**Введение**

В этой теме вы завершите изучение аутентификации пользователя. Вы уже реализовали авторизацию на стороне пользователя, теперь пришло время сделать это на стороне сервера.

Сначала вы узнаете, как устроен процесс создания пользователя. После — расскажем о специальных методах Mongoose, которые помогут с авторизацией. Следующая важная часть — создание, хранение и верификация ключей. Эти темы касаются и облегчения жизни пользователя, и безопасности вашего сервера. Важно правильно хранить ключи, чтобы ими не воспользовались злоумышленники.

Кроме этого, вы узнаете о безопасности паролей пользователя. Научитесь хешировать пароли, чтобы получить доступ к данным пользователя стало ещё сложнее. А слово «соль» приобретёт новое значение. Но об этом позже.

А пока перейдём к созданию пользователя.

# Создание пользователя

В предыдущем спринте вы разобрали понятие аутентификации — входа пользователя на сайт, — а затем написали её на стороне фронтенда. Теперь поработаем с процессами аутентификации и регистрации на стороне сервера. Регистрация с точки зрения сервера — создание нового пользователя в базе данных. В этом уроке мы расскажем, как это сделать.

## Добавление пользователя в базу данных

Сначала создадим, а затем запишем пользователя в базу данных.

### Создание

Создадим модель пользователя с двумя полями:

Скопировать кодJAVASCRIPT

*// models/user.js*

const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({

email: {

type: String,

required: true,

unique: true

},

password: {

type: String,

required: true,

minlength: 8

}

});

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

Email должен быть уникальным, поскольку пользователь проходит аутентификацию по электронной почте. Для этого мы добавляем свойство unique со значением true. Так в базе не окажется несколько пользователей с одинаковой почтой.

### Запись пользователя в БД

Настроим запись пользователя в базу данных при запросе к серверу. Для этого создадим контроллер createUser. Внутри него мы получим почту и пароль, которые пришли в запросе, и запишем их в базу:

Скопировать кодJAVASCRIPT

*// controllers/users.js*

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

exports.createUser = (req, res) => User.create({

email: req.body.email,

password: req.body.password,

})

.then((user) => res.send(user))

.catch((err) => res.status(400).send(err));

Если пользователь с такой почтой уже есть в базе, сработает блок catch и записи не произойдёт.

## Хеширование пароля модуль bcryptjs метод hash

Пароль в базе следует хранить в зашифрованном виде. Иначе пользователи окажутся в уязвимом положении: если у злоумышленника окажется доступ к базе данных, он получит доступ и к аккаунтам.

Пароли хранят в виде хеша. Из него невозможно получить пароль, поэтому злоумышленник не сможет добраться до аккаунтов пользователей даже если завладеет базой данных.

Для хеширования пароля понадобится модуль bcryptjs. Его нужно установить npm install bcryptjs, как мы делали это с другими модулями, а затем импортировать в проект:

Скопировать кодJAVASCRIPT

*// controllers/users.js*

const bcrypt = require('bcryptjs'); *// импортируем bcrypt*

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

Добавим код для хеширования в контроллер создания пользователя. За это отвечает метод hash:

Скопировать кодJAVASCRIPT

*// controllers/users.js*

const bcrypt = require('bcryptjs'); *// импортируем bcrypt*

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

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

*// хешируем пароль*

bcrypt.hash(req.body.password, 10)

.then(hash => User.create({

email: req.body.email,

password: hash, *// записываем хеш в базу*

}))

.then((user) => res.send(user))

