**Express. Введение**

В предыдущей теме вы разобрались с Node.js: научились поднимать сервер и работать с файловой системой. Нода даёт широкие возможности, поэтому может показаться, что её достаточно для решения любых задач и другие инструменты не нужны.

Но это не так. Программирование сервера во многом сводится к решению типовых задач: настроить авторизацию или реализовать логирование запросов (запись в базу данных). Поскольку разработка — дело творческое, программисты решают типовые задачи по-разному.

Такой подход может привести к проблеме с поддержкой. Например, разработчик написал код для авторизации пользователей, а потом ушёл в другую компанию. Вы приходите на его место и не понимаете, какая логика заложена в коде. Придётся либо всё переделывать, либо очень долго вникать в чужую работу, а это — головная боль для вас и уйма пущенных на ветер денег — для работодателя.

К тому же разрабатывать сервер придётся намного дольше: все детали каждый раз нужно прописывать вручную. Программистам, может, и нравится этим заниматься, но вот компаниям точно не нравится за это платить.

В таких ситуациях помогают фреймворки — наборы стандартных решений. С ними вы можете перейти из одного проекта в другой и сразу «въехать» в логику, заложенную в программу. Кроме того, выстроить логику также становится проще и быстрее — она уже расписана и реализована, а вам нужно лишь правильно воспользоваться синтаксисом.

Но есть две крайности. Одна — писать весь код на чистой ноде. Вторая — бездумно обращаться к методам фреймворка, не понимая, как они устроены. Такой подход ещё называют monkey coding («мартышкино программирование»). Лучше не впадать ни в одну из крайностей: разработчику следует пользоваться современными инструментами и понимать, как именно они работают и что за задачи решают.

В этой теме мы научим вас пользоваться самым популярным фреймворком для разработки серверного кода — Express. С помощью него вы сможете быстрее разворачивать сервер и лучше понимать код других разработчиков — на «экспрессе» написаны тысячи серверов.

Приступим!

# Приложение на экспресс

Начнём с практики. В этом уроке развернём сервер с помощью экспресса.

## Устанавливаем экспресс

Экспресс — пакет ноды, поэтому его устанавливают из NPM:

Скопировать кодBASH

*# Инициализируйте package.json, чтобы нода могла устанавливать зависимости*

npm init -y *# флаг -y ответит «Да» на все вопросы по настройке package.json*

*# Установите экспресс*

npm install express

## Настраиваем hot reload

Когда вы работали с проектами на «Реакте», то пользовались так называемым dev-режимом сборки: запускали команду npm start в Create React App. Вы могли заметить, что в этом режиме сайт автоматически обновлялся при сохранении любого файла. Такое поведение называют "hot reload" (от англ. «горячая перезагрузка»). В Create React App это делал Webpack, встроенный в сборку Create React App. В проектах на Express мы не используем Webpack — hot reload будет выполнять специальный пакет nodemon.

Hot reaload нужен только для разработки, потому установите nodemon как dev-зависимость:

Скопировать кодBASH

*# флаг -D установит nodemon как dev-зависимость*

npm install nodemon -D

Осталось, чтобы nodemon и программа запускались вместе. Откройте package.json и отредактируйте раздел "scripts":

Скопировать кодJSON

{

...

"scripts": {

"start": "nodemon index.js"

},

...

}

## Устанавливаем точку входа

Создадим файл index.js и опишем в нём приложение. Для этого подключите express, а затем создайте приложение методом express:

Скопировать кодJAVASCRIPT

const express = require('express');

const app = express();

Готово! Вы только что создали express-приложение.

## Проверяем, всё ли работает

Настроим порт, который должен слушать приложение. Возьмём его из переменной окружения:

Скопировать кодJAVASCRIPT

const express = require('express');

*// Слушаем 3000 порт*

const { PORT = 3000 } = process.env;

const app = express();

app.listen(PORT, () => {

*// Если всё работает, консоль покажет, какой порт приложение слушает*

console.log(`App listening on port ${PORT}`)

})

Запустим приложение:

Скопировать кодBASH

node index.js

*# В терминале окажется сообщение "App listening on Port 3000"*

Перейдём в браузере по адресу http://localhost:3000. Мы видим сообщение "Cannot GET /". Это означает, что экспресс запущен и работает, просто пока ему нечего нам отдать.

В следующем уроке исправим это — разберёмся, как настраивать роутинг.

# Как общаются клиент и сервер? Настройка роутинга

В предыдущем уроке вы создали сервер фреймворком express. Это круто, только пока он не работает: при запуске всплывает сообщение "Cannot GET /".

В этом уроке разберём настройки роутинга. Так сервер сможет отдавать веб-страницы и принимать данные от пользователя. В общем, делаем всё то же, что в первой теме, но вместо чистой ноды воспользуемся экспрессом.

Начнём с обработки GET-запроса, а запросы других типов рассмотрим чуть позже.

