MongoDB3.2

# 前言

## 新特性

Key new features of the 3.2 driver include:

* Support for bypassing [document validation](http://docs.mongodb.org/manual/release-notes/3.2/#document-validation) on collections where document validation has been enabled.
* Builder support for new [aggregation stages](http://docs.mongodb.org/manual/release-notes/3.2/#new-aggregation-stages) and [accumulators](http://docs.mongodb.org/manual/release-notes/3.2/#new-accumulators-for-group-stage) for the $group stage.
* Support for [read concern](http://docs.mongodb.org/manual/release-notes/3.2/#readconcern).
* Support for [version 3 text indexes](http://docs.mongodb.org/manual/release-notes/3.2/#text-index-version-3)
* Support for write concern on all DBCollection helpers for the findandmodify command

## 升级到3.2

**Upgrading from 3.1.x**

The 3.2 release is binary and source compatible with the 3.1 release, except for methods that have been added to interfaces that have been marked as unstable.

**Upgrading from 2.x**

Please see the Upgrading guide in the 3.0 driver reference documentation.

**System Requirements**

The minimum JVM is now Java 6: however, specific features require Java 7:

* SSL support requires Java 7 in order to perform host name verification, which is enabled by default. See below and on [SSL](http://mongodb.github.io/mongo-java-driver/3.2/driver/reference/connecting/ssl/) for details on how to disable host name verification.
* The asynchronous API requires Java 7, as by default it relies on [AsynchronousSocketChannel](http://docs.oracle.com/javase/7/docs/api/java/nio/channels/AsynchronousSocketChannel.html) for its implementation. See [Async](http://mongodb.github.io/mongo-java-driver/3.2/driver-async/) for details on configuring the driver to use [Netty](http://netty.io/) instead.

# MongoDB Driver

## 起步

To help you get started quickly on the new driver, follow:

* [Installation](http://mongodb.github.io/mongo-java-driver/3.2/driver/getting-started/installation-guide/)
* [Quick Tour](http://mongodb.github.io/mongo-java-driver/3.2/driver/getting-started/quick-tour/)
* [Admin Quick Tour](http://mongodb.github.io/mongo-java-driver/3.2/driver/getting-started/quick-tour-admin/)

To help you get started with the legacy DB and DBCollection API, follow:

* [Legacy Quick Tour](http://mongodb.github.io/mongo-java-driver/2.13/getting-started/quick-tour/)
* [Legacy Admin Quick Tour](http://mongodb.github.io/mongo-java-driver/2.13/getting-started/quick-tour-admin/)

### 安装指南

There are two Maven artifacts available in the 3.2 release. The preferred artifact for new applications ismongodb-driver however, we still publish the legacy mongo-java-driver uber-jar. The recommended way to get started using one of the drivers in your project is with a dependency management system.

#### MongoDB Driver

The MongoDB Driver is the updated synchronous Java driver that includes the legacy API as well as a new generic MongoCollection interface that complies with a new cross-driver CRUD specification.

**IMPORTANT**

For OSGi-based applications: due to the fact that there are classes from the com.mongodb package in both this artifact and in the mongodb-driver-core artifact, on which this depends, this artifact is not an OSGi bundle. Please use the mongo-java-driver uber jar (described below) instead.

<dependencies>

<dependency>

<groupId>org.mongodb</groupId>

<artifactId>mongodb-driver</artifactId>

<version>3.2.1</version>

</dependency>

</dependencies>

You can also download the jars [directly](https://oss.sonatype.org/content/repositories/releases/org/mongodb/mongodb-driver/3.2.1) from sonatype.

**Note:** mongodb-driver requires the following dependencies: [bson](https://oss.sonatype.org/content/repositories/releases/org/mongodb/bson/3.2.1) and [mongodb-driver-core](https://oss.sonatype.org/content/repositories/releases/org/mongodb/mongodb-driver-core/3.2.1)

#### Uber MongoDB Java Driver

An uber jar that contains everything you need; the BSON library, the core library and the mongodb-driver.

**NOTE**

For OSGi-based applications: this artifact is a valid OSGi bundle.

<dependencies>

<dependency>

<groupId>org.mongodb</groupId>

<artifactId>mongo-java-driver</artifactId>

<version>3.2.1</version>

</dependency>

</dependencies>

You can also download the jars [directly](https://oss.sonatype.org/content/repositories/releases/org/mongodb/mongo-java-driver/3.2.1) from sonatype.

### 使用入门

The following code snippets come from the QuickTour.java example code that can be found with the[driver source](https://github.com/mongodb/mongo-java-driver/blob/master/driver/src/examples/tour/QuickTour.java).

**NOTE**

See the [installation guide](http://mongodb.github.io/mongo-java-driver/3.2/driver/getting-started/installation-guide/) for instructions on how to install the MongoDB Driver.

#### Make a Connection

The following example shows five ways to connect to the database mydb on the local machine. If the database does not exist, MongoDB will create it for you.

*// To directly connect to a single MongoDB server*

*// (this will not auto-discover the primary even if it's a member of a replica set)*

MongoClient mongoClient = **new** MongoClient();

*// or*

MongoClient mongoClient = **new** MongoClient( **"localhost"** );

*// or*

MongoClient mongoClient = **new** MongoClient( **"localhost"** , 27017 );

*// or, to connect to a replica set, with auto-discovery of the primary, supply a seed list of members*

MongoClient mongoClient = **new** MongoClient(

Arrays.asList(**new** ServerAddress(**"localhost"**, 27017),

**new** ServerAddress(**"localhost"**, 27018),

**new** ServerAddress(**"localhost"**, 27019)));

*// or use a connection string*

MongoClientURI connectionString = **new** MongoClientURI(**"mongodb://localhost:27017,localhost:27018,localhost:27019"**);

MongoClient mongoClient = **new** MongoClient(connectionString);

MongoDatabase database = mongoClient.getDatabase(**"mydb"**);

At this point, the database object will be a connection to a MongoDB server for the specified database.

#### MongoClient

The MongoClient instance actually represents a pool of connections to the database; you will only need one instance of class MongoClient even with multiple threads.

**IMPORTANT**

Typically you only create one MongoClient instance for a given database cluster and use it across your application. When creating multiple instances:

* All resource usage limits (max connections, etc) apply per MongoClient instance
* To dispose of an instance, make sure you call MongoClient.close() to clean up resources

#### Get a Collection

To get a collection to operate upon, specify the name of the collection to the [getCollection()](http://api.mongodb.org/java/3.2/?com/mongodb/client/MongoDatabase.html#getCollection-java.lang.String-) method:

The following example gets the collection test:

MongoCollection<Document> collection = database.getCollection(**"test"**);

#### Insert a Document

Once you have the collection object, you can insert documents into the collection. For example, consider the following JSON document; the document contains a field info which is an embedded document:

{

**"name"** : **"MongoDB"**,

**"type"** : **"database"**,

**"count"** : 1,

**"info"** : {

x : 203,

y : 102

}

}

To create the document using the Java driver, use the [Document](http://api.mongodb.org/java/3.2/?org/bson/Document.html) class. You can use this class to create the embedded document as well.

Document doc = **new** Document(**"name"**, **"MongoDB"**)

.append(**"type"**, **"database"**)

.append(**"count"**, 1)

.append(**"info"**, **new** Document(**"x"**, 203).append(**"y"**, 102));

To insert the document into the collection, use the insertOne() method.

collection.insertOne(doc);

#### Add Multiple Documents

To add multiple documents, you can use the insertMany() method.

The following example will add multiple documents of the form:

{ **"i"** : value }

Create the documents in a loop.

List<Document> documents = **new** ArrayList<Document>();

**for** (**int** i = 0; i < 100; i++) {

documents.add(**new** Document(**"i"**, i));

}

To insert these documents to the collection, pass the list of documents to the insertMany() method.

collection.insertMany(documents);

#### Count Documents in A Collection

Now that we’ve inserted 101 documents (the 100 we did in the loop, plus the first one), we can check to see if we have them all using the [count()](http://api.mongodb.org/java/3.2/?com/mongodb/client/MongoCollection#count--.html) method. The following code should print 101.

System.out.println(collection.count());

#### Query the Collection

Use the [find()](http://api.mongodb.org/java/3.2/?com/mongodb/client/MongoCollection.html#find--) method to query the collection.

##### Find the First Document in a Collection

To get the first document in the collection, call the [first()](http://api.mongodb.org/java/3.2/?com/mongodb/client/MongoIterable.html#first--) method on the [find()](http://api.mongodb.org/java/3.2/?com/mongodb/client/MongoCollection.html#find--) operation.collection.find().first() returns the first document or null rather than a cursor. This is useful for queries that should only match a single document, or if you are interested in the first document only.

The following example prints the first document found in the collection.

Document myDoc = collection.find().first();

System.out.println(myDoc.toJson());

The example should print the following document:

{ "**\_id**" : **{ "$oid" : "551582c558c7b4fbacf16735" }**,

"**name**" : **"MongoDB"**, "**type**" : **"database"**, "**count**" : 1,

"**info**" : **{ "x" :** 203**, "y" :** 102 **}** }

**NOTE**

The \_id element has been added automatically by MongoDB to your document and your value will differ from that shown. MongoDB reserves field names that start with “\_” and “$” for internal use.

##### Find All Documents in a Collection

To retrieve all the documents in the collection, we will use the find() method. The find() method returns a FindIterable instance that provides a fluent interface for chaining or controlling find operations. Use theiterator() method to get an iterator over the set of documents that matched the query and iterate. The following code retrieves all documents in the collection and prints them out (101 documents):

MongoCursor<Document> cursor = collection.find().iterator();

**try** {

**while** (cursor.hasNext()) {

System.out.println(cursor.next().toJson());

}

} **finally** {

cursor.close();

}

Although the following idiom is permissible, its use is discouraged as the application can leak a cursor if the loop terminates early:

**for** (Document cur : collection.find()) {

System.out.println(cur.toJson());

}

#### Get A Single Document with a Query Filter

We can create a filter to pass to the find() method to get a subset of the documents in our collection. For example, if we wanted to find the document for which the value of the “i” field is 71, we would do the following:

**import** **static** com.mongodb.client.model.Filters.\*;

myDoc = collection.find(eq(**"i"**, 71)).first();

System.out.println(myDoc.toJson());

and it should just print just one document

{ "**\_id**" : **{ "$oid" : "5515836e58c7b4fbc756320b" }**, "**i**" : 71}

NOTE

Use the [Filters](http://mongodb.github.io/mongo-java-driver/3.2/builders/filters/), [Sorts](http://mongodb.github.io/mongo-java-driver/3.2/builders/sorts/), [Projections](http://mongodb.github.io/mongo-java-driver/3.2/builders/projections/) and [Updates](http://mongodb.github.io/mongo-java-driver/3.2/builders/updates/) helpers for simple and concise ways of building up queries.

#### Get a Set of Documents with a Query

We can use the query to get a set of documents from our collection. For example, if we wanted to get all documents where "i" > 50, we could write:

*// now use a range query to get a larger subset*

Block<Document> printBlock = **new** Block<Document>() {

@Override

**public** **void** apply(**final** Document document) {

System.out.println(document.toJson());

}

};

collection.find(gt(**"i"**, 50)).forEach(printBlock);

Notice we use the forEach method on FindIterable which applies a block to each document and we print all documents where i > 50.

We could also get a range, say 50 < i <= 100:

collection.find(and(gt(**"i"**, 50), lte(**"i"**, 100))).forEach(printBlock);

#### Sorting documents

We can also use the [Sorts](http://mongodb.github.io/mongo-java-driver/3.2/builders/sorts/) helpers to sort documents. We add a sort to a find query by calling the sort()method on a FindIterable. Below we use the [exists()](http://mongodb.github.io/mongo-java-driver/3.2/builders/filters/#elements) helper and sort [descending("i")](http://mongodb.github.io/mongo-java-driver/3.2/builders/sorts/#descending) helper to sort our documents:

myDoc = collection.find(exists(**"i"**)).sort(descending(**"i"**)).first();

System.out.println(myDoc.toJson());

#### Projecting fields

Sometimes we don’t need all the data contained in a document, the [Projections](http://mongodb.github.io/mongo-java-driver/3.2/builders/projections/) helpers help build the projection parameter for the find operation. Below we’ll sort the collection, exclude the \_id field by using the[Projections.excludeId](http://mongodb.github.io/mongo-java-driver/3.2/builders/projections/#exclusion) and output the first matching document:

myDoc = collection.find().projection(excludeId()).first();

System.out.println(myDoc.toJson());

#### Aggregations

Sometimes we need to aggregate the data stored in MongoDB. The [Aggregates](http://mongodb.github.io/mongo-java-driver/3.2/builders/aggregation/) helper provides builders for each of type of aggregation stage.

Below we’ll do a simple two step transformation that will calculate the value of i \* 10. First we find all Documents where i > 0 by using the [Aggregates.match](http://mongodb.github.io/mongo-java-driver/3.2/builders/aggregation/#match) helper. Then we reshape the document by using[Aggregates.project](http://mongodb.github.io/mongo-java-driver/3.2/builders/aggregation/#project) in conjunction with the [$multiply](http://docs.mongodb.org/manual/reference/operator/aggregation/multiply/) operator to calculate the “ITimes10” value:

collection.aggregate(asList(

match(gt(**"i"**, 0)),

project(Document.parse(**"{ITimes10: {$multiply: ['$i', 10]}}"**)))

).forEach(printBlock);

For [$group](http://mongodb.github.io/mongo-java-driver/3.2/builders/aggregation/#group) operations use the [Accumulators](http://api.mongodb.org/java/3.2/?com/mongodb/client/model/Accumulators.html) helper for any [accumulator operations](http://docs.mongodb.org/manual/reference/operator/aggregation/group/#accumulator-operator). Below we sum up all the values of i by using the [Aggregates.group](http://mongodb.github.io/mongo-java-driver/3.2/builders/aggregation/#group) helper in conjunction with the [Accumulators.sum](http://api.mongodb.org/java/3.2/?com/mongodb/client/model/Accumulators#sum-java.lang.String-TExpression-.html) helper:

myDoc = collection.aggregate(singletonList(group(**null**, sum(**"total"**, **"$i"**)))).first();

System.out.println(myDoc.toJson());

**NOTE**

Currently, there are no helpers for [aggregation expressions](http://docs.mongodb.org/manual/meta/aggregation-quick-reference/#aggregation-expressions). Use the [Document.parse()](http://mongodb.github.io/mongo-java-driver/3.2/bson/extended-json/) helper to quickly build aggregation expressions from extended JSON.

#### Updating documents

There are numerous [update operators](http://docs.mongodb.org/manual/reference/operator/update-field/) supported by MongoDB.

To update at most a single document (may be 0 if none match the filter), use the [updateOne](http://api.mongodb.org/java/3.2/?com/mongodb/client/MongoCollection.html#updateOne-org.bson.conversions.Bson-org.bson.conversions.Bson-) method to specify the filter and the update document. Here we use the [Updates.set](http://mongodb.github.io/mongo-java-driver/3.2/builders/updates/#set) helper to update the first document that meets the filter i equals 10 and set the value of i to 110:

collection.updateOne(eq(**"i"**, 10), set(**"i"**, 110));

To update all documents matching the filter use the [updateMany](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoCollection.html#updateMany-org.bson.conversions.Bson-org.bson.conversions.Bson-) method. Here we use the [Updates.inc](http://mongodb.github.io/mongo-java-driver/3.2/builders/updates/#increment)helper to increment the value of i by 100 where i is less than 100.

UpdateResult updateResult = collection.updateMany(lt(**"i"**, 100), inc(**"i"**, 100));

System.out.println(updateResult.getModifiedCount());

The update methods return an [UpdateResult](http://api.mongodb.org/java/3.2/?com/mongodb/client/result/UpdateResult.html) which provides information about the operation including the number of documents modified by the update.

#### Deleting documents

To delete at most a single document (may be 0 if none match the filter) use the [deleteOne](http://api.mongodb.org/java/3.2/?com/mongodb/client/MongoCollection.html#deleteOne-org.bson.conversions.Bson-) method:

collection.deleteOne(eq(**"i"**, 110));

To delete all documents matching the filter use the [deleteMany](http://api.mongodb.org/java/3.2/?com/mongodb/client/MongoCollection.html#deleteMany-org.bson.conversions.Bson-) method.  
Here we delete all documents where i is greater or equal to 100:

DeleteResult deleteResult = collection.deleteMany(gte(**"i"**, 100));

System.out.println(deleteResult.getDeletedCount());

The delete methods return a [DeleteResult](http://api.mongodb.org/java/3.2/?com/mongodb/client/result/DeleteResult.html) which provides information about the operation including the number of documents deleted.

#### Bulk operations

These new commands allow for the execution of bulk insert/update/delete operations. There are two types of bulk operations:

1. Ordered bulk operations.

Executes all the operation in order and error out on the first write error.

1. Unordered bulk operations.

Executes all the operations and reports any the errors.

Unordered bulk operations do not guarantee order of execution.

Let’s look at two simple examples using ordered and unordered operations:

*// 2. Ordered bulk operation - order is guarenteed*

collection.bulkWrite(

Arrays.asList(**new** InsertOneModel<>(**new** Document(**"\_id"**, 4)),

**new** InsertOneModel<>(**new** Document(**"\_id"**, 5)),

**new** InsertOneModel<>(**new** Document(**"\_id"**, 6)),

**new** UpdateOneModel<>(**new** Document(**"\_id"**, 1),

**new** Document(**"$set"**, **new** Document(**"x"**, 2))),

**new** DeleteOneModel<>(**new** Document(**"\_id"**, 2)),

**new** ReplaceOneModel<>(**new** Document(**"\_id"**, 3),

**new** Document(**"\_id"**, 3).append(**"x"**, 4))));

*// 2. Unordered bulk operation - no guarantee of order of operation*

collection.bulkWrite(

Arrays.asList(**new** InsertOneModel<>(**new** Document(**"\_id"**, 4)),

**new** InsertOneModel<>(**new** Document(**"\_id"**, 5)),

**new** InsertOneModel<>(**new** Document(**"\_id"**, 6)),

**new** UpdateOneModel<>(**new** Document(**"\_id"**, 1),

**new** Document(**"$set"**, **new** Document(**"x"**, 2))),

**new** DeleteOneModel<>(**new** Document(**"\_id"**, 2)),

**new** ReplaceOneModel<>(**new** Document(**"\_id"**, 3),

**new** Document(**"\_id"**, 3).append(**"x"**, 4))),

**new** BulkWriteOptions().ordered(**false**));

**IMPORTANT**

Use of the bulkWrite methods is not recommended when connected to pre-2.6 MongoDB servers, as this was the first server version to support bulk write commands for insert, update, and delete in a way that allows the driver to implement the correct semantics for BulkWriteResult and BulkWriteException. The methods will still work for pre-2.6 servers, but performance will suffer, as each write operation has to be executed one at a time.

### 管理入门

This is the second part of the MongoDB driver quick tour. In the [quick tour](http://mongodb.github.io/mongo-java-driver/3.2/driver/getting-started/quick-tour/) we looked at how to use the Java driver to execute basic CRUD operations. In this section we’ll look at some of the administrative features available in the driver.

The following code snippets come from the QuickTourAdmin.java example code that can be found with the[driver source](https://github.com/mongodb/mongo-java-driver/blob/master/driver/src/examples/tour/QuickTourAdmin.java).

**NOTE**

See the [installation guide](http://mongodb.github.io/mongo-java-driver/3.2/driver/getting-started/installation-guide/) for instructions on how to install the MongoDB Driver.

#### Setup

To get started we’ll quickly connect and create a mongoClient, database and collection variable for use in the examples below:

MongoClient mongoClient = **new** MongoClient();

MongoDatabase database = mongoClient.getDatabase(**"mydb"**);

MongoCollection<Document> collection = database.getCollection(**"test"**);

NOTE

Calling the getDatabase() on MongoClient does not create a database. Only when a database is written to will a database be created. Examples include the creation of an index or the insertion of a document into a previously non-existent collection.

