Holon Platform MongoDB Datastore Module - Reference manual

Table of Contents

1. Introduction . . . 2
   1.1. Sources and contributions . . . 3
2. Obtaining the artifacts . . . 3
   2.1. Using the Platform BOM . . . 4
3. MongoDB Datastore . . . 4
   3.1. MongoDB Datastore configuration . . . 4
      3.1.1. Database name . . . 4
      3.1.2. Common MongoDB Datastore configuration options . . . 5
      3.1.3. `Enum` codec strategy . . . 5
      3.1.4. Common Datastore API configuration options . . . 6
   3.2. Property model and MongoDB documents mapping . . . 7
      3.2.1. Nested documents . . . 7
      3.2.2. Multi value document nodes . . . 8
   3.3. Document id . . . 9
   3.4. Database collections . . . 10
   3.5. Auto-generated ids . . . 10
   3.6. Synchronous MongoDB Datastore . . . 11
   3.7. Asynchronous MongoDB Datastore . . . 14
   3.8. Reactive MongoDB Datastore . . . 17
   3.9. Custom BSON filters and sorts . . . 20
      3.9.1. BsonFilter . . . 20
      3.9.2. BsonSort . . . 20
   3.10. Transactions management . . . 21
      3.10.1. Synchronous transactions management . . . 21
      3.10.2. Asynchronous transactions management . . . 21
      3.10.3. Reactive transactions management . . . 22
   3.11. Using the `MongoDatabaseHandler` for a direct database access . . . 23
   3.12. Extending the MongoDB Datastore API . . . 23
   3.13. Expression resolvers . . . 23
      3.13.1. MongoDB Expression resolvers registration . . . 23
      3.13.2. Specific expression resolvers registration . . . 24
      3.13.3. Expression resolvers priority . . . 24
      3.13.4. Expression validation . . . 25
      3.13.5. MongoDB Datastore expressions . . . 25

3.13.6. MongoDB Expression resolution context . . . 27
3.14. Commodity factories . . . 27
4. Spring ecosystem integration . . . 29
4.1. Enable a MongoDB Datastore bean . . . 29
4.1.1. Synchronous MongoDB Datastore . . . 29
4.1.2. Asynchronous MongoDB Datastore . . . 30
4.1.3. Reactive MongoDB Datastore . . . 31
4.2. MongoDB Datastore bean configuration . . . 31
4.2.1. MongoClient bean name . . . 31
4.2.2. Enum codec strategy . . . 32
4.2.3. Mongo database configuration . . . 33
4.2.4. Primary mode . . . 33
4.2.5. MongoDB Datastore configuration properties . . . 34
4.2.6. Datastore extension and configuration using the Spring context . . . 35
4.3. Multiple MongoDB Datastores configuration . . . 37
5. Spring Boot integration . . . 38
5.1. Synchronous MongoDB Datastore . . . 39
5.2. Asynchronous MongoDB Datastore . . . 39
5.3. Reactive MongoDB Datastore . . . 39
5.4. MongoDB Datastore configuration . . . 39
5.5. Disabling the MongoDB Datastore auto-configuration feature . . . 40
5.6. Spring Boot starters . . . 40
6. Loggers . . . 41
7. System requirements . . . 41
7.1. Java . . . 41
7.2. MongoDB Java Drivers . . . 41

Copyright © 2016-2019

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 **MongoDB Datastore** is the MongoDB reference implementation of the Holon Platform Datastore API.

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

The MongoDB `Datastore` implementation uses the MongoDB Java Driver to perform data access and management operations on MongoDB databases.

1.1. Sources and contributions

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

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.mongo</groupId>
<artifactId>holon-datastore-mongo-bom</artifactId>
<version>5.5.0</version>
```

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

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.holon-platform.mongo</groupId>
      <artifactId>holon-datastore-mongo-bom</artifactId>
      <version>5.5.0</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. MongoDB Datastore

The MongoDB Datastore API is available in three implementations:

- A **synchronous** implementation, which is the reference implementation of the Holon Platform Datastore API: See Synchronous MongoDB Datastore.
- An **asynchronous** implementation, which is the reference implementation of the Holon Platform AsyncDatastore API: See Asynchronous MongoDB Datastore.
- An **reactive** implementation, which is the reference implementation of the Holon Platform ReactiveDatastore API: See Reactive MongoDB Datastore.

3.1. MongoDB Datastore configuration

Regardless of the actual implementation, each MongoDB Datastore provides a set of configuration options, described below.

3.1.1. Database name

A MongoDB Datastore is **always bound to a single MongoDB database**, so the the database name configuration is required to build a MongoDB Datastore instance.

Each builder provides a `database(String database)` method to set the **database name** to which the MongoDB Datastore is bound.

Example:

```
MongoDatastore datastore = MongoDatastore.builder() ①
    .client(getMongoClient()) ②
    .database("my_db") ③
    .build();
```

① Obtain a builder to configure and create a new MongoDB Datastore (in this example, a synchronous `MongoDatastore` implementation)

② Set the MongoDB client to use

③ Set the database name to use

3.1.2. Common MongoDB Datastore configuration options

A set of common configuration options are available from the MongoDB Datastore builder, regardless of the actual implementation used.

These common configuration options allows to:

- Configure the default **read preference** to use.
- Configure the default **read concern** to use.
- Configure the default **write concern** to use.
- Add a new bson `Codec`.
- Add a new bson `CodecProvider`.

```
MongoDatastore datastore = MongoDatastore.builder().client(getMongoClient()).database
("my_db")
    .readPreference(ReadPreference.primary()) ①
    .readConcern(ReadConcern.MAJORITY) ②
    .writeConcern(WriteConcern.UNACKNOWLEDGED) ③
    .withCodec(getCustomCodec()) ④
    .withCodecProvider(getCustomCodecProvider()) ⑤
    .build();
```

① Set the default read preference

② Set the default read concern

③ Set the default write concern

④ Add a new bson `Codec`

⑤ Add a new bson `CodecProvider`

3.1.3. `Enum` codec strategy

By default, **enumeration** type document field values are mapped to a Java `Enum` class using a **name based** matching strategy.

An **ordinal based** matching strategy is also available, using the `Enum` ordinal position to map an enumeration type document field value to a Java `Enum` class.

The enumeration codec strategies are listed in the EnumCodecStrategy enumeration and the strategy to use can be configured using the MongoDB Datastore builder `enumCodecStrategy` method.