## Роутинг GET-запроса

Для обработки GET-запроса в экспрессе есть метод get. Он принимает на вход два аргумента:

* строку с запросом;
* колбэк, предписывающий, что нужно делать, если такой запрос пришёл на сервер.

Скопировать кодJAVASCRIPT

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

*// логика обработки запроса*

});

Параметры колбэка — объекты запроса и ответа.

## Объект ответа

У объекта ответа есть метод send — он отправляет ответ. Параметром ему передают информацию, которую нужно отправить пользователю, например, веб-страницу.

Скопировать кодJAVASCRIPT

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

res.send(

`<html>

<body>

<p>Ответ на сигнал из далёкого космоса</p>

</body>

</html>`

);

});

Метод send может принимать разные данные в качестве аргумента:

Скопировать кодJAVASCRIPT

res.send('немного текста');

res.send('<p>немного html</p>');

res.send({ some: 'json' });

Кроме отправки ответа метод автоматически выставит заголовок Content-Type в соответствии с аргументом, который ему передали:

Скопировать кодJAVASCRIPT

res.send('немного текста'); *// Content-Type: text/plain*

res.send('<p>немного html</p>'); *// Content-Type: text/plain*

res.send({ some: 'json' }); *// Content-Type: application/json*

По умолчанию метод send отправляет ответ со статусом 200. Его можно изменить методом status:

Скопировать кодJAVASCRIPT

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

res.status(404);

res.send('<h1>Страница не найдена</h1>');

});

Метод status устроен так, что его и send можно использовать по цепочке. Обычно так и делают:

Скопировать кодJAVASCRIPT

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

res.status(404).send('<h1>Страница не найдена</h1>');

});

Со статус-кодами вы познакомились [в этой теме](https://praktikum.yandex.ru/learn/web/courses/4da123a7-d4aa-4bc2-b299-b48f921da09c/sprints/2798/topics/5dc7307b-8486-40a5-b418-0888ac73337e/lessons/c0b00d3c-ac98-439a-b724-45efecfc896a/). Важно присваивать разным ошибкам соответствующие статус-коды. Это нужно для правильной работы любого приложения.

У объекта ответа есть и другие методы. Будем знакомиться с ними постепенно, но если что — [вот документация](http://expressjs.com/ru/api.html#res).

## Объект запроса

Запрос содержит информацию о том, что нужно пользователю. Она может быть организована по-разному, например, в GET-запросе данные пользователя «упакованы» в строку, которая хранится в свойстве query объекта запроса.

Скопировать кодJAVASCRIPT

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

res.send(req.query);

});

В примере мы можем передавать любые данные в формате query string, сервер обработает их и пришлёт JSON-файл в виде строки.

## Обработка методов в express

В экспрессе есть методы для обработки любого запроса. Если хотите освежить их в памяти — [просмотрите этот урок](https://praktikum.yandex.ru/learn/web/courses/4da123a7-d4aa-4bc2-b299-b48f921da09c/sprints/2798/topics/5dc7307b-8486-40a5-b418-0888ac73337e/lessons/3164b264-9a7e-4100-838c-62480f73fd2d/). Они называются так же, как соответствующие HTTP-методы, и могут быть вызваны на приложении, где им в качестве параметров будут переданы эндпоинты и функции-обработчики:

Скопировать кодJAVASCRIPT

app.get('/books', getBooks);

app.post('/books', createBook);

app.put('/books', replaceBook);

app.patch('/books', updateBookInfo);

app.delete('/books', deleteBook);

Пример.

Пользователю может быть нужно одно животное. В этом случае отдавать ответ целиком не нужно. Перепишите роутинг так, чтобы пользователь мог получать свойства объекта animals.

GET-запрос должен содержать две пары значений: животное и породу. Вот примеры запросов и ответов на них:

*app.listen(PORT, () => {*

*console.log('Ссылка на сервер:');*

*console.log(BASE\_PATH);*

*});*

*app.get('/animals', (req, res) => {*

*res.send(animals[req.query.animal][req.query.type]);*

*});*

# Все говорят о роутинге. Объясняем, что это

В предыдущем уроке вы настроили роутинг на экспрессе. Сейчас для каждого запроса есть свой обработчик: когда клиент запрашивает страницу, срабатывает соответствующий обработчик.

Такая логика подходит не всегда. Невозможно написать по обработчику для каждой страницы пользователя в соцсети или к каждому плейлисту в музыкальном сервисе. Вместо этого используют динамические роуты.

## Что такое динамические роуты?

Раньше вы писали статические роуты, они обрабатывали запросы на конкретные адреса: /, /animals, /search.

Динамический роут — это обработчики шаблона адреса. То есть сразу всех адресов, подходящих под определённое правило.

