**Тема 17. Node и MongoDB**

1. **Начало работы с MongoDB.**
2. **Добавление и получение данных в MongoDB.**
3. **Удаление документов в MongoDB.**
4. **Обновление документов в MongoDB.**
5. **Express и MongoDB.**
6. **Mongoose.**
7. **Определение схемы в Mongoose.**
8. **CRUD в Mongoose.**
9. **Express и Mongoose.**

Содержание данной темы включает материалы, доступные по адресу https://metanit.com.

1. Начало работы с 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 необходим [драйвер](https://www.npmjs.com/package/mongodb).

При подключении и взаимодействии с бд в MongoDB можно выделить следующие этапы:

1. Подключение к серверу.
2. Получение объекта базы данных на сервере.
3. Получение объекта коллекции в базе данных.
4. Взаимодействие с коллекцией (добавление, удаление, получение, изменение данных).

Cоздадим новый проект. Для этого определим новый каталог, который будет называться mongoapp. Далее определим в этом каталоге новый файл package.json:

|  |  |
| --- | --- |
|  | {    "name": "mongoapp",    "version": "1.0.0",    "dependencies": {        "express": "^4.16.0",        "body-parser": "^1.18.0",        "mongodb": "^3.1.0"      }  } |

В данном случае последняя зависимость – "mongodb" как раз и представляет драйвер. Всю необходимую справочную информацию конкретно по данному драйверу можно найти на <https://mongodb.github.io/node-mongodb-native/>. Далее перейдем к этому каталогу в командной строке/терминале и для добавления всех нужных пакетов выполним команду:

npm install

**Подключение к базе данных**

Ключевым классом для работы с MongoDB является класс MongoClient, и через него будет идти все взаимодействия с хранилищем данных. Соответственно вначале надо получить MongoClient:

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

Для подключения к серверу mongodb применяется метод connect():

|  |  |
| --- | --- |
|  | const MongoClient = require("mongodb").MongoClient;    // создаем объект MongoClient и передаем ему строку подключения  const mongoClient = new MongoClient("<mongodb://localhost:27017/>", { useNewUrlParser: true });  mongoClient.connect(function(err, client){        if(err){          return console.log(err);      }      // взаимодействие с базой данных      client.close();  }); |

Вначале создается объект MongoClient. Для этого в его конструктор передается два параметра. Первый параметр – это адрес сервера. В качестве протокола адреса устанавливается "mongodb://". На локальной машине адресом будет localhost, после которого указывается номер порта. По умолчанию номер порта 27017.

Второй параметр – это необязательный объект конфигурации. MongoDb постоянно развивается. В данном случае применяется объект конфигурации, который имеет свойство useNewUrlParser: true – оно указывает инфраструктуре mongodb, что надо использовать новый парсер адреса сервера.

Далее с помощью метода connect происходит подключение к серверу. В качестве параметра метод принимает функцию обратного вызова, которая срабатывает при установке подключения. Это функция принимает два параметра: err (возникшая ошибка при установке соединения) и client (ссылка на подключенный к серверу клиент).

Если при подключении возникли ошибки, то можно использовать значение err для получения ошибки. Если же ошибки нет, то можно взаимодействовать с сервером через объект client. В конце завершения работы с бд надо закрыть соединение с помощью метода client.close().

**База данных, коллекции и документы**

Получив объект подключенного клиента, можно обращаться к базе данных на сервере. Для этого используется метод:

|  |  |
| --- | --- |
|  | client.db("название\_бд"); |

В качестве параметра в метод передается название базы данных, к которой надо подключиться.

База данных в MongoDB не имеет таблиц. Вместо этого все данные попадают в коллекции. И в рамках node.js для взаимодействия с базой данных (добавления, удаления, чтения данных) потребуется получить объект коллекции. Для этого применяется метод db.collection("название\_коллекции"), в который передается название коллекции.

В отличие от таблиц в реляционных системах, где все данные хранятся в виде строк, в коллекциях в MongoDB данные хранятся в виде документов. Например, добавим в базу данных один документ. Для этого определим в каталоге проекта следующий файл app.js:

|  |  |
| --- | --- |
|  | const MongoClient = require("mongodb").MongoClient;    const url = "<mongodb://localhost:27017/>";  const mongoClient = new MongoClient(url, { useNewUrlParser: 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();      });  }); |

В качестве базы данных здесь используется "usersdb". При этом не важно, что по умолчанию на сервере MongoDB нет подобной базы данных. При первом к ней обращении сервер автоматически ее создаст.

После подключения обращаемся к коллекции "users":

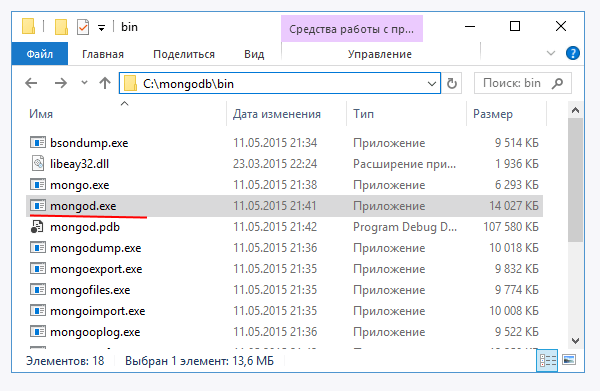
|  |  |
| --- | --- |
|  | const collection = db.collection("users"); |

Опять же неважно, что такой коллекции по умолчанию нет в бд usersdb, она также будет создана при первом обращении.

