Lab 4: MongoDB Application

Due date: January 29, 11:59pm.

Note: It is possbile that **Lab 5** will be handed out on *January 28*.

Lab Assignment

Assignment Preparation

This is a pair programming lab. Select your partner. I prefer if everyone works with a partner from their section. I will admit one team of three people to make sure everyone has a partner.

Application

For this lab we will spoof the work of a messaging service.

Essentially, a messaging service consists of two parts:

1. **Messaging Client.** A client program operated on the devices of service customers that they use to write a message and send it out. It also shows any messages created by the customer, or addressed/visible to them.
2. **Messaging Server.** The central, *distributed* service that receives messages from client program, stores them, and sends them to cilent programs upon demand.

As mentioned above, we will *spoof* both the messaging client and the messaging server. Both the client spoof and the server spoof will talk to our MongoDB server, and will insert, update and query the data.

Spoofing the Client Program.

You have already essentially created a spoof of the messenger client program: your ThghtShre JSON generator produces messages on demand. The actual ThghtShre client spoof will extend the Lab 1 program in a number of ways.

The ThghtShre client spoof is a Java program. The program shall perform the following actions.

CSR1. The program runs in a forever loop. As an option, each team can implement a way to kill the program, but in general when testing, we will simply kill the program from the OS.

CSR2. Input. The program takes as input the name of a `messenger configuration file`. The file will contain the client spoof configuration parameters.

CSR3. Configuration File. The client spoof configuration file is a file containing a single JSON object with the following format:

```
{
 "mongo": <MongoServer>,
 "port": <MongoPort>,
 "database": <DBName>,
 "collection": <CollectionName>,
 "monitor": <MonitorCollName>,
 "delay": <delayAmount>,
 "words": <wordFile>,
 "clientLog": <clientLogFile>,
 "serverLog": <serverLogFile>,
 "wordFilter": <queryWordFile>
}
```

All key-value pairs must be present, although some values may be set to `null` or empty string.

The configuration parameters are:

Parameter	Type	Explanation
`<MongoServer>`	`string`	Address of MongoDB server. Use `"localhost"` if empty or `null`
`<MongoPort>`	`integer`	Port of MongoDB server. Use 27017 if `null`
`<DBName>`	`string`	MongoDB database to connect to. Use `"test"` if empty.
`<CollectionName>`	`string`	MongoDB collection to store data in.
`<MonitorCollName>`	`string`	MongoDB collection to store monitor objects in. Must be different from `<CollectionName>`.
`<delayAmount>`	`integer`	Number of seconds between attempts to generate a message. Use 10 seconds if set to 0 or `null`.
`<wordFile>`	`string`	File containing the word list for message generation.
`<clientLogFile>`	`string`	File name of the output log file for the client program.
`<serverLogFile>`	`string`	File name of the output log file for the server program.
`<queryWordFile>`	`string`	File containing the information about the words of interest for the server.

Any parameter value that is not explicitly stated in the table above cannot be empty/null.

Note: See more about the *monitor* and monitor collection below.

CSR4. Startup. Upon startup, the program performs the following actions:

1. Reads the configuration file.
2. Estabilshes the connection with the MongoDB server specified in the config file.
3. Checks the contents of the collection `<CollectionName>` in the database `<DBName>`. Obtains the number of documents in the collection.
4. Prints startup diagnostics including:
    - Current timestamp
    - MongoDB server connection details.
    - Database and collection name.
    - Number of documents in the collection.

After performing these actions, the program starts the main loop.

CSR 5. Main loop. On each step of the loop, the program performs the following actions:

1. Wait `<delayAmount>` of seconds. You can do it deterministically and always wait that amount, or you can create a normal distribution with mean at `<delayAmount>` and standard deviation of `<delayAmount>/2` and draw from it.
2. Generate one message using you Lab 1 ThghtShre message generator.
3. Store your message generator in the `<CollectionName>` collection in the database `<DBName>`.
4. Pretty-print the generated message (message text, author, and other meta-data) together with a timestamp.

CSR 6. Additional checks. Every 40 cycles, your program, in addition to generating a message and storing it in the database, will retreive from the collection `<CollectionName>` the information about

- Total number of messages stored.
- Total number of messages written by the author of the last generated message.

This information shall be pretty-printed out.

CSR 7. Log. Your program, in addition printing output to screen shall also send it to a log file. The log file must be maintained in a way that ensures that the file has all the log records even if the program is terminated. (i.e., make sure to flush your output every time you write something out to the file).

Spoofing the Server.

The server spoof is also a Java program that run in an infinite loop and monitors what is going on in the system.

SSR1. The server spoof takes as an input parameter the configuration file name. For the sake of simplicity we maintain one configuration file (same as for the client spoof). Some parameters in the file are only used for the client, some - only for the server.

SSR2. Startup. Upon server spoof startup, the program shall perform the following actions.

1. Read the configuration file.
2. Estabilsh the connection with the MongoDB server specified in the config file.
3. Check the contents of the collection `<CollectionName>` in the database `<DBName>`. Obtain the number of documents in the collection.
4. Wipe the contents of the collection `<MonitorCollName>` in the database `<DBName>`. This is the collection where the server will push collected statistics.
5. Read the contents of the `<queryWordFile>` into an appropriate data structure. `<queryWordFile>` should have the same format as `<wordFile>` - each line contains a single word. The file represents words of interest that the server spoof will monitor for.
6. Print startup diagnostics including:
   - Current timestamp
   - MongoDB server connection details.
   - Database and collection name.
   - Number of documents in the collection.

