Java Connectivity for MongoDB

Overview

Java connectivity of MongoDB is supported by the com.mongodb package.

Working with MongoDB from Java is different than working with MongoDB from command line. Unlike other drivers (like JDBC, which allows for relational database connectivity), MongoDB Java driver wraps all commanding into an OO framework.

Connecting to the server

com.mongodb.MongoClient class. Connections to MongoDB servers are represented by instances of com.mongodb.MongoClient class.

The code to connect to a MongoDB server looks as follows:

```java
import com.mongodb.MongoClient;

public class foo {

    public String server = "cslvm31.csc.calpoly.edu"; // our Mongo Server location
    public int port = 27017; // default port

    public static void main(String args[]) {
        try {
            MongoClient client = new MongoClient(server, port); // establishing the connection
            // using a two-argument constructor
        } catch (Exception e);
    }
}
```

Constructors. MongoClient comes with the following constructors (we list only some of them here):
<table>
<thead>
<tr>
<th>Constructor</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MongoClient()</td>
<td>MongoClient()</td>
<td>connects to localhost on the default port (27017)</td>
</tr>
<tr>
<td>MongoClient(String host)</td>
<td>MongoClient(&quot;cslvm31&quot;)</td>
<td>connects to the given host on the default port (27017)</td>
</tr>
<tr>
<td>MongoClient(String host, int port)</td>
<td>MongoClient(&quot;cslvm31&quot;, 27017)</td>
<td>connects to the given host on the given port</td>
</tr>
</tbody>
</table>

There is a series of constructors that uses the com.mongodb.ServerAddress class instances as input parameters. ServerAddress instances can be constructed using the same three parameter sets as MongoClient instances:

ServerAddress(); // creates a ServerAddress instance for localhost with default port
ServerAddress(String host); // creates a ServerAddress instance for given host with default port
ServerAddress(String host, int port); // creates a ServerAddress instance for given host and port

For example, we can connect to "cslvm31" on port 27017 as follows:

```java
...
MongoClient client = MongoClient(new ServerAddress("cslvm31", 27017));
...
```

If one needs to connect to a MongoDB server with authentication on, the following constructors can be used:

```java
MongoClient(MongoClientURI uri); // pass all connection information in a URI
MongoClient(ServerAddress Addr, List<MongoCredential> credentialsList);
    // use MongoCredential objects
```

MongoClientURI class. com.mongodb.MongoClientURI is the MongoDB driver's implementation of the Universal Resource Locator. The general format for a URI string identifying MongoDB server access is:

```
mongodb://[username:password@]host1[:port1][,host2[:port2],...[,hostN[:portN]]][/[database][?options]
```

(all items in square brackets are optional; multiple hosts are used to create a replica set on the fly).

Some valid MongoDB URIs are shown below:

```java
mongodb://alex:abc123@cslvm31.csc.calpoly.edu:27017/alex
mongodb://@cslvm31/alex
mongodb://alex:abc123@cslvm31.csc.calpoly.edu:27017
```

The main constructor for MongoClientURI is:

```java
MongoClientURI(String uri)
```

The following fragment sets up an authenticated connection to a MongoDB server:

```java
... String uri = "mongodb://alex:abc123@cslvm31.csc.calpoly.edu:27017";
MongoClientURI mongoURI = MongoClientURI(uri);
MongoClient client = MongoClient(uri);
...
MongoCredentia l class. MongoDB driver uses `com.mongodb.MongoCredent ial` class to build a record storing login credentials for a MongoDB server.

The credentials are generated for a specific authentication protocol used by a MongoDB server for authentication. There are five authentication mechanisms used, and each has a static `createCredential()` method associated with them. The sixth method creates credentials for an unspecified mechanism:

<table>
<thead>
<tr>
<th>Credentials creation method</th>
<th>Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>static createCredential(String userName, String database, char[] password)</td>
<td>unspecified</td>
</tr>
<tr>
<td>static createGSSAPICredential(String userName)</td>
<td>GSSAPI SASL</td>
</tr>
<tr>
<td>static createMongoCRCredential(String userName, String database, char[] password)</td>
<td>Challenge-Response</td>
</tr>
<tr>
<td>static createX509Credential(String userName)</td>
<td>MongoDB X.509</td>
</tr>
<tr>
<td>static createPlainCredential(String userName, String source, char[] password)</td>
<td>PLAIN SASL</td>
</tr>
<tr>
<td>static createScramSha1Credential(String userName, String source, char[] password)</td>
<td>SCRAM-SHA-1 SASL</td>
</tr>
</tbody>
</table>

The following code connects to a database server that uses challenge-response mechanism.

```java
...
String host = "cslvm31";
String user = "alex";
char[] pass = { 'a', 'b', 'c', '1', '2', '3' };
String db = "alex";
MongoCredential login = MongoCredential.createCRCredential(user, db, pass);
MongoClient client = MongoClient(new ServerAddress(host), login);
...
```

MongoClient methods

MongoClient has the following relevant methods associated with it.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MongoClient.getDatabase(String databaseName)</td>
<td>analog of <code>use &lt;db&gt;</code> command.</td>
</tr>
<tr>
<td>MongoIterable&lt;String&gt; listDatabaseNames()</td>
<td>analog of <code>show dbs</code> command.</td>
</tr>
<tr>
<td>ListDatabasesIterable&lt;Document&gt; listDatabases()</td>
<td>returns an iterable collection of databases</td>
</tr>
</tbody>
</table>

Setting the database. All MongoDB operations occur inside a database. To select the working database, use the `getDatabase()` method. The following code connects to the `cslvm31` server and selects the database "alex".

