**Введение. Зачем нужны базы данных?**

Всё это время мы программировали сервер: создавали приложение, писали роутеры, мидлвэры, разбирались с обработкой запросов. Посмотрим, зачем это было нужно в целом. Всё сводилось к одному: клиент присылает запрос и получает ответ. Ответ — что-то ценное для пользователя. И это представляет ценность, потому что содержит в себе информацию. Иными словами мы описывали алгоритмы для работы с данными.

Чтобы сервер мог отдавать клиенту данные, их нужно где-то хранить. Пока мы пользовались одним из двух вариантов:

1. Записывали данные в переменную. Пока сервер работает, всё хорошо, но стоит перезагрузить его, данные окажутся стёрты. Этот вариант не годится.
2. Хранили данные в файле. Это тоже так себе способ. Создал разработчик файл users.json и записал в него данные всех пользователей:

Скопировать кодJAVASCRIPT

[

{

"name": "Ada Lovelace",

"about": "Mathematician, writer",

"\_id": "dbfe53c3c4d568240378b0c6"

},

{

"name": "Tim Berners-Lee",

"about": "Inventor, scientist",

"\_id": "d285e3dceed844f902650f40"

},

{

"name": "Alan Kay",

"about": "Computer scientist",

"\_id": "7d8c010a1c97ca2654997a95"

},

{

"name": "Alan Turing",

"about": "Mathematician, cryptanalyst",

"\_id": "f20c9c560aa652a72cba323f"

},

{

"name": "Bret Victor",

"about": "Designer, engineer",

"\_id": "8340d0ec33270a25f2413b69"

}

]

Клиент хочет получить доступ к своей учётной записи. Для этого нужно:

* 1. Прочитать файл.
  2. Пройтись по массиву объектов.
  3. При каждой итерации сравнивать свойство \_id с запрошенным.

Чем больше пользователей, тем больше нагрузка на серверный процессор, поэтому всё будет работать медленно. А для более сложных запросов описывать логику обращения к данным сложно и неудобно.

К тому же, мы хотим выбрать пользователя по идентификатору. Для этого нужно прочитать файл, пройтись по массиву объектов и, сравнивая \_id с искомым, найти пользователя.

Базы данных решают все эти проблемы. Это удобные структуры организации информации, которые позволяют быстро и безопасно работать с данными.

В следующем уроке рассмотрим виды баз данных.

# Какие бывают базы. SQL и NoSQL

При работе с данными важны скорость и надёжность. Этого не достичь без баз данных. Но прежде, чем изучать прикладные инструменты, нужно понять общую логику организации данных. Существуют два подхода: реляционные и нереляционные базы данных. В этом уроке расскажем, что представляет собой каждый подход и какой лучше выбрать.

## Реляционные БД

Такие базы данных состоят из таблиц и связей между ними.

К Стасу Басову приходит бизнесмен Виталий Сергеевич и просит сделать интернет-магазин. Фронтенд Стас напишет быстро, но нужно разобраться с организацией данных. Виталий Сергеевич просит: «Нужно, чтобы пользователи могли регистрироваться, а потом смотреть историю заказов».

Уже в этом требовании есть 3 сущности: товары, пользователи и заказы. Поможем Стасу организовать их в реляционную БД.

При реляционном подходе данные собирают в таблице. Каждая сущность — строка в таблице. Нужно заранее определиться со структурой данных, ведь столбцы для каждой сущности будут общими.

Сложим в таблицу товары:

И пользователей:

Отлично. Теперь Стасу нужна ещё одна таблица — таблица заказов. Каждый заказ относится к какому-то пользователю и содержит один или несколько товаров. Нам не нужно хранить здесь информацию о каждом товаре и пользователе. Достаточно указать идентификатор пользователя и идентификатор товара:

Чтобы составить список товаров и отрисовать его на стороне клиента, нужно сделать запрос к базе данных, который возьмёт имя каждого товара из таблицы «Товары» и сгруппирует данные по таблице «Заказы», взяв оттуда только товары нужного пользователя.

В этом и состоит суть реляционных баз данных. Информация организована в таблицы с предопределённой структурой. Мы можем добавить столбец, но тогда каждый элемент таблицы получит ещё один параметр. Мы можем получить информацию, которая нигде не содержится — список товаров с названиями. Для этого используют связи между таблицами.

Стас сдал проект, и Виталий Сергеевич доволен. Когда пользователь заходит на страницу «История заказов», он делает запрос к базе. Запрос на ходу собирает данные из разных таблиц и предоставляет их пользователю.

У такого подхода есть плюсы и минусы. Данные разделены по разным таблицам. Каждая таблица содержит информацию только о тех сущностях, за которые отвечает. В таблице «Товары» есть всё о товарах и никакой информации о заказах или пользователях. Аналогично с таблицей «Пользователи».

Тем не менее таблиц может быть десятки и сотни. Чем больше таблиц нужно обойти запросу, тем больше производительности нужно серверу. Мы можем поставить более мощный процессор и добавить оперативной памяти, но продолжать это до бесконечности не выйдет — в какой-то момент мы упрёмся в потолок производительности. Эту проблему решают нереляционные базы данных.

## Нереляционные БД

Вместо таблиц в этих БД используют коллекции — нежёсткие структуры данных, которые состоят из документов. Объясним на примере.

К Стасу снова приходит Виталий Сергеевич и просит добавить функциональность: если пользователь совершил покупку, ему дают скидку на следующий товар.

При реляционном подходе Стас написал бы код так, что при оформлении заказа к базе отправлялся бы запрос: «Проверь, есть ли в таблице „Заказы“ вот такой идентификатор». Исходя из этого высчитывалась бы скидка.

В нереляционной БД другой подход. При заказе достаточно давать пользователю параметр, который бы говорил, что заказ уже сделан. Тогда не нужно проводить дополнительный поиск по базе заказов:

Скопировать кодJAVASCRIPT

{

"id": "1234252",

"name": "Внук Виталия Сергеевича",

"madeOrder": true

},

{

"id": "1208772",

"name": "Стас Басов",

}

Это и есть документ, который хранит пользователей. Мы не привязаны к жёсткой структуре: у одного пользователя есть свойство "madeOrder", а другого — нет.

Тут Стас подумал и решил: «А переделаю всё в нереляционную БД!» Сначала он переписал таблицу с товарами в документ:

Скопировать кодJAVASCRIPT

*// Документ с товарами*

{

"id": "19960"

"name": "Волшебная палочка Гарри Поттера",

"price": 10,

"amount": 15

},

{

"id": "79789",

"name": "Автомобиль Delorean DMC-12",

"price": 500,

"amount": 7

},

{

"id": "51339",

"name": "Световой меч Люка Скайуокера",

"price": 15,

"amount": 3

},

{

"id": "16307",

"name": "Посох Гэндальфа",

"price": 20,

"amount": 2

}

Затем то же сделал с таблицей пользователей:

Скопировать кодJAVASCRIPT

*// Документ с пользователями*

{

"id": "85334",

"name": "Вито Корлеоне",

"age": 63

},

{

"id": "48222",

"name": "Николас Кейдж в роли Джона Траволты",

"age": 55

},

{

"id": "87130",

"name": "Джон Траволта в роли Николаса Кейджа",

"age": 65

},

{

"id": "17185",

"name": "Лягушонок Пепе",

"age": 14

}

Стасу осталось оформить таблицу с заказами. Тут и выявляется основное отличие реляционных баз данных от нереляционных. В последних мы не будем делать запрос, который пойдёт в связанные таблицы, а просто соберём все нужные данные в одном месте:

Скопировать кодJAVASCRIPT

*// Документ с заказами*

{

"id": "39439",

"user": {

"id": "48222",

"name": "Николас Кейдж в роли Джона Траволты",

"age": 55

},

"items": [

{

"id": "19960",

"name": "Волшебная палочка Гарри Поттера",

"price": 10,

"amountOrdered": 2

},

{

"id": "79789",

"name": "Автомобиль Delorean DMC-12",

"price": 500,

"amountOrdered": 1

},

{

"id": "51339",

"name": "Световой меч Люка Скайуокера",

"price": 15,

"amountOrdered": 1

}

]

},

{

"id": "48241",

"user": {

"id": "17185",

"name": "Лягушонок Пепе",

"age": 14

},

"items": [

{

"id": "16307",

"name": "Посох Гэндальфа",

"price": 20,

"amountOrdered": 1

},

{

"id": "51339",

"name": "Световой меч Люка Скайуокера",

"price": 15,

"amountOrdered": 1

}

]

},

Теперь не нужно собирать данные о заказах, можно просто взять их из документа с товарами. Плюс такого метода в скорости и простоте обработки запроса. Минус — в необходимости дублировать одни и те же данные. Если пользователь изменит имя, придётся менять его и в документе с заказами.

## Какую базу выбрать?

Общего ответа на этот вопрос нет. Всё зависит от задачи, которую вы решаете.

Реляционные БД сильно нагружают процессор и оперативную память сервера. Такие базы масштабируются вертикально, то есть заменой сервера на более мощный.

