



Бесплатная электронная книга

УЧУСЬ

Node.js

Free unaffiliated eBook created from  
**Stack Overflow contributors.**

#node.js

.....	1
<b>1: Node.js</b> .....	<b>2</b>
.....	2
.....	2
Examples.....	6
HTTP Hello World.....	6
Hello World.....	7
Node.js.....	8
.....	9
.....	9
NodeJS.....	9
.....	<b>10</b>
Hello World with Express.....	10
.....	11
TLS Socket: .....	12
.....	<b>12</b>
<b>!</b> .....	<b>13</b>
<b>TLS Socket</b> .....	<b>13</b>
<b>TLS Socket Client</b> .....	<b>14</b>
Hello World REPL.....	15
.....	15
.....	<b>16</b>
- HTTPS!.....	21
1. ....	21
2. ....	22
3. ....	22
<b>2: async.js</b> .....	<b>24</b>
.....	24
Examples.....	24
:.....	24

<b>async.parallel()</b> .....	<b>25</b>
.....	<b>25</b>
:	26
<b>async.series()</b> .....	<b>27</b>
:	27
<b>async.times ( )</b> .....	28
<b>async.each ( )</b> .....	29
<b>async.series ( )</b> .....	29
<b>3: CLI</b> .....	<b>31</b>
.....	31
<b>Examples</b> .....	31
.....	31
<b>4: csv js</b> .....	<b>35</b>
.....	35
<b>Examples</b> .....	35
<b>FS CSV</b> .....	35
<b>5: ECMAScript 2015 (ES6) Node.js</b> .....	<b>36</b>
<b>Examples</b> .....	36
<b>const / let</b> .....	36
.....	36
.....	36
.....	37
.....	37
<b>ES6</b> .....	38
<b>6: Eventloop</b> .....	<b>39</b>
.....	39
<b>Examples</b> .....	39
.....	39
<b>Eventloop</b> .....	<b>39</b>
<b>HTTP-</b> .....	<b>39</b>
<b>HTTP-</b> .....	<b>39</b>

<b>HTTP-</b>	<b>40</b>
<b>7: HTTP</b>	<b>42</b>
Examples	42
http server	42
http client	43
<b>8: Koa Framework v2</b>	<b>45</b>
Examples	45
Hello World	45
.....	45
<b>9: Lodash</b>	<b>46</b>
.....	46
Examples	46
.....	46
<b>10: Loopback - REST</b>	<b>47</b>
.....	47
Examples	47
.....	47
<b>11: N-API,</b>	<b>49</b>
.....	49
Examples	49
, N-API	49
<b>12: Node.js (express.js) .js</b>	<b>51</b>
.....	51
Examples	51
.....	51
, -?	51
, ?	52
Express	52
AngularJS ?	53
<b>13: Node.js v6</b>	<b>55</b>
.....	55

Examples.....	55
.....	55
.....	55
.....	55
.....	56
«» .....	56
<b>14: Node.JS MongoDB.....</b>	<b>58</b>
.....	58
Examples.....	58
.....	58
.....	59
.....	59
.....	60
.....	61
.....	61
().....	61
UpdateOne.....	61
UpdateMany.....	61
ReplaceOne.....	62
.....	63
<b>15: Node.js CORS.....</b>	<b>64</b>
Examples.....	64
CORS express.js.....	64
<b>16: Node.JS ES6.....</b>	<b>65</b>
.....	65
Examples.....	65
ES6 Babel.....	65
JS es6 NodeJS.....	67
:	67
<b>17: Node.js Oracle.....</b>	<b>70</b>
Examples.....	70

Oracle.....	70
.....	70
.....	71
<b>18: NodeJS Frameworks.....</b>	<b>73</b>
Examples.....	73
- .....	73
.....	73
.....	73
.....	73
Commander.js.....	73
Vorpal.js.....	74
<b>19: NodeJS Redis.....</b>	<b>75</b>
.....	75
Examples.....	75
.....	75
.....	76
node_redis.....	78
<b>20: nvm - .....</b>	<b>80</b>
.....	80
Examples.....	80
NVM.....	80
NVM.....	80
.....	80
.....	81
nvm Mac OSX.....	81
.....	81
, NVM .....	81
.....	82
.....	82
<b>21: OAuth 2.0.....</b>	<b>84</b>
Examples.....	84
OAuth 2 Redis - grant_type: .....	84

!	92
<b>22: package.json</b>	<b>93</b>
.....	93
Examples	93
.....	93
.....	93
<b>devDependencies</b>	<b>94</b>
.....	94
.....	95
.....	95
.....	96
package.json	96
<b>23: passport.js</b>	<b>101</b>
.....	101
Examples	101
LocalStrategy in passport.js	101
<b>24: Readline</b>	<b>103</b>
.....	103
Examples	103
.....	103
CLI	103
<b>25: Require ()</b>	<b>105</b>
.....	105
.....	105
.....	105
Examples	105
require ()	105
require () NPM	106
<b>26: Sequelize.js</b>	<b>108</b>
Examples	108
.....	108
.....	

1. sequelize.define (modelName, attributes, [options])	109
2. sequelize.import ()	109
<b>27:</b>	<b>111</b>
Examples	111
nodemon	111
<b>nodemon</b>	<b>111</b>
<b>nodemon</b>	<b>111</b>
<b>nodemon</b>	<b>111</b>
Browsersync	111
.....	111
.....	112
Windows	112
.....	112
.....	112
Grunt.js	113
Gulp.js	113
<b>API</b>	<b>113</b>
<b>28: Node.js</b>	<b>114</b>
Examples	114
Node.js -	114
Node.js -	114
<b>29:</b>	<b>116</b>
.....	116
.....	116
Examples	116
.....	116
<b>JavaScript</b>	<b>116</b>
.....	116
.....	117



<b>Node.js</b> .....	<b>118</b>
.....	119
Async.....	120
.....	120
.....	120
.....	121
.....	121
.....	121
.....	122
<b>30: / Await</b> .....	<b>124</b>
.....	124
Examples.....	124
Try-Catch.....	124
Async / Await.....	125
.....	125
.....	126
<b>31: (MongoDB Mongoose)</b> .....	<b>128</b>
Examples.....	128
.....	128
.....	128
.....	129
.....	129
<b>32: /</b> .....	<b>131</b>
.....	131
Examples.....	131
writeFile writeFileSync.....	131
.....	132
.....	132
.....	132
.....	132
readdir readdirSync.....	133

.....	133
.....	134
.....	134
unlink unlinkSync.....	134
.....	135
.....	135
.....	<b>136</b>
.....	<b>136</b>
.....	137
.....	137
.....	137
.....	138
.....	138
.....	138
.....	139
.....	139
<b>app.js</b> .....	<b>139</b>
.....	140
<b>app.js</b> .....	<b>140</b>
<b>33: - Express</b> .....	<b>141</b>
.....	141
.....	141
.....	141
<b>Examples</b> .....	<b>142</b>
.....	142
.....	142
.....	144
.....	145
.....	<b>145</b>
.....	146
.....	147

EJS.....	147
API JSON ExpressJS.....	148
.....	149
.....	149
Django.....	150
.....	150
.....	151
.....	153
: req res.....	155
POST.....	155
cookie cookie-parser.....	156
Express.....	156
Express.....	157
.....	157
,.....	158
<b>34:</b> .....	<b>159</b>
.....	159
Examples.....	159
.....	159
.....	159
console.log.....	159
console.error.....	159
console.time, console.timeEnd.....	159
.....	160
.....	160
.....	160
.....	160
.....	161
<b>35:</b> .....	<b>162</b>
Examples.....	162
.....	162
<b>36:</b> .....	<b>163</b>

.....	163
.....	163
Examples.....	163
.....	163
.....	165
<b>37:</b> .....	<b>166</b>
.....	166
.....	166
Examples.....	166
.....	166
.....	167
.....	168
<b>38: HTML</b> .....	<b>170</b>
.....	170
Examples.....	170
HTML .....	170
.....	170
server.js.....	170
<b>39: node.js</b> .....	<b>172</b>
.....	172
Examples.....	172
Node.js daemon.....	172
<b>40: Node.js</b> .....	<b>174</b>
Examples.....	174
(CSRF).....	174
SSL / TLS Node.js.....	175
HTTPS.....	176
HTTPS.....	176
1. ....	176
2. ....	177
express.js 3.....	178

<b>41:</b>	.....	<b>179</b>
Examples	.....	179
	.....	179
	.....	179
<b>42:</b>	.....	<b>181</b>
	.....	181
Examples	.....	181
HTTP-	.....	181
	.....	182
,	.....	183
	.....	183
<b>43:</b>	.....	<b>185</b>
Examples	.....	185
- SIGTERM	.....	185
<b>44: MongoDB Node.js / Express.js</b>	.....	<b>186</b>
	.....	186
	.....	186
Examples	.....	186
MongoDB	.....	186
	.....	186
Mongo	.....	187
<b>45: MSSQL</b>	.....	<b>189</b>
	.....	189
	.....	189
Examples	.....	189
SQL . mssql npm	.....	189
<b>46: MySQL</b>	.....	<b>191</b>
	.....	191
Examples	.....	191
	.....	191
	.....	191

.....	191
.....	192
MySQL.....	193
.....	193
.....	194
.....	194
.....	195
<b>47: PostgreSQL.....</b>	<b>196</b>
Examples.....	196
PostgreSQL.....	196
.....	196
<b>48: .....</b>	<b>197</b>
Examples.....	197
, .....	197
<b>49: .....</b>	<b>198</b>
.....	198
Examples.....	198
.....	198
.....	199
Facebook.....	200
.....	201
Google Passport.....	202
<b>50: Browserfiy «» .....</b>	<b>204</b>
Examples.....	204
- file.js.....	204
?.....	204
.....	204
.....	205
?.....	205
<b>51: IISNode - Node.js IIS .....</b>	<b>206</b>
.....	206

<b>/ Pitfall</b> .....	<b>206</b>
.....	206
Examples.....	206
.....	206
.....	207
Hello World Express.....	207
<b>Project Structure</b> .....	<b>207</b>
<b>server.js - -</b> .....	<b>207</b>
<b>Web.config</b> .....	<b>208</b>
.....	208
IISNode.....	208
URL-.....	208
IIS .....	209
Socket.io IISNode.....	210
<b>52: WebSocket Node.JS</b> .....	<b>212</b>
Examples.....	212
WebSocket.....	212
WebSocket .....	212
WebSocket WebSocket Server.....	212
WebSocket.....	212
<b>53:</b> .....	<b>214</b>
.....	214
Examples.....	214
TextFile .....	214
.....	215
/ .....	216
?.....	216
<b>54: Node.js</b> .....	<b>219</b>
Examples.....	219
HTTP-.....	219
.....	219

<b>55: Nodejs</b> .....	<b>221</b>
.....	.221
Examples.....	221
.....	.221
<b>2009</b> .....	<b>221</b>
<b>2010</b> .....	<b>221</b>
<b>2011</b> .....	<b>221</b>
<b>2012</b> .....	<b>222</b>
<b>2013</b> .....	<b>222</b>
<b>2014</b> .....	<b>222</b>
<b>2015</b> .....	<b>222</b>
Q1.....	222
Q2.....	223
Q3.....	223
Q4.....	223
<b>2016</b> .....	<b>223</b>
Q1.....	223
Q2.....	223
Q3.....	223
Q4.....	223
<b>56:</b> .....	<b>225</b>
Examples.....	225
.....	.225
.....	.225
.....	<b>225</b>
<b>57:</b> .....	<b>227</b>
.....	.227
.....	.227
Examples.....	227
, .....	227
.....	.228



<b>58: Node.js STDIN STDOUT -</b>	<b>230</b>
.....	230
Examples.....	230
.....	230
<b>59: JS</b>	<b>231</b>
.....	231
Examples.....	231
i18n    node.js.....	231
<b>60: ajax- Express.JS</b>	<b>233</b>
Examples.....	233
AJAX.....	233
<b>61: NodeJs</b>	<b>235</b>
.....	235
.....	235
Examples.....	235
- -.....	235
<b>62:</b>	<b>240</b>
.....	240
Examples.....	240
.....	240
<b>Macos</b>	<b>240</b>
Homebrew.....	240
MacPorts.....	240
PATH.....	240
<b>Windows</b>	<b>240</b>
.....	240
.....	241
<b>Linux</b>	<b>241</b>
Debian / Ubuntu.....	241
CentOS / Fedora / RHEL.....	241
.....	241

.....	242
.....	242
.....	<b>242</b>
.....	242
.....	242
.....	242
<b>Post Install</b> .....	<b>242</b>
.....	242
.....	243
<b>63:</b> .....	<b>245</b>
.....	245
.....	245
Examples.....	245
.....	245
.....	246
<b>64:</b> .....	<b>248</b>
.....	248
.....	248
Examples.....	249
MongoDB.....	249
MongoClient Connect().....	249
.....	250
insertOne().....	250
.....	250
find().....	251
.....	251
updateOne().....	251
.....	252
deleteOne().....	252
.....	253
deleteMany().....	253
.....	253

, .....	254
<b>65:</b> .....	<b>255</b>
Examples.....	255
MongoDB Mongoose.....	255
MongoDB Mongoose Express.js.....	255
.....	<b>255</b>
.....	<b>256</b>
.....	<b>257</b>
MongoDB Mongoose Express.js.....	257
.....	<b>257</b>
.....	<b>258</b>
.....	<b>259</b>
MongoDB Mongoose, Express.js \$ text Operator.....	260
.....	<b>260</b>
.....	<b>260</b>
.....	<b>261</b>
.....	262
.....	264
mongodb, .....	264
.....	<b>264</b>
.....	<b>265</b>
.....	<b>266</b>
<b>66:</b> .....	<b>268</b>
Examples.....	268
, ().....	268
<b>67:</b> .....	<b>269</b>
.....	269
.....	269
Examples.....	269
.....	269

