**Централизованная обработка ошибок**

До этого момента мы уделяли мало внимания ошибкам, которые возвращало наше API, и отправляли клиенту ошибку прямо там, где она возникала:

Скопировать кодJAVASCRIPT

try {

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

} catch (err) {

*// возникла ошибка,*

*// тут же отправим её, выставив статус*

return res

.status(401)

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

}

Такой способ приводит к тому, что код обработки ошибок дублируется в разных местах. Если мы решим поменять формат возвращаемой ошибки, например, добавить в JSON ещё какое-то поле, нам придётся исправлять код несколько раз.

Чтобы избежать этого, принято обрабатывать ошибки централизованно. Для этого нужно добавить приложению мидлвэр такого вида:

Скопировать кодJAVASCRIPT

app.use((err, req, res, next) => {

*// это обработчик ошибки*

});

Он отличается от обработчика запроса тем, что принимает не 3, а 4 параметра. Первый дополнительный параметр — и есть ошибка. Такой обработчик ошибки добавляют в последнюю очередь, после всех app.use. Обычно это делают где-то в конце файла app.js:

Скопировать кодJAVASCRIPT

*// app.js*

*// весь код app.js*

*// здесь обрабатываем все ошибки*

app.use((err, req, res, next) => {

res.status(500).send({ message: 'На сервере произошла ошибка' });

});

app.listen(PORT);

Но как перейти в этот обработчик ошибки при её возникновении?

**Вызов next с аргументом**

Чтобы передать обработку запроса дальше, мы привыкли вызывать next без аргумента. Если next при вызове передать аргумент, это произведёт совсем иной эффект — запрос перейдёт в обработчик ошибки:

Скопировать кодJAVASCRIPT

*// вызов next с любым аргументом*

*// передаст запрос в обработчик ошибки*

next('Аргумент');

Несмотря на то, что next можно передать любой аргумент, лучше передавать ему экземпляр ошибки:

Скопировать кодJAVASCRIPT

*// вызываем next с аргументом-ошибкой*

next(new Error('Ошибка авторизации'));

В обработчике можно использовать сообщение, переданное экземпляру ошибки при создании:

Скопировать кодJAVASCRIPT

app.use((err, req, res, next) => {

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

});

*// { "message": "Ошибка авторизации" }*

С сообщением понятно, но как в обработчике ошибки выставить ей правильный статус?

**Выставляем ошибке статус**

Самый очевидный способ — записать в экземпляр ошибки дополнительное свойство и вызвать next:

Скопировать кодJAVASCRIPT

try {

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

} catch (e) {

const err = new Error('Необходима авторизация');

err.statusCode = 401;

next(err);

}

Мы используем метод получше — сделаем свои конструкторы ошибок. Самые частые ошибки нашего приложения это:

* 400, когда с запросом что-то не так;
* 401, когда что-то не так при аутентификации или авторизации;
* 404, например, когда мы не нашли ресурс по переданному \_id;

Сделаем свои конструкторы для этих ошибок.

**Собственные конструкторы ошибок**

Для ошибок, которые часто повторяются, удобно использовать свои конструкторы. Простейший конструктор 404 ошибки будет выглядеть так:

Скопировать кодJAVASCRIPT

*// errors/not-found-err.js*

class NotFoundError extends Error {

constructor(message) {

super(message);

this.statusCode = 404;

}

}

module.exports = NotFoundError;

Всё, что делает этот конструктор, — наследует от стандартной ошибки и выставляет свойство statusCode. После создания конструктора NotFoundError, его можно импортировать в другие места кода и использовать вместе с инструкцией throw:

Скопировать кодJAVASCRIPT

const NotFoundError = require('./errors/not-found-err');

module.exports.getProfile = (req, res, next) => User

.findOne({ \_id: req.params.userId })

.then((user) => {

if (!user) {

*// если такого пользователя нет,*

*// сгенерируем исключение*

throw new NotFoundError('Нет пользователя с таким id');

}

res.send(user);

})

*// ...*

Инструкция throw генерирует исключение и обработка кода переходит в следующий блок catch, поэтому не забудьте его добавить. Это можно сделать изящно:

Скопировать кодJAVASCRIPT

const NotFoundError = require('./errors/not-found-err');

module.exports.getProfile = (req, res, next) => User

.findOne({ \_id: req.params.userId })

.then((user) => {

if (!user) {

throw new NotFoundError('Нет пользователя с таким id');

}

res.send(user);

})

.catch(next); *// добавили catch*

Такая запись catch эквивалентна следующей:

Скопировать кодJAVASCRIPT

.catch(err => next(err));

Получается, что next будет вызван с аргументом-ошибкой и запрос перейдёт в обработчик ошибки, но уже со статусом и сообщением:

Скопировать кодJAVASCRIPT

app.use((err, req, res, next) => {

res.status(err.statusCode).send({ message: err.message });

});

**Три важных правила централизованной обработки ошибок**

**1. Всегда завершайте цепочки промисов блоком catch.**

Передавайте в catch функцию next и добавляйте обработчик ошибки где-то в конце app.js. Если цепочка промисов не завершена блоком catch, это приведёт к так называемому Unhandled promise rejection (англ. «необработанное отклонение промиса»). В будущих версиях Node.js — Node завершит свою работу и приложение упадёт.

**2. Не используйте throw в завершающих блоках catch.**

Throw переводит обработку в следующий блок catch. Если использовать throw в последнем блоке catch, ему будет некуда переходить и возникнет Unhandled promise rejection.

**3. Если в обработчик пришла ошибка без статуса, возвращайте ошибку сервера.**

Мы создали свои конструкторы ошибок. Теперь, когда нужно вернуть ошибку клиенту, мы конструируем экземпляр нужной ошибки и генерируем исключение с помощью throw. Throw переводит обработку в блок catch, где его поджидает next. Кажется, всё в порядке, но есть один момент: обработка может перейти в блок catch не по причине того, что мы использовали throw, а по какой-то другой. Например, исключение может возникнуть при обращении в базу данных или наш код может просто-напросто упасть. Если ошибка сгенерирована не нами, у неё не будет свойства statusCode:

Скопировать кодJAVASCRIPT

app.use((err, req, res, next) => {

console.log(err.statusCode); *// undefined*

});

В этом случае будем считать произошедшее ошибкой сервера, возвращать 500 статус и стандартное сообщение:

Скопировать кодJAVASCRIPT

app.use((err, req, res, next) => {

*// если у ошибки нет статуса, выставляем 500*

const { statusCode = 500, message } = err;

res

.status(statusCode)

.send({

*// проверяем статус и выставляем сообщение в зависимости от него*

message: statusCode === 500

? 'На сервере произошла ошибка'

: message

});

});

**Валидация приходящих на сервер данных**

Мы уже имплементировали валидацию, определив схему модели mongoose:

Скопировать кодJAVASCRIPT

const userSchema = new Schema({

email: {

type: String,

required: true,

unique: true,

...

});

Если оставить валидацию только на этом уровне, приложение будет легко атаковать, например, DDoS-атакой. Ведь чтобы такая валидация запустилась, должен запуститься и код контроллера. А он может непреднамеренно упасть, если клиент пришлёт не то тело запроса, которое мы от него ожидаем. Также в контроллере могут быть CPU-интенсивные операции, например хеширование пароля. Таким образом, злоумышленник может посылать много запросов, которые будут нагружать процессор или, хуже того, ронять приложение.

Поэтому принято валидировать входящий запрос и, если клиент присылает не то, что мы от него ждём, контроллер просто не запускается, а клиент получает ошибку. Для этого запрос описывают схемой.

**Joi и celebrate**

**Joi** — популярная Node.js библиотека для валидации данных. Она позволяет описывать данные в интуитивно понятном виде:

Скопировать кодJAVASCRIPT

{

body: Joi.object().keys({

email: Joi.string().required().email(),

password: Joi.string().required().min(8),

name: Joi.string().required().min(2).max(30),

age: Joi.number().integer().required().min(18),

about: Joi.string().min(2).max(30),

})

}

Такое описание говорит, что body должно быть объектом с ключами:

* email — строка, обязательное поле, должно соответствовать паттерну почты;
* password — обязательная строка, минимум 8 символов;
* name — обязательная строка от 2 до 30 символов;
* age — число, целое, обязательное, минимально возможное значение — 18;
* about — строка от 2 до 30 символов.