Нереляционные БД можно масштабировать горизонтально, то есть добавлять новые серверы. На одном сервере разместить пользователей, на другом — заказы, на третьем — товары. Это делает базу данных практически неограниченно масштабируемой.

Мы выбрали нереляционный подход. Почему — расскажем в следующем уроке.

# MongoDB

В прошлом уроке мы говорили, что выбор подхода к организации данных зависит от задачи. Мы хорошенько подумали и выбрали NoSQL.

### Почему NoSQL?

Раньше аббревиатура NoSQL означала Not SQL (англ. «не SQL»). Позже подход вбирал некоторые свойства реляционного подхода, и теперь NoSQL расшифровывается как Not Only SQL (англ. «не только SQL»). Современные NoSQL базы данных комбинируют все плюсы нереляционных и некоторые сильные стороны реляционных баз данных.

Для проекта Mesto важны особенности нереляционного подхода. Если Mesto станет конкурировать по популярности с Instagram, нам будет проще масшабировать свой продукт и выдерживать конкуренцию.

Кроме того, NoSQL базы хранят данные в структурах, близких JSON. Их просто обрабатывать из JavaScript, а значит, нужно меньше заморачиваться с преобразованием данных.

### Почему MongoDB?

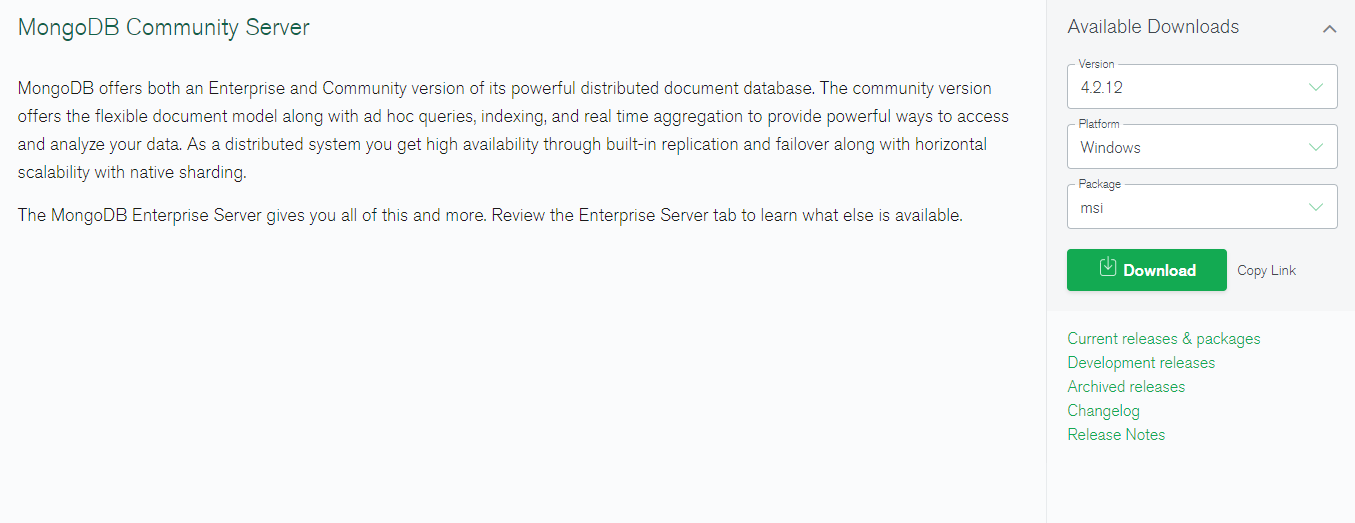
Когда подход выбран, нужно определиться с базой данных. Вариантов много: RavenDB Cassandra, MongoDB, Redis, BigTable и другие. Мы остановили свой выбор на MongoDB. Это самая распространённая NoSQL база данных среди прочих, да ещё и самая популярная среди БД, используемых с Node.js. Если в вакансии нужна Node.js, скорее всего нужна и MongoDB.

## Установка Mongo

Установите Mongo по инструкции для вашей ОС:

### Windows

**Скачайте установщик**



1. Перейдите по [ссылке](https://www.mongodb.com/download-center/community?jmp=docs)
2. в выпадающем списке Version выберите версию с текстом 4.4.5;
3. в выпадающем списке OS выберите Windows x64 X64;
4. в выпадающем списке Package выберите MSI;
5. нажмите Download.

**Запустите установщик**

|  |  |
| --- | --- |
| Запустите установщик | image |
| Нажмите Next | image |
| Поставьте галочку, что вы согласны с лицензионным соглашением и нажмите Next | image |
| Выберите Complete | image |
| Нажмите Next | image |
| Оставьте галочку около Install MongoDB Compass и нажмите Next | image |
| Нажмите Install | image |
| Дождитесь окончания установки | image |
| И нажмите Finish | image |

**Создайте директорию для базы данных**

Откройте Git Bash и перейдите в корень диска C:\

Скопировать кодBASH

cd /C

Создайте директорию для базы данных

Скопировать кодBASH

mkdir -p data/db

На этом установка завершена.

**Добавьте путь к папке bin в переменную окружения**

1. **Найдите путь к папке bin — в ней хранятся исполняемые файлы MongoDB.** Обычно он выглядит как-то так:

Скопировать кодJAVASCRIPT

C:\Program Files\MongoDB\Server\4.2\bin

1. **Откройте настройки переменных окружения.** Нажмите клавишу win и начните вводить «перемен» ("env" для интерфейса на английском). В результатах поиска выберите «Изменение переменных среды текущего пользователя» или «Edit the System Environment Variables».
2. В верхнем окне выберите переменную Path и нажмите «Изменить...».
3. В открывшемся окне нажмите «Создать», скопируйте путь к папке bin, сохраните результат и нажмите «Ок» во всех открытых окнах. Переменная окружения добавлена.

Или воспользуйтесь [официальной инструкцией](https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/#install-mdb-edition)

### macOS

Mongo на macOS устанавливается из Homebrew — популярного пакетного менеджера macOS. Его нужно установить перед установкой Mongo. Для этого запустите терминал и запускайте там все команды из инструкции.

Если вы уже устанавливали Homebrew, сразу переходите к пункту 4.

#### **Пошаговая инструкция**

1. Установите утилиты разработчика от Apple. Для этого в терминале запустите:

Скопировать кодBASH

xcode-select --install

1. Чтобы установить утилиты разработчика, терминал скачает большой файл. Поэтому установка займёт какое-то время — можете налить себе чаю. Хорошая новость в том, что это придётся сделать один раз.

Во время установки у вас попросят подтвердить лицензию на установку. После того как команда выполнится, возможно, понадобится ещё раз подтвердить ещё одну лицензию на использование компонентов среды разработки XCode. Для этого запустите команду:

Скопировать кодBASH

sudo xcodebuild -license

1. Теперь ваш компьютер готов к установке менеджера пакетов brew. Скопируйте эту команду в терминал и запустите:

Скопировать кодBASH

/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

1. Пришло время установить Mongo. По очереди запустите эти команды:

Скопировать кодBASH

brew tap mongodb/brew

brew install mongodb-community@4.4

Или воспользуйтесь официальной инструкцией: <https://docs.mongodb.com/manual/tutorial/install-mongodb-on-os-x/>

### Linux

Процесс установки описан в [официальной инструкции](https://docs.mongodb.com/manual/administration/install-on-linux/).

# Взаимодействие с Mongo

Прежде чем приступить к уроку, запустите сервер MongoDB: откройте терминал (macOS и Linux) или Git Bash (Windows) и введите команду:

Скопировать кодBASH

mongod

Если вы устанавливали Mongo на macOS через brew и команда mongod не запустила сервер, воспользуйтесь альтернативой. Но обратите внимание на номер версии mongodb — она должна совпадать с установленной:

Скопировать кодBASH

brew services start mongodb-community@4.2

Сервер запущен. Можно приступать к чтению.

## Как взаимодействовать с MongoDB

Есть несколько способов. Мы расскажем о двух: графическом интерфейсе и через сервер на ноде.