<b>68:</b>	<b>272</b>
.....	272
.....	272
.....	274
Examples.....	274
.....	274
.....	<b>274</b>
<b>NPM</b> .....	<b>275</b>
.....	275
.....	<b>277</b>
<b>NPM -</b> .....	<b>278</b>
.....	279
.....	279
.....	280
.....	281
.....	282
.....	282
.....	283
.....	284
npm .....	284
.....	285
.....	285
.....	286
.....	286
.....	286
.....	286
.....	287
.....	287
<b>69: Bluebird</b> .....	<b>288</b>
Examples.....	288
Promises.....	288
.....	288

()	288
(Promise.using)	289
	289
<b>70: POST Node.js</b>	<b>290</b>
	290
Examples	290
node.js-, POST	290
<b>71:</b>	<b>292</b>
Examples	292
Node.Js	292
	294
	294
	294
	295
<b>72:</b>	<b>296</b>
Examples	296
	296
	297
setTimeout promisified	297
<b>73: Node.js</b>	<b>298</b>
Examples	298
Node.js	298
<b>74: Node.js</b>	<b>299</b>
Examples	299
Core node.js	299
	299
	299
	300
	301
<b>75: -</b>	<b>303</b>
Examples	303

- GCM (Google Cloud Messaging System)	303
<b>76:</b>	<b>306</b>
Examples	306
fs pipe	306
fluent-ffmpeg	307
<b>77:</b>	<b>308</b>
Examples	308
.....	308
<b>78: Mongodb</b>	<b>309</b>
.....	309
.....	309
Examples	309
mongoDB Node.JS	309
mongoDB Node.JS	309
<b>79:</b>	<b>311</b>
Examples	311
PM2	311
.....	311
Forever	312
nohup	313
Forever	313
<b>80:</b>	<b>315</b>
Examples	315
.....	315
<b>81: Windows node.js</b>	<b>319</b>
.....	319
Examples	319
.....	319
.....	<b>319</b>
.....	<b>319</b>
<b>82: Node.js</b>	<b>321</b>

Examples.....	321
Event Loop.....	321
.....	<b>321</b>
.....	<b>321</b>
.....	<b>322</b>
maxSockets.....	322
.....	<b>322</b>
.....	<b>323</b>
.....	<b>323</b>
.....	<b>323</b>
gzip.....	323
<b>83: API CRUD REST.....</b>	<b>325</b>
Examples.....	325
REST API CRUD Express 3+.....	325
<b>84: Mysql.....</b>	<b>326</b>
Examples.....	326
.....	326
<b>85: .....</b>	<b>328</b>
Examples.....	328
() .....	328
.....	328
<b>86: Node.js .....</b>	<b>329</b>
Examples.....	329
NODE_ENV = "" .....	329
.....	<b>329</b>
.....	<b>329</b>
.....	330
PM2.....	330
PM2.....	331
.....	332
Forever.....	332

/ , dev, qa, .....	333
.....	334
<b>87: Node.js .....</b>	<b>336</b>
Examples.....	336
PM2 .....	336
<b>88: .....</b>	<b>338</b>
Examples.....	338
Nunjucks.....	338
<b>89: NodeJS.....</b>	<b>340</b>
Examples.....	340
, !.....	340
<b>90: Socket.io.....</b>	<b>341</b>
Examples.....	341
",!" .....	341
<b>91: -.....</b>	<b>342</b>
Examples.....	342
/w , jQuery Jade.....	342
<b>92: .....</b>	<b>344</b>
.....	344
Examples.....	344
.....	344
CORS.....	345
<b>93: .....</b>	<b>346</b>
Examples.....	346
async.....	346
<b>94: API Node.js.....</b>	<b>348</b>
Examples.....	348
GET api Express.....	348
POST api Express.....	348
<b>95: Node.js, .....</b>	<b>350</b>
.....	350



Examples.....	350
Bluebird.....	350
<b>96: TCP.....</b>	<b>353</b>
Examples.....	353
TCP-.....	353
TCP.....	353
<b>97: Arduino nodeJs.....</b>	<b>355</b>
.....	355
Examples.....	355
Js Arduino .....	355
js.....	355
Arduino.....	356
.....	356
<b>98: .....</b>	<b>358</b>
Examples.....	358
.....	358
process.argv.....	358
/ , dev, qa, .....	359
.....	360
<b>99: Route-Controller-Service ExpressJS.....</b>	<b>362</b>
Examples.....	362
--- .....	362
---.....	362
<b>user.model.js.....</b>	<b>362</b>
<b>user.routes.js.....</b>	<b>362</b>
<b>user.controllers.js.....</b>	<b>363</b>
<b>user.services.js.....</b>	<b>363</b>
<b>100: .....</b>	<b>364</b>
.....	364
.....	364
Examples.....	364

nodejs MVC API.....	364
<b>101:</b> .....	<b>367</b>
Examples.....	367
.....	367
Mocha ( ).....	367
().....	367
Mocha Asynchronous ( / ).....	367
<b>102: Node.js</b> .....	<b>369</b>
Examples.....	369
Node.js Mac OSX.....	369
Node.js Windows.....	369
<b>103: Node.JS</b> .....	<b>370</b>
Examples.....	370
NodeJS.....	370
IntelliJ / Webstorm.....	370
Linux.....	371
<b>104: Node.js</b> .....	<b>373</b>
.....	373
Examples.....	373
Error.....	373
.....	373
... catch block.....	374
<b>105: API:</b> .....	<b>376</b>
Examples.....	376
:	376
<b>106: Node.js</b> .....	<b>378</b>
Examples.....	378
Node.js Ubuntu.....	378
<b>apt</b> .....	<b>378</b>
<b>(, LTS 6.x) no</b> .....	<b>378</b>
Node.js Windows.....	378

Node Version Manager (nvm).....	379
Node.js APT.....	380
Node.js Mac .....	381
<b>Homebrew.....</b>	<b>381</b>
<b>MacPorts.....</b>	<b>381</b>
MacOS X Installer.....	382
, .....	<b>382</b>
Node.js Pl.....	382
Fish Shell !.....	382
Node.js Centos, RHEL Fedora.....	383
Node.js n.....	384
<b>107: .....</b>	<b>386</b>
Examples.....	386
multer.....	386
:.....	<b>387</b>
:.....	<b>387</b>
.....	388
<b>108: .....</b>	<b>389</b>
.....	389
Examples.....	389
.....	389
<b>109: .....</b>	<b>393</b>
.....	393
Examples.....	393
GruntJs.....	393
gruntplugins.....	394
<b>110: node.js.....</b>	<b>396</b>
Examples.....	396
node.js.....	396
ES6.....	397
ES6.....	398

<b>111:</b> .....	<b>399</b>
.....	399
Examples.....	399
.....	399
hello-world.js.....	400
.....	401
.....	402
.....	403
node_modules.....	403
.....	404
.....	<b>406</b>

---

# Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: [node-js](#)

It is an unofficial and free Node.js ebook created for educational purposes. All the content is extracted from [Stack Overflow Documentation](#), which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Node.js.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to [info@zzzprojects.com](mailto:info@zzzprojects.com)

---

# глава 1: Начало работы с Node.js

## замечания

Node.js представляет собой основанную на событиях, неблокирующую асинхронную инфраструктуру ввода-вывода, которая использует движок Google V8 JavaScript. Он используется для разработки приложений, которые сильно используют возможность запуска JavaScript как на клиенте, так и на стороне сервера и, следовательно, извлекают выгоду из повторного использования кода и отсутствия переключения контекста. Это open-source и кросс-платформенный. Приложения Node.js написаны на чистом JavaScript и могут выполняться в среде Node.js на Windows, Linux и т. Д. ...

## Версии

Версия	Дата выхода
v8.2.1	2017-07-20
v8.2.0	2017-07-19
v8.1.4	2017-07-11
v8.1.3	2017-06-29
v8.1.2	2017-06-15
v8.1.1	2017-06-13
v8.1.0	2017-06-08
v8.0.0	2017-05-30
v7.10.0	2017-05-02
v7.9.0	2017-04-11
v7.8.0	2017-03-29
v7.7.4	2017-03-21
v7.7.3	2017-03-14
v7.7.2	2017-03-08
v7.7.1	2017-03-02

<b>Версия</b>	<b>Дата выхода</b>
v7.7.0	2017-02-28
v7.6.0	2017-02-21
v7.5.0	2017-01-31
v7.4.0	2017-01-04
v7.3.0	2016-12-20
v7.2.1	2016-12-06
v7.2.0	2016-11-22
v7.1.0	2016-11-08
версия 7.0.0	2016-10-25
v6.11.0	2017-06-06
v6.10.3	2017-05-02
v6.10.2	2017-04-04
v6.10.1	2017-03-21
v6.10.0	2017-02-21
v6.9.5	2017-01-31
v6.9.4	2017-01-05
v6.9.3	2017-01-05
v6.9.2	2016-12-06
v6.9.1	2016-10-19
v6.9.0	2016-10-18
v6.8.1	2016-10-14
v6.8.0	2016-10-12
v6.7.0	2016-09-27
v6.6.0	2016-09-14
v6.5.0	2016-08-26

<b>Версия</b>	<b>Дата выхода</b>
v6.4.0	2016-08-12
v6.3.1	2016-07-21
v6.3.0	2016-07-06
v6.2.2	2016-06-16
v6.2.1	2016-06-02
v6.2.0	2016-05-17
v6.1.0	2016-05-05
v6.0.0	2016-04-26
v5.12.0	2016-06-23
v5.11.1	2016-05-05
v5.11.0	2016-04-21
v5.10.1	2016-04-05
v5.10	2016-04-01
V5.9	2016-03-16
версии 5.8	2016-03-09
v5.7	2016-02-23
v5.6	2016-02-09
v5.5	2016-01-21
v5.4	2016-01-06
v5.3	2015-12-15
v5.2	2015-12-09
v5.1	2015-11-17
v5.0	2015-10-29
V4.4	2016-03-08
v4.3	2016-02-09



<b>Версия</b>	<b>Дата выхода</b>
v4.2	2015-10-12
v4.1	2015-09-17
v4.0	2015-09-08
io.js v3.3	2015-09-02
io.js v3.2	2015-08-25
io.js v3.1	2015-08-19
io.js v3.0	2015-08-04
io.js v2.5	2015-07-28
io.js v2.4	2015-07-17
io.js v2.3	2015-06-13
io.js v2.2	2015-06-01
io.js v2.1	2015-05-24
io.js v2.0	2015-05-04
io.js v1.8	2015-04-21
io.js v1.7	2015-04-17
io.js v1.6	2015-03-20
io.js v1.5	2015-03-06
io.js v1.4	2015-02-27
io.js v1.3	2015-02-20
io.js v1.2	2015-02-11
io.js v1.1	2015-02-03
io.js v1.0	2015-01-14
v0.12	2016-02-09
v0.11	2013-03-28
V0.10	2013-03-11

Версия	Дата выхода
v0.9	2012-07-20
v0.8	2012-06-22
v0.7	2012-01-17
v0.6	2011-11-04
v0.5	2011-08-26
v0.4	2011-08-26
v0.3	2011-08-26
v0.2	2011-08-26
v0.1	2011-08-26

## Examples

### Сервер HTTP Hello World

Сначала [установите Node.js](#) для своей платформы.

В этом примере мы создадим HTTP-сервер, прослушивающий порт 1337, который отправляет `Hello, World!` в браузер. Обратите внимание, что вместо использования порта 1337 вы можете использовать любой номер порта по вашему выбору, который в настоящее время не используется какой-либо другой услугой.

Модуль `http` - это **ОСНОВНОЙ МОДУЛЬ** Node.js (модуль, входящий в состав источника Node.js, который не требует установки дополнительных ресурсов). Модуль `http` предоставляет функциональные возможности для создания HTTP-сервера с использованием `http.createServer()`. Чтобы создать приложение, создайте файл, содержащий следующий код JavaScript.

```
const http = require('http'); // Loads the http module

http.createServer((request, response) => {

  // 1. Tell the browser everything is OK (Status code 200), and the data is in plain text
  response.writeHead(200, {
    'Content-Type': 'text/plain'
  });

  // 2. Write the announced text to the body of the page
  response.write('Hello, World!\n');

  // 3. Tell the server that all of the response headers and body have been sent
```

```
response.end();

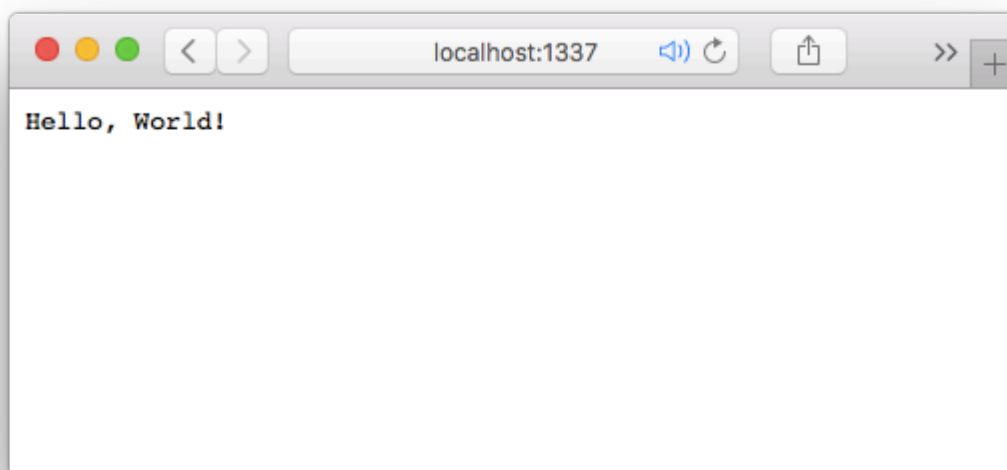
}).listen(1337); // 4. Tells the server what port to be on
```

Сохраните файл с любым именем файла. В этом случае, если мы назовем это `hello.js` мы можем запустить приложение, перейдя в каталог, в котором находится файл, и используя следующую команду:

```
node hello.js
```

После этого созданный сервер можно получить с помощью URL <http://localhost:1337> или <http://127.0.0.1:1337> в браузере.