Чтобы создать динамический роут, в URL укажите двоеточие и имя свойства, которое может меняться:

Скопировать кодJAVASCRIPT

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

*// Этот код обработает все запросы, чей путь начинается с /users/111111*

});

Обратите внимание: значение переменной в адресе не может быть пустым. Вот такой запрос не будет обработан:

Скопировать кодBASH

http://localhost:3000/users

## Параметры роута

Обработать запрос мало — нужно получить его параметры, чтобы отдавать пользователю нужные данные. В экспрессе это просто: все параметры запроса становятся свойствами JSON-объекта req.params:

Скопировать кодJAVASCRIPT

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

res.send(req.params);

*/\* Если отправить запрос на адрес "http://localhost:3000/users/123",*

*внутри объекта req.params окажется JSON-объект:*

*{ "id": "123" } \*/*

});

Формат JSON оказывается тут как никогда кстати. Особенно при обработке адресов со множественной вложенностью:

Скопировать кодJAVASCRIPT

app.get('/users/:id/albums/:album/:photo', (req, res) => {

const { id, album, photo } = req.params;

*/\* При обращению к адресу "http://localhost:3000/users/123/albums/333/2"*

*параметры запроса будут записаны в таком виде:*

*{"id":"123","album":"333","photo":"2"}*

*Мы записали их в переменные id, album и photo \*/*

res.send(`Мы на странице пользователя с id ${id}, смотрим альбом №${album} и фотографию №${photo}`);

});

## Как организовать роутинг

Мы уже не раз говорили о преимуществах модульного кода. Логику роутинга тоже следует разнести по разным файлам. Пусть один отвечает за основную часть сайта, второй — за админку, третий — за мобильное приложение.

Разберёмся, как разбить этот код на модули:

Скопировать кодJAVASCRIPT

*// Здесь функциональность точки входа*

const express = require('express');

const { PORT = 3000 } = process.env;

const app = express();

*// Здесь данные*

const users = [

{ name: 'Мария', age: 22 },

{ name: 'Виктор', age: 30 },

{ name: 'Анастасия', age: 48 },

{ name: 'Алексей', age: 51 }

];

*// Здесь роутинг*

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

if (!users[req.params.id]) {

res.send(`Такого пользователя не существует`);

*// не забудем выйти из функции*

return;

}

const { name, age } = users[req.params.id];

res.send(`Пользователь ${name}, ${age} лет`);

});

app.listen(PORT, () => {

console.log(`App listening on port ${PORT}`);

});

### Первым делом **перенесём данные в отдельный файл** db.js:

Скопировать кодJAVASCRIPT

*// db.js*

module.exports = {

users: [

{ name: 'Мария', age: 22 },

{ name: 'Виктор', age: 30 },

{ name: 'Анастасия', age: 48 },

{ name: 'Алексей', age: 51 }

]

};

Даже четыре лишние строчки засоряют код. А пользователей на сайте обычно больше четырёх (даже если это сайт администрации малообитаемого острова в Тихом океане). Так что всегда храните данные отдельно.

### Настроим роутер. Код роутинга тоже перенесём в отдельный файл.

Для этого код придётся дописать. Логика ответа на запросы описана в обработчиках get. Мы вызывали get как метод самого приложения — переменной app. В отдельном модуле с роутингом этой переменной нет. И создать её не получится — может быть только одно приложение. Поэтому в экспрессе есть метод Router. Он создаёт объект, на который мы и повесим обработчики:

Скопировать кодJAVASCRIPT

*// router.js*

const router = require('express').Router(); *// создали роутер*

const { users } = require('./db.js'); *// данные нужны для роутинга, поэтому импортируем их*

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

if (!users[req.params.id]) {

res.send(`Такого пользователя не существует`);

return;

}

const { name, age } = users[req.params.id];

res.send(`Пользователь ${name}, ${age} лет`);

});

module.exports = router; *// экспортировали роутер*

К слову, как и в предыдущем уроке, не только GET-метод может быть добавлен к объекту Router:

Скопировать кодJAVASCRIPT

router.get('/books', getBooks);

router.post('/books', createBook);

router.put('/books/:id', replaceBook);

router.patch('/books/:id', updateBookInfo);

router.delete('/books/:id', deleteBook);

Настроим точку входа. Для этого импортируем роутер в index.js и запустим его методом use. Он принимает на вход два параметра:

* начало URL — роутер будет запускаться только для запросов, начинающихся с этой строки;
* сам роутер:

Скопировать кодJAVASCRIPT

*// index.js*

const express = require('express');

const router = require('./router.js'); *// импортируем роутер*

const { PORT = 3000 } = process.env;

const app = express();

app.use('/', router); *// запускаем*

app.listen(PORT, () => {

console.log(`App listening on port ${PORT}`);

});

Первый параметр метода use позволяет сделать код модульным. Вы можете написать разные роутеры для разной функциональности:

Скопировать кодJAVASCRIPT

const express = require('express');

const router = require('./router.js');

const api = require('./api.js');

const backoffice = require('./backoffice.js');

const { PORT = 3000 } = process.env;

const app = express();

*// Для разных запросов разные роутеры.*

*// Выглядит просто отлично!*

app.use('/', router);

app.use('/api', api);

app.use('/admin', backoffice);

app.listen(PORT, () => {

console.log(`App listening on port ${PORT}`);

});

На этом всё — роутинг готов. Теперь наше приложение можно гордо называть сервером, ведь оно выполняет его главную функцию: обработку запросов.

