Лабораторна робота №3.

Node.js та MongoDB.

Що таке MongoDB

MongoDB <https://cloud.mongodb.com> реалізує новий підхід до побудови баз даних, де немає таблиць, схем, запитів SQL, зовнішніх ключів і багатьох інших речей, які притаманні об'єктно-реляційних баз даних.

З часів динозаврів було звичайною справою зберігати всі дані в реляційних базах даних (MS SQL, MySQL, Oracle, PostgresSQL). При цьому було не так важливо, а чи підходять реляційні бази даних для зберігання даного типу даних чи ні.

На відміну від реляційних баз даних MongoDB пропонує документо-орієнтовану модель даних, завдяки чому MongoDB працює швидше, має кращу масштабованість, її легше використовувати.

Але, навіть враховуючи всі недоліки традиційних баз даних і переваги MongoDB, важливо розуміти, що завдання бувають різні і методи їх вирішення бувають різні. В якійсь ситуації MongoDB дійсно поліпшить продуктивність вашої програми, наприклад, якщо треба зберігати складні за структурою дані. В іншій же ситуації краще буде використовувати традиційні реляційні бази даних. Крім того, можна використовувати змішані підхід: зберігати один тип даних в MongoDB, а інший тип даних - в традиційних БД.

Вся система MongoDB може представляти не тільки одну базу даних, що знаходиться на одному фізичному сервері. Функціональність MongoDB дозволяє розташувати кілька баз даних на декількох фізичних серверах, і ці бази даних зможуть легко обмінюватися даними і зберігати цілісність.

Формат даних у MongoDB

Одним з популярних стандартів обміну даними та їх зберігання є JSON (JavaScript Object Notation). JSON ефективно описує складні за структурою дані. Спосіб зберігання даних в MongoDB в цьому плані схожий на JSON, хоча формально JSON не використовується. Для зберігання в MongoDB застосовується формат, який називається **BSON** (БіСон) або скорочення від binary JSON.

BSON дозволяє працювати з даними швидше: швидше виконується пошук і обробка. Хоча треба зазначити, що BSON на відміну від зберігання даних в форматі JSON має невеликий недолік: в цілому дані в JSON-форматі займають менше місця, ніж в форматі BSON, з іншого боку, даний недолік з лишком окупається швидкістю.

Кросплатформенність MongoDB

MongoDB написана на C ++, тому її легко перенести на найрізноманітніші платформи. MongoDB може бути розгорнута на платформах Windows, Linux, MacOS, Solaris. Можна також завантажити вихідний код і самому скомпілювати MongoDB, але рекомендується використовувати бібліотеки з офсайта.

Документи замість рядків.

Якщо реляційні бази даних зберігають рядки, то MongoDB зберігає документи. На відміну від рядків документи можуть зберігати складну за структурою інформацію. Документ можна уявити як сховище ключів і значень.

***Ключ*** являє просту мітку, з яким асоційоване певна частина даних.

Однак при всіх відмінностях є одна особливість, яка зближує MongoDB і реляційні бази даних. У реляційних СУБД зустрічається таке поняття як ***первинний ключ***. Це поняття описує якийсь стовпчик, який має унікальні значення. У MongoDB для кожного документа є унікальний ідентифікатор, який називається \_id. І якщо явно не вказати його значення, то MongoDB автоматично згенерує для нього значення.

Кожному ключу зіставляється певне значення. Але тут також треба враховувати одну особливість: якщо в реляційних базах є чітко окреслена структура, де є поля, і якщо якесь поле не має значення, йому (в залежності від налаштувань конкретної БД) можна привласнити значення NULL. У MongoDB все інакше. Якщо якомусь ключу не сспівпоставлене значення, то цей ключ просто опускається в документі і не вживається.

Колекції

Якщо в традиційному світі SQL є таблиці, то в світі MongoDB є колекції. І якщо в реляційних БД таблиці зберігають однотипні жорстко структуровані об'єкти, то в колекції можуть містити найрізноманітніші об'єкти, що мають різну структуру і різний набір властивостей.

Реплікація

Система зберігання даних в MongoDB представляє набір реплік. У цьому наборі є основний вузол, а також може бути набір вторинних вузлів. Всі вторинні вузли зберігають цілісність і автоматично оновлюються разом з оновленням головного вузла. І якщо основний вузол з якихось причин виходить з ладу, то один з вторинних вузлів стає головним.

