Advanced Java Programming Course

MongoDB for Java Developers

Faculty of Information Technologies
Industrial University of Ho Chi Minh City

Session objectives

- https://www.mongodb.com/docs/
  - Get started with MongoDB Server
  - Get Started with Atlas
  - Get started with MongoDB Compass
  - Get started the Developer Data Platform
  - Introduction to MongoDB Data Modeling
  - Start Developing with MongoDB: MongoDB Java Drivers

Using MongoDB with Java

- https://www.mongodb.com/docs/drivers/java/sync/current/
- The fundamentals sections:
  - Connect to MongoDB
  - Using MongoDB databases and collections with the MongoDB Java driver
  - Convert between MongoDB Data Formats and Java Objects
  - CRUD Operations
  - Using aggregation operations in the MongoDB Java driver
  - Simplify your Code with Builders
  - Using indexes with the MongoDB Java driver
  - MongoDB Transactions

Using MongoDB Java Client Libraries

- MongoDB Java applications must use the official drivers
- Drivers simplify connecting to and interacting with a MongoDB deployment
- Official driver documentation is on the MongoDB website

Connect to MongoDB

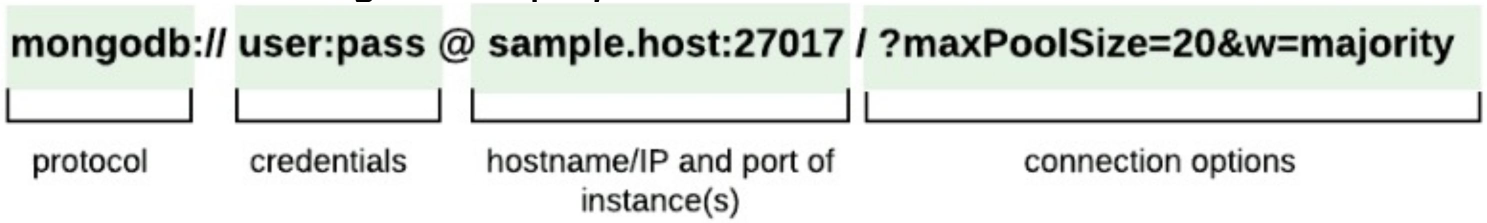
- MongoDB Java drivers are added to a Maven-based Java application by using the pom.xml file
- We need a valid connection string and the MongoDB Java driver to connect
- The connection string is available
- An application must use a single MongoClient instance
- To create a single MongoClient instance for a Java application, we can use a singleton design pattern

Connect to MongoDB

- Using the MongoClient class to connect and communicate to MongoDB.
- Use the MongoClients.create() method to construct a MongoClient.
- 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.
- All resource usage limits, such as max connections, apply to individual MongoClient instances.
- To control the behavior of a MongoClient by creating and passing in a MongoClientSettings object to the MongoClients.create() method.

Connect to MongoDB

- Connection URI
  - The connection URI provides a set of instructions that the driver uses to connect to a MongoDB deployment.


mongodb:// user:pass @ sample.host:27017 / ?maxPoolSize=20&w=majority
protocol
credentials
hostname/IP and port of instance(s)
connection options

- Specify MongoClient Settings
  - MongoClient Settings
  - Cluster Settings
  - Socket Settings
  - Connection Pool Settings
  - Server Settings
  - TLS/SSL Settings

Databases and Collections

- MongoDB organizes data into a hierarchy of the following levels:
  - Databases
  - Collections
  - Documents
- Databases are the top level of data organization in a MongoDB instance.
- Databases are organized into collections which contain documents.
- Documents contain literal data such as strings, numbers, and dates as well as other (embedded) documents.

Databases and Collections

- Access a Database
  - Use the getDatabase() method of a MongoClient instance to access a MongoDatabase in a MongoDB instance.
- Access a Collection
  - Use the getCollection() method of a MongoDatabase instance to access a MongoCollection in a database of your connected MongoDB instance.
- Get a List of Collections
  - Use the listCollectionNames() method of a MongoDatabase instance to query for a list of collections in a database
- Create a Collection: Use the createCollection() method of a MongoDatabase instance
- Drop a Collection: Use the *drop()* method of a collection from the database

Data Formats

- Document Data Format: BSON
- Document Data Format: Extended JSON
- Documents
- Document Data Format: POJOs
- Document Data Format: Records
- POJO Customization
- Codecs

Data Formats

- Document Data Format: BSON
  - BSON, or Binary JSON, is the data format that MongoDB uses to organize and store data.
  - This data format includes all JSON data structure types and adds support for types including dates, different size integers, ObjectIds, and binary data.
  - BSON Types: https://www.mongodb.com/docs/manual/reference/bson-types/