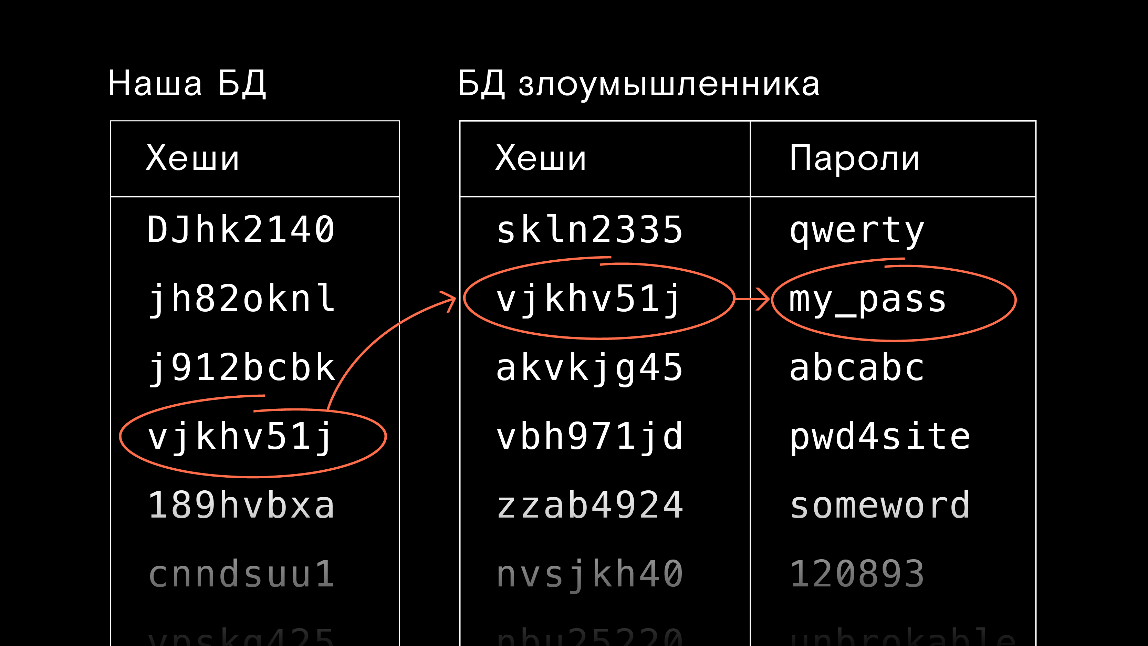
.catch((err) => res.status(400).send(err));

};

Метод принимает на вход два параметра: пароль и длину так называемой «соли» — случайной строки, которую метод добавит к паролю перед хешированем.

### Зачем нужна «соль»? Таблица поиска

Хешировать пароль без прибавления «соли» небезопасно. Можно сгенерировать множество хешей и собрать их в таблицу, а затем украсть базу данных и попытаться найти соответствие между двумя хешами. Если такое совпадение найдётся, злоумышленник получит пароль пользователя.



«Соль» решает эту проблему. Прежде чем хешировать пароль, к нему добавятся случайные символы — это полностью поменяет итоговый хеш. Даже для двух одинаковых паролей получатся разные хеши. Тогда злоумышленнику нужно составить по отдельной таблице хешей для каждой соли. Это по сути прямой перебор.

Пример

Сейчас при создании пользователя в ответе возвращается статус 201 и объект пользователя, который содержит хеш пароля. Передавать хеш пароля обратно пользователю ни к чему — это лишняя информация.

Сделайте так, чтобы в ответе на запрос создания пользователя возвращался JSON с двумя полями: \_id (идентификатор пользователя) и email (его почта).

const path = require('path');

const express = require('express');

const bodyParser = require('body-parser');

const mongoose = require('mongoose');

const bcrypt = require('bcryptjs');

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

const { PORT = 3000, BASE\_PATH } = process.env;

const app = express();

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

useNewUrlParser: true,

useCreateIndex: true,

useFindAndModify: false,

useUnifiedTopology: true,

});

app.use(bodyParser.urlencoded({ extended: false }));

app.use(bodyParser.json());

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

bcrypt.hash(req.body.password, 10)

.then((hash) => User.create({

email: req.body.email,

password: hash,

}))

.then((user) => {

res.status(201).send( { \_id: user.\_id, email: user.email });

})

.catch((err) => {

res.status(400).send(err);

});

});

app.use(express.static(path.join(\_\_dirname, 'public')));

app.listen(PORT, () => {

console.log('Ссылка на сервер:');

console.log(BASE\_PATH);

});

# Аутентификация на практике

В предыдущем уроке вы научились добавлять пользователя в базу данных. Теперь перейдём к аутентификации. В этом уроке реализуем базовый способ аутентификации — по адресу почты и паролю.

## Создаём контроллер аутентификации

Чтобы войти в систему, пользователь отправляет на сервер почту и пароль. Это удобно делать в теле запроса:

Скопировать кодJAVASCRIPT

*// controllers/users.js*

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

const { email, password } = req.body;

*// ...*

};

Если почта и пароль совпадают с теми, что есть в базе, пользователь входит на сайт. Иначе — получает сообщение об ошибке. Проверить корректность данных можно по-разному:

* поискать пользователя с полученной почтой в базе. Если пользователь нашёлся, высчитать хеш пароля и сравнить его с хешем в базе;
* посчитать хеш пароля и проверить, есть ли в базе пользователь с полученной почтой и хешем пароля.