Просте використання

Відсутність жорсткої схеми бази даних і в зв'язку з цим потреби при щонайменшій зміні концепції зберігання даних перестворювати цю схему значно полегшують роботу з базами даних MongoDB і подальшим їх масштабуванням. Крім того, економиться час розробників. Їм більше не треба думати про перестворення бази даних і витрачати час на побудову складних запитів.

GridFS

Однією з проблем при роботі з будь-якими системами баз даних є збереження даних великого розміру. Можна зберігати дані в файлах, використовуючи різні мови програмування. Деякі СУБД пропонують спеціальні типи даних для зберігання бінарних даних в БД (наприклад, BLOB в MySQL).

На відміну від реляційних СУБД MongoDB дозволяє зберігати різні документи з різним набором даних, однак при цьому розмір документа обмежується 16 мб. Але MongoDB пропонує рішення - спеціальну технологію ***GridFS***, яка дозволяє зберігати дані за розміром більше, ніж 16 мб.

Система GridFS складається з двох колекцій. У першій колекції, яка називається files, зберігаються імена файлів, а також їх метадані, наприклад, розмір. А в іншій колекції, яка називається ***chunks***, у вигляді невеликих сегментів зберігаються дані файлів, зазвичай сегментами по 256 кб.

Для тестування GridFS можна використовувати спеціальну утиліту mongofiles, яка йде в пакеті mongodb.

Початок роботи з MongoDB

Найбільш популярною системою управління базами даних для Node.js на даний час є MongoDB. Для роботи з цією платформою перш за все необхідно встановити сам сервер MongoDB. А можно скористатися вже готовими рішеннями наприклад Open Server <https://ospanel.io>. Рекомендується встановити Крім самого сервера Mongo для взаємодії з Node.js рекомендується встановити графічний клієнт Compass <https://www.mongodb.com/try/download/compass>.

При підключенні і взаємодії з БД в MongoDB можна виділити наступні етапи:

1. Підключення до сервера
2. Отримання об'єкта бази даних на сервері
3. Отримання об'єкта колекції в базі даних
4. Взаємодія з колекцією (додавання, видалення, отримання, зміна даних)

Отже, створимо новий проект. Для цього визначимо новий каталог, який буде називатися **mongoapp**. Далі визначимо в цьому каталозі новий файл **package.json**:

{

    "name": "mongoapp",

    "version": "1.0.0",

}

За допомогою менеджера пакетів npm встановимо пакети **express** та **mongodb**

npm install express

npm install mongodb

В даному випадку остання залежність - "**mongodb**" якраз і являє собою драйвер. Всю необхідну довідкову інформацію конкретно з даного драйверу можна знайти на <https://mongodb.github.io/node-mongodb-native/>.

**Підключення до бази даних**

Ключовим класом для роботи з MongoDB є клас **MongoClient**, і через нього буде йти всі взаємодії зі сховищем даних. Відповідно спочатку ми повинні отримати MongoClient:

const MongoClient = require("mongodb").MongoClient;

Для з'єднання з сервером mongodb застосовується метод **connect ()**:

const MongoClient = require("mongodb").MongoClient;

// создаем объект MongoClient и передаем ему строку подключения

const mongoClient = new MongoClient("mongodb://localhost:27017/", { useUnifiedTopology: true });

mongoClient.connect(function(err, client){

    if(err){

        return console.log(err);

    }

    // взаимодействие с базой данных

    client.close();

});

Спочатку створюється об'єкт MongoClient. Для цього в його конструктор передається два параметри. Перший параметр - це адреса сервера. В якості протоколу адреси встановлюється "mongodb://". На локальній машині адресою буде localhost, після якого вказується номер порту. За замовчуванням номер порту 27017.

Другий парамтр - це необов'язковий об'єт конфігурації. MongoDb постійно розвивається. В даному випадку застосовується об'єкт конфігурації, який має властивість useUnifiedTopology: true - він вказує, що треба використовувати єдину топологію драйвера для **node.js**.

Далі за допомогою методу **connect** відбувається підключення до сервера. Як параметр метод приймає функцію зворотнього виклику, яка спрацьовує при встанволенні підключення. Ця функція приймає два параметри: **err** (виникла помилка під час активного з'єднання) і **client** (посилання на підключений до сервера клієнт).

Якщо при підключенні виникли помилки, то ми можемо використовувати значення **err** для отримання помилки.

Якщо ж помилки немає, то ми можемо взаємодіяти з сервером через об'єкт **client**.