```
MongoDatastore datastore = MongoDatastore.builder().client(getMongoClient()).database
("my_db")
    .enumCodecStrategy(EnumCodecStrategy.ORDINAL) ①
    .build();
```

① Set the enumeration codec strategy to `ORDINAL`

3.1.4. Common Datastore API configuration options

Every MongoDB Datastore builder, regardless of the actual implementation, extends the core Datastore builder API, which provides the common Datastore configuration settings listed below.

Builder method	Arguments	Description
`dataContextId`	The *data context id* `String` value	Set the *data context id* to which the Datastore is bound. Can be used, for example, to declare configuration properties for multiple Datastore instances.
`traceEnabled`	`true` or `false`	Whether to enable Datastore operations *tracing*. When enabled, the MongoDB Datastore will log the JSON representation of the documents involved in the performed datastore operations.
`configuration`	A `DatastoreConfigProperties` instance	Set the `DatastoreConfigProperties` type configuration property set instance to use in order to read the Datastore configuration properties. See the Datastore configuration documentation section for details. This configuration properties can be used as an alternative for the programmatic configuration performed with the previous builder methods.

Example:

```
MongoDatastore datastore = MongoDatastore.builder().client(getMongoClient()).database
("my_db")
    .dataContextId("mydataContextId") ①
    .traceEnabled(true) ②
    .build();
```

① Set a *data context id* for the Datastore

② Activate operations *tracing* in log

The configuration properties can also be provided through an external configuration property source, using the properties provided by the DatastoreConfigProperties property set.

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

```
holon.datastore.trace=true
```

We can use it as configuration property source to enable the Datastore *tracing* mode:

```
MongoDatastore datastore = MongoDatastore.builder().client(getMongoClient()).database
("my_db")
    .configuration(DatastoreConfigProperties.builder().withPropertySource(
"datastore.properties").build()) ①
    .build();
```

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

3.2. Property model and MongoDB documents mapping

The Holon Platform PropertyBox type, used by the `Datastore` API to collect and handle the data model attributes values using the Property model, is mapped into a MongoDB **BSON Document** using the following conventions:

- Each `PropertyBox` instance is mapped into a MongoDB BSON *document*, using the **property set** bound to the `PropertyBox` instance to determine the available document nodes **names**.
- Each **property name** is mapped to a document **node name**. By default only the `Path` type properties are used for mapping, using the **path name** as node name.
- Each **property value** available from the `PropertyBox` instance is mapped into a document **node value**, performing suitable data type convesions if required.

3.2.1. Nested documents

Nested MongoDB documents can be represented using either:

1. Nested property names using a dot notation:

When *nested path* names are included in the property set, either using the dot notation or *parent* path declarations, they will be mapped as **nested document** properties in the MongoDB BSON Document.

For example:

```
static final StringProperty NAME = StringProperty.create("name");
static final StringProperty STREET = StringProperty.create("address.street");
static final StringProperty CITY = StringProperty.create("address.city");

static final PropertySet<?> SUBJECT = PropertySet.of(NAME, STREET, CITY);
```

The `SUBJECT` property set will be mapped into a *Document* with:

- A `name` String type node at root level.
- A nested `address` document with the `street` and `city` String type nodes.

2. Nested `PropertyBox` type properties:

A `PropertyBox` type property can be used to represent a nested document. The nested document nodes will be the one represented by the **property set** bound to `PropertyBox` type property.

For this purpose, the PropertyBoxProperty type can be used, which makes available a builder to create a new `PropertyBox` type property providing the **property set** to use.

For example:

```
static final StringProperty NAME = StringProperty.create("name");

static final StringProperty STREET = StringProperty.create("street");
static final StringProperty CITY = StringProperty.create("city");
static final PropertyBoxProperty ADDRESS = PropertyBoxProperty.create("address",
STREET, CITY);

static final PropertySet<?> SUBJECT = PropertySet.of(NAME, ADDRESS);
```

The `SUBJECT` property set will be mapped into a *Document* with the same schema of the previous example, i.e.:

- A `name` String type node at root level.
- A nested `address` document with the `street` and `city` String type nodes.

3.2.2. Multi value document nodes

Multi value (array) MongoDB Document nodes can be represented using a `Collection` type property.

The standard SetPathProperty and ListPathProperty types can be used to represent a `Set` or `List` type collection of values.

```
static final StringProperty NAME = StringProperty.create("name");

static final ListPathProperty<String> SKILLS = ListPathProperty.create("skills",
String.class); ①
static final SetPathProperty<EnumValue> QUALIFICATIONS = SetPathProperty.create(
"qualifications",
    EnumValue.class); ②

static final PropertySet<?> SUBJECT = PropertySet.of(NAME, SKILLS, QUALIFICATIONS);
```

① Will be mapped into an array type Document node value and represented as a Java `List`

② Will be mapped into an array type Document node value and represented as a Java `Set`

3.3. Document id

The MongoDB Datastore supports three data types to represent a **Document id** and to map it to a property:

1: The standard `org.bson.types.ObjectId` BSON type:

```
static final PathProperty<ObjectId> ID = PathProperty.create("_id", ObjectId.class);
```

2: The `String` type:

```
static final StringProperty ID = StringProperty.create("_id");
```

1: The `BigInteger` type:

```
static final NumericProperty<BigInteger> ID = NumericProperty.bigIntegerType("_id");
```

The MongoDB Datastore will perform all the suitable conversions, if required, during the document mapping into and from a `PropertyBox` type.

Regarding the **document id naming strategy**, the following options can be used:

1: If a *single* property is declared as property set identifier and its type is one of the valid document id types (see before), it is used as document id. In this scenario, the property name is irrelevant and **any path name can be used**.

```
static final StringProperty ID = StringProperty.create("my_document_id");
static final StringProperty NAME = StringProperty.create("name");

static final PropertySet<?> SUBJECT = PropertySet.builderOf(ID, NAME).withIdentifier
(ID).build(); ①
```

① The `my_document_id` named property will be used as document id, because it is explicitly declared as property set *identifier*

2: Otherwise, if a property with the default `_id` name is available in the propert set and its type is one of the valid document id types (see before), that property is used as document id.

```
static final StringProperty ID = StringProperty.create("_id");
static final StringProperty NAME = StringProperty.create("name");

static final PropertySet<?> SUBJECT = PropertySet.of(ID, NAME); ①
```

① The `_id` String type property will be used as document id, because it is named with the default `_id` name and its type is a valid document id type

When a document id property name different form the standard `id` is used, the MongoDB Datastore `_save`, *insert* and *update* operations will always include a `_id` document attribute anyway and its value will be synchonized with the property value declared as document id.

3.4. Database collections

For each Datastore operation, the target database **collection name** must be specified using the `DataTarget` representation, providing the collection name as data target name.

```
final static DataTarget<?> TARGET = DataTarget.named("my_collection"); ①
```

① Declare a `DataTarget` for the `my_collection` database collection name

3.5. Auto-generated ids

The MongoDB Datastore API supports the retrieving of the auto-generated document id values.

The auto-generated document id values can be obtained from the `OperationResult` object, returned by the Datastore data manipulation operations, through the `getInsertedKeys()` and related methods.

```
final PathProperty<ObjectId> ID = PathProperty.create("_id", ObjectId.class);

Datastore datastore = getMongoDatastore(); // build or obtain a MongoDB Datastore
PropertyBox value = buildPropertyBoxValue();

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

Optional<ObjectId> idValue = result.getInsertedKey(ID); ②
idValue = result.getFirstInsertedKey(ObjectId.class); ③
```

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

② Get the document id value using the `ID` property, if available

③ Get the first auto-generated document id value

The default **BRING_BACK_GENERATED_IDS** `WriteOption` can be provided to the `Datastore` API operation to bring back any auto-generated document id value into the `PropertyBox` instance which was the subject of the operation.

The auto-generated document id value will be setted in the `PropertyBox` instance using the property which acts as property set identifier. See Document id.

```
final StringProperty ID = StringProperty.create("_id"); ①
final PathProperty<String> TEXT = PathProperty.create("text", String.class);

Datastore datastore = getMongoDatastore(); // build or obtain a MongoDB Datastore

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

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

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

① Document id property declaration using the standard `_id` name

② Create the `PropertyBox` value to insert

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

④ The `ID` property value of the inserted `PropertyBox` is updated with the auto-generated document id value

3.6. Synchronous MongoDB Datastore

The `holon-datastore-mongo-sync` artifact provides the **synchronous** MongoDB `Datastore` API implementation.