Data Formats

- Documents
  - A MongoDB document is a data structure that contains key/value fields in binary JSON (BSON) format.
  - You can use documents and the data they contain in their fields to store data as well as issue commands or queries in MongoDB.
  - The following classes that help you access and manipulate the BSON data in documents:
    - Document *(package org.bson)*
    - BsonDocument *(package org.bson)*
    - JsonObject *(package org.bson.json)*
    - BasicDBObject *(package com.mongodb)*

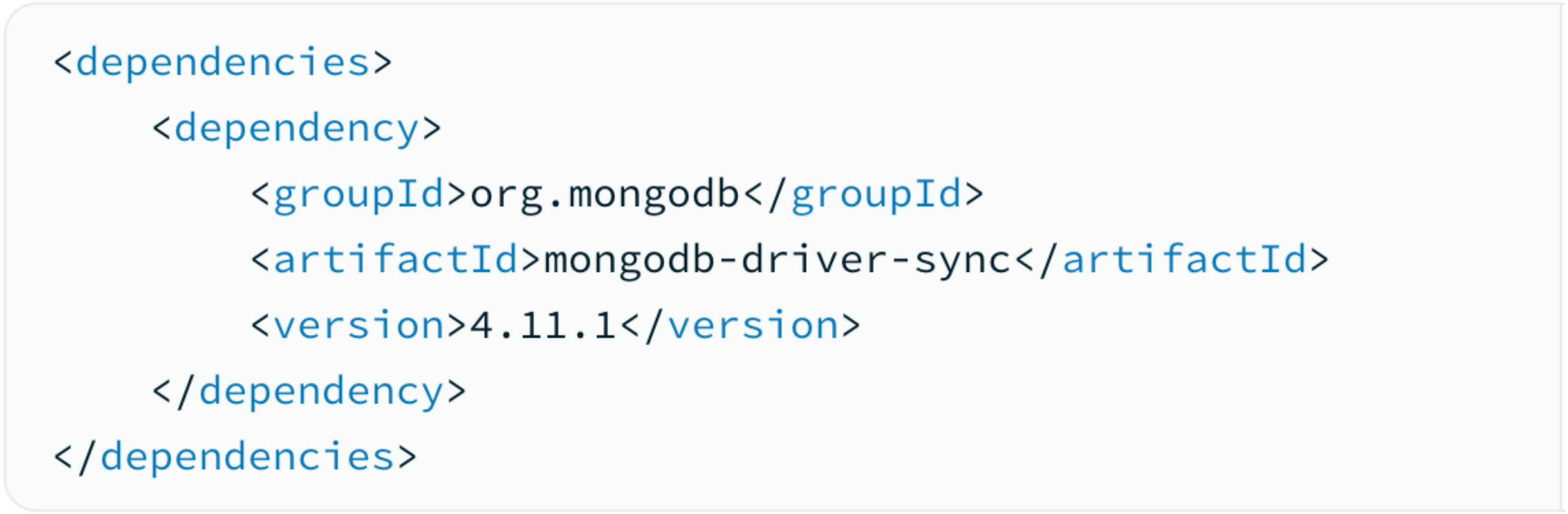
Data Formats

- Documents
  - *The following table for mappings between frequently-used BSON and Java types:*

BSON type	Java type	BSON type	Java type
Array	java.util.List	Int32	java.lang.Integer
Binary	org.bson.types.Binary	Int64	java.lang.Long
Boolean	java.lang.Boolean	Null	null
Date	java.util.Date	ObjectId	org.bson.types.ObjectId
Document	org.bson.Document	String	java.lang.String
Double	java.lang.Double	Decimal128	org.bson.types.Decimal128

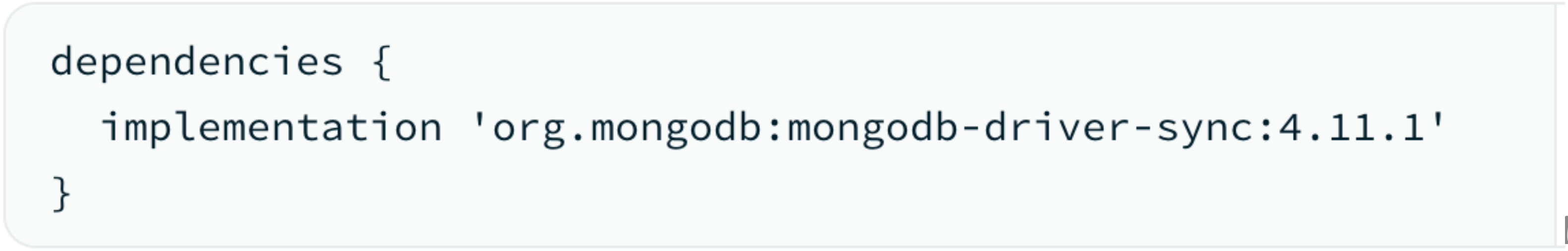
Add MongoDB as a Dependency- Sync driver