В кінці завершення роботи з БД нам треба закрити з'єднання за допомогою методу **client.close ()**.

**База даних, колекції та документи.**

Отримавши об'єкт підключеного клієнту, ми можемо звертатися до бази даних на сервері. Для цього використовується метод

client.db("назва\_бд");

Як параметр в метод передається назва бази даних, до якої ми хочемо підключитися.

База даних в MongoDB не має таблиць. Замість цього всі дані потрапляють в колекції. І в рамках node.js для взаємодії з базою даних (додавання, видалення, читання даних) нам буде потрібно отримати об'єкт колекції. Для цього застосовується метод **db.collection ( "назва\_коллекціі")**, в який передається назва колекції.

На відміну від таблиць в реляційних системах, де всі дані зберігаються у вигляді рядків, в колекціях у MongoDB дані зберігаються у вигляді документів. Наприклад, додамо в базу даних один документ. Для цього визначимо в каталозі проекту наступний файл **app.js**:

const MongoClient = require("mongodb").MongoClient;

const url = "mongodb://localhost:27017/";

const mongoClient = new MongoClient(url, { useUnifiedTopology: true });

mongoClient.connect(function(err, client){

    const db = client.db("usersdb");

    const collection = db.collection("users");

    let user = {name: "Tom", age: 23};

    collection.insertOne(user, function(err, result){

        if(err){

            return console.log(err);

        }

        console.log(result.ops);

        client.close();

    });

});

За базу даних тут використовується "**usersdb**". При цьому не важливо, що за замовчуванням на сервері MongoDB немає подібної бази даних. При першому до неї зверненні сервер автоматично її створить.

Після підключення ми звертаємося до колекції "**users**":

const collection = db.collection("users");

Знову ж неважливо, що такої колекції за замовчуванням немає в БД **usersdb**, вона також буде створена при першому зверненні.

Отримавши колекцію, ми можемо використовувати її методи. В даному випадку для додавання одного документу - об'єкта **user** застосовується метод **insertOne()**. Цей метод має два параметри - сам об'єкт що додається і функцію зворотнього виклику, яка виконується після додавання. У цій функції застосовуються два параметри: **err** (помилка, яка може виникнути під час операції) і **result** (результат операції - доданий об'єкт).

У функції зворотнього виклику інспектується доданий об'єкт за допомогою властивості **result.ops**. Причому це вже не просто об'єкт **user**, а об'єкт, який отриманий знову з бази даних і який містить ідентифікатор, встановлений при додаванні.

Вважаємо що Open Server вже запущений, якщо ні то запустіть його.

MongoDB після цього буде запущена на локальному хості.

Після цього запустимо наш файл **app.js**

>node app.js

Результат у консолі буде приблизно таким:

[ { name: 'Tom', age: 23, \_id: 6050e7cff1f3df2ba05b39f8 } ]

Як ми бачимо, крім початкових властивостей тут документ ще має додаткову властивість \_id - це унікальний ідентифікатор документу, який присвоюється сервером при додаванні.

**Додавання та отримання даних з MongoDB**

Розглянемо як додавати і отримувати дані з БД.

Для додавання ми можемо використовувати різні методи. Якщо потрібно додати один об'єкт, то застосовується метод **insertOne()**. При додаванні набору об'єктів можна використовувати метод **insertMany()**.

Метод **insertOne()** розглядався вище, тому використовуємо метод **insertMany()**. Додамо набір об'єктів і для цього змінимо файл додатка **app.js**:

const MongoClient = require("mongodb").MongoClient;

const url = "mongodb://localhost:27017/";

const mongoClient = new MongoClient(url, { useUnifiedTopology: true });

let users = [{name: "Bob", age: 34} , {name: "Alice", age: 21}, {name: "Tom", age: 45}];

mongoClient.connect(function(err, client){

    const db = client.db("usersdb");

    const collection = db.collection("users");

    collection.insertMany(users, function(err, results){

        console.log(results);

        client.close();

    });

});

Як і **insertOne()**, метод **insertMany()** в якості першого параметра приймає дані що додаються - масив об'єктів, а в якості другого - функцію зворотнього виклику, яка виконується при додаванні даних. При вдалому додаванні другий параметр функції - **results** буде містити додані дані.

Запустимо додаток у консолі.

