EEXPRESS

Express - это минималистичный и гибкий веб-фреймворк для приложений Node.js, предоставляющий обширный набор функций для мобильных и веб-приложений.

Посилання

https://expressjs.com

<https://expressjs.com/ru/>

Статус коди:

<https://developer.mozilla.org/ru/docs/Web/HTTP/Status>

Встановлення

https://expressjs.com/ru/starter/installing.htm

|  |  |
| --- | --- |
| Створюємо папку, де буде розміщуватися проект |  |
| Створюємо файл package.json , де будуть розміщуватися основні властивості проекту | npm init |
| Завантажуємо пакет Express (щоб пакет було додано у файл package.json  потрібно використовувати директиву --save) | npm install express --save |
| Створення проекту засобами express  <https://expressjs.com/ru/starter/generator.html> | Встановлення майстра формування проекту  npm install express-generator -g  Створення проекту  express --view=ejs myapp  Перехід у папку проекту і встановлення залежностей  cd myapp  npm install  Запуск додатку  set DEBUG=myapp:\*  npm start |
|  |  |

**Загальна схема використання**

|  |  |  |
| --- | --- | --- |
| Підключення модуля |  | var express = require('express'); |
| Створення об’єкта веб-додатку | var додаток = express(); | var app = express(); |
| Призначення обробника запиту | додаток . метод\_запиту ( маршрут , функція\_обробник);   * метод\_запиту: get, post, put, delete; * функція обробник:   function( об’єкт\_запиту, об’єкт\_відповіді ) { . . . } | app.get('/', function (req, res) {  res.send('Hello World!');  }); |
| Запуск веб-додатку | додаток . listen ( порт , функція-колбек);  порт – порт, який буде прослуховатися додатком;  функція-колбек – функція, яку буде викликано після запуску додатку | app.listen(3000, function () {  console.log('Example app listening on port 3000!');  }); |

| **Метод** | **Описание** |  |
| --- | --- | --- |
| [res.download()](https://expressjs.com/ru/4x/api.html#res.download) | Приглашение загрузки файла. | res.download('/report-12345.pdf');  res.download('/report-12345.pdf', 'report.pdf');  res.download('/report-12345.pdf', 'report.pdf', function(err){  if (err) {  // Handle error, but keep in mind the response may be partially-sent  // so check res.headersSent  } else {  // decrement a download credit, etc.  }  }); |
| [res.end()](https://expressjs.com/ru/4x/api.html#res.end) | Завершение процесса ответа. |  |
| [res.json()](https://expressjs.com/ru/4x/api.html#res.json) | Отправка ответа JSON. |  |
| [res.jsonp()](https://expressjs.com/ru/4x/api.html#res.jsonp) | Отправка ответа JSON с поддержкой JSONP.  <https://www.w3schools.com/js/js_json_jsonp.asp>  <https://learn.javascript.ru/ajax-jsonp> |  |
| [res.redirect()](https://expressjs.com/ru/4x/api.html#res.redirect) | Перенаправление ответа. |  |
| [res.render()](https://expressjs.com/ru/4x/api.html#res.render) | Вывод шаблона представления. |  |
| [res.send()](https://expressjs.com/ru/4x/api.html#res.send) | Отправка ответа различных типов. |  |
| [res.sendFile](https://expressjs.com/ru/4x/api.html#res.sendFile) | Отправка файла в виде потока октетов. | app.get('/file/:name', function (req, res, next) {  var options = {  root: \_\_dirname + '/public/',  dotfiles: 'deny',  headers: {  'x-timestamp': Date.now(),  'x-sent': true  }  };  var fileName = req.params.name;  res.sendFile(fileName, options, function (err) {  if (err) {  next(err);  } else {  console.log('Sent:', fileName);  }  });  }); |
| [res.sendStatus()](https://expressjs.com/ru/4x/api.html#res.sendStatus) | Установка кода состояния ответа и отправка представления в виде строки в качестве тела ответа. |  |

Основи маршрутизації

<https://expressjs.com/ru/guide/routing.html>

Під маршрутизацією розуміють прив’язку до кожного із можливих запитів відповідної функції обробника

|  |  |
| --- | --- |
| Загальна схема | Приклади |
| app.METHOD(PATH, HANDLER)   * app - екземпляр express; * METHOD - [метод запроса HTTP](http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol) (get, post, put, delete). * PATH – шлях, що було вказано у сервері (після доменного імені сервера); * HANDLER – функція-обробник, яка буде викликатися при зверненні за даним маршрутом. | Відповідь на GET у кореневому маршруті  app.get('/', function (req, res) {  res.send('Hello World!');  });  Відповідь на POST у кореневому маршруті  app.post('/', function (req, res) {  res.send('Got a POST request');  });  Відповідь на PUT за маршрутом /user:  app.put('/user', function (req, res) {  res.send('Got a PUT request at /user');  });  Відповідь на DELETE за маршрутом /user:  app.delete('/user', function (req, res) {  res.send('Got a DELETE request at /user');  }); |
|  |  |
|  |  |
|  |  |