Мы будем пользоваться первым способом — он эффективнее. При втором подходе считать хеш пароля придётся в любом случае, даже если пользователя с указанной почтой не существует. А подсчёт хеша каждый раз занимает ресурсы процессора.

### Ищем пользователя

Проверим, есть ли пользователь в базе:

Скопировать кодJAVASCRIPT

*// controllers/users.js*

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

const { email, password } = req.body;

User.findOne({ email })

.then((user) => {

if (!user) {

*// пользователь с такой почтой не найден*

}

*// пользователь найден*

});

};

Если пользователь не найден, перейдём в блок catch и вернём ответ с ошибкой:

Скопировать кодJAVASCRIPT

*// controllers/users.js*

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

const { email, password } = req.body;

User.findOne({ email })

.then((user) => {

if (!user) {

*// пользователь не найден — отклоняем промис*

*// с ошибкой и переходим в блок catch*

return Promise.reject(new Error('Неправильные почта или пароль'));

}

*// пользователь найден*

})

.catch((err) => {

*// возвращаем ошибку аутентификации*

res

.status(401)

.send({ message: err.message });

});

};

### Проверяем пароль метод bcrypt.compare

Если пользователь найден, проверим пароль: захешируем его и сравним с хешем в базе. Для этого есть метод bcrypt.compare. Он принимает на вход пароль и его хеш. Метод посчитает хеш и сравнит его с тем хешем, который мы передали вторым аргументом:

Скопировать кодJAVASCRIPT

*// controllers/users.js*

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

const { email, password } = req.body;

User.findOne({ email })

.then((user) => {

if (!user) {

return Promise.reject(new Error('Неправильные почта или пароль'));

}

*// сравниваем переданный пароль и хеш из базы*

return bcrypt.compare(password, user.password);

})

.catch((err) => {

res

.status(401)

.send({ message: err.message });

});

};

Метод bcrypt.compare работает асинхронно, поэтому результат нужно вернуть и обработать в следующем then. Если хеши совпали, в следующий then придёт true, иначе — false:

Скопировать кодJAVASCRIPT

*// controllers/users.js*

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

const { email, password } = req.body;

User.findOne({ email })

.then((user) => {

if (!user) {

return Promise.reject(new Error('Неправильные почта или пароль'));

}

return bcrypt.compare(password, user.password);

})

.then((matched) => {

if (!matched) {

*// хеши не совпали — отклоняем промис*

return Promise.reject(new Error('Неправильные почта или пароль'));

}

*// аутентификация успешна*

res.send({ message: 'Всё верно!' });

})

.catch((err) => {

res

.status(401)

.send({ message: err.message });

});

};

Мы намеренно сделали текст ошибки независимым от проблемы. Неважно, не найдена почта или неверно указан пароль — мы всегда передаём ответ «Неправильная почта или пароль». Иначе злоумышленник мог бы подобрать зарегистрированные почты и попытаться взломать именно их. В нашем случае мошенник не узнает, где допустил ошибку.

# Собственные методы моделей Mongoose

В предыдущем уроке мы написали код контроллера логина:

Скопировать кодJAVASCRIPT

*// controllers/users.js*

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

const { email, password } = req.body;

User.findOne({ email })

.then((user) => {

if (!user) {

return Promise.reject(new Error('Неправильные почта или пароль'));

}

return bcrypt.compare(password, user.password);

})

.then((matched) => {

if (!matched) {

*// хеши не совпали — отклоняем промис*

return Promise.reject(new Error('Неправильные почта или пароль'));

}

*// аутентификация успешна*

res.send({ message: 'Всё верно!' });

})

.catch((err) => {

res

.status(401)

.send({ message: err.message });

});

};

Эта цепочка промисов работает так:

* проверяет, есть ли в базе пользователь с указанной почтой;
* если пользователь найден, сверяет хеш его пароля;

В этом уроке мы улучшим код: сделаем код проверки почты и пароля частью схемы User. Для этого напишем метод findUserByCredentials, который принимает на вход два параметра — почту и пароль — и возвращает объект пользователя или ошибку.