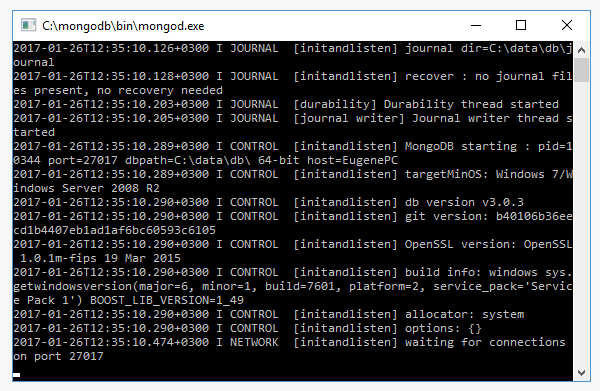
Получив коллекцию, можно использовать ее методы. В данном случае для добавления одного документа – объекта user применяется метод insertOne(). Этот метод имеет два параметра – сам добавляемый объект и функцию обратного вызова, которая выполняется после добавления. В этой функции применяются два параметра: err (ошибка, которая может возникнуть при операции) и result (результат операции – добавленный объект).

В функции обратного вызова инспектируется добавленный объект с помощью свойства result.ops. Причем это уже не просто объект user, а объект, который получен обратно из базы данных и который содержит идентификатор, установленный при добавлении.

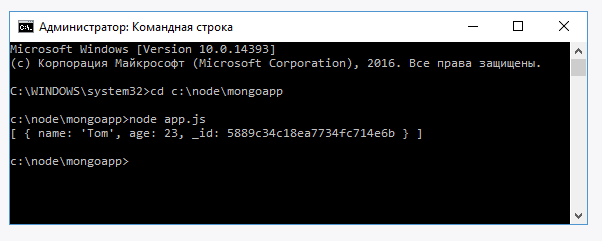
Теперь перейдем на жестком диске к каталогу, в который установлена mongodb, а в этом каталоге перейдем к папке bin:



Запустим сервер mongodb, который находится в этом каталоге и который представляет собой консольную программу mongod.



Затем запустим наш файл app.js:



Как видим, кроме начальных свойств здесь документ еще имеет дополнительное свойство \_id – это уникальный идентификатор документа, который присваивается сервером при добавлении.

1. Добавление и получение данных в MongoDB.

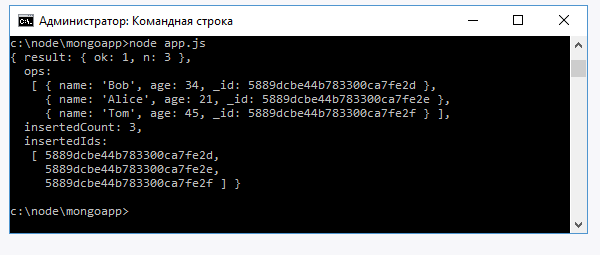
В прошлой теме были рассмотрены некоторые базовые моменты по работе с MongoDB. Теперь более подробно рассмотрим, как добавлять и получать данные из бд.

Для добавления можно использовать различные методы. Если нужно добавить один объект, то применяется метод insertOne(). При добавлении набора объектов можно использовать метод insertMany(). Метод insertOne() рассматривался в прошлом вопросе, поэтому используем метод insertMany(). Добавим набор объектов и для этого изменим файл приложения app.js:

|  |  |
| --- | --- |
|  | const MongoClient = require("mongodb").MongoClient;  const url = "<mongodb://localhost:27017/>";  const mongoClient = new MongoClient(url, { useNewUrlParser: 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();      });  }); |

Как и insertOne, метод insertMany() в качестве первого параметра принимает добавляемые данные – массив объектов, а в качестве второго – функцию обратного вызова, которая выполняется при добавлении данных. При удачном добавлении второй параметр функции – results будет содержать добавленные данные.

Запустим приложение:



Кроме самих данных параметр results будет содержать некоторую дополнительную информацию о выполненной операции добавления.

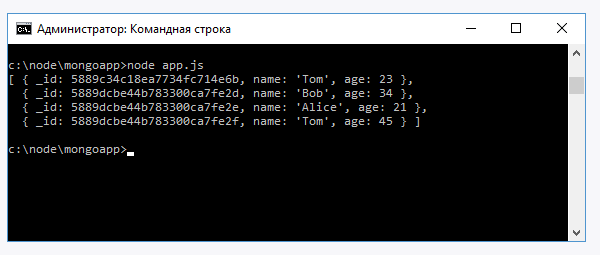
**Получение данных**

Для получения данных из коллекции применяется метод find():

|  |  |
| --- | --- |
|  | const MongoClient = require("mongodb").MongoClient;  const url = "<mongodb://localhost:27017/>";  const mongoClient = new MongoClient(url, { useNewUrlParser: 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 возвращает специальный объект – [Cursor](http://mongodb.github.io/node-mongodb-native/2.2/api/Cursor.html), и чтобы получить все данные у этого объекта вызывается метод toArray(). В этот метод передается функция обратного вызова со стандартными параметрами: err (информация об ошибке при ее наличии) и result (собственно результат выборки).