```java
...
MongoClient client = MongoClient("cslvm31");
MongoDatabase db = client.getDatabase("alex");
...
```

Note: Under most circumstances, this is the only use for a MongoClient instance in your code.

Listing Databases. Sometimes you may want to perform certain action on each database in turn. There are two ways to access the list of databases.

- `listDatabaseNames()` returns an iterable list of Strings, with each string containing a name of one database.
- `listDatabases()` returns back a list of iterable documents, with each document (instance of `org.bson.Document` class) containing information about a specific database. The database document has the following format:
{ name: <dbName>,
  sizeOnDisk: <size>,
  empty: <emptyFlag>
}

Here, <dbName> is the name of the database, (e.g., "alex"), <size> is the amount of disk space the database takes (in bytes?), and <emptyFlag> is a true or false value depending on whether the database is empty or not.

To iterate over <Document> iterables, use the following template (templated parts are in double angle brackets):

```java
<<DocIterable>>.forEach(new Block<Document>() {
    @Override
    public void apply(final Document d) {
        << CODE GOES HERE >>
    }
});
```

The following code, for example, lists all database objects.

```java
MongoClient c = new MongoClient(new ServerAddress("cslvm31",27017));
MongoDatabase db = c.getDatabase("alex");
ListDatabasesIterable<Document> dbList = c.listDatabases();
dbList.forEach(new Block<Document>() {
    @Override
    public void apply(final Document d) {
        System.out.println(d);
    }
});
```

**MongoDatabase instances**

A single MongoDB database is represented in Java as an instance of a `com.mongodb.client.MongoDatabase` class (interface).

The following important methods are available in the `MongoDatabase` interface.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void createCollection(String name)</td>
<td>create a collection with given name</td>
</tr>
<tr>
<td>void drop()</td>
<td>drop the database</td>
</tr>
<tr>
<td>String getName()</td>
<td>return the name of the database</td>
</tr>
<tr>
<td>MongoCollection&lt;Document&gt; getCollection(String name)</td>
<td>return a MongoDB collection object for a given collection</td>
</tr>
<tr>
<td>MongoIterable&lt;String&gt; listCollectionNames()</td>
<td>return a list of collection names (as strings)</td>
</tr>
<tr>
<td>ListCollectionsIterable&lt;Document&gt; listCollections()</td>
<td>return a list of collection descriptor objects</td>
</tr>
<tr>
<td>Document runCommand(Bson command)</td>
<td>execute a MongoDB command</td>
</tr>
</tbody>
</table>

**Getting a collection object.** All MongoDB queries run on collections. MongoDB driver uses instances of `com.mongodb.client.MongoCollection` to represent MongoDB collections. The `getCollection()` method retrieves the collection instance. The following code retrieves the collection "foo".

```java
...
MongoClient client = MongoClient("cslvm31");
MongoDatabase db = client.getDatabase("alex");
MongoCollection<Document> foo = db.getCollection("foo");
..."
**Listing all collections.** If you need to run an operation on all collections in the database, you can use `listCollectionNames()` or `listCollections()` methods.

These methods work similarly to `listDatabaseNames()` and `listDatabases()`.

`listCollectionNames()` returns an iterable over a list of collection names (as `String` values). This is useful when paired with `getCollection()`. The following fragment goes over all collections and performs the same action on them.

```java
MongoClient client = MongoClient("cslvm31");
final MongoDatabase db = client.getDatabase("alex"); // need to make final to allow calling from inner class
MongoIterable<String> cNames = db.listCollectionNames();
cNames.forEach(new Block<String>() {
    @Override
    public void apply(final String s) {
        MongoCollection<Document> cl = db.getCollection(s); // grab the collection
        // RUN ANY CODE YOU WANT
        // HERE !
        System.out.print(s + ", ");
        System.out.println(cl.count()); // number of documents
    }
});
```

The specific output of this code on will be something like this:

```
foo, 6
alex.obj, 10
alex.dox, 5
```

`listCollections()` returns an iterable over a list of JSON documents, each representing data about a specific MongoDB collection. The document description of a collection has the following format:

```json
{
    name: <collectionName>
    options: <options>
}
```

Here, `<collectionName>` is the name of the collection and `<options>` is the list of options used when creating the collection. Usually this list will be empty.

The following code:

```java
MongoClient client = MongoClient("cslvm31");
MongoDatabase db = client.getDatabase("alex");
ListCollectionsIterable<Document> collections = db.listCollections();
collections.forEach(new Block<Document>() {
    @Override
    public void apply(final Document col) {
        System.out.println(col);
    }
});
```

produces the following output:

```
{ "name": "foo", "options": [] },
{ "name": "alex.obj", "options": [] },
{ "name": "alex.dox", "options": [] }.
```

1This uses the state of the alex database on the server at the time of creating this handout.
Work with Collections

At the heart of all work with MongoDB are the requests for information and data manipulation addressed to a collection. With the exception of the functionality available through the `MongoDatabase.runCommand()` method, all other data manipulation happens through the methods of the `MongoCollection` class.

Insert objects

```java
void insertOne(TDocument document) // insert one object
void insertMany(List<? extends TDocument> documents) // insert a list of objects
```

Delete objects

```java
DeleteResult deleteOne(Bson filter) // delete one object given a filter document
DeleteResult deleteMany(Bson filter) // delete all objects matching the filter document
```

Update objects

```java
UpdateResult updateOne(Bson filter, Bson update) // update one object that matches a filter document
UpdateResult updateMany(Bson filter, Bson update) // update all objects matching the filter document
UpdateResult replaceOne(Bson filter, Document replacement) // replace one object that matches filter
```

Find objects

```java
FindIterable<TDocument> find() // returns all objects in a collection
FindIterable<TDocument> find(Bson filter) // returns all objects that match the filter
```