Лабораторная работа №8. Хранение данных в БД.

Целью данной работы является развитие приложения разработанного в рамках предыдущей лабораторной работе. В теоретической части показывается то, каким образом организовать работу веб-сервера на базе Node.js и Express, при этом предполагается наличие установленной СУБД MongoDB. В практической части следует продолжить выполнение индивидуального задания из предыдущей лабораторной работы с учётом требований заданных к практической части в данной лабораторной работе.

Теоретическая часть

MongoDB

Наиболее популярной системой управления базами данных для Node.js на данный момент является MongoDB. Для работы с этой платформой прежде всего необходимо установить сам сервер MongoDB. Кроме самого сервера Mongo для взаимодействия с Node.js нам необходим драйвер.

```
npm install mongodb
```

Mongoose

Mongoose представляет специальную ODM-библиотеку (Object Data Modelling) для работы с MongoDB, которая позволяет сопоставлять объекты классов и документы коллекций из базы данных. Грубо говоря, Mongoose работает подобно инструментам ORM. Официальный сайт библиотеки, где можно посмотреть всю необходимую документацию: http://mongoosejs.com.

CRUD в Mongoose

Рассмотрим, как выполнять основные операции с данными в Mongoose.

Создание документов

У объекта модели мы можем вызвать метод save():

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

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

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

async function main() {

    await mongoose.connect("mongodb://127.0.0.1:27017/usersdb");

    const tom = new User({name: "Tom", age: 34});
    // добавляем объект в БД
    await tom.save();
    console.log(tom);
}
main().catch(console.log).finally(async()=>await mongoose.disconnect());
```

Но кроме этого метода также можно использовать метод User.create():

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

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

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

async function main() {

    await mongoose.connect("mongodb://127.0.0.1:27017/usersdb");

    // добавляем объект в БД
    const user = await User.create({name: "Sam", age: 28})
    console.log(user);
}
main().catch(console.log).finally(async()=>await mongoose.disconnect());
```

В качестве параметра метод User.create() принимает сохраняемый объект и возвращает сохранённый объект.

Получение данных

Для получения данных можно использовать целый набор методов:

- find: возвращает все объекты, которые соответствуют критерию фильтрации
- findById: возвращает один объект по значению поля _id
- findOne: возвращает один объект, который соответствует критерию фильтрации

Метод find() в качестве первого параметра принимает критерий фильтрации, а второй параметр - функция обратного вызова, в которую передаются полученные из бд документы:

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

const userScheme = new Schema({name: String, age: Number});
const User = mongoose.model("User", userScheme);

async function main() {

    await mongoose.connect("mongodb://127.0.0.1:27017/usersdb");

    // получаем все объекты из БД
    const users = await User.find({});
    console.log(users);
}
main().catch(console.log).finally(async()=>await mongoose.disconnect());
```

Если в качестве критерия фильтрации передаются пустые фигурные скобки ({}), то возвращаются все объекты:

```
[
  {
    _id: new ObjectId("6377c17b71c0bd75cec4d488"),
    name: 'Bill',
    age: 41,
    __v: 0
  },
  {
```

```
    _id: new ObjectId("6377c7a46fa33e19ac7a7c41"),
    name: 'Tom',
    age: 34,
    __v: 0
  },
  {
    _id: new ObjectId("6377ce352461051cdc78252a"),
    name: 'Sam',
    age: 28,
    __v: 0
  }
]
```

Изменим код для получения только тех пользователей, у которых имя - Tom:

```
const users = await User.find({name: "Tom"});
```

Метод findOne() работает аналогично методу find, только возвращает один объект:

```
const user = await User.findOne({name: "Bill"});
```

И метод findById() возвращает документ с определенным идентификатором:

```
const id = "6377c7a46fa33e19ac7a7c41";
const user = await User.findById(id);
```

Удаление данных

Для удаления применяется метод deleteOne() (удаляет один объект) и deleteMany() (удаляет все объекты, которые соответствуют критерию). В эти методы передаётся критерий фильтрации документов на удаление. Например, удалим всех пользователей, у которых возраст равен 41:

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

const userScheme = new Schema({name: String, age: Number});
const User = mongoose.model("User", userScheme);

async function main() {

    await mongoose.connect("mongodb://127.0.0.1:27017/usersdb");

    // удаляем все объекты из БД, у которых age=41
    const result = await User.deleteMany({age:41});
    console.log(result);
}
main().catch(console.log).finally(async()=>await mongoose.disconnect());
```

