Достъп до MongoDB с Node.js – CRUD операции

I. Въведение

Създаването на нова база данни, колекция и документи, обновяването и изтриване на документи чрез използване на Node.js е аналогично на това при използване на Java. Използват се същите методи, благодарение на MongoDB Node.js Driver. Трябва да отбележим, че JavaScript има вградени възможности за работа с JSON документи и не се налага да използвате външни модули. Операциите Create, Read, Update, Delete (CRUD) позволяват да работите с данните, съхранявани в базата данни. Те се делят на две групи:

- Операции за четене.
- Операции за запис.

II. Операции за четене

Към тази група принадлежат операции за намиране и връщане на съдържанието на документи. Съществуват няколко вида операции за четене, които осъществяват достъп до данните по различни начини. Вече знаем, че извличането на информация от базата данни въз основа на набор от критерии от съществуващия набор от данни, можете да използвате един от вариантите на метод **find**. Можете също така да уточните допълнително информацията, която искате да получите, например:

- Сортиране на резултатите ($sort).
- Пропускане на върнатите резултати ($skip).
- Ограничаване на броя на върнатите резултати ($limit).
- Избор кои полета да бъдат върнати ($project).

Можете да използвате операция за агрегиране, за да извлечете желаните данни. За целта използвайте метод **aggregate**. Този тип операция ви позволява да приложите конвейер от трансформации към данните.

III. Операции за запис

Към тази група принадлежат операции за вмъкване, промяна или изтриване на документи от бази данни.

3.1 Вмъкване на документ

Ако искате да добавите нов(и) документ(и) в колекция, можете да използвате метода **insertOne** или **insertMany**. Тези методи приемат като аргумент(и) един или множество документи. Драйверът автоматично генерира уникално поле „_id” за документите, освен ако не е посочено да се работи с друг идентификатор. Синтаксисът на метод **insertOne** е следния:

```
const result = await collection.insertOne(myDocument);
```

където myDocument е JSON обект, описващ документа, който искате да вмъкнете. Може да проверите броя на реално вмъкнатите документи чрез стройността на поле “insertedCount” от обект “result”:

```
console.log(result.insertedCount);
```

Ако искате да вмъкнете множество документи в базата данни чрез един метод, използвайте **insertMany**. Синтаксисът на този метод е следния:

```
const result = await collection.insertMany(myDocuments);
```

където myDocuments е масив от JSON обекти - документите, които искате да вмъкнете.

3.2 Изтриване на документ

Ако искате да премахнете съществуващ документ от колекция, можете да използвате метод **deleteOne**. Използвайте метод **deleteMany**, за да премахнете един или повече документи. Тези методи изискват като аргумент документ за заявка, който съответства на документите, които искате да изтриете.

```
const result1 = await collection.deleteOne(query);
const result2 = await collection.deleteMany(query);
```

където "query" е JSON обект, който описва кой обект(и) искаме да изтрием, например:

```
const query = {
    timestamp: {
        $lte: new Date(2021-06-20),
        $gte: new Date(2021-06-10)
    }
};
```

В този случай се задава да се изтрият всички документи от дата 10.06.2021 до 20.06.2021. Ако използваме метод deleteOne, ще бъде изтрит само първият срещнат в колекция документа за който условието е в сила. Реалният брой на изтритите документи можете да получите чрез стойността на поле "deletedCount", например:

```
console.log(deleteResult.deletedCount);
```

3.3 Обновяване на съдържанието на документ(и)

Обновяването (актуализация) на съдържанието на документ(и) се реализира чрез метод **update**. Променят се определени полета, а останалите полета остават непроменени. За да извършите актуализация на един или повече документи, създайте документ за актуализация, в който са посочени операторът за актуализация (типът актуализация, който трябва да се извърши) и полетата и стойностите, които описват промяната. Методите за актуализация са два: **upadeOne** и **updateMany**. Синтаксисът е следния:

```
const result = await collection.updateOne(filter, updateDocument);
```

където "filter" е JSON обект, който описва правилата за филтриране на документите, а втория аргумент "updateDocument" е JSON обект, който описва какво се актуализира. Форматът на "updateDocument" е следния:

```
{
   <update operator>: {
     <field> : {...}
     },
     <field> : {...}
   },
   <update operator>: {
      ...
   }
}
```

Операторите за актуализация, които може да използвате са следните:

- $set - заменя стойността на дадено поле с определена стойност.
- $inc - увеличава или намалява стойностите на определени полета.
- $rename - преименува полета.
- $unset - премахва полета.
- $mul - умножава стойността на полето по определено число.

