

## 3.8. Expression resolvers

The JPA Datastore supports [ExpressionResolver](#) automatic registration using the [JpaDatastoreExpressionResolver](#) base type and default *Java service extensions*.

To automatically register an [ExpressionResolver](#) this way, a class implementing [JpaDatastoreExpressionResolver](#) has to be created and its qualified full name must be specified in a file named `com.holonplatform.datastore.jpa.config.JpaDatastoreExpressionResolver` placed in a

`META-INF/services` folder in classpath.

## 3.9. Commodity factories

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

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

The `JpaDatastoreCommodityContext` interface represents the JPA Datastore specific commodity context and it is provided at commodity creation time to the factories.

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

- The `EntityManagerFactory` bound the JPA Datastore;
- A `getEntityManager()` method to obtain a `EntityManager` instance;
- The `ORMPlatform` used by the JPA Datastore, if available;
- Whether the Datastore *auto-flush* mode is enabled.
- Whether the Datastore *trace* mode is enabled.

## 4. Spring framework integration

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

*Maven coordinates:*

```
<groupId>com.holon-platform.jpa</groupId>
<artifactId>holon-datastore-jpa-spring</artifactId>
<version>5.0.1</version>
```

### 4.1. Datastore setup

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

This interface creates and represents a JPA Datastore implementation which supports Spring JPA and JTA transaction management architecture, using a Spring *transactional EntityManager proxy* as Datastore `EntityManager`.

## 4.2. Datastore auto-configuration

The `EnableJpaDatastore` annotation can be used on Spring configuration classes to enable automatic JPA Datastore configuration. An available `EntityManagerFactory` type bean must be present in context to enable the JPA Datastore.

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

When a *data context id* is not specified, the JPA Datastore is bound to the unique `EntityManagerFactory` type bean registered in context. If the bean is not unique or is not present, a configuration error is thrown. The `entityManagerFactoryReference` annotation attribute can be used to specify the explicit `EntityManagerFactory` bean name to use for the JPA Datastore.

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

The *auto-flush* mode can be specified using the `autoFlush` annotation attribute.

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 JPA Datastore data manipulation methods, to automatically create or participate in a Spring transaction when these methods are invoked.

```

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

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

    @Bean
    public PlatformTransactionManager transactionManager(EntityManagerFactory emf) {
        return new JpaTransactionManager(emf);
    }

}

class MyBean {

    @Autowired
    private Datastore datastore;

}

```

#### 4.2.1. Commodity factories

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

See [Commodity factories](#) for additional details about the `JpaDatastoreCommodityFactory` type.

### 4.3. Full JPA auto-configuration

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

- **DataSource:** If a `javax.sql.DataSource` type bean is not already registered in Spring context, a `DataSource` instance is created and configured using standard JDBC `DataSourceConfigProperties`;
- **EntityManagerFactory:** A `javax.persistence.EntityManagerFactory` bean is registered and configured, along with a suitable Spring `JpaVendorAdapter` instance for ORM configuration;
- **PlatformTransactionManager:** A `org.springframework.transaction.PlatformTransactionManager`

bean is registered to be used within the Spring transaction infrastructure;

- **Datastore:** A `JpaDatastore` is configured and bound to the `EntityManagerFactory` of the JPA stack.

```
@PropertySource("jpa.properties")
@EnableJpa(entityPackageClasses = Test.class)
@Configuration
class JpaConfig {

}
```

### 4.3.1. `EnableJpa`: `DataSource` configuration

A `DataSource` type bean is automatically configured if:

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

In order to auto-configure the `DataSource` bean, a suitable set of configuration properties must be available in Spring environment (typically using one or more `PropertySource`).

See [DataSource configuration properties](#) for details.

### 4.3.2. `EnableJpa`: `EntityManagerFactory` configuration

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

If a `persistence.xml` configuration file is not available, automatic persistence unit *entity* classes configuration can be performed using either the annotation attribute:

- `entityPackages` : providing a list of package names to scan in order to map JPA *entity* classes into the `EntityManagerFactory`,

or the annotation attribute:

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

Two more annotation attributes are available for persistence unit configuration:

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

See the JPA documentation for further details about persistence unit configuration.

### 4.3.3. EnableJpa: Persistence provider (ORM) configuration

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

See `ORMPlatform` enumeration for a list of supported persistence provider implementations.

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

### 4.3.4. EnableJpa: Transaction manager configuration

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

A set of attributes are made available by the `@EnableJpa` annotation to fine tune the transaction manager configuration: `transactionSynchronization`, `defaultTimeout`, `validateExistingTransaction`, `failEarlyOnGlobalRollbackOnly`, `rollbackOnCommitFailure`.

See [EnableJpa](#) Javadocs for informations about each of these configuration attributes.

### 4.3.5. EnableJpa: JPA Datastore configuration

By default, using the `EnableJpa` annotation a JPA Datastore is automatically created and configured using the `EntityManagerFactory` of the JPA beans stack. The JPA Datastore instance will be a [Datastore setup](#), enabling all the Spring-related auto-configuration features described above.