- Maven: add the following to your pom.xml dependencies list



```
<dependencies>
    <dependency>
        <groupId>org.mongodb</groupId>
        <artifactId>mongodb-driver-sync</artifactId>
        <version>4.11.1</version>
    </dependency>
</dependencies>
```

- Gradle: add the following to your build.gradle dependencies list



```
dependencies {
  implementation 'org.mongodb:mongodb-driver-sync:4.11.1'
}
```

Connect MongoDB – Sync driver (1)

- Without authentication
  - Connection String:

```
ConnectionString connectionString = new
                                                   ConnectionString("mongodb://localhost:27017/");
MongoClient mongoClient = MongoClients.create(connectionString);
```

  - MongoClientSettings:

```
ServerAddress seed = new ServerAddress("localhost", 27017);
MongoClientSettings settings = MongoClientSettings.builder()
            .applyToClusterSettings(builder ->
                                                   builder.hosts(Arrays.asList(seed)))
            .build();
MongoClient mongoClient = MongoClients.create(settings);
```

Connect MongoDB – Sync driver (2)

- Authentication enable
  - Connection String:

```
MongoClient mongoClient =  MongoClients
.create("mongodb://<username>:<password>@<hostname>:<port>/?authSource=<aut
henticationDb>");
```

- username - your MongoDB username.
- password - your MongoDB user's password.
- hostname - network address of your MongoDB deployment, accessible by your client.
- port - port number of your MongoDB deployment.
- authenticationDb - MongoDB database that contains your user's authentication data. If you omit this parameter, the driver uses the default value admin.

Connect MongoDB – Sync driver (3)

- Authentication enable
  - MongoCredential:

```
MongoCredential credential =   MongoCredential
      .createCredential("<username>", "<authenticationDb>", "<password>");

MongoClient mongoClient =
                  MongoClients.create(MongoClientSettings.builder()
                     .applyToClusterSettings(builder ->
                                    builder.hosts(Arrays.asList(
                           new ServerAddress("<hostname>", <port>))))
                     .credential(credential)
                     .build());
```

Get all databases

- Get all databases

```
MongoIterable<String> dbNames = mongoClient.listDatabaseNames();
dbNames.iterator().forEachRemaining(t -> {System.out.println(t);});

ListDatabasesIterable<Document> databases = mongoClient.listDatabases();
databases.iterator().forEachRemaining(db ->
                                        { System.out.println(db.get("name")); });
```

- Get specific database

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

Get collections

- Get all collections

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

MongoIterable<String> collectionNames = database.listCollectionNames();

ListCollectionsIterable<Document> collections=database.listCollections();
```

- Get specific collection

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

- Create a collection

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

Query (1)

- Get all records

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

collection.find()
          .iterator()
          .forEachRemaining(doc -> {System.out.println(doc);});
```

- Get one record

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

Document doc = collection.find()
                         .first();
```

Query (2)

- Filter criteria

```
import static com.mongodb.client.model.Filters.eq;

// Creates instructions to project two document fields
Bson projectionFields = Projections.fields(
                              Projections.include("title", "imdb"),
                              Projections.excludeId());

// Retrieves the first matching document, applying a projection and a
descending sort to the results
collection.find(eq("title", "The Room"))
          .projection(projectionFields)
          .sort(Sorts.descending("imdb.rating"))
          .iterator()
          .forEachRemaining(doc -> {System.out.println(doc);});
```

Insert (1)

- Insert a Document object

```
MongoCollection<Document> collection = database.getCollection("movies");
try {
        // Inserts a sample document describing a movie into the collection
        Document movie = new Document()
                      .append("_id", new ObjectId())
                      .append("title", "Ski Bloopers");

        InsertOneResult result = collection.insertOne(movie);

        // Prints the ID of the inserted document
        System.out.println("Success! Inserted document id: " +
        result.getInsertedId());

        // Prints a message if any exceptions occur during the operation
} catch (MongoException me) {
        System.err.println("Unable to insert due to an error: " + me);
}
```

Insert (2)

- Insert a BasicDBObject object

```
MongoCollection<BasicDBObject> collection =
                   database.getCollection("movies", BasicDBObject.class);

Map<String, String> map = new HashMap<>();
map.put("_id", (new ObjectId()).toString());
map.put("title", "Ski Bloopers");

try {
    // Inserts a sample document describing a movie into the collection
    InsertOneResult result = collection.insertOne(new
BasicDBObject(map));

    // Prints the ID of the inserted document
    System.out.println("Success! Inserted document id: " +
result.getInsertedId());
    // Prints a message if any exceptions occur during the operation
} catch (MongoException me) {
    System.err.println("Unable to insert due to an error: " + me);
}
```

Insert (3)

- Insert many Document object

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

// Creates two sample documents containing a "title" field
List<Document> movieList = Arrays.asList(
            new Document().append("title", "Short Circuit 3"),
            new Document().append("title", "The Lego Frozen Movie"));

try {
     // Inserts sample documents describing movies into the collection
     InsertManyResult result = collection.insertMany(movieList);

     // Prints the IDs of the inserted documents
     System.out.println("Inserted document ids: " +
result.getInsertedIds());

// Prints a message if any exceptions occur during the operation
} catch (MongoException me) {
   System.err.println("Unable to insert due to an error: " + me);
}
```

