**Практическая работа по Express**

*Быстрый, гибкий, минималистичный веб-фреймворк для приложений*

**Цель**

Освоить веб-фреймворк Express в объеме, достаточном для построения интернет-приложений на базе [Node.js.](https://nodejs.org/)

**Предварительные требования**

Понимание принципов работы веб-приложений и веб-API. Базовые знания javascript, [Node.js](https://www.gitbook.com/book/nodejs-junior-developer-traini/super-book-of-node-js/edit), [npm](https://www.npmjs.com/" \t "_blank)

**Что такое Express?**

Express - это минималистичный и гибкий веб-фреймворк для приложений [Node.js](https://www.gitbook.com/book/nodejs-junior-developer-traini/super-book-of-node-js/edit), предоставляющий обширный набор функций для мобильных и веб-приложений.

**Установка**

Предположим, вы уже установили [Node.js](https://nodejs.org/). Создайте каталог для своего приложения и сделайте его своим рабочим каталогом.

$ mkdir myapp

$ cd myapp

С помощью команды npm init создайте файл package.json для своего приложения.

$ npm init

Примите предлагаемые значения по умолчанию для большинства пунктов, кроме следующего:

entry point: (index.js)

Введите app.js или любое другое имя главного файла по своему желанию. Если вас устраивает index.js, нажмите клавишу ВВОД, чтобы принять предложенное имя файла по умолчанию.

Теперь установите Express в каталоге app и сохраните его в списке зависимостей. Например:

$ npm install express --save

**Пример “Hello world”**

Ниже приведен пример самого простого приложения, которое можно создать с помощью Express.

В каталоге app создайте файл с именем app.js и добавьте следующий код:

var express = require('express');

var app = express();

app.get('/', function (req, res) {

res.send('Hello World!');

});

app.listen(3000, function () {

console.log('Example app listening on port 3000!');

});

Приложение запускает сервер и слушает соединения на порте 3000. Приложение выдает ответ “Hello World!” на запросы, адресованные корневому URL (/) или*маршруту*. Для всех остальных путей ответом будет **404 Not Found**.

req(запрос) и res(ответ) являются теми же объектами, которые предоставляет Node, поэтому можно вызвать req.pipe(), req.on('data', callback) и выполнить любые другие действия, не требующие участия Express.

Запустите приложение с помощью следующей команды:

$ node app.js

После этого откройте в браузере страницу <http://localhost:3000/>, чтобы просмотреть результат.

**Основы маршрутизации**

*Маршрутизация* определяет, как приложение отвечает на клиентский запрос к конкретному адресу (конечной точке), которым является URI (или путь), и определенному методу запроса HTTP (GET, POST и т.д.).

Каждый маршрут может иметь одну или несколько функций обработки, которые выполняются при сопоставлении маршрута.

Определение маршрута имеет следующую структуру:

app.METHOD(PATH, HANDLER)

Где:

* app - это экземплярexpress.
* METHOD- [метод запроса HTTP](http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol).
* PATH - путь на сервере.
* HANDLER - функция, выполняемая при сопоставлении маршрута.

Приведенные ниже элементарные примеры иллюстрируют способ определения простых маршрутов.

Ответ Hello World! на домашней странице:

app.get('/', function (req, res) {

res.send('Hello World!');

});

Ответ на запрос POST в корневом маршруте (/), на домашней странице приложения:

app.post('/', function (req, res) {

res.send('Got a POST request');

});

Ответ на запрос PUT, адресованный маршруту /user:

app.put('/user', function (req, res) {

res.send('Got a PUT request at /user');

});

Ответ на запрос DELETE, адресованный маршруту /user:

app.delete('/user', function (req, res) {

res.send('Got a DELETE request at /user');

});

Дополнительная информация о маршрутизации приведена в [руководстве по маршрутизации](https://expressjs.com/ru/guide/routing.html).

**Промежуточные обработчики (middleware)**

Функции *промежуточной обработки* (middleware) - это функции, имеющие доступ к [объекту запроса](https://expressjs.com/ru/4x/api.html#req) (req), [объекту ответа](https://expressjs.com/ru/4x/api.html#res) (res) и к следующей функции промежуточной обработки в цикле “запрос-ответ” приложения. Следующая функция промежуточной обработки, как правило, обозначается переменной next.

Функции промежуточной обработки могут выполнять следующие задачи:

* Выполнение любого кода.
* Внесение изменений в объекты запросов и ответов.
* Завершение цикла “запрос-ответ”.
* Вызов следующего промежуточного обработчика из стека.