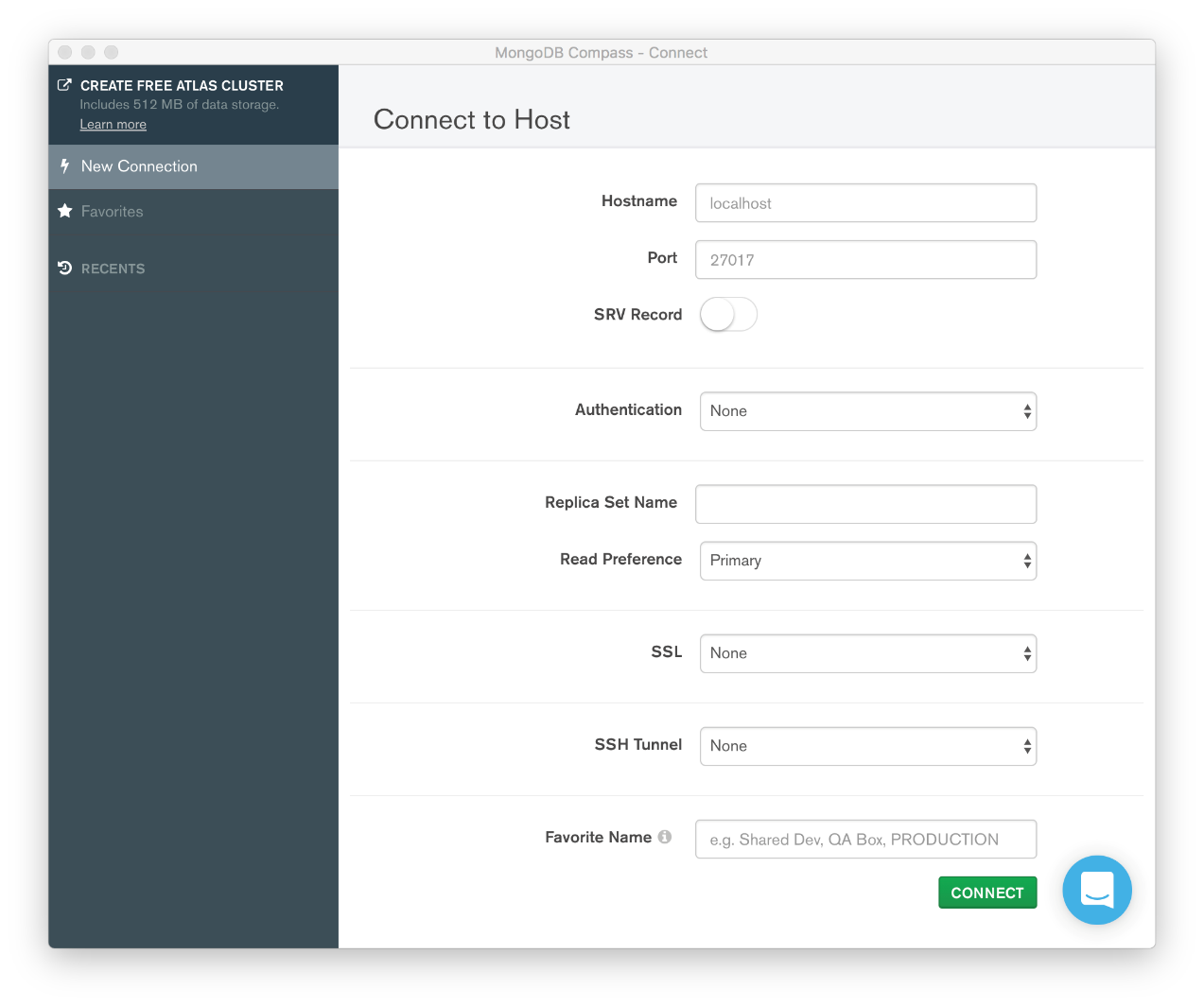
### Графический интерфейс

Это очень наглядный способ. Через графический интерфейс можно просматривать и редактировать базы данных и записи в них. От слов — к делу.

Если работаете на macOS или Linux, скачайте и установите Compass — графический интерфейс для MongoDB. На Windows он уже установлен вместе с самой Mongo.

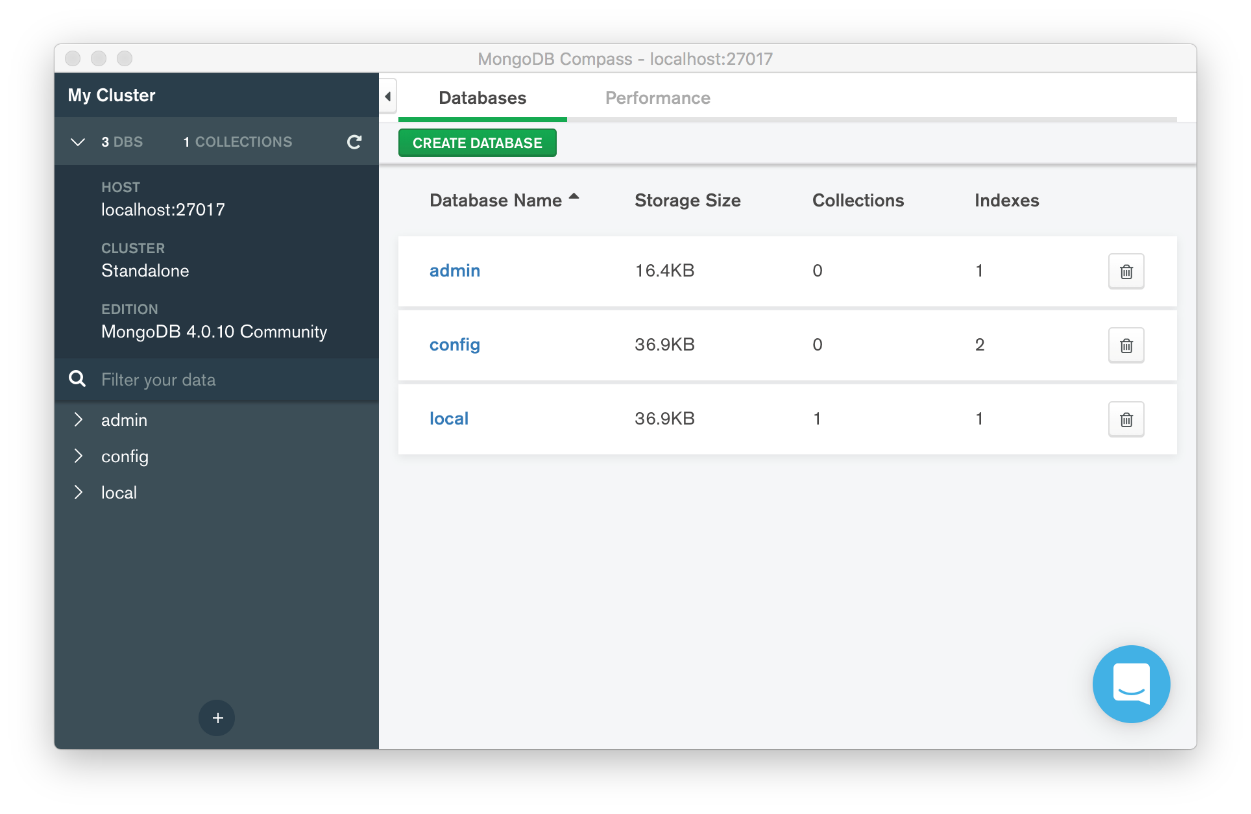
Ссылка на установщик Compass: <https://www.mongodb.com/download-center/compass>.

Откройте Compass. Перед вами окажется такой экран:

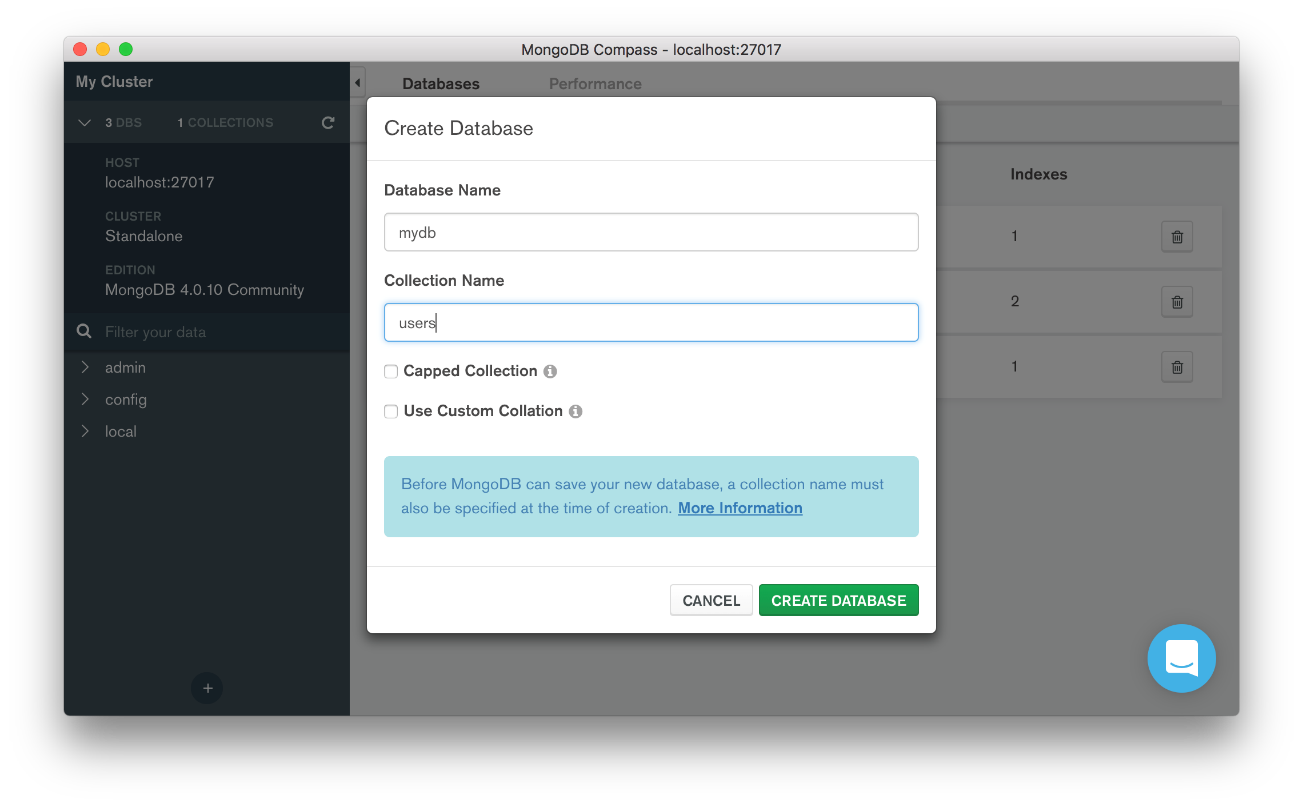


Оставьте всё как есть и нажмите Connect: Compass подключится к MongoDB серверу. Если этого не произошло и появилась красное сообщение об ошибке, проверьте, запущен ли MongoDB сервер.

