Лабораторна робота №4

**Mongoose**

Mongoose представляє спеціальну ODM-бібліотеку (*Object Data Modelling*) для роботи з MongoDB, що дозволяє розміщувати об'єкти класів та документи колекцій з баз даних. Mongoose працює подібним інструментом ORM. Офіційний сайт бібліотеки, де можна переглянути всю необхідну документацію: <http://mongoosejs.com>.

Для роботи з Mongoose продовжуємо роботу з попереднім проектом і спочатку встановимо саму бібліотеку за допомогою команди:

npm install mongoose --save

Далі визначити наступний код у файлі додатку ***app.js***:

const mongoose = require("mongoose");

const Schema = mongoose.Schema;

// установка схемы

const userScheme = new Schema({

    name: String,

    age: Number

});

// подключение

mongoose.connect("mongodb://localhost:27017/usersdb", { useUnifiedTopology: true, useNewUrlParser: true });

const User = mongoose.model("User", userScheme);

const user = new User({

    name: "Bill",

    age: 41

});

user.save(function(err){

    mongoose.disconnect();  // отключение от базы данных

    if(err) return console.log(err);

    console.log("Сохранен объект", user);

});

Тут поперед за все нам слід підключити **mongoose**:

const mongoose = require("mongoose");

Данні, які використовуються в Mongoose, описуються певною схемою. Наприклад, у попередній роботі ми зберегали в базі даних об'єкти з двома властивостями name та age. Тому описуємо тут наступну схему:

const Schema = mongoose.Schema;

// установка схемы

const userScheme = new Schema({

    name: String,

    age: Number

});

Схема містить метаданні об'єктів. Зокрема, тут встановлюємо, які властивості буде мати об'єкт і який у них буде тип даних. Тож це схема, яка описує об'єкт користувача.

Використовуючи цю схему, створюємо модель користувача:

const User = mongoose.model("User", userScheme);

Перший параметр у методі **mongoose.model** вказує назву моделі. Mongoose потім буде автоматично шукати у базі даних колекцію, назва якої відповідає назві моделі у множині. Наприклад, у данному випадку назва моделі "**User**". У множині відповідно до англійської мови це "**users**". Тому при роботі з данними моделі **User** (додавання, видалення, редагування та отримання об'єктів) mongoose буде звертатися до колекції "**users**". Якщо така колекція є в БД, то з нею буде відбуватися взаємодія. Якщо така колекція в БД відсутня, вона буде створена автоматично.

Другий параметр функцій **mongoose.model** – власне схема.

Далі ми можемо створити об'єкти цієї моделі:

const user = new User({

    name: "Bill",

    age: 41

});

Для підключення до бази даних застосовується метод **mongoose.connect()**, до якого передається адреса бази даних на сервері mongo:

mongoose.connect("mongodb://localhost:27017/usersdb", { useUnifiedTopology: true, useNewUrlParser: true });

Другий параметр передає об'єкт, який по перше встановлює топологію для підключення до MongoDB (useUnifiedTopology: true), а по друге, встановлює використання нового парсеру (useNewUrlParser: true) для парсингу адреси серверу MongoDB.

Наприкінці у об'єкта викликається метод **save**. Цей метод, визначено для всіх створюваних моделей, він зберігає поточний об'єкт у базі даних:

user.save(function(err){

    mongoose.disconnect();  // отключение от базы данных

    if(err) return console.log(err);

    console.log("Сохранен объект", user);

});

З допомогою метода **mongoose.disconnect()**; відбувається відключення від БД.

Так як метод **save** повертає **promise**, то є і інша форма збереження об'єкту:

user.save()

.then(function(doc){

    console.log("Сохранен объект", doc);

    mongoose.disconnect();  // отключение от базы данных

})

.catch(function (err){

    console.log(err);

    mongoose.disconnect();

});

За допомогою методу **then** ми можемо отримати дані, які повертає нам сервер і виконати обробку результату.

