1. Express и MongoDB

# Express

// Необходими библиотеки:

// npm install express

// npm install nodemon

//

// Стартиране на express:

// npm run start-express

// Скрипта изглежда така: "start-express": "nodemon start-express.js",

import express from 'express';

import { createClient } from 'redis';

var app = express(); // създаване нанова инстанция

app.use(express.json()); // активиране на поддръжката на json

app.listen(3001); // задаване на порт, на който да работи сървърът

// обработване на заявки

app.get('/', async function(req, res){

// Изпращане на json обект към клиента

let obj = {name:'Alex', age:10};

res.setHeader('Content-Type','application/json')

res.send(JSON.stringify(obj));

});

// прочитане на параметри от url адреса

app.get('/:id?', function(req, res){

res.send('Полъченото id е: ' + req.params.id);

});

// получаване на json съдържание през тялото на post заяка с обекта:

// {

// "user":"Alex",

// "age":10

// }

app.post('/', function(req, res){

//

console.log(req.header("Content-type"));

console.log(req.body);

console.log(req.body.user);

res.send("OK");

//res.send(req.body); // echo the result back

});

app.put('/', function(req, res){

res.send('PUT заявка');

});

app.delete('/', function(req, res){

res.send('DELETE заявка');

});

// Прихващане на всички невалидни адреси:

app.get('\*', function(req, res){

res.send('Грешка 404. Страницата не е намерена.');

});

# MongoDB

1. Обща информация

Mongo е NoSQL база от данни, която е от типа документно хранилище. Това означава, че в базата данни се съхраняват документи.

Като цяло, под документ, документните бази данни разбират структурирана или полуструктурирана информация, представена с помощта на xml, json, yaml или bson.

Специално при MongoDB се използва bson (Binary json), който представя json данни в двоична форма, с което се спестява място при съхранението на данните.

1. Mongo сървър

За да използваме база данни на Mongo, естествено трябва да имаме сървър. Тук имаме два варианта. Единият е да използваме безплатен сървър, предоставен от компанията, разработила Mongo, а другият е да изтеглим сървъра на нашите компютри и да го стартираме.

* 1. Използване на безплатен онлайн сървър

За да използваме безплатен онлайн сървър е необходимо да отворим страницата на Mongo:

<https://www.mongodb.com/>

след което от менюто Products да изберем Atlas

* 1. Използване на локален MongoDB сървър
     1. Изтегляне на MongoDB сървъра

За да използваме локален MongoDB сървър е необходимо да отворим страницата на mongo:

<https://www.mongodb.com/>

и от менюто Products да изберем Community Server

Това ще зареди страница, на която можем да изберем версията, която искаме да свалим и формата на файла, който искаме да свалим.

В нашия случай, бихме искали да изтеглим ZIP версията на сървъра за Windows, за да може след разархивиране да стартираме сървъра (не се налага инсталация).

* + 1. Стартиране на MongoDB сървъра

След като разархивираме изтегления архив, за да стартираме сървъра е необходимо да стартираме файла mongod.exe

При стартирането на файла, по подразбиране се използва папка в C:/ в която да бъдат съхранени базите данни, създадени на сървъра.

Понеже не искаме да съхраняваме базите данни там, бихме могли да си направим папка databases в разархивираната папка на Mongo сървъра (в същата папка, в която се намира папката bin).

За да укажем, че искаме да бъде използвана тази папка, за съхранение на базите данни, е необходимо да подадем допълнителен параметър при стартиране на сървъра. За тази цел, отваряме конзолния прозорец на Windows и правим папката, в която се намира mongod.exe текуща. След това, стартираме следната команда:

mongod.exe --dbpath "../databases/"

Забележка: пътят до базата данни указва, че се връщаме една папка нагоре, понеже exe файла на сървъра е в папката bin, a папката databases е извън тази папка.

За да не се налага да извършваме горните операции всеки път, при стартиране на MongoDB сървъра, бихме могли да създадем файла startup.bat в bin папката, да го отворим с notepad и в него да запишем горната команда. Така, при двукратно щракване с мишката върху bat файла, конзолата ще се отваря автоматично и сървъра ще се стартира по – лесно.