Простая веб-страница появится с текстом «Hello, World!» Вверху, как показано на скриншоте ниже.



[Редактируемый онлайн-пример.](#)

## Командная строка Hello World

Node.js также можно использовать для создания утилит командной строки. В приведенном ниже примере читается первый аргумент из командной строки и выводится сообщение Hello.

Чтобы запустить этот код в системе Unix:

1. Создайте новый файл и вставьте код ниже. Имя файла не имеет значения.
2. Сделать этот файл исполняемым с помощью `chmod 700 FILE_NAME`
3. Запустите приложение с помощью `./APP_NAME David`

В Windows вы делаете шаг 1 и запускаете его с `node APP_NAME David`

```
#!/usr/bin/env node

'use strict';

/*
  The command line arguments are stored in the `process.argv` array,
  which has the following structure:
  [0] The path of the executable that started the Node.js process
  [1] The path to this application
  [2-n] the command line arguments

  Example: [ '/bin/node', '/path/to/yourscript', 'arg1', 'arg2', ... ]
  src: https://nodejs.org/api/process.html#process_process_argv
*/

// Store the first argument as username.
var username = process.argv[2];

// Check if the username hasn't been provided.
if (!username) {

  // Extract the filename
  var appName = process.argv[1].split(require('path').sep).pop();

  // Give the user an example on how to use the app.
  console.error('Missing argument! Example: %s YOUR_NAME', appName);

  // Exit the app (success: 0, error: 1).
  // An error will stop the execution chain. For example:
  // ./app.js && ls      -> won't execute ls
  // ./app.js David && ls -> will execute ls
  process.exit(1);
}

// Print the message to the console.
console.log('Hello %s!', username);
```

## Установка и запуск Node.js

Для начала установите Node.js на компьютер разработки.

**Windows:** [перейдите на страницу](#) загрузки и загрузите / запустите программу установки.

**Mac:** [перейдите на страницу](#) загрузки и загрузите / запустите программу установки. Кроме того, вы можете установить узел через Homebrew с помощью `brew install node`. Homebrew - это командный пакет для Macintosh, и больше информации об этом можно найти на [веб-сайте Homebrew](#).

**Linux:** следуйте инструкциям для своего дистрибутива на [странице установки командной строки](#).

# Запуск программы узла

Чтобы запустить программу Node.js, просто запустите `node app.js` или `nodejs app.js`, где `app.js` является именем файла исходного кода вашего узла. Вам не нужно включать суффикс `.js` для узла, чтобы найти сценарий, который вы хотите запустить.

В качестве альтернативы в операционных системах на базе UNIX программа Node может быть выполнена как сценарий терминала. Для этого нужно начинать с `shebang`, указывающего на интерпретатор узла, например, узел `#!/usr/bin/env node`. Файл также должен быть установлен как исполняемый файл, который можно выполнить с помощью `chmod`. Теперь скрипт можно запустить из командной строки.

## Развертывание приложения онлайн

Когда вы развертываете свое приложение в (размещенной в Node.js) размещенной среде, эта среда обычно предлагает переменную окружения `PORT` которую вы можете использовать для запуска вашего сервера. Изменение номера порта на `process.env.PORT` позволяет получить доступ к приложению.

Например,

```
http.createServer(function(request, response) {
  // your server code
}).listen(process.env.PORT);
```

Кроме того, если вы хотите получить доступ к этому автономному устройству во время отладки, вы можете использовать это:

```
http.createServer(function(request, response) {
  // your server code
}).listen(process.env.PORT || 3000);
```

где `3000` - номер порта в автономном режиме.

## Отладка вашего приложения NodeJS

Вы можете использовать инспектора узлов. Запустите эту команду, чтобы установить ее через `npm`:

```
npm install -g node-inspector
```

Затем вы можете отлаживать свое приложение, используя

```
node-debug app.js
```

Репозиторий Github можно найти здесь: <https://github.com/node-inspector/node-inspector>

---

## Отладка изначально

Вы также можете отлаживать node.js изначально, запустив его следующим образом:

```
node debug your-script.js
```

Чтобы остановить ваш отладчик точно в нужной строке кода, используйте это:

```
debugger;
```

Для получения дополнительной информации см. [Здесь](#) .

В node.js 8 используйте следующую команду:

```
node --inspect-brk your-script.js
```

Затем откройте `about://inspect` в последней версии Google Chrome и выберите свой сценарий узла, чтобы получить отладочную версию DevTools от Chrome.

## Hello World with Express

В следующем примере Express используется для создания HTTP-сервера, прослушивающего порт 3000, который отвечает «Hello, World!». Экспресс является широко используемой веб-картой, которая полезна для создания HTTP API.

Сначала создайте новую папку, например `myApp` . Перейдите в `myApp` и создайте новый файл JavaScript, содержащий следующий код (например, `hello.js` его `hello.js` ). Затем установите экспресс-модуль, используя `npm install --save express` из командной строки. *Подробнее об установке пакетов см. В этой документации* .

```
// Import the top-level function of express
const express = require('express');

// Creates an Express application using the top-level function
const app = express();

// Define port number as 3000
const port = 3000;

// Routes HTTP GET requests to the specified path "/" with the specified callback function
app.get('/', function(request, response) {
  response.send('Hello, World!');
});

// Make the app listen on port 3000
```

```
app.listen(port, function() {
  console.log('Server listening on http://localhost:' + port);
});
```

В командной строке выполните следующую команду:

```
node hello.js
```

Откройте ваш браузер и перейдите по `http://localhost:3000` или `http://127.0.0.1:3000` чтобы увидеть ответ.

Для получения дополнительной информации о структуре Express вы можете проверить раздел « [Веб-приложения с экспрессом](#) »

## Приветственная базовая маршрутизация

Как только вы поймете, как создать [HTTP-сервер](#) с узлом, важно понять, как заставить его «делать» вещи на основе пути, к которому пользователь перешел. Это явление называется «маршрутизация».

Самый простой пример этого - проверить `if (request.url === 'some/path/here')`, а затем вызвать функцию, которая отвечает новым файлом.

Пример этого можно увидеть здесь:

```
const http = require('http');

function index (request, response) {
  response.writeHead(200);
  response.end('Hello, World!');
}

http.createServer(function (request, response) {

  if (request.url === '/') {
    return index(request, response);
  }

  response.writeHead(404);
  response.end(http.STATUS_CODES[404]);

}).listen(1337);
```

Если вы продолжите определять свои «маршруты», подобные этому, вы получите одну массивную функцию обратного вызова, и мы не хотим такого гигантского беспорядка, поэтому давайте посмотрим, можем ли мы это очистить.

Во-первых, давайте сохраним все наши маршруты в объекте:

```
var routes = {
```

```
'/': function index (request, response) {
  response.writeHead(200);
  response.end('Hello, World!');
},
'/foo': function foo (request, response) {
  response.writeHead(200);
  response.end('You are now viewing "foo"');
}
}
```

Теперь, когда мы сохранили 2 маршрута в объекте, теперь мы можем проверить их в нашем основном обратном вызове:

```
http.createServer(function (request, response) {

  if (request.url in routes) {
    return routes[request.url](request, response);
  }

  response.writeHead(404);
  response.end(http.STATUS_CODES[404]);

}).listen(1337);
```

Теперь каждый раз, когда вы пытаетесь перемещаться по сайту, он проверяет наличие этого пути в ваших маршрутах и вызывает соответствующую функцию. Если маршрут не найден, сервер ответит 404 (не найден).

И там у вас есть - маршрутизация с помощью API HTTP Server очень проста.

## TLS Socket: сервер и клиент

Единственными существенными отличиями между этим и обычным TCP-соединением являются закрытый ключ и общедоступный сертификат, который вам нужно будет установить в объект опции.

---

## Как создать ключ и сертификат

Первым шагом в этом процессе безопасности является создание закрытого ключа. И что это за секретный ключ? В принципе, это набор случайных шумов, которые используются для шифрования информации. Теоретически вы можете создать один ключ и использовать его для шифрования всего, что хотите. Но лучше всего иметь разные ключи для конкретных вещей. Потому что, если кто-то украл ваш секретный ключ, это похоже на то, что кто-то украл ваши ключи от дома. Представьте себе, если вы использовали тот же ключ для блокировки своего автомобиля, гаража, офиса и т. Д.

```
openssl genrsa -out private-key.pem 1024
```

После того, как у нас есть наш закрытый ключ, мы можем создать CSR (запрос подписи



сертификата), который является нашим запросом на то, чтобы секретный ключ был подписан модным авторитетом. Вот почему вы должны вводить информацию, связанную с вашей компанией. Эта информация будет видна авторитетом подписания и использована для проверки вас. В нашем случае не имеет значения, что вы набираете, поскольку на следующем этапе мы сами подпишем наш сертификат.

```
openssl req -new -key private-key.pem -out csr.pem
```

Теперь, когда мы закончили работу с бумагой, пришло время притвориться, что мы классный авторитет подписания.

```
openssl x509 -req -in csr.pem -signkey private-key.pem -out public-cert.pem
```

Теперь, когда у вас есть закрытый ключ и публичный сертификат, вы можете установить безопасное соединение между двумя приложениями NodeJS. И, как вы можете видеть в примере кода, это очень простой процесс.

---

## Важный!

Поскольку мы создали публичный сертификат, честно говоря, наш сертификат бесполезен, потому что мы - ничто. Сервер NodeJS не будет доверять такому сертификату по умолчанию, и поэтому нам нужно сказать ему, чтобы он действительно доверял нашему сертификату со следующей опцией `rejectUnauthorized: false`. **Очень важно** : никогда не устанавливайте эту переменную в `true` в производственной среде.

---

## Сервер TLS Socket

```
'use strict';

var tls = require('tls');
var fs = require('fs');

const PORT = 1337;
const HOST = '127.0.0.1'

var options = {
  key: fs.readFileSync('private-key.pem'),
  cert: fs.readFileSync('public-cert.pem')
};

var server = tls.createServer(options, function(socket) {

  // Send a friendly message
  socket.write("I am the server sending you a message.");

  // Print the data that we received
  socket.on('data', function(data) {

    console.log('Received: %s [it is %d bytes long]',
```

```

        data.toString().replace(/(\n)/gm, ""),
        data.length);

    });

    // Let us know when the transmission is over
    socket.on('end', function() {

        console.log('EOT (End Of Transmission)');

    });

});

// Start listening on a specific port and address
server.listen(PORT, HOST, function() {

    console.log("I'm listening at %s, on port %s", HOST, PORT);

});

// When an error occurs, show it.
server.on('error', function(error) {

    console.error(error);

    // Close the connection after the error occurred.
    server.destroy();

});

```

## TLS Socket Client

```

'use strict';

var tls = require('tls');
var fs = require('fs');

const PORT = 1337;
const HOST = '127.0.0.1'

// Pass the certs to the server and let it know to process even unauthorized certs.
var options = {
    key: fs.readFileSync('private-key.pem'),
    cert: fs.readFileSync('public-cert.pem'),
    rejectUnauthorized: false
};

var client = tls.connect(PORT, HOST, options, function() {

    // Check if the authorization worked
    if (client.authorized) {
        console.log("Connection authorized by a Certificate Authority.");
    } else {
        console.log("Connection not authorized: " + client.authorizationError)
    }

    // Send a friendly message

```

```
    client.write("I am the client sending you a message.");
  });

  client.on("data", function(data) {

    console.log('Received: %s [it is %d bytes long]',
      data.toString().replace(/\n/gm, ""),
      data.length);

    // Close the connection after receiving the message
    client.end();

  });

  client.on('close', function() {

    console.log("Connection closed");

  });

  // When an error occurs, show it.
  client.on('error', function(error) {

    console.error(error);

    // Close the connection after the error occurred.
    client.destroy();

  });
});
```

## Hello World в REPL

При вызове без аргументов Node.js запускает REPL (Read-Eval-Print-Loop), также известный как « *оболочка узла* ».

В командной строке введите `node .`

```
$ node
>
```

В командной строке узла `>` введите «Hello World!».

```
$ node
> "Hello World!"
'Hello World!'
```

## Основные модули

Node.js - движок Javascript (движок V8 для Google для Chrome, написанный на C ++), который позволяет запускать Javascript за пределами браузера. Хотя для расширения возможностей Node доступны многочисленные библиотеки, в комплект поставки входит набор *основных модулей*, реализующих основные функции.

В настоящее время в узле имеется 34 основных модуля:

```
[ 'assert',
  'buffer',
  'c/c++_addons',
  'child_process',
  'cluster',
  'console',
  'crypto',
  'deprecated_apis',
  'dns',
  'domain',
  'Events',
  'fs',
  'http',
  'https',
  'module',
  'net',
  'os',
  'path',
  'punycode',
  'querystring',
  'readline',
  'repl',
  'stream',
  'string_decoder',
  'timers',
  'tls_(ssl)',
  'tracing',
  'tty',
  'dgram',
  'url',
  'util',
  'v8',
  'vm',
  'zlib' ]
```

Этот список был получен из API документации Node <https://nodejs.org/api/all.html> (файл JSON: <https://nodejs.org/api/all.json> ).

---

## Все основные модули на первый взгляд

### утверждать

Модуль `assert` предоставляет простой набор тестов утверждения, которые можно использовать для проверки инвариантов.

