Large-Scale and Multi-Structured Databases

MongoDB Java Driver

Prof. Pietro Ducange

Ing. Alessio Schiavo

alessio.schiavo@phd.unipi.it

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

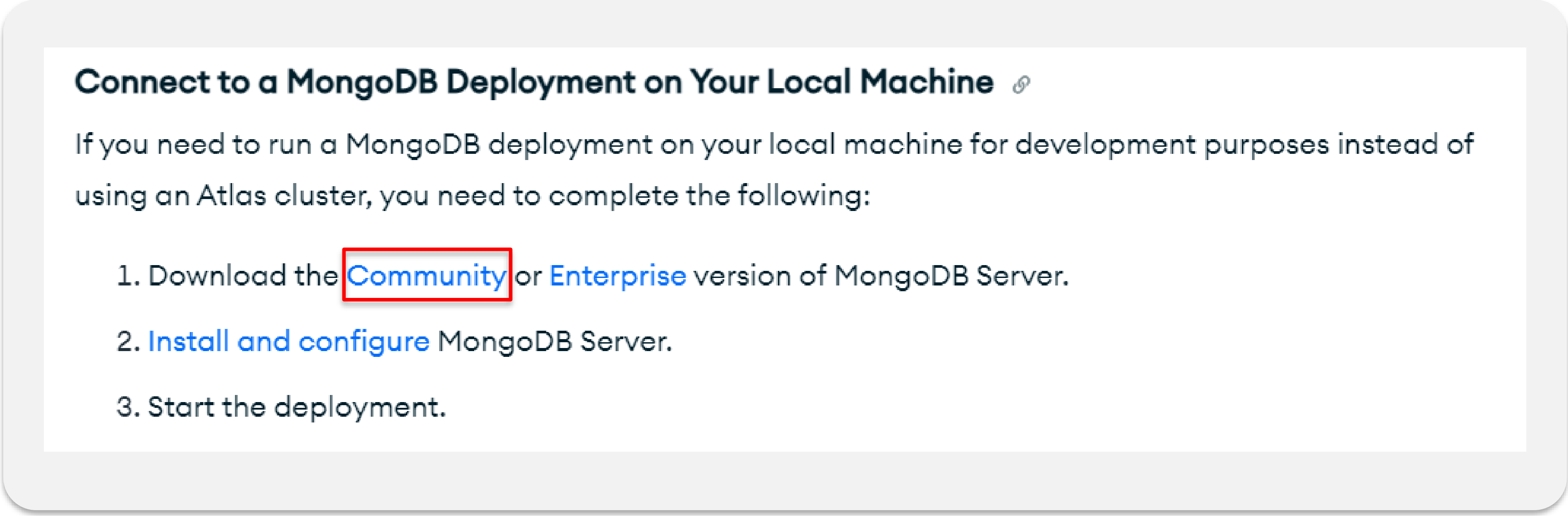
UNIVERSITÀ DI PISA

CROSSLAB
Innovation for industry 4.0

Copyright Issues

Most of the information included this presentation have been extracted from the official documentation of MongoDB Java Driver (http://mongodb.github.io/mongo-java-driver/).

Pre-requisites (I)



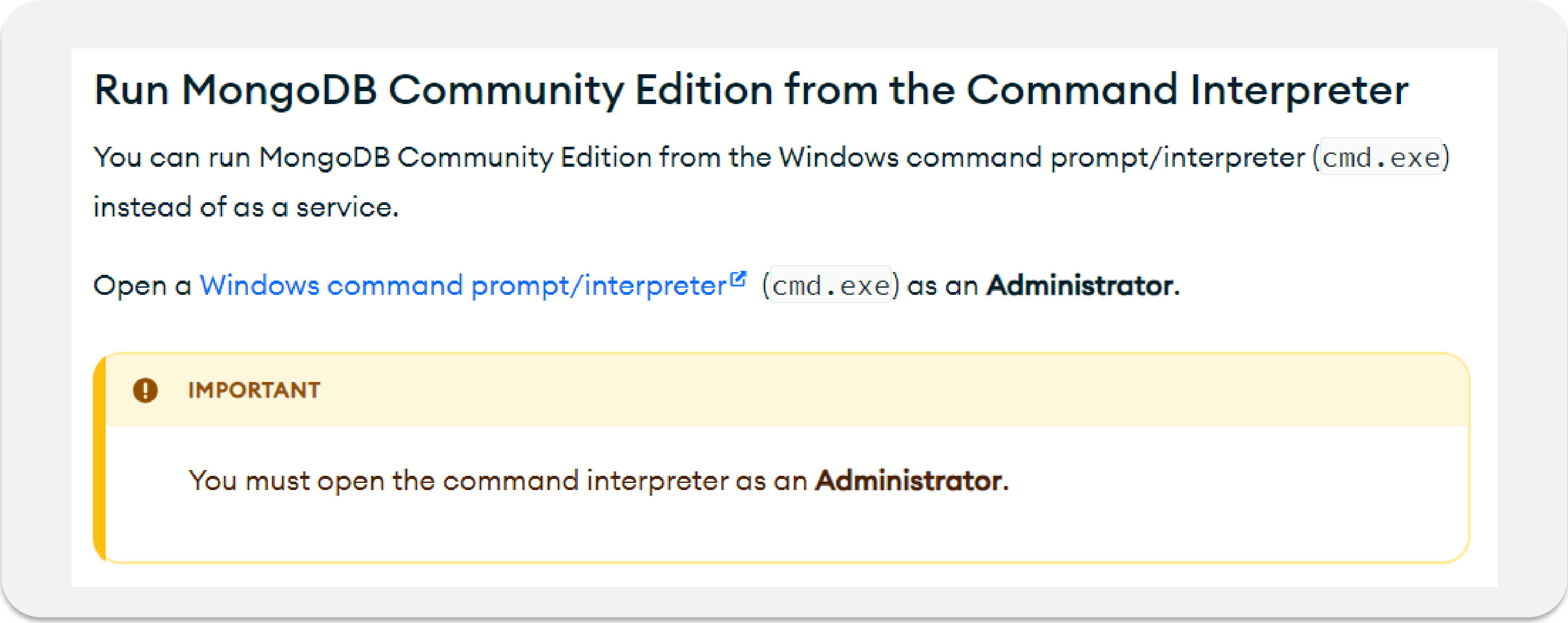
Connect to a MongoDB Deployment on Your Local Machine