Если текущая функция промежуточной обработки не завершает цикл “запрос-ответ”, она должна вызвать next()для передачи управления следующей функции промежуточной обработки. В противном случае запрос зависнет.

Ниже представлены элементы вызова функции промежуточного обработчика:

1. Метод HTTP, к которому применяется данный промежуточный обработчик.
2. Путь (маршрут), к которому применяется данный промежуточный обработчик.
3. Функция промежуточного обработчика.
4. Аргумент обратного вызова для функции промежуточного обработчика, именуемый "next" согласно стандарту.
5. Аргумент [ответа](https://expressjs.com/ru/4x/api.html#res) HTTP, именуемый "res" согласно стандарту.
6. Аргумент [запроса](https://expressjs.com/ru/4x/api.html#req) HTTP, именуемый "req" согласно стандарту.

Ниже приводится простой пример промежуточного обработчика “myLogger”. Эта функция печатает слово “LOGGED” при прохождении запроса, адресованного приложению, через приложение. Данная функция промежуточного обработчика присвоена переменной с именем myLogger.

var myLogger = function (req, res, next) {

console.log('LOGGED');

next();

};

Обратите внимание на вызов next() выше. Вызов этой функции активирует следующую функцию промежуточной обработки в приложении. Функция next() не является частью Node.js или Express API, но представляет собой третий аргумент, передаваемый в функцию промежуточного обработчика. Функция next() могла бы иметь любое имя, но, согласно стандарту, она всегда называется “next”. Во избежание путаницы, рекомендуется всегда придерживаться данного стандарта.

Для того чтобы загрузить функцию промежуточного обработчика вызовите app.use() с указанием соответствующей функции. Например, приведенный ниже код загружает функцию промежуточного обработчика myLogger перед маршрутом к корневому расположению (/).

var express = require('express');

var app = express();

var myLogger = function (req, res, next) {

console.log('LOGGED');

next();

};

app.use(myLogger);

app.get('/', function (req, res) {

res.send('Hello World!');

});

app.listen(3000);

Каждый раз при получении запроса приложение выводит на терминал сообщение “LOGGED”.

Порядок загрузки промежуточных обработчиков очень важен: функции промежуточных обработчиков, загруженные первыми, выполняются в первую очередь.

Если myLogger загружается после маршрута к корневому расположению, запрос никогда не достигает его, и приложением не выводится сообщение “LOGGED”, поскольку обработчик маршрута корневого пути завершает цикл “запрос-ответ”.

Промежуточный обработчик myLogger всего лишь выводит сообщение, затем передает запрос далее, следующему промежуточному обработчику в стеке, путем вызова функции next().

В следующем примере выполняется добавление свойства requestTime в объект запроса. Назовем эту функцию промежуточного обработчика “requestTime”.

var requestTime = function (req, res, next) {

req.requestTime = Date.now();

next();

};

Теперь приложением используется функция промежуточного обработчика requestTime. Кроме того, функция обратного вызова маршрута корневого расположения (пути) использует свойство, добавленную функций промежуточного обработчика в req(объект запроса).

var express = require('express');

var app = express();

var requestTime = function (req, res, next) {

req.requestTime = Date.now();

next();

};

app.use(requestTime);

app.get('/', function (req, res) {

var responseText = 'Hello World!';

responseText += 'Requested at: ' + req.requestTime + '';

res.send(responseText);

});

app.listen(3000);

Если запрос адресован корневому каталогу приложения, приложение выводит на экран системное время запроса в браузере.

Благодаря наличию доступа к объекту запроса, объекту ответа, следующей функции промежуточного обработчика в стеке и к API Node.js в целом, возможности, связанные с промежуточными обработчиками, являются бесконечными.

Дополнительная информация о промежуточных обработчиках Express содержится в разделе [Использование промежуточных обработчиков Express](https://expressjs.com/ru/guide/using-middleware.html)

**Предоставление статических файлов в Express**

Для предоставления статических файлов, например, изображений, файлов CSS и JavaScript в Express используется функция промежуточной обработки express.static.

Для того чтобы начать непосредственное предоставление файлов, необходимо передать имя каталога, в котором находятся статические ресурсы, в функцию промежуточной обработки express.static. Например, воспользуйтесь приведенным ниже кодом для предоставления изображений, файлов CSS и JavaScript, расположенных в каталоге public:

app.use(express.static('public'));

Теперь можно загрузить файлы, находящиеся в каталоге public directory:

http://localhost:3000/images/kitten.jpg

http://localhost:3000/css/style.css

http://localhost:3000/js/app.js

http://localhost:3000/images/bg.png

http://localhost:3000/hello.html

Express выполняет поиск файлов относительно статического каталога, поэтому имя статического каталога не является частью URL.