Сделать это позволяет Mongoose. Чтобы добавить собственный метод, запишем его в свойство statics нужной схемы:

Скопировать кодJAVASCRIPT

*// models/user.js*

const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({

email: {

type: String,

required: true,

unique: true

},

password: {

type: String,

required: true,

minlength: 8

}

});

*// добавим метод findUserByCredentials схеме пользователя*

*// у него будет два параметра — почта и пароль*

userSchema.statics.findUserByCredentials = function (email, password) {

};

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

Осталось написать код метода. В будущем мы хотим использовать метод вот так:

Скопировать кодJAVASCRIPT

User.findUserByCredentials('stasbasov@yandex.ru', 'StasBasov1989')

.then(user => {

*// получаем объект пользователя, если почта и пароль подошли*

})

.catch(err => {

*// получаем ошибку, если нет*

});

### Описываем метод findUserByCredentials

Чтобы найти пользователя по почте, нам потребуется метод findOne, которому передадим на вход email. Метод findOne принадлежит модели User, поэтому обратимся к нему через ключевое слово this:

Скопировать кодJAVASCRIPT

*// models/user.js*

userSchema.statics.findUserByCredentials = function (email, password) {

*// попытаемся найти пользовател по почте*

return this.findOne({ email }) *// this — это модель User*

.then((user) => {

*// не нашёлся — отклоняем промис*

if (!user) {

return Promise.reject(new Error('Неправильные почта или пароль'));

}

*// нашёлся — сравниваем хеши*

return bcrypt.compare(password, user.password);

});

};

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

Функция findUserByCredentials не должна быть стрелочной. Это сделано, чтобы мы могли пользоваться this. Иначе оно было бы задано статически, ведь стрелочные функции запоминают значение this при объявлении.

Осталось добавить обработку ошибки, когда хеши не совпадают. Опишем этот код в ещё одном обработчике then.

Сначала разберём, как делать не стоит:

Скопировать кодJAVASCRIPT

*// models/user.js*

userSchema.statics.findUserByCredentials = function (email, password) {

*// попытаемся найти пользователя по почте*

return this.findOne({ email })

.then((user) => {

*// не нашёлся — отклоняем промис*

if (!user) {

return Promise.reject(new Error('Неправильные почта или пароль'));

}

*// нашёлся — сравниваем хеши*

return bcrypt.compare(password, user.password);

})

.then((matched) => {

if (!matched) *// отклоняем промис*

return user; *// но переменной user нет в этой области видимости*

});

};

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

Здесь во втором then мы обращаемся к объекту user, которого нет в этой области видимости. Он остался в предыдущем then.

Чтобы решить эту проблему, следует организовать цепочку промисов иначе: добавить обработчик then вызову метода bcrypt.compare:

Скопировать кодJAVASCRIPT

*// models/user.js*

const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({

email: {

type: String,

required: true,

unique: true

},

password: {

type: String,

required: true,

minlength: 8

}

});

userSchema.statics.findUserByCredentials = function (email, password) {

return this.findOne({ email })

.then((user) => {

if (!user) {

return Promise.reject(new Error('Неправильные почта или пароль'));

}

return bcrypt.compare(password, user.password)

.then((matched) => {

if (!matched) {

return Promise.reject(new Error('Неправильные почта или пароль'));

}

return user; *// теперь user доступен*

});

});

};

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

Код метода готов. Теперь мы можем применить его в обработчике аутентификации:

Скопировать кодJAVASCRIPT

*// controllers/users.js*

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

const { email, password } = req.body;

return User.findUserByCredentials(email, password)

.then((user) => {

*// аутентификация успешна! пользователь в переменной user*

})

.catch((err) => {

*// ошибка аутентификации*

res

.status(401)

.send({ message: err.message });

});

};

Теперь нужно сделать так, чтобы система запоминала пользователя и ему не приходилось вводить данные каждый раз, когда он открывает приложение. Как это сделать — разберём в следующем уроке.

О собственных методах схем в документации Mongoose: <https://mongoosejs.com/docs/guide.html#statics>.

# Как не проходить аутентификацию при каждом заходе на сайт. JWT

Вы уже разобрали алгоритм аутентификации. Пользователи могут ввести электронную почту, пароль, и залогиниться.

Скопировать кодJAVASCRIPT

*// controllers/users.js*

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

const { email, password } = req.body;

return User.findUserByCredentials(email, password)

.then((user) => {

*// аутентификация успешна!*

})

.catch((err) => {

res

.status(401)

.send({ message: err.message });

});

};

Но вводить почту и пароль при каждом переходе на сайт неудобно. Лучше, когда браузер запоминает, что вход на сайт уже выполнен. Так пользователю не нужно вводить данные повторно — как только он откроет страницу, сразу оказывается залогинен.

В теме про «Реакт» вы уже выполняли сохранение токена. Теперь научимся создавать токен на сервере. А в конце темы поговорим про верификацию токена.