Следва пример, който променя една от оценките на студент с идентификатор “21705001”:

```
const filter = {id: "21705001"}
const updateDocument = {
   $set: {
      "grade.НБД.value": 5
   }
};
const result = await collection.updateOne(filter, updateDocument);
console.log(`Брой обновени полета: ${result.modifiedCount}`);
```

За да обновите множество документи с една заявка използвайте метод updateMany. Синтаксисът е същият като този за updateOne. В зависимост от филтъра ще бъдат обновени 0, 1 или повече документи.

Обновяване на полета от масиви

Ако трябва да модифицирате масив, можете да използвате оператор за актуализиране на масива в извикването на метода за актуализиране, например:

- $ - позиционен оператор - първи елемент от масива.
- $[] - оператор за съвпадение на всички елементи на масива.
- $[<идентификатор>] - филтриран позиционен оператор.

За да извършите актуализация само на първия елемент от масива на всеки документ, който съвпада с документа на заявката ви, използвайте оператора за актуализация на позиционен масив **$**. Този оператор за актуализация се позовава на масива, съответстващ на филтъра на заявката, и не може да се използва за позоваване на масив, вложен в този масив. Ако искате да обновите поле или полета от всички документи в масива използвайте оператор **$[]**. За случаите, в които трябва да получите достъп до вложените масиви, използвайте позиционния оператор **$[<идентификатор>]**. Стойността на „идентификатор” се задава в обекта “arrayFilters”, който се предава като последен аргумент за метод update. Този обект представлява масив от филтри, които определят кои елементи от масива да бъдат включени в актуализацията, например:

```
const filter = { };
const updateDocument = {
    $set: {"arrayName.$[arrayItem].fieldName": value }
};
const updateOptions =
{
    arrayFilters: [{
       "arrayItem.fieldName1": value1,
       "arrayItem.fieldName2": value2
    }]
};
const result
= await collection.updateMany(filter, updateDocument, updateOptions);
```

Ако искате да обновите стойността на елемент от масив използвайте оператор $set. Ако трябва да вмъкнете нов елемент в масив - използвайте оператор $push.

3.4 Замяна на съдържанието на документ(и)

Методите за замяна премахват всички съществуващи полета в един или повече документи и ги заместват с определени полета и техните стойности. За целта се използват методи **replaceOne** и **replaceMany**. Синтаксисът е следния:

```
const result = await collection.replaceOne(filter, replacementDocument);
```

За да извършите операция за замяна, създайте документ за замяна, който се състои от полетата и стойностите, които искате да вмъкнете в операцията за замяна. Документите за замяна използват следния формат:

```
{
   <field>: {
      <value>
   },
   <field>: {
      ...
   }
}
```

Следва пример, който заменя всички полета от документ с поле “id” = “21705001” с поле “newField”=5. Не може да заменяте само служебното поле “_id”. Броят на модификациите може да получите чрез поле “modifiedCount” от обект result.

```
const filter = {id: "21705005"}
const replacementDocument = {
   newField: 5
};
const result = await collection.replaceOne(filter, replacementDocument);
console.log(`Брой заместени документи: ${result.modifiedCount}`);
```

3.5 Вмъкване или актуализиране с една операция

В някои случаи може да се наложи да избирате между операция за вмъкване и актуализация в зависимост от това дали документът съществува. В тези случаи можете да оптимизирате логиката на приложението си, като използвате опцията “**upsert**”. Тя е налична в следните методи:

- updateOne

- updateMany
- replaceOne

Ако филтърът от заявката, предаден на тези методи, не намери съвпадения и стойността на “upsert” е true, сървърът за управление на MongoDB ще вмъкне нов документ. В противен случай зададените полета ще се обновят (update) или ще заместят старите полета (replace).

Задачи за изпълнение

Задача 1: *Създайте Node.js приложение, което* обновява оценката по дисциплина НБД на всички студенти от база “students”, колекция “data1”.

Тъй като трябва да се обновят оценките за всички студенти ще използваме метод **updateMany**. Поради тази причина не е необходимо да се задава условие за филтриране на документите. При подобни случаи аргумент “filter” трябва да бъде празен JSON обект. Чрез атрибут “updateDocument” трябва да се зададе, че се обновява оценката (поле “value”) по дисциплината “НБД”. Следователно, трябва да се използва оператор $set и позиционен идентификатор с филтриране. От всички дисциплини в масив “grade” трябва да се работи само с тези за които поле “subject” е със стойност “НБД”. Това се постига чрез опция “arrayFilters”. Филтрирането е в зависимост от стойност на поле “subject”. Едно примерно решение на задачата е показано на Фиг. 1.

```
const filter = { };
    const updateDocument = {
      $set: {"grade.$[arrayItem].value": 4}
    };
const updateOptions = {
    arrayFilters: [{
        "arrayItem.subject": "НБД"
    }]
};
const result =
   await collection.updateMany(filter, updateDocument, updateOptions);
console.log(`Брой обновени документи: ${result.modifiedCount}`);
```