Update (1)

- Update one Document object

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

// Filter object to update
Document query = new Document().append("title", "Cool Runnings 2");

// Creates instructions to update the values of three document fields
Bson updates = Updates.combine(
                                Updates.set("runtime", 99),
                                Updates.addToSet("genres", "Sports"),
                                Updates.currentTimestamp("lastUpdated"));
// Instructs the driver to insert a new document if none match the query
UpdateOptions options = new UpdateOptions().upsert(true);

try {
   UpdateResult result = collection.updateOne(query, updates, options);

   System.out.println("Modified document count: " +
result.getModifiedCount());
   System.out.println("Upserted id: " + result.getUpsertedId());

} catch (MongoException me) {
   System.err.println("Unable to update due to an error: " + me);
}
```

Update (2)

- Find and Update Document object

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

// Filter object to update
Document query = new Document().append("title", "Cool Runnings 2");

// Creates instructions to update the values of three document fields
Bson updates = Updates.combine(
                                Updates.set("runtime", 99),
                                Updates.addToSet("genres", "Sports"),
                                Updates.currentTimestamp("lastUpdated"));

FindOneAndUpdateOptions optionAfter = new
                              FindOneAndUpdateOptions()
                                   .returnDocument(ReturnDocument.AFTER);

collection.findOneAndUpdate(query, updates, optionAfter);
```

Delete (1)

- Delete one Document object

```java
MongoDatabase database = mongoClient.getDatabase("movies");

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

Bson query = eq("title", "The Garbage Pail Kids Movie");

try {
   DeleteResult result = collection.deleteOne(query);

   System.out.println("Deleted document count: " +
result.getDeletedCount());

// Prints a message if any exceptions occur during the operation
} catch (MongoException me) {
   System.err.println("Unable to delete due to an error: " + me);
}
```

Delete (2)

- Find and Delete Document object

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

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

Bson query = eq("title", "The Garbage Pail Kids Movie");

try {
   Document doc = collection.findOneAndDelete(query);

   System.out.println(doc);

// Prints a message if any exceptions occur during the operation
} catch (MongoException me) {
   System.err.println("Unable to delete due to an error: " + me);
}
```

Delete (3)

- Delete many document object

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

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

Bson query = gte("title", "The Garbage Pail Kids Movie");

try {
   DeleteResult result = collection.deleteMany(query);

   System.out.println(result);

   // Delete the entire collection and its metadata (indexes, chunk
metadata, etc).
   collection.drop();


// Prints a message if any exceptions occur during the operation
} catch (MongoException me) {
   System.err.println("Unable to delete due to an error: " + me);
}
```

Using POJO (Plain old Java Object)

- POJO: is an ordinary Java object, not bound by any special restriction.
- 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 the CodecRegistry.
  - In a MongoClient via MongoClientSettings
  - In a MongoDatabase via its withCodecRegistry method
  - In a MongoCollection via its withCodecRegistry method
- Ref: https://mongodb.github.io/mongo-java-driver/4.11/driver/getting-started/quick-start-pojo/

Configure the Driver for POJOs (1)

- To configure the driver to use POJOs, you must specify the following components:
  - PojoCodecProvider: instance that has codecs that define how to encode and decode the data between the POJO format and BSON. The provider also specifies which POJO classes or packages that the codecs apply to.
  - CodecRegistry: instance that contains the codecs and other related information.
  - MongoDatabase or MongoCollection: instance that is configured to use the CodecRegistry.
  - MongoCollection: instance that is created with the POJO document class bound to the TDocument generic type.

Configure the Driver for POJOs (2)

- Using the PojoCodecProvider.builder() to create and configure a CodecProvider

```
// Configure the PojoCodecProvider
CodecProvider pojoCodecProvider =
PojoCodecProvider.builder().automatic(true).build();

// Add the PojoCodecProvider instance to a CodecRegistry. The CodecRegistry
allows you to specify one or more codec providers to encode the POJO data
CodecRegistry pojoCodecRegistry =
fromRegistries(getDefaultCodecRegistry(),fromProviders(pojoCodecProvider));

// Configure the MongoDatabase or MongoCollection instance to use the codecs
in the CodecRegistry.
MongoDatabase database = mongoClient.getDatabase("movie")
                                    .withCodecRegistry(pojoCodecRegistry);

// Pass your POJO class to your call to getCollection()
MongoCollection<Movie> collection = database.getCollection("movie",
Movie.class);
```

Configure the Driver for POJOs (3)