If you need to run a MongoDB deployment on your local machine for development purposes instead of using an Atlas cluster, you need to complete the following:

1. Download the Community or Enterprise version of MongoDB Server.
2. Install and configure MongoDB Server.
3. Start the deployment.

https://www.mongodb.com/docs/manual/installation/#std-label-tutorials-installation

Pre-requisites (II)



Run MongoDB Community Edition from the Command Interpreter

You can run MongoDB Community Edition from the Windows command prompt/interpreter (`cmd.exe`) instead of as a service.

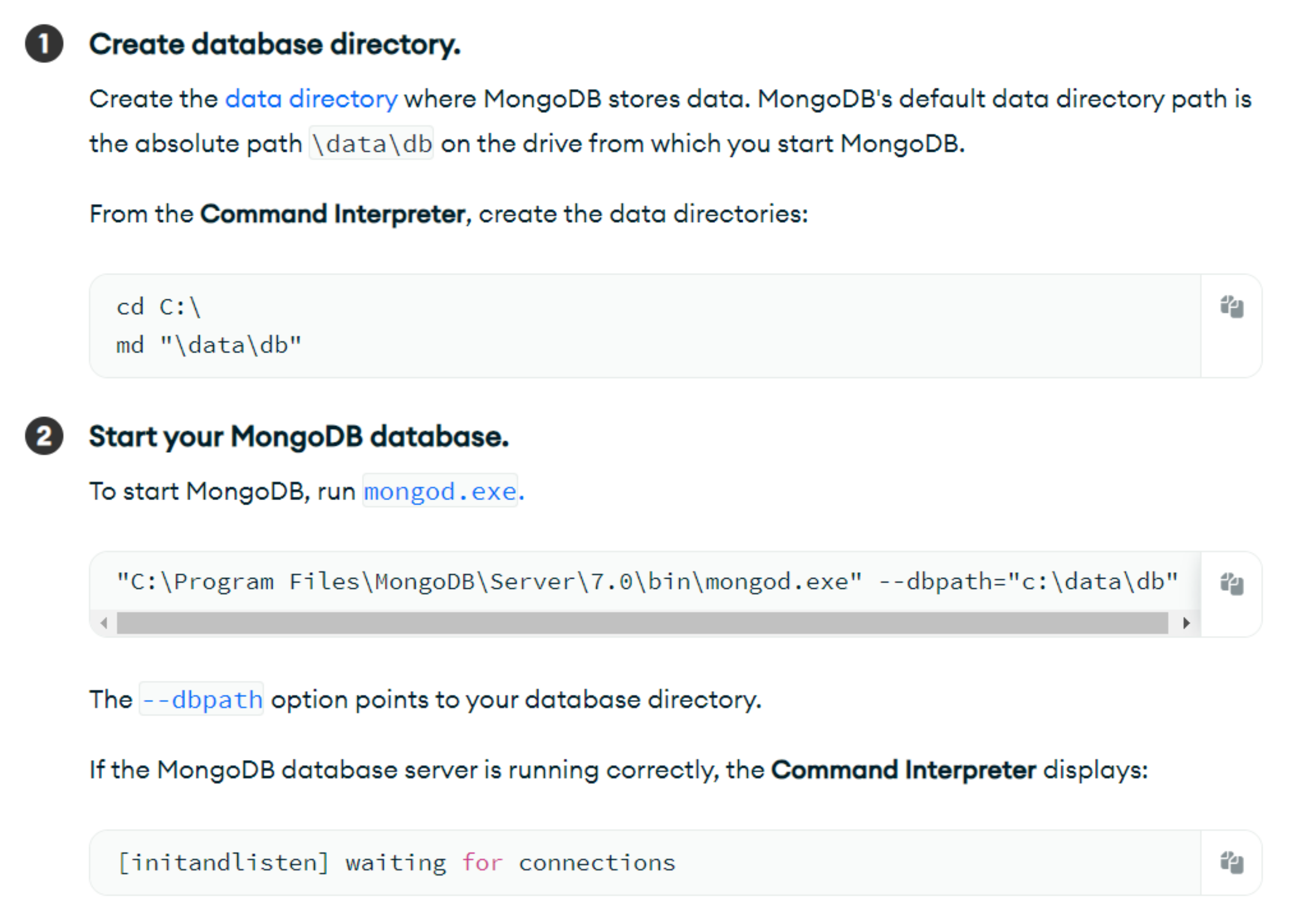
Open a Windows command prompt/interpreter (`cmd.exe`) as an **Administrator**.

IMPORTANT

You must open the command interpreter as an **Administrator**.

https://www.mongodb.com/docs/manual/tutorial/install-mongodb-on-windows/#std-label-run-mongodb-from-cmd

Pre-requisites (III)



1 Create database directory.

Create the data directory where MongoDB stores data. MongoDB's default data directory path is the absolute path `\data\db` on the drive from which you start MongoDB.