Фиг. 1 Примерно решение на Задача 1

Задача 2: *Създайте Node.js приложение, което създава нова база данни, колекция и документи към колекцията. Документите да се четат от JSON файл. Да се използват JavaScript обещания, за да се синхронизира работата на използваните методи.*

Ще използваме възможностите на Node.js за работа с файлове чрез модул “fs”. Константите, които приложението ще използва, са показани на Фиг. 2. Зареждането на модул “fs” се реализира чрез метод “require”. Останалите константи са свързани с имената на базата данни, колекцията и опциите, които се предават като аргумент на метод connect.

```
const fs = require('fs');
const MongoClient = require('mongodb').MongoClient;
const url = "mongodb://localhost:27017";

const jsonFile = 'filename.json';
const databaseName = 'databaseName';
const collectionName = "collName";

const options = {
    serverSelectionTimeoutMS: 3000,
    connectTimeoutMS: 3000,
    socketTimeoutMS: 3000,
    useUnifiedTopology: true
};
```

Фиг. 2 Използвани константи, Задача 2

Програмният език JavaScript използва интерфейс *Promise*, за да създаде обект-обещание. Чрез този обект е възможно да дефинираме асинхронно работещ код, както и да получим информация в момента, когато кодът завърши без или с грешка. Синтаксисът на създаване на обещание е показан на Фиг. 3. Ако асинхронния код от тялото на анонимната функция завърши успешно се вика метод **resolve** (ред 5). При грешка се вика метод **reject** (ред 8). И на двата метода се предава един аргумент – JSON обект с резултата или JSON обект, който описва грешката.

```
1  var promise = new Promise(function(resolve, reject) {
2      // асинхронен код, изпълнението на който
3      // завършва с уставяване на флаг статус
4      if (статус === "ок") {
5          resolve(обект);
6      }
7      else {
8          reject(обект);
9      }
10 });
```

Фиг. 3 Създаване на обект-обещание

Когато се изпълни метод **resolve** се вика метод **then**, приложен към обекта-обещание. Следователно, той изпълнява ролята на callback функция. При извиквана на метод **reject**, управлението се предава на **catch** клауза (виж Фиг. 4).

```
1  promise.then(function(обект) {
2      // успешно изпълнен код (resolve)
3  }).catch(function(обект) {
4      // неуспешно изпълнен код
5  })
```

Фиг. 4 Обработка на събития resolve и reject

На Фиг. 5 е показан програмния код на метод **createDatabase** чрез който се създава базата данни, колекция и се записват документите в колекцията. Методът изисква три аргумента:

- file – файлова спецификация на JSON файла с документите.
- database – име на базата данни.
- collection – име на колекцията.

```
const createDatabase = (file, database, collection) =>

    new Promise((resolve, reject) => {

        process.stdout.write("Read data from file: ");
        fs.readFile(file, (error, data) => {
            if (error) return reject(error);
            process.stdout.write('OK\n');
            var jsonData = JSON.parse(data);

            process.stdout.write("Connect to server: ");
            MongoClient.connect(url, options, (error, conn) => {
                if (error) return reject(error);
                process.stdout.write('OK\n');
                let db = conn.db(database);

                process.stdout.write("Create collection: ");
                db.createCollection(collection, (error, result) => {
                    if (error) return reject(error);
                    process.stdout.write('OK\n');

                    process.stdout.write("Insert docs: ");
                    result.insertMany(jsonData, (error, result) => {
                        if (error) return reject(error);
                        conn.close();
                        return resolve(result.insertedCount +
                                                " documents inserted.");
                    });

                });

            });

        });

    });
```

Фиг. 5 Метод createDatabase - Задача 2

Форматът на JSON файла трябва да съдържа масив с документи. Съдържанието на файла се прочита чрез метод **readFile** от модул “fs”. Методът приема два аргумента - файловата спецификация и ламбда функция, която се активира след като readFile върне отговор. При успешно прочитане на данните, посредством метод parse конвертираме съдържанието на файла от низ до JSON обект – jsonData.

Следва свързване с MongoDB сървъра чрез метод **connect**. На метода се предават три аргумента: URL, опции към connect и ламбда функция, която се активира след като метод connect върне резултат. При успешно създаване на комуникационен канал се създава базата данни чрез метод **db**. Следва създаване на желаната колекция. Това се реализира чрез метод **createCollection**. Остана само да вмъкнем документите (jsonData) в новата колекция. Това се реализира чрез метод **insertMany**.