* + 1. Изтегляне и работа с Compass – GUI за достъп до MongoDB сървър
       1. Изтеглянена Compas

За да работим по-лесно със сървъра – да наблюдаваме неговото състояние, да създаваме, изтриваме и модифицираме нови бази данни, колекции и записи, можем да използваме Compass.

Приложението предоставя визуален достъп до даден MongoDB сървър. С помощта на приложението бихме могли да се вържем не само към локалният сървър, но и към облачния, за който стана дума по-рано.

За да изтеглим Compass, е необходимо да отворим страницата на mongo:

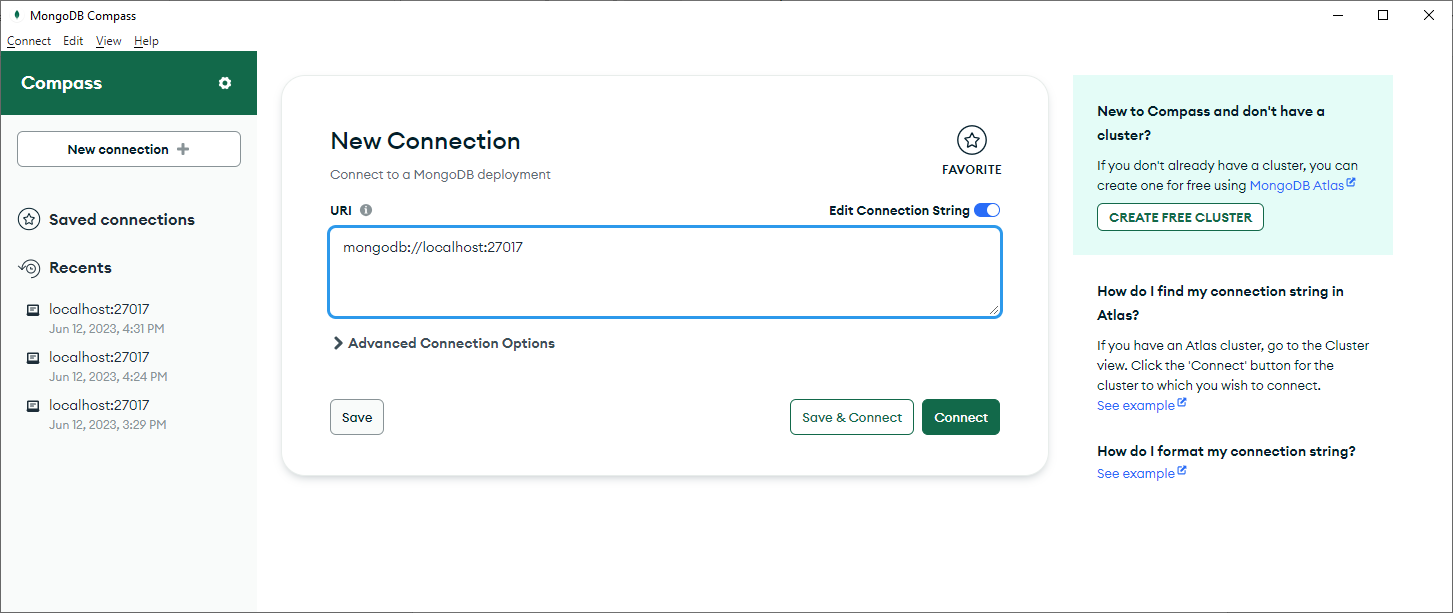
http://www.mongodb.com

и от менюто Products, от секцията Tools, да изберем Compass

В отворилия се прозорец ще видим бутон Download now, който ще ни заведе до страницата, откъдето можем да свалим софтуера.

Софтуера се предлага в няколко варианта. Например ако за версия 1.40.4 Изберем „More Options”“ ще видим hithub страница, откъдето можем да изтеглим варианти EXE, MSI или ZIP. Ако изтеглим ZIP варианта, когато го разархивираме, можем да стартираме приложението от файла MongoDBCompass.exe