### буфер

До введения `TypedArray` в ECMAScript 2015 (ES6) язык JavaScript не имел механизма для чтения или обработки потоков двоичных данных. Класс `Buffer` был представлен как часть API Node.js, чтобы можно было взаимодействовать с октетными потоками в контексте таких потоков, как потоки TCP и операции с файловой системой.

Теперь, когда `TypedArray` был добавлен в ES6, класс `Buffer` реализует `Uint8Array` API таким образом, который более оптимизирован и подходит для случаев использования Node.js.

## C / C ++ \_ аддоны

Node.js Addons - это динамически связанные общие объекты, написанные на C или C ++, которые могут быть загружены в Node.js с помощью функции `require()` и использованы так же, как если бы они были обычным модулем Node.js. Они используются, прежде всего, для обеспечения интерфейса между JavaScript, запущенным в библиотеках Node.js и C / C ++.

### child\_process

Модуль `child_process` предоставляет возможность генерировать дочерние процессы способом, похожим, но не идентичным, для `fork()` (3).

### кластер

Один экземпляр Node.js работает в одном потоке. Чтобы воспользоваться преимуществами многоядерных систем, пользователь иногда захочет запустить кластер из процессов Node.js для обработки нагрузки. Модуль кластера позволяет вам легко создавать дочерние процессы, которые используют общий доступ к портам сервера.

### приставка

`console` модуль предоставляет простую консоль отладки, похожую на механизм консоли JavaScript, предоставляемый веб-браузерами.

### крипто-

`crypto` модуль обеспечивает криптографическую функциональность, которая включает в себя набор оберток для хэша OpenSSL, HMAC, шифр, расшифровывать, подписывать и проверять функции.

### deprecated\_apis

Node.js может испортить API, если: (a) использование API считается небезопасным; (b) был предоставлен улучшенный альтернативный API или (c) нарушение изменений в API ожидается в будущем крупном выпуске,

## DNS

Модуль `dns` содержит функции, принадлежащие двум различным категориям:

1. Функции, которые используют базовые средства операционной системы для выполнения разрешения имен, и которые не обязательно выполняют какую-либо сетевую связь. Эта категория содержит только одну функцию: `dns.lookup()`.
2. Функции, которые подключаются к реальному DNS-серверу для выполнения

разрешения имен и *всегда* используют сеть для выполнения DNS-запросов. Эта категория содержит все функции в модуле `dns` *кроме* `dns.lookup()` .

## Домен

**Этот модуль ожидает отмены** . После того, как API-интерфейс замены будет завершен, этот модуль будет полностью устарел. У большинства конечных пользователей **не** должно быть причин использовать этот модуль. Пользователи, которые абсолютно должны обладать функциональностью, которую предоставляют домены, могут на это рассчитывать, но в будущем они должны будут перейти к другому решению.

## События

Большая часть основного API-интерфейса Node.js построена вокруг идиоматической асинхронной архитектуры, управляемой событиями, в которой определенные типы объектов (называемые «эмиттеры») периодически излучают именованные события, которые вызывают функции-объекты («слушатели»).

## ФС

Файловый ввод-вывод обеспечивается простыми оболочками вокруг стандартных функций POSIX. Для использования этого модуля `require('fs')` . Все методы имеют асинхронные и синхронные формы.

## HTTP

Интерфейсы HTTP в Node.js предназначены для поддержки многих функций протокола, которые традиционно трудно использовать. В частности, большие, возможно, закодированные в блоке сообщения. Интерфейс не требует буферизации целых запросов или ответов - пользователь может передавать данные.

## HTTPS

HTTPS - это протокол HTTP через TLS / SSL. В Node.js это реализовано как отдельный модуль.

## Модуль

Node.js имеет простую систему загрузки модуля. В Node.js файлы и модули соответствуют друг другу (каждый файл рассматривается как отдельный модуль).

## Сеть

`net` модуль предоставляет асинхронную сеть обертки. Он содержит функции для создания серверов и клиентов (называемых потоками). Вы можете включить этот модуль с

```
require('net');
```

## Операционные системы

Модуль `os` предоставляет ряд методов, связанных с операционной системой.

## дорожка

Модуль `path` предоставляет утилиты для работы с файловыми и каталогами.

## Punycode

**Версия модуля `punycode`, входящего в состав `Node.js`, устарела .**

## Строка запроса

Модуль `querystring` предоставляет утилиты для синтаксического анализа и форматирования строк запроса URL.

## Readline

Модуль `readline` предоставляет интерфейс для чтения данных из `Readable stream` (например, `process.stdin`) по одной строке за раз.

## РЕПЛ

Модуль `repl` предоставляет реализацию Read-Eval-Print-Loop (REPL), которая доступна как в виде отдельной программы, так и в других приложениях.

## ПОТОК

Поток представляет собой абстрактный интерфейс для работы с потоковыми данными в `Node.js`. Модуль `stream` предоставляет базовый API, который упрощает сбор объектов, реализующих интерфейс потока.

Существует множество объектов потока, предоставляемых `Node.js`. Например, запрос на HTTP-сервер и `process.stdout` являются экземплярами потока.

## string\_decoder

Модуль `string_decoder` предоставляет API для декодирования объектов `Buffer` в строки таким образом, который сохраняет кодированные многобайтовые символы UTF-8 и UTF-16.

## таймеры

Модуль `timer` предоставляет глобальный API для функций планирования, которые будут вызываться в некоторый будущий период времени. Поскольку функции таймера являются глобальными, нет необходимости `require('timers')` использовать API.

Функции таймера в `Node.js` реализуют аналогичный API как API таймеров, предоставляемый веб-браузерами, но используют другую внутреннюю реализацию, которая построена вокруг [цикла событий Node.js](#).

## tls\_ (SSL)

Модуль `tls` обеспечивает реализацию протоколов безопасности транспортного уровня (TLS) и протокола Secure Socket Layer (SSL), которые построены поверх OpenSSL.

## трассировка

Trace Event предоставляет механизм для централизации информации трассировки, созданной V8, ядром узла и кодом пользовательского пространства.

Трассировку можно включить, передав `--trace-events-enabled` при запуске приложения Node.js.

## TTY

Модуль `tty` предоставляет `tty.ReadStream` и `tty.WriteStream`. В большинстве случаев нет необходимости или возможно использовать этот модуль напрямую.

## dgram

Модуль `dgram` обеспечивает реализацию сокетов UDP Datagram.

## URL

Модуль `url` предоставляет утилиты для разрешения URL-адресов и анализа.

## Util

Модуль `util` в первую очередь предназначен для поддержки собственных внутренних API-интерфейсов Node.js. Однако многие утилиты также полезны для разработчиков приложений и модулей.

## v8

Модуль `v8` предоставляет API-интерфейсы, специфичные для версии **V8**, встроенной в двоичный файл Node.js.

*Примечание*. API и реализация могут быть изменены в любое время.

## В.М.

Модуль `vm` предоставляет API для компиляции и запуска кода в контекстах виртуальной машины V8. Код JavaScript может быть скомпилирован и запущен немедленно или скомпилирован, сохранен и запущен позже.

*Примечание*. Модуль `vm` не является механизмом безопасности. **Не используйте его для запуска ненадежного кода**.

## Zlib



Модуль `zlib` обеспечивает функции сжатия, реализованные с использованием Gzip и Deflate / Inflate.

## Как запустить базовый веб-сервер HTTPS!

После установки в вашей системе узла `node.js` вы можете просто выполнить описанную ниже процедуру, чтобы получить базовый веб-сервер с поддержкой как HTTP, так и HTTPS!

---

---

### Шаг 1. Создание центра сертификации

1. создайте папку, в которой вы хотите сохранить свой ключ и сертификат:

```
mkdir conf
```

- 
2. перейдите в этот каталог:

```
cd conf
```

- 
3. возьмите этот файл `ca.cnf` для использования в качестве ярлыка конфигурации:

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/ca.cnf
```

- 
4. создайте новый центр сертификации, используя эту конфигурацию:

```
openssl req -new -x509 -days 9999 -config ca.cnf -keyout ca-key.pem -out ca-cert.pem
```

- 
5. теперь, когда у нас есть наш `ca-key.pem` сертификации в `ca-key.pem` и `ca-cert.pem`, давайте сгенерируем закрытый ключ для сервера:

```
openssl genrsa -out key.pem 4096
```

- 
6. захватите этот файл `server.cnf` для использования в качестве ярлыка конфигурации:

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/server.cnf
```

- 
7. сгенерировать запрос подписи сертификата с использованием этой конфигурации:

```
openssl req -new -config server.cnf -key key.pem -out csr.pem
```

- 
8. Подпишите запрос:

```
openssl x509 -req -extfile server.cnf -days 999 -passin "pass:password" -in csr.pem -CA ca-
```

```
cert.pem -CAkey ca-key.pem -CAcreateserial -out cert.pem
```

---

## Шаг 2. Установите сертификат как корневой сертификат.

1. скопируйте свой сертификат в папку корневых сертификатов:

```
sudo cp ca-crt.pem /usr/local/share/ca-certificates/ca-crt.pem
```

---

2. обновить магазин CA:

```
sudo update-ca-certificates
```

---

## Шаг 3. Запуск сервера узла.

Во-первых, вы хотите создать файл `server.js`, содержащий ваш фактический код сервера.

Минимальная настройка для HTTPS-сервера в Node.js будет примерно такой:

```
var https = require('https');
var fs = require('fs');

var httpsOptions = {
  key: fs.readFileSync('path/to/server-key.pem'),
  cert: fs.readFileSync('path/to/server-crt.pem')
};

var app = function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}

https.createServer(httpsOptions, app).listen(4433);
```

Если вы также хотите поддерживать HTTP-запросы, вам нужно сделать только эту небольшую модификацию:

```
var http = require('http');
var https = require('https');
var fs = require('fs');

var httpsOptions = {
  key: fs.readFileSync('path/to/server-key.pem'),
  cert: fs.readFileSync('path/to/server-crt.pem')
};

var app = function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}
```

```
http.createServer(app).listen(8888);  
https.createServer(httpsOptions, app).listen(4433);
```

1. перейдите в каталог, где находится ваш `server.js` :

```
cd /path/to
```

---

2. `server.js` :

```
node server.js
```

Прочитайте Начало работы с Node.js онлайн: <https://riptutorial.com/ru/node-js/topic/340/начало-работы-с-node-js>

# глава 2: async.js

## Синтаксис

- **Каждый обратный вызов должен быть написан с помощью этого синтаксиса:**
- функция `callback (err, result [, arg1 [, ...]])`
- **Таким образом, вы вынуждены сначала вернуть ошибку и не можете игнорировать их обработку позже.** `null` - это соглашение в отсутствие ошибок
- `callback (null, myResult);`
- **Ваши обратные вызовы могут содержать больше аргументов, чем `err` и `результат`, но это полезно только для определенного набора функций (водопад, `seq`, ...)**
- `callback (null, myResult, myCustomArgument);`
- **И, конечно, отправлять ошибки. Вы должны это сделать и обрабатывать ошибки (или, по крайней мере, регистрировать их).**
- обратный вызов (`ERR`);

## Examples

### Параллельно: многозадачность

`async.parallel (tasks, afterTasksCallback)` будет выполнять набор задач параллельно и **ждать окончания всех задач** (сообщенных вызовом функции **обратного вызова**).

Когда задачи завершены, *асинхронный* вызов главного обратного вызова со всеми ошибками и всеми результатами задач.

```
function shortTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfShortTime');
  }, 200);
}

function mediumTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfMediumTime');
  }, 500);
}

function longTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfLongTime');
  }, 1000);
}
```

```
    }, 1000);
  }

  async.parallel([
    shortTimeFunction,
    mediumTimeFunction,
    longTimeFunction
  ],
  function(err, results) {
    if (err) {
      return console.error(err);
    }

    console.log(results);
  });
```

**Результат:** ["resultOfShortTime", "resultOfMediumTime", "resultOfLongTime"] .

---

## Вызовите `async.parallel()` с объектом

Вы можете заменить параметр массива *задач* объектом. В этом случае результаты будут также объектом **с теми же ключами, что и задачи** .

Очень полезно вычислить некоторые задачи и легко найти каждый результат.

```
async.parallel({
  short: shortTimeFunction,
  medium: mediumTimeFunction,
  long: longTimeFunction
},
function(err, results) {
  if (err) {
    return console.error(err);
  }

  console.log(results);
});
```

**Результат:** {short: "resultOfShortTime", medium: "resultOfMediumTime", long: "resultOfLongTime"} .

---

## Разрешение нескольких значений

Каждой параллельной функции передается обратный вызов. Этот обратный вызов может либо вернуть ошибку в качестве первого аргумента, либо значения успеха после этого. Если обратный вызов передается несколько значений успеха, эти результаты возвращаются как массив.

```
async.parallel({
  short: function shortTimeFunction(callback) {
```

```

    setTimeout(function() {
      callback(null, 'resultOfShortTime1', 'resultOfShortTime2');
    }, 200);
  },
  medium: function mediumTimeFunction(callback) {
    setTimeout(function() {
      callback(null, 'resultOfMediumTime1', 'resultOfMeiumTime2');
    }, 500);
  }
},
function(err, results) {
  if (err) {
    return console.error(err);
  }

  console.log(results);
});

```

## Результат:

```

{
  short: ["resultOfShortTime1", "resultOfShortTime2"],
  medium: ["resultOfMediumTime1", "resultOfMediumTime2"]
}

```

## Серия: независимая монозагрузка

[async.series \(tasks, afterTasksCallback\)](#) выполнит набор задач. Каждая задача выполняется **за другой**. Если задача выходит из строя, **async** немедленно прекращает выполнение и переходит в основной обратный вызов.