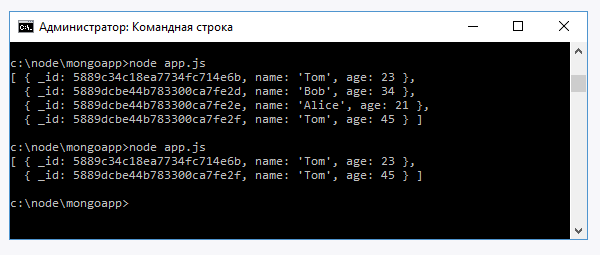
И если запустить приложение, то увидим все ранее добавленные данные:



С помощью метода find() можно дополнительно отфильтровать извлекаемые документы. Например, надо найти всех пользователей, у которых имя – Tom:

|  |  |
| --- | --- |
|  | const MongoClient = require("mongodb").MongoClient;  const url = "<mongodb://localhost:27017/>";  const mongoClient = new MongoClient(url, { useNewUrlParser: 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();      });  }); |

В качестве параметра в find передается объект, который устанавливает параметры фильтрации. В частности, что свойство name должно быть равно "Tom".



Можно устанавливать дополнительные критерии фильтрации, например, добавим фильтрацию по возрасту:

|  |  |
| --- | --- |
|  | collection.find({name: "Tom", age: 23}).toArray(function(err, results){      console.log(results);      client.close();  }); |

Метод findOne() работает аналогично, только позволяет получить один документ:

|  |  |
| --- | --- |
|  | const MongoClient = require("mongodb").MongoClient;  const url = "<mongodb://localhost:27017/>";  const mongoClient = new MongoClient(url, { useNewUrlParser: 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();      });  }); |

И также в методе findOne() можно применять фильтрацию:

|  |  |
| --- | --- |
|  | db.collection("users").findOne({name: "Bob"}, function(err, doc){      console.log(doc);      client.close();  }); |

1. Удаление документов в MongoDB.

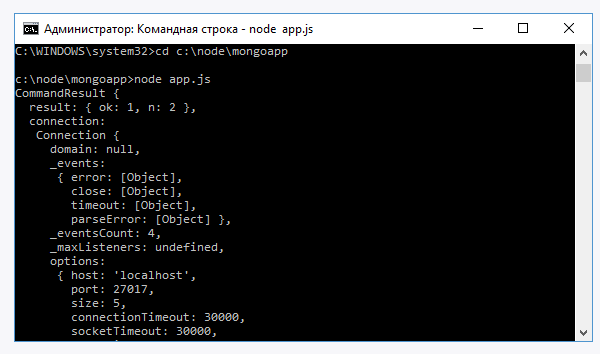
Удалять документы в MongoDB можно различными способами. Здесь надо отметить следующие методы коллекции:

* deleteMany(): удаляет все документы, которые соответствуют определенному критерию;
* deleteOne(): удаляет один документ, который соответствует определенному критерию;
* findOneAndDelete(): получает и удаляет один документ, который соответствует определенному критерию;
* drop(): удаляет всю коллекцию.

*Метод deleteMany.* Удалим всех пользователей, у которых имя "Tom":

|  |  |
| --- | --- |
|  | const MongoClient = require("mongodb").MongoClient;  const url = "<mongodb://localhost:27017/>";  const mongoClient = new MongoClient(url, { useNewUrlParser: 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();      });  }); |

Первый параметр в методе – фильтр документов, а второй – функция обратного вызова, в которой можно получить результат удаления. При этом результат удаления будет представлять сложный объект, содержащий подробную информацию:



*Метод deleteOne.* Метод deleteOne() аналогичен методу deleteMany() за тем исключением, что удаляет только один объект:

|  |  |
| --- | --- |
|  | const MongoClient = require("mongodb").MongoClient;  const url = "<mongodb://localhost:27017/>";  const mongoClient = new MongoClient(url, { useNewUrlParser: true });  mongoClient.connect(function(err, client){      if(err) return console.log(err);      const db = client.db("usersdb");      db.collection("users").deleteOne({name: "Bob"}, function(err, result){          console.log(result);          client.close();      });  }); |

*Метод findOneAndDelete.* Метод findOneAndDelete() удаляет один документ по определенному критерию, но по сравнению с методом deleteOne() он возвращает удаленный документ:

|  |  |
| --- | --- |
|  | const MongoClient = require("mongodb").MongoClient;  const url = "<mongodb://localhost:27017/>";  const mongoClient = new MongoClient(url, { useNewUrlParser: true });  mongoClient.connect(function(err, client){      if(err) return console.log(err);      const db = client.db("usersdb");      db.collection("users").findOneAndDelete({age: 21}, function(err, result){          console.log(result);          client.close();      });  }); |

*Метод drop.* Метод drop() удаляет всю коллекцию:

|  |  |
| --- | --- |
|  | const MongoClient = require("mongodb").MongoClient;  const url = "<mongodb://localhost:27017/>";  const mongoClient = new MongoClient(url, { useNewUrlParser: true });  mongoClient.connect(function(err, client){      if(err) return console.log(err);      const db = client.db("usersdb");      db.collection("users").drop(function(err, result){          console.log(result);          client.close();      });  }); |

**Обновление документов в MongoDB**

Для обновления элементов в MongoDB есть несколько методов:

* updateOne: обновляет один документ, который соответствует критерию фильтрации, и возвращает информацию об операции обновления;
* updateMany: обновляет все документы, которые соответствуют критерию фильтрации, и возвращает информацию об операции обновления;
* findOneAndUpdate: обновляет один документ, который соответствует критерию фильтрации, и возвращает обновленный документ.