Maven coordinates:

```
<groupId>com.holon-platform.mongo</groupId>
<artifactId>holon-datastore-mongo-sync</artifactId>
<version>5.5.0</version>
```

The MongoDatastore interface represents the synchronous MongoDB Datastore API implementation, extending the Datastore API.

A synchronous MongoDB Java driver is required in classpath in order for the datastore to work.

The `MongoDatastore` API, besides the standard `Datastore` API operations, provides a `builder()` method which can be used to create and configure a `MongoDatastore` API istance.

A `com.mongodb.client.MongoClient` reference is required for the `MongoDatastore` instance setup.

```
MongoDatastore datastore = MongoDatastore.builder() ①
    .client(getMongoClient()) ②
    .build();
```

① Obtain a builder to configure and create a new `MongoDatastore` instance

② Set the MongoDB client to use

If you want to reach the goal of a ***complete abstraction*** from the persistence store technology and the persistence model, the core `Datastore` API interface should be used instead of the specific `MongoDatastore` API by your application code. This way, the concrete `Datastore` API implementation may be replaced by a different one at any time, without any change to the codebase.

To reach the goal of a complete abstraction from the persistence store technology and the persistence model, the core `Datastore` API interface should be used by your application code, instead of the specific `MongoDatastore` API:

```
Datastore datastore = MongoDatastore.builder() ①
    .client(getMongoClient()) ②
    .build();
```

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

② Set the MongoDB client to use

For example, given the following property model declaration:

```
final static StringProperty ID = StringProperty.create("_id"); ①
final static StringProperty NAME = StringProperty.create("name"); ②

final static PropertySet<?> SUBJECT = PropertySet.builderOf(ID, NAME).withIdentifier
(ID).build(); ③

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

① Document id property using the standard `_id` name and `String` as value type

② A `String` type document property declation which maps the `name` document attribute

③ Property set declaration including the `ID` and `NAME` properties. The `ID` property is explicitly declared as the identifier property, even if in this case it would not have been necessary because the `ID` property is mapped to the standard `_id` document id attribute name.

④ Data target declation using the `my_collection` database collection name

Below an example of some datastore operations using the *synchronous* MongoDB Datastore API:

```
Datastore datastore = MongoDatastore.builder().client(getMongoClient()).database("
test").build(); ①

PropertyBox value = PropertyBox.builder(SUBJECT).set(NAME, "My name").build();

OperationResult result = datastore.save(TARGET, value); ②
Optional<String> documentId = result.getInsertedKey(ID); ③

result = datastore.insert(TARGET, value, DefaultWriteOption.BRING_BACK_GENERATED_IDS);
④
documentId = value.getValueIfPresent(ID); ⑤

value.setValue(NAME, "Updated name");
datastore.update(TARGET, value); ⑥

datastore.delete(TARGET, value); ⑦

long count = datastore.query(TARGET).filter(NAME.contains("fragment").or(NAME
.startsWith("prefix"))).count(); ⑧

Stream<PropertyBox> results = datastore.query(TARGET).filter(NAME.isNotNull()).sort(
ID.desc()).stream(SUBJECT); ⑨

Optional<String> id = datastore.query(TARGET).filter(NAME.eq("My name")).findOne(ID);
⑩

result = datastore.bulkUpdate(TARGET).set(NAME, "Updated").filter(NAME.isNull())
.execute(); ⑪
long affected = result.getAffectedCount();

result = datastore.bulkDelete(TARGET).filter(NAME.endsWith("suffix")).execute(); ⑫
affected = result.getAffectedCount();
```

① Build a MongoDB `Datastore` bound to the `test` database name

② Save (insert if not present or update otherwise) given `PropertyBox` value in the `my_collection` database collection name (using the `TARGET` declaration)

③ Get the auto-generated document id value, mapped to the `ID` property declaration

④ Insert given value using the `BRING_BACK_GENERATED_IDS` default write option

⑤ The auto-generated document id value will be setted in the provided `PropertyBox` value, bound to the `ID` property

⑥ Update given value

⑦ Delete the value

⑧ Perform a *count* query on the `my_collection` database collection, providing some filters

⑨ Perform a query on the `my_collection` database collection, providing a filter and a sort declaration and projecting the query results as a stream of `PropertyBox` values

⑩ Perform a query to obtain a single value of the `ID` property, if available

⑪ Perform a *bulk* update

⑫ Perform a *bulk* delete

3.7. Asynchronous MongoDB Datastore

The `holon-datastore-mongo-async` artifact provides the **asynchronous** MongoDB `AsyncDatastore` API implementation.

Maven coordinates:

```
<groupId>com.holon-platform.mongo</groupId>
<artifactId>holon-datastore-mongo-async</artifactId>
<version>5.5.0</version>
```

The AsyncMongoDatastore interface represents the asynchronous MongoDB Datastore API implementation, extending the AsyncDatastore API.

An asynchronous MongoDB Java driver is required in classpath in order for the datastore to work.

The `AsyncMongoDatastore` API, besides the standard `AsyncDatastore` API operations, provides a `builder()` method which can be used to create and configure an `AsyncMongoDatastore` API istance.

A `com.mongodb.reactivestreams.client.MongoClient` reference is required for the `AsyncMongoDatastore` instance setup.

```
AsyncMongoDatastore datastore = AsyncMongoDatastore.builder() ①
    .client(getMongoClient()) ②
    .build();
```

① Obtain a builder to configure and create a new `AsyncMongoDatastore` instance

② Set the MongoDB client to use

If you want to reach the goal of a ***complete abstraction*** from the persistence store technology and the persistence model, the `AsyncDatastore` API interface should be used instead of the specific `AsyncMongoDatastore` API by your application code. This way, the concrete `AsyncDatastore` API implementation may be replaced by a different one at any time, without any change to the codebase.

To reach the goal of a complete abstraction from the persistence store technology and the persistence model, the `AsyncDatastore` API interface should be used by your application code, instead of the specific `AsyncMongoDatastore` API:

```
AsyncDatastore datastore = AsyncMongoDatastore.builder() ①
    .client(getMongoClient()) ②
    .build();
```

① Obtain a builder to configure and create a new `AsyncMongoDatastore` instance and expose it as a `AsyncDatastore` API type

② Set the MongoDB client to use

The `AsyncDatastore` API uses the `java.util.concurrent.CompletionStage` interface to provide the asynchronous operation results.

Below an example of some datastore operations using the *asynchronous* MongoDB Datastore API:

```
AsyncDatastore datastore = AsyncMongoDatastore.builder().client(getMongoClient())
.database("test").build(); ①

PropertyBox value = PropertyBox.builder(SUBJECT).set(NAME, "My name").build();

datastore.save(TARGET, value) ②
    .thenApply(result -> result.getInsertedKey(ID)).thenAccept(id -> {
      Optional<String> documentId = id; ③
    });

datastore.insert(TARGET, value, DefaultWriteOption.BRING_BACK_GENERATED_IDS) ④
    .thenApply(result -> value.getValueIfPresent(ID)).thenAccept(id -> {
      Optional<String> documentId = id; ⑤
    });

value.setValue(NAME, "Updated name");
CompletionStage<OperationResult> result = datastore.update(TARGET, value); ⑥

result = datastore.delete(TARGET, value); ⑦

CompletionStage<Long> count = datastore.query(TARGET)
    .filter(NAME.contains("fragment").or(NAME.startsWith("prefix"))).count(); ⑧