Compass покажет уже имеющиеся базы данных. Создадим ещё одну. Нажмите Create Database:

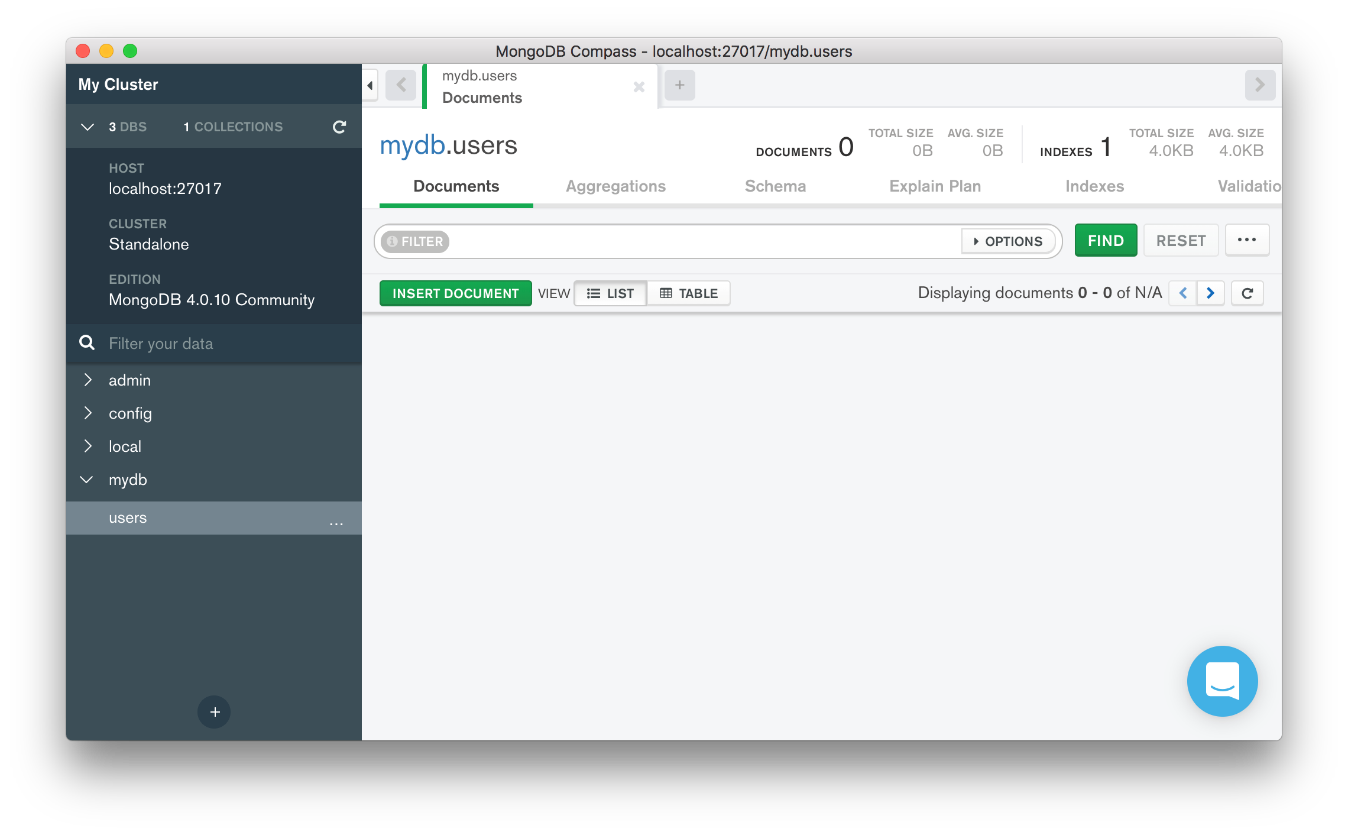


Введите имя новой БД (mydb) и первой коллекции в ней (users). Затем жмите Create Database:

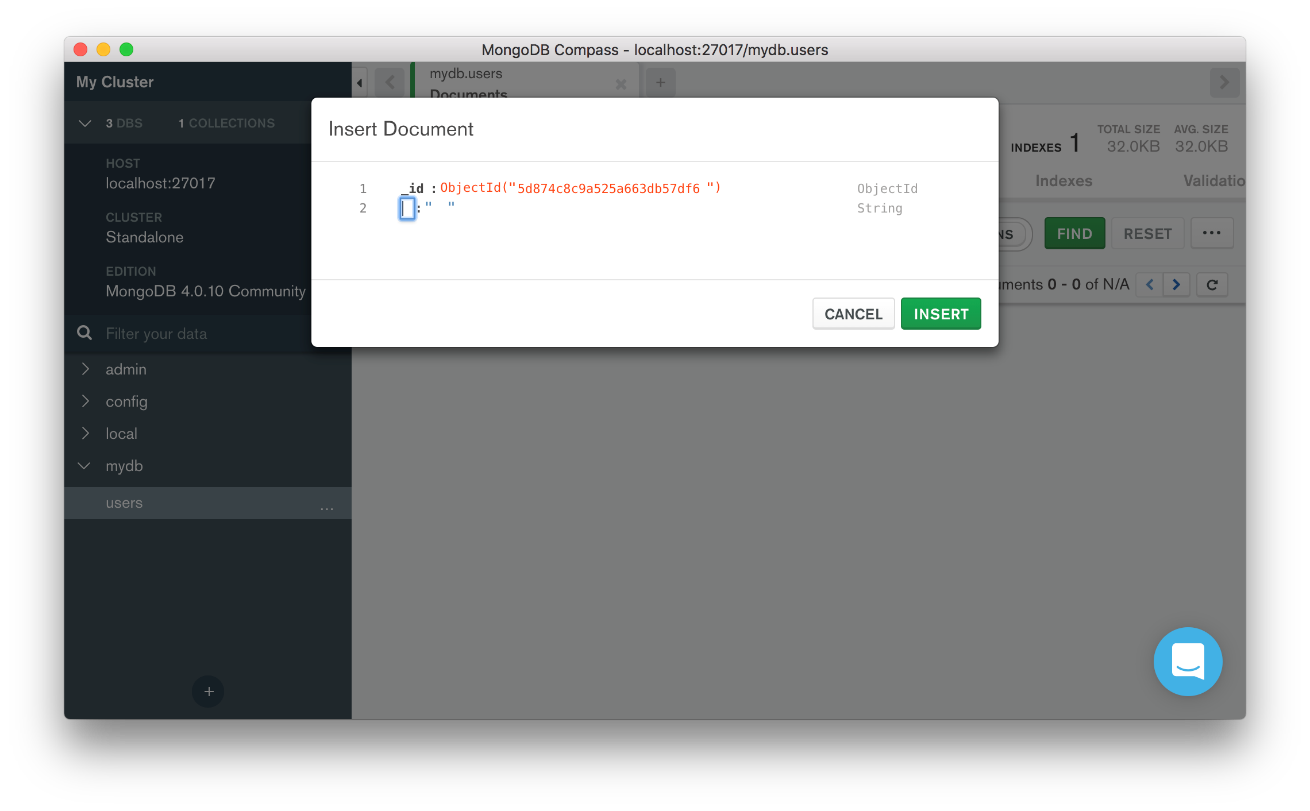


Имя базы нечувствительно к регистру: userdb и UserDB — одна и та же база. Чтобы не создавать конфликтующих имён, пользуйтесь только строчными буквами.