- Include the following static imports before your class definition

```
import static
com.mongodb.MongoClientSettings.getDefaultCodecRegistry;
import static
org.bson.codecs.configuration.CodecRegistries.fromProviders;
import static
org.bson.codecs.configuration.CodecRegistries.fromRegistries;
```

Using POJO - Perform CRUD Operations

```
// Insert two Movie instances
Movie movie1 = new Movie("title 1", "Cool Runnings 1");
Movie movie2 = new Movie("title 2", "Cool Runnings 2");
collection.insertMany(Arrays.asList(movie1, movie2));

// Update a document
collection.updateOne(
       Updates.addToSet("genres", "Sports")
);

// Delete a document
collection.deleteOne(Filters.eq("title", "The Room"));

// Return and print all documents in the collection
List<Movie> movies = new ArrayList<>();
collection.find().into(movies);

System.out.println(movies);
```

Aggregation

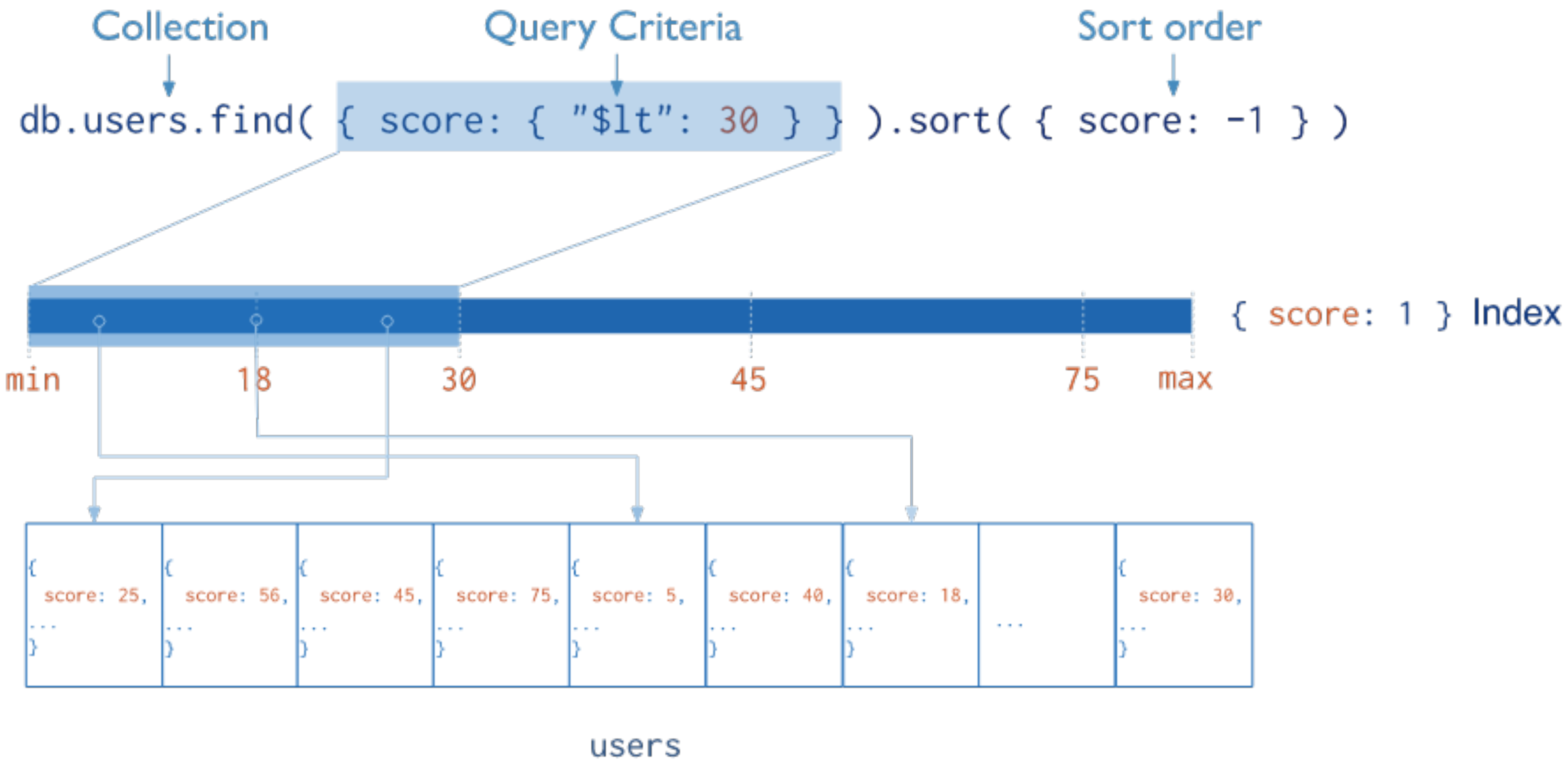
- Ref: https://www.mongodb.com/docs/drivers/java/sync/current/fundamentals/aggregation/
- Aggregation operations to perform the following actions:
  - Perform find operations
  - Rename fields
  - Calculate fields
  - Summarize data
  - Group values

```
collection.aggregate(
         Arrays.asList(
            Aggregates.match(Filters.eq("categories", "Bakery")),
            Aggregates.group("$stars", Accumulators.sum("count", 1))
         )
// Prints the result of the aggregation operation as JSON
).forEach(doc -> System.out.println(doc.toJson()));
```