#### Get A List of Databases

You can get a list of the available databases:

**for** (String name: mongoClient.listDatabaseNames()) {

System.out.println(name);

}

#### Drop A Database

You can drop a database by name using a MongoClient instance:

mongoClient.getDatabase(**"databaseToBeDropped"**).drop();

#### Create A Collection

Collections in MongoDB are created automatically simply by inserted a document into it. Using the[createCollection](http://api.mongodb.org/java/3.2/?com/mongodb/client/MongoDatabase.html#createCollection-java.lang.String-) method, you can also create a collection explicitly in order to customize its configuration. For example, to create a capped collection sized to 1 megabyte:

database.createCollection(**"cappedCollection"**,

**new** CreateCollectionOptions().capped(**true**).sizeInBytes(0x100000));

#### Get A List of Collections

You can get a list of the available collections in a database:

**for** (String name : database.listCollectionNames()) {

System.out.println(name);

}

#### Drop A Collection

You can drop a collection by using the drop() method:

collection.drop();

#### Create An Index

MongoDB supports secondary indexes. To create an index, you just specify the field or combination of fields, and for each field specify the direction of the index for that field. We can use the [Indexes](http://mongodb.github.io/mongo-java-driver/3.2/builders/indexes/) helpers to create index keys:

*// create an ascending index on the "i" field*

collection.createIndex(Indexes.ascending(**"i"**));

#### Get a List of Indexes on a Collection

Use the listIndexes() method to get a list of indexes. The following lists the indexes on the collectiontest:

**for** (**final** Document index : collection.listIndexes()) {

System.out.println(index.toJson());

}

The example should print the following indexes:

{ "**v**" : 1, "**key**" : **{ "\_id" :** 1 **}**, "**name**" : **"\_id\_"**, "**ns**" : **"mydb.test"** }

{ "**v**" : 1, "**key**" : **{ "i" :** 1 **}**, "**name**" : **"i\_1"**, "**ns**" : **"mydb.test"** }

#### Text indexes

MongoDB also provides text indexes to support text search of string content. Text indexes can include any field whose value is a string or an array of string elements. To create a text index use the [Indexes.text](http://mongodb.github.io/mongo-java-driver/3.2/builders/indexes/#text-index)static helper:

*// create a text index on the "content" field*

coll.createIndex(Indexes.text(**"content"**));

As of MongoDB 2.6, text indexes are now integrated into the main query language and enabled by default (here we use the [Filters.text](http://mongodb.github.io/mongo-java-driver/3.2/builders/filters/#evaluation) helper):

*// Insert some documents*

collection.insertOne(**new** Document(**"\_id"**, 0).append(**"content"**, **"textual content"**));

collection.insertOne(**new** Document(**"\_id"**, 1).append(**"content"**, **"additional content"**));

collection.insertOne(**new** Document(**"\_id"**, 2).append(**"content"**, **"irrelevant content"**));

*// Find using the text index*

**long** matchCount = collection.count(Filters.text(**"textual content -irrelevant"**));

System.out.println(**"Text search matches: "** + matchCount);

*// Find using the $language operator*

Bson textSearch = Filters.text(**"textual content -irrelevant"**, **new** TextSearchOptions().language(**"english"**));

matchCount = collection.count(textSearch);

System.out.println(**"Text search matches (english): "** + matchCount);

*// Find the highest scoring match*

Document projection = **new** Document(**"score"**, **new** Document(**"$meta"**, **"textScore"**));

Document myDoc = collection.find(textSearch).projection(projection).first();

System.out.println(**"Highest scoring document: "** + myDoc.toJson());

and it should print:

Text search matches: 2

Text search matches (english): 2

Highest scoring document: { "**\_id**" : 1, "**content**" : **"additional content"**, "**score**" : 0.75}

For more information about text search see the [text index](http://docs.mongodb.org/manual/core/index-text) and [$text query operator](http://docs.mongodb.org/manual/reference/operator/query/text) documentation.

#### Running a command

While not all commands have a specific helper, however you can run any [command](http://docs.mongodb.org/manual/reference/command) by using the[runCommand()](http://api.mongodb.org/java/3.2/?com/mongodb/client/MongoDatabase.html#runCommand-org.bson.conversions.Bson-com.mongodb.ReadPreference-) method. Here we call the [buildInfo](http://mongodb.github.io/mongo-java-driver/3.2/driver/getting-started/quick-tour-admin/%7B%7B%20docsref%20%22reference/command/buildInfo%22%20%7D%7D) command:

Document buildInfo = database.runCommand(**new** Document(**"buildInfo"**, 1));

System.out.println(buildInfo);

## 进阶指南

The reference documentation for the Java driver focuses on high-level documentation and use-cases. The Javadoc-generated API reference is [here](http://api.mongodb.org/java/3.2/).

* [Connecting](http://mongodb.github.io/mongo-java-driver/3.2/driver/reference/connecting/): Documentation of the driver’s support for connecting to MongoDB servers
* [CRUD](http://mongodb.github.io/mongo-java-driver/3.2/driver/reference/crud/): Documentation of the driver’s support for CRUD operations
* [Management](http://mongodb.github.io/mongo-java-driver/3.2/driver/reference/management/): Documentation of the driver’s support for logging and monitoring of its internal state

### Connecting

The reference documentation for connecting to a MongoDB server deployment is divided into three sections:

* [Connection Settings](http://mongodb.github.io/mongo-java-driver/3.2/driver/reference/connecting/connection-settings/): documentation of the various ways to specify the properties of a connection
* [Authenticating](http://mongodb.github.io/mongo-java-driver/3.2/driver/reference/connecting/authenticating/): detailed documentation of the various ways to specify authentication credentials
* [SSL](http://mongodb.github.io/mongo-java-driver/3.2/driver/reference/connecting/ssl/): Detailed documentation of the various ways to specify the properties of an SSL connection

#### Connection Settings

The Java driver has two ways of specifying the settings of a connection to a MongoDB server deployment.

##### Connection String

The [connection string](http://docs.mongodb.org/manual/reference/connection-string/) is the simplest way to specify the properties of a connection. . A connection string mostly follows [RFC 3986](http://tools.ietf.org/html/rfc3986), with the exception of the domain name. For MongoDB, it is possible to list multiple domain names separated by a comma. Below are some example connection strings.

* For a standalone mongod, mongos, or a direct connection to a member of a replica set:
* mongodb://host:27017
* To connect to multiple mongos or a replica set:
* mongodb://host1:27017,host2:27017

The [authentication guide](http://mongodb.github.io/mongo-java-driver/3.2/driver/reference/connecting/authenticating/) contains information on how to provide credentials in the connection string.

##### The Database Component

The database component is optional and is used to indicate which database to authenticate against. When the database component is not provided, the “admin” database is used.

mongodb://host:27017/mydb

Above, the database by the name of “mydb” is where the credentials are stored for the application.

**NOTE**

Some drivers utilize the database component to indicate which database to work with by default. The Java driver, while it parses the database component, does not use the database component for anything other than authentication.

##### Options

Many options can be provided via the connection string. The ones that cannot may be provided in a[MongoClientOptions](http://api.mongodb.org/java/3.2/?com/mongodb/MongoClientOptions.html) instance. To provide an option, append a ? to the connection string and separate options by an &.

mongodb://host:27017/?replicaSet=rs0&maxPoolSize=200

The above connection string sets the “replicaSet” value to “rs0” and the “maxPoolSize” to “200”.

For a comprehensive list of the available options, see the [MongoClientURI](http://api.mongodb.org/java/3.2/?com/mongodb/MongoClientURI.html) documentation.

##### MongoClient

A [MongoClient](http://api.mongodb.org/java/3.2/?com/mongodb/MongoClient.html) instance will be the root object for all interaction with MongoDB. It is all that is needed to handle connecting to servers, monitoring servers, and performing operations against those servers. Without any arguments, constructing a [MongoClient](http://api.mongodb.org/java/3.2/?com/mongodb/MongoClient.html) instance will connect to “localhost” port 27017.

MongoClient client = **new** MongoClient();

Alternatively, a connection string may be provided:

MongoClient client = **new** MongoClient(**new** MongoClientURI(**"mongodb://host:27017,host2:27017/?replicaSet=rs0"**));

Finally, the [MongoClientOptions](http://api.mongodb.org/java/3.2/?com/mongodb/MongoClientOptions.html) class provides an in-code way to set the same options from a connection string. This is sometimes necessary, as the connection string does not allow an application to configure as many properties of the connection as MongoClientOptions.  
[MongoClientOptions](http://api.mongodb.org/java/3.2/?com/mongodb/MongoClientOptions.html) instances are immutable, so to create one an application uses a builder:

MongoClientOptions options = MongoClientOptions.builder().cursorFinalizerEnabled(**false**).build();

MongoClient client = **new** MongoClient(options);

It’s also possible to combine [MongoClientOptions](http://api.mongodb.org/java/3.2/?com/mongodb/MongoClientOptions.html) with [MongoClientURI](http://api.mongodb.org/java/3.2/?com/mongodb/MongoClientURI.html), for situations in which an application needs to set some options in code but others via the connection string:

MongoClientURI uri = **new** MongoClientURI(**"mongodb://host:27017,host2:27017/?replicaSet=rs0"**,MongoClientOptions.builder().cursorFinalizerEnabled(**false**))

MongoClient client = **new** MongoClient(uri);

#### SSL

The Java driver supports SSL connections to MongoDB servers using the underlying support for SSL provided by the JDK. You can configure the driver to use SSL either with MongoClientURI or withMongoClientOptions.

With MongoClientURI, specify `ssl=true as a query parameter, as in:

**new** MongoClientURI(**"mongodb://localhost/?ssl=true"**)

With MongoClientOptions, set the sslEnabled property to true, as in:

MongoClientOptions.builder().sslEnabled(**true**).build()

##### Host name verification

By default, the driver ensures that the host name included in the server’s SSL certificate(s) matches the host name(s) provided when constructing a MongoClient. However, this host name verification requires a Java 7 JVM, as it relies on additions to the javax.net.SSLParameters class that were introduced in Java 7. If your application must run on Java 6, or for some other reason you need to disable host name verification, you must expicitly indicate this in MongoClientOptions using the sslInvalidHostNameAllowed property:

MongoClientOptions.builder().sslEnabled(**true**).sslInvalidHostNameAllowed(**true**).build()

##### JVM system properties

A typical application will need to set several JVM system properties to ensure that the client is able to validate the SSL certificate presented by the server:

* javax.net.ssl.trustStore: the path to a trust store containing the certificate of the signing authority
* javax.net.ssl.trustStorePassword: the password to access this trust store

The trust store is typically created with the [keytool](http://docs.oracle.com/javase/8/docs/technotes/tools/unix/keytool.html) command line program provided as part of the JDK. For example:

keytool -importcert -trustcacerts -file <path to certificate authority file>

-keystore <path to trust store> -storepass <password>

A typical application will also need to set several JVM system properties to ensure that the client presents an SSL certificate to the MongoDB server:

* javax.net.ssl.keyStore: the path to a key store containing the client’s SSL certificates
* javax.net.ssl.keyStorePassword: the password to access this key store

The key store is typically created with the [keytool](http://docs.oracle.com/javase/8/docs/technotes/tools/unix/keytool.html) or the [openssl](https://www.openssl.org/docs/apps/openssl.html) command line program.

For more information on configuring a Java application for SSL, please refer to the  
[JSSE Reference Guide](http://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html).

#### Authentication

The Java driver supports all MongoDB [authentication mechanisms](http://docs.mongodb.org/manual/core/authentication/), including those only available in the MongoDB [Enterprise Edition](http://docs.mongodb.org/manual/administration/install-enterprise/).

An authentication credential is represented as an instance of the [MongoCredential](http://api.mongodb.org/java/3.2/?com/mongodb/MongoCredential.html) class, which includes static factory methods for each of the supported authentication mechanisms. A list of these instances must be passed to the driver via one of several [MongoClient](http://api.mongodb.org/java/3.2/?com/mongodb/MongoClient.html) constructors that take a parameter of typeList<MongoCredential>. Alternatively, a single [MongoCredential](http://api.mongodb.org/java/3.2/?com/mongodb/MongoCredential.html) can be created implicity via a[MongoClientURI](http://api.mongodb.org/java/3.2/?com/mongodb/MongoClientURI.html) and passed to a [MongoClient](http://api.mongodb.org/java/3.2/?com/mongodb/MongoClient.html) constructor that takes a [MongoClientURI](http://api.mongodb.org/java/3.2/?com/mongodb/MongoClientURI.html) parameter.

**NOTE**

Given the flexibility of role-based access control in MongoDB, it is usually sufficient to authenticate with a single user, but, for completeness, the driver accepts a list of credentials.

##### Default authentication mechanism

MongoDB 3.0 changed the default authentication mechanism from [MONGODB-CR](http://docs.mongodb.org/manual/core/authentication/#mongodb-cr-authentication) to [SCRAM-SHA-1](http://docs.mongodb.org/manual/core/authentication/#scram-sha-1-authentication). To create a credential that will authenticate properly regardless of server version, create a credential using the following static factory method:

**import** com.mongodb.MongoCredential;

*// ...*

String user; *// the user name*

String database; *// the name of the database in which the user is defined*

**char**[] password; *// the password as a character array*

*// ...*

MongoCredential credential = MongoCredential.createCredential(user,

database,

password);

or with a connection string:

MongoClientURI uri = **new** MongoClientURI(**"mongodb://user1:pwd1@host1/?authSource=db1"**);

This is the recommended approach as it will make upgrading from MongoDB 2.6 to MongoDB 3.0 seamless, even after [upgrading the authentication schema](http://docs.mongodb.org/manual/release-notes/3.0-scram/#upgrade-mongodb-cr-to-scram).

##### SCRAM-SHA-1

To explicitly create a credential of type [SCRAM-SHA-1](#scram-sha-1-authentication) use the following static factory method:

MongoCredential credential = MongoCredential.createScramSha1Credential(user,

database,

password);

or with a connection string:

MongoClientURI uri = **new** MongoClientURI(**"mongodb://user1:pwd1@host1/?authSource=db1&authMechanism=SCRAM-SHA-1"**);

##### MONGODB-CR

To explicitly create a credential of type [MONGODB-CR](http://docs.mongodb.org/manual/core/authentication/#mongodb-cr-authentication) use the following static factory method:

MongoCredential credential = MongoCredential.createMongoCRCredential(user,

database,

password);

or with a connection string:

MongoClientURI uri = **new** MongoClientURI(**"mongodb://user1:pwd1@host1/?authSource=db1&authMechanism=MONGODB-CR"**);

Note that this is not recommended as a credential created in this way will fail to authenticate after an authentication schema upgrade from MONGODB-CR to SCRAM-SHA-1.

##### x.509

The [x.509](http://docs.mongodb.org/manual/core/authentication/#x-509-certificate-authentication) mechanism authenticates a user whose name is derived from the distinguished subject name of the X.509 certificate presented by the driver during SSL negotiation. This authentication method requires the use of SSL connections with certificate validation and is available in MongoDB 2.6 and newer. To create a credential of this type use the following static factory method:

String user; *// The x.509 certificate derived user name, e.g. "CN=user,OU=OrgUnit,O=myOrg,..."*

MongoCredential credential = MongoCredential.createMongoX509Credential(user);

or with a connection string:

MongoClientURI uri = **new** MongoClientURI(**"mongodb://subjectName@host1/?authMechanism=MONGODB-X509"**);

See the MongoDB server [x.509 tutorial](http://docs.mongodb.org/manual/tutorial/configure-x509-client-authentication/#add-x-509-certificate-subject-as-a-user) for more information about determining the subject name from the certificate.

##### Kerberos (GSSAPI)

[MongoDB Enterprise](http://www.mongodb.com/products/mongodb-enterprise) supports proxy authentication through a Kerberos service. To create a credential of type [Kerberos (GSSAPI)](http://docs.mongodb.org/manual/core/authentication/#kerberos-authentication) use the following static factory method:

String user; *// The Kerberos user name, including the realm, e.g. "user1@MYREALM.ME"*

*// ...*

MongoCredential credential = MongoCredential.createGSSAPICredential(user);

or with a connection string:

MongoClientURI uri = **new** MongoClientURI(**"mongodb://username%40REALM.com@host1/?authMechanism=GSSAPI"**);

**NOTE**

The method refers to the GSSAPI authentication mechanism instead of Kerberos because technically the driver is authenticating via the [GSSAPI](https://tools.ietf.org/html/rfc4752) SASL mechanism.

To successfully authenticate via Kerberos, the application typically must specify several system properties so that the underlying GSSAPI Java libraries can acquire a Kerberos ticket:

java.security.krb5.realm=MYREALM.ME

java.security.krb5.kdc=mykdc.myrealm.me

NOTE

The GSSAPI authentication mechanism is supported only in the following environments:

* Linux: Java 6 and above
* Windows: Java 7 and above with [SSPI](https://msdn.microsoft.com/en-us/library/windows/desktop/aa380493)
* OS X: Java 7 and above

##### LDAP (PLAIN)

[MongoDB Enterprise](http://www.mongodb.com/products/mongodb-enterprise) supports proxy authentication through a Lightweight Directory Access Protocol (LDAP) service. To create a credential of type [LDAP](http://docs.mongodb.org/manual/core/authentication/#ldap-proxy-authority-authentication) use the following static factory method:

String user; *// The LDAP user name*

**char**[] password; *// The LDAP password*

*// ...*

MongoCredential credential = MongoCredential.createPlainCredential(user, **"$external"**, password);

or with a connection string:

MongoClientURI uri = **new** MongoClientURI(**"mongodb://user1@host1/?authSource=$external&authMechanism=PLAIN"**);

NOTE

The method refers to the plain authentication mechanism instead of LDAP because technically the driver is authenticating via the [PLAIN](https://www.ietf.org/rfc/rfc4616.txt) SASL mechanism.

### CRUD

For a walkthrough of the main CRUD operations please refer to the [Quick Tour](http://mongodb.github.io/mongo-java-driver/3.2/driver/getting-started/quick-tour/).

All CRUD-related methods in the Java driver are accessed through the [MongoCollection](http://api.mongodb.org/java/3.2/?com/mongodb/client/MongoCollection.html) interface. Instances of [MongoCollection](http://api.mongodb.org/java/3.2/?com/mongodb/client/MongoCollection.html) can be obtained from a  
[MongoClient](http://api.mongodb.org/java/3.2/?com/mongodb/client/MongoClient.html) instance by way of a [MongoDatabase](http://api.mongodb.org/java/3.2/?com/mongodb/client/MongoDatabase.html):

MongoClient client = **new** MongoClient();

MongoDatabase database = client.getDatabase(**"mydb"**);

MongoCollection<Document> collection = database.getCollection(**"mycoll"**);

[MongoCollection](http://api.mongodb.org/java/3.2/?com/mongodb/client/MongoCollection.html) is a generic interface: the TDocument type parameter is the class that clients use to insert or replace documents in a collection, and the default type returned from find and aggregate.

The single-argument getCollection method returns an instance of MongoCollection<Document>, and so with this type of collection an application uses instances of the Document class:

MongoCollection<Document> collection = database.getCollection(**"mycoll"**);

*// insert a document*

Document document = **new** Document(**"x"**, 1)

collection.insertOne(document);

document.append(**"x"**, 2).append(**"y"**, 3);

*// replace a document*

collection.replaceOne(Filters.eq(**"\_id"**, document.get(**"\_id"**)), document);

*// find documents*

List<Document> foundDocument = collection.find().into(**new** ArrayList<Document>());

#### CodecRegistry

An overload of the getCollection method allows clients to specify a different class for representing BSON documents. For example, users of the legacy CRUD API from the 2.x driver series may wish to continue using BasicDBObject in order to ease the transition to the new CRUD API:

*// Pass BasicDBObject.class as the second argument*

MongoCollection<BasicDBObject> collection = database.getCollection(**"mycoll"**, BasicDBObject.class);

*// insert a document*

BasicDBObject document = **new** BasicDBObject(**"x"**, 1)

collection.insertOne(document);

document.append(**"x"**, 2).append(**"y"**, 3);

*// replace a document*

collection.replaceOne(Filters.eq(**"\_id"**, document.get(**"\_id"**), document);

*// find documents*

List<BasicDBObject> foundDocument = collection.find().into(**new** ArrayList<BasicDBObject>());

There are two requirements that must be met for any class used in this way:

* a Codec for it must be registered in the MongoCollection’s CodecRegistry
* the Codec must be one that encodes and decodes a full BSON document (and not just, for example, a single BSON value like an Int32)

By default, a MongoCollection is configured with Codecs for three classes:

* Document
* BasicDBObject
* BsonDocument

Applications, however, are free to register Codec implementations for other classes by customizing theCodecRegistry. New CodecRegistry instances are configurable at three levels:

* In a MongoClient via MongoClientOptions
* In a MongoDatabase via its withCodecRegistry method
* In a MongoCollection via its withCodecRegistry method

Consider the case of encoding and decoding instances of the UUID class. The Java driver by default encodes instances of UUID using a byte ordering that is not compatible with other MongoDB drivers, and changing the default would be quite dangerous. But it is possible for new applications that require interoperability across multiple drivers to be able to change that default, and they can do that with aCodecRegistry.