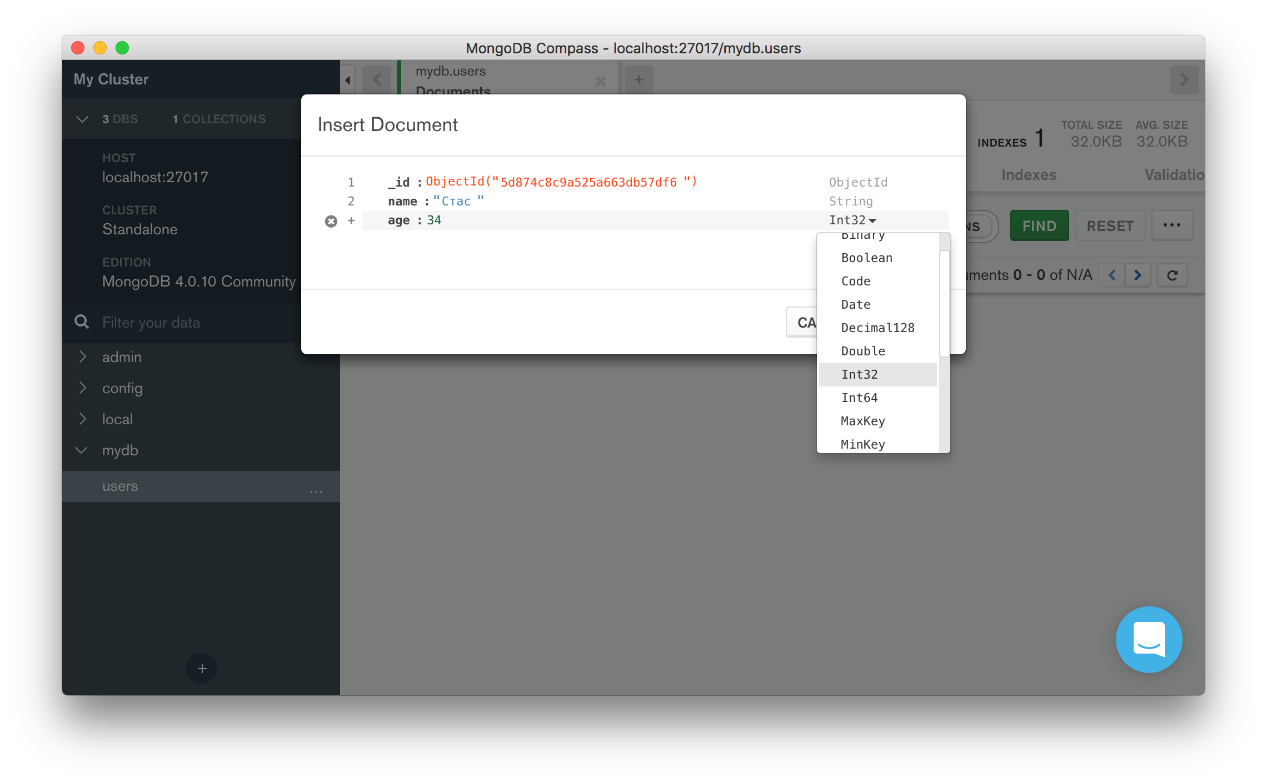
База данных состоит из коллекций, например, users, posts, comments. Ограничимся пока что одной — users. Нажмите Insert Document и создайте в ней документ:



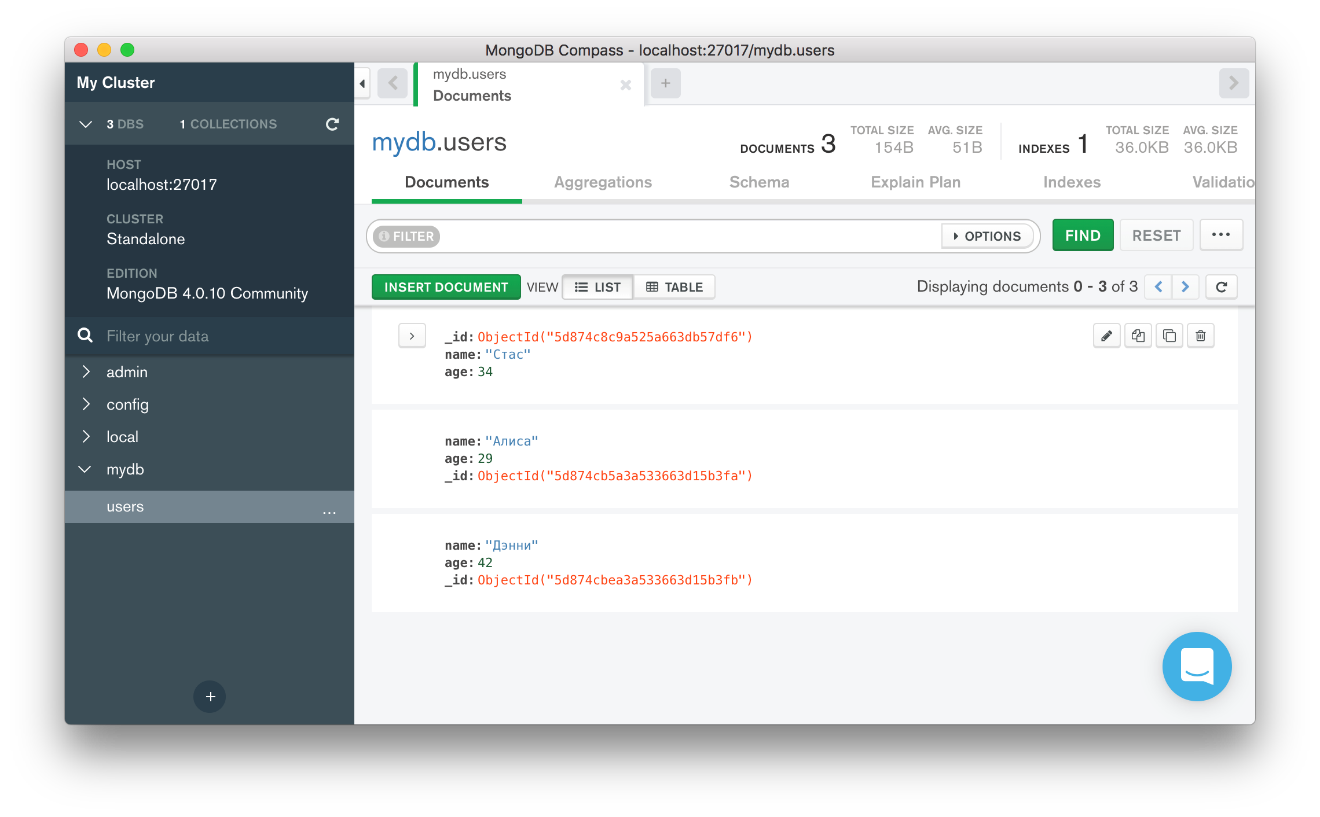
Откроется окно создания нового документа. У него уже есть одно поле — \_id:



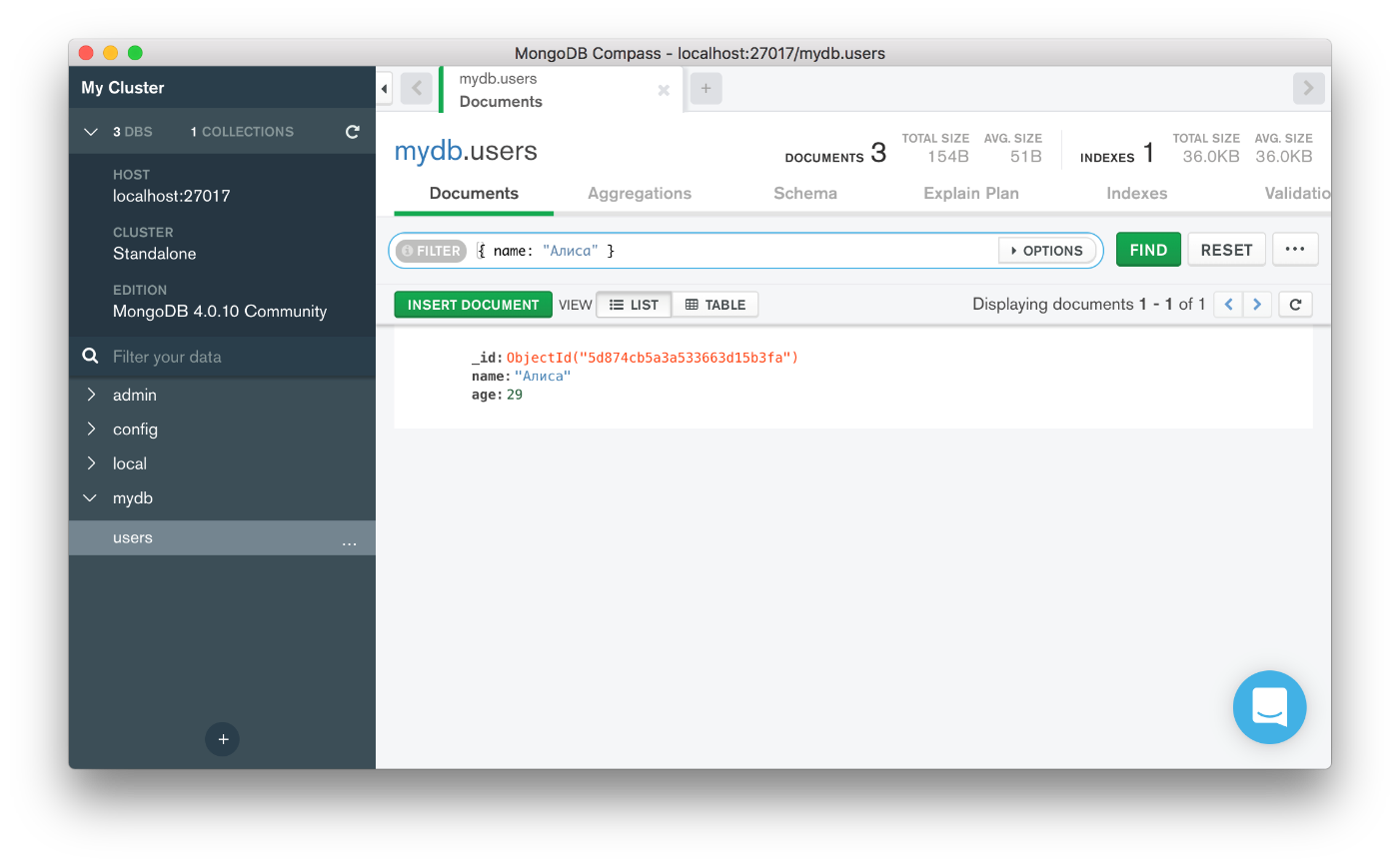
Это уникальный идентификатор документа, MongoDB создаёт его сам. Добавьте ещё два поля: name и age. В правом столбце можно выбрать тип данных, записанных в поле. Для age выберите числовой тип — Int32:



Затем нажмите Insert — и документ будет создан. Создайте ещё несколько:



Документы в коллекции можно фильтровать. Это помогает искать их в базе. Просто напишите запрос в текстовом поле Filter. Запрос должен представлять собой объект:



Вот ещё несколько примеров запросов:

Скопировать кодJAVASCRIPT

{} *// найдёт все документы коллекции*

{ name: 'Стас' } *// найдёт документы, где поле name равно "Стас"*

{ name: 'Стас', age: 34 } *// найдёт документы, где поле name равно "Стас" и поле age равно 34*

Искать документы через графический интерфейс очень удобно. Но добавлять их таким способом приходится нечасто — обычно этим занимается сервер. Для этого базу данных нужно связать с приложением на JavaScript. Этим мы и займёмся в следующем уроке.

## Дополнительные Ссылки

**Установщик Compass**

<https://www.mongodb.com/download-center/compass>

**Compass. Как писать запросы**

<https://docs.mongodb.com/manual/tutorial/query-documents/>

**Mongo Shell. Документация**

<https://docs.mongodb.com/manual/mongo/#working-with-the-mongo-shell>

# Подключение к mongo из JavaScript. Mongoose

В предыдущем уроке мы рассказывали, как взаимодействовать с базой данных из графического интерфейса. Но документы в БД создаёт сервер, а не человек.

Нужно как-то «подружить» сервер и базу данных. Наш сервер запрограммирован на JavaScript, который не умеет работать с документами. Зато в JS есть другие структуры данных — объекты. Наша задача: научить JavaScript работать с документами как с объектами.

Чтобы подружить JS с документами, существуют специальные инструменты — ODM, или Object Document Mapper (англ. «сопоставитель объектов и документов»). У каждой БД есть свой сопоставитель. У MongoDB он называется Mongoose и представляет собой мост между двумя мирами: миром документов в базе данных и миром объектов JavaScript.

### Устанавливаем Mongoose

Это делается через npm. Откройте в терминале папку с проектом и запустите команду:

Скопировать кодBASH

npm i mongoose

Теперь его можно импортировать как модуль и пользоваться.

### Подключаемся к серверу mongo

Эту задачу выполняет метод mongoose.connect:

Скопировать кодJAVASCRIPT

*// app.js — входной файл*

const express = require('express');

const mongoose = require('mongoose');

const app = express();

*// подключаемся к серверу mongo*

mongoose.connect('mongodb://localhost:27017/mydb', {

useNewUrlParser: true,

useCreateIndex: true,

useFindAndModify: false

});

*// подключаем мидлвары, роуты и всё остальное...*

app.listen(3000);

Этот метод принимает на вход 2 параметра: адрес сервера базы данных и объект опций.

1. Адрес состоит из двух частей. Первая — mongodb://localhost:27017 — адрес сервера mongo по умолчанию. Он запускается на localhost на 27017 порту. Вторая часть — mydb — имя базы данных.
2. У объекта опций может быть много свойств. Нам важны три:
   * useNewUrlParser: true,
   * useCreateIndex: true,
   * useFindAndModify: false.

Мы не будем вдаваться в долгие объяснения, зачем нужны эти опции. Но если их не указать, у Mongoose в будущем возникнут проблемы совместимости с MongoDB. Дело в том, что логику взаимодействия MongoDB с Mongoose сейчас переписывают, и мы указываем эти опции, чтобы приложение вскоре не сломалось. Поэтому пропишите опции как у нас.

### Запускаем сервер MongoDB

Сервер mongo запускают командой mongod:

Скопировать кодBASH

mongod

Это нужно делать до запуска Node.js приложения. Иначе оно не сможет подключиться к базе данных и взаимодействовать с ней.

# Схемы и модели

Одна из особенностей нереляционных баз данных — отсутствие схемы. Схема — набор требований к данным: сколько полей у записи, какой длины может быть значение каждого поля, какие символы в нём допустимы.

Схема накладывает ограничения на данные, которые записаны в базу. Иногда это удобно, иногда нет. Современные NoSQL базы позволяют использовать схемы выборочно.

В MongoDB нет поддержки схем по умолчанию, но мы можем добавить их через Mongoose. Это нужно, чтобы проверять, соответствует ли документ схеме, прежде чем записывать его в БД. Фактически, мы сможем валидировать данные перед записью.

### Создадим схему

В REST API есть ресурсы — в проекте Mesto это пользователи и карточки. Каждый из ресурсов должен соответствовать задуманной структуре: например, у пользователя должно быть имя и информация о себе. Зададим схему для пользователя через Mongoose:

Скопировать кодJAVASCRIPT

*// models/user.js*

const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({

name: { *// у пользователя есть имя — опишем требования к имени в схеме:*

type: String, *// имя — это строка*

required: true, *// оно должно быть у каждого пользователя, так что имя — обязательное поле*

minlength: 2, *// минимальная длина имени — 2 символа*

maxlength: 30, *// а максимальная — 30 символов*

},

gender: {

type: String, *// гендер — это строка*

enum: ['м', 'ж', 'другой'] *// gender может принимать одно из трёх значений*

},

about: String, *// тип — String*

});

Такая схема говорит, что каждый пользователь в системе должен:

* обладать именем длиной от 2 до 30 символов;
* иметь гендер, который может принимать одно из трёх значений;
* иметь свойство about, причём строковое.

Иногда стандартных свойств схемы недостаточно. В Mongoose вы можете писать и более тонкие способы проверки данных. Для этого существует свойство validate. Это объект, свойствами которого являются:

* validator — функция валидации. Она должна возвращать булевое значение.
* message — сообщение об ошибке. Срабатывает в том случае, если функция валидации возвращает false.

Посмотрим на примере. В схему пользователя добавим свойство age, значение которого не должно быть меньше 18:

Скопировать кодJAVASCRIPT

*// models/user.js*

const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({

name: { *// у пользователя есть имя — опишем требования к имени в схеме:*

type: String, *// имя — это строка*

required: true, *// оно должно быть у каждого пользователя, так что имя — обязательное поле*

minlength: 2, *// минимальная длина имени — 2 символа*

maxlength: 30, *// а максимальная — 30 символов*

},

gender: {

type: String, *// гендер — это строка*

enum: ['м', 'ж', 'другой'] *// gender может принимать одно из трёх значений*

},

age: { *// у пользователя есть возраст*

type: Number, *// возраст - число*

validate: { *// опишем свойство validate*

validator(v) { *// validator - функция проверки данных. v - значение свойства age*

return v >= 18; *// если возраст меньше 18, вернётся false*

},

message: 'Вам должно быть больше 18 лет!', *// когда validator вернёт false, будет использовано это сообщение*

}

},

about: String, *// тип — String*

});

Это простая проверка данных. Более сложную можно сделать с помощью регулярных выражений. Об этом расскажем позже.