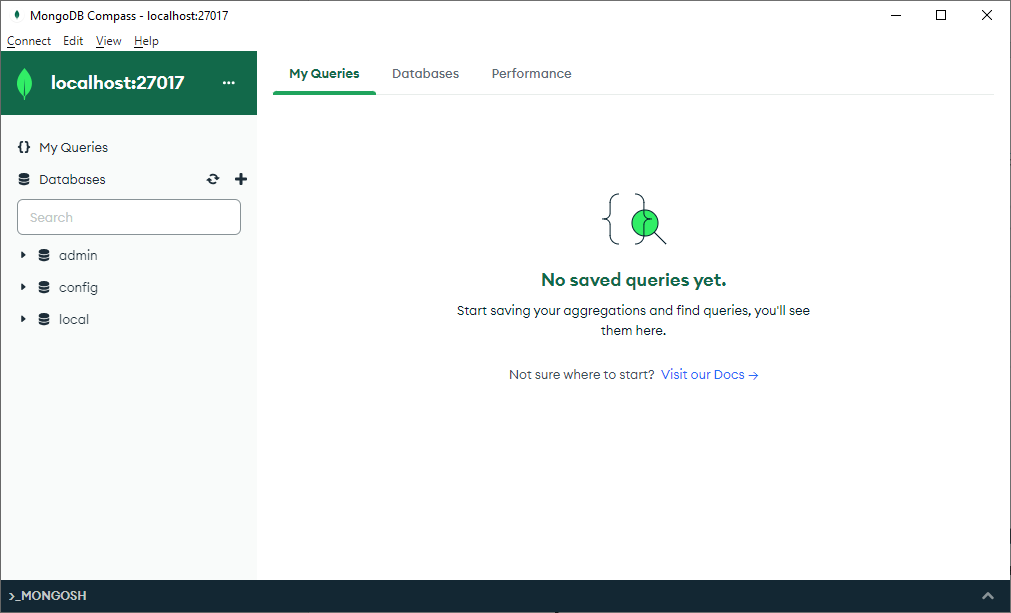
След стартиране, ще видим следният прозорец:



От бутона Connect можем да се свържем към локалния сървър, като не се изискват допълнителни настройки, тъй като сървъра се търси на адрес localhost на порта по подразбиране, използван от MongoDB сървъра 27017.

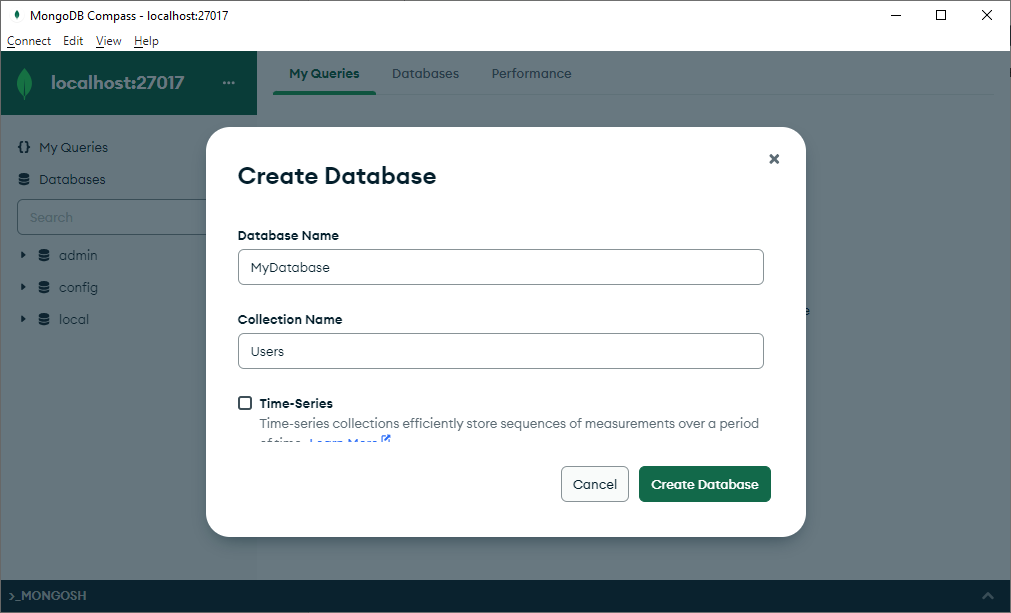
* + - 1. Работа с Compas

След като се свържем към сървъра, ще видим прозорец, подобен на следния:

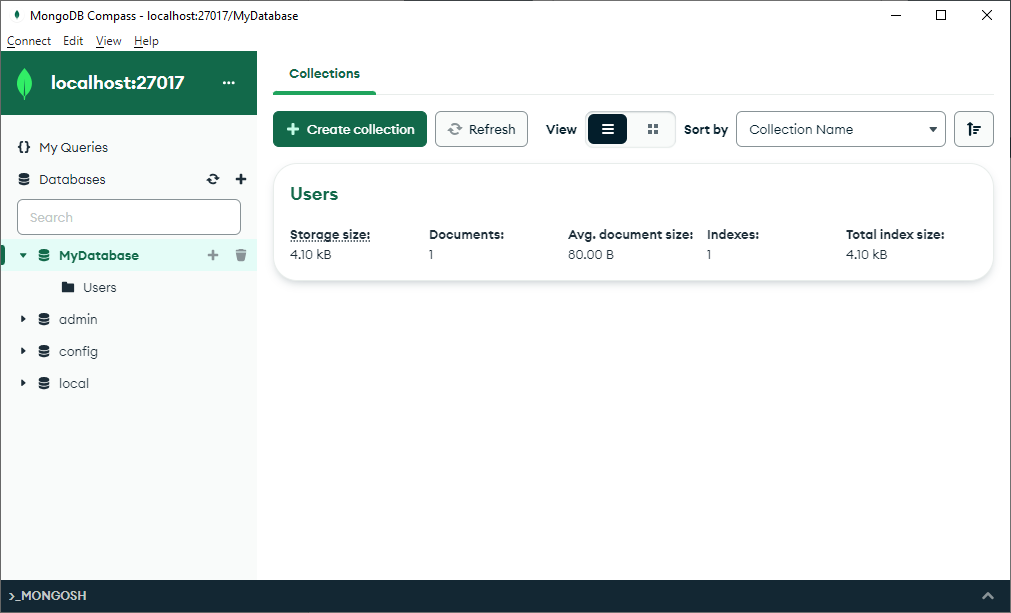


За да създадем нова база от данни, можем да щракнем върху кръглия бутон +, Срещу надписа Databases. При създаването на нова база данни, задължително трябва да създадем и една колекция.

Следващата фигура, показва как извлежда прозореца за създаване на нова база от данни:



Ако щракнем върху някоя база данни, от списъка в ляво, например върху базата данни MyDatabase, ще видим прозорец, подобен на следния:



В случая, виждаме че в базата ранни има една колекция с името Users, имаща само един документ (един запис).

Ако направим някакви промени по колекцията (например добавим още записи) или по базата данни (например добавим още колекции) от нашата програма, тогава за да видим промените в Compass е необходимо да изберем менюто View🡪Reload Data, за да заредим новите данни от сървъра.

1. Изпълнение на основни операции в Mongo
   1. Добавяне на mongoose към нашето React (node js) приложение

За да се свържем към сървъра, първо е необходимо да добавим библиотеката на mongo, наречена mongoose:

npm install mongoose

Документация на библиотеката можем да открием на следния адрес:

https://mongoosejs.com/docs/

* 1. Свързване към сървъра

За да се свържем към сървъра, е необходимо да първо да създадем схема. Схемата, която създаваме се свързва със дадена колекция, намираща се на сървъра. Схемата описва променливите, намиращи се в дадената колекция. Например, ако на сървъра имаме колекцията Users съдържаща записи със следната структура:

{

"username": "user1",

"password": "pass1",

"email": "email@email.com",

"age": 23,

"dateAdded": "2023-10-09T21:00:00.000Z",

"admnin": false

}

То можем в нашето React приложение да създадем следната схема:

const { Schema } = mongoose;

const userSchema = new Schema(

{

  username: String, // String е съкратено от {type: String}

  password: String,

  email: String,

  age: Number,

  dateAdded: { type: Date, default: Date.now },

  admin: Boolean,

//\_id: {type: Schema.Types.Number }

//\_\_v: { type: Number, select: false}

},

//{ versionKey: false }

);

Възможните типове са (https://mongoosejs.com/docs/schematypes.html):

String

Number

Date

Buffer

Boolean

Mixed

ObjectId

Array

Decimal128

Map

UUID

Щом сме създали схемата, на базата на нея трябва да създадем клас, който ще ни помогне да създаваме обекти от него. Класът се нарича модел и можем да го създадем по следния начин:

const User = mongoose.model('users', userSchema);

В горният пример, методът model() приема две променливи: името на колекцията към която ще се свързваме и името на схемата, която да използваме. След изпълнението на метода, получаваме клас, от който можем да създаваме обекти.