## Как создавать токен. пакет jsonwebtoken метод jwt.sign

## Для создания токенов воспользуемся пакетом jsonwebtoken. Его нужно установить $ npm install jsonwebtoken и импортировать в проект:

Скопировать кодJAVASCRIPT

*// controllers/users.js*

const jwt = require('jsonwebtoken'); *// импортируем модуль jsonwebtoken*

Затем вызовем метод jwt.sign, чтобы создать токен:

Скопировать кодJAVASCRIPT

*// controllers/users.js*

const jwt = require('jsonwebtoken'); *//*

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

const { email, password } = req.body;

return User.findUserByCredentials(email, password)

.then((user) => {

*// создадим токен*

const token = jwt.sign({ \_id: user.\_id }, 'some-secret-key');

*// вернём токен*

res.send({ token });

})

.catch((err) => {

res

.status(401)

.send({ message: err.message });

});

};

Методу sign мы передали два аргумента: пейлоуд токена и секретный ключ подписи:

Скопировать кодJAVASCRIPT

const token = jwt.sign({ \_id: user.\_id }, 'some-secret-key');

Пейлоуд токена — зашифрованный в строку объект пользователя. Для шифрования в плейлоуд можно передать сколько угодно информации. Но ни к чему занимать трафик и отправлять лишние данные. Мы рекомендуем шифровать только самое необходимое — идентификатор. Его достаточно, чтобы однозначно определить пользователя.

У метода sign есть третий необязательный параметр — объект опций. Полный список опций описан в документации модуля jsonwebtoken. Нас интересует одна — expiresIn. Это время, в течение которого токен остаётся действительным. Можно передать число, тогда метод sign сочтёт его за количество секунд:

Скопировать кодJAVASCRIPT

const token = jwt.sign(

{ \_id: user.\_id },

'some-secret-key',

{ expiresIn: 3600 } *// токен будет просрочен через час после создания*

);

Можно передать строку, которая состоит из числа и буквы. Буква означает единицы измерения — миллисекунды, минуты, часы или дни:

Скопировать кодJAVASCRIPT

expiresIn: '120ms' *// 120 миллисекунд*

expiresIn: '15m' *// 15 минут*

expiresIn: '2h' *// 2 часа*

expiresIn: '7d' *// 7 дней*

Если опцию expiresIn не передать вовсе, токен никогда не будет просрочен.

# Защита роутов авторизацией

В предыдущих уроках вы разобрались с созданием и отправкой токена пользователю. Но это только половина задачи. Чтобы действительно облегчить жизнь пользователю и избавить его от необходимости постоянно авторизовываться в приложении, нужно научиться извлекать и проверять токен, а также защищать роуты от неавторизованных пользователей.

## Достаём токен из заголовка метод replace

Авторизация в приложении работает как мидлвэр. Если предоставлен верный токен, запрос проходит на дальнейшую обработку. Иначе запрос переходит контроллеру, который возвращает клиенту сообщение об ошибке.

Создадим для мидлвэров папку middlewares и в ней файл auth.js — тут мы напишем код для авторизации запроса:

Скопировать кодJAVASCRIPT

*// middlewares/auth.js*

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

*// тут будет вся авторизация*

};

Достанем из заголовка токен. Сначала обработаем ошибку — случай, когда токена нет в заголовке:

Скопировать кодJAVASCRIPT

*// middlewares/auth.js*

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

*// достаём авторизационный заголовок*

const { authorization } = req.headers;

*// убеждаемся, что он есть или начинается с Bearer*

if (!authorization || !authorization.startsWith('Bearer ')) {

return res

.status(401)

.send({ message: 'Необходима авторизация' });

}

};

Если токен на месте, извлечём его. Для этого вызовем метод replace, чтобы выкинуть из заголовка приставку 'Bearer ':

Скопировать кодJAVASCRIPT

*// middlewares/auth.js*

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

const { authorization } = req.headers;

if (!authorization || !authorization.startsWith('Bearer ')) {

return res

.status(401)

.send({ message: 'Необходима авторизация' });

}

*// извлечём токен*

const token = authorization.replace('Bearer ', '');

};

Таким образом, в переменную token запишется только JWT.