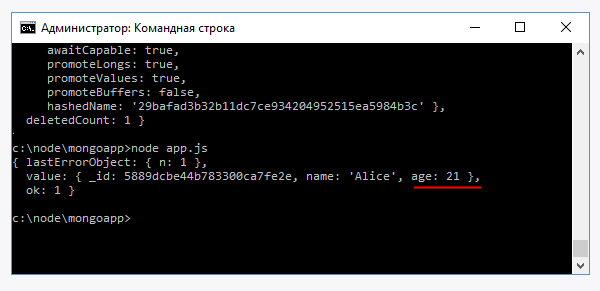
*Метод findOneAndUpdate.* Метод findOneAndUpdate() обновляет один элемент. Он принимает следующие параметры:

1. Критерий фильтрации документа, который надо обновить.
2. Параметр обновления.
3. Дополнительные опции обновления, которые по умолчанию имеют значение null.
4. Функция обратного вызова, которая выполняется при обновлении.

Например, обновим первого пользователя в бд, у которого возраст 21 год:

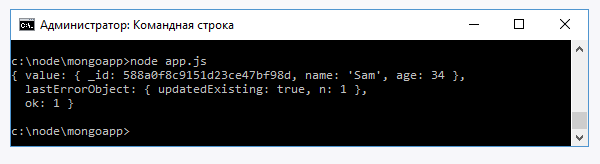
|  |  |
| --- | --- |
|  | const MongoClient = require("mongodb").MongoClient;    const url = "<mongodb://localhost:27017/>";  const mongoClient = new MongoClient(url, { useNewUrlParser: true });    let users = [{name: "Bob", age: 34} , {name: "Alice", age: 21}, {name: "Tom", age: 45}];  mongoClient.connect(function(err, 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();              }          );      });  }); |

Сначала здесь происходит добавление 3 пользователей в базу данных, а после добавления идет обновление. Для обновления применяется объект { $set: {age: 25}}. Параметр $set позволяет обновить значения для одного поля или группы полей. В данном случае изменяется поле age. Третий параметр – функция обратного вызова выводит результат обновления. По умолчанию это старое состояние измененного документа:



Но, допустим, после обновления надо получать не старое, а новое состояние измененного документа. Для этого можно задать дополнительные опции обновления.

|  |  |
| --- | --- |
|  | const MongoClient = require("mongodb").MongoClient;    const url = "<mongodb://localhost:27017/>";  const mongoClient = new MongoClient(url, { useNewUrlParser: 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();          }      );  }); |



*Метод updateMany.* Метод updateMany() позволяет обновить все документы из коллекции, которые соответствуют критерию фильтрации:

|  |  |
| --- | --- |
|  | const MongoClient = require("mongodb").MongoClient;    const url = "<mongodb://localhost:27017/>";  const mongoClient = new MongoClient(url, { useNewUrlParser: 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();          }      );  }); |

*Метод updateOne.* Метод updateOne() аналогичен методу updateMany за тем исключением, что обновляет только один элемент. В отличие от метода findOneAndUpdate() он не возвращает измененный документ:

|  |  |
| --- | --- |
|  | const MongoClient = require("mongodb").MongoClient;    const url = "<mongodb://localhost:27017/>";  const mongoClient = new MongoClient(url, { useNewUrlParser: 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();          }      );  }); |

1. Express и MongoDB.

Теперь объединим в одном приложении обработку запросов с помощью Express и работу с данными в MongoDB. Для этого определим следующий файл приложения 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/>", { useNewUrlParser: 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 каждого типа запросов здесь определен свой обработчик Express. И в каждом из обработчиков каждый раз обращаемся к базе данных. Чтобы не открывать и закрывать подключение каждый раз при каждом запросе, открываем подключение в самом начале и только после открытия подключения запускаем прослушивание входящих запросов:

|  |  |
| --- | --- |
|  | 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...");      });  }); |

Поскольку все взаимодействие будет идти с коллекцией users, то получаем ссылку на эту коллекцию в локальную переменную приложения app.locals.collection. Затем через эту переменную можно получить доступ к коллекции в любом месте приложения.

В конце работы скрипта можно закрыть подключение, сохраненное в переменную dbClient:

|  |  |
| --- | --- |
|  | process.on("SIGINT", () => {      dbClient.close();      process.exit();  }); |

В данном случае прослушиваем событие "SIGINT", которое генерируется при нажатии комбинации CTRL+C в консоли, что завершит выполнение скрипта. Когда приходит GET-запрос к приложению, то возвращаем в ответ клиенту все документы из базы данных:

|  |  |
| --- | --- |
|  | 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)      });    }); |

Если в GET-запросе передается параметр id, то возвращаем только одного пользователя из базы данных по этому 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);      });  }); |

Когда приходит POST-запрос, с помощью парсера jsonParser получаем отправленные данные и по ним создаем объект, который добавляем в базу данных посредством метода insertOne():