Вернёмся к типам данных. При создании схем мы будем применять 5 основных типов данных:

Скопировать кодJAVASCRIPT

String *// строка*

Number *// число*

Date *// дата*

Boolean *// логическое: true или false*

Array *// массив*

### Свойства-объекты

Если свойство документа должно быть объектом, придётся воспользоваться методом Schema дважды:

* первый вызов нужен для создания схемы объекта: в нём мы описываем, какую структуру должно иметь свойство;
* второй — для передачи описанной схемы в свойство, которое должно иметь эту структуру:

Скопировать кодJAVASCRIPT

*// models/user.js*

const mongoose = require('mongoose');

*// создадим схему документа «Домашнее животное»*

const petSchema = new mongoose.Schema({

name: {

type: String,

required: true,

minlength: 2,

maxlength: 30,

},

age: Number

});

*// Когда схема готова, передадим её в свойство, которое должно соответствовать описанному шаблону:*

const userSchema = new mongoose.Schema({

...

pet: petSchema *// опишем свойство pet этой схемой*

});

### Свойства-массивы

Массивы нужны для хранения однотипных данных, поэтому описание схемы массива сводится к описанию шаблона элемента:

Скопировать кодJAVASCRIPT

*// models/user.js*

const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({

...

hobbies: [{ *// описываем схему для одного элемента и заключаем её в квадратные скобки*

type: String,

minlength: 2,

maxlength: 30,

}]

});

Эта схема определяет свойство hobbies: оно должно содержать массив строк. Каждая — длиной от 2 до 30 символов.

### Создание модели на основе схемы

Схемой мы определили, каким должен быть документ в базе. Перейдём к следующему этапу — созданию самих документов. Для этого на основе схемы строится модель.

Модель — это «обёртка» из методов вокруг схемы. Благодаря ей мы можем читать, добавлять, удалять и обновлять документы. В Mongoose модель создают методом mongoose.model.

Скопировать кодJAVASCRIPT

*// models/user.js*

const mongoose = require('mongoose');

*// Опишем схему:*

const userSchema = new mongoose.Schema({

name: {

type: String,

required: true,

minlength: 2,

maxlength: 30,

},

about: String,

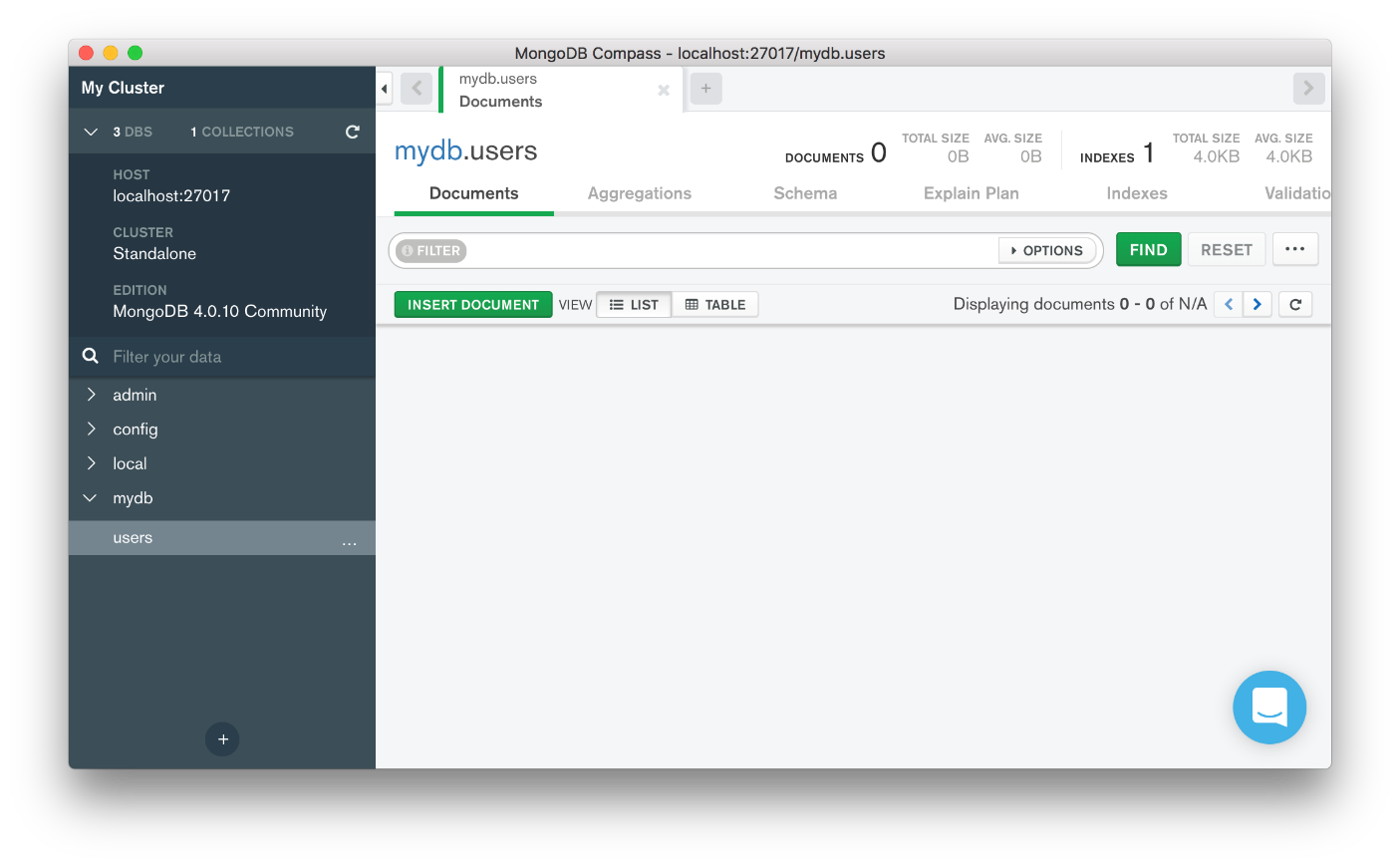
});

*// создаём модель и экспортируем её*

module.exports = mongoose.model('user', userSchema);

Мы передали методу mongoose.model два аргумента: имя модели и схему, которая описывает будущие документы.

Аккуратно — тут можно запутаться. Первый аргумент — имя модели — должно быть существительным в единственном числе. Но Compass отображает его во множественном. Дело в том, что Mongoose автоматически добавляет букву "s" в конце имени коллекции:



### Создание моделей: резюме

1. Определите ресурсы API.
2. Опишите схемы этих ресурсов.
3. Создайте модели на основе схем.

Когда модели готовы, можно использовать их для взаимодействия с базой: создавать, изменять и удалять документы в ней. Этим и займёмся в следующем уроке.

### Дополнительные ссылки

Типы данных в схемах Mongoose: <https://mongoosejs.com/docs/schematypes.html>

Больше про схемы: <https://mongoosejs.com/docs/guide.html>

# Создание, чтение, обновление и удаление документов

В предыдущем уроке мы создали модель. Она нужна, чтобы взаимодействовать с соответствующей коллекцией документов. Модель пользователя позволяет управлять коллекцией пользователей, модель карточек — коллекцией карточек.

С документом можно совершить одно из 4-х действий:

* создать (create),
* прочитать (read),
* обновить (update),
* удалить (delete).

Эти четыре действия обозначают аббревиатурой CRUD — по первым буквам английских названий. Разберём наиболее распространённые методы для каждого из этих действий.

### Создание документов — C

За это отвечает метод модели create. Он принимает на вход объект с данными, которые нужно записать в базу:

Скопировать кодJAVASCRIPT

*// routes/users.js*

*/\**

*всякий код для создания роутеров и т.п.*

*\*/*

*// импортируем модель*

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

router.post('/', (req, res) => {

const { name, about } = req.body; *// получим из объекта запроса имя и описание пользователя*

User.create({ name, about }); *// создадим документ на основе пришедших данных*

});

Метод create может быть промисом — ему можно добавить обработчики then и catch. Так обычно и делают, чтобы вернуть клиенту данные или ошибку:

Скопировать кодJAVASCRIPT

*// routes/users.js*

*/\**

*всякий код для создания роутеров и т.п.*

*\*/*

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

router.post('/', (req, res) => {

const { name, about } = req.body;

User.create({ name, about })

*// вернём записанные в базу данные*

.then(user => res.send({ data: user }))

*// данные не записались, вернём ошибку*

.catch(err => res.status(500).send({ message: 'Произошла ошибка' }));

});

### Чтение документов — R

Прочитать документ — значит найти его и получить данные. Искать можно разными способами. Три самых популярных: findById, findOne и find.

* Поиск конкретного документа. Метод findById ищет запись по идентификатору, то есть свойству \_id, которое есть у каждого документа в MongoDB. Поэтому чтобы найти конкретный документ, передайте методу findById идентификатор в строковом виде:

Скопировать кодJAVASCRIPT

*// routes/users.js*

*/\**

*всякий код для создания роутеров и т.п.*

*\*/*

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

router.get('/:id', (req, res) => {

User.findById(req.params.id)

.then(user => res.send({ data: user }))

.catch(err => res.status(500).send({ message: 'Произошла ошибка' }));

});

* Поиск одного документа по параметрам. Метод findOne возвращает первый документ, соответствующий запросу:

Скопировать кодJAVASCRIPT

*// найти первое совпадение с полем name равным "Стас Басов"*

User.findOne({ name: 'Стас Басов' });

* Поиск всех документов по параметрам. Метод find работает так же, как findOne, но возвращает все документы по запросу:

Скопировать кодJAVASCRIPT

*// найти всех тридцатилетних*

User.find({ age: 30 });

*// найти вообще всех*

User.find({});

### Обновление документов — U

Тут всё очень похоже на чтение. При обновлении мы находим запись и меняем её свойства.