Запустити додаток та виконаємо додавання об'єкту:

PS D:\mongoapp> node app.js

Сохранен объект { \_id: 6055f57a3c3ece339c2c0848, name: 'Bill', age: 41, \_\_v: 0 }

PS D:\mongoapp>

**Визначення схеми у Mongoose**

Схема у Mongoose визначає метаданні моделі - її властивості, типи даних та ряд іншої інформації.

У попередній темі схема визначається наступним чином:

const userScheme = new Schema({

    name: String,

    age: Number

});

**Визначення типу:**

В якості типу даних можно вказувати одне з наступних значень:

* String
* Number
* Date
* Buffer
* Boolean
* Mixed
* Objectid
* Array
* Decimal128
* Map

Якщо властивість являє собою складний об'єкт, то в якості типу вказуємо визначення цього об'єкту:

const userScheme = new Schema({

    name: String,

    age: Number,

    company: {

        name: String,

        employee: [String], // тип - массив строк

        date: Date

    }

});

Детальніше про визначення типів можна переглянути за посиланням <http://mongoosejs.com/docs/schematypes.html>.

Але назва властивостей і їх тип - це не все, що ми можемо тут визначити.

**Значення за замовчуванням**

За допомогою параметру **default** ми можемо вказати значення за замовчуванням для властивості. Наприклад:

const userScheme = new Schema({

    name: {

        type: String,

        default: "NoName"

    },

    age: {

        type: Number,

        default: 22

    }

});

Створимо трьох користувачів:

const User = mongoose.model("User", userScheme);

const user1 = new User(); // name - NoName, age - 22

const user2 = new User({name: "Tom"}); // name - Tom, age - 22

const user3 = new User({age:34}); // name - NoName, age - 34

Якщо у користувача не визначена властивість, то для нього використовується значення за замовчуваням.

**Валідація**

Mongoose має перелік вбудованих правил валідації, які ми можемо вказати у схемі:

* **required**: вимагає обов'язкової наявності значення для властивості;
* **min** та **max**: задають мінімальне та максимальне значення для числових даних;
* **minlength** і **maxlength**: задають мінімальну і максимальну довжину для рядків;
* **enum**: рядок повинен представляти одно з значень у вказаному масиві рядків;
* **match**: рядок повинен відповідати регулярному виразу.

Якщо ми спробуємо додати некоректні дані в БД,

const mongoose = require("mongoose");

const Schema = mongoose.Schema;

// подключение

mongoose.connect("mongodb://localhost:27017/usersdb", { useUnifiedTopology: true, useNewUrlParser: true });

// установка схемы

const userScheme = new Schema({

    name: {

        type: String,

        required: true,

        minlength:3,

        maxlength:20

    },

    age: {

        type: Number,

        required: true,

        min: 1,

        max:100

    }

});

const User = mongoose.model("User", userScheme);

const user = new User({name: "Li"});

user.save(function(err){

    mongoose.disconnect();

    if(err) return console.log(err);

    console.log("Сохранен объект user", user);

});

то запит на додавання поверне помилку.

При запуску на консолі ми повідомимо детальну інформацію про помилку:

Error: User validation failed: age: Path `age` is required., name: Path `name` (`Li`) is shorter than the minimum allowed length (3).

і тому подібне…

**Версія документу**

За замовчуванням під час збереження даних Mongoose додає спеціальне поле **\_\_v**, яке вказує версію документу:

const user = new User({name: "Tom", age: 34});

user.save(function(err){

    mongoose.disconnect();

    if(err) return console.log(err);

    console.log("Сохранен объект user", user);

});

Сохранен объект user { \_id: 605607f4150c40139842dc78, name: 'Tom', age: 34, \_\_v: 0 }

Не завжди таке поле необхідно, і його можно відключити, додавши в схему об'єкту **{versionKey: false}**:

const userScheme = new Schema(

    {name: String, age: Number},

    { versionKey: false }

);

**CRUD у Mongoose**

Розглянемо, як виконувати основні операції з данними в Mongoose.