* 1. Добавяне на нови документи (записи) към базата от данни

След като сме получили референция към дадена колекция (клас, създаден, чрез model()), бихме могли да добавим нови документи (записи) към колекцията. Можем да използваме два варианта: да създадем обект и да го запишем, чрез метода save() на обекта или да използваме статичния метод User.create():

const userNiki = new User({

                                    username: "Niki",

                                    password: "pass1",

                                    email: "niki@niki.com",

                                    age: 10,

                                    dateAdded: Date.now(),

                                    admin: false

                                });

        await userNiki.save();

или:

await User.create({

                            username: "Viki",

                            password: "pass2",

                            email: "viki@viki.com",

                            age: 24,

                            dateAdded: Date.now(),

                            admin: false

                        });

1. **Други операции**

Намиране по ID:

user = await User.findById("65521fde53093e2ed3a0d672")

Избиране на всички:

users = await User.find()

users е масив от обекти. Можем да вземем името на първия обект така:

users[0].username

Филтриране:

users = await User.find().where('age').gt(20).lte(120)

.where("username").eq("Niki")

Обновяване по ID:

await User.findByIdAndUpdate("65522458816e0f537b83e0ee", {"age":27, "admin":false})

users = await User.findById("65522458816e0f537b83e0ee")

Обновяване по ID:

users = await User.findById("65522458816e0f537b83e0ee")

users.increment(); // Увеличава версията

users.age=45; // променя годините

await users.save(); // записва промените

users = await User.findById("65522458816e0f537b83e0ee")

Обновяване на много записи:

let res = await User.updateMany({"age":{$gte:10}}, {"age":11})

//res.modifiedCount; // Number of documents modified

users = await User.find()

let res = await User.updateMany({"username":"Alex"}, {"age":11})

Изтриване по ID:

user = await User.findByIdAndDelete("65521fde53093e2ed3a0d672")

Изтриване на много записи:

let res = await User.deleteMany({"age":{$gte:10}})

//res.modifiedCount; // Number of documents modified

users = await User.find()

* + 1. Създаване и добавяне на прост документ

Ето как можем да създадем съвсем прост документ, който след това да вмъкнем в колекцията:

// insert {"Name":"Viki"}

Document doc1 = **new** Document("Name", "Niki");

collection.insertOne(doc1);

Виждаме, че първо сме създали съвсем прост bson документ , който има следната структура: {"Name":"Niki"}. След това сме вмъкнали този документ в колекцията.

Ако отворим Compass и погледнем вмъкнатия документ, ще видим , че той изглежда така:

{

"\_id" : {"$oid":"5fc602906d962631b6c423b6"},

"Name" : "Niki" }

}

Това означава, че при вмъкването, документа автоматично е получил уникален идентификатор. Такъв идентификатор получават всички вмъкнати документи.

Забележка: В една колекция могат да се вмъкват документи с различна структура. Така например единият документ може да има Name, другият може да има Address, третият Address и Name и т.н.

* + 1. Създаване и добавяне на по-сложни документи с помощта на метода append()

Бихме могли да създадем и по-сложни обекти, с помощта на Document. Класа има фунцкия append(), която позволява добавяне на още променливи към bson обекта. Ето един пример:

// {

// "first\_name" : "Ivan",

// "last\_name" : "Ivanov",

// "position" : "Java Developer",

// "years\_of\_service" : 3,

// "skills" : ["java","mongodb"],

// "manager" : {

// "first\_name" : "Petyr",

// "last\_name" : "Petrov"

// }

// }

**int** random = **new** Random().nextInt(20);

// Създаваме List по един от двата начина:

ArrayList<String> list1 = **new** ArrayList<String>();

list1.add("java");

list1.add("mongodb");

list1.add("web");

//List<String> list2 = Arrays.asList("java", "mongodb", "web");

Document employee = **new** Document()

.append("first\_name", "Ivan")

.append("last\_name", "Ivanov")

.append("title", "Java Developer")

.append("years\_of\_service", random)

.append("skills", list1)

.append("manager", **new** Document()

.append("first\_name", "Petyr")

.append("last\_name", "Petrov"));

collection.insertOne(employee);

В горния пример, първо е показана структура на bson документа, който ще се създава. След това са декларирани променливи random, за годините стаж и List<String> за масива от умения (skills).