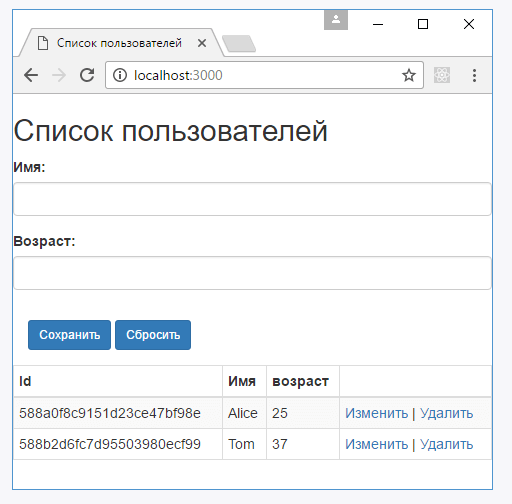
|  |  |
| --- | --- |
|  | app.post("/api/users", jsonParser, function (req, res) {        if(!req.body) return res.sendStatus(400);        const userName = req.body.name;      const userAge = req.body.age;      const user = {name: userName, age: userAge};        const collection = req.app.locals.collection;      collection.insertOne(user, function(err, result){            if(err) return console.log(err);          res.send(user);      });  }); |

При получении PUT-запроса также получаем отправленные данные и с помощью метода findOneAndUpdate() обновляем данные в БД. И в методе app.delete(), который срабатывает при получении запроса DELETE, вызываем метод findOneAndDelete() для удаления данных. Таким образом, в каждом обработчике Express задействуем определенный метод по работе с MongoDB. Теперь создадим в папке проекта новый каталог "public" и определим в этом каталоге 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" />      <script src="<https://code.jquery.com/jquery-2.2.4.min.js>"></script>  </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          function GetUsers() {              $.ajax({                  url: "/api/users",                  type: "GET",                  contentType: "application/json",                  success: function (users) {                      var rows = "";                      $.each(users, function (index, user) {                          // добавляем полученные элементы в таблицу                          rows += row(user);                      })                      $("table tbody").append(rows);                   }              });          }          // Получение одного пользователя          function GetUser(id) {              $.ajax({                  url: "/api/users/"+id,                  type: "GET",                  contentType: "application/json",                  success: function (user) {                      var form = document.forms["userForm"];                      form.elements["id"].value = user.\_id;                      form.elements["name"].value = user.name;                      form.elements["age"].value = user.age;                  }              });          }          // Добавление пользователя          function CreateUser(userName, userAge) {              $.ajax({                  url: "api/users",                  contentType: "application/json",                  method: "POST",                  data: JSON.stringify({                      name: userName,                      age: userAge                  }),                  success: function (user) {                      reset();                      $("table tbody").append(row(user));                  }              })          }          // Изменение пользователя          function EditUser(userId, userName, userAge) {              $.ajax({                  url: "api/users",                  contentType: "application/json",                  method: "PUT",                  data: JSON.stringify({                      id: userId,                      name: userName,                      age: userAge                  }),                  success: function (user) {                      reset();                      console.log(user);                      $("tr[data-rowid='" + user.\_id + "']").replaceWith(row(user));                  }              })          }            // сброс формы          function reset() {              var form = document.forms["userForm"];              form.reset();              form.elements["id"].value = 0;          }            // Удаление пользователя          function DeleteUser(id) {              $.ajax({                  url: "api/users/"+id,                  contentType: "application/json",                  method: "DELETE",                  success: function (user) {                      console.log(user);                      $("tr[data-rowid='" + user.\_id + "']").remove();                  }              })          }          // создание строки для таблицы          var row = function (user) {              return "<tr data-rowid='" + user.\_id + "'><td>" + user.\_id + "</td>" +                     "<td>" + user.name + "</td> <td>" + user.age + "</td>" +                     "<td><a class='editLink' data-id='" + user.\_id + "'>Изменить</a> | " +                      "<a class='removeLink' data-id='" + user.\_id + "'>Удалить</a></td></tr>";          }          // сброс значений формы          $("#reset").click(function (e) {                e.preventDefault();              reset();          })            // отправка формы          $("form").submit(function (e) {              e.preventDefault();              var id = this.elements["id"].value;              var name = this.elements["name"].value;              var age = this.elements["age"].value;              if (id == 0)                  CreateUser(name, age);              else                  EditUser(id, name, age);          });            // нажимаем на ссылку Изменить          $("body").on("click", ".editLink", function () {              var id = $(this).data("id");              GetUser(id);          })          // нажимаем на ссылку Удалить          $("body").on("click", ".removeLink", function () {              var id = $(this).data("id");              DeleteUser(id);          })            // загрузка пользователей          GetUsers();      </script>  </body>  </html> |

Для упрощения отправки запросов в данном случае применяется библиотека jquery. В принципе код index.html вкратце обсуждался в статье про создание [API в Node.js](https://metanit.com/web/nodejs/4.11.php), 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 можно управлять пользователями, которые хранятся в базе данных MongoDB:



1. Mongoose.

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

Для работы с Mongoose продолжим работу с проектом из прошлых тем и вначале установим саму библиотеку с помощью команды:

npm install mongoose --save

Далее определим следующий код в файле приложения 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>", { 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);  }); |

Здесь прежде всего надо подключить mongoose:

|  |  |
| --- | --- |
|  | const mongoose = require("mongoose"); |

Данные, которые используются в Mongoose, описываются определенной схемой. Например, в прошлых темах сохраняли в базу данных объекты с двумя свойствами name и age. Поэтому описываем здесь следующую схему:

|  |  |
| --- | --- |
|  | const Schema = mongoose.Schema;    // установка схемы  const userScheme = new Schema({      name: String,      age: Number  }); |

Схема содержит метаданные объектов. В частности, здесь устанавливаем, какие свойства будет иметь объект и какой у них будет тип данных. То есть это схема, которая описывает объект пользователя.