Результат буде виглядати приблизно так:

{

result: { ok: 1, n: 3 },

ops: [

{ name: 'Bob', age: 34, \_id: 6050eb9cc86c1c3be0a28618 },

{ name: 'Alice', age: 21, \_id: 6050eb9cc86c1c3be0a28619 },

{ name: 'Tom', age: 45, \_id: 6050eb9cc86c1c3be0a2861a }

],

insertedCount: 3,

insertedIds: {

'0': 6050eb9cc86c1c3be0a28618,

'1': 6050eb9cc86c1c3be0a28619,

'2': 6050eb9cc86c1c3be0a2861a

}

}

Окрім самих даних параметр **results** буде містити деяку додаткову інформацію про виконану операцію додавання.

**Отримання даних**

Для отримання даних з колекції застосовується метод **find()**:

const MongoClient = require("mongodb").MongoClient;

const url = "mongodb://localhost:27017/";

const mongoClient = new MongoClient(url, { useUnifiedTopology: true });

mongoClient.connect(function(err, client){

    const db = client.db("usersdb");

    const collection = db.collection("users");

    if(err) return console.log(err);

    collection.find().toArray(function(err, results){

        console.log(results);

        client.close();

    });

});

Метод **find** повертає спеціальний об'єкт - [Cursor](http://mongodb.github.io/node-mongodb-native/2.2/api/Cursor.html), і щоб отримати всі дані у цього об'єкта викликається метод **toArray()**. У цей метод передається функція зворотнього виклику зі стандартними параметрами: **err** (інформація про помилку при її наявності) і **result** (власне результат вибірки).

І якщо ми запустимо додаток, то побачимо все раніше додані дані:

[

{ \_id: 6050e7cff1f3df2ba05b39f8, name: 'Tom', age: 23 },

{ \_id: 6050eb9cc86c1c3be0a28618, name: 'Bob', age: 34 },

{ \_id: 6050eb9cc86c1c3be0a28619, name: 'Alice', age: 21 },

{ \_id: 6050eb9cc86c1c3be0a2861a, name: 'Tom', age: 45 }

]

За допомогою методу **find()** ми можемо додатково відфільтрувати отримані документи. Наприклад, нам треба знайти всіх користувачів, у яких ім'я - Tom:

const MongoClient = require("mongodb").MongoClient;

const url = "mongodb://localhost:27017/";

const mongoClient = new MongoClient(url, { useUnifiedTopology: true });

mongoClient.connect(function(err, client){

    const db = client.db("usersdb");

    const collection = db.collection("users");

    if(err) return console.log(err);

    collection.find({name: "Tom"}).toArray(function(err, results){

        console.log(results);

        client.close();

    });

});

Як параметр в **find** передається об'єкт, який встановлює параметри фільтрації. Зокрема, що властивість name має дорівнювати "Tom".

[

{ \_id: 6050e7cff1f3df2ba05b39f8, name: 'Tom', age: 23 },

{ \_id: 6050eb9cc86c1c3be0a2861a, name: 'Tom', age: 45 }

]

Ми можемо встановлювати додаткові критерії фільтрації, наприклад, додамо фільтрацію за віком:

collection.find({name: "Tom", age: 23}).toArray(function(err, results){

    console.log(results);

    client.close();

});

Метод **findOne()** працює аналогічно, тільки дозволяє отримати один документ:

const MongoClient = require("mongodb").MongoClient;

const url = "mongodb://localhost:27017/";

const mongoClient = new MongoClient(url, { useUnifiedTopology: true });

mongoClient.connect(function(err, client){

    if(err) return console.log(err);

    const db = client.db("usersdb");

    db.collection("users").findOne(function(err, doc){

        console.log(doc);

        client.close();

    });

});

І також в методі **findOne()** можна застосовувати фільтрацію:

db.collection("users").findOne({name: "Bob"}, function(err, doc){

    console.log(doc);

    client.close();

});

**Видалення документів у MongoDB**

Видаляти документи в MongoDB можна різними способами. Тут треба зазначити наступні методи колекції:

* **deleteMany()**: видаляє всі документи, які відповідають певним критерієм
* **deleteOne()**: видаляє один документ, який відповідає певним критерієм
* **findOneAndDelete()**: отримує і видаляє один документ, який відповідає певним критерієм
* **drop()**: видаляє всю колекцію

Розглянемо кожен з них окремо.