Освен масива, виждаме че променливата manager всъщност е обект, който има две променливи (first\_name и last\_name).

* + 1. Създаване и добавяне на по-сложни документи с помощта на обект от тип Map

Конструктора на класа Document() приема иобект от тип Map<String, Object>. Това означава, че можем да създаваме bson докъменти на базата на Map. Ето как може да стане това:

//{

// "Name":"Ivo",

// "age":20

//}

HashMap<String, Object> map = **new** HashMap<String, Object>();

map.put("Name", "Ivo");

map.put("age", 20);

//{

// "city":"Ryse",

// "country":"Bulgaria"

//}

HashMap<String,Object> map2 = **new** HashMap<String,Object>();

map2.put("city", "Ruse");

map2.put("country", "Bulgaria");

//{

// "Name":"Ivo",

// "age":20,

// "address": {

// "city":"Ryse",

// "country":"Bulgaria"

// }

//}

map.put("address", map2);

// Създаване на bson документ и добавянето му в БД

Document doc2 = **new** Document(map);

collection.insertOne(doc2);

В горния пример първо създаваме два map-a, който съхраняват променливите на bson във вид на ключ-стойност. След това, добавяме втория обект към първия под ключа address. Накрая, създаваме bson докъмент от първия map и го добавяме към базата от данни.

* 1. Преброяване на записите по дадени критерии

След като сме получили референция към дадена колекция, бихме могли да получим бройката на всички записи или бройката на записите, които отговарят на дадени критерии. Това става с помощта на функцията c countDocuments() или с помощта на функцията estimatedDocumentCount(), използвана за бързо преброяване. Ето как бихме могли да преброим всички документи в дадена колекция:

**long** allDoc1 = collection.estimatedDocumentCount();

или чрез

**long** allDoc2 = collection.countDocuments();

За да преброим всички документи по даден критерии, бихме могли да използваме следния код:

Document filter = **new** Document("firstName", "Niki");

**long** numOfNiki = collection.countDocuments(filter);

Виждаме, че първо се създава филтър, който указва, че трябва да се получат бройката на записите, чиито поле firstName има стойност Niki. Филтъра всъщност представлява bson обект. След като сме създали филтъра, го подаваме на преброяващата функция.

В следващата точка, ще видим как можем да извършим по-сложни филтрирания.

* 1. Създаване на по-сложни филтри

Всички по-сложни филтри също представляват bson документи.

* + 1. Обединяване на няколко условия с AND

За да създадем филтър, обединяващ няколко условия, чрез оператора AND, просто използваме метода append():

Document filter =

**new** Document().append("firstName", "Niki")

.append("age", 27);

Горното можем да запишем и като:

Document filter = **new** Document();

filter.append("firstName", "Niki");

filter.append("age", 27);

или като:

Document filter = **new** Document("firstName", "Niki");

filter.append("age", 27);

* + 1. Обединяване на няколко условия с OR

За да създадем подобен филтър е необходимо да използваме MongoDB оператора $or, като ключ на bson документа и списък от bson документи като стойност на оператора $or:

List<Document> filtersList =

Arrays.*asList*(

**new** Document("firstName", "Ivan"),

**new** Document("age", 27));

Document filter = **new** Document("$or", filtersList);

Забележка: вместо оператора $or можем да използваме оператора $and

* + 1. Операторите gt, lt, gte, lte, eq, ne

Можем да използваме и оператори като gt, lt, gte, lte, eq, ne по следния начин (gte се заменя снякои от другите оператори):

Document ageFilter = **new** Document("age",

**new** Document("$gte",18));

* + 1. Операторите in и nin

Операторите позволяват проверка дали дадена променлива е в изброен списък от стойности при in или е извън този списък при nin. Ето как бихме могли да проверим (да получим) всички обекти, чиито години стойности са 10, 20 или 30:

Document ageFilter = **new** Document("age",

**new** Document("$in", Arrays.*asList*(10,20,30)));

* + 1. Търсене в масиви, чрез $all и $elemMatch

Когато дадена променлива в bson документа е масив, можем да потърсим за дадени елементи, които имат всяка една от стойност от даден списък (чрез оператора $all) или можем да потърсим всички, които имат конкретна стойност в масива си (чрез оператора $elemMatch):

