	Course	**Databases and Information Systems 2020**		
	Exercise Sheet	**5**		
	Points	–		
	Release Date	**9th June 2020**	Due Date	**23rd June 2020**

5 NoSQL: MongoDB

This exercise sheet comprises two parts: first, a tutorial to make you accustomed to MongoDB and, second, the implementation of some features of an application that works with MongoDB to store IMDb movie data and related tweets.

Note:

- You can find an unfinished Java project in the moodle, together with some datasets. Particularly helpful resources apart from our tutorial are:

  - MongoDB online documentation: `http://docs.mongodb.org/`

  - MongoDB Java API online documentation:
    `http://docs.mongodb.org/ecosystem/tutorial/getting-started-with-java-driver/`

  - miscellaneous MongoDB tutorials: `http://docs.mongodb.org/manual/tutorial/`

  - information about docker in general `https://docs.docker.com/get-docker/`

  - information about the mongo docker image: `https://hub.docker.com/_/mongo`

- In this tutorial we are using Docker as environment. If necessary, you have to adjust some operating system specific commands for your operating system and have to use the correct MongoDB version. Installation under native Linux, macOS or Windows should work without any limitations.

For your Report:

- Please summarize the results of task 5.1. Provide some screenshots or copy the console output of your commands. Show at least one example per operation (Insert, Query, Update, Delete, Indexing) Feel free to further familiarize with mongodb by using one of the official tutorials.

- For Task 5.2 please describe what you have done and provide some screenshots of your application, for example the results of queries in the webapp. Further, upload the working program (as a .zip-file) together with your report in the moodle.

5.1 Tutorial

5.1.1 Starting a MongoDB Instance

First, we start the mongoDB instance. The most convenient way is, in our opinion, using Docker which is available for all mayor systems: `https://docs.docker.com/get-docker/`.

To get the system running please load the material (zip-archive) from Moodle and extract it. Go into this directory using a terminal of your choice (bash, Powershell, etc). Make sure that you are in the same directory as the file "docker-compose.yaml". This file describes the environment we are about to set up. Make sure your docker-engine is running on your machine. (For example Docker Desktop on Windows or docker service on Linux. If in doubt, see the official docker documentation.) You can now start the docker MongoDB with:

```
docker-compose up -d
```

Please note that in this example no volumes are used, so inserted data is lost if you remove the container with "docker-compose down" or similar. But you can stop and start the instance with

```
docker-compose stop
docker-compsoe start
```

Under `http://localhost:8081` you will now find an overview of the MongoDB instance.

Alternatively you can install the MongoDB directly on your system. In this case the steps for the setup vary depending on your system and you have to adapt the commands in the console.

5.1.2 Inserting Data

You can connect to to the CLI with:

```
docker run -it --network mongo_default --rm mongo mongo --host mongo -u root -p password
```

By default, you are connected to a `test` database, but for this tutorial, switch to database `imdb`:

```
use imdb
```

Databases and collections in MongoDB are created implicitly while data is inserted. In this tutorial, you will create an `imdb` database with a collection of `films`.

You can list the available databases, . . .

```
show dbs
```

. . . but the `imdb` database does not exist, yet. There are also no collections, so far, . . .

```
show collections
```

. . . so create one by inserting a document:

```
db.films.insert({
  title: "Star Trek Into Darkness",
  year: 2013,
  genre: [
    "Action",
    "Adventure",
    "Sci-Fi",
    ],
  actors: [
    "Pine, Chris",
    "Quinto, Zachary",
    "Saldana, Zoe",
  ],
  releases: [
    {
      country : "USA",
      date : ISODate("2013-05-17"),
      prerelease: true
    },
    {
      country : "Germany",
      date : ISODate("2003-05-16"),
      prerelease: false
    }
```

```
  ]
})
```

As you can verify by calling `show collections` again, now there is a `films` collection.

You can list the contents of the newly created collection by calling the `find()` function:

```
db.films.find().pretty()
```

Alternatively you can visit `http://127.0.0.1:8081/db/imdb`

As you can see, now there is an `_id` field which is unique for every document.