Метод User.deleteMany() возвращает объект, который содержит информацию об операции удаления:

```
{ acknowledged: true, deletedCount: 1 }
```

Так, свойства deletedCount хранит количество удалённых строк
Применение метода deleteOne() для удаления одного документа будет аналогичным:

```
const result = await User.deleteOne({name:"Tom"})
console.log(result);    // { acknowledged: true, deletedCount: 1 }
```

Также для удаления одного документа можно использовать метод findOneAndDelete():

```
const user = await User.findOneAndDelete({name:"Sam"})
console.log(user);
```

В качестве результата он возвращает удалённый документ.

```
{ _id: new ObjectId("6377bca2d16bfca92631cc10"), name: 'Sam', age: 28 }
```

И частная разновидность этого метода - удаление по полю _id в виде метода findByIdAndDelete():

```
const id = "6377c72806fb915eb6621ffd";
const user = await User.findByIdAndDelete(id)
console.log(user);
```

Изменение данных

Для обновления данных в модели предусмотрены методы updateOne() и updateMany(). Первый метод обновляет один документ, который соответствует критерию, а второй метод обновляет все документы, которые соответствую критерию выборки:

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

const userScheme = new Schema({name: String, age: Number});
const User = mongoose.model("User", userScheme);

async function main() {

    await mongoose.connect("mongodb://127.0.0.1:27017/usersdb");

    // У всех документов изменяем значение поля name с "Tom" на "Tom Smith"
    const result = await User.updateOne({name: "Tom"}, {name: "Tom Smith"})
    console.log(result);
}
main().catch(console.log).finally(async()=>await mongoose.disconnect());
```

Первый параметр метода - критерий фильтрации. В данном случае мы находим всех пользователей, у которых имя "Tom". А второй параметр описывает, что и как надо изменить. То есть здесь мы меняем имя на "Tom Smith". Возвращает метод результат операции обновления:

```
{
  acknowledged: true,
  modifiedCount: 1,
  upsertedId: null,
  upsertedCount: 0,
  matchedCount: 1
}
```

Аналогично работает метод updateMany.

Обновление по id

Нередко для обновления используется фильтрация по _id. И на этот случай мы можем использовать метод findByIdAndUpdate():

```
const id = "6377ce352461051cdc78252a";
const user = await User.findByIdAndUpdate(id, {name: "Sam", age: 25});
console.log("Обновленный объект", user);
```

Первый параметр метода - значения для поля _id у обновляемого документа, а второй - набор новых значений для полей объекта. Результатом метода является обновленный документ:

```
Обновленный объект {
  _id: new ObjectId("6377ce352461051cdc78252a"),
  name: 'Sam',
  age: 28,
  __v: 0
}
```

Но по умолчанию передаётся старое состояние документа. Если же нам надо получить документ уже в изменённом состоянии, то в метод findByIdAndUpdate необходимо передать в качестве третьего параметра объект {new: true} (при значении false возвращается старая копия):

```
const id = "6377ce352461051cdc78252a";
const user = await User.findByIdAndUpdate(id,
                                          {name: "Mike", age: 21}, {new: true});
console.log("Обновленный объект", user);
```

Если нам необходимо обновить и возвратить обновленный документ не только по id, а вообще по любому критерию, то можно использовать метод findOneAndUpdate:

```
const user = await User.findOneAndUpdate(
                           {name: "Mike"},
                           {name: "Alex", age:24}, {new: true});
console.log("Обновленный объект", user);
```

Первый параметр представляет критерий выборки. Второй параметр представляет обновлённые значения документа. Третий параметр указывает, что мы хотим возвратить вариант документа именно после обновления - {new: true}. Результат метода - обновленный документ.

Пример: Express и Mongoose

Рассмотрим, как совместить Mongoose и Express и выполнять операции с данными, когда приходят те или иные запросы к серверу. Для этого определим следующий файл приложения app.js:

```
const mongoose = require("mongoose");
const express = require("express");
const Schema = mongoose.Schema;
const app = express();

app.use(express.static("public"));
app.use(express.json());

const userScheme = new Schema({name: String, age: Number}, {versionKey: false});
const User = mongoose.model("User", userScheme);
```