В следующем уроке разберём, чем занят сервер между получением запроса и выдачей ответа.

# Промежуточная обработка. Как улучшить код посредством middlewares

В заданиях предыдущего урока вы описали логику роутинга:

Скопировать кодJAVASCRIPT

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

*// логика обработки начиналась*

const { id } = req.params;

if (!users[id]) {

res.send({ error: 'Такого пользователя нет' });

return;

}

*// логика обработки заканчивалась, отправлялся ответ*

res.send(users[id]);

});

Это отличный код для поставленной задачи. Но часто придётся реализовывать что-то более глобальное, например, логирование или аутентификацию пользователя. Если прописывать эту логику внутри роутеров, кода станет так много, что в нём будет сложно разобраться.

Эту проблему решают мидлвэры (middleware) — функции промежуточной обработки. Они позволяют описать логику обработки запроса в отдельном модуле, что сокращает код роутера.

В мидлвэр удобно выносить и типовые задачи. Например, для любого запроса сервер проверяет наличие записи в базе данных. Благодаря мидлвэру эту логику не нужно прописывать в каждом роутере.

### Создаём мидлвэр

Разобьём код роутера на отдельные функции: одна служит для проверки, есть ли пользователь в нашей базе, другая — для отправки ответа с нужным пользователем.

Начнём с ответа:

Скопировать кодJAVASCRIPT

*// Вынесём отправку ответа в отдельную функцию*

const sendUser = (req, res) => {

const { name, age } = users[req.params.id];

res.send(`Пользователь ${name}, ${age} лет`);

};

Переходим ко второй функции — мидлвэру. Она проверяет, существует ли пользователь:

* если нет, отправляет в ответе ошибку;
* если да, вызывает функцию отправки ответа с конкретным пользователем — уже написанную sendUser.

Переведём это на джаваскриптовый:

Скопировать кодJAVASCRIPT

*// Проверим, существует ли пользователь:*

const doesUserExist = (req, res, next) => {

if (!users[req.params.id]) {

res.send(`Такого пользователя не существует`);

return;

}

next(); *// вызываем next*

};

const sendUser = (req, res, next) => {

res.send(users[req.params.id]);

};

Если пользователь найден, вызовем функцию, переданную третьим аргументом. Осталось написать обработчик запроса:

Скопировать кодJAVASCRIPT

router.get('/users/:id', doesUserExist);

router.get('/users/:id', sendUser);

Тут и начинается магия экспресса. Фреймворк написан так, что если для одного запроса есть несколько обработчиков, он будет считать функцией next следующий обработчик этого же запроса.

Разберём код ещё раз. Вот функция doesUserExist:

Скопировать кодJAVASCRIPT

const doesUserExist = (req, res, next) => {

if (!users[req.params.id]) {

res.send(`Такого пользователя не существует`);

return; *// если пользователя нет, мы выйдем из функции и больше ничего происходить не будет*

}

next(); *// если движок дошёл до функции next, он будет искать следующий обработчик того же запроса*

};

Допустим, нам нужен ещё один мидлвэр, чтобы проверять уровень доступа пользователя. Мы также опишем его логику и запустим колбэк next, а затем просто вставим новый обработчик запроса между уже существующими:

Скопировать кодJAVASCRIPT

router.get('/users/:id', doesUserExist);

router.get('/users/:id', doesUserHasPermission); *// Просто добавили обработчик, не трогая код других мидлвэров*

router.get('/users/:id', sendUser);

Обратите внимание: колбэк мидлвэра принято всегда называть next. Старайтесь не отходить от этого правила, чтобы ваш код был понятен другим разработчикам.

### Подключаем мидлвэр

Для использования мидлвэр мы будем пользоваться методом use, но с одним параметром — самим мидлвэром:

Скопировать кодJAVASCRIPT

app.use(middleware);

Скопировать кодJAVASCRIPT

*// index.js*

const express = require('express');

const routes = require('./routes.js');

const { PORT = 3000 } = process.env;

const app = express();

const logger = (req, res, next) => {

console.log('Запрос залогирован!');

next();

};

app.use(logger);

app.use('/', router);

app.listen(PORT, () => {

console.log(`App listening on port ${PORT}`);

});

Пример

Реализуйте логирование. Напишите мидлвэр, который при каждом запросе к серверу выводит в консоль:

* время запроса;
* метод, которым запрос был сделан.

Код пишите в файле app.js.

const express = require('express');

const routes = require('./routes.js');

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

const app = express();

const timeLog = (req, res, next) => {

const now = new Date();

console.log(req.method, now);

next();

};

app.use(timeLog);

app.use('/', routes);

app.listen(PORT, () => {

console.log('Ссылка на сервер:');

console.log(BASE\_PATH);

});

# Продвинутые мидлвэры: парсер данных

В предыдущем уроке вы написали мидлвэр для логирования данных. Но этого не придётся делать часто — для типовых задач уже есть готовые инструменты. Они позволят сэкономить время и не изобретать велосипед. О таких готовых решениях расскажем в уроке.