Для использования нескольких каталогов, содержащих статические ресурсы, необходимо вызвать функцию промежуточной обработки express.static несколько раз:

app.use(express.static('public'));

app.use(express.static('files'));

Express выполняет поиск файлов в том порядке, в котором указаны статические каталоги в функции промежуточной обработки express.static.

Для того чтобы создать префикс виртуального пути (то есть, пути, фактически не существующего в файловой системе) для файлов, предоставляемых с помощью функции express.static, необходимо [указать путь монтирования](https://expressjs.com/ru/4x/api.html#app.use) для статического каталога, как показано ниже:

app.use('/static', express.static('public'));

Теперь можно загрузить файлы, находящиеся в каталоге public, указанного в префиксе пути /static.

http://localhost:3000/static/images/kitten.jpg

http://localhost:3000/static/css/style.css

http://localhost:3000/static/js/app.js

http://localhost:3000/static/images/bg.png

http://localhost:3000/static/hello.html

Тем не менее, путь, переданный в функцию express.static, указан относительно каталога, из которого запускается процесс node. В случае запуска приложения Express из другого каталога, безопаснее использовать абсолютный путь к каталогу для предоставления файлов:

app.use('/static', express.static(\_\_dirname + '/public'));

**Использование шаблонизаторов в Express**

Для того чтобы отображать в Express файлы шаблонов, необходимо задать следующие параметры приложения:

* views, каталог, в котором находятся файлы шаблонов. Например: app.set('views', './views')
* view engine, используемый шаблонизатор. Например: app.set('view engine', 'pug')

Затем установите соответствующий пакет npm шаблонизатора:

$ npm install pug --save

Шаблонизаторы, совместимые с Express, например, Pug, экспортируют функцию \_\_express(filePath, options, callback), вызываемую с помощью функции res.render() для вывода кода шаблона.

Это правило действует не для всех шаблонизаторов. Библиотека [Consolidate.js](https://www.npmjs.org/package/consolidate) соблюдает его путем преобразования всех популярных шаблонизаторов Node.js, благодаря чему работает в Express без проблем.

После указания механизма визуализации (view engine) не нужно указывать его или загружать модуль шаблонизатора в приложение; Express загружает модуль внутренними средствами, как показано далее (для примера, приведенного выше).

app.set('view engine', 'pug');

Создайте файл шаблона Pug с именем index.pug в каталоге views со следующим содержанием:

html

head

title!= title

body

h1!= message

Затем создайте маршрут для вывода файла index.pug. Если свойство view engine не задано, необходимо указать расширение файла view. В противном случае, можно не указывать расширение.

app.get('/', function (req, res) {

res.render('index', { title: 'Hey', message: 'Hello there!'});

});

При выполнении запроса к домашней странице файл index.pug будет отображаться как HTML.

Для получения дополнительной информации о работе шаблонизаторов в Express обратитесь к разделу [“Разработка шаблонизаторов для Express”](https://expressjs.com/ru/advanced/developing-template-engines.html).

**Обработка ошибок**

Функции промежуточного обработчика для обработки ошибок определяются так же, как и другие функции промежуточной обработки, но с указанием для функции обработки ошибок не трех, а четырех аргументов: (err, req, res, next). Например:

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

console.error(err.stack);

res.status(500).send('Something broke!');

});

Промежуточный обработчик для обработки ошибок должен быть определен последним, после указания всех app.use() и вызовов маршрутов; например:

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

var methodOverride = require('method-override');

app.use(bodyParser());

app.use(methodOverride());

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

// logic

});

Ответы, поступающие из функции промежуточной обработки, могут иметь любой формат, в зависимости от ваших предпочтений. Например, это может быть страница сообщения об ошибке HTML, простое сообщение или строка JSON.