```
async function main() {

    try{
        await mongoose.connect("mongodb://127.0.0.1:27017/usersdb");
        app.listen(3000);
        console.log("Сервер ожидает подключения...");
    }
    catch(err) {
        return console.log(err);
    }
}

app.get("/api/users", async (req, res)=>{
    // получаем всех пользователей
    const users = await User.find({});
    res.send(users);
});

app.get("/api/users/:id", async(req, res)=>{

    const id = req.params.id;
    // получаем одного пользователя по id
    const user = await User.findById(id);
    if(user) res.send(user);
    else res.sendStatus(404);
});

app.post("/api/users", jsonParser, async (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});
    // сохраняем в бд
    await user.save();
    res.send(user);
});

app.delete("/api/users/:id", async(req, res)=>{

    const id = req.params.id;
    // удаляем по id
    const user = await User.findByIdAndDelete(id);
    if(user) res.send(user);
    else res.sendStatus(404);
});

app.put("/api/users", jsonParser, async (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};
     // обновляем данные пользователя по id
    const user = await User.findOneAndUpdate({_id: id}, newUser, {new: true});
    if(user) res.send(user);
    else res.sendStatus(404);
});
```

```
main();
// прослушиваем прерывание работы программы (ctrl-c)
process.on("SIGINT", async() => {

    await mongoose.disconnect();
    console.log("Приложение завершило работу");
    process.exit();
});
```

По сути здесь производятся все те операции, которые были рассмотрены в прошлой теме. Единственное, что можно отметить, это запуск сервера после удачного подключения к базе данных usersdb в функции mongoose.connect:

```
await mongoose.connect("mongodb://127.0.0.1:27017/usersdb");
app.listen(3000);
console.log("Сервер ожидает подключения...");
```

Теперь создадим в папке проекта новый каталог "public" и определим в этом каталоге файл index.html.
В файле index.html определим следующий код:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <style>
    td, th {padding:5px;min-width:90px;max-width:200px; text-align:start;}
    .btn {padding:4px; border:1px solid #333; background-color: #eee; border-
radius: 2px; margin:5px; cursor:pointer;}
    </style>
</head>
<body>
<h2>Список пользователей</h2>
<form name="userForm">
    <input type="hidden" name="id" value="0" />
    <p>
        <label>Имя:</label><br>
        <input name="name" />
    </p>
    <p>
        <label>Возраст:</label><br>
        <input name="age" type="number" />
    </p>
    <p>
        <button id="submitBtn" type="submit">Сохранить</button>
        <button id="resetBtn">Сбросить</button>
    </p>
</form>
<table>
    <thead><tr><th>Id</th><th>Имя</th><th>Возраст</th><th></th></tr></thead>
    <tbody></tbody>
</table>
<script>
const tbody = document.querySelector("tbody");
// 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();
        users.forEach(user => {
            // добавляем полученные элементы в таблицу
            tbody.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();
        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"];
    console.log(form);
    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("class", "btn");
    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("class", "btn");
    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("resetBtn").addEventListener("click", 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>
```

Код index.html вкратце обсуждался в примере к прошлой лабораторной, здесь же весь код практически повторяется.
И поскольку Express в качестве хранилища статических файлов использует папку public, то при обращении к приложению по корневому маршруту http://localhost:3000 клиент получит данный файл.
Запустим приложение, обратимся к приложению по адресу http://localhost:3000 и мы сможем взаимодействовать с базой данных MongoDB через Mongoose, рисунок 8.1:

Список пользователей

Имя:

Возраст:

Сохранить Сбросить

Id	Имя	Возраст		
65607cbb78bf2f22f32aa3c3	Bob	34	Изменить	Удалить
65607cbb78bf2f22f32aa3c5	Tom	45	Изменить	Удалить

Рис. 8.1 Express и Mongoose для работы с БД MongoDB в приложении на Node.js

Практическая часть

1. Ознакомиться с теоретической частью.[14]
2. Доработать веб сервер, разработанный в рамках предыдущей лабораторной работы, обеспечив хранение данных не в массиве на сервере, а в СУБД MongoDB, которая установлена локально.
3. Использовать серверную часть учебного пример как основу для реализации запросов к базе данных.
4. Реализовать CRUD-запросы по отношению к данным хранящимся на сервере в СУБД MongoDB.
5. Для работы с MongoDB использовать Mongoose.
6. Для конструирования содержимого html-документов использовать jQuery.

14 Расширенные теоретические материалы Лекция №8.