## Сборка пакетов: body-parser

Как вы помните из предыдущей темы, при передаче между клиентом и сервером данные бьются на блоки, а на принимающей стороне собираются обратно. Тогда мы писали код для сборки пакетов:

Скопировать кодJAVASCRIPT

const postForm = (req, res) => {

let body = ''; *// устанавливаем в переменную body пустую строку*

req.on('data', (chunk) => {

body += chunk; *// добавляем каждый приходящий пакет в body*

});

req.on('end', () => { *// когда всё пришло*

*// можем работать с объектом из колбэк метода on*

});

};

Тяжеловато. Особенно для такой базовой задачи как приём данных.

От ненужной сложности нас защищает мидлвэр body-parser. Он самостоятельно объединяет все пакеты, так что мы можем об этом не задумываться:

Скопировать кодJAVASCRIPT

const postForm = (req, res) => {

const { body } = req;

*// затем мы можем просто обращаться к body как к объекту*

};

## Подключим к проекту

Body-parser устанавливают из NPM:

Скопировать кодBASH

npm i body-parser

Затем его нужно импортировать:

Скопировать кодJAVASCRIPT

*// app.js*

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

Все методы для работы с пакетами находятся в объекте bodyParser, остаётся только выбрать нужные: это зависит от типа данных, которые нам приходят.

Скопировать кодJAVASCRIPT

app.use(bodyParser.json()); *// для собирания JSON-формата*

app.use(bodyParser.urlencoded({ extended: true })); *// для приёма веб-страниц внутри POST-запроса*

Аргументом методу bodyParser.urlencoded мы передаём объект опций. "extended: true" означает, что данные в полученном объекте body могут быть любых типов. При значении false, в свойства body могут попасть только строки и массивы. Это уже относится к более тонкой настройке, и мы не будем подробно затрагивать эту тему, но оставим ссылку на статью в дополнительных материалах.

Если вам могут прийти данные разных типов, просто задействуйте все нужные методы, — как в примере, Body-parser сам разберётся, какие данные пришли и как их собирать.

# Отдача html и статичных файлов в Express

Мы разобрались с передачей данных в ответе на запрос. Но пользователю нужно отдавать и статичные файлы: разметку, стили, скрипты, изображения. Писать обработчики для всей этой информации неудобно и не нужно, вместо этого научимся пользоваться методом static.

По умолчанию у пользователей нет доступа к файлам на вашем сервере. Это можно изменить методом static. Он принимает адрес папки на вход и делает её доступной «снаружи»:

Скопировать кодJAVASCRIPT

app.use(express.static(\_\_dirname)); *// сделали всё, что есть на сервере, доступным пользователю*

Такой код даст доступ к корню сервера, но там могут храниться данные, которые пользователям видеть не нужно, например, ключи доступа к другим ресурсам или базы данных. Файлы, которые нужно отдавать клиентам, складывают в отдельную папку — её принято называть "public".

Скопировать кодJAVASCRIPT

app.use(express.static(path.join(\_\_dirname, 'public'))); *// теперь клиент имеет доступ только к публичным файлам*

Метод static задаёт определённую логику отправки файлов. Если обратиться к корню сервера, мы автоматически попадём в папку public и получим файл index.html. Получается, что не нужно добавлять в запрос имя файла, достаточно просто обратиться в корень — по адресу "/".

Публичная папка — отправная точка в проекте, которую мы уже указали с помощью express.static. Осталось добавить относительные пути ко всем файлам:

| **НЕТ** | **ДА** |
| --- | --- |
| /public/style.css | /style.css |

# Кеширование ответа сервера в express

Кеширование позволяет сохранить определённые данные на клиенте или сервере, чтобы не отвечать на один и тот же запрос несколько раз. Это ускоряет работу приложений и взаимодействие сервера с клиентом.

GET-запросы кешируются браузерами по умолчанию. POST-запросы, наоборот, по умолчанию не кешируются, а PUT, PATCH и DELETE не кешируются вообще.

За логику кеширования отвечают специальные заголовки GET и POST запросов, которые указываются на сервере. Об этих заголовках и расскажем.