From the **Command Interpreter**, create the data directories:

```
cd C:\
md "\data\db"
```

2 Start your MongoDB database.

To start MongoDB, run `mongod.exe`.

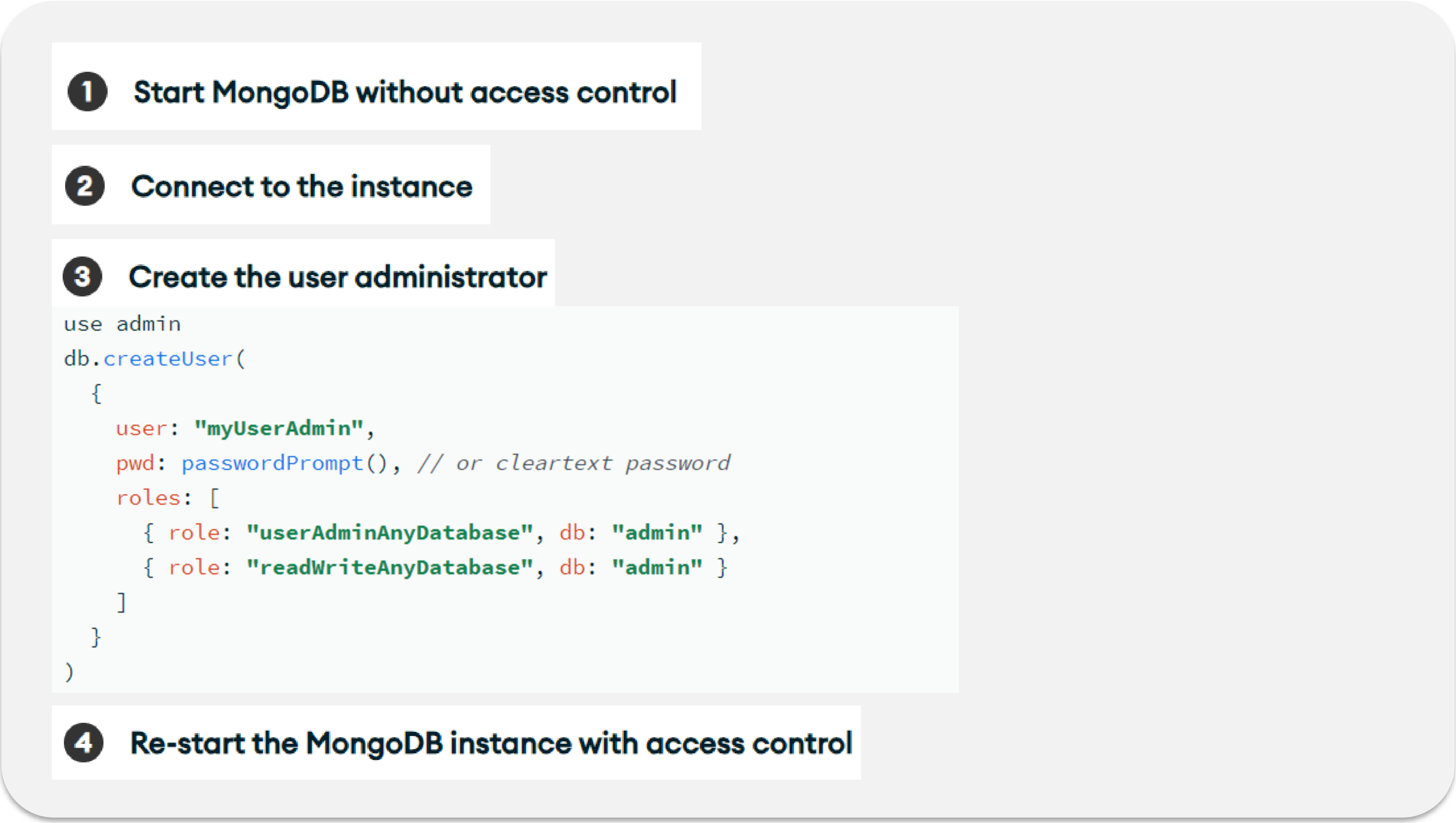
```
"C:\Program Files\MongoDB\Server\7.0\bin\mongod.exe" --dbpath="c:\data\db"
```

The `--dbpath` option points to your database directory.

If the MongoDB database server is running correctly, the **Command Interpreter** displays:

```
[initandlisten] waiting for connections
```

Pre-requisites (IV)



1. Start MongoDB without access control

2. Connect to the instance

3. Create the user administrator

```
use admin
db.createUser(
  {
    user: "myUserAdmin",
    pwd: passwordPrompt(), // or cleartext password
    roles: [
      { role: "userAdminAnyDatabase", db: "admin" },
      { role: "readWriteAnyDatabase", db: "admin" }
    ]
  }
)
```

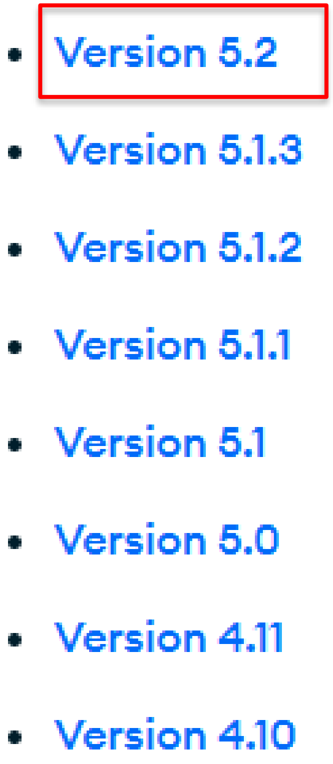
4. Re-start the MongoDB instance with access control

https://www.mongodb.com/docs/manual/tutorial/configure-scram-client-authentication/

MongoDB Java Driver

This driver allows us to manipulate using Java Application data stored in a MongoDB database.