CompletionStage<Stream<PropertyBox>> results = datastore.query(TARGET).filter(NAME
.isNotNull()).sort(ID.desc())
    .stream(SUBJECT); ⑨

CompletionStage<Optional<String>> id = datastore.query(TARGET).filter(NAME.eq("My
name")).findOne(ID); ⑩

datastore.bulkUpdate(TARGET).set(NAME, "Updated").filter(NAME.isNull()).execute()
.thenAccept(r -> { ⑪
  long affected = r.getAffectedCount();
});

datastore.bulkDelete(TARGET).filter(NAME.endsWith("suffix")).execute().thenAccept(r ->
{ ⑫
  long affected = r.getAffectedCount();
});
```

① Build a MongoDB `AsyncDatastore` bound to the `test` database name

② Save (insert if not present or update otherwise) given `PropertyBox` value in the `my_collection` database collection name (using the `TARGET` declaration)

③ Get the auto-generated document id value, mapped to the `ID` property declaration

④ Insert given value using the `BRING_BACK_GENERATED_IDS` default write option

⑤ The auto-generated document id value will be setted in the provided `PropertyBox` value, bound to the `ID` property

⑥ Update given value

⑦ Delete the value

⑧ Perform a *count* query on the `my_collection` database collection, providing some filters

⑨ Perform a query on the `my_collection` database collection, providing a filter and a sort declaration and projecting the query results as a stream of `PropertyBox` values

⑩ Perform a query to obtain a single value of the `ID` property, if available

⑪ Perform a *bulk* update

⑫ Perform a *bulk* delete

3.8. Reactive MongoDB Datastore

The `holon-datastore-mongo-reactor` artifact provides the **reactive** MongoDB `AsyncDatastore` API implementation, using the Project Reactor `Mono` and `Flux` to provide and handle the datastore operations results.

Maven coordinates:

```
<groupId>com.holon-platform.mongo</groupId>
<artifactId>holon-datastore-mongo-reactor</artifactId>
<version>5.5.0</version>
```

The ReactiveMongoDatastore interface represents the asynchronous MongoDB Datastore API implementation, extending the ReactiveDatastore API.

A reactive MongoDB Java driver is required in classpath in order for the datastore to work.

The `ReactiveMongoDatastore` API, besides the standard `ReactiveDatastore` API operations, provides a `builder()` method which can be used to create and configure a `ReactiveMongoDatastore` API istance.

A `com.mongodb.reactivestreams.client.MongoClient` reference is required for the `ReactiveMongoDatastore` instance setup.

```
ReactiveMongoDatastore datastore = ReactiveMongoDatastore.builder() ①
    .client(getMongoClient()) ②
    .build();
```

① Obtain a builder to configure and create a new `ReactiveMongoDatastore` instance

② Set the MongoDB client to use

If you want to reach the goal of a ***complete abstraction*** from the persistence store technology and the persistence model, the `ReactiveDatastore` API interface should be used instead of the specific `ReactiveMongoDatastore` API by your application code. This way, the concrete `ReactiveDatastore` API implementation may be replaced by a different one at any time, without any change to the codebase.

To reach the goal of a complete abstraction from the persistence store technology and the persistence model, the `ReactiveDatastore` API interface should be used by your application code, instead of the specific `ReactiveMongoDatastore` API:

```
ReactiveDatastore datastore = ReactiveMongoDatastore.builder() ①
    .client(getMongoClient()) ②
    .build();
```

① Obtain a builder to configure and create a new `ReactiveMongoDatastore` instance and expose it as a `ReactiveDatastore` API type

② Set the MongoDB client to use

The `ReactiveDatastore` API uses the Project Reactor `reactor.core.publisher.Mono` and `reactor.core.publisher.Flux` objects to provide the operation results.

Below an example of some datastore operations using the *reactive* MongoDB Datastore API:

```
ReactiveDatastore datastore = ReactiveMongoDatastore.builder().client(getMongoClient(
)).database("test")
    .build(); ①

PropertyBox value = PropertyBox.builder(SUBJECT).set(NAME, "My name").build();

datastore.save(TARGET, value) ②
    .map(result -> result.getInsertedKey(ID)).doOnSuccess(id -> {
      Optional<String> documentId = id; ③
    });

datastore.insert(TARGET, value, DefaultWriteOption.BRING_BACK_GENERATED_IDS) ④
    .doOnSuccess(id -> {
      Optional<String> documentId = value.getValueIfPresent(ID); ⑤
    });

value.setValue(NAME, "Updated name");
datastore.update(TARGET, value); ⑥

datastore.delete(TARGET, value); ⑦

Mono<Long> count = datastore.query(TARGET).filter(NAME.contains("fragment").or(NAME
.startsWith("prefix")))
    .count(); ⑧

Flux<PropertyBox> results = datastore.query(TARGET).filter(NAME.isNotNull()).sort(ID
.desc()).stream(SUBJECT); ⑨

Mono<String> id = datastore.query(TARGET).filter(NAME.eq("My name")).findOne(ID); ⑩

Mono<OperationResult> result = datastore.bulkUpdate(TARGET).set(NAME, "Updated")
.filter(NAME.isNull())
    .execute(); ⑪
result.doOnSuccess(r -> {
  long affected = r.getAffectedCount();
});

result = datastore.bulkDelete(TARGET).filter(NAME.endsWith("suffix")).execute(); ⑫
result.doOnSuccess(r -> {
  long affected = r.getAffectedCount();
});
```

① Build a MongoDB `ReactiveDatastore` bound to the `test` database name

② Save (insert if not present or update otherwise) given `PropertyBox` value in the `my_collection` database collection name (using the `TARGET` declaration)

③ Get the auto-generated document id value, mapped to the `ID` property declaration

④ Insert given value using the `BRING_BACK_GENERATED_IDS` default write option

⑤ The auto-generated document id value will be setted in the provided `PropertyBox` value, bound to

the `ID` property

⑥ Update given value

⑦ Delete the value

⑧ Perform a *count* query on the `my_collection` database collection, providing some filters

⑨ Perform a query on the `my_collection` database collection, providing a filter and a sort declaration and projecting the query results as a Flux of `PropertyBox` values

⑩ Perform a query to obtain a single value of the `ID` property, if available

⑪ Perform a *bulk* update

⑫ Perform a *bulk* delete

3.9. Custom BSON filters and sorts

The MongoDB Datastore APIs support custom **BSON** filters and sorts expressions, i.e. `QueryFilter` and `QuerySort` type expressions for which the **BSON expression** that represents the filtering and sorting conditions is directly provided.

3.9.1. BsonFilter

The BsonFilter interface is a `QueryFilter` implementation to provide the query restrictions using a **BSON expression**, and it can be used just like any other `QueryFilter` implementation in query definitions.

```
Datastore datastore = getMongoDatastore(); // build or obtain a MongoDB Datastore

long count = datastore.query(DataTarget.named("my_collection"))
    .filter(BsonFilter.create(Filters.size("my_field", 3))) ①
    .count();
```

① Use `BsonFilter` to provide a BSON filter expression that matches all documents where the value of the `my_field` field is an array of the specified size (3)

3.9.2. BsonSort

The BsonSort interface is a `QuerySort` implementation to provide the query sort using a **BSON expression**, and it can be used just like any other `QuerySort` implementation in query definitions.

```
Datastore datastore = getMongoDatastore(); // build or obtain a MongoDB Datastore

Stream<String> values = datastore.query(DataTarget.named("my_collection"))
    .sort(BsonSort.create(Sorts.descending("my_field"))) ①
    .stream(MY_PROPERTY);
```

① Use `BsonSort` to provide a BSON sort specification for a descending sort on the `my_field` field

3.10. Transactions management

The MongoDB Datastore API implementations support **transactions** management, if transaction support is available from the concrete MongoDB database engine.

The multi-document ACID transactions support is available since the **version 4.0** of MongoDB platform and the MongoDB Java driver version **3.8 or higher** is required.

The transactions management operations are made available using:

- The Transactional API for the **synchronous** MongoDB Datastore implementation.
- The AsyncTransactional API for the **asynchronous** MongoDB Datastore implementation.
- The ReactiveTransactional API for the **reactive** MongoDB Datastore implementation.

3.10.1. Synchronous transactions management

The `isTransactional()` or `requireTransactional()` methods of the `Datastore` API can be used to obtain the *transactional* API which can be used for transactions management.

The Transaction interface is used as transaction representation and provides methods to inspect the transaction status and perform the **commit** and **rollback** operations.

```
final Datastore datastore = getMongoDatastore(); // build or obtain a MongoDB
Datastore

datastore.requireTransactional().withTransaction(tx -> { ①
  datastore.save(TARGET, value);
  tx.commit(); ②
});

OperationResult result = datastore.requireTransactional().withTransaction(tx -> { ③
  return datastore.save(TARGET, value);
}, TransactionConfiguration.withAutoCommit()); ④
```

① Obtain the `Transactional` API to execute one or more Datastore operation within a transaction

② Commit the transaction

③ Obtain the `Transactional` API to execute the Datastore operation within a transaction and return a value

④ The transaction is configured with the *auto commit* mode, this way the transaction is automatically committed at the transactional operation end if no error occurred

3.10.2. Asynchronous transactions management

The `isTransactional()` or `requireTransactional()` methods of the `AsyncDatastore` API can be used to obtain the *transactional* API which can be used for transactions management.

The AsyncTransaction interface is used as transaction representation and provides methods to inspect the transaction status and perform the **commit** and **rollback** operations.

```
final AsyncDatastore datastore = getMongoDatastore(); // build or obtain a MongoDB
Datastore

CompletionStage<Boolean> committed = datastore.requireTransactional().withTransaction
(tx -> { ①
  datastore.save(TARGET, value);
  return tx.commit(); ②
});

CompletionStage<OperationResult> result = datastore.requireTransactional()
.withTransaction(tx -> { ③
  return datastore.save(TARGET, value);
}, TransactionConfiguration.withAutoCommit()); ④
```

① Obtain the `AsyncTransactional` API to execute one or more Datastore operation within a transaction

② Commit the transaction

③ Obtain the `AsyncTransactional` API to execute the Datastore operation within a transaction and return a value

④ The transaction is configured with the *auto commit* mode, this way the transaction is automatically committed at the transactional operation end if no error occurred

3.10.3. Reactive transactions management

The `isTransactional()` or `requireTransactional()` methods of the `ReactiveDatastore` API can be used to obtain the *transactional* API which can be used for transactions management.

The ReactiveTransaction interface is used as transaction representation and provides methods to inspect the transaction status and perform the **commit** and **rollback** operations.

```
final ReactiveDatastore datastore = getMongoDatastore(); // build or obtain a MongoDB
Datastore

Flux<Boolean> committed = datastore.requireTransactional().withTransaction(tx -> { ①
  datastore.save(TARGET, value);
  return tx.commit().flux(); ②
});

Flux<OperationResult> result = datastore.requireTransactional().withTransaction(tx ->
{ ③
  return datastore.save(TARGET, value).flux();
}, TransactionConfiguration.withAutoCommit()); ④
```

① Obtain the `ReactiveTransactional` API to execute one or more Datastore operation within a

transaction

② Commit the transaction

③ Obtain the `ReactiveTransactional` API to execute the Datastore operation within a transaction and return a value

④ The transaction is configured with the *auto commit* mode, this way the transaction is automatically committed at the transactional operation end if no error occurred

3.11. Using the `MongoDatabaseHandler` for a direct database access

The MongoDatabaseHandler API can be used obtain a direct reference to the `MongoDatabase` object which represents the Mongo database configured for the current Datastore implementation.

The `MongoDatabase` reference can be used to perform database operations with the MongoDB Java driver directly, using BSON expressions and outside of the standard Datastore API operations.

Each specific MongoDB datastore implementation provides the `MongoDatabaseHandler` support, using a consistent `MongoDatabase` reference type.

```
MongoDatastore datastore = getMongoDatastore();

long count = datastore.withDatabase(db -> { ①
  db.createCollection("test");
  return db.getCollection("test").countDocuments();
});
```

① Get the `MongoDatabase` reference and use it to perform some operations

3.12. Extending the MongoDB Datastore API

3.13. Expression resolvers

The `Datastore` API can be extended using the `ExpressionResolver` API, to add new expression resolution strategies, modify existing ones and to handle new `Expression` types.

See the Datastore API extension documentation section for details.

3.13.1. MongoDB Expression resolvers registration

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

1. Using the MongoDB `Datastore` API instance:

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

```
Datastore datastore = MongoDatastore.builder() //
    .withExpressionResolver(new MyExpressionResolver()) ①
    .build();
```

① Register a new `ExpressionResolver`

Or using the `Datastore` API itself, which extends the `ExpressionResolverSupport` API:

```
datastore.addExpressionResolver(new MyExpressionResolver()); ①
```

① Register and new `ExpressionResolver`

The same registration options are available for the `AsyncDatastore` and `ReactiveDatastore` API and builders.

2. Automatic registration using *Java service extensions*:

The MongoDB Datastore implementatons support `ExpressionResolver` automatic registration using the MongoDatastoreExpressionResolver base type and the default *Java service extensions* modality.

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

When this registration method is used, the expression resolvers defined this way will be registered for **any MongoDB Datastore API instance**.

3.13.2. Specific expression resolvers registration

All the default `Datastore` API operations supports **operation specific** expression resolvers registration, through the `ExpressionResolverSupport` API.

An `ExpressionResolver` registered for a specific `Datastore` API operation execution will be available only for the execution of that operation, and will be ignored by any other `Datastore` API operation.

For example, to register an expression resolver only for a single `Query` execution, the `Query` builder API can be used:

```
long result = datastore.query().target(DataTarget.named("test")) //
    .withExpressionResolver(new MyExpressionResolver()) ①
    .count();
```

① Register an expression resolver only for the specific `Query` operation definition

3.13.3. Expression resolvers priority

According to the standard convention, the `javax.annotation.Priority` annotation can be used on

`ExpressionResolver` classes to indicate in what order the expression resolvers bound to the same *type resolution pair* (i.e. the expression type handled by a resolver and the target expression type into which it will be resolved) must be applied.

The less is the `javax.annotation.Priority` number assigned to a resolver, the higher will be it's priority order.

All the default MongoDB Datastore expression resolvers have the *minimum* priority order, allowing to override their behavior and resolution strategies with custom expression resolvers with a higher assigned priority order (i.e. a priority number less then `Integer.MAX_VALUE`).

3.13.4. Expression validation

The internal MongoDB Datastore *BSON composer engine* will perform **validation** on any `Expression` instance to resolve and each corresponding resolved `Expression` instance, using the default expression `validate()` method.

So the `validate()` method can be used to implement custom expression validation logic and throw an `InvalidExpressionException` when validation fails.

3.13.5. MongoDB Datastore expressions

Besides the standard Datastore API expressions, such as `DataTarget`, `QueryFilter` and `QuerySort`, which can be used to extend the `Datastore` API with new expression implementations and new resolution strategies, the MongoDB `Datastore` API can be extended using a set of **specific BSON resolution expressions**, used by the internal *BSON composer engine* to implement the resolution and composition strategy to obtain the BSON expressions and document mappings from the `Datastore` API meta-language expressions.

These expressions are available from the `com.holonplatform.datastore.mongo.core.expression` package of the `holon-datastore-mongo-core` artifact.

The BsonExpression is the expression which represents a *BSON expression part*, used to compose the actual BSON statement which will be executed using the MongoDB Java driver.

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

For example, supposing to have a `IdIs` class which represents a `QueryFilter` expression type to represent the expression *"the `_id` column name value is equal to a given ObjectId type value"*:

```
class IdIs implements QueryFilter {

  private final ObjectId value;

  public IdIs(ObjectId value) {
    this.value = value;
  }

  public ObjectId getValue() {
    return value;
  }

  @Override
  public void validate() throws InvalidExpressionException {
    if (value == null) {
      throw new InvalidExpressionException("Id value must be not null");
    }
  }

}
```

We want to create an `ExpressionResolver` class to resolve the `IdIs` expression directly into a **BSON epression**, using the `BsonExpression` type. Using the convenience `create` method of the `ExpressionResolver` API, we can do it in the following way:

```
final ExpressionResolver<IdIs, BsonExpression> keyIsResolver = ExpressionResolver
.create( //
    IdIs.class, ①
    BsonExpression.class, ②
    (keyIs, ctx) -> Optional.of(BsonExpression.create(Filters.eq("_id", keyIs.
getValue())))); ③
```

① Expression type to resolve

② Target expression type

③ Expression resolution logic: since we resolve the `IdIs` expression directly into a `BsonExpression` type, the BSON filter representation is provided

After the `ExpressionResolver` is registered in the `Datastore` API, the new `IdIs` expression can be used in the `Datastore` API operations which support the `QueryFilter` expression type just like any other filter expression. For example, in a `Query` expression:

```
Datastore datastore = MongoDatastore.builder().withExpressionResolver(keyIsResolver)
①
    .build();

Query query = datastore.query().filter(new IdIs(new ObjectId("xxxx"))); ②
```

① Register the new expression resolver

② Use the `IdIs` expression in a query definition

Other expression types are used to represent the elements of a query or a Datastore operation. These expression types often represent an *intermediate* expression type, between the highest abstract layer (i.e. an expression of the `Datastore` API meta-language) and the final BSON expression representation.

An example is the BsonProjection to represent a query projection.

3.13.6. MongoDB Expression resolution context

The MongoDB Datastore APIs make available an extension of the standard expression `ResolutionContext`, to provide a set of configuration attributes and BSON resolution context specific operations.

This resolution context extension is represented by the MongoResolutionContext API.

3.14. Commodity factories

The MongoDB `Datastore` API supports *Datastore commodities* registration using a suitable commodity factory type, which is specific for each MongoDB datastore implementation:

- For the **synchronous** `Datastore` implementation, the SyncMongoDatastoreCommodityFactory type.
- For the **asynchronous** `AsyncDatastore` and the **reactive** `ReactiveDatastore` implementation, the AsyncMongoDatastoreCommodityFactory type.

See the Datastore commodities definition and registration documentation section to learn how the *Datastore commodity* architecture can be used to provide extensions to the default `Datastore` API.

These commodity factory types provide a specialized MongoDatastoreCommodityContext API extension as commodity context, to make available a set of MongoDB specific configuration attributes and references, for example:

- The current database name and reference.
- The current client session, if available.
- The current codec registry.
- The default read preference, read concern and write concern.
- The available *expression resolvers*.

Furthermore, it makes available some API methods to interact with the underlying database, such as the `withDatabase(MongoDatabaseOperation operation)` method.

Example: definition of a commodity which provides a method to create a database collection:

```
interface MyCommodity extends DatastoreCommodity { ①

  void createCollection(String name);

}

class MyCommodityImpl implements MyCommodity { ②

  private final MongoDatabaseHandler<MongoDatabase> databaseHandler;

  public MyCommodityImpl(MongoDatabaseHandler<MongoDatabase> databaseHandler) {
    super();
    this.databaseHandler = databaseHandler;
  }

  @Override
  public void createCollection(String name) {
    databaseHandler.withDatabase(db -> {
      db.createCollection(name);
    });
  }

}

class MyCommodityFactory implements SyncMongoDatastoreCommodityFactory<MyCommodity> {
③

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

  @Override
  public MyCommodity createCommodity(SyncMongoDatastoreCommodityContext context)
      throws CommodityConfigurationException {
    return new MyCommodityImpl(context);
  }

}
```

① Datastore commodity API

② Commodity implementation

③ Commodity factory implementation

A Datastore commodity factory class which extends the `SyncMongoDatastoreCommodityFactory` or the `AsyncMongoDatastoreCommodityFactory` interface can be registered in a MongoDB Datastore in two ways:

1. Direct registration using the MongoDB Datastore API builder:

The MongoDB Datastore APIs supports the commodity factory registration using the `withCommodity` builder method.

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

① Register the `MyCommodityFactory` commodity factory in given Datastore implementation

2. Automatic registration using the *Java service extensions*:

To automatically register an commodity factory using the standard *Java service extensions* based method, a class implementing either `SyncMongoDatastoreCommodityFactory` or `AsyncMongoDatastoreCommodityFactory` has to be created and its qualified full name must be specified in a file named `com.holonplatform.datastore.mongo.sync.config.SyncMongoDatastoreCommodityFactory` or `com.holonplatform.datastore.mongo.async.config.AsyncMongoDatastoreCommodityFactory`, placed in the `META-INF/services` folder of the classpath.

When this registration method is used, the commodity factories defined this way will be registered for **any MongoDB Datastore API instance**.

4. Spring ecosystem integration

The `holon-datastore-mongo-spring` artifact provides integration with the Spring framework for the MongoDB Datastore APIs.

Maven coordinates:

```
<groupId>com.holon-platform.mongo</groupId>
<artifactId>holon-datastore-mongo-spring</artifactId>
<version>5.5.0</version>
```

4.1. Enable a MongoDB Datastore bean

A set of Spring configuration class annotations are provided to enable a MongoDB datastore implementation and make it available as a Spring bean.

4.1.1. Synchronous MongoDB Datastore

For the **synchronous** `Datastore` implementation, the EnableMongoDatastore annotation can be used on a Spring configuration class.

To enable a MongoDB `Datastore` bean the following preconditions must be met:

- A synchronous MongoDB `com.mongodb.client.MongoClient` type bean must be available. By default, the `mongoClient` bean name is used to detect the MongoClient bean reference. The client reference bean name can be changed using the `mongoClientReference()` annotation attribute.

- A database name must be specified using the `database()` annotation attribute.