Когда задачи завершены успешно, **асинхронный** вызов «мастер» обратного вызова со всеми ошибками и всеми результатами задач.

```

function shortTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfShortTime');
  }, 200);
}

function mediumTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfMediumTime');
  }, 500);
}

function longTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfLongTime');
  }, 1000);
}

async.series([

```

```
    mediumTimeFunction,
    shortTimeFunction,
    longTimeFunction
  ],
  function(err, results) {
    if (err) {
      return console.error(err);
    }

    console.log(results);
  });
```

**Результат:** ["resultOfMediumTime", "resultOfShortTime", "resultOfLongTime"] .

## Вызовите `async.series()` с объектом

Вы можете заменить параметр массива *задач* объектом. В этом случае результаты будут также объектом **с теми же ключами, что и задачи** .

Очень полезно вычислить некоторые задачи и легко найти каждый результат.

```
async.series({
  short: shortTimeFunction,
  medium: mediumTimeFunction,
  long: longTimeFunction
},
function(err, results) {
  if (err) {
    return console.error(err);
  }

  console.log(results);
});
```

**Результат:** {short: "resultOfShortTime", medium: "resultOfMediumTime", long: "resultOfLongTime"}

## Водопад: зависящая монозадание

[async.waterfall \(tasks, afterTasksCallback\)](#) выполнит набор задач. Каждая задача выполняется **за другой, а результат задачи передается следующей задаче** . Поскольку `async.series ()` , если задача не выполняется, `async` прекращает выполнение и немедленно вызывает основной обратный вызов.

Когда задачи завершены успешно, *асинхронный* вызов «мастер» обратного вызова со всеми ошибками и всеми результатами задач.

```
function getUserRequest(callback) {
  // We simulate the request with a timeout
  setTimeout(function() {
```

```

    var userResult = {
      name : 'Aamu'
    };

    callback(null, userResult);
  }, 500);
}

function getUserFriendsRequest(user, callback) {
  // Another request simulate with a timeout
  setTimeout(function() {
    var friendsResult = [];

    if (user.name === "Aamu"){
      friendsResult = [{
        name : 'Alice'
      }, {
        name: 'Bob'
      }];
    }

    callback(null, friendsResult);
  }, 500);
}

async.waterfall([
  getUserRequest,
  getUserFriendsRequest
],
function(err, results) {
  if (err) {
    return console.error(err);
  }

  console.log(JSON.stringify(results));
});

```

**Результат:** `results` содержат второй параметр обратного вызова последней функции водопада, который в этом случае является `friendsResult`.

## `async.times` (лучше обращаться за цикл)

Чтобы выполнить функцию в цикле в Node.js, это хорошо использовать `for` петли для коротких циклов. Но цикл длинный, использование `for` цикла увеличивает время обработки, которое может привести к зависанию процесса узла. В таких сценариях вы можете использовать: **`async.times`**

```

function recursiveAction(n, callback)
{
  //do whatever want to do repeatedly
  callback(err, result);
}

async.times(5, function(n, next) {
  recursiveAction(n, function(err, result) {
    next(err, result);
  });
}, function(err, results) {

```



```
// we should now have 5 result
});
```

Это вызывается параллельно. Когда мы хотим называть его по одному, используйте: **async.timesSeries**

## **async.each** (для эффективного управления массивом данных)

Когда мы хотим обрабатывать массив данных, лучше использовать **async.each**. Когда мы хотим что-то выполнить со всеми данными и хотим получить окончательный ответ, как только все будет сделано, этот метод будет полезен. Это обрабатывается параллельно.

```
function createUser(userName, callback)
{
  //create user in db
  callback(null)//or error based on creation
}

var arrayOfData = ['Ritu', 'Sid', 'Tom'];
async.each(arrayOfData, function(eachUserName, callback) {

  // Perform operation on each user.
  console.log('Creating user '+eachUserName);
  //Returning callback is must. Else it wont get the final callback, even if we miss to
  return one callback
  createUser(eachUserName, callback);

}, function(err) {
  //If any of the user creation failed may throw error.
  if( err ) {
    // One of the iterations produced an error.
    // All processing will now stop.
    console.log('unable to create user');
  } else {
    console.log('All user created successfully');
  }
});
```

Чтобы сделать один за раз, можно использовать **async.eachSeries**

## **async.series** (для обработки событий один за другим)

*/ В async.series все функции выполняются последовательно, и объединенные выходы каждой функции передаются в окончательный обратный вызов. например,*

```
var async = require ('async'); async.series ([function (callback) {console.log ('First Execute ..');
callback (null, 'userPersonalData');}, function (callback) {console.log ('Second Execute ..'); callback
(null, 'userDependentData');}], function (err, result) {console.log (result);});
```

//Выход:

Первое исполнение .. Второе выполнение .. ['userPersonalData', 'userDependentData'] //

результат

Прочитайте `async.js` онлайн: <https://riptutorial.com/ru/node-js/topic/3972/async-js>

---

# глава 3: CLI

## Синтаксис

- узел [параметры] [опции v8] [script.js | -e "script"] [аргументы]

## Examples

### Параметры командной строки

```
-v, --version
```

Добавлено в: v0.1.3 Версия печатного узла.

```
-h, --help
```

Добавлено в: v0.1.3 Параметры командной строки узла печати. Результат этой опции менее подробный, чем этот документ.

```
-e, --eval "script"
```

Добавлено в: v0.5.2 Оцените следующий аргумент как JavaScript. Модули, которые определены в REPL, также могут использоваться в скрипте.

```
-p, --print "script"
```

Добавлено в: v0.6.4 Идентично -e, но печатает результат.

```
-c, --check
```

Добавлено в: v5.0.0 Синтаксис проверяет скрипт без выполнения.

```
-i, --interactive
```

Добавлено в: v0.7.7 Открывает REPL, даже если stdin не является терминалом.

```
-r, --require module
```

Добавлено в: v1.6.0 Предварительно загрузите указанный модуль при запуске.

Выполняет требования к правилам разрешения модуля (). модуль может быть либо путем к файлу, либо именем узла узла.

```
--no-deprecation
```

Добавлено в: v0.8.0 Предупреждения о недопустимости молчания.

```
--trace-deprecation
```

Добавлено в: v0.8.0 Трассировка стека стека для отклонений.

```
--throw-deprecation
```

Добавлено в: v0.11.14 Ошибки броска для отклонений.

```
--no-warnings
```

Добавлено в: v6.0.0 Молчание всех предупреждений о процессах (включая устаревания).

```
--trace-warnings
```

Добавлено в: v6.0.0 Трассировка стека стека для предупреждений процесса (включая устаревания).

```
--trace-sync-io
```

Добавлено в: v2.1.0 Печатает трассировку стека всякий раз, когда синхронный ввод-вывод обнаруживается после первого поворота цикла события.

```
--zero-fill-buffers
```

Добавлено в: v6.0.0 Автоматически нуль заполняет все вновь назначенные экземпляры Buffer и SlowBuffer.

```
--preserve-symlinks
```

Добавлено в: v6.3.0 Поручает загрузчику модуля сохранять символические ссылки при разрешении и кешировании модулей.

По умолчанию, когда Node.js загружает модуль из пути, который символически связан с другим местоположением на диске, Node.js будет разыменовывать ссылку и использовать фактический «реальный путь» модуля на диске как как идентификатор и как корневой путь для поиска других модулей зависимостей. В большинстве случаев это поведение по умолчанию приемлемо. Однако при использовании символически связанных одноранговых зависимостей, как показано в приведенном ниже примере, поведение по умолчанию приводит к тому, что исключение вызывается, если модуль А пытается требовать модуля В как равноправную зависимость:

```

{appDir}
├── app
│   ├── index.js
│   └── node_modules
│       ├── moduleA -> {appDir}/moduleA
│       └── moduleB
│           ├── index.js
│           └── package.json
└── moduleA
    ├── index.js
    └── package.json

```

Флаг командной строки `-preserve-symlinks` указывает Node.js использовать путь символической ссылки для модулей в противоположность реальному пути, позволяя найти сопоставленные одноранговые зависимости.

Обратите внимание, однако, что использование `-preserve-symlinks` может иметь другие побочные эффекты. В частности, символически связанные собственные модули могут не загружаться, если они связаны из нескольких мест в дереве зависимостей (Node.js увидит их как два отдельных модуля и попытается загрузить модуль несколько раз, в результате чего будет выбрано исключение).

```
--track-heap-objects
```

Добавлено в: v2.4.0 Отслеживает распределение кучи объектов для снимков кучи.

```
--prof-process
```

Добавлено в: v6.0.0 Обработка профилировщика процесса v8, сгенерированного с использованием опции v8 `--prof`.

```
--v8-options
```

Добавлено в: v0.1.3 Параметры командной строки v8.

Примечание: опции v8 позволяют разделять слова как дефисами (-), так и подчеркиваниями (\_).

Например, `--stack-trace-limit` эквивалентно `-stack_trace_limit`.

```
--tls-cipher-list=list
```

Добавлено в: v4.0.0 Укажите альтернативный список шифрования TLS по умолчанию. (Требуется, чтобы Node.js был создан с поддержкой криптографии (по умолчанию))

```
--enable-fips
```

Добавлено в: v6.0.0 Включить криптографию, совместимую с FIPS, при запуске.

(Требуется, чтобы Node.js был создан с `./configure --openssl-fips`)

```
--force-fips
```

Добавлено в: v6.0.0 Зафиксировать криптование, совместимое с FIPS при запуске. (Невозможно отключить код сценария.) (Те же требования, что и `--enable-fips`)

```
--icu-data-dir=file
```

Добавлено в: v0.11.15 Укажите путь загрузки данных ICU. (переопределяет `NODE_ICU_DATA`)

```
Environment Variables
```

```
NODE_DEBUG=module[,...]
```

Добавлено в: v0.1.32 `'` - список основных модулей, которые должны печатать отладочную информацию.

```
NODE_PATH=path[:...]
```

Добавлено в: v0.1.32 `'` - выделенный список каталогов, предваряющих путь поиска модуля.

Примечание: в Windows это вместо этого список, разделенный `<<';>>`.

```
NODE_DISABLE_COLORS=1
```

Добавлено в: v0.3.0 Когда в REPL не будет использоваться 1 цвет, не будет использоваться.

```
NODE_ICU_DATA=file
```

Добавлено в: v0.11.15 Путь данных для данных ICU (Intl object). Расширяют связанные данные при компиляции с поддержкой `small-icu`.

```
NODE_REPL_HISTORY=file
```

Добавлено в: v5.0.0 Путь к файлу, используемому для хранения постоянной истории REPL. Путь по умолчанию - `~ / .node_repl_history`, который переопределяется этой переменной. Установка значения в пустую строку (`""` или `""`) отключает постоянную историю REPL.

Прочитайте CLI онлайн: <https://riptutorial.com/ru/node-js/topic/6013/cli>

---

# глава 4: csv парсер в узле js

## Вступление

Чтение данных из csv можно обрабатывать разными способами. Одним из решений является чтение файла `csv` в массив. Оттуда вы можете работать над массивом.

## Examples

### Использование FS для чтения в CSV

`fs` - [API файловой системы](#) в узле. Мы можем использовать метод `readFile` для нашей переменной `fs`, передать ему файл, формат и функцию `data.csv`, которая считывает и разбивает `csv` для дальнейшей обработки.

**Предполагается, что у вас есть файл с именем `data.csv` в той же папке.**

```
'use strict'

const fs = require('fs');

fs.readFile('data.csv', 'utf8', function (err, data) {
  var dataArray = data.split(/\r?\n/);
  console.log(dataArray);
});
```

Теперь вы можете использовать массив, как и любой другой, чтобы работать над ним.

Прочитайте [csv парсер в узле js онлайн](https://riptutorial.com/ru/node-js/topic/9162/csv-парсер-в-узле-js): <https://riptutorial.com/ru/node-js/topic/9162/csv-парсер-в-узле-js>

# глава 5: ECMAScript 2015 (ES6) с Node.js

## Examples

### const / let объявления

В отличие от `var`, `const` / `let` привязаны к лексической области, а не к сфере действия.

```
{
  var x = 1 // will escape the scope
  let y = 2 // bound to lexical scope
  const z = 3 // bound to lexical scope, constant
}

console.log(x) // 1
console.log(y) // ReferenceError: y is not defined
console.log(z) // ReferenceError: z is not defined
```

[Запуск в RunKit](#)

### Функции стрелок

Функции стрелки автоматически привязываются к «этой» лексической области окружающего кода.

```
performSomething(result => {
  this.someVariable = result
})
```

против

```
performSomething(function(result) {
  this.someVariable = result
}.bind(this))
```

### Пример функции стрелки

Рассмотрим этот пример, который выводит квадраты чисел 3, 5 и 7:

```
let nums = [3, 5, 7]
let squares = nums.map(function (n) {
  return n * n
})
console.log(squares)
```

[Запуск в RunKit](#)



Функция, переданная в `.map` также может быть записана как функция стрелки, удалив ключевое слово `function` и вместо этого добавив стрелку `=>` :

```
let nums = [3, 5, 7]
let squares = nums.map((n) => {
  return n * n
})
console.log(squares)
```

## Запуск в RunKit

Однако это может быть написано еще более кратким. Если тело функции состоит только из одного оператора, и этот оператор вычисляет возвращаемое значение, фигурные скобки обертывания тела функции могут быть удалены, а также ключевое слово `return` .

```
let nums = [3, 5, 7]
let squares = nums.map(n => n * n)
console.log(squares)
```

## Запуск в RunKit

### деструктурирующий

```
let [x,y, ...nums] = [0, 1, 2, 3, 4, 5, 6];
console.log(x, y, nums);

let {a, b, ...props} = {a:1, b:2, c:3, d:{e:4}}
console.log(a, b, props);

let dog = {name: 'fido', age: 3};
let {name:n, age} = dog;
console.log(n, age);
```

### течь

```
/* @flow */

function product(a: number, b: number){
  return a * b;
}

const b = 3;
let c = [1,2,3,,{}];
let d = 3;

import request from 'request';

request('http://dev.markitondemand.com/MODApis/Api/v2/Quote/json?symbol=AAPL', (err, res,
payload)=>{
  payload = JSON.parse(payload);
  let {LastPrice} = payload;
  console.log(LastPrice);
});
```

## Класс ES6

```
class Mammel {
  constructor(legs) {
    this.legs = legs;
  }
  eat() {
    console.log('eating...');
  }
  static count() {
    console.log('static count...');
  }
}

class Dog extends Mammel {
  constructor(name, legs) {
    super(legs);
    this.name = name;
  }
  sleep() {
    super.eat();
    console.log('sleeping');
  }
}

let d = new Dog('fido', 4);
d.sleep();
d.eat();
console.log('d', d);
```

Прочитайте ECMAScript 2015 (ES6) с Node.js онлайн: <https://riptutorial.com/ru/node-js/topic/6732/ecmascript-2015--es6--c-node-js>

---

## глава 6: Eventloop

### Вступление

В этой статье мы обсудим, как появилась концепция Eventloop и как ее можно использовать для высокопроизводительных серверов и приложений, управляемых событиями, таких как графические интерфейсы.