Затем, используя эту схему, создаем модель пользователя:

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

Первый параметр в методе mongoose.model указывает на название модели. Mongoose затем будет автоматически искать в базе данных коллекцию, название которой соответствует названию модели во множественном числе. Например, в данном случае название модели "User". Во множественном числе в соответствии с правилами английского языка это "users". Поэтому при работе с данными модели User (добавлении, удалении, редактировании и получении объектов) mongoose будет обращаться к коллекции "users". Если такая коллекция есть в бд, то с ней будет идти взаимодействие. Если такой коллекции в базе данных нет, то она будет создана автоматически.

Второй параметр функции mongoose.model – схема.

Далее можно создавать объекты этой модели:

|  |  |
| --- | --- |
|  | const user = new User({      name: "Bill",      age: 41  }); |

Для подключения к базе данных применяется метод mongoose.connect(), в который передается адрес базы данных на сервере mongo:

|  |  |
| --- | --- |
|  | mongoose.connect("<mongodb://localhost:27017/usersdb>"); |

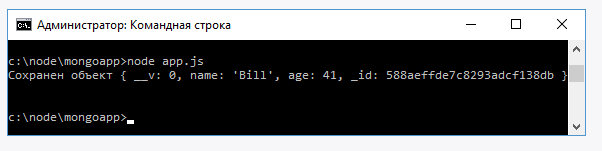
И в конце у объекта вызывается метод save. Этот метод определен для всех создаваемых моделей, он сохраняет текущий объект в базу данных:

|  |  |
| --- | --- |
|  | user.save(function(err){        mongoose.disconnect();  // отключение от базы данных      if(err) return console.log(err);        console.log("Сохранен объект", user);  }); |

С помощью метода mongoose.disconnect(); происходит отключение от бд. Так как метод save возвращает promise, то есть и другая форма сохранения объекта:

|  |  |
| --- | --- |
|  | user.save()  .then(function(doc){      console.log("Сохранен объект", doc);      mongoose.disconnect();  // отключение от базы данных  })  .catch(function (err){      console.log(err);      mongoose.disconnect();  }); |

С помощью метода then можно получить данные, которые возвратил нам сервер и выполнить обработку результата. Запустим приложение и выполним добавление объекта:



1. Определение схемы в Mongoose.

Схема в Mongoose определяет метаданные модели – ее свойства, типы данных и ряд другой информации. В прошлой теме схема определялась следующим образом:

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

Определение типа

В качестве типа данных можно указывать одно из следующих значений: String, Number, Date, Buffer, Boolean, Mixed, Objectid, Array, Decimal128, Map. Если свойство представляет сложный объект, то в качестве типа указываем определение этого объекта:

|  |  |
| --- | --- |
|  | const userScheme = new Schema({      name: String,      age: Number,      company: {          name: String,          employee: [String], // тип - массив строк          date: Date      }  }); |

Подробнее про определение типов можно посмотреть по ссылке <http://mongoosejs.com/docs/schematypes.html>. Но название свойств и их тип – это не все, что можно здесь определить.

**Значение по умолчанию**

С помощью параметра default можно указать значение по умолчанию для свойства. Например:

|  |  |
| --- | --- |
|  | const userScheme = new Schema({      name: {          type: String,          default: "NoName"      },      age: {          type: Number,          default: 22      }  }); |

Создадим трех пользователей:

|  |  |
| --- | --- |
|  | 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 |

Если у пользователя не определено свойство, то для него используется значение по умолчанию.

**Валидация**

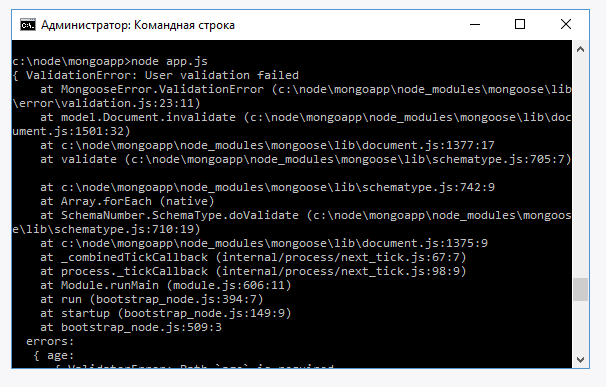
Mongoose имеет ряд встроенных правил валидации, которые можно указать в схеме:

* required: требует обязательного наличия значения для свойства;
* min и max: задают минимальное и максимальное значения для числовых данных;
* minlength и maxlength: задают минимальную и максимальную длину для строк;
* enum: строка должна представлять одно из значений в указанном массиве строк;
* match: строка должна соответствовать регулярному выражению.

Если попытаться добавить некорректные данные в БД, то запрос на добавление вернет ошибку:

|  |  |
| --- | --- |
|  | const mongoose = require("mongoose");  const Schema = mongoose.Schema;    // подключение  mongoose.connect("<mongodb://localhost:27017/usersdb>", { 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);  }); |

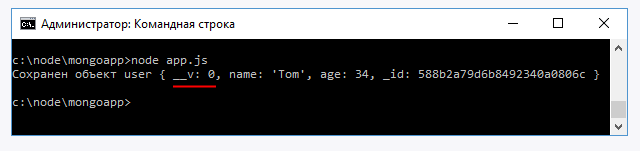
При запуске на консоли увидим подробную информацию об ошибке:



**Версия документа**

По умолчанию при сохранении данных Mongoose добавляет специальное поле \_\_v, которое указывает на версию документа:

|  |  |
| --- | --- |
|  | 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);  }); |



Не всегда подобное поле необходимо, и его можно отключить, добавив в схему объект { versionKey: false }:

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

1. CRUD в Mongoose.

Рассмотрим, как выполнять основные операции с данными в Mongoose.