### Кеширующие заголовки

**Expires**

Данные могут устаревать. Заголовок Expires позволяет настроить, как долго закешированные данные будут актуальны. После того, как этот период истечёт, данные будут запрошены с сервера, а не взяты из кеша. Сам кеш при этом обновится:

Скопировать код

Expires: Fri, 20 May 2016 19:20:49 IST

**Cache-Control**

В этом заголовке указывают одну из директив для настройки обращения с закешированными данными. Вот наиболее используемые:

Скопировать код

// управление кешированием

Cache-Control: only-if-cached // если ответ в кеше, взять его оттуда, не отправляя запрос

Cache-Control: no-cache // кеш сразу становится просроченным, а затем данные ответа в кеше проверяются

// управление временем жизни

Cache-Control: max-age=30000 // максимальное время в секундах, в течение которого ресурс считается актуальным

~~~~// управление обновлением кеша

Cache-Control: must-revalidate // если кеш просроченый, будет отправлен запрос на проверку данных в кеше

// другое

Cache-Control: no-store // кеш отключён

Данные от клиента к серверу идут не напрямую, а через другие машины — прокси-серверы. На этих промежуточных машинах данные тоже могут быть закешированы. Этим также можно управлять, установив заголовок Cache-Control:

Скопировать код

Cache-Control: private // данные могут быть закешированы только на клиенте

Cache-Control: public // данные могут быть закешированы на промежуточных серверах

Cache-Control: proxy-revalidate // перед использованием данных из кеша прокси-сервера, их актуальность должна быть проверена

Cache-Control: no-transform // прокси не могут применять к ресурсу никакие преобразования. Другие настройки — private, public и proxy-revalidate — таких ограничений не накладывают

В Cache-Control можно указать сразу несколько директив через запятую:

Скопировать код

Cache-Control: max-age=30000, must-revalidate

Кроме Expires и Cache-Control, есть два заголовка, которые отвечают за валидацию кеша — ETag и Last-Modified.

**ETag**

Чтобы при каждой валидации не отправлять кеш серверу целиком, был придуман механизм хеширования. Он заключается в следующем: определённый алгоритм анализирует кеш и на основе этого создаёт строку — [хеш](https://praktikum.yandex.ru/learn/web/courses/dbf98e55-0f76-444b-850c-4538708ad571/sprints/1425/topics/b4072eed-2089-45c0-9382-98ea71202341/lessons/72688d32-ec42-41de-a818-a219c59d88b2/) (о нём говорили в теме о технологии Git). Такая строка уникальна для любого набора данных. Можно хранить в кеше только эту строку и передавать её для валидации — это куда эффективнее.

В заголовок Etag можно записать хеш. Если тело ответа меняется, изменится и хеш:

Скопировать код

ETag: "abcd1234567n34jv"

**Last-Modified**

Заголовок Last-Modified хранит дату последнего изменения данных на сервере, что позволяет валидировать данные в кеше:

Скопировать код

Last-Modified: Fri, 10 May 2016 09:17:49 IST

### Кеширующие заголовки в express

Express по умолчанию работает так:

* Создаёт заголовок ответа Etag, который посылает в каждом ответе сервера. Таким образом кешируется каждый ответ.
* Значение заголовка запоминает браузер и при следующем GET-запросе отправляет Etag внутри другого заголовка — If-None-Match.
* Если значение заголовка If-None-Match совпадает с каким-то кешем на сервере, ответ придёт в статусе 304 Not Modified. И браузер возьмёт значение ответа из своего кеша.

POST-запросы по умолчанию не кешируются. Да и на практике к кешированию POST-запросов прибегают редко. Но оно возможно: достаточно вручную выставить кеширующие заголовки.

Пример

Сервер возвращает статическую HTML-страницу. Недавно в нём пофиксили баг, но пользователи продолжают сообщать о нём. Вероятно, на их машинах или промежуточных прокси-серверах закеширована старая версия HTML-файла.

Отключим на время кэширование, чтобы пользователи получили новую страницу с сервера.

В файле middlewares.js мы создали заготовку мидлвэра — функцию setNoCacheHeaders. В теле этой функции установите заголовки, которые отключат кеширование.

Заголовки ответа выставляют методом ответа header:

res.header('Content-Type', 'application/json');

Отключить кеширование ресурса можно заголовком Cache-Control*.*

const setNoCacheHeaders = (req, res, next) => {

res.header('Cache-Control', 'no-store');

next();

};

module.exports = {

setNoCacheHeaders

};

res.header('Content-Type', 'application/json');

const path = require('path');

const express = require('express');

const { setNoCacheHeaders } = require('./middlewares');

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

const app = express();

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

app.listen(PORT, () => {

console.log('Ссылка на сервер');

console.log(BASE\_PATH);

});

app.use(setNoCacheHeaders);

# CORS. Обработка ошибок

Сейчас наше приложение отлично работает на локальном сервере и по умолчанию оно готово принимать запросы с любых страниц.