**Створення документів**

Вище в загальних рисах було описано створення та додавання об'єктів у Mongoose. Зокрема, у об'єкту моделі ми можемо викликати метод **save()**:

const user = new User({name: "Tom", age: 34});

user.save(function(err){

    mongoose.disconnect();

    if(err) return console.log(err);

    console.log("Сохранен объект user", user);

});

Окрім цього методу також можна використовувати метод **User.create()**:

const mongoose = require("mongoose");

const Schema = mongoose.Schema;

// подключение

mongoose.connect("mongodb://localhost:27017/usersdb", { useUnifiedTopology: true, useNewUrlParser: true });

// установка схемы

const userScheme = new Schema({

    name: {

        type: String,

        required: true,

        minlength:3,

        maxlength:20

    },

    age: {

        type: Number,

        required: true,

        min: 1,

        max:100

    }

});

const User = mongoose.model("User", userScheme);

User.create({name: "Tom", age: 34}, function(err, doc){

    mongoose.disconnect();

    if(err) return console.log(err);

    console.log("Сохранен объект user", doc);

});

Перший параметр методу - об'єкт, що зберігається.

**Отримання даних**

Для отримання даних можна використовувати цілий набір методів:

* **find**: повертає всі об'єкти, які відповідають критеріям фільтрації;
* **findById**: повертає один об'єкт за допомогою поля \_id;
* **findOne**: повертає один об'єкт, який відповідає критерію фільтрації.

Метод **find()** у якості першого параметру приймає критерій фільтрації, а другий параметр - функція зворотнього виклику, у яку передаються отримані з БД документи:

const mongoose = require("mongoose");

const Schema = mongoose.Schema;

mongoose.connect("mongodb://localhost:27017/usersdb", { useUnifiedTopology: true, useNewUrlParser: true });

const userScheme = new Schema({name: String, age: Number}, {versionKey: false});

const User = mongoose.model("User", userScheme);

User.find({}, function(err, docs){

    mongoose.disconnect();

    if(err) return console.log(err);

    console.log(docs);

});

Якщо в якості критерію фільтрації передаються пусті фігурні дужки ({}), то повертаються всі об'єкти:

[

{ \_id: 60511a1e0e7b462a506b960f, name: 'Tom Junior', age: 33 },

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

{ \_id: 60511bab0cedc73af07d9c1b, name: 'Alice', age: 21 },

{ \_id: 6055f57a3c3ece339c2c0848, name: 'Bill', age: 41, \_\_v: 0 },

{ \_id: 605607f4150c40139842dc78, name: 'Tom', age: 34, \_\_v: 0 }

]

Змінемо код для отримання тільки тих користувачів, у яких ім'я - Том:

User.find({name: "Tom"}, function(err, docs){

    mongoose.disconnect();

    if(err) return console.log(err);

    console.log(docs);

});

Метод **findOne()** працює аналогічно методу **find**, лише повертає один об'єкт:

User.findOne({name: "Tom"}, function(err, doc){

    mongoose.disconnect();

    if(err) return console.log(err);

    console.log(doc);

});

Метод **findById()** повертає документ із визначеним ідентифікатором:

const id = "588a0f8c9151d23ce47bf98d";

User.findById(id, function(err, doc){

    mongoose.disconnect();

    if(err) return console.log(err);

    console.log(doc);

});

**Видалення даних**

Для видалення застосовується метод **deleteOne()** (видаляє один об'єкт) і **deleteMany()** (видаляє всі об'єкти, що відповідають критеріям). У цих методах передається критерій фільтрації документів на видалення. Наприклад, видалимо всіх користувачів, у яких вік дорівнює 33:

const mongoose = require("mongoose");

const Schema = mongoose.Schema;

mongoose.connect("mongodb://localhost:27017/usersdb", { useUnifiedTopology: true, useNewUrlParser: true });

const userScheme = new Schema({name: String, age: Number}, {versionKey: false});

const User = mongoose.model("User", userScheme);