***deleteMany***

Видалимо всіх користувачів, у яких ім'я "Tom":

const MongoClient = require("mongodb").MongoClient;

const url = "mongodb://localhost:27017/";

const mongoClient = new MongoClient(url, { useUnifiedTopology: true });

mongoClient.connect(function(err, client){

    if(err) return console.log(err);

    const db = client.db("usersdb");

    db.collection("users").deleteMany({name: "Tom"}, function(err, result){

        console.log(result);

        client.close();

    });

});

Перший параметр в методі - фільтр документів, а другий - функція зворотнього виклику, в якій ми можемо отримати результат видалення. При цьому результат видалення представлятиме складний об'єкт, який містить детальну інформацію.

***deleteOne***

Метод **deleteOne()** аналогічний методу **deleteMany()** за винятком того, що видаляє тільки один об'єкт:

db.collection("users").deleteOne({name: "Bob"}, function(err, result){

        console.log(result);

        client.close();

    });

***findOneAndDelete***

Метод **findOneAndDelete()** видаляє один документ за певним критерієм, але в порівнянні з методом **deleteOne()** він повертає видалений документ:

db.collection("users").findOneAndDelete({age: 21}, function(err, result){

        console.log(result);

        client.close();

    });

***drop***

Метод **drop()** видаляє всю колекцію:

db.collection("users").drop(function(err, result){

        console.log(result);

        client.close();

    });

**Оновлення документів у MongoDB**

Для оновлення елементів в MongoDB є кілька методів:

* **updateOne**: оновлює один документ, який відповідає критерію фільтрації, і повертає інформацію про операції поновлення
* **updateMany**: оновлює всі документи, які відповідають критерію фільтрації, і повертає інформацію про операції поновлення
* **findOneAndUpdate**: оновлює один документ, який відповідає критерію фільтрації, і повертає оновлений документ

**findOneAndUpdate**

Метод **findOneAndUpdate()** оновлює один елемент. Він приймає такі параметри:

1. Критерій фільтрації документа, який треба оновити;
2. Параметр оновлення;
3. Додаткові опції оновлення, які за умовчанням мають значення null;
4. Функція зворотного виклику, яка виконується при оновленні;

Наприклад, оновимо першого користувача в БД, у якого вік - 21:

const MongoClient = require("mongodb").MongoClient;

const url = "mongodb://localhost:27017/";

const mongoClient = new MongoClient(url, { useUnifiedTopology: true });

let users = [{name: "Bob", age: 34} , {name: "Alice", age: 21}, {name: "Tom", age: 45}];

mongoClient.connect(function(err, client){

    if(err) return console.log(err);

    const db = client.db("usersdb");

    const col = db.collection("users");

    col.insertMany(users, function(err, results){

        col.findOneAndUpdate(

            {age: 21}, // критерий выборки

            { $set: {age: 25}}, // параметр обновления

            function(err, result){

                console.log(result);

                client.close();

            }

        );

    });

});

Спочатку тут відбувається додавання 3 користувачів в базу даних, а після додавання йде оновлення.

Для оновлення застосовується об'єкт {$set: {age: 25}}. Параметр $set дозволяє оновити значення для одного поля або групи полів. В даному випадку змінюється поле **age**.

Третій параметр - функція зворотнього виклику виводить результат оновлення. За замовчуванням це старий стан зміненого документу:

{

lastErrorObject: { n: 1, updatedExisting: true },

value: { \_id: 60511a1e0e7b462a506b960e, name: 'Alice', age: 21 },

ok: 1

}

Але, припустимо, після поновлення ми хочемо отримувати не старий, а новий стан зміненого документу. Для цього ми можемо поставити додаткові опції оновлення.

const MongoClient = require("mongodb").MongoClient;

const url = "mongodb://localhost:27017/";

const mongoClient = new MongoClient(url, { useUnifiedTopology: true });

mongoClient.connect(function(err, client){

    if(err) return console.log(err);

    const db = client.db("usersdb");

    const col = db.collection("users");

    col.findOneAndUpdate(

        {name: "Bob"},              // критерий выборки

        { $set: {name: "Sam"}},     // параметр обновления

        {                           // доп. опции обновления

            returnOriginal: false

        },

        function(err, result){

            console.log(result);

            client.close();

        }

    );

});