**Создание документов**

В прошлых вопросах было в общих чертах описано создание и добавление объектов в Mongoose. В частности, у объекта модели можно вызвать 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);  }); |

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

|  |  |
| --- | --- |
|  | const mongoose = require("mongoose");  const Schema = mongoose.Schema;  // подключение  mongoose.connect("<mongodb://localhost:27017/usersdb>", { 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);  }); |

Первый параметр метода – сохраняемый объект.

**Получение данных**

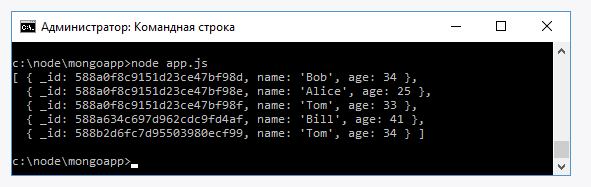
Для получения данных можно использовать целый набор методов:

* 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;   mongoose.connect("<mongodb://localhost:27017/usersdb>", { 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);  }); |

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



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

|  |  |
| --- | --- |
|  | User.find({name: "Tom"}, function(err, docs){      mongoose.disconnect();      if(err) return console.log(err);      console.log(docs);  }); |

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

|  |  |
| --- | --- |
|  | User.findOne({name: "Tom"}, function(err, doc){      mongoose.disconnect();      if(err) return console.log(err);      console.log(doc);  }); |

И метод findById() возвращает документ с определенным идентификатором:

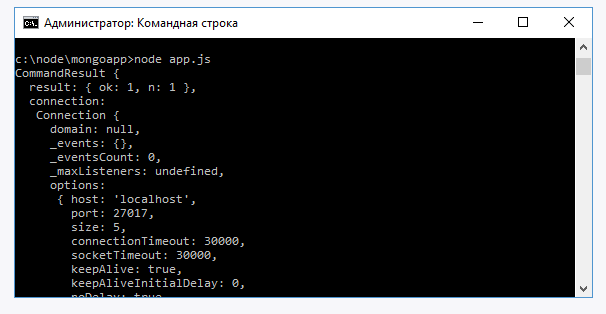
|  |  |
| --- | --- |
|  | const id = "588a0f8c9151d23ce47bf98d";  User.findById(id, function(err, doc){      mongoose.disconnect();      if(err) return console.log(err);      console.log(doc);  }); |

**Удаление данных**

Для удаления применяется метод remove(). В этот метод передается критерий фильтрации документов на удаление. Например, удалим всех пользователей, у которых возраст равен 33:

|  |  |
| --- | --- |
|  | const mongoose = require("mongoose");  const Schema = mongoose.Schema;   mongoose.connect("<mongodb://localhost:27017/usersdb>", { useNewUrlParser: true });  const userScheme = new Schema({name: String, age: Number}, {versionKey: false});  const User = mongoose.model("User", userScheme);   User.remove({age:33}, function(err, result){      mongoose.disconnect();      if(err) return console.log(err);      console.log(result);  }); |

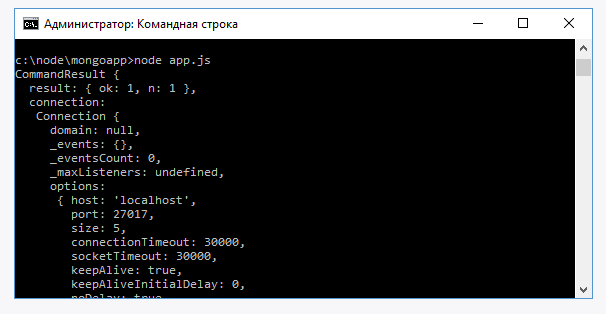
Объект result, который передается в функцию обратного вызова, содержит информацию об операции удаления:



Если же надо удалить один документ, то можно использовать метод findOneAndDelete():

|  |  |
| --- | --- |
|  | User.findOneAndDelete({name:"Bill"}, function(err, doc){      mongoose.disconnect();      if(err) return console.log(err);       console.log("Удален пользователь ", doc);  }); |

В отличие от метода remove здесь в функцию обратного вызова передается удаленный документ:



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

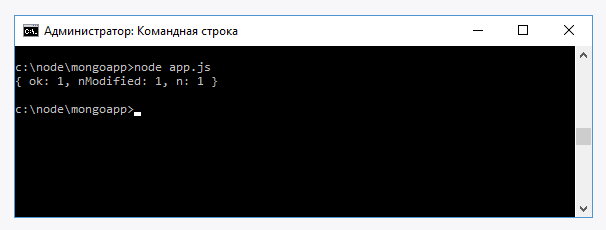
|  |  |
| --- | --- |
|  | var id = "588a0f8c9151d23ce47bf98d";  User.findByIdAndDelete(id, function(err, doc){      mongoose.disconnect();       if(err) return console.log(err);      console.log("Удален пользователь ", doc);  }); |

**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;   mongoose.connect("<mongodb://localhost:27017/usersdb>", { 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);  }); |

Первый параметр метода – критерий фильтрации. В данном случае находим всех пользователей, у которых имя "Tom". А второй параметр описывает, что и как надо изменить. То есть здесь меняем имя на "Tom Smith". В функцию обратного вызова передается результат операции:

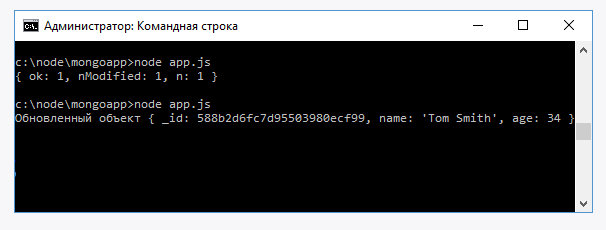


Аналогично работает метод updateMany.