MongoDB - Indexes

- Indexes support the efficient execution of queries in MongoDB. Without indexes, MongoDB must perform a collection scan, i.e. scan every document in a collection, to select those documents that match the query statement.
- If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect.
- MongoDB defines indexes at the collection level and supports indexes on any field or sub-field of the documents in a MongoDB collection.

MongoDB - Indexes



MongoDB - Indexes

Default _id Index

- MongoDB creates a unique index on the _id field during the creation of a collection.

Create an Index

- Creates indexes on collections:

```
db.collection.createIndex( <keys>, <options> )
```

Options:

  - An ascending index: 1
  - A descending index: -1

MongoDB - Indexes

Index Types (1)

- **Single Field:** MongoDB supports the creation of user-defined ascending/descending indexes on a single field of a document.
- **Compound Indexes:** MongoDB supports compound indexes, where a single index structure holds references to multiple fields within a collection's documents.
- **Multikey Indexes:** To index a field that holds an array value, MongoDB creates an index key for each element in the array.

MongoDB - Indexes

Index Types (2)

- **Text Indexes:** MongoDB provides text indexes to support text search queries on string content.

  To create index on a field that contains a string or an array of string elements, include the field and specify the string literal “text” in the index document.

  Ex: `db.people.createIndex( {firstname: "text" } )`

- **Wildcard Indexes:** MongoDB 4.2 introduces wildcard indexes for supporting queries against unknown or arbitrary fields.
  - Create a wildcard index on a field:

    `db.collection.createIndex( { "fieldA.$**" : 1 } )`
  - Create a Wildcard Index on All Fields:

    `db.collection.createIndex( { "$**" : 1 } )`

MongoDB - Indexes

Index Properties

- **Unique Indexes:** A unique index ensures that the indexed fields do not store duplicate values.

  Create a Unique Index:

  ```
  db.collection.createIndex( <keys>, { unique: true } )
  ```

- **Partial Indexes:** Partial indexes only index the documents in a collection that meet a specified filter expression.

  To create a partial index, use **db.collection.createIndex()** method with the partialFilterExpression option.

  ```
  Ex: db.restaurants.createIndex(
     { cuisine: 1, name: 1 },
     { partialFilterExpression: { rating: { $gt: 5 } } }
  )
  ```