{

lastErrorObject: { n: 1, updatedExisting: true },

value: { \_id: 60511bab0cedc73af07d9c1a, name: 'Sam', age: 34 },

ok: 1

}

**updateMany**

Метод **updateMany()** дозволяє оновити всі документи з колекції, які відповідають критерію фільтрації:

const MongoClient = require("mongodb").MongoClient;

const url = "mongodb://localhost:27017/";

const mongoClient = new MongoClient(url, { useUnifiedTopology: true });

mongoClient.connect(function(err, client){

    if(err) return console.log(err);

    const db = client.db("usersdb");

    const col = db.collection("users");

    col.updateMany(

        {name: "Sam"},              // критерий фильтрации

        { $set: {name: "Bob"}},     // параметр обновления

        function(err, result){

            console.log(result);

            client.close();

        }

    );

});

**updateOne**

Метод **updateOne()** аналогічний методу updateMany за тим винятком, що оновлює тільки один елемент. На відміну від методу **findOneAndUpdate()** він не повертає змінений документ:

const MongoClient = require("mongodb").MongoClient;

const url = "mongodb://localhost:27017/";

const mongoClient = new MongoClient(url, { useUnifiedTopology: true });

mongoClient.connect(function(err, client){

    if(err) return console.log(err);

    const db = client.db("usersdb");

    const col = db.collection("users");

    col.updateOne(

        {name: "Tom"},

        { $set: {name: "Tom Junior", age:33}},

        function(err, result){

            console.log(result);

            client.close();

        }

    );

});

**Express та MongoDB**

Тепер об'єднаємо в одному додатку обробку запитів за допомогою Express і роботу з даними в MongoDB. Для цього визначимо наступний файл додатка **app.js**:

const express = require("express");

const MongoClient = require("mongodb").MongoClient;

const objectId = require("mongodb").ObjectID;

const app = express();

const jsonParser = express.json();

const mongoClient = new MongoClient("mongodb://localhost:27017/", { useUnifiedTopology: true });

let dbClient;

app.use(express.static(\_\_dirname + "/public"));

mongoClient.connect(function(err, client){

    if(err) return console.log(err);

    dbClient = client;

    app.locals.collection = client.db("usersdb").collection("users");

    app.listen(3000, function(){

        console.log("Сервер ожидает подключения...");

    });

});

app.get("/api/users", function(req, res){

    const collection = req.app.locals.collection;

    collection.find({}).toArray(function(err, users){

        if(err) return console.log(err);

        res.send(users)

    });

});

app.get("/api/users/:id", function(req, res){

    const id = new objectId(req.params.id);

    const collection = req.app.locals.collection;

    collection.findOne({\_id: id}, function(err, user){

        if(err) return console.log(err);

        res.send(user);

    });

});

app.post("/api/users", jsonParser, function (req, res) {

    if(!req.body) return res.sendStatus(400);

    const userName = req.body.name;

    const userAge = req.body.age;

    const user = {name: userName, age: userAge};

    const collection = req.app.locals.collection;

    collection.insertOne(user, function(err, result){

        if(err) return console.log(err);

        res.send(user);

    });

});

app.delete("/api/users/:id", function(req, res){

    const id = new objectId(req.params.id);

    const collection = req.app.locals.collection;

    collection.findOneAndDelete({\_id: id}, function(err, result){

        if(err) return console.log(err);

        let user = result.value;

        res.send(user);

    });

});

app.put("/api/users", jsonParser, function(req, res){

    if(!req.body) return res.sendStatus(400);

    const id = new objectId(req.body.id);

    const userName = req.body.name;

    const userAge = req.body.age;

    const collection = req.app.locals.collection;

    collection.findOneAndUpdate({\_id: id}, { $set: {age: userAge, name: userName}},

         {returnOriginal: false },function(err, result){

        if(err) return console.log(err);

        const user = result.value;

        res.send(user);

    });

});

// прослушиваем прерывание работы программы (ctrl-c)

process.on("SIGINT", () => {

    dbClient.close();

    process.exit();

});

Для кожного типу запитів тут визначено свій обробник Express. І в кожному з обробників ми кожен раз звертаємося до бази даних. Щоб не відкривати і закривати підключення кожного разу при кожному запиті, ми відкриваємо підключення на самому початку і тільки після відкриття підключення запускаємо прослуховування вхідних запитів:

mongoClient.connect(function(err, client){

    if(err) return console.log(err);

    dbClient = client;

    app.locals.collection = client.db("usersdb").collection("users");

    app.listen(3000, function(){

        console.log("Сервер ожидает подключения...");

    });

});

Оскільки вся взаємодія буде йти з колекцією **users**, то отримуємо посилання на цю колекцію в локальну змінну додатку **app.locals.collection**. Потім через цю змінну ми зможемо отримати доступ до колекції в будь-якому місці програми.