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

|  |  |
| --- | --- |
|  | const id = "588b2d6fc7d95503980ecf99";  User.findByIdAndUpdate(id, {name: "Sam", age: 25}, function(err, user){      mongoose.disconnect();      if(err) return console.log(err);      console.log("Обновленный объект", user);  }); |

Первый параметр метода – значения для поля \_id у обновляемого документа, а второй – набор новых значений для полей объекта. В функцию обратного вызова передается обновленный документ:



Но по умолчанию передается старое состояние документа. Если же надо получить документ уже в измененном состоянии, то в метод findByIdAndUpdate необходимо передать еще один параметр:

|  |  |
| --- | --- |
|  | 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);  }); |

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

|  |  |
| --- | --- |
|  | 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);  }); |

Первый параметр представляет критерий выборки. Второй параметр представляет обновленные значения документа. Третий параметр указывает, что возвращаем вариант документа именно после обновления – {new: true} – при значении false возвращается старая копия, которая была до обновления. И четвертый параметр – функция обработчика, вызываемая при получении результата.

1. 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();  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>", { useNewUrlParser: true }, 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 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:

|  |  |
| --- | --- |
|  | mongoose.connect("<mongodb://localhost:27017/usersdb>", { useNewUrlParser: true }, function(err){      if(err) return console.log(err);      app.listen(3000, function(){          console.log("Сервер ожидает подключения...");      });  }); |

Теперь создадим в папке проекта новый каталог "public" и определим в этом каталоге 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" />      <script src="<https://code.jquery.com/jquery-2.2.4.min.js>"></script>  </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          function GetUsers() {              $.ajax({                  url: "/api/users",                  type: "GET",                  contentType: "application/json",                  success: function (users) {                      var rows = "";                      $.each(users, function (index, user) {                          // добавляем полученные элементы в таблицу                          rows += row(user);                      })                      $("table tbody").append(rows);                   }              });          }          // Получение одного пользователя          function GetUser(id) {              $.ajax({                  url: "/api/users/"+id,                  type: "GET",                  contentType: "application/json",                  success: function (user) {                      var form = document.forms["userForm"];                      form.elements["id"].value = user.\_id;                      form.elements["name"].value = user.name;                      form.elements["age"].value = user.age;                  }              });          }          // Добавление пользователя          function CreateUser(userName, userAge) {              $.ajax({                  url: "api/users",                  contentType: "application/json",                  method: "POST",                  data: JSON.stringify({                      name: userName,                      age: userAge                  }),                  success: function (user) {                      reset();                      $("table tbody").append(row(user));                  }              })          }          // Изменение пользователя          function EditUser(userId, userName, userAge) {              $.ajax({                  url: "api/users",                  contentType: "application/json",                  method: "PUT",                  data: JSON.stringify({                      id: userId,                      name: userName,                      age: userAge                  }),                  success: function (user) {                      reset();                      console.log(user);                      $("tr[data-rowid='" + user.\_id + "']").replaceWith(row(user));                  }              })          }            // сброс формы          function reset() {              var form = document.forms["userForm"];              form.reset();              form.elements["id"].value = 0;          }            // Удаление пользователя          function DeleteUser(id) {              $.ajax({                  url: "api/users/"+id,                  contentType: "application/json",                  method: "DELETE",                  success: function (user) {                      console.log(user);                      $("tr[data-rowid='" + user.\_id + "']").remove();                  }              })          }          // создание строки для таблицы          var row = function (user) {              return "<tr data-rowid='" + user.\_id + "'><td>" + user.\_id + "</td>" +                     "<td>" + user.name + "</td> <td>" + user.age + "</td>" +                     "<td><a class='editLink' data-id='" + user.\_id + "'>Изменить</a> | " +                      "<a class='removeLink' data-id='" + user.\_id + "'>Удалить</a></td></tr>";          }          // сброс значений формы          $("#reset").click(function (e) {                e.preventDefault();              reset();          })            // отправка формы          $("form").submit(function (e) {              e.preventDefault();              var id = this.elements["id"].value;              var name = this.elements["name"].value;              var age = this.elements["age"].value;              if (id == 0)                  CreateUser(name, age);              else                  EditUser(id, name, age);          });            // нажимаем на ссылку Изменить          $("body").on("click", ".editLink", function () {              var id = $(this).data("id");              GetUser(id);          })          // нажимаем на ссылку Удалить          $("body").on("click", ".removeLink", function () {              var id = $(this).data("id");              DeleteUser(id);          })            // загрузка пользователей          GetUsers();      </script>  </body>  </html> |

Для упрощения отправки запросов в данном случае применяется библиотека jquery. 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 можно взаимодействовать с базой данных MongoDB через Mongoose:

