# MongoDB

Mongo - это noSQL БД. NoSQL BD делятся на несколько типов:

* БД с типом записей в виде документов - такие, как MongoDB
* БД с типом записей в виде ключ-значения, такие как Redis. Фишка - скорость работы
* БД с типом записей в виде колонок, такие как Cassandra
* БД с типом записей в виде графов, такие как Neo4j

В первом типе БД каждый кусочек информации - это **документ**. Записи могут в разных форматах, самый частоиспользуемый - JSON.

Документы могут быть объединены в **коллекции** документов.

Наобор коллекций документов и есть база данных.

## Монго сохраняет данные на диске в BSON (Binary JSON):

* хранит префикс с длинной каждого значения - позволяет быстро пропускать ненужные поля, т.к. известна длинна конкретно нужного поля
* хранит иформацию о типе поля - строка, число и др
* хранит objectID

ObjectId - каждому документу присваивается свой ID. Если ID не указан специально, Монго даст ему свой. Обьект в Монго выглядит следующим образом:

{

"\_id": ObjectId("56ce74c0b0280ad12nc323xsad23),

"name": "Ultrapizza",

"description": "Test"

}

ObjectId занимает 12 байт и состоит из:

* timestamp - 4 байта
* Machine Id - 3 байта
* Process Id - 2 байта
* Increment - 3 байта

id.getTimestamp() возвратит timestamp в ISO Date формате

# Отличия NoSQL от SQL

В SQL - таблица (table), в NoSQL - коллекция (collection)

В SQL - запись (row), в NoSQL - документ (document)

В SQL - колонка (column), и у каждой записи в таблице будут одинаковые колонки

В NoSQL - поле (field), у каждой записи в документе поля могут отличаться полностью

# Установка

Скачиваем с оф. сайта Монго, ставим следуя базовым инструкциям. В папке проекта или рядом с ним создаем папку mongo в ней создаем папку data. Заходим в нее, копируем путь.

Для windows: запускаем cmd от имени императора, переходим в папку с Монго Сервером:

cd C:\Program Files\MongoDB\Server\3.2\bin

Дальше нам нужно указать для монго, что надо бы создать БД в папке с проектом, создаем там файл mongo, в нем файл data, пишем:

mongod.exe --dbpath="D:\Git\my\_notes\server\_side\node-examples\mongo\data"

После этого Монго сразу распакует в папку data свои файлы. Если че-то не получилось, смотреть тут <https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/>

# Использование

НЕ ЗАКРЫВАЯ предыдущее окно, открываем новое, там пишем:

cd C:\Program Files\MongoDB\Server\3.2\bin

mongo.exe

Это откроет консоль управления базой данных

Команды

show dbs - показывает все БД

use conFusion - создает и переходит в новую БД conFusion. РЕГИСТРОЗАВИСИМ!!

db - показывает текущую БД

show collections или show tables - выведет имена коллекций

db.help() - выведет методы

Создать документ dishes и вставить в него данные:

db.dishes.insert({name: "Ultrapizza", description: "testing pizza"});

db.dishes.find(); - выводит инфу по документу dishes

db.dishes.find().pretty(); - выводит инфу в табличном виде

Создать новую переменную, посмотреть на нее, получить время

var id = new ObjectId();

id

id.getTimestamp();

Изменение (UPDATE) существующей записи

db.dishes.update({name: "Ultrapizza"}, {$set:{description: "changed description"}});

первое значение - уникальное имя записи, которую будем менять

второе - что меняем, в обьекте

## id

То, что отличает NoSQL от SQL БД.

Составной id позволяет с легкостью масштабировать приложения при увеличинии нагрузки, т.к. нет необходимости вычислять, какой id меньше или больше другого

id составляется из парметров:

* первые 4 байта - это timestamp, потому нет необходимости специально указывать время добавления записи
* следующие три байта - идентификатор машины, на которой запущена ДБ
* следующие 2 байта - идентификатор процесса
* последние 3 байта - и есть идентификатор записи, то, что делает и SQL

result.ops[0].\_id.getTimestamp() - покажет дату добавления в формате Timestamp

Создать новую переменную, посмотреть на нее, получить время

var id = new ObjectId();

console.log(id);

id.getTimestamp();

# Установка robomongo - GUI для mongo

Заходим на <https://robomongo.org/download>

Качаем, устанавливаем по дефолту, открываем

Открывается окно, жмем там строчку **Create**

Name **-** назовем как-нибудь, например Local Mongo Database

Остальные поля не трогаем, если монго установлен локально

Двойной клик откроет древовидное дерево папок

# Mongo DB native - модуль для Node.js

У Монго есть свой драйвер для Ноды. Создаем папку, назовем ее например node-mongodb - это и будет папка проекта. Заходим в нее, создаем папку node\_modules, копируем путь, переходим по нему в cmd и пишем:

npm install mongodb --save

**Пример 1.**

Создаем файл mongo-connect.js, в него пишем

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

// 1st arg - url where DB lives, 2nd - callback

MongoClient.connect('mongodb://localhost:27017/TodoApp', (err, db) => {

if(err){

return console.log('Unable to connect to MongoDB server');

}

console.log('Connected to MongoDB server');

// insert new document into collection

db.collection('Todos').insertOne({

text: 'Something info',

complete: false

}, (err, result) => {

if (err) {

return console.log('Unable to insert Todo', err)

}

// show all docs

console.log(JSON.stringify(result.ops, null, 2));

// show timestamp

console.log(result.ops[0].\_id.getTimestamp());

});

db.close();

});

Это создаст базу данных TodoApp по указанному адресу. База создается автоматически. **Важно!** если в БД не добавлено никаких данных, БД не будет видна (не создастся).

Тут мы сразу добавляем коллекцию Todo, в которую добавляем документ с двумя свойствами

Сразу после добавления через insertOne в консоли отобразиться добавленный при помощи **result.ops**. JSON.stringify нужен для красивого и понятного отображения с отступами.

**Важно!** result.ops покажет добавленные только что документы, а не всю таблицу целиком!

# КУРС ПО МОГО ОТ UDEMY

## ПОДКЛЮЧЕНИЕ MONGOOSE

При подключении монго всегда нужно указывать порт. users\_test - название БД. Если ее нет, монго ее создаст

const mongoose = require('mongoose');

// mongoose.connect('mongodb://65.34.5.12:4000/users\_test');

mongoose.connect('mongodb://localhost/users\_test');

mongoose.connection

.once('open', () => console.log('Mongoose connected'))

.on('error', (error) => console.warn('Warning', error));

## СОЗДАНИЕ НОВОЙ МОДЕЛИ И СХЕМЫ

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const UserSchema = new Schema({

name: String,

});

const User = mongoose.model('user', UserSchema);

module.exports = User;

## ТЕСТЫ

Сделаем отдельный конфигурацийонный файл **helper.test.js**, где будут указаны настройки подключения и обнуление базы перед основными тестами

const mongoose = require('mongoose');

// mongoose.connect('mongodb://65.34.5.12:4000/users\_test');

mongoose.connect('mongodb://localhost/users\_test');

mongoose.connection

.once('open', () => console.log('Mongoose connected'))

.on('error', (error) => console.warn('Warning', error));

beforeEach((done) => {

mongoose.connection.collections.users.drop();

done();

});

После этого сделаем базовый тестовый файл с функцией создания коллекции **create.test.js**

**const assert = require('assert');**

**const User = require('../src/user');**

**describe('creating records', () => {**

**it('saves a user', (done) => {**

**const joe = new User({name: 'Joe'});**

**joe.save()**

**.then(() => {**

**// isNew === true - document has not been saved**

**assert(!joe.isNew);**

**done();**

**})**

**})**

**});**

## ТЕСТЫ С УКАЗАНИЕМ МАКСИМАЛЬНОГО ВРЕМЯ ОЖИДАНИЯ И ВЫНОСОМ В ФУНКЦИЮ

const assert = require('assert');

const User = require('../src/user');

describe('Updating a user', () => {

let joe;

beforeEach((done) => {

joe = new User({name: 'Joe'});

joe.save().then(() => done());

});

function assertName(operation, name, done) {

operation

.then(() => User.findOne({name}))

.then((user) => {

assert(user.name === name);

done();

})

.catch((e) => {

})

}

it('Instance type using set and save', (done) => {

// cause .set() hasn't promise chaining

// best using when we called different func to change some document,

// and then save all together to prevent call DB every time to make a little change

joe.set('name', 'Alex');

// .set() saves in memory, so we need to save into DB after .set called

assertName(joe.save(), 'Alex', done);

}).timeout('10s'); // set max timeout for every it. Doesn't work on describe

it('A model instance can update', (done) => {

assertName(joe.update({name: 'Alex'}), 'Alex', done);

}).timeout('10s');

});

**UPDATE OPERATORS**

Если нужно произвести какое-то изменение внутри множества документов коллекции Монго, что бы не дергать каждый рез ДБ через цикл, есть операторы, которые заточены специально под Монго  
<https://docs.mongodb.com/manual/reference/operator/update/>

it('A user can have their postCount incremented by 1', (done) => {

// increment for every user with name Joe

User.update({name: joe.name}, {$inc: {postCount: 1}})

.then(() => User.findOne({name: joe.name}))

.then((user) => {

assert(user.postCount === joe.postCount + 1);

done();

})

}).timeout('10s');

## ВАЛИДАЦИЯ В МОНГО

В схему можно добавить проверку обязательности наличия свойства (например имени), а так же проверку длины и прочее.

required - первый аргумент boolean, второй - сообщение об ошибке

validate - обьект, который принимает свойства:

* validator - функция, которая может проверять что угодно
* message - сообщение, которое выводится если проверка провалена

const UserSchema = new Schema({

name: {

type: String,

required: [true, 'Name is required'],

validate: {

validator: (name) => name.length > 2,

message: 'Name must must be longer than 2 characters'

}

},

postCount: Number

});

Использование

.validateSync() используется для локальной проверки, когда результат можно присвоить в переменную

.validate() - асинхронная функция, используется когда нужно сделать обращение к БД или к удаленному серверу, в ней вызывается callback после того, как валидация выполнена

it('Requires a user name', () => {

const user = new User({name: undefined});

const validationResult = user.validateSync();

const {message} = validationResult.errors.name;

assert(message === 'Name is required');

});

// асинхронный пример

it('Requires a user name', () => {

const user = new User({name: undefined});

user.validate((validationResult) => {

})

})

## RELATIONS - ВЛОЖЕННЫЕ ОТНОШЕНИЯ В MONGO

Для простого примера отношений создается вложенная схема. Например есть юзеры, которые пишут посты. Для этого в модели юзеров создается схема для постов:

Сделаем файл **post.js:**

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const PostSchema = new Schema({

title: String,

});

module.exports = PostSchema;

Теперь включим его в модель **user.js.** Монго будет понимать, что это должен быть массив с вложенными объектами**:**

const mongoose = require('mongoose');

const PostSchema = require('./post');

const Schema = mongoose.Schema;

const UserSchema = new Schema({

name: String,

postCount: Number,

posts: [PostSchema]

});

const User = mongoose.model('user', UserSchema);

module.exports = User;

## ВИРТУАЛЬНЫЕ ТИПЫ

Этим странным словом именуется то, что не относится к БД Монго, а уже подсчитывается на сервере, сразу после извлечения документа из БД. Например подсчет количества постов для каждого юзера - на сервере будет подсчитываться длинна массива posts.

Используется метод ES6 - **getter**, который возвращает значение вычисляемой функции, которая этому геттеру передается.