The main suggestion is to follow the official documentation available at https://www.mongodb.com/docs/drivers/java/sync/current/



- Version 5.2
- Version 5.1.3
- Version 5.1.2
- Version 5.1.1
- Version 5.1
- Version 5.0
- Version 4.11
- Version 4.10

Installation

- The recommended way to get started using ***one of the drivers*** in your project is with a dependency management system, such as ***MAVEN***.

- The current official indication is to use the ***mongodb-driver-sync*** in the Java application.

- Specify in the pom file the following dependency

```
<dependencies>
  <dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongodb-driver-sync</artifactId>
    <version>5.2.0</version>
  </dependency>
</dependencies>
```

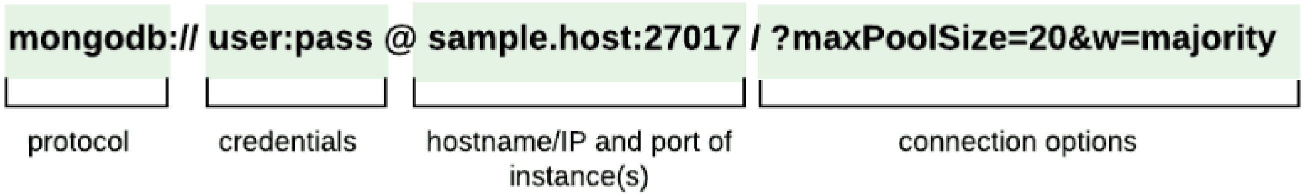
Making a Connection

- The ***MongoClients.create(),*** allows us to make a connection to a running MongoDB instance.

- A simple way to connect to a MongoDB server is:

```
MongoClients.create(connectionString)
```

```
MongoClient myClient =
MongoClients.create("mongodb://user:pass@localhost:27017");
```

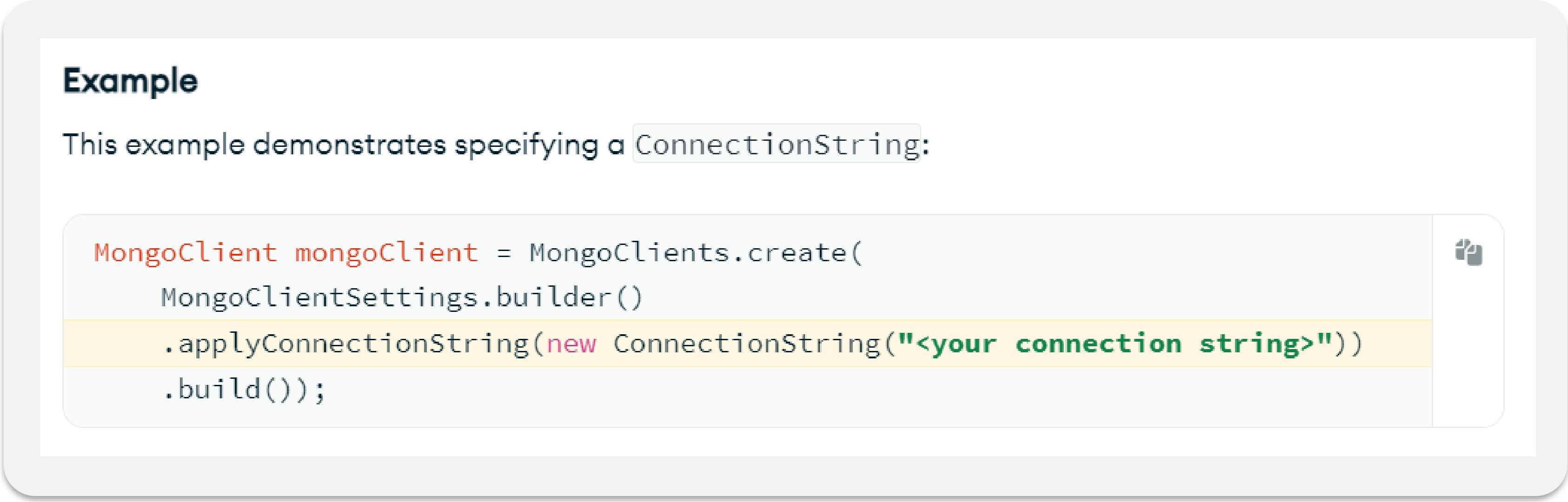


mongodb:// user:pass @ sample.host:27017 / ?maxPoolSize=20&w=majority

https://www.mongodb.com/docs/drivers/java/sync/current/fundamentals/connection/connection-options/#std-label-connection-options

Some Important Considerations

- You can control the behavior of your MongoClient by creating and passing in **a MongoClientSettings object** to the MongoClients.create() method.



Example

This example demonstrates specifying a `ConnectionString`:

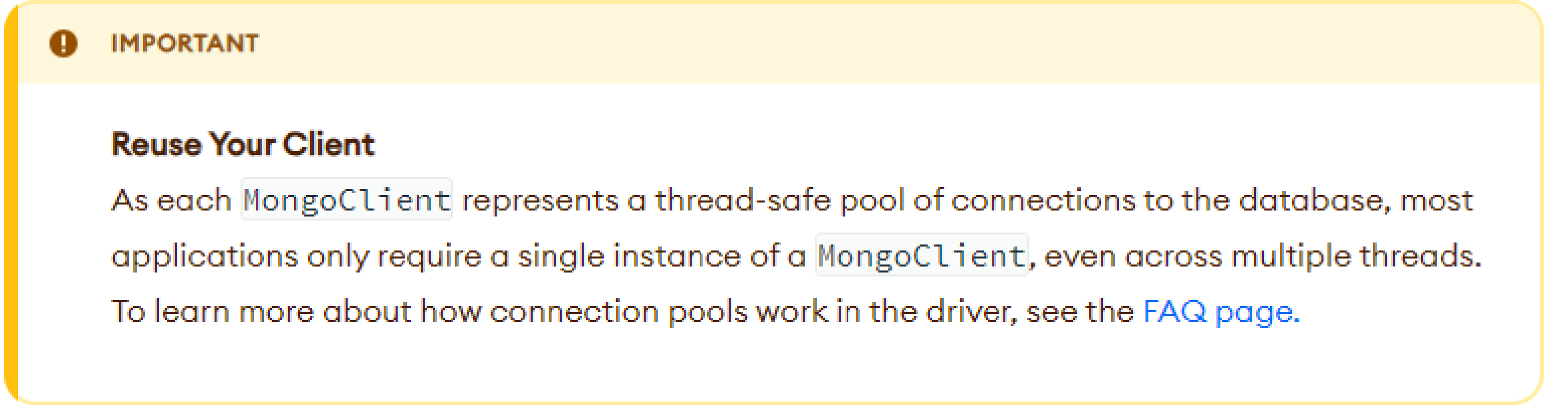
```
MongoClient mongoClient = MongoClients.create(
    MongoClientSettings.builder()
    .applyConnectionString(new ConnectionString("<your connection string>"))
    .build());
```

https://www.mongodb.com/docs/drivers/java/sync/current/fundamentals/connection/mongoclientsettings/

Some Important Considerations

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

- **IMPORTANT:** Mongoclient is thread-safe. Multiple access to the single instance is managed by the class itself



IMPORTANT

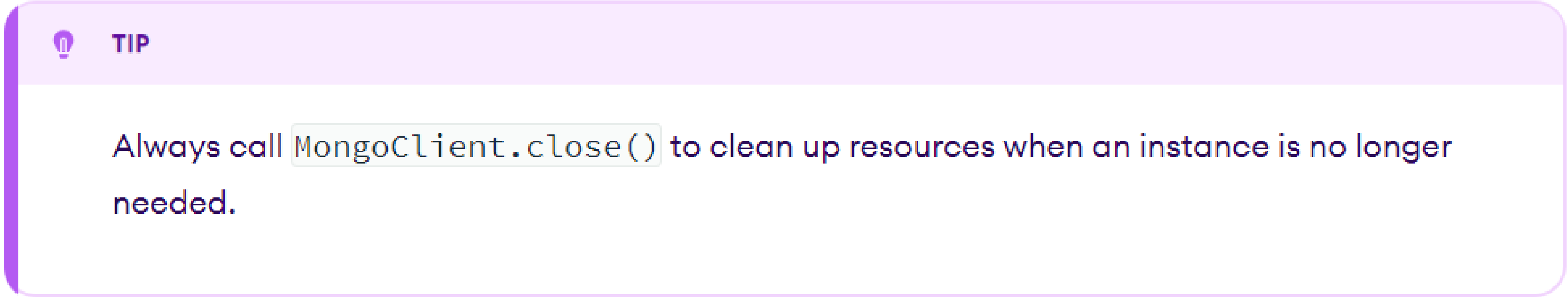
Reuse Your Client

As each `MongoClient` represents a thread-safe pool of connections to the database, most applications only require a single instance of a `MongoClient`, even across multiple threads. To learn more about how connection pools work in the driver, see the FAQ page.

https://www.mongodb.com/docs/drivers/java/sync/current/faq/#std-label-java-faq-connection-pool

Some Important Considerations

- Typically, for simple projects, we only create ***one MongoClient instance*** for a given MongoDB deployment (e.g. standalone, replica set, or a sharded cluster) and use it ***across the whole application.***



TIP

Always call `MongoClient.close()` to clean up resources when an instance is no longer needed.

- Call ***MongoClient.close()*** to clean up resources at the end of the application.

Access a Database

- Once we have a MongoClient instance connected to a MongoDB deployment, we can use the ***MongoClient.getDatabase()*** method to access a database.

- ***Specify the name*** of the database to the getDatabase() method. If a database does not exist, MongoDB creates the database when you first store data for that database.

- The following example accesses the mydb database:

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

Access a Collection

- Once we have a ***MongoDatabase*** instance, we can use its ***getCollection()*** method to access a collection.

- ***Specify the name*** of the collection to the getCollection() method. If a collection does not exist, MongoDB creates the collection when you first store data for that collection.

- For example, using the database instance, the following statement accesses the ***collection*** named ***test*** in the mydb database:

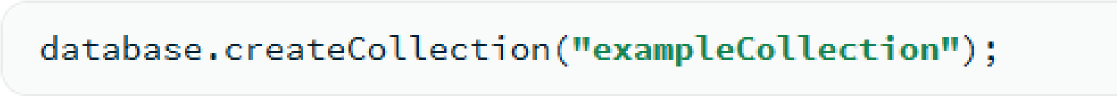
```
MongoCollection<Document> collection = database.getCollection("test");
```

Code overview

```
import com.mongodb.client.*;
import com.mongodb.ConnectionString;

//Create connection string
ConnectionString uri = new ConnectionString("mongodb://localhost:27017”);
//Create a mongoDB client
MongoClient myClient = MongoClients.create(uri);

//Connect to mydb database
MongoDatabase database = mongoClient.getDatabase("mydb");

//Select the collection test
MongoCollection<Document> collection = database.getCollection("test");

…
//insert, remove, update elements to/from the collection
…

//Close mongoDB connection and release resources
myClient.close();
```