Загалом express підтримує такі HTTP методи: get, post, put, head, delete, options, trace, copy, lock, mkcol, move, purge, propfind, proppatch, unlock, report, mkactivity, checkout, merge, m-search, notify, subscribe, unsubscribe, patch, search и connect.

**Способи задання маршрутів**

При заданні маршрутів можна задавати : конкретні шляхи, шаблони рядків та регулярні вирази

|  |  |
| --- | --- |
| задання конкретних шляхів | Обробка запиту за кореневим маршрутом /  (приклад запиту: <http://localhost:3000> **/**).  app.get('/', function (req, res) {  res.send('root');  });  Обробка запиту за маршрутом /about  (приклад запиту: <http://localhost:3000> **/about**).  app.get('/about', function (req, res) {  res.send('about');  });  Обробка запиту за маршрутом /about  (приклад запиту: <http://localhost:3000> **/random.text**).  app.get('/random.text', function (req, res) {  res.send('random.text');  }); |
| Задання шаблонів маршрутів  Символы ?, +, \* и () представляют собой подмножества соответствующих им регулярных выражений. Дефис (-) и точка (.) интерпретируются буквально в путях на основе строк. | Приведенный ниже путь маршрута сопоставляет acd и abcd.  app.get('/ab?cd', function(req, res) {  res.send('ab?cd');  });  Этот путь маршрута сопоставляет abcd, abbcd, abbbcd и т.д.  app.get('/ab+cd', function(req, res) {  res.send('ab+cd');  });  Этот путь маршрута сопоставляет abcd, abxcd, abRABDOMcd, ab123cd и т.д.  app.get('/ab\*cd', function(req, res) {  res.send('ab\*cd');  });  Данный путь маршрута сопоставляет /abe и /abcde.  app.get('/ab(cd)?e', function(req, res) {  res.send('ab(cd)?e');  }); |
| Задання маршрутів за допомогою регулярних виразів | Данный путь маршрута сопоставляет любой элемент с “a” в имени маршрута.  app.get(/a/, function(req, res) {  res.send('/a/');  });  Данный маршрут сопоставляет butterfly и dragonfly, но не butterflyman, dragonfly man и т.д.  app.get(/.\*fly$/, function(req, res) {  res.send('/.\*fly$/');  }); |
|  |  |
|  |  |