*// Replaces the default UuidCodec with one that uses the new standard UUID representation*

CodecRegistry codecRegistry =

CodecRegistries.fromRegistries(CodecRegistries.fromCodecs(**new** UuidCodec(UuidRepresentation.STANDARD)),

MongoClient.getDefaultCodecRegistry());

*// globally*

MongoClientOptions options = MongoClientOptions.builder()

.codecRegistry(codecRegistry).build();

MongoClient client = **new** MongoClient(**new** ServerAddress(), options);

*// or per database*

MongoDatabase database = client.getDatabase(**"mydb"**)

.withCodecRegistry(codecRegistry);

*// or per collection*

MongoCollection<Document> collection = database.getCollection(**"mycoll"**)

.withCodecRegistry(codecRegistry);

#### Write Concern

Applications can configure the WriteConcern that a MongoCollection uses for write operations. LikeCodecRegistry, the WriteConcern can be configured at three levels:

* In a MongoClient via MongoClientOptions
* In a MongoDatabase via its withWriteConcern method
* In a MongoCollection via its withWriteConcern method

#### Read Preference

Applications can configure the ReadPreference that a MongoCollection uses for read operations. LikeWriteConcern, the ReadPreference can be configured at three levels:

* In a MongoClient via MongoClientOptions
* In a MongoDatabase via its withReadPreference method
* In a MongoCollection via its withReadPreference method

#### Immutability of MongoDatabase and MongoCollection

Instance of MongoDatabase and MongoCollection are immutable, so rather than mutate the state of theMongoCollection on which they are invoked, the three methods discussed above return new instances. Applications should therefore be sure to store the result of the method call. For example:

*// CORRECT: The results of the method calls are chained and the final one is referenced*

*// by collection*

MongoCollection<Document> collection = database.getCollection(**"mycoll"**).withWriteConcern(WriteConcern.JOURNALED)

.withReadPreference(ReadPreference.primary())

.withCodecRegistry(newRegistry);

*// INCORRECT: withReadPreference returns a new instance of MongoCollection*

*// It does not modify the collection it's called on. So this will*

*// have no effect*

collection.withReadPreference(ReadPreference.secondary());

### GridFS

GridFS is a specification for storing and retrieving files that exceed the BSON-document size limit of 16MB.

Instead of storing a file in a single document, GridFS divides a file into parts, or chunks, and stores each of those chunks as a separate document. By default GridFS limits chunk size to 255k. GridFS uses two collections to store files. The chunks collection stores the file chunks, and the files collection stores the file metadata.

When you query a GridFS store for a file, the driver or client will reassemble the chunks as needed. GridFS is useful not only for storing files that exceed 16MB but also for storing any files for which you want access without having to load the entire file into memory.

**NOTE**

For more information about GridFS see the [MongoDB GridFS documentation](http://docs.mongodb.org/manual/core/gridfs/).

The following code snippets come from the GridFSTour.java example code that can be found with the[driver source](https://github.com/mongodb/mongo-java-driver/blob/master/driver/src/examples/gridfs/GridFSTour.java).

#### Connecting to GridFS

Interactions with GridFS are done via the [GridFSBucket](http://api.mongodb.org/java/3.2/?com/mongodb/client/gridfs/GridFSBucket.html) class. To create a GridFSBucket use the[GridFSBuckets](http://api.mongodb.org/java/3.2/?com/mongodb/client/gridfs/GridFSBuckets.html) factory class.

Creating a GridFSBucket requires an instance of a [MongoDatabase](http://api.mongodb.org/java/3.2/?com/mongodb/client/MongoDatabase.html) and you can optionally provide a custom bucket name.

The following example shows how to create a GridFSBucket:

*// Create a gridFSBucket using the default bucket name "fs"*

GridFSBucket gridFSBucket = GridFSBuckets.create(myDatabase);

*// Create a gridFSBucket with a custom bucket name "files"*

GridFSBucket gridFSBucket = GridFSBuckets.create(myDatabase, **"files"**);

#### Uploading to GridFS

There are two main ways to upload data into GridFS.

##### UploadFromStream

The [uploadFromStream](http://api.mongodb.org/java/3.2/?com/mongodb/client/gridfs/GridFSBucket.html#openUploadStream-java.lang.String-com.mongodb.client.gridfs.model.GridFSUploadOptions-) method reads the contents of an [InputStream](http://docs.oracle.com/javase/8/docs/api/index.html?java/io/InputStream.html) and saves it to the GridFSBucket. The size of the chunks defaults to 255 bytes, but can be configured via the [GridFSUploadOptions](http://api.mongodb.org/java/3.2/?com/mongodb/client/gridfs/model/GridFSUploadOptions.html).

The following example uploads an InputStream into GridFSBucket:

*// Get the input stream*

InputStream streamToUploadFrom = **new** FileInputStream(**new** File(**"/tmp/mongodb-tutorial.pdf"**));

*// Create some custom options*

GridFSUploadOptions options = **new** GridFSUploadOptions()

.chunkSizeBytes(1024)

.metadata(**new** Document(**"type"**, **"presentation"**));

ObjectId fileId = gridFSBucket.uploadFromStream(**"mongodb-tutorial"**, streamToUploadFrom, options);

##### OpenUploadStream

The [openUploadStream](http://api.mongodb.org/java/3.2/?com/mongodb/client/gridfs/GridFSBucket.html#openUploadStream-java.lang.String-com.mongodb.client.gridfs.model.GridFSUploadOptions-) method returns a [GridFSUploadStream](http://api.mongodb.org/java/3.2/?mongodb/client/gridfs/GridFSUploadStream.html) which extends [OutputStream](http://docs.oracle.com/javase/8/docs/api/index.html?java/io/OutputStream.html) and can be written to.

The GridFSUploadStream buffers data until it reaches the chunkSizeBytes and then inserts the chunk into the chunks collection. When the GridFSUploadStream is closed, the final chunk is written and the file metadata is inserted into the files collection.

The following example uploads an into GridFSBucket via the returned OutputStream:

**byte**[] data = **"Data to upload into GridFS"**.getBytes(StandardCharsets.UTF\_8);

GridFSUploadStream uploadStream = gridFSBucket.openUploadStream(**"sampleData"**, options);

uploadStream.write(data);

uploadStream.close();

System.out.println(**"The fileId of the uploaded file is: "** + uploadStream.getFileId().toHexString());

**NOTE**

GridFS will automatically create indexes on the files and chunks collections on first upload of data into the GridFS bucket.

#### Finding files stored in GridFS

To find the files stored in the GridFSBucket use the [find](http://api.mongodb.org/java/3.2/?com/mongodb/client/gridfs/GridFSBucket.html#find--) method.

The following example prints out the filename of each file stored:

gridFSBucket.find().forEach(

**new** Block<GridFSFile>() {

@Override

**public** **void** apply(**final** GridFSFile gridFSFile) {

System.out.println(gridFSFile.getFilename());

}

});

You can also provide a custom filter to limit the results returned. The following example prints out the filenames of all files with a “image/png” value set as the contentType in the user defined metadata document:

gridFSBucket.find(eq(**"metadata.contentType"**, **"image/png"**)).forEach(

**new** Block<GridFSFile>() {

@Override

**public** **void** apply(**final** GridFSFile gridFSFile) {

System.out.println(gridFSFile.getFilename());

}

});

#### Downloading from GridFS

There are four main ways to download data from GridFS.

##### DownloadFromStream

The [downloadToStream](http://api.mongodb.org/java/3.2/?com/mongodb/client/gridfs/GridFSBucket.html#downloadToStream-org.bson.types.ObjectId-java.io.OutputStream-) method reads the contents from MongoDB and writes the data directly to the provided [OutputStream](http://docs.oracle.com/javase/8/docs/api/index.html?java/io/OutputStream.html).

The following example downloads a file into the provided OutputStream:

FileOutputStream streamToDownloadTo = **new** FileOutputStream(**"/tmp/mongodb-tutorial.pdf"**);

gridFSBucket.downloadToStream(fileId, streamToDownloadTo);

streamToDownloadTo.close();

System.out.println(streamToDownloadTo.toString());

##### DownloadToStreamByName

If you don’t know the [ObjectId](http://api.mongodb.org/java/3.2/?org/bson/types/ObjectId.html) of the file you want to download, then you use the [downloadToStreamByName](http://api.mongodb.org/java/3.2/?com/mongodb/client/gridfs/GridFSBucket.html#downloadToStreamByName-java.lang.String-java.io.OutputStream-com.mongodb.client.gridfs.model.GridFSDownloadByNameOptions-)method. By default it will download the latest version of the file. Use the [GridFSDownloadByNameOptions](http://api.mongodb.org/java/3.2/?com/mongodb/client/gridfs/model/GridFSDownloadByNameOptions.html) to configure which version to download.

The following example downloads the original version of the file named “mongodb-tutorial” into theOutputStream:

FileOutputStream streamToDownloadTo = **new** FileOutputStream(**"/tmp/mongodb-tutorial.pdf"**);

GridFSDownloadByNameOptions downloadOptions = **new** GridFSDownloadByNameOptions().revision(0);

gridFSBucket.downloadToStreamByName(**"mongodb-tutorial"**, streamToDownloadTo, downloadOptions);

streamToDownloadTo.close();

##### OpenDownloadStream

The [openDownloadStream](http://api.mongodb.org/java/3.2/?com/mongodb/client/gridfs/GridFSBucket.html#openDownloadStream-org.bson.types.ObjectId-) method returns a [GridFSDownloadStream](http://api.mongodb.org/java/3.2/?mongodb/client/gridfs/GridFSDownloadStream.html) which extends [InputStream](http://docs.oracle.com/javase/8/docs/api/index.html?java/io/InputStream.html) and can be read from.

The following example reads from the GridFSBucket via the returned InputStream:

GridFSDownloadStream downloadStream = gridFSBucket.openDownloadStream(fileId);

**int** fileLength = (**int**) downloadStream.getGridFSFile().getLength();

**byte**[] bytesToWriteTo = **new** **byte**[fileLength];

downloadStream.read(bytesToWriteTo);

downloadStream.close();

System.out.println(**new** String(bytesToWriteTo, StandardCharsets.UTF\_8));

##### OpenDownloadStreamByName

You can also open a GridFSDownloadStream by searching against the filename, using the[openDownloadStreamByName](http://api.mongodb.org/java/3.2/?com/mongodb/client/gridfs/GridFSBucket.html#openDownloadStreamByName-java.lang.String-com.mongodb.client.gridfs.model.GridFSDownloadByNameOptions-) method. By default it will download the latest version of the file. Use the[GridFSDownloadByNameOptions](http://api.mongodb.org/java/3.2/?com/mongodb/client/gridfs/model/GridFSDownloadByNameOptions.html) to configure which version to download.

The following example downloads the latest version of the file named “sampleData” into the OutputStream:

GridFSDownloadStream downloadStream = gridFSBucket.openDownloadStreamByName(**"sampleData"**);

**int** fileLength = (**int**) downloadStream.getGridFSFile().getLength();

**byte**[] bytesToWriteTo = **new** **byte**[fileLength];

downloadStream.read(bytesToWriteTo);

downloadStream.close();

System.out.println(**new** String(bytesToWriteTo, StandardCharsets.UTF\_8));

#### Renaming files

If you should need to rename a file, then the [rename](http://api.mongodb.org/java/3.2/?com/mongodb/client/gridfs/GridFSBucket.html#rename-org.bson.types.ObjectId-java.lang.String-) method can be used.

The following example renames a file to “mongodbTutorial”:

gridFSBucket.rename(fileId, **"mongodbTutorial"**);

**NOTE**

The rename method requires an ObjectId rather than a filename to ensure the correct file is renamed.

To rename multiple revisions of the same filename, first retrieve the full list of files. Then for every file that should be renamed then execute rename with the corresponding \_id.

#### Deleting files

To delete a file from the GridFSBucket use the [delete](http://api.mongodb.org/java/3.2/?com/mongodb/client/gridfs/GridFSBucket.html#delete-org.bson.types.ObjectId-) method.

The following example deletes a file from the GridFSBucket:

gridFSBucket.delete(fileId);

### Management

The driver provides two mechanisms for examining its state:

* [Logging](http://mongodb.github.io/mongo-java-driver/3.2/driver/reference/management/logging/): Comprehensive logging of all operations using [SLF4J](http://www.slf4j.org/)
* [Monitoring](http://mongodb.github.io/mongo-java-driver/3.2/driver/reference/management/monitoring/): Built on [JMX](http://docs.oracle.com/javase/8/docs/technotes/guides/jmx/)

#### Logging

By default, logging is enabled via the popular [SLF4J](http://www.slf4j.org/) API. The use of [SLF4J](http://www.slf4j.org/) is optional; the driver will use SLF4J if the driver detects the presence of SLF4J in the classpath. Otherwise, the driver will fall back to JUL (java.util.logging).

The driver uses the following logger names:

* org.mongodb.driver: the root logger
  + cluster: for logs related to monitoring of the MongoDB servers to which the driver connects
  + connection: for logs related to connections and connection pools
  + protocol: for logs related to protocol message sent to and received from a MongoDB server
    - insert: for logs related to insert messages and responses
    - update: for logs related to update messages and responses
    - delete: for logs related to delete messages and responses
    - query: for logs related to query messages and responses
    - getmore: for logs related to getmore messages and responses
    - killcursor: for logs related to killcursor messages and responses
    - command: for logs related to command messages and responses
  + uri: for logs related to connection string parsing
  + management: for logs related to JMX

#### Monitoring(JMX)

The driver uses [JMX](http://docs.oracle.com/javase/8/docs/technotes/guides/jmx/) to create [MXBeans](http://docs.oracle.com/javase/tutorial/jmx/mbeans/mxbeans.html) that allow an application or end user to monitor various aspects of the driver.

The driver creates MXBean instances of a single type: [ConnectionPoolStatisticsMBean](http://api.mongodb.org/java/3.2/?com/mongodb/management/ConnectionPoolStatisticsMBean.html). The driver registers one ConnectionPoolStatisticsMBean instance per each server it connects to. For example, in the case of a replica set, the driver creates an instance per each non-hidden member of the replica set.

Each MXBean instance is required to be registered with a unique object name, which consists of a domain and a set of named properties. All MXBean instances created by the driver are under the domain"org.mongodb.driver". Instances of ConnectionPoolStatisticsMBean will have the following properties:

* clusterId: a client-generated unique identifier, required to ensure object name uniqueness in situations where an application has multiple MongoClient instances connected to the same MongoDB server deployment
* host: the host name of the server
* port: the port on which the server is listening
* minSize: the minimum allowed size of the pool, including idle and in-use members
* maxSize: the maximum allowed size of the pool, including idle and in-use members
* size: the current size of the pool, including idle and and in-use members
* waitQueueSize: the current size of the wait queue for a connection from this pool
* checkedOutCount: the current count of connections that are currently in use

##### Command Monitoring

The driver implements the [command monitoring specification](https://github.com/mongodb/specifications/blob/master/source/command-monitoring/command-monitoring.rst), which allows an application to attach its own event listeners that are notified when commands are started and when they sucessfully completed or fail.

Command listeners are registered individually for each instance of MongoClient by configuringMongoClientOptions with one or more instances of a class that implements the [CommandListener](http://api.mongodb.org/java/3.2/?com/mongodb/event/CommandListener.html)interface. Consider the following, obviously simplistic, implementation of the CommandListener interface:

**public** **class** TestCommandListener **implements** CommandListener {

@Override

**public** **void** commandStarted(**final** CommandStartedEvent event) {

System.out.println(String.format(**"Sent command '%s:%s' with id %s to database '%s' "**

+ **"on connection '%s' to server '%s'"**,

event.getCommandName(),

event.getCommand().get(event.getCommandName()),

event.getRequestId(),

event.getDatabaseName(),

event.getConnectionDescription()

.getConnectionId(),

event.getConnectionDescription().getServerAddress()));

}

@Override

**public** **void** commandSucceeded(**final** CommandSucceededEvent event) {

System.out.println(String.format(**"Successfully executed command '%s' with id %s "**

+ **"on connection '%s' to server '%s'"**,

event.getCommandName(),

event.getRequestId(),

event.getConnectionDescription()

.getConnectionId(),

event.getConnectionDescription().getServerAddress()));

}

@Override

**public** **void** commandFailed(**final** CommandFailedEvent event) {

System.out.println(String.format(**"Failed execution of command '%s' with id %s "**

+ **"on connection '%s' to server '%s' with exception '%s'"**,

event.getCommandName(),

event.getRequestId(),

event.getConnectionDescription()

.getConnectionId(),

event.getConnectionDescription().getServerAddress(),

event.getThrowable()));

}

}

and an instance of MongoClientOptions configured with an instance of TestCommandListener:

MongoClientOptions options = MongoClientOptions.builder()

.addCommandListener(**new** TestCommandListener())

.build();

A MongoClient configured with these options will print a message to System.out before sending each command to a MongoDB server, and another message upon either successful completion or failure of each command.

# MongoDB Async Driver

Welcome to the MongoDB Async Java driver documentation hub.

* **Getting Started**

The [Getting Started](http://mongodb.github.io/mongo-java-driver/3.2/driver-async/getting-started/) guide contains installation instructions and a simple tutorial to get up and running quickly.

* **Reference**

For more detailed documentation, see the [Reference](http://mongodb.github.io/mongo-java-driver/3.2/driver-async/reference/) guide.

## 起步

### 安装指南

The recommended way to get started using one of the drivers in your project is with a dependency management system.

**IMPORTANT**

The MongoDB Async Driver requires either [Netty](http://netty.io/) or Java 7.

MavenGradle

#### MongoDB Async Driver

The new asynchronous API that can leverage either Netty or Java 7’s AsynchronousSocketChannel for fast and non-blocking IO.

<dependencies>

<dependency>

<groupId>org.mongodb</groupId>

<artifactId>mongodb-driver-async</artifactId>

<version>3.2.1</version>

</dependency>

</dependencies>

You can also download the jars [directly](https://oss.sonatype.org/content/repositories/releases/org/mongodb/mongodb-driver-async/3.2.1) from sonatype.

**Note:** mongodb-driver-async requires the following dependencies: [bson](https://oss.sonatype.org/content/repositories/releases/org/mongodb/bson/3.2.1) and [mongodb-driver-core](https://oss.sonatype.org/content/repositories/releases/org/mongodb/mongodb-driver-core/3.2.1)

### 使用入门

The following code snippets come from the QuickTour.java example code that can be found with the[async driver source](https://github.com/mongodb/mongo-java-driver/blob/master/driver-async/src/examples/tour/QuickTour.java).

**NOTE**

See the [installation guide](http://mongodb.github.io/mongo-java-driver/3.2/driver-async/getting-started/installation-guide/#mongodb-async-driver) for instructions on how to install the MongoDB Async Driver.

#### Going Async with Callbacks

The MongoDB Async driver provides an asynchronous API that can leverage either Netty or Java 7’sAsynchronousSocketChannel for fast and non-blocking IO.

The API mirrors the new Synchronous API from the MongoDB Driver, but any methods that cause network IO take a SingleResponseCallback<T> and return immediately, where T is the type of response for the document.

The SingleResponseCallback<T> interface requires the implementation of a single methodonResult(T result, Throwable t) which is called when the operation has completed. The resultparameter contains the result of the operation, if successful. If the operation failed for any reason then the tcontains the Throwable reason for the failure.

IMPORTANT

It’s important to always check for errors in any SingleResponseCallback<T> implementation and handle them appropriately! Below the error checks are left out only for the sake of brevity.

#### Make a Connection

The following example shows multiple ways to connect to the database mydb on the local machine, using the [MongoClients.create](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoClients.html#create-com.mongodb.ConnectionString-) helper.

*// To directly connect to the default server localhost on port 27017*

MongoClient mongoClient = MongoClients.create();

*// Use a Connection String*

MongoClient mongoClient = MongoClients.create(**"mongodb://localhost"**);

*// or a Connection String*

MongoClient mongoClient = MongoClients.create(**new** ConnectionString(**"mongodb://localhost"**));

*// or provide custom MongoClientSettings*

ClusterSettings clusterSettings = ClusterSettings.builder().hosts(asList(**new** ServerAddress(**"localhost"**))).build();

MongoClientSettings settings = MongoClientSettings.builder().clusterSettings(clusterSettings).build();

MongoClient mongoClient = MongoClients.create(settings);

MongoDatabase database = mongoClient.getDatabase(**"mydb"**);

At this point, the database object will be a connection to a MongoDB server for the specified database.