### Examples

Как эволюционировала концепция цикла событий.

---

## Eventloop в псевдокоде

Цикл событий представляет собой цикл, ожидающий событий, а затем реагирует на эти события

```
while true:
    wait for something to happen
    react to whatever happened
```

---

## Пример однопоточного HTTP-сервера без цикла события

```
while true:
    socket = wait for the next TCP connection
    read the HTTP request headers from (socket)
    file_contents = fetch the requested file from disk
    write the HTTP response headers to (socket)
    write the (file_contents) to (socket)
    close(socket)
```

Вот простая форма HTTP-сервера, которая представляет собой однопоточный, но не цикл событий. Проблема здесь в том, что он ждет, пока каждый запрос не будет закончен, прежде чем начинать обработку следующего. Если для чтения заголовков HTTP-запросов или для извлечения файла с диска требуется некоторое время, мы сможем начать обработку следующего запроса, пока мы ждем его завершения.

Наиболее распространенное решение - сделать программу многопоточной.

# Пример многопоточного HTTP-сервера без цикла события

```
function handle_connection(socket):
    read the HTTP request headers from (socket)
    file_contents = fetch the requested file from disk
    write the HTTP response headers to (socket)
    write the (file_contents) to (socket)
    close(socket)
while true:
    socket = wait for the next TCP connection
    spawn a new thread doing handle_connection(socket)
```

Теперь мы сделали наш небольшой HTTP-сервер многопоточным. Таким образом, мы можем сразу перейти к следующему запросу, поскольку текущий запрос выполняется в фоновом потоке. Многие серверы, включая Apache, используют этот подход.

Но это не идеально. Одно ограничение состоит в том, что вы можете создавать только столько потоков. Для рабочих нагрузок, где у вас огромное количество подключений, но каждое подключение требует только внимания каждый раз, многопоточная модель не будет работать очень хорошо. Решением для этих случаев является использование цикла событий:

---

## Пример HTTP-сервера с циклом события

```
while true:
    event = wait for the next event to happen
    if (event.type == NEW_TCP_CONNECTION):
        conn = new Connection
        conn.socket = event.socket
        start reading HTTP request headers from (conn.socket) with userdata = (conn)
    else if (event.type == FINISHED_READING_FROM_SOCKET):
        conn = event.userdata
        start fetching the requested file from disk with userdata = (conn)
    else if (event.type == FINISHED_READING_FROM_DISK):
        conn = event.userdata
        conn.file_contents = the data we fetched from disk
        conn.current_state = "writing headers"
        start writing the HTTP response headers to (conn.socket) with userdata = (conn)
    else if (event.type == FINISHED_WRITING_TO_SOCKET):
        conn = event.userdata
        if (conn.current_state == "writing headers"):
            conn.current_state = "writing file contents"
            start writing (conn.file_contents) to (conn.socket) with userdata = (conn)
        else if (conn.current_state == "writing file contents"):
            close(conn.socket)
```

Надеюсь, этот псевдокод понятен. Вот что происходит: мы ждем, когда что-то случится.

Всякий раз, когда создается новое соединение или существующее соединение требует нашего внимания, мы решаем его, а затем возвращаемся к ожиданию. Таким образом, мы хорошо себя чувствуем, когда есть много соединений, и каждый из них редко требует внимания.

В реальном приложении (не псевдокоде), запущенном в Linux, часть «ожидание следующего события» будет реализована путем вызова системного вызова `poll ()` или `epoll ()`. «Начало чтения / записи чего-то в сокет» будет реализовано путем вызова системных вызовов `recv ()` или `send ()` в неблокирующем режиме.

*Ссылка:*

[1]. «Как работает цикл событий?» [Онлайн]. Доступно: <https://www.quora.com/How-does-an-event-loop-work>

Прочитайте Eventloop онлайн: <https://riptutorial.com/ru/node-js/topic/8652/eventloop>

# глава 7: HTTP

## Examples

### http server

Основной пример HTTP-сервера.

напишите следующий код в файле `http_server.js`:

```
var http = require('http');

var httpPort = 80;

http.createServer(handler).listen(httpPort, start_callback);

function handler(req, res) {

    var clientIP = req.connection.remoteAddress;
    var connectUsing = req.connection.encrypted ? 'SSL' : 'HTTP';
    console.log('Request received: ' + connectUsing + ' ' + req.method + ' ' + req.url);
    console.log('Client IP: ' + clientIP);

    res.writeHead(200, "OK", {'Content-Type': 'text/plain'});
    res.write("OK");
    res.end();
    return;
}

function start_callback(){
    console.log('Start HTTP on port ' + httpPort)
}
```

то из вашего местоположения `http_server.js` выполните следующую команду:

```
node http_server.js
```

вы должны увидеть этот результат:

```
> Start HTTP on port 80
```

теперь вам нужно протестировать свой сервер, вам нужно открыть свой интернет-браузер и перейти к этому URL-адресу:

```
http://127.0.0.1:80
```

если ваша машина под управлением Linux-сервера вы можете протестировать ее следующим образом:

```
curl 127.0.0.1:80
```

вы должны увидеть следующий результат:

```
ok
```

в вашей консоли, которая запускает приложение, вы увидите следующие результаты:

```
> Request received: HTTP GET /  
> Client IP: ::ffff:127.0.0.1
```

## http client

базовый пример для http-клиента:

напишите следующий код в файле `http_client.js`:

```
var http = require('http');  
  
var options = {  
  hostname: '127.0.0.1',  
  port: 80,  
  path: '/',  
  method: 'GET'  
};  
  
var req = http.request(options, function(res) {  
  console.log('STATUS: ' + res.statusCode);  
  console.log('HEADERS: ' + JSON.stringify(res.headers));  
  res.setEncoding('utf8');  
  res.on('data', function (chunk) {  
    console.log('Response: ' + chunk);  
  });  
  res.on('end', function (chunk) {  
    console.log('Response ENDED');  
  });  
});  
  
req.on('error', function(e) {  
  console.log('problem with request: ' + e.message);  
});  
  
req.end();
```

то из вашего местоположения `http_client.js` выполните следующую команду:

```
node http_client.js
```

вы должны увидеть этот результат:

```
> STATUS: 200  
> HEADERS: {"content-type":"text/plain","date":"Thu, 21 Jul 2016 11:27:17
```

```
GMT", "connection": "close", "transfer-encoding": "chunked"}  
> Response: OK  
> Response ENDED
```

note: этот пример зависит от примера HTTP-сервера.

Прочитайте HTTP онлайн: <https://riptutorial.com/ru/node-js/topic/2973/http>



---

# глава 8: Кoa Framework v2

## Examples

### Пример Hello World

```
const Koa = require('koa')

const app = new Koa()

app.use(async ctx => {
  ctx.body = 'Hello World'
})

app.listen(8080)
```

### Обработка ошибок с использованием промежуточного программного обеспечения

```
app.use(async (ctx, next) => {
  try {
    await next() // attempt to invoke the next middleware downstream
  } catch (err) {
    handleError(err, ctx) // define your own error handling function
  }
})
```

Прочитайте Кoa Framework v2 онлайн: <https://riptutorial.com/ru/node-js/topic/6730/koa-framework-v2>

---

# глава 9: Lodash

## Вступление

Lodash - удобная утилита для JavaScript.

## Examples

### Фильтрация коллекции

В приведенном ниже фрагменте кода показаны различные способы фильтрации по массиву объектов с помощью lodash.

```
let lodash = require('lodash');

var countries = [
  {"key": "DE", "name": "Deutschland", "active": false},
  {"key": "ZA", "name": "South Africa", "active": true}
];

var filteredByFunction = lodash.filter(countries, function (country) {
  return country.key === "DE";
});
// => [{"key": "DE", "name": "Deutschland"}];

var filteredByObjectProperties = lodash.filter(countries, { "key": "DE" });
// => [{"key": "DE", "name": "Deutschland"}];

var filteredByProperties = lodash.filter(countries, ["key", "ZA"]);
// => [{"key": "ZA", "name": "South Africa"}];

var filteredByProperty = lodash.filter(countries, "active");
// => [{"key": "ZA", "name": "South Africa"}];
```

Прочитайте Lodash онлайн: <https://riptutorial.com/ru/node-js/topic/9161/lodash>

# глава 10: Loorback - разъем на основе REST

## Вступление

Разъемы на основе отдыха и способы их устранения. Мы все знаем, что Loorback не обеспечивает элегантность подключений на основе REST

## Examples

### Добавление сетевого соединителя

```
// В этом примере получен ответ от iTunes
{
  "остальное": {
    «name»: «rest»,
    «коннектор»: «отдых»,
    «debug»: true,
    "опции": {
      «useQueryString»: true,
      «таймаут»: 10000,
      «заголовки»: {
        «принимает»: «application / json»,
        "content-type": "application / json"
      }
    },
    «операции»: [
      {
        «шаблон»: {
          «метод»: «GET»,
          «url»: «https://itunes.apple.com/search»,
          "query": {
            "term": "{keyword}",
            «страна»: «{страна = IN}»,
            «media»: «{itemType = music}»,
            «limit»: «{limit = 10}»,
            "Явный": "false"
          }
        },
        «функции»: {
          «поиск»: [
            "ключевое слово",
            "страна",
            "тип элемента",
            «Предел»
          ]
        }
      },
      {
        «шаблон»: {
          «метод»: «GET»,
          «url»: «https://itunes.apple.com/lookup»,
```

```
    "query": {
      "я сделал}"
    }
  },
  «функции»: {
    "findById": [
      "я бы"
    ]
  }
}
]
```

Прочитайте **Loorback - разъем на основе REST** онлайн: <https://riptutorial.com/ru/node-js/topic/9234/loopback---разъем-на-основе-rest>

# глава 11: N-API,

## Вступление

N-API - это новый и лучший способ создания собственного модуля для NodeJS. N-API находится на ранней стадии, поэтому у него может быть несогласованная документация.

## Examples

### Привет, N-API

Этот модуль регистрирует hello-функцию на модуле hello. hello function prints Hello world на консоли с `printf` и возвращает 1373 из встроенной функции в javascript caller.

```
#include <node_api.h>
#include <stdio.h>

napi_value say_hello(napi_env env, napi_callback_info info)
{
    napi_value retval;

    printf("Hello world\n");

    napi_create_number(env, 1373, &retval);

    return retval;
}

void init(napi_env env, napi_value exports, napi_value module, void* priv)
{
    napi_status status;
    napi_property_descriptor desc = {
        /*
         * String describing the key for the property, encoded as UTF8.
         */
        .utf8name = "hello",
        /*
         * Set this to make the property descriptor object's value property
         * to be a JavaScript function represented by method.
         * If this is passed in, set value, getter and setter to NULL (since these members
         won't be used).
         */
        .method = say_hello,
        /*
         * A function to call when a get access of the property is performed.
         * If this is passed in, set value and method to NULL (since these members won't be
         used).
         * The given function is called implicitly by the runtime when the property is
         accessed
         * from JavaScript code (or if a get on the property is performed using a N-API call).
         */
        .getter = NULL,
```

```

    /*
    * A function to call when a set access of the property is performed.
    * If this is passed in, set value and method to NULL (since these members won't be
used).
    * The given function is called implicitly by the runtime when the property is set
    * from JavaScript code (or if a set on the property is performed using a N-API call).
    */
    .setter = NULL,
    /*
    * The value that's retrieved by a get access of the property if the property is a
data property.
    * If this is passed in, set getter, setter, method and data to NULL (since these
members won't be used).
    */
    .value = NULL,
    /*
    * The attributes associated with the particular property. See
napi_property_attributes.
    */
    .attributes = napi_default,
    /*
    * The callback data passed into method, getter and setter if this function is
invoked.
    */
    .data = NULL
};
/*
* This method allows the efficient definition of multiple properties on a given object.
*/
status = napi_define_properties(env, exports, 1, &desc);

if (status != napi_ok)
    return;
}

NAPI_MODULE(hello, init)

```

Прочитайте N-API, онлайн: <https://riptutorial.com/ru/node-js/topic/10539/n-api->

---

# глава 12: Node.js (express.js) с угловым.js

## Пример кода

### Вступление

В этом примере показано, как создать базовое экспресс-приложение, а затем выполнить функцию AngularJS.