Способи задання обробників

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Задання однієї функції обробника | додаток . метод\_запиту ( маршрут , функція\_обробник); | app.get('/example/a', function (req, res) {  res.send('Hello from A!');  }); |
| Задання декількох окремих функцій обробників | додаток . метод\_запиту ( маршрут ,  функція\_обробник\_1 (req, res, next)  {  // . . . команди . . .  next(); // Передача управління наступній функції  },  функція\_обробник\_2 (req, res, next)  {  // . . . команди . . .  next(); // Передача управління наступній функції  },  . . .  ); | app.get('/example/b',  function (req, res, next) {  console.log('the response will be sent by the next function ...');  next();  },  function (req, res) {  res.send('Hello from B!');  });  ===========================  const jsonParser = express.json();  app.post("/user",  **jsonParser**,  function (request, response) {      console.log(request.body);      if(!request.body) return response.sendStatus(400);      console.log(request.body);      response.json(`${request.body.userName} - ${**request.body.userAge**}`);  }); |
| Задання масиву функцій | додаток . метод\_запиту ( маршрут , масив\_функцій\_обробників );  //- - - - - - Опис функцій обробників - - - - - | **//- - - - - - Опис функцій обробників - - - -**  var cb0 = function (req, res, next) {  console.log('CB0');  next();  }  var cb1 = function (req, res, next) {  console.log('CB1');  next();  }  var cb2 = function (req, res) {  res.send('Hello from C!');  }  **//- - Задання масиву функцій обробників - - -**  app.get('/example/c', [cb0, cb1, cb2]); |
| Комбінований (і окремі функції, і масив функцій |  | var cb0 = function (req, res, next) {  console.log('CB0');  next();  }  var cb1 = function (req, res, next) {  console.log('CB1');  next();  }  app.get('/example/d', [cb0, cb1], function (req, res, next) {  console.log('the response will be sent by the next function ...');  next();  }, function (req, res) {  res.send('Hello from D!');  }); |
| Задання для одного маршруту відповідних функцій-обробників для різних методів запиту | додаток . route(маршрут)  . метод\_запиту\_1 (функція\_обробник )  . метод\_запиту\_2 (функція\_обробник )  . метод\_запиту\_3 (функція\_обробник )  . . . . . . . .  . метод\_запиту (функція\_обробник ); | app.route(‘/book’)  // ---- Для методу GET -----  .get(function(req, res) {  res.send(‘Get a random book’);  })  // ---- Для методу POST-----  .post(function(req, res) {  res.send(‘Add a book’);  })  // ---- Для методу PUT -----  .put(function(req, res) {  res.send(‘Update the book’);  }); |
| Задання обробників маршрутів за допомогою express.Router | // 1. створення Router  **const router = express.Router();**  // 2.заданяння дочірніх маршрутів  **router**.use("/ дочірн.маршрут\_1 ", function(request, response){    . . . . . . . .  });  **router**.use("/ дочірн.маршрут\_2 ", function(request, response){    . . . . . . . .  });  module.exports = router;  . . . . . . . . . . . . . . . . .  //3.приєднання до основного маршруту  **app.use("/осн.маршрут", router);** | express.Router С помощью класса express.Router можно создавать модульные, монтируемые обработчики маршрутов. Экземпляр Routerпредставляет собой комплексную систему промежуточных обработчиков и маршрутизации; по этой причине его часто называют “мини-приложением”.  В приведенном ниже примере создается маршрутизатор в виде модуля, в него загружается функция промежуточной обработки, определяется несколько маршрутов, и модуль маршрутизатора монтируется в путь в основном приложении.  Создайте файл маршрутизатора с именем birds.js в каталоге приложения со следующим содержанием:  var express = require('express');  var router = express.Router();  // middleware that is specific to this router  router.use(function timeLog(req, res, next) {  console.log('Time: ', Date.now());  next();  });  // define the home page route  router.get('/', function(req, res) {  res.send('Birds home page');  });  // define the about route  router.get('/about', function(req, res) {  res.send('About birds');  });  module.exports = router;  Потом загрузите модуль маршрутизации в приложение:  var birds = require('./birds');  ...  app.use('/birds', birds);  Данное приложение теперь сможет обрабатывать запросы, адресованные ресурсам /birds и /birds/about, а также вызывать специальную функцию промежуточной обработки timeLog данного маршрута. |

Використання параметрів у маршрутах

При виконанні запиту маршрут може містити деякі додатків параметри (окрім параметрів get запиту). Як правило це використовують для того, щоб виокремити важливі для аналізу частини маршруту. Для цього використовують символ “:”. Доступ до таких параметрів здійснюється з використанням **request.params**

|  |  |  |
| --- | --- | --- |
|  |  |  |
| передача одного параметра |  | const express = require("express");  const app = express();    app.get("/products**/:productId**", function (request, response) {    response.send("productId: " + **request.params**["productId"])  }); |
| передача декількох параметрів |  | app.get("/book**/:pageName**.:**pageExt**", function (request, response) {      let pageName = **request.params**["pageName"];      let pageExt = **request.params**["pageExt"];      response.send(`Запрошенный файл: ${pageName}.${pageExt}`);  }); |

**Використання middleware**

Функції проміжкової обробки (допоміжні функції) – це функції, які можуть бути використані для виконання деяких додаткових дій перед застосуванням кінцевої функції обробки запиту. Такі функції мають доступ до об’єктів запиту (request) і відповіді (response). Як додатковий параметр передається покажчик next на наступну функції, яку слід викликати. Виклик наступної функції повинен знаходитись всередині функції.

функція\_проміжкової\_обробки (req, res, next)

{

// . . . команди . . .

next(); // Передача управління наступній функції

}

Фунції проміжковї обробки можуть бути використані для:

* виконання деяких команд;
* додавання нових зміна властивостей у об’єктах req, res;
* завершення циклу «запит відповідь».

Можна виділити такі типи функцій проміжкової обробки :

* проміжковий обробник рівня маршруту (викликається тільки у випадку зверненні за вказаним маршрутом);
* проміжковий обробник рівня додатку (викликається при будь-якому запиті);
* проміжковий обробник для обробки помилок.