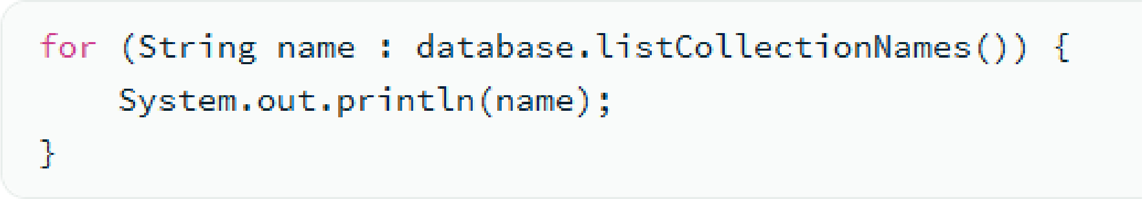
Handling Collections basics

- To **create a Collection using the Java driver**, we can use the ***createCollection*** method of a MongoDatabase instance. For example, let us create a collection called “exampleCollection”:



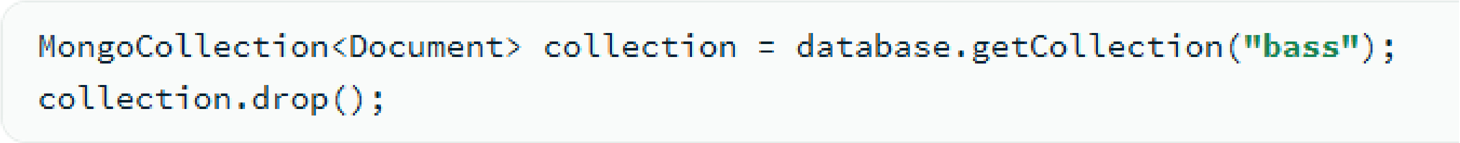
```
database.createCollection("exampleCollection");
```

- To **get list of existing Collections using the Java driver**, we can use the *MongoDatabase.listCollectionNames()* method:



```
for (String name : database.listCollectionNames()) {
    System.out.println(name);
}
```

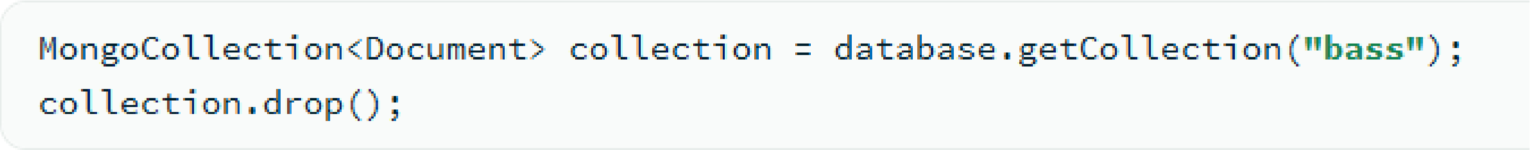
- To **drop a Collection using the Java driver**, we can use the *MongoCollection.drop()* method:



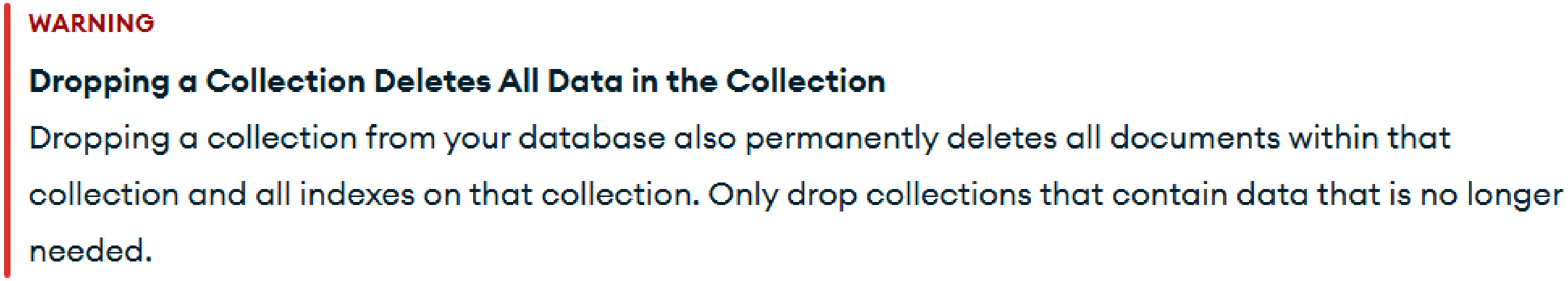
```
MongoCollection<Document> collection = database.getCollection("bass");
collection.drop();
```

Handling Collections basics

- To **drop a Collection using the Java driver**, we can use the *MongoCollection.drop()* method:



```
MongoCollection<Document> collection = database.getCollection("bass");
collection.drop();
```



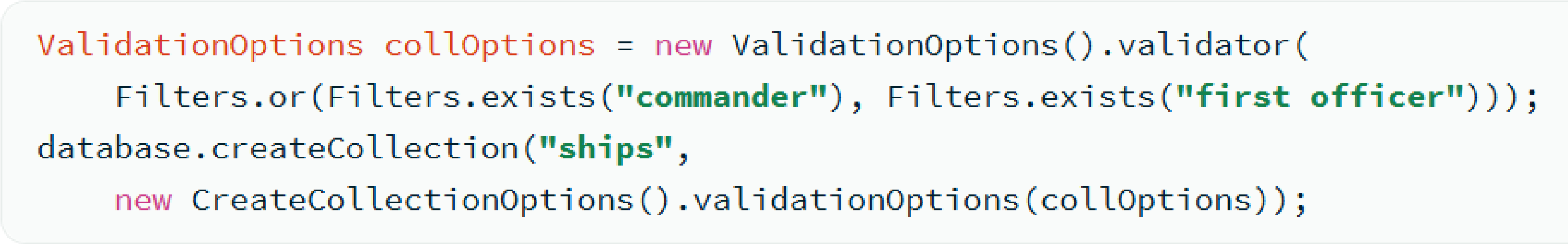
WARNING

Dropping a Collection Deletes All Data in the Collection

Dropping a collection from your database also permanently deletes all documents within that collection and all indexes on that collection. Only drop collections that contain data that is no longer needed.