Now insert some more films:

```
db.films.insert({
  title: "Iron Man 3",
  year: 2013,
  genre: [
    "Action",
    "Adventure",
    "Sci-Fi",
    ],
  actors: [
    "Downey Jr., Robert",
    "Paltrow, Gwyneth",
  ]
}) // no releases

db.films.insert({
  title: "This Means War",
  year: 2011,
  genre: [
    "Action",
    "Comedy",
    "Romance",
    ],
  actors: [
    "Pine, Chris",
    "Witherspoon, Reese",
    "Hardy, Tom",
  ],
  releases: [
    {
      country : "USA",
      date : ISODate("2011-02-17"),
      prerelease: false
    },
    {
      country : "UK",
      date : ISODate("2011-03-01"),
      prerelease: true
    }
  ]
})

db.films.insert({
  title: "The Amazing Spider-Man 2",
  year: 2014,
```

```
  genre: [
    "Action",
    "Adventure",
    "Fantasy",
    ],
  actors: [
    "Stone, Emma",
    "Woodley, Shailene"
  ]
}) // also no releases
```

5.1.3 Querying

Now query your collection. Have MongoDB return all films with title **"Iron Man 3"** by calling:

```
db.films.find({title: "Iron Man 3"})
```

Using `findOne` instead of `find` produces at most one result (in pretty format):

```
db.films.findOne({title: "Iron Man 3"})
```

Regular expressions can also be used to query a collection. In this tutorial, a short notation is used where the actual regular expression is bounded by slashes (/). The following call yields all movies that start with the letter T:

```
db.films.find({title: /^T/})
```

If you are only interested in certain attributes, you can use projection to thin out the produced result. While the selection criteria are given by the first argument of `find`, the projection is given by the second argument. An example:

```
db.films.find({title: /^T/}, {title: 1})
```

By default, the `_id` is part of the output, so you have to explicitly suppress it, if you don't want to have it returned by MongoDB:

```
db.films.find({title: /^T/}, {_id: 0, title: 1})
```

You can also use conditional operators, for example to perform range queries. The following returns the titles of all films starting with the letter T where the year attribute is greater than 2009 and less than or equal to 2011:

```
db.films.find({
  year: {
    $gt:  2009,
    $lte: 2011
  },
  title: /^T/
},
{ _id: 0,
  title: 1
})
```

For a logical disjunction of the selection criteria, use the *or* operator:

```
db.films.find({
  $or: [
      {year: {
          $gt:  2009,
          $lte: 2011
      }},
      {title: /^T/}
    ]
},
{ _id: 0,
  title: 1
})
```

There are also some options that can be appended to the regular expression, e.g. `i` to achieve case-insensitivity. The following call returns the titles of all movies whose title contains lowercase t, . . .

```
db.films.find({title: /t/}, {_id: 0, title: 1})
```

. . . whereas the following call also returns titles that contain a T (uppercase):

```
db.films.find({title: /t/i}, {_id: 0, title: 1})
```

You can query for exact matches in lists, . . .

```
db.films.find({genre: "Adventure"}, {_id: 0, title: 1, genre: 1})
```

. . . but you can also query for partial matches:

```
db.films.find({genre: /^A/}, {_id: 0, title: 1, genre: 1})
```

There are also more complex operators for more complex selection criteria, e.g. the `$all` operator. The following call prints the title and actors of every movie for which each of two given regular expressions matches at least one of its actors:

```
db.films.find({actors: {$all: [/pine/i, /zachary/i]}}, {_id: 0, title: 1, actors: 1})
```

In contrast, the `$nin` operator checks for the lack of matching values, i.e. actor names that do not match either one of the given regular expressions:

```
db.films.find({actors: {$nin: [/pine/i, /zachary/i]}}, {_id: 0, title: 1, actors: 1})
```

The `$exists` operator can be used to check for the existence of an attribute, e.g. to select only movies with undefined **releases**:

```
db.films.find({releases: {$exists: false}}, {_id:0, title: 1})
```

In MongoDB, it is also possible to query nested data, i.e. subdocuments. The following returns the title and releases of every movie that is known to be released in the UK:

```
db.films.find({'releases.country': "UK"}, {_id:0, title: 1, releases: 1})
```

Please note that you have to use quotes to address nested fields.