* Обновление конкретной записи. Для этого есть метод findByIdAndUpdate, который принимает на вход идентификатор в строковом виде. Вторым параметром ему передают объект со свойствами, которые нужно обновить:

Скопировать кодJAVASCRIPT

*// routes/users.js*

*// ...*

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

router.patch('/:id', (req, res) => {

*// обновим имя найденного по \_id пользователя*

User.findByIdAndUpdate(req.params.id, { name: 'Виктор Гусев' })

.then(user => res.send({ data: user }))

.catch(err => res.status(500).send({ message: 'Произошла ошибка' }));

});

* Поиск первого совпадения с запросом и обновление его. Для этого есть метод findOneAndUpdate, который принимает на вход два объекта: первый с параметрами поиска, второй — с данными, которые нужно обновить:

Скопировать кодJAVASCRIPT

*// найти первое совпадение с полем name равным "Стас Басов" и заменить имя на "Виктор Гусев"*

User.findOneAndUpdate({ name: 'Стас Басов' }, { name: 'Виктор Гусев' });

* Поиск совпадения с запросом и преобразование его. Аналогично можно превратить всех Стасов Басовых в Викторов Гусевых методом update:

Скопировать кодJAVASCRIPT

*// найти все совпадения с полем name равным "Стас Басов" и заменить имя на "Виктор Гусев"*

User.update({ name: 'Стас Басов' }, { name: 'Виктор Гусев' });

Есть тонкость в работе методов обновления: по умолчанию параметр, который получает на вход обработчик then — это документ до обновления:

Скопировать кодJAVASCRIPT

User.findByIdAndUpdate(req.params.id, { name: 'Виктор Гусев' })

*// user здесь — это документ до обновления*

.then(user => res.send({ data: user }));

Это можно исправить. Поэтому третьим аргументом методы обновления документов принимают объект опций. Нас интересуют три из них:

| **ОПЦИЯ** | **ЧТО ЗНАЧИТ** | **ЗНАЧЕНИЕ ПО УМОЛЧАНИЮ** |
| --- | --- | --- |
| new | передать обновлённый объект на вход обработчику then | false |
| runValidators | валидировать новые данные перед записью в базу | false |
| upsert | если документ не найден, создать его | false |

Благодаря объекту опций мы можем передать в then уже обновлённую запись. А также настроить валидацию и создать документ, если он не был найден:

Скопировать кодJAVASCRIPT

User.findByIdAndUpdate(

req.params.id,

{ name: 'Виктор Гусев' },

*// Передадим объект опций:*

{

new: true, *// обработчик then получит на вход обновлённую запись*

runValidators: true, *// данные будут валидированы перед изменением*

upsert: true *// если пользователь не найден, он будет создан*

}

)

.then(user => res.send({ data: user }))

.catch(user => res.send({ "Данные не прошли валидацию. Либо произошло что-то совсем немыслимое" }));

### Удаление документов — D

* **Удаление конкретной записи.** Для этого есть метод findByIdAndRemove:

Скопировать кодJAVASCRIPT

*// routes/users.js*

*// ...*

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

router.delete('/:id', (req, res) => {

User.findByIdAndRemove(req.params.id)

.then(user => res.send({ data: user }))

.catch(err => res.status(500).send({ message: 'Произошла ошибка' }));

});

* **Удаление первого совпадения.** Этой цели служит findOneAndRemove:

Скопировать кодJAVASCRIPT

*// удалим пользователя на основе имени*

User.findOneAndRemove({ name: 'Стас Басов' });

* **Удаление всех совпадений.** Для этого вызывайте deleteMany:

Скопировать кодJAVASCRIPT

*// удалим всех тридцатилетних*

User.deleteMany({ age: 30 });

## Ссылки

В этом уроке мы перечислили лишь основные методы, но у mongoose моделей методов очень много. Все их можно найти в официальной документации:

<https://mongoosejs.com/docs/api/model.html>

# Как структурировать код. Контроллеры

В предыдущем уроке мы связали роуты с базой данных. Когда на роут приходит запрос, мы обращаемся к базе и возвращаем клиенту ответ:

Скопировать кодJAVASCRIPT

*// routes/users.js*

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

router.post('/', (req, res) => {

const { name, about } = req.body;

*// записываем данные в базу*

User.create({ name, about })

*// возвращаем записанные в базу данные пользователю*

.then(user => res.send({ data: user }))

*// если данные не записались, вернём ошибку*

.catch(err => res.status(500).send({ message: 'Произошла ошибка' }));

});

Если роутов много, они теряются в коде. Для удобства работы разделим код на два файла: файл контроллеров и файл маршрутов. Вынесем обработчики в отдельную папку controllers.

Скопировать кодJAVASCRIPT

*// controllers/users.js*

*// это файл контроллеров*

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

module.exports.createUser = (req, res) => {

const { name, about } = req.body;

User.create({ name, about })

.then(user => res.send({ data: user }))

.catch(err => res.status(500).send({ message: 'Произошла ошибка' }));

};

Скопировать кодJAVASCRIPT

*// routes/users.js*

*// это файл маршрутов*

const { createUser } = require('../controllers/users');

router.post('/', createUser);

**Контроллер** — функция, ответственная за взаимодействие с моделью. То есть это функция, которая выполняет создание, чтение, обновление или удаление документа.

Контроллер в express также называют «последней мидлвэрой». Потому что внутри неё мы не вызываем next, а возвращаем ответ пользователю.

Файл контроллеров описывает логику обработки запросов. А файл маршрутов определяет, при каком запросе эту логику применять.

# Настраиваем связи

Раньше NoSQL означала полностью нереляционный подход к созданию баз данных. Со временем нереляционные базы данных стали вбирать в себя некоторые сильные стороны реляционных. И одна из таких сильных сторон — связи. В этом уроке разберёмся, как настроить связи между моделями в MongoDB.

## 1. Настраиваем связь двух схем

Построение связи начинается со схемы. Представим приложение, в котором есть две сущности: пользователи и объявления. Вот схема пользователя:

Скопировать кодJAVASCRIPT

const userSchema = new mongoose.Schema({

name: { *// у пользователя есть только имя*

type: String,

minlength: 2,

maxlength: 20,

required: true,

},

});

module.exports = mongoose.model('user', userSchema);

Пользователи могут создавать объявления, которые описываются двумя полями: title и text — название объявления и его текст:

Скопировать кодJAVASCRIPT

const adSchema = new mongoose.Schema({

title: {

type: String,

minlength: 2,

maxlength: 20,

required: true,

},

text: {

type: String,

minlength: 2,

required: true,

},

});

module.exports = mongoose.model('ad', adSchema);

Но нужно как-то определять, кто автор каждого объявления. Поэтому объявлению необходимо ещё одно поле — создатель, назовём его creator. В этом поле будем хранить ссылку на автора объявления.

Лучшая ссылка из одного документа на другой — идентификатор. Mongo автоматически создаёт поле \_id — уникальный идентификатор для каждого документа. Этот идентификатор позволяет связать один документ с другим.

Чтобы сделать это на уровне схемы, полю следует установить специальный тип — mongoose.Schema.Types.ObjectId и свойство ref. В это свойство записывают имя модели, на которую мы ссылаемся:

Скопировать кодJAVASCRIPT

const adSchema = new mongoose.Schema({

title: {

type: String,

minlength: 2,

maxlength: 20,

required: true,

},

text: {

type: String,

minlength: 2,

required: true,

},

*// создаём поле creator*

creator: {

type: mongoose.Schema.Types.ObjectId,

ref: 'user',

required: true

},

});

## 2. При создании документа записываем \_id

В первом пункте мы создали модель, указав, что в поле creator должен быть идентификатор документа пользователя. Теперь этот идентификатор нужно записывать в поле creator при создании нового объявления:

Скопировать кодJAVASCRIPT

*// controllers/ads.js*

const Ad = require('../models/ad');

module.exports.createAd = (req, res) => {

const { title, text, creatorId } = req.body;

Ad.create({ title, text, creator: creatorId })

.then(ad => res.send({ data: ad }));

};

## 3. Получаем всю информацию — метод populate

Мы связали две модели. Тем не менее схема объявления хранит только идентификатор пользователя. Чтобы получить всю информацию об авторе объявления, нужно вызвать метод populate, передав ему имя поля:

Скопировать кодJAVASCRIPT

*// controllers/ads.js*

const Ad = require('../models/ad');

module.exports.getAds = (req, res) => {

Ad.find({})

.populate('creator')

.then(ad => res.send({ data: ad }));

};

Чтобы отправить в ответе несколько полей, полученных из связей, методу populate следует передавать массив:

Скопировать кодJAVASCRIPT

*// controllers/ads.js*

const Ad = require('../models/ad');

module.exports.getAds = (req, res) => {

Ad.find({})

.populate(['creator', 'followers'])

.then(ad => res.send({ data: ad }));

};

## Ссылки

Более подробно про связи можно узнать в документации mongoose

<https://mongoosejs.com/docs/populate.html>