**this** возвращает тут ссылку на конкретный экземпляр модели User - в док. mongooose

<http://mongoosejs.com/docs/api.html#Virtualtype>

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const UserSchema = new Schema({

name: String

});

// setup postCount as virtual type - it will be calculate after user will fetch

// user ES5 getter - its important to use "function" keyword, not "() =>"

// because in this case keyword "this" will belong to whole file,

// not user model instance

UserSchema.virtual('postCount').get(function (){

return this.posts.length;

});

const User = mongoose.model('user', UserSchema);

module.exports = User;

Проверка:

const joe = new User({

name: 'Joe',

posts: [{title: 'PostTitle'}]

});

console.log(joe.postCount) // 1

## POPULATE RELATIONS IN MONGO

Когда БД становится сложной, она начинает быть похожей на обычную реляционную БД, с отельными коллекциями.

Например у есть юзеры, которые пишут посты. Присобачим посты в **user.js** через ссылку:

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const UserSchema = new Schema({

name: String,

blogPosts: [{

type: Schema.Types.ObjectId,

ref: 'blogPost'

}]

});

const User = mongoose.model('user', UserSchema);

module.exports = User;

type - указываем, что нужно хранить только ID поста, как ссылку

ref - нужно указать модель, точно ту, которую мы укажем в модели **blogPost.js:**

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const BlogPostSchema = new Schema({

title: String,

content: String

});

const BlogPost = mongoose.model('blogPost', BlogPostSchema);

module.exports = BlogPost;

Извлечь документы из привязанных коллекций можно через **populate.**

User.findOne({name: 'Joe'})

.populate('blogPosts')

.then((user) => {

});

Что бы взять большую вложенность, необходимо populate вкладывать друг в друга. При этом производительность системы сильно падает

Сделаем еще одну модель - **comment.js**, которые можно оставлять под каждым blogPost, комменты могут осталвлять зареганные юзеры из user.js

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const CommentSchema = new Schema({

content: String,

user: {

type: Schema.Types.ObjectId,

ref: 'user'

}

});

const Comment = mongoose.model('comment', CommentSchema);

module.exports = Comment;

Теперь, что бы извлечь все дерево (граф) вложенности, будем использовать рекурсивное **populate.** В конце каждого вложенного populate нужно обязательно указывать модель, по которой будут извлекаться данные:

User.findOne({name: 'Joe'})

.populate({

path: 'blogPosts',

populate: {

path: 'comments',

model: 'comment',

populate: {

path: 'user',

model: 'user'

}

}

})

.then((user) => {

assert(user.name === joe.name);

assert(user.blogPosts[0].title === blogPost.title);

assert(user.blogPosts[0].comments[0].content === comment.content);

assert(user.blogPosts[0].comments[0].user.name === joe.name);

done();

});

## Mongo middleware

Есть pre-event и post-event middleware (вызываются до и после эвента, например до и после save());

Например понадобилось удалить юзера, и все посты, которые он написал.

Можно сделать это в модели **user.js**, написав

const BlogPost = require('./blogpost')

Однако это не вариант, т.к. если понадобиться сделать аналогичную операцию в **blogPost.js** с моделью User, придется тоже писать

const User = require('./user)

Что приведет к зацикливания загрузок моделей - blogPost ссылается на user и наоборот. Во избежание подобных ситуаций можно подгрузить модель средствами mongoose. В user.js пишем:

**UserSchema.pre('remove', function (next){**

**const BlogPost = mongoose.model('blogPost');**

**BlogPost.remove({ \_id: {$in: this.blogPosts} })**

**.then(() => next());**

**});**

**this.blogPosts** содержит массиов с ID прикрепленных постов (можно увидеть на UserSchema выше)

**$in** перебирает всю коллекцию и проверяет, находится ли ID из операции выше в итерируемом объекте. Если да - производит операцию (в данном случае - удаление)

next - пробрасывает операцию дальше. Указывать обязательно

## QUERY HELPERS

Например есть список юзеров, у которых есть имя. Мы хотим пропустить первого, и взять 2го и 3го

Для этого извлечем всю коллекцию, отсортируем по имени, а потом пропустим 1 и возьмем следующих 2:

User.find({})

.sort({name: 1}) // -1 для сортировки - большие значения впереди (DESC)

.skip(1) // пропустить 1

.limit(2) // взять после пропуска только 2 последующих

.then((users) => {

assert(users.length === 2);

});