NOTE

There is no callback required for getDatabase("mydb") as there is no network IO required. AMongoDatabase instance provides methods to interact with a database but the database might not actually exist and will only be created on the insertion of data via some means; e.g. the creation of a collection or the insertion of documents which do require callbacks as they require network IO.

#### MongoClient

The MongoClient instance actually represents a pool of connections to the database; you will only need one instance of class MongoClient even with multiple concurrently executing asynchronous operations.

**IMPORTANT**

Typically you only create one MongoClient instance for a given database cluster and use it across your application. When creating multiple instances:

* All resource usage limits (max connections, etc) apply per MongoClient instance
* To dispose of an instance, make sure you call MongoClient.close() to clean up resources

#### Get a Collection

To get a collection to operate upon, specify the name of the collection to the[getCollection(String collectionName)](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoDatabase.html#getCollection-java.lang.String-) method:

The following example gets the collection test:

MongoCollection<Document> collection = database.getCollection(**"test"**);

#### Insert a Document

Once you have the collection object, you can insert documents into the collection. For example, consider the following JSON document; the document contains a field info which is an embedded document:

{

**"name"** : **"MongoDB"**,

**"type"** : **"database"**,

**"count"** : 1,

**"info"** : {

x : 203,

y : 102

}

}

To create the document using the Java driver, use the [Document](http://api.mongodb.org/java/3.2/?org/bson/Document.html) class. You can use this class to create the embedded document as well.

Document doc = **new** Document(**"name"**, **"MongoDB"**)

.append(**"type"**, **"database"**)

.append(**"count"**, 1)

.append(**"info"**, **new** Document(**"x"**, 203).append(**"y"**, 102));

To insert the document into the collection, use the insertOne() method.

collection.insertOne(doc, **new** SingleResultCallback<Void>() {

@Override

**public** **void** onResult(**final** Void result, **final** Throwable t) {

System.out.println(**"Inserted!"**);

}

});

As SingleResponseCallback<T> is a [functional interface](https://docs.oracle.com/javase/specs/jls/se8/html/jls-9.html#jls-9.8) and it can be implemented as a lambda for users on Java 8:

collection.insertOne(doc, (Void result, **final** Throwable t) -> System.out.println(**"Inserted!"**));

Once the document has been inserted the onResult callback will be called and it will print “Inserted!”. Remember, in a normal application you would always check for the presence of errors in the t variable.

#### Add Multiple Documents

To add multiple documents, you can use the insertMany() method.

The following example will add multiple documents of the form:

{ **"i"** : value }

Create the documents in a loop.

List<Document> documents = **new** ArrayList<Document>();

**for** (**int** i = 0; i < 100; i++) {

documents.add(**new** Document(**"i"**, i));

}

To insert these documents to the collection, pass the list of documents to the insertMany() method.

collection.insertMany(documents, **new** SingleResultCallback<Void>() {

@Override

**public** **void** onResult(**final** Void result, **final** Throwable t) {

System.out.println(**"Documents inserted!"**);

}

});

#### Count Documents in A Collection

Now that we’ve inserted 101 documents (the 100 we did in the loop, plus the first one), we can check to see if we have them all using the [count()](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoCollection#count--.html) method. The following code should print 101.

collection.count(

**new** SingleResultCallback<Long>() {

@Override

**public** **void** onResult(**final** Long count, **final** Throwable t) {

System.out.println(count);

}

});

#### Query the Collection

Use the [find()](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoCollection.html#find--) method to query the collection.

##### Find the First Document in a Collection

To get the first document in the collection, call the [first()](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoIterable.html#first--) method on the [find()](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoCollection.html#find--) operation.collection.find().first() returns the first document or null rather than a cursor. This is useful for queries that should only match a single document, or if you are interested in the first document only.

**NOTE**

Sometimes you will need the same or similar callbacks more than once. In these situations it makes sense to DRY (Do not Repeat Yourself) up your code and save the callback either as a concrete class or assign to a variable as below:

SingleResultCallback<Document> printDocument = **new** SingleResultCallback<Document>() {

@Override

**public** **void** onResult(**final** Document document, **final** Throwable t) {

System.out.println(document.toJson());

}

};

The following example passes the printDocument callback to the first method:

collection.find().first(printDocument);

The example will print the following document:

{ "**\_id**" : **{ "$oid" : "551582c558c7b4fbacf16735" }**,

"**name**" : **"MongoDB"**, "**type**" : **"database"**, "**count**" : 1,

"**info**" : **{ "x" :** 203**, "y" :** 102 **}** }

**NOTE**

The \_id element has been added automatically by MongoDB to your document and your value will differ from that shown. MongoDB reserves field names that start with “\_” and “$” for internal use.

##### Find All Documents in a Collection

To retrieve all the documents in the collection, we will use the find() method. The find() method returns a FindIterable instance that provides a fluent interface for chaining or controlling find operations. Use theforEach() method to provide a Block to apply to each document and a callback that is run once the iteration has finished. The following code retrieves all documents in the collection and prints them out (101 documents) and then finally prints out “Operation Finished!”:

Block<Document> printDocumentBlock = **new** Block<Document>() {

@Override

**public** **void** apply(**final** Document document) {

System.out.println(document.toJson());

}

};

SingleResultCallback<Void> callbackWhenFinished = **new** SingleResultCallback<Void>() {

@Override

**public** **void** onResult(**final** Void result, **final** Throwable t) {

System.out.println(**"Operation Finished!"**);

}

};

collection.find().forEach(printDocumentBlock, callbackWhenFinished);

#### Get A Single Document with a Query Filter

We can create a filter to pass to the find() method to get a subset of the documents in our collection. For example, if we wanted to find the document for which the value of the “i” field is 71, we would do the following (reusing the printDocument callback):

**import** **static** com.mongodb.client.model.Filters.\*;

collection.find(eq(**"i"**, 71)).first(printDocument);

will eventually print just one document:

{ "**\_id**" : **{ "$oid" : "5515836e58c7b4fbc756320b" }**, "**i**" : 71}

**NOTE**

Use the [Filters](http://mongodb.github.io/mongo-java-driver/3.2/builders/filters/), [Sorts](http://mongodb.github.io/mongo-java-driver/3.2/builders/sorts/), [Projections](http://mongodb.github.io/mongo-java-driver/3.2/builders/projections/) and [Updates](http://mongodb.github.io/mongo-java-driver/3.2/builders/updates/) helpers for simple and concise ways of building up queries.

#### Get a Set of Documents with a Query

We can use the query to get a set of documents from our collection. For example, if we wanted to get all documents where "i" > 50, we could write (reusing printDocumentBlock block andcallbackWhenFinished callback):

*// now use a range query to get a larger subset*

collection.find(gt(**"i"**, 50)).forEach(printDocumentBlock, callbackWhenFinished);

which should print the documents where i > 50.

We could also get a range, say 50 < i <= 100:

collection.find(and(gt(**"i"**, 50), lte(**"i"**, 100))).forEach(printDocumentBlock, callbackWhenFinished);

#### Sorting documents

We can also use the [Sorts](http://mongodb.github.io/mongo-java-driver/3.2/builders/sorts/) helpers to sort documents. We add a sort to a find query by calling the sort()method on a FindIterable. Below we use the [exists()](http://mongodb.github.io/mongo-java-driver/3.2/builders/filters/#elements) helper and sort [descending("i")](http://mongodb.github.io/mongo-java-driver/3.2/builders/sorts/#descending) helper to sort our documents:

collection.find(exists(**"i"**)).sort(descending(**"i"**)).first(printDocument);

#### Projecting fields

Sometimes we don’t need all the data contained in a document. The [Projections](http://mongodb.github.io/mongo-java-driver/3.2/builders/projections/) helpers can be used to build the projection parameter for the find operation and limit the fields returned.  
Below we’ll sort the collection, exclude the \_id field and output the first matching document:

collection.find().projection(excludeId()).first(printDocument);

#### Aggregations

Sometimes we need to aggregate the data stored in MongoDB. The [Aggregates](http://mongodb.github.io/mongo-java-driver/3.2/builders/aggregation/) helper provides builders for each of type of aggregation stage.

Below we’ll do a simple two step transformation that will calculate the value of i \* 10. First we find all Documents where i > 0 by using the [Aggregates.match](http://mongodb.github.io/mongo-java-driver/3.2/builders/aggregation/#match) helper. Then we reshape the document by using[Aggregates.project](http://mongodb.github.io/mongo-java-driver/3.2/builders/aggregation/#project) in conjunction with the [$multiply](http://docs.mongodb.org/manual/reference/operator/aggregation/multiply/) operator to calculate the “ITimes10” value:

collection.aggregate(asList(

match(gt(**"i"**, 0)),

project(Document.parse(**"{ITimes10: {$multiply: ['$i', 10]}}"**)))

).forEach(printDocumentBlock, callbackWhenFinished);

For [$group](http://mongodb.github.io/mongo-java-driver/3.2/builders/aggregation/#group) operations use the [Accumulators](http://api.mongodb.org/java/3.2/?com/mongodb/client/model/Accumulators.html) helper for any [accumulator operations](http://docs.mongodb.org/manual/reference/operator/aggregation/group/#accumulator-operator). Below we sum up all the values of i by using the [Aggregates.group](http://mongodb.github.io/mongo-java-driver/3.2/builders/aggregation/#group) helper in conjunction with the [Accumulators.sum](http://api.mongodb.org/java/3.2/?com/mongodb/client/model/Accumulators#sum-java.lang.String-TExpression-.html) helper:

collection.aggregate(singletonList(group(**null**, sum(**"total"**, **"$i"**)))).first(printDocument);

**NOTE**

Currently, there are no helpers for [aggregation expressions](http://docs.mongodb.org/manual/meta/aggregation-quick-reference/#aggregation-expressions). Use the [Document.parse()](http://mongodb.github.io/mongo-java-driver/3.2/bson/extended-json/) helper to quickly build aggregation expressions from extended JSON.

#### Updating documents

There are numerous [update operators](http://docs.mongodb.org/manual/reference/operator/update-field/) supported by MongoDB.

To update at most a single document (may be 0 if none match the filter), use the [updateOne](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoCollection.html#updateOne-org.bson.conversions.Bson-org.bson.conversions.Bson-) method to specify the filter and the update document. Here we use the [Updates.set](http://mongodb.github.io/mongo-java-driver/3.2/builders/updates/#set) helper to update the first document that meets the filter i equals 10 and set the value of i to 110:

collection.updateOne(eq(**"i"**, 10), set(**"i"**, 110),

**new** SingleResultCallback<UpdateResult>() {

@Override

**public** **void** onResult(**final** UpdateResult result, **final** Throwable t) {

System.out.println(result.getModifiedCount());

}

});

To update all documents matching the filter use the [updateMany](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoCollection.html#updateMany-org.bson.conversions.Bson-org.bson.conversions.Bson-) method. Here we use the [Updates.inc](http://mongodb.github.io/mongo-java-driver/3.2/builders/updates/#increment)helper to increment the value of i by 100 where i is less than 100.

collection.updateMany(lt(**"i"**, 100), inc(**"i"**, 100),

**new** SingleResultCallback<UpdateResult>() {

@Override

**public** **void** onResult(**final** UpdateResult result, **final** Throwable t) {

System.out.println(result.getModifiedCount());

}

});

The update methods return an [UpdateResult](http://api.mongodb.org/java/3.2/?com/mongodb/client/result/UpdateResult.html), which provides information about the operation including the number of documents modified by the update.

#### Deleting documents

To delete at most a single document (may be 0 if none match the filter) use the [deleteOne](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoCollection.html#deleteOne-org.bson.conversions.Bson-) method:

collection.deleteOne(eq(**"i"**, 110), **new** SingleResultCallback<DeleteResult>() {

@Override

**public** **void** onResult(**final** DeleteResult result, **final** Throwable t) {

System.out.println(result.getDeletedCount());

}

});

To delete all documents matching the filter use the [deleteMany](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoCollection.html#deleteMany-org.bson.conversions.Bson-) method. Here we delete all documents where i is greater or equal to 100:

collection.deleteMany(gte(**"i"**, 100), **new** SingleResultCallback<DeleteResult>() {

@Override

**public** **void** onResult(**final** DeleteResult result, **final** Throwable t) {

System.out.println(result.getDeletedCount());

}

});

The delete methods return a [DeleteResult](http://api.mongodb.org/java/3.2/?com/mongodb/client/result/DeleteResult.html), which provides information about the operation including the number of documents deleted.

#### Bulk operations

These commands allow for the execution of bulk insert/update/delete operations. There are two types of bulk operations:

1. Ordered bulk operations.

Executes all the operation in order and error out on the first write error.

1. Unordered bulk operations.

Executes all the operations and reports any the errors.

Unordered bulk operations do not guarantee order of execution.

Let’s look at two simple examples using ordered and unordered operations:

SingleResultCallback<BulkWriteResult> printBatchResult = **new** SingleResultCallback<BulkWriteResult>() {

@Override

**public** **void** onResult(**final** BulkWriteResult result, **final** Throwable t) {

System.out.println(result);

}

};

*// 2. Ordered bulk operation - order is guaranteed*

collection.bulkWrite(

Arrays.asList(**new** InsertOneModel<>(**new** Document(**"\_id"**, 4)),

**new** InsertOneModel<>(**new** Document(**"\_id"**, 5)),

**new** InsertOneModel<>(**new** Document(**"\_id"**, 6)),

**new** UpdateOneModel<>(**new** Document(**"\_id"**, 1),

**new** Document(**"$set"**, **new** Document(**"x"**, 2))),

**new** DeleteOneModel<>(**new** Document(**"\_id"**, 2)),

**new** ReplaceOneModel<>(**new** Document(**"\_id"**, 3),

**new** Document(**"\_id"**, 3).append(**"x"**, 4))),

printBatchResult

);

*// 2. Unordered bulk operation - no guarantee of order of operation*

collection.bulkWrite(

Arrays.asList(**new** InsertOneModel<>(**new** Document(**"\_id"**, 4)),

**new** InsertOneModel<>(**new** Document(**"\_id"**, 5)),

**new** InsertOneModel<>(**new** Document(**"\_id"**, 6)),

**new** UpdateOneModel<>(**new** Document(**"\_id"**, 1),

**new** Document(**"$set"**, **new** Document(**"x"**, 2))),

**new** DeleteOneModel<>(**new** Document(**"\_id"**, 2)),

**new** ReplaceOneModel<>(**new** Document(**"\_id"**, 3),

**new** Document(**"\_id"**, 3).append(**"x"**, 4))),

**new** BulkWriteOptions().ordered(**false**),

printBatchResult

);

**IMPORTANT**

Use of the bulkWrite methods is not recommended when connected to pre-2.6 MongoDB servers, as this was the first server version to support bulk write commands for insert, update, and delete in a way that allows the driver to implement the correct semantics for BulkWriteResult and BulkWriteException. The methods will still work for pre-2.6 servers, but performance will suffer, as each write operation has to be executed one at a time.

[Installation Guide](http://mongodb.github.io/mongo-java-driver/3.2/driver-async/getting-started/installation-guide/)

[Admin Quick Tour](http://mongodb.github.io/mongo-java-driver/3.2/driver-async/getting-started/quick-tour-admin/)

### 管理入门

This is the second part of the MongoDB driver quick tour. In the [quick tour](http://mongodb.github.io/mongo-java-driver/3.2/driver-async/getting-started/quick-tour/) we looked at how to use the Async Java driver to execute basic CRUD operations. In this section we’ll look at some of the administrative features available in the driver.

The following code snippets come from the QuickTourAdmin.java example code that can be found with the[driver source](https://github.com/mongodb/mongo-java-driver/blob/master/driver-async/src/examples/tour/QuickTourAdmin.java).

**NOTE**

See the [installation guide](http://mongodb.github.io/mongo-java-driver/3.2/driver-async/getting-started/installation-guide/) for instructions on how to install the MongoDB Driver.

#### Setup

To get started we’ll quickly connect and create a mongoClient, database and collection variable for use in the examples below:

MongoClient mongoClient = **new** MongoClient(**new** ConnectionString(**"mongodb://localhost"**));

MongoDatabase database = mongoClient.getDatabase(**"mydb"**);

MongoCollection<Document> collection = database.getCollection(**"test"**);

**NOTE**

Calling the getDatabase() on MongoClient does not create a database. Only when a database is written to will a database be created. Examples include the creation of an index or the insertion of a document into a previously non-existent collection.

**NOTE**

Sometimes you will need the same or similar callbacks more than once. In these situations it makes sense to DRY (Do not Repeat Yourself) up your code and save the callback either as a concrete class or assign to a variable as below:

SingleResultCallback<Void> callbackWhenFinished = **new** SingleResultCallback<Void>() {

@Override

**public** **void** onResult(**final** Void result, **final** Throwable t) {

System.out.println(**"Operation Finished!"**);

}

};

#### Get A List of Databases

You can get a list of the available databases:

mongoClient.listDatabaseNames().forEach(**new** Block<String>() {

@Override

**public** **void** apply(**final** String s) {

System.out.println(s);

}

}, callbackWhenFinished);

#### Drop A Database

You can drop a database by name using a MongoClient instance:

mongoClient.getDatabase(**"databaseToBeDropped"**).drop(callbackWhenFinished);

#### Create A Collection

Collections in MongoDB are created automatically simply by inserted a document into it. Using the[createCollection](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoDatabase.html#createCollection-java.lang.String-com.mongodb.async.SingleResultCallback-) method, you can also create a collection explicitly in order to customize its configuration. For example, to create a capped collection sized to 1 megabyte:

database.createCollection(**"cappedCollection"**,

**new** CreateCollectionOptions().capped(**true**).sizeInBytes(0x100000),

callbackWhenFinished);

#### Get A List of Collections

You can get a list of the available collections in a database:

database.listCollectionNames().forEach(**new** Block<String>() {

@Override

**public** **void** apply(**final** String databaseName) {

System.out.println(databaseName);

}

}, callbackWhenFinished);

#### Drop A Collection

You can drop a collection by using the drop() method:

collection.drop(callbackWhenFinished);

#### Create An Index

MongoDB supports secondary indexes. To create an index, you just specify the field or combination of fields, and for each field specify the direction of the index for that field. We can use the [Indexes](http://mongodb.github.io/mongo-java-driver/3.2/builders/indexes/) helpers to create index keys:

*// create an ascending index on the "i" field*

collection.createIndex(Indexes.ascending(**"i"**), callbackWhenFinished);

#### Get a List of Indexes on a Collection

Use the listIndexes() method to get a list of indexes. The following creates a printDocumentBlockBlock that prints out the Json version of a document and then passes that block to the forEach method on a [mongoIterable](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoIterable.html) so that it will printout all the indexes on the collection test:

Block<Document> printDocumentBlock = **new** Block<Document>() {

@Override

**public** **void** apply(**final** Document document) {

System.out.println(document.toJson());

}

};

collection.listIndexes().forEach(printDocumentBlock, callbackWhenFinished);

The example should print the following indexes:

{ "**v**" : 1, "**key**" : **{ "\_id" :** 1 **}**, "**name**" : **"\_id\_"**, "**ns**" : **"mydb.test"** }

{ "**v**" : 1, "**key**" : **{ "i" :** 1 **}**, "**name**" : **"i\_1"**, "**ns**" : **"mydb.test"** }

Operation Finished!

#### Text indexes

MongoDB also provides text indexes to support text search of string content. Text indexes can include any field whose value is a string or an array of string elements. To create a text index use the [Indexes.text](http://mongodb.github.io/mongo-java-driver/3.2/builders/indexes/#text-index)static helper:

*// create a text index on the "content" field*

coll.createIndex(Indexes.text(**"content"**), callbackWhenFinished);

As of MongoDB 2.6, text indexes are now integrated into the main query language and enabled by default (here we use the [Filters.text](http://mongodb.github.io/mongo-java-driver/3.2/builders/filters/#evaluation) helper):

*// Insert some documents*

collection.insertOne(**new** Document(**"\_id"**, 0).append(**"content"**, **"textual content"**), callbackWhenFinished);

collection.insertOne(**new** Document(**"\_id"**, 1).append(**"content"**, **"additional content"**), callbackWhenFinished);

collection.insertOne(**new** Document(**"\_id"**, 2).append(**"content"**, **"irrelevant content"**), callbackWhenFinished);

*// Find using the text index*

**long** matchCount = collection.count(text(**"textual content -irrelevant"**));

System.out.println(**"Text search matches: "** + matchCount);

*// Find using the $language operator*

Bson textSearch = Filters.text(**"textual content -irrelevant"**, **new** TextSearchOptions().language(**"english"**));

matchCount = collection.count(textSearch);

System.out.println(**"Text search matches (english): "** + matchCount);

*// Find the highest scoring match*

*// Find using the text index*

collection.count(text(**"textual content -irrelevant"**), **new** SingleResultCallback<Long>() {

@Override

**public** **void** onResult(**final** Long matchCount, **final** Throwable t) {

System.out.println(**"Text search matches: "** + matchCount);

}

});

*// Find using the $language operator*

Bson textSearch = text(**"textual content -irrelevant"**, **"english"**);

collection.count(textSearch, **new** SingleResultCallback<Long>() {

@Override

**public** **void** onResult(**final** Long matchCount, **final** Throwable t) {

System.out.println(**"Text search matches (english): "** + matchCount);

}

});

*// Find the highest scoring match*

Document projection = **new** Document(**"score"**, **new** Document(**"$meta"**, **"textScore"**));

collection.find(textSearch).projection(projection).first(**new** SingleResultCallback<Document>() {

@Override

**public** **void** onResult(**final** Document highest, **final** Throwable t) {

System.out.println(**"Highest scoring document: "** + highest.toJson());

}

});

and it should print:

Text search matches: 2

Text search matches (english): 2

Highest scoring document: { "**\_id**" : 1, "**content**" : **"additional content"**, "**score**" : 0.75}

For more information about text search see the [text index](http://docs.mongodb.org/manual/core/index-text) and [$text query operator](http://docs.mongodb.org/manual/reference/operator/query/text) documentation.

#### Running a command

While not all commands have a specific helper, however you can run any [command](http://docs.mongodb.org/manual/reference/command) by using the[runCommand()](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoDatabase.html#runCommand-org.bson.conversions.Bson-com.mongodb.ReadPreference-com.mongodb.async.SingleResultCallback-) method. Here we call the [buildInfo](http://mongodb.github.io/mongo-java-driver/3.2/driver-async/getting-started/quick-tour-admin/%7B%7B%20docsref%20%22reference/command/buildInfo%22%20%7D%7D) command:

database.runCommand(**new** Document(**"buildInfo"**, 1), **new** SingleResultCallback<Document>() {

@Override

**public** **void** onResult(**final** Document buildInfo, **final** Throwable t) {

System.out.println(buildInfo);

}

});

[Quick Tour](http://mongodb.github.io/mongo-java-driver/3.2/driver-async/getting-started/quick-tour/)

[Reference](http://mongodb.github.io/mongo-java-driver/3.2/driver-async/reference/)

## 进阶指南

The reference documentation for the Java driver focuses on high-level documentation and use-cases. The Javadoc-generated API reference is [here](http://api.mongodb.org/java/3.2/?).

* [Connecting](http://mongodb.github.io/mongo-java-driver/3.2/driver-async/reference/connecting/): Documentation of the driver’s support for connecting to MongoDB servers
* [CRUD](http://mongodb.github.io/mongo-java-driver/3.2/driver-async/reference/crud/): Documentation of the driver’s support for CRUD operations
* [Management](http://mongodb.github.io/mongo-java-driver/3.2/driver-async/reference/management/): Documentation of the driver’s support for logging and monitoring of its internal state

### Connecting

The reference documentation for connecting to a MongoDB server deployment is divided into three sections:

* [Connection Settings](http://mongodb.github.io/mongo-java-driver/3.2/driver-async/reference/connecting/connection-settings/): documentation of the various ways to specify the properties of a connection
* [Authenticating](http://mongodb.github.io/mongo-java-driver/3.2/driver-async/reference/connecting/authenticating/): detailed documentation of the various ways to specify authentication credentials
* [SSL](http://mongodb.github.io/mongo-java-driver/3.2/driver-async/reference/connecting/ssl/): Detailed documentation of the various ways to specify the properties of an SSL connection

#### Connection Settings

The Java driver has two ways of specifying the settings of a connection to a MongoDB server deployment.

##### Connection String

The [connection string](http://docs.mongodb.org/manual/reference/connection-string/) is the simplest way to specify the properties of a connection. . A connection string mostly follows [RFC 3986](http://tools.ietf.org/html/rfc3986), with the exception of the domain name. For MongoDB, it is possible to list multiple domain names separated by a comma. Below are some example connection strings.

* For a standalone mongod, mongos, or a direct connection to a member of a replica set:

mongodb://host:27017

* To connect to multiple mongos or a replica set:

mongodb://host1:27017,host2:27017

The [authentication guide](http://mongodb.github.io/mongo-java-driver/3.2/driver-async/reference/connecting/authenticating/) contains information on how to provide credentials in the connection string.

##### The Database Component

The database component is optional and is used to indicate which database to authenticate against. When the database component is not provided, the “admin” database is used.

mongodb://host:27017/mydb

Above, the database by the name of “mydb” is where the credentials are stored for the application.

**NOTE**

Some drivers utilize the database component to indicate which database to work with by default. The Java driver, while it parses the database component, does not use the database component for anything other than authentication.

##### Options

Many options can be provided via the connection string. The ones that cannot may be provided in a[MongoClientSettings](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoClientSettings.html) instance. To provide an option, append a ? to the connection string and separate options by an &.

mongodb://host:27017/?replicaSet=rs0&maxPoolSize=200

The above connection string sets the “replicaSet” value to “rs0” and the “maxPoolSize” to “200”.

For a comprehensive list of the available options, see the [ConnectionString](http://api.mongodb.org/java/3.2/?com/mongodb/ConnectionString.html) documentation.

##### MongoClient

A [MongoClient](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoClient.html) instance will be the root object for all interaction with MongoDB. It is all that is needed to handle connecting to servers, monitoring servers, and performing operations against those servers.

To create a MongoClient use the [MongoClients.create()](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoClients.html#create-com.mongodb.ConnectionString-) static helper. Without any argumentsMongoClients.create() will return a [MongoClient](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoClient.html) instance will connect to “localhost” port 27017.

MongoClient client = MongoClients.create();

Alternatively, a connection string may be provided:

MongoClient client = MongoClients.create(**new** ConnectionString(**"mongodb://host:27017,host2:27017/?replicaSet=rs0"**));

Finally, the [MongoClientSettings](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoClientSettings.html) class provides an in-code way to set the same options from a connection string. This is sometimes necessary, as the connection string does not allow an application to configure as many properties of the connection as MongoClientSettings.  
[MongoClientSettings](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoClientSettings.html) instances are immutable, so to create one an application uses a builder:

ClusterSettings clusterSettings = ClusterSettings.builder().hosts(asList(**new** ServerAddress(**"localhost"**))).description(**"Local Server"**).build();

MongoClientSettings settings = MongoClientSettings.builder().clusterSettings(clusterSettings).build();

MongoClient client = MongoClients.create(settings);

##### Netty Configuration

By default, the async driver relies on the [AsynchronousSocketChannel](http://docs.oracle.com/javase/7/docs/api/java/nio/channels/AsynchronousSocketChannel.html) class, introduced in Java 7. If configured properly, the driver will use [Netty](http://netty.io/) instead. An application must use Netty for the following reasons:

* The application is configured to use SSL to communicate with the MongoDB server.
* The application runs on Java 6.

To configure the driver to use Netty, the application must configure the MongoClientSettings appropriately:

MongoClientSettings.builder()

.streamFactoryFactory(**new** NettyStreamFactoryFactory())

.build();

By default the Netty-based streams will use the [NioEventLoopGroup](http://netty.io/4.0/api/io/netty/channel/nio/NioEventLoopGroup.html) and Netty’s [defaultByteBufAllocator](http://netty.io/4.0/api/io/netty/buffer/ByteBufAllocator.html#DEFAULT), but these are configurable via the [NettyStreamFactoryFactory](http://api.mongodb.org/java/3.2/?com/mongodb/connection/netty/NettyStreamFactoryFactory.html) constructor.

**NOTE**

Netty may also be configured by setting the org.mongodb.async.type system property to netty, but this should be considered as deprecated as of the 3.1 driver release.

#### SSL

The async Java driver supports SSL connections to MongoDB servers using the underlying support for SSL provided by Netty. You can configure the driver to use SSL with MongoClientSettings by setting the sslEnabled property to true and the stream factory to [NettyStreamFactoryFactory](http://api.mongodb.org/java/3.2/?com/mongodb/connection/netty/NettyStreamFactoryFactory.html), as in:

MongoClientSettings.builder()

.sslSettings(SslSettings.builder()

.enabled(**true**)

.build())

.streamFactoryFactory(**new** NettyStreamFactoryFactory())

.build()

See [Netty Configuration](http://mongodb.github.io/mongo-java-driver/3.2/driver-async/reference/connecting/connection-settings/#netty-configuration) for details on configuring Netty.

##### Host name verification

By default, the driver ensures that the host name included in the server’s SSL certificate(s) matches the host name(s) provided when constructing a MongoClient. However, this host name verification requires a Java 7 JVM, as it relies on additions to the javax.net.SSLParameters class that were introduced in Java 7. If your application must run on Java 6, or for some other reason you need to disable host name verification, you must expicitly indicate this in SslSettings using the invalidHostNameAllowed property:

MongoClientSettings.builder()

.sslSettings(SslSettings.builder()

.enabled(**true**)

.invalidHostNameAllowed(**true**)

.build())

.streamFactoryFactory(**new** NettyStreamFactoryFactory())

.build()

##### JVM system properties

A typical application will need to set several JVM system properties to ensure that the client is able to validate the SSL certificate presented by the server:

* javax.net.ssl.trustStore: the path to a trust store containing the certificate of the signing authority
* javax.net.ssl.trustStorePassword: the password to access this trust store

The trust store is typically created with the [keytool](http://docs.oracle.com/javase/8/docs/technotes/tools/unix/keytool.html) command line program provided as part of the JDK. For example:

keytool -importcert -trustcacerts -file <path to certificate authority file>

-keystore <path to trust store> -storepass <password>

A typical application will also need to set several JVM system properties to ensure that the client presents an SSL certificate to the MongoDB server:

* javax.net.ssl.keyStore: the path to a key store containing the client’s SSL certificates
* javax.net.ssl.keyStorePassword: the password to access this key store

The key store is typically created with the [keytool](http://docs.oracle.com/javase/8/docs/technotes/tools/unix/keytool.html) or the [openssl](https://www.openssl.org/docs/apps/openssl.html) command line program.

For more information on configuring a Java application for SSL, please refer to the  
[JSSE Reference Guide](http://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html).

#### Authentication

The Java driver supports all MongoDB [authentication mechanisms](http://docs.mongodb.org/manual/core/authentication/), including those only available in the MongoDB [Enterprise Edition](http://docs.mongodb.org/manual/administration/install-enterprise/).

An authentication credential is represented as an instance of the [MongoCredential](http://api.mongodb.org/java/3.2/?com/mongodb/MongoCredential.html) class, which includes static factory methods for each of the supported authentication mechanisms. A list of these instances must be passed to the driver via a [MongoClients](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoClients.html) static factory method that takes a [MongoClientSettings](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoClientSettings.html)parameter. Alternatively, a single MongoCredential can be created implicity via a [ConnectionString](http://api.mongodb.org/java/3.2/?com/mongodb/ConnectionString.html) and passed to a MongoClients static factory method that takes a ConnectionString parameter.

**NOTE**

Given the flexibility of role-based access control in MongoDB, it is usually sufficient to authenticate with a single user, but, for completeness, the driver accepts a list of credentials.

##### Default authentication mechanism

MongoDB 3.0 changed the default authentication mechanism from [MONGODB-CR](http://docs.mongodb.org/manual/core/authentication/#mongodb-cr-authentication) to [SCRAM-SHA-1](http://docs.mongodb.org/manual/core/authentication/#scram-sha-1-authentication). To create a credential that will authenticate properly regardless of server version, create a credential using the following static factory method:

**import** com.mongodb.MongoCredential;

*// ...*

String user; *// the user name*

String database; *// the name of the database in which the user is defined*

**char**[] password; *// the password as a character array*

*// ...*

MongoCredential credential = MongoCredential.createCredential(user,

database,

password);

or with a connection string:

ConnectionString uri = **new** ConnectionString(**"mongodb://user1:pwd1@host1/?authSource=db1"**);

This is the recommended approach as it will make upgrading from MongoDB 2.6 to MongoDB 3.0 seamless, even after [upgrading the authentication schema](http://docs.mongodb.org/manual/release-notes/3.0-scram/#upgrade-mongodb-cr-to-scram).

##### SCRAM-SHA-1

To explicitly create a credential of type [SCRAM-SHA-1](#scram-sha-1-authentication) use the following static factory method:

MongoCredential credential = MongoCredential.createScramSha1Credential(user,

database,

password);

or with a connection string:

ConnectionString uri = **new** ConnectionString(**"mongodb://user1:pwd1@host1/?authSource=db1&authMechanism=SCRAM-SHA-1"**);

##### MONGODB-CR

To explicitly create a credential of type [MONGODB-CR](http://docs.mongodb.org/manual/core/authentication/#mongodb-cr-authentication) use the following static factory method:

MongoCredential credential = MongoCredential.createMongoCRCredential(user,

database,

password);

or with a connection string:

ConnectionString uri = **new** ConnectionString(**"mongodb://user1:pwd1@host1/?authSource=db1&authMechanism=MONGODB-CR"**);

Note that this is not recommended as a credential created in this way will fail to authenticate after an authentication schema upgrade from MONGODB-CR to SCRAM-SHA-1.

##### x.509

The [x.509](http://docs.mongodb.org/manual/core/authentication/#x-509-certificate-authentication) mechanism authenticates a user whose name is derived from the distinguished subject name of the X.509 certificate presented by the driver during SSL negotiation. This authentication method requires the use of SSL connections with certificate validation and is available in MongoDB 2.6 and newer. To create a credential of this type use the following static factory method:

String user; *// The x.509 certificate derived user name, e.g. "CN=user,OU=OrgUnit,O=myOrg,..."*

MongoCredential credential = MongoCredential.createMongoX509Credential(user);

or with a connection string:

ConnectionString uri = **new** ConnectionString(**"mongodb://subjectName@host1/?authMechanism=MONGODB-X509"**);

See the MongoDB server [x.509 tutorial](http://docs.mongodb.org/manual/tutorial/configure-x509-client-authentication/#add-x-509-certificate-subject-as-a-user) for more information about determining the subject name from the certificate.

##### Kerberos (GSSAPI)

[MongoDB Enterprise](http://www.mongodb.com/products/mongodb-enterprise) supports proxy authentication through a Kerberos service. To create a credential of type [Kerberos (GSSAPI)](http://docs.mongodb.org/manual/core/authentication/#kerberos-authentication) use the following static factory method:

String user; *// The Kerberos user name, including the realm, e.g. "user1@MYREALM.ME"*

*// ...*

MongoCredential credential = MongoCredential.createGSSAPICredential(user);

or with a connection string:

ConnectionString uri = **new** ConnectionString(**"mongodb://username%40REALM.com@host1/?authMechanism=GSSAPI"**);

**NOTE**

The method refers to the GSSAPI authentication mechanism instead of Kerberos because technically the driver is authenticating via the [GSSAPI](https://tools.ietf.org/html/rfc4752) SASL mechanism.

To successfully authenticate via Kerberos, the application typically must specify several system properties so that the underlying GSSAPI Java libraries can acquire a Kerberos ticket:

java.security.krb5.realm=MYREALM.ME

java.security.krb5.kdc=mykdc.myrealm.me

**NOTE**

The GSSAPI authentication mechanism is supported only in the following environments:

* Linux: Java 6 and above
* Windows: Java 7 and above with [SSPI](https://msdn.microsoft.com/en-us/library/windows/desktop/aa380493)
* OS X: Java 7 and above

##### LDAP (PLAIN)

[MongoDB Enterprise](http://www.mongodb.com/products/mongodb-enterprise) supports proxy authentication through a Lightweight Directory Access Protocol (LDAP) service. To create a credential of type [LDAP](http://docs.mongodb.org/manual/core/authentication/#ldap-proxy-authority-authentication) use the following static factory method:

String user; *// The LDAP user name*

**char**[] password; *// The LDAP password*

*// ...*

MongoCredential credential = MongoCredential.createPlainCredential(user, **"$external"**, password);

or with a connection string:

ConnectionString uri = **new** ConnectionString(**"mongodb://user1@host1/?authSource=$external&authMechanism=PLAIN"**);

**NOTE**

The method refers to the plain authentication mechanism instead of LDAP because technically the driver is authenticating via the [PLAIN](https://www.ietf.org/rfc/rfc4616.txt) SASL mechanism.

### CRUD

For a walkthrough of the main CRUD operations please refer to the [Quick Tour](http://mongodb.github.io/mongo-java-driver/3.2/driver-async/getting-started/quick-tour/).

All CRUD-related methods in the Java driver are accessed through the [MongoCollection](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoCollection.html) interface. Instances of [MongoCollection](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoCollection.html) can be obtained from a  
[MongoClient](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoClient.html) instance by way of a [MongoDatabase](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoDatabase.html):

MongoClient client = MongoClients.create();

MongoDatabase database = client.getDatabase(**"mydb"**);

MongoCollection<Document> collection = database.getCollection(**"mycoll"**);

[MongoCollection](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/MongoCollection.html) is a generic interface: the TDocument type parameter is the class that clients use to insert or replace documents in a collection, and the default type returned from find and aggregate.

The single-argument getCollection method returns an instance of MongoCollection<Document>, and so with this type of collection an application uses instances of the Document class:

MongoCollection<Document> collection = database.getCollection(**"mycoll"**);

*// insert a document*

Document document = **new** Document(**"x"**, 1)

collection.insertOne(document, **new** SingleResultCallback<Void>() {

@Override

**public** **void** onResult(**final** Void result, **final** Throwable t) {

System.out.println(**"Inserted!"**);

}

});

...

document.append(**"x"**, 2).append(**"y"**, 3);

*// replace a document*

collection.replaceOne(Filters.eq(**"\_id"**, document.get(**"\_id"**), document,

**new** SingleResultCallback<UpdateResult>() {

@Override

**public** **void** onResult(**final** UpdateResult result, **final** Throwable t) {

System.out.println(result.getModifiedCount());

}

});

...

*// find documents*

collection.find().into(**new** ArrayList<Document>(),

**new** SingleResultCallback<List<Document>>() {

@Override

**public** **void** onResult(**final** List<Document> result, **final** Throwable t) {

System.out.println(**"Found Documents: #"** + result.size());

}

});

#### CodecRegistry

An overload of the getCollection method allows clients to specify a different class for representing BSON documents. For example, users my wish to use the type-safe [BsonDocument](http://api.mongodb.org/java/3.2/?org/bson/BsonDocument.html) with the CRUD API:

*// Pass BsonDocument.class as the second argument*

MongoCollection<BsonDocument> collection = database.getCollection(**"mycoll"**, BsonDocument.class);

*// insert a document*

BsonDocument document = **new** BsonDocument(**"x"**, **new** BsonInt32(1));

collection.insertOne(document, **new** SingleResultCallback<Void>() {

@Override

**public** **void** onResult(**final** Void result, **final** Throwable t) {

System.out.println(**"Inserted!"**);

}

});

...

document.append(**"x"**, **new** BsonInt32(2)).append(**"y"**, **new** BsonInt32(3));

*// replace a document*

collection.replaceOne(Filters.eq(**"\_id"**, document.get(**"\_id"**), document,

**new** SingleResultCallback<UpdateResult>() {

@Override

**public** **void** onResult(**final** UpdateResult result, **final** Throwable t) {

System.out.println(result.getModifiedCount());

}

});

...

*// find documents*

collection.find().into(**new** ArrayList<BsonDocument>(),

**new** SingleResultCallback<List<BsonDocument>>() {

@Override

**public** **void** onResult(**final** List<BsonDocument> result, **final** Throwable t) {

System.out.println(**"Found BsonDocuments: #"** + result.size());

}

});

There are two requirements that must be met for any class used in this way:

* a Codec for it must be registered in the MongoCollection’s CodecRegistry
* the Codec must be one that encodes and decodes a full BSON document (and not just, for example, a single BSON value like an Int32)

By default, a MongoCollection is configured with Codecs for two classes:

* Document
* BsonDocument

Applications, however, are free to register Codec implementations for other classes by customizing theCodecRegistry. New CodecRegistry instances are configurable at three levels:

* In a MongoClient via MongoClientOptions
* In a MongoDatabase via its withCodecRegistry method
* In a MongoCollection via its withCodecRegistry method

Consider the case of encoding and decoding instances of the UUID class. The Java driver by default encodes instances of UUID using a byte ordering that is not compatible with other MongoDB drivers, and changing the default would be quite dangerous. But it is possible for new applications that require interoperability across multiple drivers to be able to change that default, and they can do that with aCodecRegistry.

*// Replaces the default UuidCodec with one that uses the new standard UUID representation*

CodecRegistry codecRegistry =

CodecRegistries.fromRegistries(CodecRegistries.fromCodecs(**new** UuidCodec(UuidRepresentation.STANDARD)),

MongoClient.getDefaultCodecRegistry());

*// globally*

MongoClientSettings clientSettings = MongoClients.create(**"mongodb://localhost"**).ggetSettings();

newClientSettings = MongoClientSettings.builder(clientSettings).codecRegistry(codecRegistry).build();

MongoClient client = MongoClients.create(newClientSettings);

*// or per database*

MongoDatabase database = client.getDatabase(**"mydb"**)

.withCodecRegistry(codecRegistry);

*// or per collection*

MongoCollection<Document> collection = database.getCollection(**"mycoll"**)

.withCodecRegistry(codecRegistry);

#### Write Concern

Applications can configure the WriteConcern that a MongoCollection uses for write operations. LikeCodecRegistry, the WriteConcern can be configured at three levels:

* In a MongoClient via MongoClientOptions
* In a MongoDatabase via its withWriteConcern method
* In a MongoCollection via its withWriteConcern method

#### Read Preference

Applications can configure the ReadPreference that a MongoCollection uses for read operations. LikeWriteConcern, the ReadPreference can be configured at three levels:

* In a MongoClient via MongoClientOptions
* In a MongoDatabase via its withReadPreference method
* In a MongoCollection via its withReadPreference method

#### Immutability of MongoDatabase and MongoCollection

Instance of MongoDatabase and MongoCollection are immutable, so rather than mutate the state of theMongoCollection on which they are invoked, the three methods discussed above return new instances. Applications should therefore be sure to store the result of the method call. For example:

*// CORRECT: The results of the method calls are chained and the final one is referenced*

*// by collection*

MongoCollection<Document> collection = database.getCollection(**"mycoll"**)

.withWriteConcern(WriteConcern.JOURNALED)

.withReadPreference(ReadPreference.primary())

.withCodecRegistry(newRegistry);

*// INCORRECT: withReadPreference returns a new instance of MongoCollection*

*// It does not modify the collection it's called on. So this will*

*// have no effect*

collection.withReadPreference(ReadPreference.secondary());

### Management

The driver provides two mechanisms for examining its state:

* [Logging](http://mongodb.github.io/mongo-java-driver/3.2/driver-async/reference/management/logging/): Comprehensive logging of all operations using [SLF4J](http://www.slf4j.org/)
* [Monitoring](http://mongodb.github.io/mongo-java-driver/3.2/driver-async/reference/management/monitoring/): Built on [JMX](http://docs.oracle.com/javase/8/docs/technotes/guides/jmx/)

#### Logging

By default, logging is enabled via the popular [SLF4J](http://www.slf4j.org/) API. The use of [SLF4J](http://www.slf4j.org/) is optional; the driver will use SLF4J if the driver detects the presence of SLF4J in the classpath. Otherwise, the driver will fall back to JUL (java.util.logging).

The driver uses the following logger names:

* org.mongodb.driver: the root logger
  + cluster: for logs related to monitoring of the MongoDB servers to which the driver connects
  + connection: for logs related to connections and connection pools
  + protocol: for logs related to protocol message sent to and received from a MongoDB server
    - insert: for logs related to insert messages and responses
    - update: for logs related to update messages and responses
    - delete: for logs related to delete messages and responses
    - query: for logs related to query messages and responses
    - getmore: for logs related to getmore messages and responses
    - killcursor: for logs related to killcursor messages and responses
    - command: for logs related to command messages and responses
  + uri: for logs related to connection string parsing
  + management: for logs related to JMX

#### Monitoring

The driver uses [JMX](http://docs.oracle.com/javase/8/docs/technotes/guides/jmx/) to create [MXBeans](http://docs.oracle.com/javase/tutorial/jmx/mbeans/mxbeans.html) that allow an application or end user to monitor various aspects of the driver.

The driver creates MXBean instances of a single type: [ConnectionPoolStatisticsMBean](http://api.mongodb.org/java/3.2/?com/mongodb/management/ConnectionPoolStatisticsMBean.html). The driver registers one ConnectionPoolStatisticsMBean instance per each server it connects to. For example, in the case of a replica set, the driver creates an instance per each non-hidden member of the replica set.

Each MXBean instance is required to be registered with a unique object name, which consists of a domain and a set of named properties. All MXBean instances created by the driver are under the domain"org.mongodb.driver". Instances of ConnectionPoolStatisticsMBean will have the following properties:

* clusterId: a client-generated unique identifier, required to ensure object name uniqueness in situations where an application has multiple MongoClient instances connected to the same MongoDB server deployment
* host: the host name of the server
* port: the port on which the server is listening
* minSize: the minimum allowed size of the pool, including idle and in-use members
* maxSize: the maximum allowed size of the pool, including idle and in-use members
* size: the current size of the pool, including idle and and in-use members
* waitQueueSize: the current size of the wait queue for a connection from this pool
* checkedOutCount: the current count of connections that are currently in use

### Observables

The MongoDB Async Driver is fully callback based and makes extensive use of [SingleResultCallback<T>](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/SingleResultCallback.html)to achieve this. The SingleResultCallback<T> interface requires the implementation of a single methodonResult(T result, Throwable t) which is called once the operation has completed or errored. Theresult parameter contains the result of the operation if successful. If the operation failed for any reason then the t contains the Throwable reason for the failure. This pattern allows the users application logic to be deferred until the underlying network IO to MongoDB has been completed.

The callback pattern is extremely flexible but can quickly become unwieldy if the application logic requires a chain of operations. Nesting of callbacks can make code harder to read and give the appearance of making the codebase more complex that it actually is. To help with this we also have released two observable based asynchronous drivers:

1. [MongoDB Reactive Streams Driver](http://mongodb.github.io/mongo-java-driver-reactivestreams/)
2. [MongoDB RxJava Driver](http://mongodb.github.io/mongo-java-driver-rx/)

These observable drivers follow similar patterns that split the logic into onNext, onError andonComplete(d) methods. These methods split out the logic used bySingleResultCallback<T>.onResult(T result, Throwable t) into individual components that can make the code easier to reason with.

The MongoDB Async Driver provides a factory and interfaces that do the heavy lifting of converting callback based operations into an observable operations. There are three interfaces [Observable](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/Observable.html), [Subscription](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/Subscription.html)and [Observer](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/Observer.html). The [Observables](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/Observables.html) helpers convert any callback based operations into observable operations.

**NOTE**

The interfaces are similar to Publisher, Subscription and Subscriber interfaces from the [reactive streams](http://www.reactive-streams.org/) JVM implementation. However, we prefer the name Observerable to Publisher andObserver to Subscriber for readability purposes.

#### Observable

The [Observable](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/Observable.html) represents a MongoDB operation which emits its results to the Observer based on demand requested by the Subscription to the Observable.

#### Subscription

A [Subscription](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/Subscription.html) represents a one-to-one lifecycle of an Observer subscribing to an Observable. ASubscription to an Observable can only be used by a single Observer. The purpose of aSubscription is to control demand and to allow unsubscribing from the Observable.

#### Observer

An [Observer](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/Observer.html) provides the mechanism for receiving push-based notifications from the Observable. Demand for these events is signalled by its Subscription. On subscription to an Observable theObserver will be passed the Subscription via the onSubscribe(Subscription subscription). Demand for results is signaled via the Subscription and any results are passed to theonNext(TResult result) method. If there is an error for any reason the onError(Throwable e) will be called and no more events passed to the Observer. Alternatively, when the Observer has consumed all the results from the Observable the onComplete() method will be called.

#### Wrapping a MongoIterable

With the [Observables](http://api.mongodb.org/java/3.2/?com/mongodb/async/client/Observables.html) factory creating an Observable from a MongoIterable is simple.

In the following example we iterate and print out all json forms of documents in a collection:

Observables.observe(collection.find()).subscribe(**new** Observer<Document>(){

@Override

**void** onSubscribe(**final** Subscription subscription) {

System.out.println(**"Subscribed and requesting all documents"**);

subscription.request(Long.MAX\_VALUE);

}

@Override

**void** onNext(**final** Document document) {

System.out.println(document.toJson());

}

@Override

**void** onError(**final** Throwable e) {

System.out.println(**"There was an error: "** + e.getMessage());

}

@Override

**void** onComplete() {

System.out.println(**"Finished iterating all documents"**);

}

});

#### Wrapping a callback based method

Creating an Observable from any callback based methods requires the wrapping of the method inside a[Block](http://api.mongodb.org/java/3.2/?com/mongodb/Block.html). This allows the execution of the operation to be delayed, until demand is request by theSubscription. The method must use the supplied callback to convert the results into an Observable.

In the following example we print out the count of the number of documents in a collection:

Block<SingleResultCallback<Long>> operation = **new** Block<SingleResultCallback<Long>>() {

@Override

**void** apply(**final** SingleResultCallback<Long> callback) {

collection.count(callback);

}

};

*// Or in Java 8 syntax:*

operation = (Block<SingleResultCallback<Long>>) collection::count;

Observables.observe(operation).subscribe(**new** Observer<Long>(){

@Override

**void** onSubscribe(**final** Subscription subscription) {

System.out.println(**"Subscribed and requesting the count"**);

subscription.request(1);

}

@Override

**void** onNext(**final** Long count) {

System.out.println(**"The collection has "** + count + **" documents"**);

}

@Override

**void** onError(**final** Throwable e) {

System.out.println(**"There was an error: "** + e.getMessage());

}

@Override

**void** onComplete() {

System.out.println(**"Finished"**);

}

});

#### Back Pressure

In the following example, the Subscription is used to control demand when iterating an Observable. This is similar in concept to the MongoIterable.forEach method but allows demand-driven iteration:

Observables.observe(collection.find()).subscribe(**new** Observer<Document>(){

**private** **long** batchSize = 10;

**private** **long** seen = 0;

**private** Subscription subscription;

@Override

**void** onSubscribe(**final** Subscription subscription) {

**this**.subscription = subscription;

subscription.request(batchSize);

}

@Override

**void** onNext(**final** Document document) {

System.out.println(document.toJson());

seen += 1;

**if** (seen == batchSize) {

seen = 0;

subscription.request(batchSize);

}

}

@Override

**void** onError(**final** Throwable e) {

System.out.println(**"There was an error: "** + e.getMessage());

}

@Override

**void** onComplete() {

System.out.println(**"Finished iterating all documents"**);

}

});

# BSON

The BSON library comprehensively supports [BSON](http://www.bsonspec.org/), the data storage and network transfer format that MongoDB uses for “documents”. BSON, short for Binary [JSON](http://json.org/), is a binary-encoded serialization of JSON-like documents.

BSONJ的Java库提供了对BSON格式的全面性支持，包括MongoDB用于”文档”操作的数据存储和网络传输格式的支持。BSON是二进制JSON的简称，它是一个被二进制编码的类JSON的文档。

* [Documents](http://mongodb.github.io/mongo-java-driver/3.2/bson/documents/): Documentation of the driver’s support for BSON document representations
* [Readers and Writers](http://mongodb.github.io/mongo-java-driver/3.2/bson/readers-and-writers/): Documentation of the driver’s support for stream-based reading and writing of BSON documents
* [Codec and CodecRegistry](http://mongodb.github.io/mongo-java-driver/3.2/bson/codecs/): Documentation of the driver’s Codec API, an abstraction for producing and consuming BSON document representations using the stream-based readers and writers
* [Extended JSON](http://mongodb.github.io/mongo-java-driver/3.2/bson/extended-json/): Documentation of the driver’s support for MongoDB Extended JSON

## Documents

The driver includes several classes and interfaces used for representing BSON documents.

Java驱动包提供了几个类和接口，用于映射和代表BSON文档。

### BsonDocument

Although generally not needed by users of the high-level driver API, the [BsonDocument](http://api.mongodb.org/java/3.2/?org/bson/BsonDocument.html) class is central to the way that documents are managed internally by the driver. The BsonDocument class can represent dynamically structured documents of any complexity with a type-safe API. For instance, the document

{

**"a"** : **"MongoDB"**,

**"b"** : [ 1, 2 ]

}

can be constructed as a BsonDocument as follows:

**new** BsonDocument().append(**"a"**, **new** BsonString(**"MongoDB"**))

.append(**"b"**, **new** BsonArray(Arrays.asList(**new** BsonInt32(1), **new** BsonInt32(2))));

The type safety comes from BsonDocument implementing Map<String, BsonValue>, so even built-in types like int, String and List must be wrapped in a sub-class of BsonValue. For a complete list ofBsonValue sub-types, please consult the [BsonValue](http://api.mongodb.org/java/3.2/?org/bson/BsonValue.html) API documentation.

### Document-推荐

Most applications will use the [Document](http://api.mongodb.org/java/3.2/?org/bson/Document.html) class instead. Like BsonDocument, the Document class can represent dynamically structured documents of any complexity; however, the typing is much looser, asDocument implements Map<String, Object>. As a result, the same document as above can be constructed using the Document class as follows:

**new** Document().append(**"a"**, **"MongoDB"**)

.append(**"b"**, Arrays.asList(1, 2));

There is less code to write, but runtime errors are possible if you inadvertently add an instance of an unsupported value type.

The most commonly used value types are:

| **BSON type** | **Java type** |
| --- | --- |
| Document | org.bson.Document |
| Array | java.util.List |
| Date | java.util.Date |
| Boolean | java.lang.Boolean |
| Double | java.lang.Double |
| Int32 | java.lang.Integer |
| Int64 | java.lang.Long |
| String | java.lang.String |
| Binary | org.bson.types.Binary |
| ObjectId | org.bson.types.ObjectId |
| Null | null |

It is actually possible to change these mappings; the mechanism for doing so is covered [later](http://mongodb.github.io/mongo-java-driver/3.2/bson/codecs/) in this reference .

### DBObject-旧API

Although not recommended for new applications, those upgrading from the 2.x driver series may continue to use the [DBObject](http://api.mongodb.org/java/3.2/?com/mongodb/DBObject.html) interface to represent BSON documents. DBObject is similar to Document in that it represents BSON values as Object, but it has a few shortcomings that were impossible to overcome:

* it is an interface rather than a class, so it’s API can not be extended without breaking binary compatibility
* it doesn’t actually implement Map<String, Object>
* because it is an interface, a separate concrete class called [BasicDBObject](http://api.mongodb.org/java/3.2/?com/mongodb/BasicDBObject.html) which implements that interface, is required

### Bson-扩展时使用

To tie these all together, the driver contains a small but powerful interface called [Bson](http://api.mongodb.org/java/3.2/?org/bson/conversions/Bson.html). Any class that represents a BSON document, whether included in the driver itself or from a third party, can implement this interface and can then be used any place in the high-level API where a BSON document is required. The three classes discussed above all implement this interface and so can be used interchangeably based on the needs of a given application. For example:

collection.find(**new** BsonDocument(**"x"**, **new** BsonInt32(1)));

collection.find(**new** Document(**"x"**, 1));

collection.find(**new** BasicDBObject(**"x"**, 1));

## BsonWriter and BsonReader

The various implementations of the Bson interface discussed in the previous section all represent BSON documents using an underlying Java Map instance. However, they are not directly responsible for reading and writing their representations from and to BSON. Instead, this process is delegated to [BsonWriter](http://api.mongodb.org/java/3.2/?org/bson/BsonWriter.html) and[BsonReader](http://api.mongodb.org/java/3.2/?org/bson/BsonReader.html), abstract classes that expose methods for iterative, stream-based processing of BSON documents.

### BsonWriter

The BsonWriter class exposes methods for writing a BSON document. Consider the task of writing the document

{

**"a"** : **"MongoDB"**,

**"b"** : [

{ **"c"**: 1 }

]

}

The following code will stream a document of this structure to a BsonWriter:

BsonWriter writer = ... *// Construct a BsonWriter*

writer.writeStartDocument();

writer.writeName(**"a"**);

writer.writeString(**"MongoDB"**);

writer.writeName(**"b"**);

writer.writeStartArray();

writer.writeStartDocument();

writer.writeName(**"c"**);

writer.writeInt32(1);

writer.writeEndDocument();

writer.writeEndArray();

writer.writeEndDocument();

The indentation is not necessary: it’s just to clarify that the stream of events written to the BsonWriter, although written iteratively, have an implicit hierarchical structure. The BsonWriter validates that the events create only properly structured BSON documents. Otherwise, it throws a BsonSerializationException.

The two most important classes to extend BsonWriter are [BsonBinaryWriter](http://api.mongodb.org/java/3.2/?org/bson/BsonBinaryWriter.html) and [JsonWriter](http://api.mongodb.org/java/3.2/?org/bson/json/JsonWriter.html).BsonBinaryWriter writes the BSON document as a stream of bytes in accordance with the [BSON](http://www.bsonspec.org/)specification, while JsonWriter writes the BSON document as a stream of characters in accordance with[MongoDB Extended JSON](http://docs.mongodb.org/manual/reference/mongodb-extended-json/).

### BsonReader

The BsonReader class exposes methods for reading a BSON document. Consider the task of reading the document written above with a BsonReader:

BsonReader reader = ... *// Construct a BsonReader*

reader.readStartDocument();

reader.readName(); *// read the name "a"*

reader.readString(); *// read string "MongoDB"*

reader.readName(); *// read the name "b"*

reader.readStartArray();

reader.readStartDocument();

reader.readName(); *// read the name "c"*

reader.readInt32(); *// read the integer 1*

reader.readEndDocument();

reader.readEndArray();

reader.readEndDocument();

As with the writer example, the indentation is not necessary: it’s just to clarify that the stream of events read from the BsonWriter, although written iteratively, have an implicit hierarchical structure. The BsonReader will throw a BsonSerializationException if the events read do not match the structure of the document that is being read from.

In most situations an application will not know the exact structure of the document being read. For that reason, BsonReader exposes a few methods that allow an application to peak ahead so that it can figure out what method to call next. Consider a situation where an application must read a BSON document with an unknown structure:

reader.readStartDocument();

**while** (reader.readBsonType() != BsonType.END\_OF\_DOCUMENT) {

String fieldName = reader.readName();

**switch** (reader.getCurrentBsonType()) {

**case** INT32:

**int** intValue = reader.readInt32();

**break**;

**case** INT64:

**long** longValue = reader.readInt64();

**break**;

*// ... handle each supported field type*

}

}

reader.readEndDocument();

In this example, the application iterates through the fields of the document until it reachesEND\_OF\_DOCUMENT. For each field, it reads the name and then the value based on the BsonType of the field.

A similar pattern can be used to read a BSON array:

reader.readStartArray();

**while** (reader.readBsonType() != BsonType.END\_OF\_DOCUMENT) {

**switch** (reader.getCurrentBsonType()) {

**case** INT32:

**int** intValue = reader.readInt32();

**break**;

**case** INT64:

**long** longValue = reader.readInt64();

**break**;

*// ... handle each supported field type*

}

}

reader.readEndArray();

The only significant difference between reading an array and reading a document is that, since the elements of an array do not have names, there is no field name to read, only a series of values.

The two most important classes to extend BsonReader are [BsonBinaryReader](http://api.mongodb.org/java/3.2/?org/bson/BsonBinaryReader.html) and [JsonReader](http://api.mongodb.org/java/3.2/?org/bson/json/JsonReader.html).BsonBinaryReader reads the BSON document as a stream of bytes in accordance with the [BSON](http://www.bsonspec.org/)specification, while JsonReader reads the BSON document as a stream of characters in accordance with[MongoDB Extended JSON](http://docs.mongodb.org/manual/reference/mongodb-extended-json/).

## Codec and CodecRegistry

In the last section we saw how to use the [BsonReader](http://api.mongodb.org/java/3.2/?org/bson/BsonReader.html" \t "_blank) and [BsonWriter](http://api.mongodb.org/java/3.2/?org/bson/BsonWriter.html" \t "_blank) API to read and write BSON documents. But writing code at that low a level is tedious and error-prone, so in practice these algorithms are packaged in implementations of the [Codec](http://api.mongodb.org/java/3.2/?org/bson/codecs/Codec.html) interface.

在上面的部分中我们了解到如何使用[BsonReader](http://api.mongodb.org/java/3.2/?org/bson/BsonReader.html) 和[BsonWriter](http://api.mongodb.org/java/3.2/?org/bson/BsonWriter.html) 的API进行BSON文档的读和写。但是在那种低级别封装的情况下进行编码会非常的枯燥并且极易出错，所以在实践中这些原始的语法是被打包在BSON编解码器接口的内部实现中使用的。

### Codec

The Codec interface abstracts the processes of decoding a BSON value into a Java object using a BsonReader and encoding a Java object into a BSON value using a BsonWriter. The BSON value can be as simple as a boolean or as complex as a document or array.

Let’s look at a simple Codec implementation that encodes a Java Integer to a BSON Int32, and vice versa:

**public** **class** IntegerCodec **implements** Codec<Integer> {

@Override

**public** **void** encode(**final** BsonWriter writer, **final** Integer value, **final** EncoderContext encoderContext) {

writer.writeInt32(value);

}

@Override

**public** Integer decode(**final** BsonReader reader, **final** DecoderContext decoderContext) {

**return** reader.readInt32();

}

@Override

**public** Class<Integer> getEncoderClass() {

**return** Integer.class;

}

}

The encode method takes a BsonWriter and an Integer and calls the writeInt32 method on theBsonWriter with the value of the Integer, while the decode method takes a BsonReader and calls thereadInt32 method on the BsonReader, returning the value as an Integer.

A Codec implementation than encodes to and decodes from a BSON document or array is more complicated, and would typically rely on a set of simpler Codec implementations for the basic BSON value types. For this, it can rely on a CodecRegistry.

### CodecRegistry

A [CodecRegistry](http://api.mongodb.org/java/3.2/?org/bson/codecs/configuration/CodecRegistry.html) contains a set of Codec instances that are accessed according to the Java classes that they encode from and decode to. Instances of CodecRegistry are generally created via static factory methods on the [CodecRegistries](http://api.mongodb.org/java/3.2/?org/bson/codecs/configuration/CodecRegistries.html) class. Consider the simplest of these methods, one that takes a list ofCodecs:

CodecRegistry registry = CodecRegistries.fromCodecs(**new** IntegerCodec(), **new** LongCodec(), ...);

This returns an immutable CodecRegistry instance containing all the Codec instances passed to the fromCodecs method. They can be accessed like this:

Codec<Integer> integerCodec = codecRegistry.get(Integer.class);

Codec<Long> longCodec = codecRegistry.get(Long.class);

Now consider a Codec for the Document class. This Codec implementation, in order to decode and encode the values for each field in the document, must be constructed with a CodecRegistry to look up the Codec instances for each type of value. But how could one construct an instance of that Codec? You would have to pass an instance to the CodecRegistries.fromCodecs method, but you don’t have aCodecRegistry yet to pass to the constructor. You need some way to delay the construction of the Document Codec until after the CodecRegistry has been constructed. For that we use a CodecProvider.

### CodecProvider

A [CodecProvider](http://api.mongodb.org/java/3.2/?org/bson/codecs/configuration/CodecProvider.html) is a factory for Codec instances. Unlike CodecRegistry, its get method takes not only a Class, but also a CodecRegistry, allowing a CodecProvider implementation to construct Codecinstances that require a CodecRegistry to look up Codec instances for the values contained within it. Consider a CodecProvider for the Document class:

**public** **class** DocumentCodecProvider **implements** CodecProvider {

@Override

**public** <T> Codec<T> get(**final** Class<T> clazz, **final** CodecRegistry registry) {

**if** (clazz == Document.class) {

*// construct DocumentCodec with a CodecRegistry*

**return** (Codec<T>) **new** DocumentCodec(registry);

}

*// CodecProvider returns null if it's not a provider for the requresed Class*

**return** **null**;

}

}

The DocumentCodec, because it is constructed with a CodecRegistry, can now use that registry to look upCodec instances for the values contained in each Document that it encodes.

One more problem remains, however. Consider the problem of encoding values to a BSON DateTime. An application may want to encode to a BSON DateTime instances of both the original Java Date class as well as the Java 8 Instant class. It’s easy to create implemenations of Codec<Date> andCodec<Instant>, and either one can be used for encoding. But when decoding, a Document Codec also has to choose which Java type to decode a BSON DateTime to. Rather than hard-coding it in theDocumentCodec, the decision is abstracted via the BsonTypeClassMap class.

### BsonTypeClassMap

The [BsonTypeClassMap](http://api.mongodb.org/java/3.2/?org/bson/codecs/BsonTypeClassMap.html) class simply maps each value in the BsonType enumeration to a Java class. It contains a sensible set of default mappings that can easily be changed by passing an aMap<BsonType, Class<?>> instance to the constructor with any replacement mappings to apply. Consider the case where an application wants to decode all BSON DateTime values to a Java 8 Instant instead of the default Date:

Map<BsonType, Class<?>> replacements = **new** HashMap<BsonType, Class<?>>();

replacements.put(BsonType.DATE\_TIME, Instant.class);

BsonTypeClassMap bsonTypeClassMap = **new** BsonTypeClassMap(replacements);

This will replace the default mapping of BSON DateTime to Date to one from BSON DateTime toInstant.

Putting it all together, we can added a BsonTypeClassMap to the DocumentCodecProvider shown above:

**public** **class** DocumentCodecProvider **implements** CodecProvider {

**private** **final** BsonTypeClassMap bsonTypeClassMap;

**public** DocumentCodecProvider(**final** BsonTypeClassMap bsonTypeClassMap) {

**this**.bsonTypeClassMap = bsonTypeClassMap;

}

@Override

**public** <T> Codec<T> get(**final** Class<T> clazz, **final** CodecRegistry registry) {

**if** (clazz == Document.class) {

*// construct DocumentCodec with a CodecRegistry and a BsonTypeClassMap*

**return** (Codec<T>) **new** DocumentCodec(registry, bsonTypeClassMap);

}

**return** **null**;

}

}

The DocumentCodec, because it is constructed with both a BsonTypeClassMap and a CodecRegistry, can first use the BsonTypeClassMap to determine with type to decode each BSON value to, then use theCodecRegistry to look up the Codec for that Java type.

Finally, we create a CodecRegistry instance

CodecRegistry defaultCodecRegistry = ...

DocumentCodecProvider documentCodecProvider = ...

Codec<Instant> instantCodec = ...

codecRegistry = CodecRegistries.fromRegistries(CodecRegistries.fromCodecs(instantCodec),

CodecRegistries.fromProviders(documentCodecProvider),

defaultCodecRegistry);

using two additional static factory methods from the CodecRegistries class: one that takes a list ofCodecProviders and one which takes a list of CodecRegistrys.

## MongoDB Extended JSON

As discussed earlier, the Java driver supports reading and writing BSON documents represented as  
[MongoDB Extended JSON](http://docs.mongodb.org/manual/reference/mongodb-extended-json/). Both variants are supported:

* Strict Mode: representations of BSON types that conform to the [JSON RFC](http://www.json.org/). This is the format that[mongoexport](http://docs.mongodb.org/manual/reference/program/mongoexport/) produces and [mongoimport](http://docs.mongodb.org/manual/reference/program/mongoimport/) consumes.
* Shell Mode: a superset of JSON that the [MongoDB shell](http://docs.mongodb.org/manual/tutorial/getting-started-with-the-mongo-shell/) can parse.

Furthermore, the Document class provides two sets of convenience methods for this purpose:

* toJson(): a set of overloaded methods that convert a Document instance to a JSON string
* parse(): a set of overloaded static factory methods that convert a JSON string to a Document instance

### Writing JSON

Consider the task of implementing a [mongoexport](http://docs.mongodb.org/manual/reference/program/mongoexport/)-like tool using the Java driver.

String outputFilename; *// initialize to the path of the file to write to*

MongoCollection<Document> collection; *// initialize to the collection from which you want to query*

BufferedWriter writer = **new** BufferedWriter(**new** FileWriter(outputFilename));

**try** {

**for** (Document doc : collection.find()) {

writer.write(doc.toJson());

writer.newLine();

} **finally** {

writer.close();

}

The Document.toJson() method constructs an instance of a JsonWriter with its default settings, which will write in strict mode with no new lines or indentation.

You can override this default behavior by using one of the overloads of toJson(). As an example, consider the task of writing a JSON string that can be copied and pasted into the MongoDB shell:

SimpleDateFormat fmt = **new** SimpleDateFormat(**"dd/MM/yy"**);

Date first = fmt.parse(**"01/01/2014"**);

Date second = fmt.parse(**"01/01/2015"**);

Document doc = **new** Document(**"startDate"**, **new** Document(**"$gt"**, first).append(**"$lt"**, second));

System.out.println(doc.toJson(**new** JsonWriterSettings(JsonMode.SHELL)));

This code snippet will print out MongoDB shell-compatible JSON, which can then be pasted into the shell:

{ **"startDate"** : { **"$gt"** : ISODate(**"2014-01-01T05:00:00.000Z"**), **"$lt"** : ISODate(**"2015-01-01T05:00:00.000Z"**) } }

### Reading JSON

Consider the task of implementing a [mongoimport](http://docs.mongodb.org/manual/reference/program/mongoimport/)-like tool using the Java driver.

String inputFilename; *// initialize to the path of the file to read from*

MongoCollection<Document> collection; *// initialize to the collection to which you want to write*

BufferedReader reader = **new** BufferedReader(**new** FileReader(inputFilename));

**try** {

String json;

**while** ((json = reader.readLine()) != **null**) {

collection.insertOne(Document.parse(json));

}

} **finally** {

reader.close();

}

The Document.parse() static factory method constructs an instance of a JsonReader with the given string and returns an instance of an equivalent Document instance. JsonReader automatically detects the JSON flavor in the string, so you do not need to specify it.

## Installation

The BSON library is a required dependency of all the MongoDB Java drivers and if using a dependency management system, it will be automatically installed alongside the driver, however, it can be used as a standalone library. The recommended way to get started using one of the drivers in your project is with a dependency management system.

MavenGradle

### BSON

This library comprehensively supports [BSON](http://www.bsonspec.org/), the data storage and network transfer format that MongoDB uses for “documents”. BSON is short for Binary [JSON](http://json.org/), is a binary-encoded serialization of JSON-like documents.

<dependencies>

<dependency>

<groupId>org.mongodb</groupId>

<artifactId>bson</artifactId>

<version>3.2.1</version>

</dependency>

</dependencies>

You can also download the jars [directly](https://oss.sonatype.org/content/repositories/releases/org/mongodb/bson/3.2.1) from sonatype.

# Builders

The driver provides several classes that make it easier to use the CRUD API.

* [Filters](http://mongodb.github.io/mongo-java-driver/3.2/builders/filters/): Documentation of the driver’s support for building query filters
* [Projections](http://mongodb.github.io/mongo-java-driver/3.2/builders/projections/): Documentation of the driver’s support for building projections
* [Sorts](http://mongodb.github.io/mongo-java-driver/3.2/builders/sorts/): Documentation of the driver’s support for building sort criteria
* [Aggregation](http://mongodb.github.io/mongo-java-driver/3.2/builders/aggregation/): Documentation of the driver’s support for building aggregation pipelines
* [Updates](http://mongodb.github.io/mongo-java-driver/3.2/builders/updates/): Documentation of the driver’s support for building updates
* [Indexes](http://mongodb.github.io/mongo-java-driver/3.2/builders/indexes/): Documentation of the driver’s support for creating index keys

## Filters

The [Filters](http://api.mongodb.org/java/3.2/?com/mongodb/client/model/Filters.html) class provides static factory methods for all the MongoDB query operators. Each method returns an instance of the [Bson](http://mongodb.github.io/mongo-java-driver/3.2/bson/documents/#bson) type, which can in turn be passed to any method that expects a query filter.

For brevity, you may choose to import the methods of the Filters class statically:

**import** com.mongodb.client.model.Filters.\*;

All the examples below assume this static import.

### Comparison

The comparison operator methods include:

* eq: Matches values that are equal to a specified value.
* gt: Matches values that are greater than a specified value.
* gte: Matches values that are greater than or equal to a specified value.
* lt: Matches values that are less than a specified value.
* lte: Matches values that are less than or equal to a specified value.
* ne: Matches all values that are not equal to a specified value.
* in: Matches any of the values specified in an array.
* nin: Matches none of the values specified in an array.

**Examples**

This example creates a filter that selects all documents where the value of the qty field equals 20:

eq(**"qty"**, 20)

which will render as:

{

"**qty**" : 20

}

This example creates a filter that selects all documents where the value of the qty field is either 5 or 20:

in(**"qty"**, 5, 15)

### Logical

The logical operator methods include:

* and: Joins filters with a logical AND and selects all documents that match the conditions of both filters.
* or: Joins filters with a logical OR and selects all documents that match the conditions of either filters.
* not: Inverts the effect of a query expression and selects documents that do not match the filter.
* nor: Joins filters with a logical NOR and selects all documents that fail to match both filters.

Examples

This example creates a filter that selects all documents where ther value of the qty field is greater than 20and the value of the user field equals "jdoe":

and(gt(**"qty"**, 20), eq(**"user"**, **"jdoe"**))

The and method generates a $and operator only if necessary, as the query language implicity ands together all the elements in a filter. So the above example will render as:

{

"**qty**" : **{ "$gt" :** 20 **}**,

"**user**" : **"jdoe"**

}

This example creates a filter that selects all documents where the price field value equals 0.99 or 1.99; and the sale field value is equal to true or the qty field value is less than 20:

and(or(eq(**"price"**, 0.99), eq(**"price"**, 1.99)

or(eq(**"sale"**, **true**), lt(**"qty"**, 20)))

This query cannot be constructed using an implicit and operation, because it uses the $or operator more than once. So it will render as:

{

"**$and**" :

**[**

**{ "$or" : [ { "price" :** 0.99 **}, { "price" :** 1.99 **} ] },**

**{ "$or" : [ { "sale" : true }, { "qty" : { "$lt" :** 20 **} } ] }**

**]**

}

### Arrays

The array operator methods include:

* all: Matches arrays that contain all elements specified in the query
* elemMatch: Selects documents if element in the array field matches all the specified $elemMatch conditions
* size: Selects documents if the array field is a specified size

Examples

This example selects documents with a tags array containing both "ssl" and "security":

all(**"tags"**, Arrays.asList(**"ssl"**, **"security"**))

### Elements

The elements operator methods include:

* exists: Selects documents that have the specified field.
* type: Selects documents if a field is of the specified type.

Examples

This example selects documents that have a qty field and its value does not equal 5 or 15:

and(exists(**"qty"**), nin(**"qty"**, 5, 15))

This example selects documents that have a qty field with the type of BsonInt32:

type(**"qty"**, BsonType.INT32)

Available with MongoDB 3.2, this example selects any documents that have a qty field with any “number” bson type:

type(**"qty"**, **"number"**)

### Evaluation

The evaluation operator methods include:

* mod: Performs a modulo operation on the value of a field and selects documents with a specified result.
* regex: Selects documents where values match a specified regular expression.
* text: Selects documemts matching a full-text search expression.
* where: Matches documents that satisfy a JavaScript expression.

Examples

This example assumes a collection that has a text index in the field abstract. It selects documents that have a abstract field containing the term coffee:

text(**"coffee"**)

Available with MongoDB 3.2, a version 3 text index allows case-sensitive searches. This example selects documents that have an abstract field containing the exact term coffee:

text(**"coffee"**, **new** TextSearchOptions().caseSensitive(**true**))

Available with MongoDB 3.2, a version 3 text index allows diacritic-sensitive searches. This example selects documents that have an abstract field containing the exact term café:

text(**"café"**, **new** TextSearchOptions().diacriticSensitive(**true**))

### Bitwise

The bitwise query operators, available with MongoDB 3.2 include:

* bitsAllSet: Selects documents where the all the specified bits of a field are set (i.e. 1).
* bitsAllClear: Selects documents where the all the specified bits of a field are clear (i.e. 0).
* bitsAnySet: Selects documents where at least one of the specified bits of a field are set (i.e. 1).
* bitsAnyClear: Selects documents where at least one of the specified bits of a field are clear (i.e. 0)

Examples

The example selects documents that have a bitField field with bits set at positions of the corresponding bitmask 50 (i.e. 00110010):

bitsAllSet(**"bitField"**, 50)

### Geospatial

The geospatial operator methods include:

* geoWithin: Selects all documents containing a field whose value is a GeoJSON geometry that falls within within a bounding GeoJSON geometry.
* geoWithinBox: Selects all documents containing a field with grid coordinates data that exist entirely within the specified box.
* geoWithinPolygon: Selects all documents containing a field with grid coordinates data that exist entirely within the specified polygon.
* geoWithinCenter: Selects all documents containing a field with grid coordinates data that exist entirely within the specified circle.
* geoWithinCenterSphere: Selects geometries containing a field with geospatial data (GeoJSON or legacy coordinate pairs) that exist entirely within the specified circle, using spherical geometry.
* geoIntersects: Selects geometries that intersect with a GeoJSON geometry. The 2dsphere index supports $geoIntersects.
* near: Returns geospatial objects in proximity to a point. Requires a geospatial index. The 2dsphere and 2d indexes support $near.
* nearSphere: Returns geospatial objects in proximity to a point on a sphere. Requires a geospatial index. The 2dsphere and 2d indexes support $nearSphere.

To make it easier to construct GeoJSON-based filters, the driver also include a full GeoJSON class hierarchy:

* [Point](http://api.mongodb.org/java/3.2/?com/mongodb/client/model/geojson/Point.html): A representation of a GeoJSON Point.
* [MultiPoint](http://api.mongodb.org/java/3.2/?com/mongodb/client/model/geojson/MultiPoint.html): A representation of a GeoJSON MultiPoint.
* [LineString](http://api.mongodb.org/java/3.2/?com/mongodb/client/model/geojson/LineString.html): A representation of a GeoJSON LineString.
* [MultiLineString](http://api.mongodb.org/java/3.2/?com/mongodb/client/model/geojson/MultiLineString.html): A representation of a GeoJSON MultiLineString.
* [Polygon](http://api.mongodb.org/java/3.2/?com/mongodb/client/model/geojson/Polygon.html): A representation of a GeoJSON Polygon.
* [MultiPolygon](http://api.mongodb.org/java/3.2/?com/mongodb/client/model/geojson/MultiPolygon.html): A representation of a GeoJSON MultiPolygon.
* [GeometryCollection](http://api.mongodb.org/java/3.2/?com/mongodb/client/model/geojson/GeometryCollection.html): A representation of a GeoJSON GeometryCollection.

Examples

This example creates a filter that selects all documents where the geo field contains a GeoJSON Geometry object that falls within the given polygon:

Polygon polygon = **new** Polygon(Arrays.asList(**new** Position(0, 0),

**new** Position(4, 0),

**new** Position(4, 4),

**new** Position(0, 4),

**new** Position(0, 0)));

geoWithin(**"geo"**, polygon))

Similarly, this example creates a filter that selects all documents where the geo field contains a GeoJSON Geometry object that intersects the given Point:

geoIntersects(**"geo"**, **new** Point(**new** Position(4, 0)))

## Projections

The [Projections](http://api.mongodb.org/java/3.2/?com/mongodb/client/model/Projections.html) class provides static factory methods for all the MongoDB projection opererators. Each method returns an instance of the [Bson](http://mongodb.github.io/mongo-java-driver/3.2/bson/documents/#bson) type, which can in turn be passed to any method that expects a projection.

For brevity, you may choose to import the methods of the Projections class statically:

**import** **static** com.mongodb.client.model.Projections.\*;

All the examples below assume this static import.

### Inclusion

By default, all fields of each document are projected. To specify the inclusion of one or more fields (which implicitly excludes all other fields except \_id), use the include method.

This example includes the quantity field and (implicitly) the \_id field:

include(**"quantity"**)

This example includes the quantity and totalAmount fields and (implicitly) the \_id field:

include(**"quantity"**, **"totalAmount"**)

### Exclusion

To specify the exclusion of one or more fields (which implicitly includes all other fields), use the excludemethod.

This example excludes the quantity field:

exclude(**"quantity"**)

This example excludes the quantity and totalAmount fields:

exclude(**"quantity"**, **"totalAmount"**)

### Exclusion of \_id

To specify the exclusion of the \_id field, use the excludeId method:

excludeId()

which is just shorthand for:

exclude(**"\_id"**)

### Array Element Match with a Supplied Filter

To specify a projection that includes only the first element of an array that matches a supplied query filter (the[elemMatch](http://docs.mongodb.org/manual/reference/operator/projection/elemMatch) operator), use the elemMatch method that takes a field name and a filter.

This example projects the first element of the orders array where the quantity field is greater that 3:

elemMatch(**"orders"**, Filters.gt(**"quantity"**, 3))

### Array Element Match with an Implicit Filter

To specify a projection that includes only the first element of an array that matches the filter supplied as part of the query (the [positional $ operator](http://docs.mongodb.org/manual/reference/operator/projection/positional/#projection)), use the elemMatch method that takes just a field name.

This example projects the first element of the orders array that matches the query filter:

elemMatch(**"orders"**)

### Slice

To project [a slice of an array](http://docs.mongodb.org/manual/reference/operator/projection/slice), use one of the slice methods.

This example projects the first 7 elements of the tags array:

slice(**"tags"**, 7)

This example skips the first 2 elements of the tags array and projects the next 5:

slice(**"tags"**, 2, 5)

### Text Score

To specify a projection of [the score of a $text query](http://docs.mongodb.org/manual/reference/operator/query/text/#return-the-text-search-score), use the metaTextScore method to specify the name of the projected field.

This example projects the text score as the value of the score field:

metaTextScore(**"score"**)

### Combining Projections

To combine multiple projections, use the fields method.

This example includes the quantity and totalAmount fields and excludes the \_id field:

fields(include(**"quantity"**, **"totalAmount"**), excludeId())

## Sorts Criteria

The [Sorts](http://api.mongodb.org/java/3.2/?com/mongodb/client/model/Sorts.html) class provides static factory methods for all the MongoDB sort criteria operators. Each method returns an instance of the [Bson](http://mongodb.github.io/mongo-java-driver/3.2/bson/documents/#bson) type, which can in turn be passed to any method that expects sort criteria.

For brevity, you may choose to import the methods of the Sorts class statically:

**import** com.mongodb.client.model.Sorts.\*;

All the examples below assume this static import.

### Ascending

To specify an ascending sort, use one of the ascending methods.

This example specifies an ascending sort on the quantity field:

ascending(**"quantity"**)

This example specifies an ascending sort on the quantity field, followed by an ascending sort on thetotalAmount field:

ascending(**"quantity"**, **"totalAmount"**)

### Descending

To specify a descending sort, use one of the descending methods.

This example specifies a descending sort on the quantity field:

descending(**"quantity"**)

This example specifies a descending sort on the quantity field, followed by a descending sort on thetotalAmount field:

descending(**"quantity"**, **"totalAmount"**)

### Text Score

To specify a sort by [the score of a $text query](http://docs.mongodb.org/manual/reference/operator/query/text/#sort-by-text-search-score), use the metaTextScore method to specify the name of the projected field.

This example specifies a sort on the score of a $text query that will be projected into the scoreValue field in a projection on the same query:

metaTextScore(**"scoreValue"**)

### Combining sort criteria

To specify the combination of multiple sort criteria, use the orderBy method.

This example specifies an ascending sort on the quantity field, followed by an ascending sort on thetotalAmount field, followed by a descending sort on the orderDate field:

orderBy(ascending(**"quantity"**, **"totalAmount"**), descending(**"orderDate"**))

## Aggregation

The [Aggregates](http://api.mongodb.org/java/3.2/?com/mongodb/client/model/Aggregates.html) class provides static factory methods that build [aggregation pipeline operators](http://docs.mongodb.org/manual/reference/operator/aggregation/). Each method returns an instance of the [Bson](http://mongodb.github.io/mongo-java-driver/3.2/bson/documents/#bson) type, which can in turn be passed to the aggregate method ofMongoCollection.

For brevity, you may choose to import the methods of the Aggregates class statically:

**import** com.mongodb.client.model.Aggregates.\*;

All the examples below assume this static import.

### Match

The [$match](http://docs.mongodb.org/manual/reference/operator/aggregation/match/) pipeline stage passes all documents matching the specified filter to the next stage. Though the filter can be an instance of any class that implements Bson, it’s convenient to combine with use of the[Filters](http://api.mongodb.org/java/3.2/?com/mongodb/client/model/Filters.html) class. In the example below, it’s assumed that the eq method of the Filters class has been statically imported.

This example creates a pipeline stage that matches all documents where the author field is equal to"Dave":

match(eq(**"author"**, **"Dave"**))

### Project

The [$project](http://docs.mongodb.org/manual/reference/operator/aggregation/project/) pipeline stage passes the projected fields of all documents to the next stage. Though the projection can be an instance of any class that implements Bson, it’s convenient to combine with use of the[Projections](http://api.mongodb.org/java/3.2/?com/mongodb/client/model/Projections.html) class. In the example below, it’s assumed that the include, excludeId, and fieldsmethods of the Projections class have been statically imported.

This example creates a pipeline stage that excludes the \_id field but includes the title and authorfields:

project(fields(include(**"title"**, **"author"**), excludeId()))

#### Projecting Computed Fields

The $project stage can project computed fields as well.

This example simply projects the qty field into a new field called quantity. In other words, it renames the field:

project(computed(**"quantity"**, **"$qty"**))

### Sample

The [$sample](http://docs.mongodb.org/manual/reference/operator/aggregation/sample/) pipeline stage randomly select N documents from its input. This example creates a pipeline stage that randomly selects 5 documents from the collection:

sample(5)

### Sort

The [$sort](http://docs.mongodb.org/manual/reference/operator/aggregation/sort/) pipeline stage passes all documents to the next stage, sorted by the specified sort criteria. Though the sort criteria can be an instance of any class that implements Bson, it’s convenient to combine with use of the [Sorts](http://api.mongodb.org/java/3.2/?com/mongodb/client/model/Sorts.html) class. In the example below, it’s assumed that the descending, ascending, andorderBy methods of the Sorts class have been statically imported.

This example creates a pipeline stage that sorts in descending order according to the value of the age field and then in ascending order according to the value of the posts field:

sort(orderBy(descending(**"age"**), ascending(**"posts"**)))

### Skip

The [$skip](http://docs.mongodb.org/manual/reference/operator/aggregation/skip/) pipeline stage skips over the specified number of documents that pass into the stage and passes the remaining documents to the next stage.

This example skips the first 5 documents:

skip(5)

### Limit

The [$limit](http://docs.mongodb.org/manual/reference/operator/aggregation/limit/) pipeline stage limits the number of documents passed to the next stage.

This example limits the number of documents to 10:

limit(10)

### Lookup

Starting in 3.2, MongoDB provides a new [$lookup](http://docs.mongodb.org/manual/reference/operator/aggregation/lookup/) pipeline stage that performs a left outer join with another collection to filter in documents from the joined collection for processing.

This example performs a left outer join on the fromCollection collection, joining the local field to thefrom field and outputted in the joinedOutput field:

lookup(**"fromCollection"**, **"local"**, **"from"**, **"joinedOutput"**)

### Group

The [$group](http://docs.mongodb.org/manual/reference/operator/aggregation/group/) pipeline stage groups documents by some specified expression and outputs to the next stage a document for each distinct grouping. A group consists of an \_id which specifies the expression on which to group, and zero or more [accumulators](http://docs.mongodb.org/manual/reference/operator/aggregation/group/#accumulator-operator) which are evaluated for each grouping. To simplify the expression of accumulators, the driver includes an [Accumulators](http://api.mongodb.org/java/3.2/?com/mongodb/client/model/Accumulators.html) class with static factory methods for each of the supported accumulators. In the example below, it’s assumed that the sum and avg methods of theAccumulators class have been statically imported.

This example groups documents by the value of the customerId field, and for each group accumulates the sum and average of the values of the quantity field into the totalQuantity and averageQuantity fields, respectively.

group(**"$customerId"**, sum(**"totalQuantity"**, **"$quantity"**), avg(**"averageQuantity"**, **"$quantity"**))

### Unwind

The [$unwind](http://docs.mongodb.org/manual/reference/operator/aggregation/unwind/) pipeline stage deconstructs an array field from the input documents to output a document for each element.

This example outputs, for each document, a document for each element in the sizes array:

unwind(**"$sizes"**)

Available with MongoDB 3.2, this example also includes any documents that have missing or null values for the $sizes field or where the $sizes list is empty:

unwind(**"$sizes"**, **new** UnwindOptions().preserveNullAndEmptyArrays(**true**))

Available with MongoDB 3.2, this example unwinds the sizes array and also outputs the array index into the $position field:

unwind(**"$sizes"**, **new** UnwindOptions().includeArrayIndex(**"$position"**))

### Out

The [$out](http://docs.mongodb.org/manual/reference/operator/aggregation/out/) pipeline stage outputs all documents to the specified collection. It must be the last stage in any aggregate pipeline:

This example writes the pipeline to the authors collection:

out(**"authors"**)

### Creating a Pipeline

The above pipeline operators are typically combined into a list and passed to the aggregate method of aMongoCollection. For instance:

collection.aggregate(Arrays.asList(match(eq(**"author"**, **"Dave"**)),

group(**"$customerId"**, sum(**"totalQuantity"**, **"$quantity"**),

avg(**"averageQuantity"**, **"$quantity"**))

out(**"authors"**)));

## Updates

The [Updates](http://api.mongodb.org/java/3.2/?com/mongodb/client/model/Updates.html) class provides static factory methods for all the MongoDB update operators. Each method returns an instance of the [Bson](http://mongodb.github.io/mongo-java-driver/3.2/bson/documents/#bson) type, which can in turn be passed to any method that expects an update.

For brevity, you may choose to import the methods of the Updates class statically:

**import** com.mongodb.client.model.Updates.\*;

All the examples below assume this static import.

### Field Updates

This section describes update operators that apply to the value of an entire field.

#### Set

The [$set](http://docs.mongodb.org/manual/reference/operator/update/set/) update operator sets the value of a field to to the specified value.

This example sets the value of the quantity field to 11:

set(**"quantity"**, 11)

#### Unset

The [$unset](http://docs.mongodb.org/manual/reference/operator/update/unset/) update operator deletes the field with the given name.

This example deletes the quantity field:

unset(**"quantity"**)

#### Set On Insert

The [$setOnInsert](http://docs.mongodb.org/manual/reference/operator/update/setOnInsert/) update operator sets the value of a field to the given value, but only if the update is an[upsert](http://docs.mongodb.org/manual/tutorial/modify-documents/#specify-upsert-true-for-the-update-specific-fields-operation) that results in an insert of a document.

This example sets the value of the defaultQuantity field to 10 if an upsert resulted in the insert of a document:

setOnInsert(**"defaultQuantity"**, 10)

#### Increment

The [$inc](http://docs.mongodb.org/manual/reference/operator/update/inc/) update operator increments the value of a numeric field by a specified value.

This example increments the value of the quantity field by 5:

inc(**"quantity"**, 5)

#### Multiply

The [$mul](http://docs.mongodb.org/manual/reference/operator/update/mul/) update operator multiplies the value of a numeric field by a specified value.

This example multiplies the value of the price field by 1.2:

mul(**"price"**, 1.2)

#### Rename

The [$rename](http://docs.mongodb.org/manual/reference/operator/update/rename/) update operator renames a field.

This example renames the qty field to quantity:

rename(**"qty"**, **"quantity"**)

#### Min

The [$min](http://docs.mongodb.org/manual/reference/operator/update/min/) update operator updates the value of the field to a specified value if the specified value is less than the current value of the field .

This example sets the value of the lowScore field to the minimum of its current value and 150:

min(**"lowScore"**, 150)

#### Max

The [$max](http://docs.mongodb.org/manual/reference/operator/update/max/) update operator updates the value of the field to a specified value if the specified value is greater than the current value of the field .

This example sets the value of the highScore field to the maximum of its current value and 900:

max(**"highScore"**, 900)

#### Current Date

The [$currentDate](http://docs.mongodb.org/manual/reference/operator/update/currentDate/) update operator sets the value of the field with the specified name to the current date, either as a BSON [date](http://docs.mongodb.org/manual/reference/bson-types/#document-bson-type-date) or as a BSON [timestamp](http://docs.mongodb.org/manual/reference/bson-types/#document-bson-type-timestamp).

This example sets the value of the lastModified field to the current date as a BSON date type:

currentDate(**"lastModified"**)

This example sets the value of the lastModified field to the current date as a BSON timestamp type:

currentTimestamp(**"lastModified"**)

#### Bit

The [$bit](http://docs.mongodb.org/manual/reference/operator/update/bit/) update operator performs a bitwise update of the integral value of a field.

This example performs a bitwise AND between the number 10 and the integral value of the mask field:

bitwiseAnd(**"mask"**, 10)

This example performs a bitwise OR between the number 10 and the integral value of the mask field:

bitwiseOr(**"mask"**, 10)

This example performs a bitwise XOR between the number 10 and the integral value of the mask field:

bitwiseXor(**"mask"**, 10)

### Array Updates

This section describes update operators that apply to the contents of the array value of a field.

#### Add to Set

The [$addToSet](http://docs.mongodb.org/manual/reference/operator/update/addToSet/) update operator adds a value to an array unless the value is already present, in which case $addToSet does nothing to that array.

This example adds the value "a" to the array value of the `letters’ field:

addToSet(**"letters"**, **"a"**)

This example adds each of the values "a", "b", and "c" to the array value of the `letters’ field:

addEachToSet(**"letters"**, Arrays.asList(**"a"**, **"b"**, **"c"**))

#### Pop

The [$pop](http://docs.mongodb.org/manual/reference/operator/update/pop/) update operator removes the first or last element of an array.

This example pops the first element off of the array value of the scores field:

popFirst(**"scores"**)

This example pops the last element off of the array value of the scores field:

popLast(**"scores"**)

#### Pull All

The [$pullAll](http://docs.mongodb.org/manual/reference/operator/update/pullAll/) update operator removes all instances of the specified values from an existing array.

This example removes the scores 0 and 5 from the scores array:

pullAll(**"scores"**, Arrays.asList(0, 5))

#### Pull

The [$pull](http://docs.mongodb.org/manual/reference/operator/update/pull/) update operator removes from an existing array all instances of a value or values that match a specified query.

This example removes the value 0 from the scores array:

pull(**"scores"**, 0)

This example removes all elements from the votes array that are greater than or equal to 6:

pullByFilter(Filters.gte(**"votes"**, 6))

#### Push

The [$push](http://docs.mongodb.org/manual/reference/operator/update/push/) update operator appends a specified value to an array.

This examples pushes the value 89 to the scores array:

push(**"scores"**, 89)

This examples pushes each of the values 89, 90, and 92 to the scores array:

pushEach(**"scores"**, Arrays.asList(89, 90, 92))

This example pushes each of the values 89, 90, and 92 to the start of the scores array:

pushEach(**"scores"**, Arrays.asList(89, 90, 92), **new** PushOptions().position(0))

This example pushes each of the values 89, 90, and 92 to the scores array, sorts the array in descending order, and removes all but the first 5 elements of the array:

pushEach(**"scores"**, Arrays.asList(89, 90, 92), **new** PushOptions().sort(-1).slice(5))

This example pushes each of the documents { wk: 5, score: 8 }, { wk: 6, score: 7 }, and{ wk: 7, score: 6 } to the quizzes array, sorts the array in descending order by score, and removes all but the last 3 elements of the array:

pushEach(**"quizzes"**,

Arrays.asList(**new** Document(**"week"**, 5).append(**"score"**, 8),

**new** Document(**"week"**, 6).append(**"score"**, 7),

**new** Document(**"week"**, 7).append(**"score"**, 6)),

**new** PushOptions().sortDocument(Sorts.descending(**"score"**)).slice(-3))

### Combining Multiple Update Operators

Often, an application will need to atomically update multiple fields of a single document by combine two or more of the update operators described above.

This example sets the value of the quantity field to 11, the value of the total field to 30.40, and pushes each of the values 4.99, 5.99, and 10.99 to the array value of the prices field:

combine(set(**"quantity"**, 11),

set(**"total"**, 30.40),

pushEach(**"prices"**, Arrays.asList(4.99, 5.99, 10.99)))

## Indexes

The [Indexes](http://api.mongodb.org/java/3.2/?com/mongodb/client/model/Indexes.html) class provides static factory methods for all the MongoDB Index key types.  
Each method returns an instance of the [Bson](http://mongodb.github.io/mongo-java-driver/3.2/bson/documents/#bson) type, which can in turn be used with the createIndexmethods.

For brevity, you may choose to import the methods of the Indexes class statically:

**import** com.mongodb.client.model.Indexes.\*;

All the examples below assume this static import.

### Ascending

To specify an ascending index key, use one of the ascending methods.

This example specifies an ascending index key for the quantity field:

ascending(**"quantity"**)

This example specifies a compound index key composed of the quantity field sorted in ascending order and the totalAmount field sorted in ascending order:

ascending(**"quantity"**, **"totalAmount"**)

### Descending

To specify a descending index key, use one of the descending methods.

This example specifies a descending index key on the quantity field:

descending(**"quantity"**)

This example specifies a compound index key composed of the quantity field sorted in descending order and the totalAmount field sorted in descending order:

descending(**"quantity"**, **"totalAmount"**)

### Compound indexes

To specify a compound index, use the compoundIndex method.

This example specifies a compound index key composed of the quantity field sorted in ascending order, followed by the totalAmount field sorted in ascending order, followed by the orderDate field sorted in descending order:

compoundIndex(ascending(**"quantity"**, **"totalAmount"**), descending(**"orderDate"**))

### Text Index

To specify a [text](http://docs.mongodb.org/manual/core/index-text) index key, use the text method.

This example specifies a text index key for the description field:

text(**"description"**)

### Hashed Index

To specify a [hashed](http://docs.mongodb.org/manual/core/index-hashed) index key, use the hashed method.

This example specifies a hashed index key for the timestamp field:

hashed(**"timestamp"**)

### Geospatial Indexes

There are also helpers for creating the index keys for the various [geospatial indexes](http://docs.mongodb.org/manual/applications/geospatial-indexes) supported by mongodb.

#### 2dsphere

To specify a [2dsphere](http://docs.mongodb.org/manual/core/2dsphere/) index key, use one of the geo2dsphere methods.

This example specifies a 2dsphere index on the location field:

geo2dsphere(**"location"**)

#### 2d

To specify a [2d](http://docs.mongodb.org/manual/core/2d/) index key, use the geo2d method.

**IMPORTANT**

A 2d index is for data stored as points on a two-dimensional plane and is intended for legacy coordinate pairs used in MongoDB 2.2 and earlier.

This example specifies a 2d index on the points field:

geo2d(**"points"**)

#### geoHaystack

To specify a [geoHaystack](http://docs.mongodb.org/manual/core/geohaystack/) index key, use the geoHaystack method.

**IMPORTANT**

For queries that use spherical geometry, a 2dsphere index is a better option than a haystack index. 2dsphere indexes allow field reordering; geoHaystack indexes require the first field to be the location field. Also, geoHaystack indexes are only usable via commands and so always return all results at once.

This example specifies a geoHaystack index on the position field and an ascending index on the typefield:

geoHaystack(**"position"**, ascending(**"type"**))

# Issues & Help

We are lucky to have a vibrant MongoDB Java community with lots of varying experience of using the Java driver. We often find the quickest way to get support for general questions is through the [mongodb-user google group](http://groups.google.com/group/mongodb-user) or through [stackoverflow](http://stackoverflow.com/questions/tagged/mongodb+java). Please also refer to our own [support channels](http://www.mongodb.org/about/support) documentation.

## Bugs / Feature Requests

If you think you’ve found a bug or want to see a new feature in the Java driver, please open a case in our issue management tool, JIRA:

* [Create an account and login](https://jira.mongodb.org/).
* Navigate to [the JAVA project](https://jira.mongodb.org/browse/JAVA).
* Click **Create Issue** - Please provide as much information as possible about the issue type and how to reproduce it.

Bug reports in JIRA for the Java driver and the Core Server (i.e. SERVER) project are **public**.

If you’ve identified a security vulnerability in a driver or any other MongoDB project, please report it according to the [instructions here](http://docs.mongodb.org/manual/tutorial/create-a-vulnerability-report).

## Pull Requests

We are happy to accept contributions to help improve the driver. We will guide user contributions to ensure they meet the standards of the codebase. Please ensure that any pull requests include documentation, tests and also pass a the gradle checks.

To get started check out the source and work on a branch:

$ git clone https://github.com/mongodb/mongo-java-driver.git

$ cd mongo-java-driver

$ git checkout -b myNewFeature

Finally, ensure that the code passes gradle checks.

$ ./gradlew check