В учёбе это не принципиально, но в реальных проектах — небезопасно. Чтобы это исправить, нужно задать заголовки для поддерживаемых сайтов с безопасными запросами. Подробнее мы рассматривали [CORS в предыдущем спринте](https://praktikum.yandex.ru/learn/web/courses/4da123a7-d4aa-4bc2-b299-b48f921da09c/sprints/2798/topics/5dc7307b-8486-40a5-b418-0888ac73337e/lessons/8f98665e-1efd-4a6c-baed-10f7ec7891b7/).

Чтобы сделать приложение более безопасным, в express ему добавляют мидлвэр:

Скопировать кодJAVASCRIPT

app.use(function(req, res, next) {

res.header('Access-Control-Allow-Origin', 'https://praktikum.tk');

next();

});

В этом же мидлвэре настраивают разрешённые заголовки и методы:

Скопировать кодJAVASCRIPT

app.use(function(req, res, next) {

res.header('Access-Control-Allow-Origin', 'https://praktikum.tk');

res.header('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content-Type, Accept');

res.header('Access-Control-Allow-Methods', 'GET,HEAD,PUT,PATCH,POST,DELETE');

next();

});

### Как разрешить запросы с нескольких ресурсов

В заголовок Access-Control-Allow-Origin можно записать либо один URL, либо сразу все. Иногда нужно разрешить кросс-доменные запросы с нескольких ресурсов, но не всех. Например, с локального сервера и продакшн-сайта.

У любого запроса есть заголовок Origin. Он содержит адрес, с которого идёт этот запрос. Мы можем создать массив разрешённых доменов, а затем проверять, есть ли среди них тот, что указан в заголовке Origin.

Если домен найден среди разрешённых, просто переписываем значение Origin в заголовок ответа Access-Control-Allow-Origin:

Скопировать кодJAVASCRIPT

*// Массив разешённых доменов*

const allowedCors = [

'https://praktikum.tk',

'http://praktikum.tk',

'localhost:3000'

];

app.use(function(req, res, next) {

const { origin } = req.headers; *// Записываем в переменную origin соответствующий заголовок*

if (allowedCors.includes(origin)) { *// Проверяем, что значение origin есть среди разрешённых доменов*

res.header('Access-Control-Allow-Origin', origin);

}

next();

});

Защита от кросс-доменных запросов — это хорошо. Но ещё лучше уметь этой защитой управлять.

Пример

В файле utils.js допишите функцию getRandomQuote, чтобы она возвращала случайную цитату из массива quotes.

const { quotes } = require('./data');

module.exports.getRandomQuote = () => quotes[Math.floor(Math.random() \* (quotes.length))];

Пример cors

API почти готов. Перейдите на любой сайт в новой вкладке браузера и откройте консоль. Сделайте запрос к серверу методом fetch:

Скопировать кодJAVASCRIPT

fetch(*/\* передайте ссылку на сервер, она в терминале \*/*)

.then(res => res.json())

.then(res => console.log(res));

В консоли не окажется результата. Всё из-за CORS.

Поскольку наш API должен быть открытым, кросс-доменные запросы нужно разрешить. В теории урока мы делали это вручную, выставляя нужные заголовки, а здесь воспользуемся готовым модулем — cors.

Мы уже установили его в проект, осталось импортировать его и применить. Используя модуль cors, разрешите кросс-доменные запросы на всех роутах приложения.

Чтобы выполнить это задание, разберитесь в работе модуля cors по спецификации: <https://www.npmjs.com/package/cors>.

Пользоваться модулем cors достаточно просто. Его нужно применить:

app.use(cors());

const express = require('express');

const { getRandomQuote } = require('./utils')

const cors = require('cors')

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

const app = express();

app.use(cors());

app.listen(PORT, () => {

console.log('Ссылка на сервер');

console.log(BASE\_PATH);

});

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

res.send({

quote: getRandomQuote()

})

# Заключение

Первое знакомство с Express позади. Работать с ним заметно проще, чем писать код на ноде без сторонних библиотек и фреймворков.

Вспомним, что вы изучили в этой теме.

### Роутеры

Роутеры помогают удобно организовать код. Мы можем разбить их на отдельные файлы и передавать как колбэки методам обработки запросов. В экспрессе есть встроенный роутер:

Скопировать кодJAVASCRIPT

const router = require('express').Router();

router.get('/users/:id', doesUserExist);

Если все роуты содержатся во входном файле, роутер можно не создавать, а передавать методу use самого приложения:

Скопировать кодJAVASCRIPT

app.use('/users/:id', doesUserExist);

Роутер удобно использовать, чтобы объединять логику работы с одним эндпоинтом. Например, для работы с пользователями мы можем сделать и передать в приложение такой роутер:

Скопировать кодJAVASCRIPT

const router = require('express').Router();

router.get('/', getAllUsers);

router.post('/', createUser);

router.get('/:id', doesUserExist);

router.patch('/:id', updateUser);

app.use('/users', router);

Так мы можем описать логику работы с пользователем, например, в отдельном файле, а затем передать её в приложение. В таком случае у нас будут реализованы методы GET/POST для эндпоинта /users и GET/PATCH для эндпоинта /users/:id. Удобно!

### Мидлвэры

Основной шаблон экспресса — middleware. Это функции промежуточной обработки, которые выполняются в порядке их добавления. Мидлвэры можно добавлять всему приложению или отдельному роуту:

Скопировать кодJAVASCRIPT

*// мидлвэра всего приложения*

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

*// мидлвэры роута*

app.use('/users/:id', checkRequest);

router.get('/users/:id', doesUserExist);

Мидлвэры роутов срабатывают только для запросов, удовлетворяющих шаблону роута, который передаётся первым аргументом, например: /users/:id.

Эти функции позволяют организовать код, чтобы его было проще поддерживать. Например, такой код проверяет, есть ли пользователь, и возвращает его, если он есть:

Скопировать кодJAVASCRIPT

router.get('/users/:id', doesUserExist);

router.get('/users/:id', sendUser);

Впоследствии, если вы решите добавить функциональность, то уже не будете трогать существующий код, а просто добавите ещё один мидлвэр:

Скопировать кодJAVASCRIPT

router.get('/users/:id', doesUserExist);

*// добавили промежуточную проверку*

router.get('/users/:id', doesUserHasPermission);

router.get('/users/:id', sendUser);

### Готовые пакеты

Если вам в голову приходит какая-то функциональность, скорее всего она уже есть в готовом пакете для экспресса в npm. Такие пакеты подключают к приложению как мидлвэры. Так мы подключали body-parser:

Скопировать кодJAVASCRIPT

*// подключаем body-parser как мидлвэру всего приложения*

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

Надеемся, вам понравилось работать с экспрессом: с ним код писать проще и быстрее. Мы будем и дальше работать с фреймворком. Эта тема — только начало.

# Дополнительные материалы

Советуем пройти этот урок уже после того, как сделаете проектное задание.

### Как решить проблемы совместимости JavaScript

В Node.js есть та же проблема совместимости, что и в браузерах. Новые возможности JavaScript появляются в ноде не сразу. Эту проблему решает удобная [таблица](https://kangax.github.io/compat-table/es6/), по которой можно проверить поддерживается функциональность или ещё нет.

### Особенности асинхронности в Node.js

Асинхронная модель ноды очень похожа на браузерную. Однако и тут есть свои нюансы. Почитать о них можно [в этой статье](https://medium.com/webbdev/js-db3d35ffed7e).

### Как передавать огромные файлы

В предыдущей теме вы познакомились с потоками. Когда разворачиваете сервер на Node.js, умение работать с потоками не нужно — в express за нас это делает парсер. Но если приходится работать с файлами весом от десятка мегабайт, потоки незаменимы. Иначе вам не построить стабильное и быстрое приложение. Разобраться с потоками поможет [этот материал](https://jscomplete.com/learn/node-beyond-basics/node-streams) (на английском языке).

### Как выстроить сложную логику роутинга

Обычно чем больше проект, тем больше в нём роутеров. Мы рассказали об азах организации роутинга по модулям. Если хотите разобраться, как разбить на модули сложную логику роутинга, советуем прочитать [этот туториал](https://scotch.io/tutorials/keeping-api-routing-clean-using-express-routers) (на английском языке).

**Настройте линтер**

Он отлавливает ошибки и следит за единообразием кода. Мы пользуемся линтером eslint. В него можно сразу загружать готовые конфигурации — наборы настроек, сделанные на основе стайлгайдов (принципов написания кода). Мы будем работать по самому популярному стайлгайду — Airbnb.

Для начала установите три dev-зависимости:

* Сам ESLint. [Инструкцию можно найти по ссылке](https://www.npmjs.com/package/eslint).

// Вы можете установить ESLint с помощью npm:

$ npm install eslint --save-dev

Затем вы должны настроить файл конфигурации:

$ ./node\_modules/.bin/eslint –init

Если нет доступа, то: Проблема заключается в том, что политика выполнения скриптов запрещает выполнять эти самые скрипты. Узнать текущее значение политики можно командой:

|  |  |
| --- | --- |
| 1 | Get-ExecutionPolicy |

Для своей системы я получил значение:

|  |  |
| --- | --- |
| 1 | Restricted |

Чтобы разрешить выполнение файлов с расширением .ps1, то есть чтобы запустить скрипт PowerShell в Windows, выполните команду:

|  |  |
| --- | --- |
| 1 | Set-ExecutionPolicy unrestricted |

После этого вы можете запустить ESLint в любом файле или каталоге, подобном этому:

$ ./node\_modules/.bin/eslint yourfile.js

* Ещё две dev-зависимости, чтобы линтер работал по правилам стайлгайда Airbnb:

Скопировать кодBASH

eslint-config-airbnb-base

eslint-plugin-import

В корне проекта создайте файл .eslintrc и добавьте в него:

Скопировать код

{

"extends": "airbnb-base"

}

В файл package.json добавьте новую команду lint. При её запуске выполняется запуск eslint: npx eslint. После выполнения в консоли не должно быть сообщений об ошибках.

И последний шаг: установите расширение eslint в редактор кода. В Visual Studio Code оно так и называется — "ESLint". После установки расширения перезагрузите редактор.

export async function register(name, email, password) {

const res = await fetch(`${MYAPI\_BASE\_URL}/signup`, {

method: 'POST',

credentials: 'include',

headers: {

"Content-Type": "application/json",

},

body: JSON.stringify({ name, email, password }),

});

const json = res.json();

return json;