Document Validation

- Document validation provides the ability to validate documents against a series of filter during writes to a collection. You can specify these filters using the **ValidationOptions** class, which accepts a series of **Filters**



```
ValidationOptions collOptions = new ValidationOptions().validator(
    Filters.or(Filters.exists("commander"), Filters.exists("first officer")));
database.createCollection("ships",
    new CreateCollectionOptions().validationOptions(collOptions));
```

- Schema validation is most useful for an established application where you have a good sense of how to organize your data.

- After you add schema validation rules to a collection:
  - **All** document inserts **must match the rules**.
  - The **schema validation level defines how the rules are applied** to existing documents and document updates.

https://www.mongodb.com/docs/manual/core/schema-validation/

Create a Document

- To create the document using the Java driver, we can use the ***Document class***. For example, consider the following JSON document:

string	**{ "name" : "MongoDB",**
number	**"count" : 1,**
list	**"versions": ["v3.2", "v3.0", "v2.6"],**
embedded obj	**"info" : { x : 203, y : 102 } }**

```
Document doc = new Document("name", "MongoDB")
               .append("count", 1)
               .append("versions", Arrays.asList("v3.2", "v3.0", "v2.6"))
               .append("info", new Document("x", 203).append("y", 102));
```

- Otherwise, you can use a string that represent the json file (Be careful to escape double quotes!)

```
Document doc = Document.parse("{name:\“Alessio\", surname:\“Schiavo\"}");
```

https://learn.mongodb.com/learn/course/mongodb-crud-operations-in-java/lesson-1-working-with-mongodb-documents-in-java/learn?client=customer&page=1

Insert Document

- To insert a single document into the collection, we can use the collection's insertOne() method.

```
collection.insertOne(doc);
```

- To insert a set of documents, contained into a list of documents we can use the following example of code:

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

[…populate documents…]

//Insert multiple documents
collection.insertMany(documents);
//count the # of docs in a collection
System.out.println(collection.countDocuments());
```

https://learn.mongodb.com/learn/course/mongodb-crud-operations-in-java/lesson-2-inserting-a-document-in-java-applications/learn?client=customer

Query a Collection

- To query a collection, we can use the collection's ***find() method***.

- We can call the method without any arguments ***to query all documents*** in a collection or ***pass a filter*** to query for documents that ***match the*** filter ***criteria***.

- The following example retrieves all documents in the collection and prints the returned documents:

```
try (MongoCursor<Document> cursor = myColl.find().iterator())
{
   while (cursor.hasNext())
   {
      System.out.println(cursor.next().toJson());
   }
}
```

https://learn.mongodb.com/learn/course/mongodb-crud-operations-in-java/lesson-3-querying-a-mongodb-collection-in-java-applications/learn?client=customer&page=1

Show results of a query

- We can iterate through query results by using a consumer function (statically or locally defined)

```
import java.util.function.Consumer;

Consumer<Document> printDocuments = doc ->
      {System.out.println(doc.toJson());};

myColl.find().forEach(printDocuments);
```

- Collect results in a list

```
List<Document> results =
    myColl.find().into(new ArrayList<>);
```

Specify a Query Filter

- To query for documents that match certain conditions, pass a ***filter object*** to the find() method.

- To facilitate creating filter objects, Java driver provides the Filters helper (link)

- The following example retrieves all documents in the collection where 50 < i <= 100 and prints the returned documents:

```
try (MongoCursor<Document> cursor =
         myColl.find(and(gt("i", 50), lte("i", 100))).iterator())
{
    while (cursor.hasNext())
    {
        System.out.println(cursor.next().toJson());
    }
}
```

Update Documents

- To update documents in a collection, we can use the collection’s ***updateOne*** and ***updateMany*** methods.

- These functions needs two parameters:
  - ***A filter object*** to determine the document or documents to update.
  - ***An update document*** that specifies the modifications. Check the manual for a list of the available operators

```
collection.updateOne(
            eq(“name", “Alessio”),
            set(“age", 28));
```

Update Documents: Examples

- The following example updates the first document that meets the filter ***i equals 10*** and ***sets*** the value of ***i to 110:***

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

- The following example ***increments the value of i by 100*** for all documents where the value of field ***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 which provides information about the operation including the number of documents modified by the update.

https://learn.mongodb.com/learn/course/mongodb-crud-operations-in-java/lesson-4-updating-documents-in-java-applications/learn?client=customer&page=1

Delete Documents

- To delete documents from a collection, we can use the collection’s ***deleteOne*** and ***deleteMany*** methods.

- As a parameter it requires just a ***filter object*** to select the documents to delete. The example deletes at ***most one document*** that meets the filter ***i equals 110:***

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

- The following example deletes ***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*** which provides information about the operation including the number of documents deleted.

Aggregation pipeline

- Aggregation operations process data in your MongoDB collections and return **computed** results.

- An aggregation pipeline is similar to a car factory: the **aggregation pipeline** is the assembly line, **aggregation stages** are the assembly stations, and **operator expressions** are the specialized tools.

- **Find Operations**:
  - Select *what* documents to return
  - Select *what* fields to return
  - Sort the results

- **Aggregation Operations**:
  - Perform find operations
  - **Rename** fields
  - **Calculate** fields
  - **Summarize** data
  - **Group** values

Aggregation pipeline

- To perform aggregation, pass a list of aggregation stages to the ***MongoCollection.aggregate()*** method.

- For a complete list of aggregations, check the reference documentation