User.deleteMany({age:33}, function(err, result){

    mongoose.disconnect();

    if(err) return console.log(err);

    console.log(result);

});

Об'єкт **result**, який передається у функцію зворотнього виклику, містить інформацію про операцію видалення:

{ n: 1, ok: 1, deletedCount: 1 }

Застосування методу **deleteOne()** для видалення одного документу буде аналогічним:

User.deleteOne({name:"Bill"}, function(err, result){

    mongoose.disconnect();

    if(err) return console.log(err);

    console.log(result);

});

Також для видалення одного документу можна використовувати метод **findOneAndDelete()**:

User.findOneAndDelete({name:"Bill"}, function(err, doc){

    mongoose.disconnect();

    if(err) return console.log(err);

    console.log("Удален пользователь ", doc);

});

Тут у функцію зворотнього виклику передається видалений документ.

І різновид цього методу - видалення по полю \_id у вигляді методу **findByIdAndDelete()**:

var id = "588a0f8c9151d23ce47bf98d";

User.findByIdAndDelete(id, function(err, doc){

    mongoose.disconnect();

    if(err) return console.log(err);

    console.log("Удален пользователь ", doc);

});

**Зміна даних**

Для оновлення даних у моделях передбачених методів **updateOne()** та **updateMany()**. Перший метод оновлює один документ, який відповідає критеріям, а другий метод оновлює всі документи, що відповідають критеріям вибору:

const mongoose = require("mongoose");

const Schema = mongoose.Schema;

mongoose.connect("mongodb://localhost:27017/usersdb", { useUnifiedTopology: true, useNewUrlParser: true });

const userScheme = new Schema({name: String, age: Number}, {versionKey: false});

const User = mongoose.model("User", userScheme);

User.updateOne({name: "Tom"}, {name: "Tom Smith"}, function(err, result){

    mongoose.disconnect();

    if(err) return console.log(err);

    console.log(result);

});

Перший параметр методу - критерій фільтрації. У даному випадку ми знаходимо всіх користувачів, за ім'ям "Tom". Другий параметр описує, що і як надо змінити. Тобто тут ми змінюємо ім'я на "Tom Smith". У функцію зворотнього виклику передається результат операції:

{ n: 1, nModified: 1, ok: 1 }

Аналогічним чином працює метод **updateMany()**.

Доволі часто для оновлення використовується фільтрація за \_id. У цьому випадку ми можемо використовувати метод **findByIdAndUpdate()**:

mongoose.set("useFindAndModify", false);

const id = "60511bab0cedc73af07d9c1b";

User.findByIdAndUpdate(id, {name: "Sam", age: 25}, function(err, user){

    mongoose.disconnect();

    if(err) return console.log(err);

    console.log("Обновленный объект", user);

});

Перший параметр методу - значення для поля \_id документа що оновлюється, а другий - набір нових значень для полів об'єкту. У функції зворотнього виклику передається оновлений документ.

Слід враховувати, що для виконання цих методів визначається налаштування **useFindAndModify = false:**

mongoose.set("useFindAndModify", false);

Це налаштування також можна встановити при створенні підключення:

mongoose.connect("mongodb://localhost:27017/usersdb",

                { useUnifiedTopology: true, useNewUrlParser: true, useFindAndModify: false },

                function(err){

    //...............