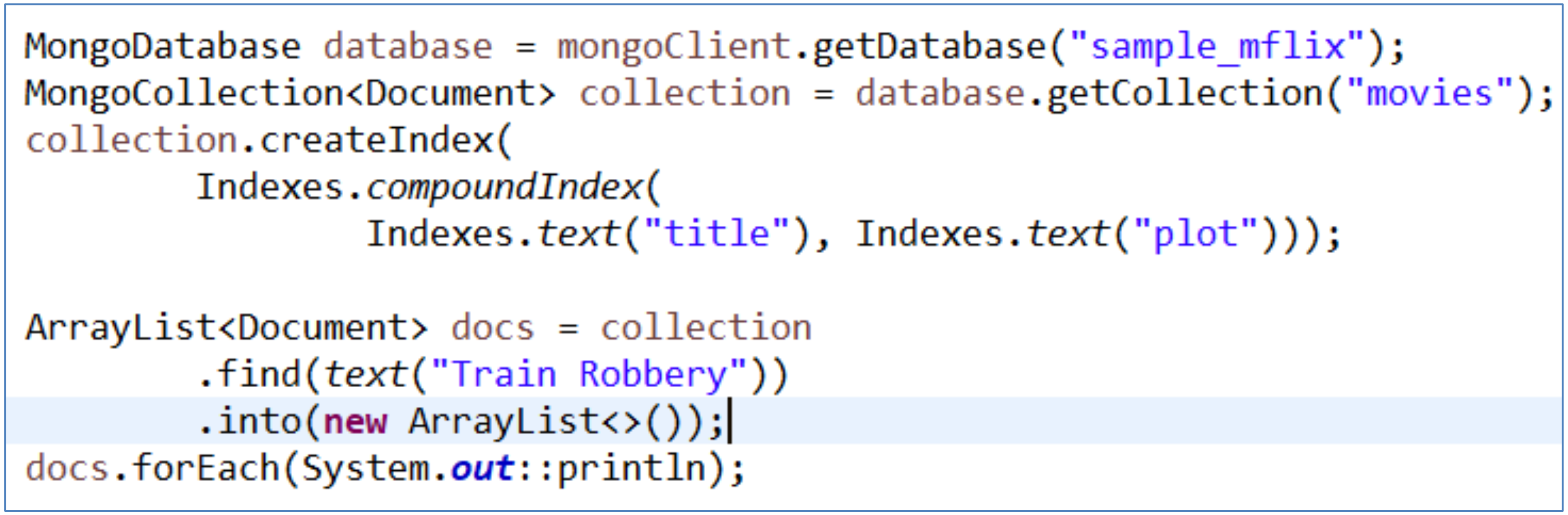
MongoDB - Indexes

Index Properties

For example, the following operation creates a compound index that indexes only the documents with a rating field greater than 5.

```
db.restaurants.createIndex(
   { cuisine: 1, name: 1 },
   { partialFilterExpression: { rating: { $gt: 5 } } }
)
```

MongoDB - Indexes



```
MongoDatabase database = mongoClient.getDatabase("sample_mflix");
MongoCollection<Document> collection = database.getCollection("movies");
collection.createIndex(
        Indexes.compoundIndex(
                Indexes.text("title"), Indexes.text("plot")));

ArrayList<Document> docs = collection
        .find(text("Train Robbery"))
        .into(new ArrayList<>());
docs.forEach(System.out::println);
```

MongoDB - Indexes

Manage Indexes

- View Existing Indexes: db.collection.getIndexes()
- Remove Indexes:
  - Remove Specific Index: db.collection.dropIndex()
  - Remove All Indexes: db.collection.dropIndexes()

Ref:

- https://www.mongodb.com/docs/drivers/java/sync/current/fundamentals/indexes/

Transactions in MongoDB

- What is a Transaction?

  A single unit of logic composed of multiple different database operations, which exhibits the following properties:

  - Atomic: either completes in its entirety or has no effect whatsoever (rolls back and is not left only partially complete)
  - Consistent: each transaction observes the latest current database state in the correct write ordering
  - Isolated: the state of an inflight transaction is not visible to other concurrent inflight transactions (and vice versa)
  - Durable: changes are persisted and cannot be lost if there is a system failure

Transactions in MongoDB

Transaction Syntax:

```
with client.start_session() as s:
    s.start_transaction()
        // Transaction Body
    s.commit_transaction()
```

Transactions in MongoDB

- Transaction steps:
  - Start a client session.
  - Define the transaction options (optional).
  - Define the sequence of operations to perform inside the transactions.
  - Start the transaction by using the ClientSession's `startTransaction()` method.
  - Finally, close the client session.

Creating MongoDB Transactions in Java Applications

- Example: Canceled a product of Order with orderId, productId
- Solution:
  - *Step 1:* Start a client session

    ```
    ClientSession session = mongoClient.startSession();
    ```
  - *Step 2:* Start a transaction from client session.

    ```
    session.startTransaction();
    ```
  - *Step 3:* Define the operations to be performed within the transaction.
    - *Step 3.1:* Get the Order using the orderId.
    - *Step 3.2:* Get the OrderDetail using the orderId and productId.
    - *Step 3.3:* Remove the OrderDetail identified by the orderId and productId from the current class.
    - *Step 3.4:* Update total of Order identified by the orderId
    - *Step 3.5:* Register the Order to Order History collection.

    Note: Each operation in the transaction must be associated with the session (i.e. pass in the session to each operation).
  - *Step 4:* has 2 cases
    - *Success → commit transaction:* `session.commitTransaction();`
    - *Failure (or has error) → rollback transaction:* `session.abortTransaction()`
  - *Step 5:* End the client session.

    ```
    session.close();
    ```

Reactive Streams Programming

- What is Reactive Streams?

  http://www.reactive-streams.org/

  Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure.

Reactive Streams Programming

Core Features of Reactive Streams Programming

1. New Programming Paradigm
2. Asynchronous and non blocking
3. Functional Style code
4. Data Flow as event driven stream
5. Backpressure on data streams

Reactive Streams specification

- Reactive Streams specification for Stream-oriented libraries for the JVM
  - process a potentially unbounded number of elements in sequence,
  - asynchronously passing elements between components with mandatory non-blocking backpressure.
- API Components:
  - Publisher
  - Subscriber
  - Subscription
  - Processor

Reactive Streams specification

```
public interface Publisher<T> {
    void subscribe(Subscriber<? super T> subscriber);
}
```

```
public interface Subscriber<T> {
    void onSubscribe(Subscription sub);
    void onNext(T item);
    void onError(Throwable ex);
    void onComplete();
}
```

```
public static interface Subscription {
    public void request(long n);
    public void cancel();
}
```

```
public static interface Processor<T,R>
                                          extends Subscriber<T>, Publisher<R> {
}
```