Applying more complex selection criteria on a nested document, however, is a little tricky. For example, if you wanted MongoDB to return all movies that had their prerelease in the USA, you might try something like this:

```
db.films.find({'releases.country': "USA", 'releases.prerelease': true}, {_id:0, title: 1,
    releases: 1})
```

However, This Means War is also returned, but was prereleased in the UK. The call above actually returns all movies that have some prerelease *or* were released in the USA. To only select movies were both applies to the same release, the `$elemMatch` can be used:

```
db.films.find({
  releases: {
    $elemMatch: {
      country: "USA",
      prerelease: true
    }
  }
},
{_id: 0, title: 1, releases: 1}
)
```

Naturally, there are many other operators not covered by this tutorial.

5.1.4 Update

You can also add or update fields in a document by using the `$set` operator. For example, you can add a `rating` field to one of the movies:

```
db.films.update(
  {title: "Star Trek Into Darkness"},
  {$set: {rating: 6.4}}
)
```

If you do not use the `$set` operator, every document fulfilling the selection criteria will be replaced, so be careful!

To increment a number of value, you can use the `$inc` operator:

```
db.films.update(
  {title: "Star Trek Into Darkness"},
  {$inc: {rating: 0.1}}
)
```

Again, there are many other different operators for different purposes, e.g. `$unset`, `$inc`, `$pop`, `$push`, `$pushAll` or `$addToSet`.

5.1.5 Delete

You can remove documents with the `remove` function. It actually works almost like the `find` function; you only don't use the projection parameter. If, for example, you want to remove all film documents whose title starts with the letter T, you can first query for all such movies...

```
db.films.find({title: /^T/})
```

...to verify that your selection criteria is correct and then replaced the `find` in your call by `remove`:

```
db.films.remove({title: /^T/})
```

	Course	**Databases and Information Systems 2020**		
	Exercise Sheet	**5**		
	Points	–		
	Release Date	**9ᵗʰ June 2020**	Due Date	**23ʳᵈ June 2020**

5.1.6 Speed Up Queries By Indexing

We want to use a larger data set, for this we want to import the provided .json file "movies.json"

In a new terminal where the file movies.json is located, execute one of the following commands, depending on your environment:

Bash:

```
docker exec -i mongo_db sh -c 'mongoimport -u root -p password -c movies -d imdb
    --authenticationDatabase=admin --drop' < ./src/data/movies.json
```

PowerShell:

```
cmd.exe /c "docker exec -i mongo_db sh -c " mongoimport -u root -p password -c movies -d
    imdb --authenticationDatabase=admin --drop " < ./src/data/movies.json"
```

Or if Mongo is installed without docker:

```
mongoimport.exe --db imdb --collection movies --file "/path/to/movies.json"
```

Now get back to the mongoDB CLI and execute:

```
db.movies.find({
  rating: {
    $gt: 6.14,
    $lt: 7.78
  }
}).explain("executionStats")
```

Note that `explain()` gives you information on the query execution. Write down the amount of time (*executionTimeMillis*) the execution of your query took!

To make the query execution of a little faster, create an index on the `rating` field. . .

```
db.movies.ensureIndex({rating: 1})
```

. . . and repeat the query! It now should have been performed faster (results may vary from run to run and depending on your system).

5.2 Movie mApp

Since you got to know some of the basics of MongoDB, now it's time for you to have a look at the MongoDB Java API! In our Movie mApp application, you can query IMDb movie data and also find out where on the world people are tweeting about movies. However, some of the most important features have not been implemented yet! Your task is to import the Java project `Blatt5.zip` into IDE of your choice(as IntelliJ or Eclipse) and implement the missing features. Dependencies are resolved by Maven.