SSR3. Main Loop. The server spoof runs in an infinite loop. On each loop iteration, the program shall perform the following actions:

1. Sleep for `3 * <delayAmount>` seconds. (Unlike the client spoof, the server spoof shall check upon things in regular intervals).

2. Send a number of information requests (see below) to the MongoDB server and receive the responses.
3. From the obtained responses build a *monitor JSON document* and insert that document into the `<MonitorCallName>` monitor collection.
4. Pretty print the information about the current status of the system.
5. Search in the collection `<CollectionName>` for *new* documents that might contains any of the words from the `<queryWordFile>`.
6. Print out any of the messages containing the words of interest.
7. Update the totals in the monitor data collection.

Parts of this process are explained in more detail below.

SSR4. Information Requests. The server spoof shall obtain the following information about the current state of the message collection.

- Total number of unique messages.
- Total number of unique users to sent at least one message.
- Total number of messages inserted since last checkpoint.
- Total number of messages for each status (`public`, `protected`, etc..).
- Total number of messages for each destination (`self`, `all`, etc..). Count all messages addressed to a specific user as a single destination.

SSR5. Monitor JSON document. Based on the information obtained about the current status of the message collection, your program shall prepare a JSON object storing this information. The format of the JSON object is:

```
{
 "time": <currentTimeTimestamp>,
 "messages": <nMessages>,
 "users": <nUniqueUsers>,
 "new": <nNewMessages>,
 "statusStats" : [{<status1>: <nMessagesForStatus1>}, ...,{<statusK>:<nMessagesForStatusK>}],
 "recepientStats":  [{<recepient1>: <nMessagesForRecepient1>}, ...,{<recepientM>:<nMessagesForRecepientK>}]
}
```

Here:

Paremeter	Meaning
`<currentTimeTimestamp>`	The timestamp at which the monitor record is created.
`<nMessages>`	Total number of messages in message collection.
`<nUniqueUsers>`	Total number of unique users (user ids) that authored at least one message.
`<nNewMessages>`	Number of messages added to the collection since the last check-in.
`<status1>,...,<statusK>`	`"public"`, `"private"`, `"protected"` (in any order).
`<nMessagesForStatusX>`	number of messages in the collection with the given status (one per status).
`<recepient1>,...,<recepientM>`	`"self"`, `"all"`, `"self"`, `"subscribers"`, `"userId"`
`<nMessagesForRecepientX>`	number of messages addressed to a specific category of receptients (one per category).

The monitor JSON document shall then be inserted into the Monitor collection as well as saved to the server log file.

SSR6. Pretty Printing status. The status information obtained from the message collection and placed into the monitor collection shall also be pretty-printed to the terminal. The format of the printout is left up to individual teams. You shall print all the information stored in the monitor JSON object in a clear, readable way.

SSR7. Word search. After obtaining the current message collection statistics, the server shall conduct a search for ***newly inserted messages*** that contain any of the words in the `<QueryWordFile>`. For each query word, the server shall make a request, retrieve search results, and, if non-empty, print out the keyword and all the *new* documents (i.e., documents inserted afte the previous checkpoint) that contain it.

Note: you are only printing this information on screen and into the log file. The monitor collection shall remain intact.

Note 2. IMPORTANT: Because the server can start with a non-empty message collection (and because such a message collection may be very large), your program shall conduct word searches **starting on its SECOND checkpoint.**

SSR8. Update the monitor totals record. In addition to *checkpoint* JSON objects, the monitor collection shall contain one object which we refer to as the monitor totals record. This JSON object shall have the following format:

```
{
  "recordType": "monitor totals",
  "msgTotals": [<msgCnt1>,...,<msgCntN>],
  "userTotals": [<userCnt1>,...,<userCntN>],
  "newMsgTotals": [<newMsgCnt1>,...,<newMsgCntN>],
}
```

Here:

Parameter	Explanation
`<msgCntX>`	total number of messages recorded in checkpoint number X.
`<userCntX>`	total number of users recorded in checkpoint number X.
`<newMsgCntX>`	total number of new messages recorded in checkpoint number X.

(Note: the `"recordType"` key-value pair is there so that you could retrieve this object easily from the monitor collection.)

On each iteration of the loop, the server program shall:

- Retrieve the monitor totals record from the monitor collection.

- Add the new counts for the total number of messages, number of unique users and number of new messages to the appropriate arrays in the record.
- Update the monitor totals record in the monitor collection.

SSR9. General notes. When developing the server software, please note:

1. You many need to request more information from the two data collections than what is explicitly specified. You will need to figure out what queries to run.
2. You may want to store more objects in the monitor collection. Essentially, if you need to pass any data from one loop iteration to another, consider putting it in a document stored in the monitor collection. You cannot change the format of the documents we described above, but you can store documents *other* than the ones described above in the monitor collection.
3. Certain information needs may require a query (or more) to MongoDB, and then - finishing touches in Java. Do not hesitate to do so if needed. You want MongoDB to do most of your work, but you may finish off the queries inside your server program if you cannot figure out how to do certain things in MongoDB.

Submission

Name your programs `ClientSpoof.java` and ServerSpoof.java. Submit all the code you have developed for both programs. This requires resubmission of your modified/updated Lab 1-1 code. Submit one sample configuration file, one words file and one query words file you have used in your debugging.

All submitted files must contain your name on them.

Submit all your code in a single archive (zip or tar.gz).

Use `handin` to submit as follows:

Section 01:

```
$ handin dekhtyar lab04-01 <FILES>
```

Section 03:

```
$ handin dekhtyar lab04-03 <FILES>
```

Good Luck!