Приложение Express может использовать следующие типы промежуточных обработчиков:

* [Промежуточный обработчик уровня приложения](https://expressjs.com/ru/guide/using-middleware.html#middleware.application)
* [Промежуточный обработчик уровня маршрутизатора](https://expressjs.com/ru/guide/using-middleware.html#middleware.router)
* [Промежуточный обработчик для обработки ошибок](https://expressjs.com/ru/guide/using-middleware.html#middleware.error-handling)
* [Встроенные промежуточные обработчики](https://expressjs.com/ru/guide/using-middleware.html#middleware.built-in)
* [Промежуточные обработчики сторонних поставщиков ПО](https://expressjs.com/ru/guide/using-middleware.html#middleware.third-party)

**Використання для обробки запитів**

|  |  |  |
| --- | --- | --- |
| Проміжковий обробник рівня маршруту | додаток . метод\_запиту ( маршрут ,  функція\_обробник\_1 (req, res, next)  {  // . . . команди . . .  next(); // Передача управління наступній функції  },  функція\_обробник\_2 (req, res, next)  {  // . . . команди . . .  next(); // Передача управління наступній функції  },  . . .  ); | app.get('/example/b',  function (req, res, next) {  console.log('the response will be sent by the next function ...');  next();  },  function (req, res) {  res.send('Hello from B!');  });  ===========================  const jsonParser = express.json();  app.post("/user",  **jsonParser**,  function (request, response) {      console.log(request.body);      if(!request.body) return response.sendStatus(400);      console.log(request.body);      response.json(`${**request.body.userName**} - ${**request.body.userAge**}`);  }); |
| Проміжковий обробник рівня додатку | var доп\_функція=function (req, res, next) {  . . . . . . . . . . . .  next();  };  //-------- Викликається для будь-якого запиту  app.use ( доп\_функція ); | 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);  **======================================**  **Приклад. Допоміжна функція додає додаткову інформацію (поточну дату) до request**  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); |
| Проміжковий обробник помилок (повиен містити 4 аргументи,  вказується останнім)  <https://expressjs.com/ru/guide/error-handling.html> | var доп\_функція=function (err, req, res, next) {  . . . . . . . . . . . .  }; | app.use(function(err, req, res, next) {  console.error(err.stack);  res.status(500).send('Something broke!');  }); |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

Передача управління від одного обробника до іншого next('route')

Обработчики маршрутов позволяют определить несколько маршрутов для одного пути. В приведенном ниже примере определено два маршрута для запросов GET, адресованных ресурсам в пути /user/:id. Второй маршрут не создает никаких неудобств, но его вызов никогда не будет выполнен, поскольку первый маршрут завершает цикл “запрос-ответ”.

В данном примере представлен вспомогательный стек промежуточных обработчиков для обработки запросов GET, адресованных ресурсам в пути /user/:id.

app.get('/user/:id', function (req, res, next) {

console.log('ID:', req.params.id);

next();

}, function (req, res, next) {

res.send('User Info');

});

// handler for the /user/:id path, which prints the user ID

app.get('/user/:id', function (req, res, next) {

res.end(req.params.id);

});

Для того чтобы пропустить остальные функции дополнительной обработки в стеке промежуточных обработчиков маршрутизатора, вызовите next('route') для передачи управления следующему маршруту. **ПРИМЕЧАНИЕ**: next('route')работает только в функциях промежуточной обработки, загруженных с помощью функций app.METHOD() или router.METHOD().

В данном примере представлен вспомогательный стек промежуточных обработчиков для обработки запросов GET, адресованных ресурсам в пути /user/:id.

app.get('/user/:id', function (req, res, next) {

// if the user ID is 0, skip to the next route

if (req.params.id == 0) next('route');

// otherwise pass the control to the next middleware function in this stack

else next(); //

}, function (req, res, next) {

// render a regular page

res.render('regular');

});

// handler for the /user/:id path, which renders a special page

app.get('/user/:id', function (req, res, next) {

res.render('special');

});

## Промежуточный обработчик уровня маршрутизатора

Промежуточный обработчик уровня маршрутизатора работает так же, как и промежуточный обработчик уровня приложения, но он привязан к экземпляру express.Router().

var router = express.Router();

Загрузите промежуточный обработчик уровня маршрутизатора с помощью функций router.use() и router.METHOD().