В целях упорядочения (и для фреймворков более высокого уровня) можно определить несколько функций промежуточной обработки ошибок точно так же, как это допускается для обычных функций промежуточной обработки. Например, для того чтобы определить обработчик ошибок для запросов, совершаемых с помощью XHR, и для остальных запросов, можно воспользоваться следующими командами:

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

var methodOverride = require('method-override');

app.use(bodyParser());

app.use(methodOverride());

app.use(logErrors);

app.use(clientErrorHandler);

app.use(errorHandler);

В данном примере базовый код logErrors может записывать информацию о запросах и ошибках в stderr, например:

function logErrors(err, req, res, next) {

console.error(err.stack);

next(err);

}

Кроме того, в данном примере clientErrorHandler определен, как указано ниже; в таком случае ошибка явным образом передается далее следующему обработчику:

function clientErrorHandler(err, req, res, next) {

if (req.xhr) {

res.status(500).send({ error: 'Something failed!' });

} else {

next(err);

}

}

“Обобщающая” функция errorHandler может быть реализована так:

function errorHandler(err, req, res, next) {

res.status(500);

res.render('error', { error: err });

}

При передаче какого-либо объекта в функцию next()(кроме строки 'route'), Express интерпретирует текущий запрос как ошибку и пропустит все остальные функции маршрутизации и промежуточной обработки, не являющиеся функциями обработки ошибок. Для того чтобы обработать данную ошибку определенным образом, необходимо создать маршрут обработки ошибок, как описано в следующем разделе.

Если задан обработчик ошибок с несколькими функциями обратного вызова, можно воспользоваться параметром route, чтобы перейти к следующему обработчику маршрута. Например:

app.get('/a\_route\_behind\_paywall',

function checkIfPaidSubscriber(req, res, next) {

if(!req.user.hasPaid) {

// continue handling this request

next('route');

}

}, function getPaidContent(req, res, next) {

PaidContent.find(function(err, doc) {

if(err) return next(err);

res.json(doc);

});

});

В данном примере обработчик getPaidContent будет пропущен, но выполнение всех остальных обработчиков в app для /a\_route\_behind\_paywall будет продолжено.

Вызовы next() и next(err) указывают на завершение выполнения текущего обработчика и на его состояние. next(err) пропускает все остальные обработчики в цепочке, кроме заданных для обработки ошибок, как описано выше.

Также в данном примере следует обратить внимание на то, что функции middleware могут подключаться к роуту как доплонительные параметры.

**Стандартный обработчик ошибок**

В Express предусмотрен встроенный обработчик ошибок, который обрабатывает любые возможные ошибки, встречающиеся в приложении. Этот стандартный обработчик ошибок добавляется в конец стека функций промежуточной обработки.

В случае передачи ошибки в next()без обработки с помощью обработчика ошибок, такая ошибка будет обработана встроенным обработчиком ошибок. Ошибка будет записана на клиенте с помощью трассировки стека. Трассировка стека не включена, если приложение запущено в рабочем режиме (production).

Для запуска приложения в рабочем режиме необходимо задать для переменной среды NODE\_ENV значение production.

При вызове next() с ошибкой после начала записи ответа (например, если ошибка обнаружена во время включения ответа в поток, направляемый клиенту), стандартный обработчик ошибок Express закрывает соединение и отклоняет запрос.

Поэтому при добавлении нестандартного обработчика ошибок вам потребуется делегирование в стандартные механизмы обработки ошибок в Express в случае, если заголовки уже были отправлены клиенту:

function errorHandler(err, req, res, next) {

if (res.headersSent) {

return next(err);

}

res.status(500);

res.render('error', { error: err });

}

**Практическое задание**

1. Создать веб сервер на Express
2. Обеспечить обработку запросов GET (должен возвращать параметры запроса) и POST (должен возвращать тело запроса),
3. Обеспечить выдачу статического контента
4. Обеспечить обаботку ошибки 404

**Глоссарий**

| **Термин** | **Значение** |
| --- | --- |
| application (приложение) | В общем смысле, одна или несколько программ, предназначенных для выполнения операций с определенной целью. В контексте Express - программа, использующая API Express, запущенный на платформе Node.js. Также называется объектом приложения. |
| API | Интерфейс программирования приложений. Рекомендуется расшифровывать данную аббревиатуру при первомупоминании. |
| request (запрос) | Запрос HTTP. Клиент передает на сервер сообщение-запрос по протоколу HTTP, а сервер возвращает ответ. В запросе должен использоваться один из нескольких методов запроса, например, GET, POST и т.д. |
| response (ответ) | Ответ HTTP. Сервер возвращает клиенту сообщение-ответ по протоколу HTTP. Ответ содержит информацию о состоянии выполнения запроса, а также в теле сообщения может быть включено запрашиваемое содержимое. |
| route (маршрут) | Часть URL, идентифицирующая ресурс. Например, в <http://foo.com/products/id>, “/products/id” является маршрутом. |

**Ссылки**

1. <https://expressjs.com/ru/> - домашняя страница Express
2. <https://expressjs.com/ru/4x/api.html> - справочник API
3. <https://expressjs.com/ru/resources/books-blogs.html> - книги и блоги