### Examples

#### Создание нашего проекта.

Мы готовы пойти так, мы бежим, снова с консоли:

```
mkdir our_project
cd our_project
```

Теперь мы находимся там, где будет жить наш код. Чтобы создать главный архив нашего проекта, вы можете запустить

#### Хорошо, но как мы создаем проект экспресс-скелета?

Это просто:

```
npm install -g express express-generator
```

Linux-дистрибутивы и Mac должны использовать **sudo** для установки, потому что они установлены в каталоге nodejs, доступ к которому доступен только пользователю **root** . Если все будет хорошо, мы сможем, наконец, создать скелет экспресс-приложения, просто запустите

```
express
```

Эта команда создаст внутри нашей папки приложение для экспресс-примера. Структура следующая:

```
bin/
public/
routes/
views/
app.js
package.json
```

Теперь, если мы запустим **npm**, **запустите** go to <http://localhost:3000>, мы увидим, что экспресс-приложение работает и работает, справедливо, что мы создали экспресс-приложение без особых проблем, но как мы можем смешать это с AngularJS? ,

## Как работает экспресс, ненадолго?

**Express** - это структура, построенная поверх **Nodejs** , вы можете увидеть официальную документацию на [Express Site](#) . Но для нашей цели нам нужно знать, что **Express** является ответственным, когда мы печатаем, например, <http://localhost:3000/home> для рендеринга домашней страницы нашего приложения. Из созданного недавно созданного приложения мы можем проверить:

```
FILE: routes/index.js
var express = require('express');
var router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

module.exports = router;
```

Этот код говорит нам, что когда пользователь переходит на <http://localhost:3000>, он должен отображать представление **индекса** и передавать **JSON** с свойством **title** и значением **Express**. Но когда мы проверяем каталог **views** и открываем **index.jade**, мы можем видеть это:

```
extends layout
block content
  h1= title
  p Welcome to #{title}
```

Это еще одна мощная функция Express, **механизмы шаблонов** , они позволяют отображать контент на странице, передавая ему переменные или наследуя другой шаблон, чтобы ваши страницы были более компактными и более понятными для других. Расширение файла **.jade**, насколько я знаю, **Джейд** изменил имя для **Pug** , в основном это тот же движок шаблонов, но с некоторыми обновлениями и модификациями ядра.

## Установка мошенника и обновление шаблона Express.

Хорошо, чтобы начать использовать **Pug** в качестве механизма шаблонов нашего проекта, нам нужно запустить:

```
npm install --save pug
```

Это установит **Pug** как зависимость от нашего проекта и сохранит его в **package.json** .



Чтобы использовать его, нам нужно изменить файл **app.js** :

```
var app = express();
// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'pug');
```

И замените линию движка с помощью мопса, и все. Мы снова можем запустить наш проект с **запуском npm**, и мы увидим, что все работает нормально.

## Как AngularJS вписывается во все это?

AngularJS - это Javascript **MVW** (Model-View-Whatever) Framework, который в основном используется для создания приложения **SPA** (Simple Page Application), довольно прост, вы можете перейти на [сайт AngularJS](#) и загрузить последнюю версию, которая является **v1.6.4**.

После того, как мы загрузили AngularJS, когда нужно скопировать файл в нашу **общедоступную / javascripts** папку внутри нашего проекта, небольшое объяснение, это папка, которая служит для статических ресурсов нашего сайта, изображений, css, javascript файлов и так далее. Конечно, это можно настроить через файл **app.js**, но мы будем держать его простым. Теперь мы создаем файл с именем **ng-app.js**, файл, в котором наше приложение будет жить, внутри нашей общей папки javascripts, где находится AngularJS. Чтобы привести AngularJS вверх, нам нужно изменить содержимое **views / layout.pug** следующим образом:

```
doctype html
html (ng-app='first-app')
  head
    title= title
    link (rel='stylesheet', href='/stylesheets/style.css')
  body (ng-controller='indexController')
    block content

    script (type='text-javascript', src='javascripts/angular.min.js')
    script (type='text-javascript', src='javascripts/ng-app.js')
```

Что мы здесь делаем ?, мы включаем в себя ядро AngularJS и наш недавно созданный файл **ng-app.js**, поэтому, когда шаблон визуализируется, он будет вызывать AngularJS, обратит внимание на использование директивы **ng-app**, это говорит AngularJS, что это наше имя приложения, и он должен придерживаться этого.

Итак, содержание нашего **ng-app.js** будет:

```
angular.module('first-app', [])
  .controller('indexController', ['$scope', indexController]);

function indexController($scope) {
  $scope.name = 'sigfried';
}
```

Мы используем самую основную функцию AngularJS здесь, **двустороннюю привязку данных**, это позволяет нам мгновенно обновлять содержимое нашего представления и контроллера, это очень простое объяснение, но вы можете провести исследование в Google или StackOverflow, чтобы увидеть как это работает.

Итак, у нас есть основные блоки нашего приложения AngularJS, но есть что-то, что нам нужно сделать, нам нужно обновить нашу страницу index.pug, чтобы увидеть изменения нашего углового приложения, давайте сделаем это:

```
extends layout
block content
  div (ng-controller='indexController')
    h1= title
    p Welcome {{name}}
    input (type='text' ng-model='name')
```

Здесь мы просто привязываем вход к нашему определенному имени свойства в области AngularJS внутри нашего контроллера:

```
$scope.name = 'sigfried';
```

Цель этого заключается в том, что всякий раз, когда мы меняем текст во вводе, вышеприведенный абзац обновляет его содержимое внутри {{name}}, это называется **интерполяцией**, снова другая функция AngularJS для отображения нашего контента в шаблоне.

Итак, все настроено, теперь мы можем запустить **npm**, перейдем к <http://localhost:3000> и посмотрим наше экспресс-приложение, обслуживающее страницу, и AngularJS, управляющее интерфейсом приложения.

Прочитайте [Node.js \(express.js\) с угловым.js](#) Пример кода онлайн:

<https://riptutorial.com/ru/node-js/topic/9757/node-js--express-js--с-угловым-js-пример-кода>

---

# глава 13: Node.js v6 Новые возможности и улучшения

## Вступление

С узлом 6 становится новой версией узла LTS. Мы можем увидеть ряд улучшений в языке через новые стандарты ES6. Мы рассмотрим некоторые новые функции и примеры того, как их реализовать.

## Examples

### Параметры функции по умолчанию

```
function addTwo(a, b = 2) {  
    return a + b;  
}  
  
addTwo(3) // Returns the result 5
```

С добавлением параметров функции по умолчанию вы можете сделать аргументы опциональными и по умолчанию использовать их по вашему выбору.

### Параметры останова

```
function argumentLength(...args) {  
    return args.length;  
}  
  
argumentLength(5) // returns 1  
argumentLength(5, 3) //returns 2  
argumentLength(5, 3, 6) //returns 3
```

Предваряя последний аргумент вашей функции с помощью `...` все аргументы, переданные функции, считаются как массив. В этом примере мы получаем проход в нескольких аргументах и получаем длину массива, созданного из этих аргументов.

### Оператор распространения

```
function myFunction(x, y, z) { }  
var args = [0, 1, 2];  
myFunction(...args);
```

Синтаксис распространения позволяет выражать выражение в местах, где ожидаются несколько аргументов (для вызовов функций) или нескольких элементов (для литералов

массива) или нескольких переменных. Как и остальные параметры, просто представьте свой массив с помощью ...

## Функции стрелки

Функция Arrow - это новый способ определения функции в ECMAScript 6.

```
// traditional way of declaring and defining function
var sum = function(a,b)
{
    return a+b;
}

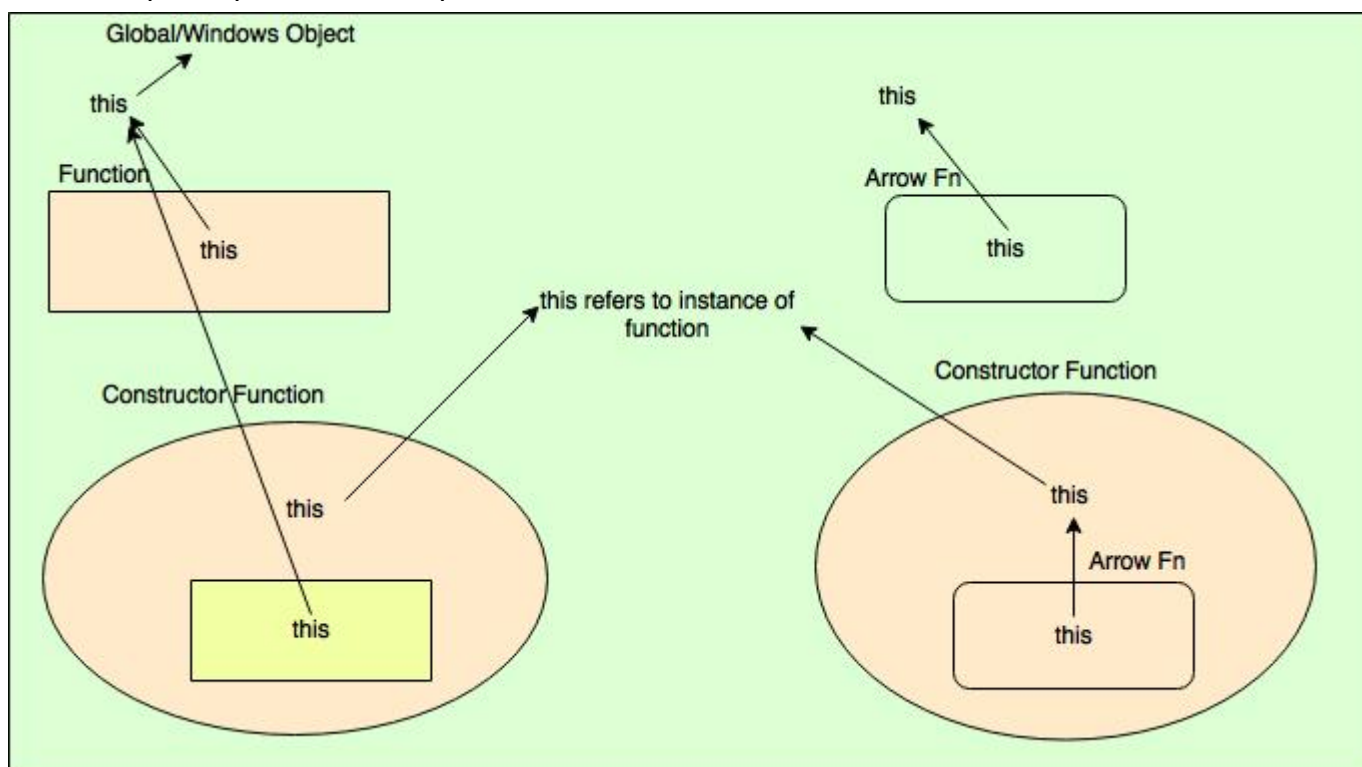
// Arrow Function
let sum = (a, b)=> a+b;

//Function definition using multiple lines
let checkIfEven = (a) => {
    if( a % 2 == 0 )
        return true;
    else
        return false;
}
```

### «это» в функции стрелки

**это** в функции относится к экземпляру объекта, используемому для вызова этой функции, но **это** в функции стрелок равно *этой* функции, в которой определена функция стрелки.

Давайте разберемся с диаграммой



Понимание использования примеров.

```

var normalFn = function(){
  console.log(this) // refers to global/window object.
}

var arrowFn = () => console.log(this); // refers to window or global object as function is
defined in scope of global/window object

var service = {

  constructorFn : function(){

    console.log(this); // refers to service as service object used to call method.

    var nestedFn = function(){
      console.log(this); // refers window or global object because no instance object
was used to call this method.
    }
    nestedFn();
  },

  arrowFn : function(){
    console.log(this); // refers to service as service object was used to call method.
    let fn = () => console.log(this); // refers to service object as arrow function
defined in function which is called using instance object.
    fn();
  }
}

// calling defined functions
constructorFn();
arrowFn();
service.constructorFn();
service.arrowFn();

```

В функции стрелок *это* лексическая область действия, которая является областью функции, в которой определена функция стрелки.

Первый пример - традиционный способ определения функций, и, следовательно, *это* относится к *глобальному / оконному* объекту.

Во втором примере *это* используется внутри функции стрелки, следовательно, *это* относится к области, где она определена (которая является окном или глобальным объектом). В третьем примере *это* служебный объект, так как объект службы используется для вызова функции.

В четвертом примере функция стрелки определена и вызывается из функции, область действия которой является *сервисом*, поэтому она печатает *служебный* объект.

*Примечание: глобальный объект печатается в объекте Node.js и Windows в браузере.*

Прочитайте [Node.js v6 Новые возможности и улучшения онлайн](https://riptutorial.com/ru/node-js/topic/8593/node-js-v6-новые-возможности-и-улучшения):

<https://riptutorial.com/ru/node-js/topic/8593/node-js-v6-новые-возможности-и-улучшения>

---

# глава 14: Node.JS и MongoDB.

## замечания

Это основные операции CRUD для использования mongo db с nodejs.

Вопрос: Существуют ли другие способы сделать то, что делается здесь?

Ответ: Да, есть много способов сделать это.

Вопрос: Является ли использование мангуста необходимым?

Ответ: Нет. Есть другие доступные пакеты, которые могут вам помочь.

Вопрос: Где я могу получить полную документацию о мангусте?

Ответ: [Нажмите здесь](#)

## Examples

### Подключение к базе данных

Для подключения к базе данных mongo из приложения-узла требуется mongoose.