```
Bson myMatch = Aggregates.match(Filters.eq(“categories", “Bakery"));
Bson myGroup = Aggregates.group("$stars", Accumulators.sum("count", 1));

collection.aggregate(Arrays.asList(myMatch, myGroup))
             .forEach(printDocuments());
```

Aggregation pipeline (2)

- Import static filters, aggregations, projections and accumulators to improve readability of your code

```
Import static com.mongodb.client.model.Aggregates.*;
Import static com.mongodb.client.model.Accumjulators.*;
Import static com.mongodb.client.model.Projections.*;
Import static com.mongodb.client.model.Filters.*;

Bson myMatch = match(eq(“categories", “Bakery"));
Bson myGroup = group("$stars", sum("count", 1));

collection.aggregate(Arrays.asList(myMatch, myGroup))
             .forEach(printDocuments());
```

Pipeline - match

- The **match** operator filters the documents to pass only the ones that match the specified condition(s) to the next pipeline stage.

```
//Strings
Bson myMatch = match(eq("categories", "Bakery"));

//Integers
Bson myMatch2 = match(gte("pop", 50000));

//Logic operators
Bson myMatch3 = match(
   and(eq("name",“XYZ Coffee Bar"), eq(“categories", “Coffee")));

collection.aggregate(Arrays.asList(myMatch, myMatch2, myMatch3))
   .forEach(doc -> System.out.println(doc.toJson()));
```

Pipeline - group

- The **group** operator groups input documents by the specified `_id` expression (first argument) and for each distinct grouping. It returns a document.
- This operator can include accumulators (sum, avg, max, min, …)

```
Bson groupSingle = group("$city", sum("totPop", "$pop"));
```

- For **multiple fields grouping** it is better to define directly a document:

```
Bson groupMultiple = new Document("$group",
    new Document("_id", new Document("city", "$city")
        .append("state", "$state"))
        .append("totalPop", new Document("$sum", "$pop")));

//Same as defining the following document
//{$group: {_id:{city:$city, state:$state}, totalPop:{$sum:$pop}}}
```

Pipeline - project

- The **project** operator passes along the documents with the requested fields to the next stage in the pipeline.
- The specified fields can be existing fields from the input documents or newly computed fields.
- I can exclude, include or compute new fields

```
Bson p1 = project(fields(include("city")));
Bson p2 = project(fields(
               exludeId(),include("city", "state")));
Bson p3 = project(fields(
               computed("myID", "_id"),include("city")));
```

Pipeline - sort

- The **sort** operator Sorts all input documents and returns them to the pipeline in sorted order.
- The order can be *ascending* or *descending*

```
Bson s1 = sort(descending("pop"));
```

Compound Sort:

```
Bson s2 = sort(ascending("city"), descending("pop"));

collection.aggregate(Arrays.asList(s2))
                 .forEach(printDocuments());
```

Pipeline – limit, skip

- The **limit** operator limits the number of documents passed to the next stage
- In conjunction with **limit**, we can implement queries that search, for example, for the top 3 biggest cities
- The **skip** operator, instead, skips over the specified number of documents that pass into the next stage

```
//Find the 2nd and 3rd biggest cities
Bson mySort = sort(descending("pop"));
Bson myLimit = limit(2);
Bson mySkip = skip(1);

collection.aggregate(Arrays.asList(mySort, mySkip, myLimit))
                 .forEach(printDocuments());
```

Pipeline – unwind

- The **unwind** operator **deconstructs an array field** from the input documents to **output a document for *each* element**
- Each output document identical to the input doc except for the value of the *grades* field which now holds a value from the original *grades* array:

```
//Find the average score for the American cusine
Bson m = match(eq("cusine", "American"));
Bson u = unwind("grades");
Bson g = group("$cusine", avg("avgGrade","$grades");

collection.aggregate(Arrays.asList(m, u, g))
                 .forEach(printDocuments());
```

Create Indexes

- To create an index on a field or fields, pass an index specification document to the ***createIndex()*** method.

- An index key specification document contains the ***fields*** to index and the ***index type*** for each field:

new Document(<field1>, <type1>).append(<field2>, <type2>) ...

- For an **ascending** index type, specify **+1** for <type>. For a **descending** index type, specify **-1** for <type>.

- The following example creates an ascending index on the city field:

```
collection.createIndex(new Document("city", 1));
```

Exercises

ZIPS dataset

1. Find zip codes of Texan cities.
2. Find cities' zips with a population of at least 100’000, but no more than 200’000.
3. Find the 5 most populated cities.
4. For each state, find the average population of its cities.

POSTS dataset

1. Find posts published after 2012-11-20.
2. Find posts with the tag 'computer'.
3. Find, for each tag, the total number of posts.
4. Find the top three commentators according to the number of comments.
5. Find the most versatile commentator. *Versatile* means that he/she commented on the highest number of *distinct* topics (a.k.a. tags). For a tag to count, he/she must have at least five comments about that topic.

Help: You can find useful examples on ZIPS and POSTS datasets here.

Suggested Readings

Students are invited to read the official documentation of MongoDB.

The documentation is available at (latest version 4.1.1): http://mongodb.github.io/mongo-java-driver/

Check here https://docs.mongodb.com/ecosystem/drivers/ for drivers for ***different programming languages*** and their details.