## Верифицируем токен метод verify модуля jsonwebtoken

После извлечения токена из запроса нужно убедиться, что пользователь прислал именно тот токен, который был выдан ему ранее. Такую проверку осуществляет метод verify модуля jsonwebtoken. Метод принимает на вход два параметра — токен и секретный ключ, которым этот токен был подписан:

Скопировать кодJAVASCRIPT

*// middlewares/auth.js*

const jwt = require('jsonwebtoken');

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

const { authorization } = req.headers;

if (!authorization || !authorization.startsWith('Bearer ')) {

return res

.status(401)

.send({ message: 'Необходима авторизация' });

}

const token = authorization.replace('Bearer ', '');

*// верифицируем токен*

const payload = jwt.verify(token, 'some-secret-key');

};

Метод jwt.verify вернёт пейлоуд токена, если тот прошёл проверку. Если же с токеном что-то не так, вернётся ошибка. Чтобы её обработать, нужно обернуть метод jwt.verify в try...catch:

Скопировать кодJAVASCRIPT

*// middlewares/auth.js*

const jwt = require('jsonwebtoken');

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

const { authorization } = req.headers;

if (!authorization || !authorization.startsWith('Bearer ')) {

return res

.status(401)

.send({ message: 'Необходима авторизация' });

}

const token = authorization.replace('Bearer ', '');

let payload;

try {

*// попытаемся верифицировать токен*

payload = jwt.verify(token, 'some-secret-key');

} catch (err) {

*// отправим ошибку, если не получилось*

return res

.status(401)

.send({ message: 'Необходима авторизация' });

}

};

Поскольку у let и const блочная область видимости, мы не можем объявить переменную payload внутри блока try — так она будет не видна снаружи. Поэтому мы объявляем её до try ключевым словом let, а записываем значение позже — в фигурных скобках блока try.

## Записываем пейлоуд в запрос

Токен верифицирован, пейлоуд извлечён. Теперь нужно записать пейлоуд в запрос — свойство req.user. Так следующий мидлвэр сможет определить, кем этот запрос был выполнен:

Скопировать кодJAVASCRIPT

*// middlewares/auth.js*

const jwt = require('jsonwebtoken');

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

const { authorization } = req.headers;

if (!authorization || !authorization.startsWith('Bearer ')) {

return res

.status(401)

.send({ message: 'Необходима авторизация' });

}

const token = authorization.replace('Bearer ', '');

let payload;

try {

payload = jwt.verify(token, 'some-secret-key');

} catch (err) {

return res

.status(401)

.send({ message: 'Необходима авторизация' });

}

req.user = payload; *// записываем пейлоуд в объект запроса*

next(); *// пропускаем запрос дальше*

};

## Защищаем роуты

Авторизационный мидлвэр готов — применим его. Один из вариантов: добавить мидлвэр всему приложению:

Скопировать кодJAVASCRIPT

*// app.js*

const express = require('express');

const bodyParser = require('body-parser');

const mongoose = require('mongoose');

const { createUser, login } = require('./controllers/auth');

const auth = require('./middlewares/auth');

const app = express();

*// роуты, не требующие авторизации,*

*// например, регистрация и логин*

app.post('/signup', createUser);

app.post('/signin', login);

*// авторизация*

app.use(auth);

*// роуты, которым авторизация нужна*

app.use('/cards', require('./routes/cards'));

Другой вариант — добавить мидлвэр отдельному маршруту. Тогда его следует передать вторым аргументом обработчику запроса:

Скопировать кодJAVASCRIPT

*// сначала вызовется auth, а затем,*

*// если авторизация успешна, createCard*

app.post('/cards', auth, createCard);

## Достаём объект пользователя

При успешной авторизации в объекте запроса появится свойство user, в которое запишется пейлоуд токена. Его можно использовать в обработчиках:

Скопировать кодJAVASCRIPT

*// controllers/cards.js*

module.exports.createCard = (req, res) => Card.create({

name: req.body.name,

link: req.body.link,

owner: req.user.\_id *// используем req.user*

});

# Заключение

Вы изучили сложную и важную тему. И теперь можете (наконец-то) сделать собственный стартап с блэкджеком и авторизацией. В проектной работе вы закрепите полученные навыки и свяжете фронтенд часть приложения с авторизацией на стороне сервера.

До этого вы умели получать, отправлять и хранить токены на стороне клиента. Теперь вы знаете, как создавать и проверять токены на стороне сервера.

Слово «соль» приобрело новое значение — это не только необходимый ингредиент для очень многих блюд, но и строка, которая добавляется к паролю перед хешированием. Другими словами, теперь вы знаете, как безопасно хранить данные в базе.

С этими знаниями пора двигаться к заключительным темам программы.