На Фиг. 6 е показан програмния код, който вика метод **createDatabase**.

```
createDatabase(jsonFile, databaseName, collectionName)
  .then(result => console.log(result))
  .catch(error => {
      process.stdout.write(`(ERROR: ${error.message})`);
      process.exit()
  });
```

Фиг. 6 Викане на метод createDatabase - Задача 2

Вижда се, че е използван JavaScript обект-обещание, за да се синхронизира момента на получаване на информация от сървъра. Освен това, по този начин се съсредоточава обработката на изключенията само на едно място – тялото на клауза catch.

Задача 3: *Създайте Node.js приложение, което създава нова база данни с име "sensors" и колекция с име "data". Всеки документ в тази колекция трябва да съдържа информация за сензор от безжична сензорна мрежа. Сензорите могат да бъдат два типа: за измерване на температура и за измерване на относителна влажност на въздуха. Всеки сензор има идентификатор, тип, показание и времева марка (кога е направено измерването).*

За да решим задачата ще ни трябват следните константи:

```
const databaseName = 'sensors';
const collectionName = "data";

const numberOfRecords = 10000;
const numberOfSensors = 20;
const sensorTypes = ['temperature', 'humidity'];
const numberOfSensorTypes = sensorTypes.length;
```

Фиг. 7 Използвани константи - Задача 3

Тъй като ще генерираме случайни стойности за идентификатор на сензор, за тип на сензор и времева марка ще използваме два метода, които генерират случайни числа в интервала [min, max]. Техния програмен код е показан на Фиг. 8. Първият метод връща случайно число, а вторият – случайно число като форматиран низ. Последното е необходимо, за да генерираме случайна дата и час.

```
function getRandomNumberInRange(min, max) {
    return Math.floor((Math.random() * (max - min + 1) + min));
}
function getRandomNumberInRangeAsString(min, max) {
    let random = getRandomNumberInRange(min, max);
    return String(random).padStart(2,'0');
}
```

Фиг. 8 Методи за генериране на случайни числа в зададен интервал, Задача 3

Програмният код на метод **createDatabase**, който създава базата данни и я заселва с документи, е показан на Фиг. 9. След създаване на базата данни и колекцията се преминава към формиране на масив от документи в паметта. За целта се използва масив с име "records". Чрез for цикъл се получават всички необходими полета и формиране на JSON обект от тях – обект "records". Вмъкването на всеки документ в масива се реализира чрез метод **push**. След това остава само да въведем документите в колекцията чрез метод **insertMany**. При успешно изпълнение на метод **createDatabase** се връща броя на вмъкнатите документи чрез метод **resolve**. При някак тип грешка, обектът, който описва грешката, се предава за обработка към клауза **catch** чрез викане на метод **reject**.

Викането на метод createDatabase се реализира с програмния код показан на Фиг. 10.

```
const createDatabase = (database, collection) =>
    new Promise((resolve, reject) => {
        process.stdout.write("Connect to server: ");
        MongoClient.connect(url, options, (error, conn) => {
            if (error) return reject(error);
            process.stdout.write('OK\n');
            let db = conn.db(database);
            process.stdout.write("Create collection: ");
            db.createCollection(collection, (error, result) => {
                if (error) return reject(error);
                process.stdout.write('OK\n');
                process.stdout.write("Insert docs: ");
                let records = [];
                for (let i=0; i<numberOfRecords; i++) {
                    let sensorTypeId =
                    getRandomNumberInRange(0, numberOfSensorTypes-1);
                    let sensorTypeName = sensorTypes[sensorTypeId];
                    let temperature = getRandomNumberInRange(-10, 38);
                    let humidity = getRandomNumberInRange(20, 99);

                    let day = getRandomNumberInRangeAsString(1, 31);
                    let hour = getRandomNumberInRangeAsString(0, 24);
                    let min = getRandomNumberInRangeAsString(0, 59);
                    let sec = getRandomNumberInRangeAsString(0, 59);
                    let sensorId = getRandomNumberInRange(1, numberOfSensors);
                    let record = {
                        sensorId: sensorId,
                        timestamp:
                           new Date(`2021-06-${day}T${hour}:${min}:${sec}Z`),
                        type: sensorTypeName,
                        value: sensorTypeName ===
                                'temperature' ? temperature : humidity
                    }
                    records.push(record);
                }
                result.insertMany(records, (error, result) => {
                    if (error) return reject(error);
                    conn.close();
                    return resolve(result.insertedCount +
                                            " documents inserted.");
                });

            });

        });
    });
```

Фиг. 9 Програмен код на метод createDatabase - Задача 3

```
createDatabase(databaseName, collectionName)
  .then(result => console.log(result))
  .catch(error => {
      process.stdout.write(`(ERROR: ${error.message})`);
      process.exit();
  });
```

Фиг. 10 Викане на метод createDatabase, Задача 3