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

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

- `autoFlush` : to enable the JPA Datastore *auto flush* mode. When auto-flush mode is enabled, the `EntityManager.flush()` method is invoked after each Datastore data manipulation operation to ensure the underlying database synchronization.
- `transactionalDatastore` : to add *transactional* behaviour to suitable Datastore methods, i.e. to automatically create or participate in a transaction when methods are invoked.

### 4.3.6. Using data context id for multiple data sources

When a *data context id* is specified through the `dataContextId` annotation attribute, all the JPA stack beans auto-configured with the `EnableJpa` annotation are bound to the specified *data context id* name, allowing the support for multiple data sources and persistence units.

When a *data context id* is specified:

- Any configuration property (for `DataSource` or JPA beans configuration) must be provided using the *data context id* as prefix. For example, if the *data context id* is named `test`, the JDBC

DataSource url must be specified the following way: `holon.datasource.test.url=...` and so on;

- The *data context id* will be used as persistence unit name;
- Each of the auto-configured JPA stack bean will be *qualified* using the *data context id* name, to allow dependency injection using a Spring `Qualifier`.

In case of multiple data context ids, the `primary()` attribute of the `EnableJpa` annotation can be used to mark one the JPA beans stack as primary candidate for dependency injection when a qualifier is not specified.

```

@Configuration
@PropertySource("jpa.properties")
@EnableTransactionManagement
class MultiJpaConfig {

    @Configuration
    @EnableJpa(dataContextId = "one", entityPackageClasses = Test1.class)
    class ConfigOne {
    }

    @Configuration
    @EnableJpa(dataContextId = "two", entityPackageClasses = Test2.class)
    class ConfigTwo {
    }

}

@Autowired
@Qualifier("one")
private DataSource dataSource1;
@Autowired
@Qualifier("one")
private EntityManagerFactory entityManagerFactory1;
@Autowired
@Qualifier("one")
private PlatformTransactionManager transactionManager1;
@Autowired
@Qualifier("one")
private JpaDatastore datastore1;

@Autowired
@Qualifier("two")
private DataSource dataSource2;
@Autowired
@Qualifier("two")
private EntityManagerFactory entityManagerFactory2;
@Autowired
@Qualifier("two")
private PlatformTransactionManager transactionManager2;
@Autowired
@Qualifier("two")
private JpaDatastore datastore2;

```

[[JPA configuration properties]] == JPA configuration properties

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

The available configuration properties are listed below:

Table 1. JPA configuration properties

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

The [JpaConfigProperties](#) can be loaded from a number of sources using the default [ConfigPropertySet](#) builder interface.

Using the Spring integration, all [Environment](#) registered [PropertySources](#) will be enabled as a [JpaConfigProperties](#) source.

## 5. Spring Boot integration

The [holon-datastore-jpa-spring-boot](#) artifact provides integration with [Spring Boot](#) for JPA stack and 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.jpa</groupId>
<artifactId>holon-datastore-jpa-spring-boot</artifactId>
<version>5.0.1</version>
```

### 5.1. JPA Datastore auto-configuration

The JPA datastore is auto-configured only when:

- A [JpaDatastore](#) type bean is not already registered in Spring context
- A valid [EntityManagerFactory](#) type bean is available in Spring context

The JPA Datastore auto-configuration behaviour is the same of the one adopted by the [EnableJpaDatastore](#) annotation. See [Datastore auto-configuration](#) for details.

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

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

## 5.2. Full JPA stack auto-configuration

A full JPA beans stack is auto-configured only when:

- An `EntityManagerFactory` type bean is not already registered in Spring context.

The JPA stack auto-configuration behaviour is the same of the one adopted by the `EnableJpa` annotation. See [Full JPA auto-configuration](#) for details.

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

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

### 5.2.1. JPA entities scan

When the full JPA stack auto-configuration is enabled, the `JpaEntityScan` repeatable annotation can be used on Spring configuration classes to specify the base packages to scan for JPA entity classes.

## 5.3. Spring Boot starters

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

**1. JPA Datastore starter using Hibernate** provides the dependencies to the Holon JPA Datastore 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`). Furthermore, this starter provides the **Hibernate ORM** and the **HikariCP** DataSource dependencies.

*Maven coordinates:*

```
<groupId>com.holon-platform.jpa</groupId>  
<artifactId>holon-starter-jpa-hibernate</artifactId>  
<version>5.0.1</version>
```

**2. JPA Datastore starter using EclipseLink** provides the dependencies to the Holon JPA Datastore 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`). Furthermore, this starter provides the **EclipseLink ORM** and the **HikariCP** DataSource dependencies.

*Maven coordinates:*

```
<groupId>com.holon-platform.jpa</groupId>  
<artifactId>holon-starter-jpa-eclipselink</artifactId>  
<version>5.0.1</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 **JPA Datastore** module is [com.holonplatform.datastore.jpa](#).

## 7. System requirements

### 7.1. Java

The Holon Platform JPA Datastore module requires [Java 8](#) or higher.

### 7.2. JPA

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

### 7.3. Persistence providers

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

- [Hibernate ORM](#) version **4.x** or **5.x**
- [EclipseLink](#) version **2.5 or higher**