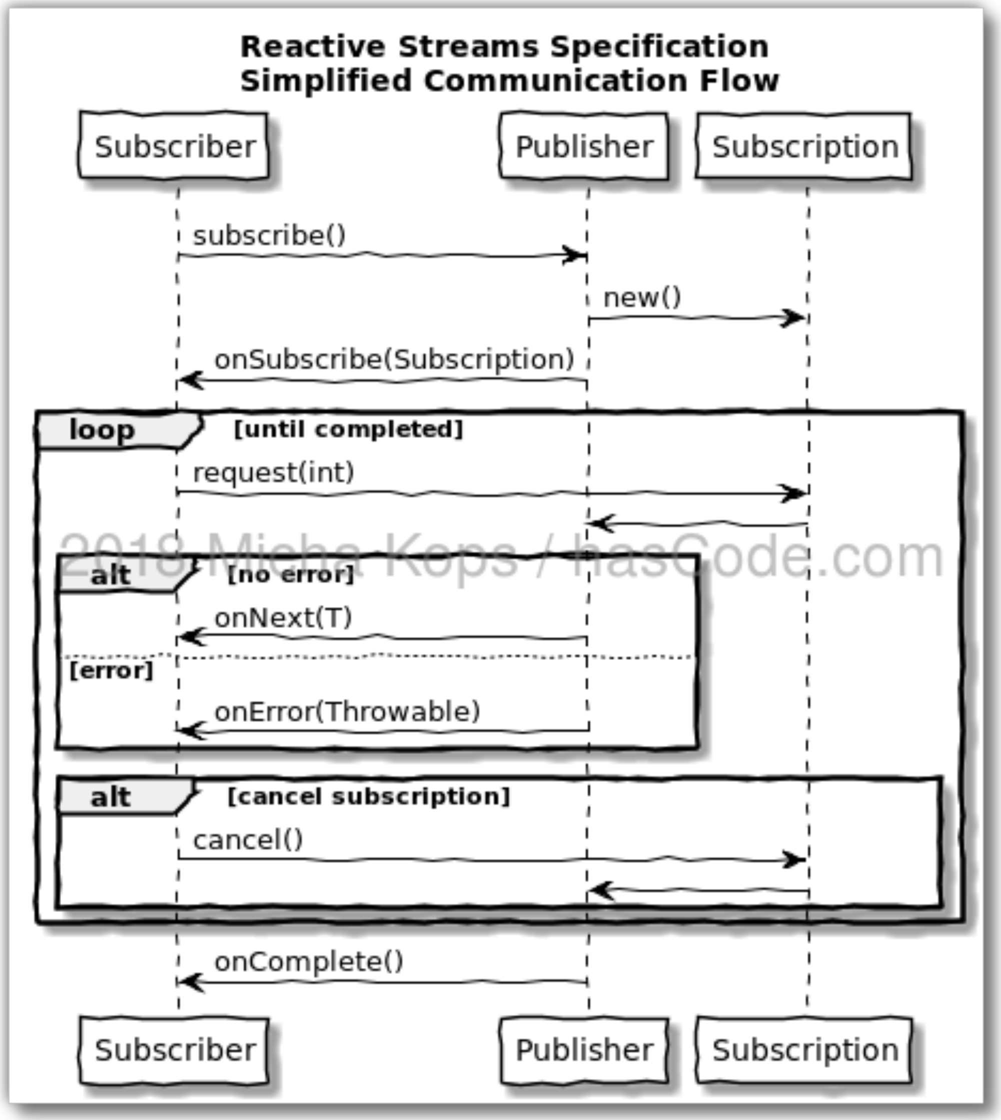
Reactive Streams specification

- A Publisher is a provider of a potentially unbounded number of sequenced elements, publishing them according to the demand received from its Subscriber(s).
- In response to a call to Publisher.subscribe(Subscriber) the possible invocation sequences for methods on the Subscriber are given by the following protocol:

  onSubscribe onNext (onError | onComplete)?*

- A Subscriber MUST signal demand via Subscription.request(long n) to receive onNext signals.
- Subscription.request and Subscription.cancel MUST only be called inside of its Subscriber context
- The Subscription MUST allow the Subscriber to call Subscription.request synchronously from within onNext or onSubscribe

Reactive Streams specification



Reactive Streams in Java 9

- Interfaces in java.util.concurrent.Flow
- SubmissionPublisher standalone bridge to Reactive Streams
- Tie-ins to CompletableFuture and Stream

MongoDB Java Reactive Streams (1)

Ref:

- https://www.mongodb.com/docs/drivers/reactive-streams/
- https://mongodb.github.io/mongo-java-driver/4.11/driver-reactive/getting-started/quick-start/

- Installation
- Make a Connection
- Access a Database
- Access a Collection
- Data Formats
- Insert a Document / Insert Multiple Documents
- Query the Collection
- Update a Single Document / Update Multiple Documents
- Delete a Single Document / Delete All Documents

MongoDB Java Reactive Streams (2)

- Indexes
- Aggregation Framework
- Text Search
- Builders
- Transactions