```
@Configuration
@EnableMongoDatastore(database = "test") ①
class ConfigSync {

  @Bean
  public com.mongodb.client.MongoClient mongoClient() { ②
    return com.mongodb.client.MongoClients.create();
  }

}

@Autowired
Datastore datastore; ③
```

① Use `test` as database name

② MongoClient bean definition

③ A synchronous `Datastore` bean is configured and available from Spring context

4.1.2. Asynchronous MongoDB Datastore

For the **asynchronous** `AsyncDatastore` implementation, the EnableMongoAsyncDatastore annotation can be used on a Spring configuration class.

To enable a MongoDB `AsyncDatastore` bean the following preconditions must be met:

- An asynchronous MongoDB `com.mongodb.reactivestreams.client.MongoClient` type bean must be available. By default, the `mongoClient` bean name is used to detect the MongoClient bean reference. The client reference bean name can be changed using the `mongoClientReference()` annotation attribute.
- A database name must be specified using the `database()` annotation attribute.

```
@Configuration
@EnableMongoAsyncDatastore(database = "test") ①
class ConfigAsync {

  @Bean
  public com.mongodb.reactivestreams.client.MongoClient mongoClient() { ②
    return com.mongodb.reactivestreams.client.MongoClients.create();
  }

}

@Autowired
AsyncDatastore asyncDatastore; ③
```

① Use `test` as database name

② MongoClient bean definition

③ An asynchronous `AsyncDatastore` bean is configured and available from Spring context

4.1.3. Reactive MongoDB Datastore

For the **reactive** `ReactiveDatastore` implementation, the EnableMongoReactiveDatastore annotation can be used on a Spring configuration class.

To enable a MongoDB `ReactiveDatastore` bean the following preconditions must be met:

- An asynchronous MongoDB `com.mongodb.reactivestreams.client.MongoClient` type bean must be available. By default, the `mongoClient` bean name is used to detect the MongoClient bean reference. The client reference bean name can be changed using the `mongoClientReference()` annotation attribute.
- A database name must be specified using the `database()` annotation attribute.

```
@Configuration
@EnableMongoReactiveDatastore(database = "test") ①
class ConfigReactive {

  @Bean
  public com.mongodb.reactivestreams.client.MongoClient mongoClient() { ②
    return com.mongodb.reactivestreams.client.MongoClients.create();
  }

}

@Autowired
ReactiveDatastore reactiveDatastore; ③
```

① Use `test` as database name

② MongoClient bean definition

③ A reactive `ReactiveDatastore` bean is configured and available from Spring context

4.2. MongoDB Datastore bean configuration

All the annotations described above to enable a MongoDB Datastore bean provide a set of configuration options, which can be setted using the annotations attributes.

4.2.1. MongoClient bean name

By default, the `mongoClient` bean name is used to detect the `MongoClient` bean reference to use and bind to the MongoDB Datastore implementation. The client reference bean name can be changed using the `mongoClientReference()` annotation attribute.

```
@Configuration
@EnableMongoDatastore(database = "test", mongoClientReference = "syncMongoClient") ①
@EnableMongoAsyncDatastore(database = "test", mongoClientReference = "
asyncMongoClient") ②
class Config {

  @Bean
  public com.mongodb.client.MongoClient syncMongoClient() {
    return com.mongodb.client.MongoClients.create();
  }

  @Bean
  public com.mongodb.reactivestreams.client.MongoClient asyncMongoClient() {
    return com.mongodb.reactivestreams.client.MongoClients.create();
  }

}

@Autowired
Datastore datastore;

@Autowired
AsyncDatastore asyncDatastore;
```

① The `syncMongoClient` bean name is used to enable the synchronous `Datastore` bean

② The `asyncMongoClient` bean name is used to enable the asynchronous `AsyncDatastore` bean

4.2.2. Enum codec strategy

The enumeration type value codec strategy can be configured using the `enumCodecStrategy()` annotation attribute. See `Enum` codec strategy for details.

```
@Configuration
@EnableMongoDatastore(database = "test", enumCodecStrategy = EnumCodecStrategy.
ORDINAL) ①
class Config {

  @Bean
  public com.mongodb.client.MongoClient mongoClient() {
    return com.mongodb.client.MongoClients.create();
  }

}
```

① Set `ORDINAL` as enumeration codec strategy

4.2.3. Mongo database configuration

The following annotation attributes can be used to set the default read preference, read concern and write concern to use:

- The `readPreference()` annotation attribute can be use to set the default **read preference**.
- The `readConcern()` annotation attribute can be use to set the default **read concern**.
- The `writeConcern()` annotation attribute can be use to set the default **write concern**.

```
@Configuration
@EnableMongoDatastore(database = "test", readPreference = MongoReadPreference.PRIMARY, ①
    readConcern = MongoReadConcern.LOCAL, ②
    writeConcern = MongoWriteConcern.ACKNOWLEDGED ③
)
class Config2 {

  @Bean
  public com.mongodb.client.MongoClient mongoClient() {
    return com.mongodb.client.MongoClients.create();
  }

}
```

① Set the default read preference

② Set the default read concern

③ Set the default write concern

4.2.4. Primary mode

Each annotation provides a `primary()` attribute which can be used to control the *primary mode* of the MongoDB Datastore bean registration.

If the *primary mode* is set to `PrimaryMode.TRUE`, the `Datastore` bean created with the corresponding annotation will be marked as **primary** in the Spring application context, meaning that will be the one provided by Spring in case of multiple available candidates, when no specific bean name or qualifier is specified in the dependency injection declaration.

This behaviour is similar to the one obtained with the Spring `@Primary` annotation at bean definition time.

By default, the *primary mode* is set to `PrimaryMode.AUTO`, meaning that the registred MongoDB Datastore bean will be marked as **primary** only when the `MongoClient` bean to which is bound is registered as primary candidate bean.

4.2.5. MongoDB Datastore configuration properties

When a Mongo Datastore bean is configured using one of the available annotations (`@EnableMongoDatastore`, `@EnableMongoAsyncDatastore`, `@EnableMongoReactiveDatastore`), the Spring environment is automatically used as configuration properties source.

This way, many Datastore configuration settings can be provided using a configuration property with the proper name and value.

The supported configuration properties are:

1. The standard Datastore configuration properties, avaible from the DatastoreConfigProperties property set (See Datastore configuration).

The configuration property prefix is `holon.datastore` and the following properties are available:

Table 1. Datastore configuration properties

Name	Type	Meaning
holon.datastore. **trace**	Boolean (`true` / `false`)	Enable/disable Datastore operations *tracing*.

2. An additional set of properties, provided by the MongoDatastoreConfigProperties property set, which can be used as an alternative for the annotations attributes described in the previous sections.

Table 2. MongoDB Datastore configuration properties

Name	Type	Meaning
holon.datastore.mongo. **database**	`String`	Set the database name to which the Datastore is bound.
holon.datastore.mongo. **primary**	Boolean (`true` / `false`)	Mark the MongoDB Datastore bean as *primary* candidate for dependency injection when more than one definition is available.
holon.datastore.mongo. **read-preference**	A valid `String` which identifies one of the names listed in the `MongoReadPreference` enumeration.	Set the default read preference to use.
holon.datastore.mongo. **read-concern**	A valid `String` which identifies one of the names listed in the `MongoReadConcern` enumeration.	Set the default read concern to use.
holon.datastore.mongo. **write-concern**	A valid `String` which identifies one of the names listed in the `MongoWriteConcern` enumeration.	Set the default write concern to use.

Name	Type	Meaning
holon.datastore.mongo. **enum-codec-strategy**	A valid `String` which identifies one of the names listed in the `EnumCodecStrategy` enumeration.	The enumeration type values codec strategy to use. See `Enum` codec strategy.

Example of Datastore configuration properties:

```
holon.datastore.trace=true ①

holon.datastore.mongo.database=test ②
holon.datastore.mongo.read-preference=PRIMARY ③
```

① Enable tracing

② Set the database name to `test`

③ Set the default read preference to `PRIMARY`

4.2.6. Datastore extension and configuration using the Spring context

The MongoDB Datastore implementation supports the standard Holon Platform Datastore Spring integration features for Datastore beans configuration and extension, which includes:

- Datastore **configuration post processing** using DatastorePostProcessor type beans.
- Datastore **extension** through ExpressionResolver registration using DatastoreResolver annotated beans.
- Datastore **commodity factory** registration using DatastoreCommodityFactory annotated beans.