// Филтър за всички обекти, в чиито умения присъстват и трите стойности "java", "mongodb", "web"

List<Document> filtersList =

Arrays.*asList*("java", "mongodb", "web");

Document filterAll = **new** Document("skills",

**new** Document("$all", filtersList));

// Филтър за всички обекти, в чиито умения присъства стойността java

Document findDocument = **new** Document("skills",

**new** Document("$elemMatch", **new** Document("$eq", "java")));

* + 1. Търсене в подобекти

Когато даден обект притежава на свой ред обекти, както например:

{

"first\_name" : "Ivan",

"last\_name" : "Ivanov",

"title" : "Java Developer",

"years\_of\_service" : 3,

"skills" : ["java","mongodb"],

"manager" : {

"first\_name" : "Petyr",

"last\_name" : "Petrov"

}

}

Понякога ще искаме да създадем филтри за стойности на променливите на manager.

В това няма нищо сложно. Просто записваме името на обекта-променлива, последвано от точка и името на неговата променлива, която искаме да използваме. Например:

manager.first\_name

Ето как изглежда един примерен филтър:

Document filter = **new** Document("manager.first\_name", "Niki");

* 1. Търсене на документи

След като сме добавили известен брой документи, в някои случаи бихме искали да потърсим за един или няколко документа, отговарящи на даден критерий.

Търсенето всъщност е много лесно. То става с помощта наметода find():

Document filter = **new** Document("firstName","Viki");

FindIterable<Document> list = collection.find(filter);

Виждаме, че find() приема филтър, по който да извърши търсенето. В конкретния случай, филтъра е доста прост, но можем да го направим и по-сложен, чрез употребата на операторите, разгледани в предишната точка.

find() връща колекция от документи (FindIterable<Document>). Тази колекция можем да обходим, чрез итератор по следния начин:

MongoCursor<Document> it = list.iterator();

**while**(it.hasNext())

{

Document document = it.next();

Object firstName = document.get("firstName");

System.***out***.println(firstName);

}

Забележка: В горният случай използваме get(), като му подаваме даден ключ (променлива) и получаваме нейната стойност като обект. Ако обаче знаем типа на променливата, която е съхранена зад даден ключ, тогава вместо get()„ можем да използваме getString(), getInteger(), getDouble(), getBoolean(), getLong() или getList().

Вместо да обхождаме всички записи с итератор е възможно да използвам метода into(). Методът приема колекция от документи, преминава през целия списък от филтрирани елементи и добавя всеки един от тях към списъка. Накрая получаваме списък с всички елементи от заявката:

Document filter = **new** Document("firstName","Viki");

FindIterable<Document> list = collection.find(filter);

List<Document> results = **new** ArrayList<Document>();

list.into(results);

**for**(**int** i=0 ; i<results.size(); i++)

{

Document document = results.get(i);

String firstName = document.getString("firstName");

System.***out***.println(firstName);

}

* 1. Обновяване на стойност в даден документ
     1. Обновяване на променлива

За да обновим дадена променлива е необходимо първо да открием съответния обект, след което да подаден новата стойност за негово поле, чрез оператора $set:

Document findDocument = **new** Document("firstName", "Ivo");

Document updateDocument = **new** Document("$set",

**new** Document("firstName", "Niki"));

collection.updateMany(findDocument, updateDocument);

Горният код ще замени първото име на всички документи, в които имаме първо име Ivo, от Ivo на Niki.

Ако вместо функцията updateMany() използваме updateOne(), ще се замени само първият елемент от всички открити елементи, имащи за първо име Ivo.

Забележка: Освен $set има и други оператори, които могат да бъдат използвани за обновяване на стойността на дадена променлива. Списък с всички оператори може да е види на:

<https://docs.mongodb.com/manual/reference/operator/update-field/>

* + 1. Обновяване на списък

За да добавим нова стойност към списък е необходимо първо да открием дадения запис, който съдържа списъка, след което да подаден новата стойност, която да бъде добавена към списъка, чрез $push:

Document findDocument = **new** Document("firstName", "Ivo");

Document updateDocument = **new** Document("$push",

**new** Document("skills", "security"));

collection.updateMany(findDocument, updateDocument);

Горният код ще открие всички докъменти, чиито първи имена са Ivo и към тяхната променлива-списък skills ще добави security.