В приведенном ниже примере с помощью промежуточного обработчика уровня маршрутизатора создается копия системы промежуточных обработчиков, представленной выше для обработчиков уровня приложения:

var app = express();

var router = express.Router();

// a middleware function with no mount path. This code is executed for every request to the router

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

console.log('Time:', Date.now());

next();

});

// a middleware sub-stack shows request info for any type of HTTP request to the /user/:id path

router.use('/user/:id', function(req, res, next) {

console.log('Request URL:', req.originalUrl);

next();

}, function (req, res, next) {

console.log('Request Type:', req.method);

next();

});

// a middleware sub-stack that handles GET requests to the /user/:id path

router.get('/user/:id', function (req, res, next) {

// if the user ID is 0, skip to the next router

if (req.params.id == 0) next('route');

// otherwise pass control to the next middleware function in this stack

else next(); //

}, function (req, res, next) {

// render a regular page

res.render('regular');

});

// handler for the /user/:id path, which renders a special page

router.get('/user/:id', function (req, res, next) {

console.log(req.params.id);

res.render('special');

});

// mount the router on the app

app.use('/', router);

## Промежуточный обработчик для обработки ошибок

Промежуточный обработчик ошибок всегда содержит **четыре** аргумента. Для определения данной функции как обработчика ошибок необходимо указать четыре аргумента. Даже если вам не нужно использовать объект next, необходимо указать его, чтобы сохранить сигнатуру. В противном случае, объект next будет интерпретирован как обычный промежуточный обработчик, который не будет обрабатывать ошибки.

Определите функции промежуточного обработчика для обработки ошибок так же, как другие функции промежуточной обработки, но с указанием не трех, а четырех аргументов в сигнатуре (err, req, res, next)):

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

console.error(err.stack);

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

});

Подробная информация о промежуточном обработчике ошибок приведена в разделе [Обработка ошибок](https://expressjs.com/ru/guide/error-handling.html).

## Встроенные промежуточные обработчики

Начиная с версии 4.x, Express не является зависимым от [Connect](https://github.com/senchalabs/connect). За исключением express.static, все функции промежуточной обработки, ранее включенные в Express, находятся в отдельных модулях. Ознакомьтесь со [списком функций промежуточной обработки](https://github.com/senchalabs/connect#middleware).

#### express.static(root, [options])

Единственной встроенной функцией промежуточной обработки в Express является express.static. Эта функция основана на [serve-static](https://github.com/expressjs/serve-static) и отвечает за предоставление статических ресурсов приложения Express.

Аргумент root указывает на корневой каталог, из которого предоставляются статические ресурсы.

Необязательный объект options может содержать следующие свойства:

| **Свойство** | **Описание** | **Тип** | **По умолчанию** |
| --- | --- | --- | --- |
| dotfiles | Опция для предоставления файлов с точкой. Возможные значения - “allow”, “deny”, “ignore” | Строка | “ignore” |
| etag | Включение или отключение генерации etag | Булевский | true |
| extensions | Установка альтернативных вариантов расширений файлов. | Массив | [] |
| index | Отправка файла индекса каталога. Установите значение false, чтобы отключить индексацию каталога. | Смешанный | “index.html” |
| lastModified | Установка в заголовке Last-Modified даты последнего изменения файла в ОС. Возможные значения: true или false. | Булевский | true |
| maxAge | Установка значения свойства max-age в заголовке Cache-Control, в миллисекундах, или в виде строки в [формате ms](https://www.npmjs.org/package/ms) | Число | 0 |
| redirect | Перенаправление к заключительному символу “/”, если имя пути - это каталог. | Булевский | true |
| setHeaders | Функция для установки заголовков HTTP, предоставляемых с файлом. | Функция |  |

Ниже приводится пример использования функции промежуточной обработки express.static с объектом дополнительных опций:

var options = {

dotfiles: 'ignore',

etag: false,

extensions: ['htm', 'html'],

index: false,

maxAge: '1d',

redirect: false,

setHeaders: function (res, path, stat) {

res.set('x-timestamp', Date.now());

}

}

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

Для каждого приложения допускается наличие нескольких статических каталогов:

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

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

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

Дополнительную информацию о функции serve-static и ее опциях можно найти в документации по [serve-static](https://github.com/expressjs/serve-static).

## Промежуточные обработчики сторонних поставщиков ПО

Для расширения функциональности приложений Express используются промежуточные обработчики сторонних поставщиков ПО.

Установите модуль Node.js для соответствующей функциональной возможности, затем загрузите его в приложение на уровне приложения или на уровне маршрутизатора.

В приведенном ниже примере показана установка и загрузка функции промежуточной обработки для синтаксического анализа cookie cookie-parser.

$ npm install cookie-parser

var express = require('express');

var app = express();

var cookieParser = require('cookie-parser');

// load the cookie-parsing middleware

app.use(cookieParser());

Список функций промежуточных обработчиков, предоставляемых сторонними поставщиками ПО и часто используемых в Express, приведен в разделе [Промежуточные обработчики сторонних поставщиков ПО](https://expressjs.com/ru/resources/middleware.html).

**Використання шаблонізаторів**

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

**Обробка помилок**

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# 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.

## 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 в рабочую среду.

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 });

}