});

Але за замовченям передається старий стан документу. Якщо ж нам потрібно отримати документ вже у зміненому стані, то у метод **findByIdAndUpdate** необхідно передати ще один параметр:

mongoose.set("useFindAndModify", false);

User.findByIdAndUpdate(id, {name: "Bilbo Baggins", age: 111}, {new: true}, function(err, user){

    mongoose.disconnect();

    if(err) return console.log(err);

    console.log("Обновленный объект", user);

});

Якщо нам необхідно оновити та повернути оновлений документ не лише за **id**, а взагалі за будь-яким критерієм, можна використовувати метод **findOneAndUpdate**:

mongoose.set("useFindAndModify", false);

User.findOneAndUpdate({name: "Sam"}, {name: "Bill", age:24}, {new: true}, function(err, user){

    mongoose.disconnect();

    if(err) return console.log(err);

    console.log("Обновленный объект", user);

});

Перший параметр представляє критерій вибору. Другий параметр представляє оновлені значення документа. Третій параметр вказує, що ми хочемо повернути варіант документа саме після оновлення - **{new: true}** - при значенні **false** повертається стара копія, яка була до оновлення. І четвертий параметр - функція обробника, що викликається під час отримання результату.

**Express та Mongoose**

Розглянемо, як використати Mongoose та Express і виконувати операції з данними, коли приходять ті чи інші запити до серверу. Для цього визначимо наступний файл додатків app.js:

const mongoose = require("mongoose");

const express = require("express");

const Schema = mongoose.Schema;

const app = express();

const jsonParser = express.json();

const userScheme = new Schema({name: String, age: Number}, {versionKey: false});

const User = mongoose.model("User", userScheme);

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

mongoose.connect("mongodb://localhost:27017/usersdb", { useUnifiedTopology: true, useNewUrlParser: true, useFindAndModify: false }, function(err){

    if(err) return console.log(err);

    app.listen(3000, function(){

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

    });

});

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

    User.find({}, function(err, users){

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

        res.send(users)

    });

});

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

    const id = req.params.id;

    User.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 = new User({name: userName, age: userAge});

    user.save(function(err){

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

        res.send(user);

    });

});

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

    const id = req.params.id;

    User.findByIdAndDelete(id, function(err, user){

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

        res.send(user);

    });

});

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

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

    const id = req.body.id;

    const userName = req.body.name;

    const userAge = req.body.age;

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

    User.findOneAndUpdate({\_id: id}, newUser, {new: true}, function(err, user){

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

        res.send(user);

    });

});

По суті тут виконуються всі операції, які були розглянуті вище. Єдине, що можна відзначити, це запуск серверу після вдалого підключення до бази даних **usersdb** у функції **mongoose.connect**:

mongoose.connect("mongodb://localhost:27017/usersdb", { useUnifiedTopology: true, useNewUrlParser: true, useFindAndModify: false }, function(err){

    if(err) return console.log(err);

    app.listen(3000, function(){

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

    });

});

Тепер створимо в папці проекту нову теку "**public**" та визначемо у цій теці 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();

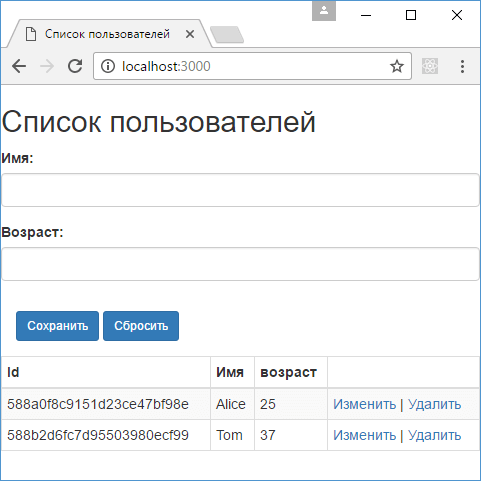
    </script>

</body>

</html>

Оскільки Express у якості сховища статичних файлів використовує теку **public**, то під час звернення до додатку за кореневим маршрутом <http://localhost:3000> клієнт отримає данний файл.

Запустимо додаток та звернемося до нього за адресою <http://localhost:3000> і ми маємо змогу взаємодіяти з БД MongoDB за допомогою Mongoose:



**Завдання для самостійної роботи**

Модіфікувати створений додаток з використанням mongoose і власної бази MongoDB

**Посилання**

Сайт mongoose

<https://mongoosejs.com/>