For Exercise 5.2, we will reimport the data. Please make sure that the .json files are available under src/data/*.json, this way the app will import the data once at start.

Figure 5.2 shows the architecture of the project. It is a browser-based web application powered by a REST/HTTP backend which communicates with MongoDB. To start the REST server, simply run the RestServer class. You can then switch to your favorite browser and go to `http://localhost:5900`. The server will respond with the web application. The application uses asynchronous Ajax calls to fetch JSON data from the server.
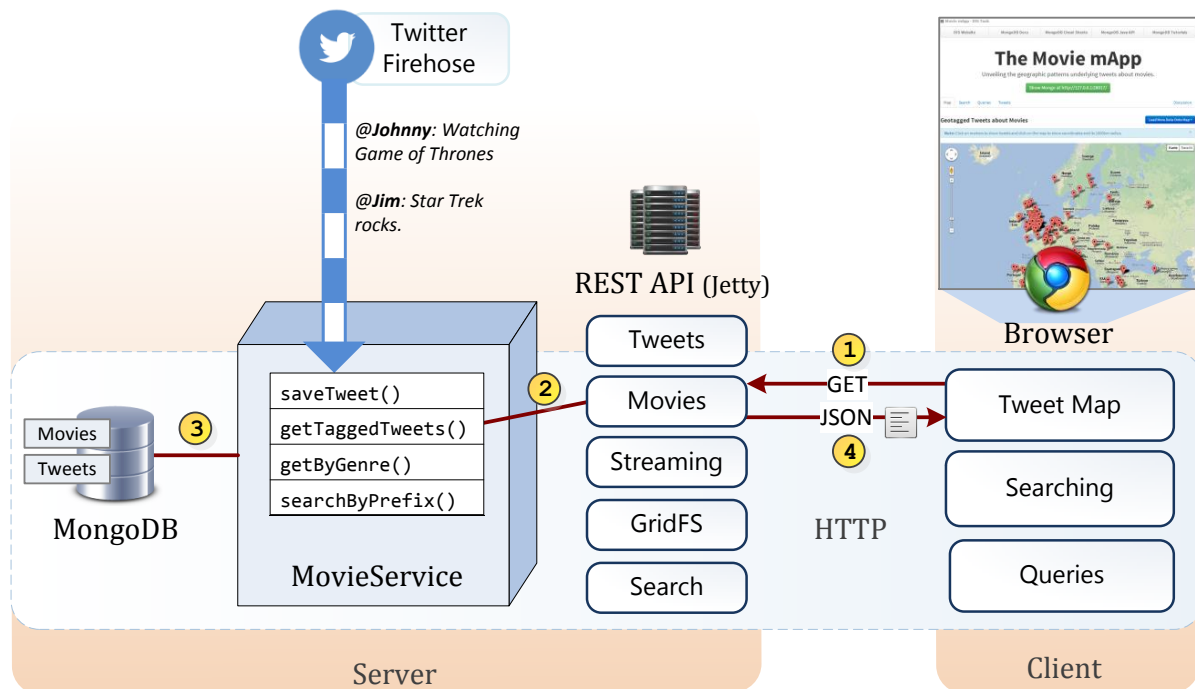
Abbildung 1: Architecture of the mApp project.

All this is already implemented. You only need to implement the missing parts in the `MovieService` class which encapsulates the MongoDB calls. **Places to work on are highlighted with TODOs.**

The RestServer will use your methods in this class – for example `getTaggedTweets()` to show tweets on the map. You should use the web frontend to check if your implementation of the methods works as expected. Show that this is working in your report.

The streaming of tweets from the Twitter API is included and working from the begin on, so you can stream real-time data into MongoDB. The sample dataset, too, already contains tweets to work with, as the streaming API may run into rate-limits. A part which will be new for you is GridFS. It is a distributed file system which is part of MongoDB. The movie application uses it to store poster images pulled from the International Movie Database using the respective methods in the `MovieService`.