```
@DatastoreResolver ①
class MyFilterExpressionResolver implements QueryFilterResolver<MyFilter> {

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

  @Override
  public Optional<QueryFilter> resolve(MyFilter expression, ResolutionContext context)
      throws InvalidExpressionException {
    // implement actual MyFilter expression resolution
    return Optional.empty();
  }

}

@Component
class MyDatastorePostProcessor implements DatastorePostProcessor { ②

  @Override
  public void postProcessDatastore(ConfigurableDatastore datastore, String
datastoreBeanName) {
    // configure Datastore
  }

}

@DatastoreCommodityFactory ③
class MyCommodityFactory implements SyncMongoDatastoreCommodityFactory<MyCommodity> {

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

  @Override
  public MyCommodity createCommodity(SyncMongoDatastoreCommodityContext context)
      throws CommodityConfigurationException {
    // create commodity instance
    return new MyCommodity();
  }

}
```

① Automatically register a Datastore expression resolver using the `@DatastoreResolver` annotation

② Post process Datastore configuration using a `DatastorePostProcessor` type Spring bean

③ Automatically register a Datastore commodity factory using the `@DatastoreCommodityFactory` annotation

When a Mongo Datastore bean is configured using one of the available annotations (`@EnableMongoDatastore`, `@EnableMongoAsyncDatastore`, `@EnableMongoReactiveDatastore`), the Datastore extension and configuration using the Spring context is automatically enabled.

When one of the available annotation is not used to configure the MongoDB datastore bean, the Datastore extension and configuration support can be explicitly enabled using the `@EnableDatastoreConfiguration` annotation on Spring configuration classes.

4.3. Multiple MongoDB Datastores configuration

When more than one MongoDB Datastore bean has to be configured using the `@Enable*` annotations, the `dataContextId` attribute can be used to assign a different **data context id** to each MongoDB Datastore bean definition, in order to:

- Provide different sets of configuration properties using the same Spring environment.
- Provide a default *name pattern matching strategy* with the `MongoClient` bean definition to use for each MongoDB Datastore to configure: if not directly specified with the `mongoClientReference()` attribute, the `MongoClient` bean definition to use for each MongoDB Datastore will be detected in Spring context using the bean name pattern: `mongoClient_{datacontextid}` where `{datacontextid}` is equal to the `dataContextId` attribute of the annotation.

When a *data context id* is defined, a Spring **qualifier** named the same as the *data context id* will be associated to the generated MongoDB Datastore bean definitions, and such qualifier can be later used to obtain the right MongoDB Datastore instance through dependency injection.

```
@Configuration class Config {

  @Configuration
  @EnableMongoDatastore(database = "test", dataContextId = "one") ①
  static class Config1 {

    @Bean(name = "mongoClient_one")
    public com.mongodb.client.MongoClient mongoClient() {
      return com.mongodb.client.MongoClients.create();
    }

  }

  @Configuration
  @EnableMongoDatastore(database = "test", dataContextId = "two") ②
  static class Config2 {

    @Bean(name = "mongoClient_two")
    public com.mongodb.client.MongoClient mongoClient() {
      return com.mongodb.client.MongoClients.create();
    }

  }

}

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

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

① Configure the first MongoDB Datastore using `one` as *data context id*: by default the bean named `mongoClient_one` will be used as `MongoClient`

② Configure the second MongoDB Datastore using `two` as *data context id*: by default the bean named `mongoClient_two` will be used as `MongoClient`

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

5. Spring Boot integration

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

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

Maven coordinates:

```
<groupId>com.holon-platform.mongo</groupId>
<artifactId>holon-datastore-mongo-spring-boot</artifactId>
<version>5.5.0</version>
```

The MongoDB Datastore auto-configuration class support the configuration of the three available MongoDB Datastore implementations:

5.1. Synchronous MongoDB Datastore

A synchronous MongoDB `Datastore` is auto-configured only when:

- A `MongoDatastore` type bean is not already available from the Spring application context.
- A valid `com.mongodb.client.MongoClient` type bean is available from the Spring application context.

5.2. Asynchronous MongoDB Datastore

An asynchronous MongoDB `AsyncDatastore` is auto-configured only when:

- A `AsyncMongoDatastore` type bean is not already available from the Spring application context.
- A valid `com.mongodb.reactivestreams.client.MongoClient` type bean is available from the Spring application context.

5.3. Reactive MongoDB Datastore

A reactive MongoDB `ReactiveDatastore` is auto-configured only when:

- A `ReactiveMongoDatastore` type bean is not already available from the Spring application context.
- A valid `com.mongodb.reactivestreams.client.MongoClient` type bean is available from the Spring application context.
- The `holon-datastore-mongo-reactor` artifact (and the Project Reactor core library) is available from classpath.

5.4. MongoDB Datastore configuration

The Spring Boot application properties can be used to configure the MongoDB Datastore beans, for example:

application.yml

```
holon:
  datastore:
    trace: true
  mongo:
    database: test
```

See MongoDB Datastore configuration properties.

5.5. Disabling the MongoDB Datastore auto-configuration feature

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

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

5.6. Spring Boot starters

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

1. Default MongoDB Datastore starter provides the dependencies to the Holon MongoDB Datastore Spring Boot integration artifact `holon-datastore-mongo-spring-boot` and the synchronous and asynchronous MongoDB Datastore implementations.

The **MongoDB Java driver** library is included in the starters dependencies.

Furthermore, the following additional dependencies are provided:

- The Holon Platform Core Module Spring Boot integration base starter (`holon-starter`).
- The base Spring Boot starter (`spring-boot-starter`), see the Spring Boot starters documentation for details.

Maven coordinates:

```
<groupId>com.holon-platform.mongo</groupId>
<artifactId>holon-starter-mongo-datastore</artifactId>
<version>5.5.0</version>
```

2. Reactive MongoDB Datastore starter provides the same dependencies as the default MongoDB Datastore starter, adding the `holon-datastore-mongo-reactor` artifact to provide the reactive MongoDB Datastore implementation and auto-configuration.

Maven coordinates:

```
<groupId>com.holon-platform.mongo</groupId>
<artifactId>holon-starter-mongo-datastore-reactor</artifactId>
<version>5.5.0</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 **MongoDB Datastore** module is `com.holonplatform.datastore.mongo`.

7. System requirements

7.1. Java

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

7.2. MongoDB Java Drivers

The MongoDB Java Driver version 3.5+ is required to use this module.

The **MongoDB Java Driver version 3.8+** is recommended for a full compatibility with the module APIs, for example to use the transactions management support.

For **asynchronous and reactive** interaction support, the MongoDB Reactive Streams Java Driver is required.