Чтобы подключить валидацию Joi в качестве мидлвэр, мы будем использовать библиотеку **celebrate**. Перед использованием её нужно установить в проект “npm i **celebrate”**, после чего импортировать и подключить как мидлвэр к роуту:

Скопировать кодJAVASCRIPT

const { celebrate, Joi } = require('celebrate');

router.post('/posts', celebrate({

body: Joi.object().keys({

title: Joi.string().required().min(2).max(30),

text: Joi.string().required().min(2),

}),

}), createPost);

Такой мидлвэр валидирует тело запроса. В нём должно быть два поля — title и text. Если тело запроса не пройдёт валидацию, контроллер createPost вообще не запустится.

Кроме тела запроса, celebrate позволяет валидировать заголовки, параметры или req.query:

Скопировать кодJAVASCRIPT

const { celebrate, Joi } = require('celebrate');

router.delete('/:postId', celebrate({

*// валидируем параметры*

params: Joi.object().keys({

postId: Joi.string().alphanum().length(24),

}),

headers: Joi.object().keys({

*// валидируем заголовки*

}),

query: Joi.object().keys({

*// валидируем query*

}),

}), deletePost);

По умолчанию Joi не допускает полей, которые не перечислены в объекте валидации. Чтобы изменить это поведение, нужно после вызова метода keys вызвать метод unknown с аргументом true:

Скопировать кодJAVASCRIPT

const { celebrate, Joi } = require('celebrate');

router.delete('/:postId', celebrate({

headers: Joi.object().keys({

*// валидируем заголовки*

}).unknown(true),

}), deletePost);

**Ошибки**

Если запрос не проходит описанную валидацию, celebrate передаст его дальше, но не в контроллер, а в обработчик ошибки. Чтобы отправить клиенту ошибку, в celebrate есть специальный мидлвэр — errors:

Скопировать кодJAVASCRIPT

*// app.js*

const { errors } = require('celebrate');

*// ...*

*// обработчики ошибок*

app.use(errors()); *// обработчик ошибок celebrate*

*// наш централизованный обработчик*

app.use((err, req, res, next) => {

*// ...*

});

errors() будет обрабатывать только ошибки, которые сгенерировал celebrate. Все остальные ошибки он передаст дальше, где их перехватит централизованный обработчик.

Статус ошибки, которую возвращает celebrate, — 400, а тело ответа имеет такой вид:

Скопировать кодJAVASCRIPT

{

"statusCode": 400,

"error": "Bad Request",

"message": "child \"name\" fails because [\"name\" is required]",

"validation": {

"source": "body",

"keys": [

"name"

]

}

}

Поле message позволяет клиенту понять, что не так с его запросом. В данном случае celebrate сообщает, что в теле запроса отсутствует обязательное поле name.

**Ссылки**

Документация Joi: <https://joi.dev/api/>.

Документация celebrate: <https://github.com/arb/celebrate>.

# Заключение

В этой теме вы познакомились с библиотекой Joi и узнали, как подключить её в качестве мидлвэр к роуту. С помощью этой библиотеки вы сможете защитить контроллеры от возможных атак и ненужных запусков. Это поможет уменьшить нагрузку на приложение.

Joi умеет отправлять и обрабатывать ошибки, но их все равно нужно перехватывать новым централизованным обработчиком. С его помощью вы научились в одном месте описывать всю логику обработки ошибок, и кодовая база стала меньше и понятнее. Теперь вы сумеете создать конструкторы для ошибок, которые часто повторяются, и код не придётся писать повторно.

В этой теме вы завершили изучение фреймворка express. Эти знания вам предстоит применить в проектной работе спринта. В следующей теме расскажем о регулярных выражениях. А в следующем спринте вы уже поработаете с собственным сервером!