В кінці роботи скрипта ми можемо закрити підключення, збережене в змінну **dbClient**:

process.on("SIGINT", () => {

    dbClient.close();

    process.exit();

});

В даному випадку ми слухаємо подію "**SIGINT**", яке генерується при натисненні комбінації **CTRL + C** в консолі, що завершить виконання скрипта.

Коли приходить GET-запит до додатка, то повертаємо у відповідь клієнту всі документи з бази даних:

app.get("/api/users", function(req, res){

    const collection = req.app.locals.collection;

    collection.find({}).toArray(function(err, users){

        if(err) return console.log(err);

        res.send(users)

    });

});

Якщо в GET-запиті передається параметр **id**, то повертаємо тільки одного користувача з бази даних з цього **id**:

app.get("/api/users/:id", function(req, res){

    const id = new objectId(req.params.id);

    const collection = req.app.locals.collection;

    collection.findOne({\_id: id}, function(err, user){

        if(err) return console.log(err);

        res.send(user);

    });

});

Коли приходить POST-запит, за допомогою парсера **jsonParser** отримуємо відправлені дані і по ним створюємо об'єкт, який додаємо в базу даних за допомогою методу **insertOne()**:

app.post("/api/users", jsonParser, function (req, res) {

    if(!req.body) return res.sendStatus(400);

    const userName = req.body.name;

    const userAge = req.body.age;

    const user = {name: userName, age: userAge};

    const collection = req.app.locals.collection;

    collection.insertOne(user, function(err, result){

        if(err) return console.log(err);

        res.send(user);

    });

});

При отриманні PUT-запиту також отримуємо відправлені дані і за допомогою методу **findOneAndUpdate()** оновлюємо дані в БД.

І в методі **app.delete()**, який спрацьовує при отриманні запиту DELETE, викликаємо метод **findOneAndDelete()** для видалення даних.

Таким чином, в кожному обработчике Express задіємо певний метод по роботі з MongoDB.

Тепер створимо в папці проекту новий каталог "**public**" і визначимо в цьому каталозі файл **index.html**:

<!DOCTYPE html>

<html>

<head>

    <meta charset="utf-8" />

    <meta name="viewport" content="width=device-width" />

    <title>Список пользователей</title>

    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" rel="stylesheet" />

</head>

<body>

    <h2>Список пользователей</h2>

    <form name="userForm">

        <input type="hidden" name="id" value="0" />

        <div class="form-group">

            <label for="name">Имя:</label>

            <input class="form-control" name="name" />

        </div>

        <div class="form-group">

            <label for="age">Возраст:</label>

            <input class="form-control" name="age" />

        </div>

        <div class="panel-body">

            <button type="submit" class="btn btn-sm btn-primary">Сохранить</button>

            <a id="reset" class="btn btn-sm btn-primary">Сбросить</a>

        </div>

    </form>

    <table class="table table-condensed table-striped table-bordered">

        <thead><tr><th>Id</th><th>Имя</th><th>возраст</th><th></th></tr></thead>

        <tbody>

        </tbody>

    </table>

    <script>

    // Получение всех пользователей

        async function GetUsers() {

            // отправляет запрос и получаем ответ

            const response = await fetch("/api/users", {

                method: "GET",

                headers: { "Accept": "application/json" }

            });

            // если запрос прошел нормально

            if (response.ok === true) {

                // получаем данные

                const users = await response.json();

                let rows = document.querySelector("tbody");

                users.forEach(user => {

                    // добавляем полученные элементы в таблицу

                    rows.append(row(user));

                });

            }

        }

        // Получение одного пользователя

        async function GetUser(id) {

            const response = await fetch("/api/users/" + id, {

                method: "GET",

                headers: { "Accept": "application/json" }

            });

            if (response.ok === true) {

                const user = await response.json();

                const form = document.forms["userForm"];

                form.elements["id"].value = user.\_id;

                form.elements["name"].value = user.name;

                form.elements["age"].value = user.age;

            }

        }

        // Добавление пользователя

        async function CreateUser(userName, userAge) {

            const response = await fetch("api/users", {

                method: "POST",

                headers: { "Accept": "application/json", "Content-Type": "application/json" },

                body: JSON.stringify({

                    name: userName,

                    age: parseInt(userAge, 10)

                })

            });

            if (response.ok === true) {

                const user = await response.json();

                reset();

                document.querySelector("tbody").append(row(user));