Забележка: Освен добавяне на стойност, чрез $push можем да премахваме или променяме стойности, чрез други оператори. Списък с всички оператори, които можем да приложим при списъците може да е види на:

https://docs.mongodb.com/manual/reference/operator/update-array/

* 1. Изтриване на документ

Изтриването на документ също е лесно. Трябва да укажем филтър, чрез който да се открие елемента или елементите, които искаме да изтрием:

Document deleteFilter = **new** Document("firstName","Viki");

collection.deleteMany(deleteFilter);

Разбира се, вместо deleteMany(), Можем да използваме deleteOne(), за да изтрием само първия елемент, който съответства на тъсещият критерий.

* 1. Автоматично увеличаващи се номера на ключове в MongoDB

Както знаем , MongoDB автоматично добавя \_id към всеки добавен запис, който не съдържа такова поле. Това означава, че всеки добавен запис притежава уникален идентификатор.

Ако зададем \_id при вмъкване на запис в базата от данни, тогава можем да подадем желана от нас стойнот (напр. цяло число или текст). Ако обаче при вмъкване на записа не подадем поле \_id, тогава MongoDB автоматично ще добави такова поле към записа, като стойността на полето ще бъде от тип ObjectId.

ObjectId е тип, който съдържа в себе си датата на добавяне на съответния запис и допълнителна информация, правеща записа уникален. Това означава, че извличайки съответния запис, можем да разберем точната дата на вмъкването му в базата данни. На базата на ObjectId, MongoDB позволява и сортиране на записите според реда на добавяне към базата от данни.

Забележка: Всеки ObjectId обект заема 12 байта в базата от данни.

Тъй като ObjectId представлява „произволни“ байтове, в някои случаи това не ни върши много добра работа и бихме искали да имаме записи, чиито униклни идентификатори започват от 1 и нарастват последователно нагоре.

Mongo не предлага такава функционалност директно, но с няколко последователни операции, бихме могли да постигнем това поведение. Ето и стъпките, които биха ни погнали за това:

1. Създаваме колекцията counters, която ще съхранява текущата стойност на уникалните идентификатори на всички колекции, които имат нужда от подобна функционалност:

MongoCollection<Document> countersCollection = database.getCollection("counters");

1. Добавяме нов запис, който има следната структура:

{

"\_id":"collection\_name",

"sequence\_value", 1

}

Това може да стане със следния код:

Document doc = **new** Document()

.append("\_id", "имe\_на\_колекция") .append("sequence\_value", 1);

countersCollection.insertOne(doc);

С горния код, добавихме запис, който има за \_id името на колекцията, която ще поддържа увеличаващи се идентификатори. sequence\_value пък ще показва последния използван идентификатор.

1. Вмъкване на нов запис

Когато има нужда да се добави нов запис, можем да използваме следния код:

// Намиране на следващото свободно id

Document sequenceDoc =

countersCollection.find().first();

**int** nextAvailableId =

sequenceDoc.getInteger("sequence\_value");

// Вмъкване

Document newDocument = **new** Document()

.append("\_id", nextAvailableId)

.append("name","Viki");

collection.insertOne(newDocument);

Това, което правим с горния код е да намерим следващото свободно id от колекцията countersCollection, като получим първия запис, който в случая е единствения, след което от него получим стойността на sequence\_valus. В променливата nextAvailableId.

След това, използваместойността nextAvailableId при добавянена новия запис към колекцията collection

1. Обновяване на следващата свободна стойност

След като използвахме стойността от полето sequence\_value е необходимо да я обновим, за да може при следващото вмъкване в колекцията collection да имаме отново уникалнастойност. Обновяването (увеличението с единица), може да стане със сления код:

Document findDocument = **new** Document("\_id", "имe\_на\_колекция");

Document updateDocument = **new** Document("$inc",

**new** Document("sequence\_value", 1));

countersCollection.updateOne(findDocument,

updateDocument);

С горния код, указваме, че искаме да търсим за дадена колекция с име имe\_на\_колекция (което разбира се ще бъде различно в зависимост от името на колекцията, която ще има увеличаващи се \_id-та). В документа updataDocument сме указали, че искаме MongoDB да изпълни оператоца $inc като увеличи стойността на полето sequence\_value с единица.