            }

        }

        // Изменение пользователя

        async function EditUser(userId, userName, userAge) {

            const response = await fetch("api/users", {

                method: "PUT",

                headers: { "Accept": "application/json", "Content-Type": "application/json" },

                body: JSON.stringify({

                    id: userId,

                    name: userName,

                    age: parseInt(userAge, 10)

                })

            });

            if (response.ok === true) {

                const user = await response.json();

                reset();

                document.querySelector("tr[data-rowid='" + user.\_id + "']").replaceWith(row(user));

            }

        }

        // Удаление пользователя

        async function DeleteUser(id) {

            const response = await fetch("/api/users/" + id, {

                method: "DELETE",

                headers: { "Accept": "application/json" }

            });

            if (response.ok === true) {

                const user = await response.json();

                document.querySelector("tr[data-rowid='" + user.\_id + "']").remove();

            }

        }

        // сброс формы

        function reset() {

            const form = document.forms["userForm"];

            form.reset();

            form.elements["id"].value = 0;

        }

        // создание строки для таблицы

        function row(user) {

            const tr = document.createElement("tr");

            tr.setAttribute("data-rowid", user.\_id);

            const idTd = document.createElement("td");

            idTd.append(user.\_id);

            tr.append(idTd);

            const nameTd = document.createElement("td");

            nameTd.append(user.name);

            tr.append(nameTd);

            const ageTd = document.createElement("td");

            ageTd.append(user.age);

            tr.append(ageTd);

            const linksTd = document.createElement("td");

            const editLink = document.createElement("a");

            editLink.setAttribute("data-id", user.\_id);

            editLink.setAttribute("style", "cursor:pointer;padding:15px;");

            editLink.append("Изменить");

            editLink.addEventListener("click", e => {

                e.preventDefault();

                GetUser(user.\_id);

            });

            linksTd.append(editLink);

            const removeLink = document.createElement("a");

            removeLink.setAttribute("data-id", user.\_id);

            removeLink.setAttribute("style", "cursor:pointer;padding:15px;");

            removeLink.append("Удалить");

            removeLink.addEventListener("click", e => {

                e.preventDefault();

                DeleteUser(user.\_id);

            });

            linksTd.append(removeLink);

            tr.appendChild(linksTd);

            return tr;

        }

        // сброс значений формы

        document.getElementById("reset").click(function (e) {

            e.preventDefault();

            reset();

        })

        // отправка формы

        document.forms["userForm"].addEventListener("submit", e => {

            e.preventDefault();

            const form = document.forms["userForm"];

            const id = form.elements["id"].value;

            const name = form.elements["name"].value;

            const age = form.elements["age"].value;

            if (id == 0)

                CreateUser(name, age);

            else

                EditUser(id, name, age);

        });

        // загрузка пользователей

        GetUsers();

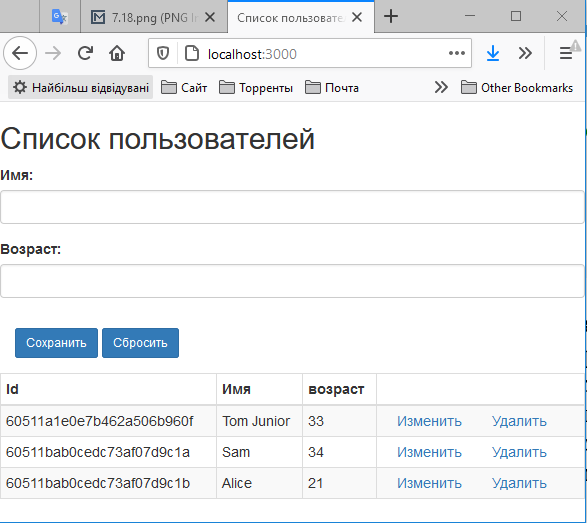
    </script>

</body>

</html>

Для спрощення відправки запитів в даному випадку застосовується бібліотека **JQuery**. І оскільки Express в якості сховища статичних файлів використовує папку public, то при зверненні до програми по кореневого маршруту <http://localhost:3000> клієнт отримає даний файл.

Запустимо додаток, звернемося до додатка за адресою <http://localhost:3000> і ми зможемо керувати користувачами, які зберігаються в базі даних MongoDB:



**Завдання для самостійної роботи**

Модіфікувати створений додаток з використанням власної бази MongoDB

**Посилання**

Сайт MongoDB

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

MongoDB Compass

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