Установка Mongoose. Перейдите к гуцу вашего приложения и установите мангуст на

```
npm install mongoose
```

Затем мы подключаемся к базе данных.

```
var mongoose = require('mongoose');

//connect to the test database running on default mongod port of localhost
mongoose.connect('mongodb://localhost/test');

//Connecting with custom credentials
mongoose.connect('mongodb://USER:PASSWORD@HOST:PORT/DATABASE');

//Using Pool Size to define the number of connections opening
//Also you can use a call back function for error handling
mongoose.connect('mongodb://localhost:27017/consumers',
  {server: { poolSize: 50 }},
  function(err) {
    if(err) {
      console.log('error in this')
      console.log(err);
    }
  }
);
```

```
        // Do whatever to handle the error
    } else {
        console.log('Connected to the database');
    }
});
```

## Создание новой коллекции

С Mongoose все происходит от схемы. Позволяет создать схему.

```
var mongoose = require('mongoose');

var Schema = mongoose.Schema;

var AutoSchema = new Schema({
  name : String,
  countOf: Number,
});
// defining the document structure

// by default the collection created in the db would be the first parameter we use (or the
plural of it)
module.exports = mongoose.model('Auto', AutoSchema);

// we can over write it and define the collection name by specifying that in the third
parameters.
module.exports = mongoose.model('Auto', AutoSchema, 'collectionName');

// We can also define methods in the models.
AutoSchema.methods.speak = function () {
  var greeting = this.name
    ? "Hello this is " + this.name+ " and I have counts of "+ this.countOf
    : "I don't have a name";
  console.log(greeting);
}
mongoose.model('Auto', AutoSchema, 'collectionName');
```

Помните, что методы должны быть добавлены в схему перед компиляцией с помощью `mongoose.model()`, как показано выше.

## Вставка документов

Для вставки нового документа в коллекцию мы создаем объект схемы.

```
var Auto = require('models/auto')
var autoObj = new Auto({
  name: "NewName",
  countOf: 10
});
```

Мы сохраняем его следующим образом

```
autoObj.save(function(err, insertedAuto) {
```

```
if (err) return console.error(err);
insertedAuto.speak();
// output: Hello this is NewName and I have counts of 10
});
```

Это добавит новый документ в сборник

## Чтение

Чтение данных из коллекции очень просто. Получение всех данных коллекции.

```
var Auto = require('models/auto')
Auto.find({}, function (err, autos) {
  if (err) return console.error(err);
  // will return a json array of all the documents in the collection
  console.log(autos);
})
```

Чтение данных с условием

```
Auto.find({countOf: {$gte: 5}}, function (err, autos) {
  if (err) return console.error(err);
  // will return a json array of all the documents in the collection whose count is
  greater than 5
  console.log(autos);
})
```

Вы также можете указать второй параметр как объект того, что нужно всем полям

```
Auto.find({}, {name:1}, function (err, autos) {
  if (err) return console.error(err);
  // will return a json array of name field of all the documents in the collection
  console.log(autos);
})
```

Поиск одного документа в коллекции.

```
Auto.findOne({name:"newName"}, function (err, auto) {
  if (err) return console.error(err);
  //will return the first object of the document whose name is "newName"
  console.log(auto);
})
```

Поиск одного документа в коллекции по id.

```
Auto.findById(123, function (err, auto) {
  if (err) return console.error(err);
  //will return the first json object of the document whose id is 123
  console.log(auto);
})
```



## обновление

Для обновления коллекций и документов мы можем использовать любой из этих методов:

### МЕТОДЫ

- Обновить()
- updateOne ()
- updateMany ()
- replaceOne ()

---

## Обновить()

Метод `update ()` изменяет один или несколько документов (параметры обновления)

```
db.lights.update(  
  { room: "Bedroom" },  
  { status: "On" }  
)
```

Эта операция ищет коллекцию «огни» для документа , где `room` является **спальней** (первый параметр). Затем он обновляет согласующие документы о `status` собственности **On** (второй параметр) и возвращает объект `WriteResult` который выглядит следующим образом :

```
{ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 }
```

---

## UpdateOne

Метод `UpdateOne ()` изменяет один документ (параметры обновления)

```
db.countries.update(  
  { country: "Sweden" },  
  { capital: "Stockholm" }  
)
```

Эта операция производит поиск коллекции "стран для документа , где `country` является **Швеция** (первый параметр). Затем он обновляет согласующие документы собственности `capital` в **Стокгольм** (второй параметр) и возвращает объект `WriteResult` который выглядит следующим образом :

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

# UpdateMany

Метод `UpdateMany()` изменяет многоуровневые документы (параметры обновления)

```
db.food.updateMany(  
  { sold: { $lt: 10 } },  
  { $set: { sold: 55 } }  
)
```

Эта операция обновляет все документы (в коллекции «еда»), где `sold` **менее 10** \* (1-й параметр), установив `sold` на **55**. Затем он возвращает объект `WriteResult`, который выглядит так:

```
{ "acknowledged" : true, "matchedCount" : a, "modifiedCount" : b }
```

`a` = Количество согласованных документов

`b` = Количество измененных документов

---

# ReplaceOne

Заменяет первый соответствующий документ (заменяющий документ)

В этом примере коллекция, называемая **странами**, содержит 3 документа:

```
{ "_id" : 1, "country" : "Sweden" }  
{ "_id" : 2, "country" : "Norway" }  
{ "_id" : 3, "country" : "Spain" }
```

Следующая операция заменяет документ `{ country: "Spain" }` документом `{ country: "Finland" }`

```
db.countries.replaceOne(  
  { country: "Spain" },  
  { country: "Finland" }  
)
```

И возвращает:

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

В настоящее время **страны, в которых** собраны примеры, содержат:

```
{ "_id" : 1, "country" : "Sweden" }  
{ "_id" : 2, "country" : "Norway" }  
{ "_id" : 3, "country" : "Finland" }
```

## Удаление

Удаление документов из коллекции в мангусте выполняется следующим образом.

```
Auto.remove({_id:123}, function(err, result){
  if (err) return console.error(err);
  console.log(result); // this will specify the mongo default delete result.
});
```

Прочитайте Node.JS и MongoDB. онлайн: <https://riptutorial.com/ru/node-js/topic/7505/node-js-и-mongodb->

# глава 15: Node.js с CORS

## Examples

### Включить CORS в express.js

Поскольку node.js часто используется для создания API, правильная настройка CORS может быть спасателем жизни, если вы хотите иметь возможность запрашивать API из разных доменов.

В качестве примера мы настроим его для более широкой конфигурации (авторизуем все типы запросов из любого домена).

На сервере server.js после инициализации express:

```
// Create express server
const app = express();

app.use((req, res, next) => {
  res.header('Access-Control-Allow-Origin', '*');

  // authorized headers for preflight requests
  // https://developer.mozilla.org/en-US/docs/Glossary/preflight_request
  res.header('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content-Type,
  Accept');
  next();

  app.options('*', (req, res) => {
    // allowed XHR methods
    res.header('Access-Control-Allow-Methods', 'GET, PATCH, PUT, POST, DELETE, OPTIONS');
    res.send();
  });
});
```

Обычно узел запускается за прокси на рабочих серверах. Поэтому обратный прокси-сервер (такой как Apache или Nginx) будет отвечать за конфигурацию CORS.

Чтобы удобно адаптировать этот сценарий, возможно только включить node.js CORS, когда он находится в разработке.

Это легко сделать, проверив `NODE_ENV` :

```
const app = express();

if (process.env.NODE_ENV === 'development') {
  // CORS settings
}
```

Прочитайте Node.js с CORS онлайн: <https://riptutorial.com/ru/node-js/topic/9272/node-js-c-cors>

---

# глава 16: Node.JS с ES6

## Вступление

ES6, ECMAScript 6 или ES2015 - это новейшая [спецификация](#) JavaScript, которая вводит некоторый синтаксический сахар в язык. Это большое обновление для языка и множество новых [функций](#)

Более подробную информацию о узлах и ES6 можно найти на их сайте <https://nodejs.org/en/docs/es6/>

## Examples

### Узел ES6 Поддержка и создание проекта с Babel

Вся спецификация ES6 еще не реализована полностью, поэтому вы сможете использовать только некоторые из новых функций. Вы можете увидеть список текущих поддерживаемых функций ES6 на странице <http://node.green/>

С NodeJS v6 была довольно хорошая поддержка. Поэтому, если вы используете NodeJS v6 или выше, вы можете пользоваться ES6. Тем не менее, вы также можете использовать некоторые из неизданных функций, а некоторые из-за пределов. Для этого вам нужно будет использовать транспилер

Можно запустить транспилятор во время выполнения и сборки, чтобы использовать все функции ES6 и многое другое. Самый популярный транспилятор для JavaScript называется [Babel](#)

Babel позволяет использовать все функции из спецификации ES6 и некоторые дополнительные функции не в спецификации с «stage-0», например, `import thing from 'thing'` **ВМЕСТО** `var thing = require('thing')`

Если бы мы хотели создать проект, в котором мы использовали бы функции stage-0, такие как импорт, нам нужно было бы добавить Babel в качестве транспилятора. Вы увидите проекты с использованием реакции, а Vue и другие шаблоны на основе commonJS реализуют этап-0 довольно часто.

создать новый проект узла

```
mkdir my-es6-app
cd my-es6-app
npm init
```

Установите Babel на предустановку ES6 и этап-0

```
npm install --save-dev babel-preset-es2015 babel-preset-stage-2 babel-cli babel-register
```

Создайте новый файл с именем `server.js` и добавьте базовый HTTP-сервер.

```
import http from 'http'

http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'})
  res.end('Hello World\n')
}).listen(3000, '127.0.0.1')

console.log('Server running at http://127.0.0.1:3000/')
```

Обратите внимание, что мы используем `import http from 'http'` -порт для `import http from 'http'` это функция stage-0, и если она работает, это означает, что мы правильно работаем с транспилером.

Если вы запустите `node server.js` он не сможет понять, как обрабатывать импорт.

Создание файла `.babelrc` в корневой директории вашего каталога и добавление следующих параметров

```
{
  "presets": ["es2015", "stage-2"],
  "plugins": []
}
```

теперь вы можете запустить сервер с `node src/index.js --exec babel-node`

Завершение работы - неплохо запустить транспилатор во время выполнения в производственном приложении. Однако мы можем внедрить некоторые скрипты в нашем пакете `json`, чтобы упростить работу.

```
"scripts": {
  "start": "node dist/index.js",
  "dev": "babel-node src/index.js",
  "build": "babel src -d dist",
  "postinstall": "npm run build"
},
```

Вышеупомянутое будет на `npm install` строить переданный код в каталог `dist`, чтобы `npm start` использовать перекодированный код для нашего производственного приложения.

`npm run dev` будет загружать сервер и время загрузки буфера, что отлично и предпочтительно при работе над проектом локально.

Идя дальше, вы можете установить `nodemon` `npm install nodemon --save-dev` чтобы следить за изменениями и перезагружать приложение узла.

Это действительно ускоряет работу с `babel` и `NodeJS`. В вас `package.json` просто обновите

скрипт «dev», чтобы использовать nodemon

```
"dev": "nodemon src/index.js --exec babel-node",
```

## Использовать JS es6 в приложении NodeJS

JS es6 (также известный как es2015) представляет собой набор новых функций для языка JS, цель которого заключается в том, чтобы сделать его более интуитивным при использовании ООП или при решении современных задач разработки.

### Предпосылки:

1. Ознакомьтесь с новыми функциями es6 по адресу <http://es6-features.org> - он может вас разъяснить, если вы действительно собираетесь использовать его в своем следующем приложении NodeJS
2. Проверьте уровень совместимости версии вашего узла на [странице http://node.green](http://node.green)
3. Если все в порядке - давайте код!

Вот очень короткая выборка простого приложения `hello world` с JS es6

```
'use strict'  
  
class Program  
{  
  constructor()  
  {  
    this.message = 'hello es6 :)';  
  }  
  
  print()  
  {  
    setTimeout(() =>  
    {  
      console.log(this.message);  
  
      this.print();  
  
    }, Math.random() * 1000);  
  }  
}  
  
new Program().print();
```

Вы можете запустить эту программу и наблюдать, как она печатает одно и то же сообщение снова и снова.

Теперь .. пусть ломает его по строкам:

```
'use strict'
```

Эта строка действительно требуется, если вы собираетесь использовать js es6. `strict` режим, умышленно, имеет разную семантику из обычного кода (пожалуйста, прочитайте больше об этом на MDN - [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict\\_mode](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode))

```
class Program
```

Невероятно - ключевое слово `class` ! Просто для быстрой справки - до es6 единственный способ определить класс в js был с ключевым словом ... `function` !

```
function MyClass() // class definition
{
}

var myClassObject = new MyClass(); // generating a new object with a type of MyClass
```

При использовании ООП класс является очень фундаментальной способностью, которая помогает разработчику представлять определенную часть системы (разрыв кода имеет решающее значение, когда код становится больше .. например: при написании кода на стороне сервера)

```
constructor()
{
  this.message = 'hello es6 :)';
}
```

Вы должны признать - это довольно интуитивно! Это `ctor` моего класса - эта уникальная «функция» будет возникать каждый раз, когда объект создается из этого конкретного класса (в нашей программе - только один раз)

```
print()
{
  setTimeout(() => // this is an 'arrow' function
  {
    console.log(this.message);

    this.print(); // here we call the 'print' method from the class template itself (a recursion in this particular case)

  }, Math.random() * 1000);
}
```

Поскольку `print` определен в области класса - это фактически метод, который может быть вызван либо из объекта класса, либо из самого класса!

Итак .. до сих пор мы определили наш класс .. время его использовать:

```
new Program().print();
```



Что действительно равно:

```
var prog = new Program(); // define a new object of type 'Program'  
  
prog.print(); // use the program to print itself
```

**В заключение:** JS es6 может упростить ваш код - сделать его более понятным и понятным (по сравнению с предыдущей версией JS) .. вы можете попытаться переписать существующий код и увидеть разницу для себя

НАСЛАЖДАТЬСЯ :)

Прочитайте Node.JS с ES6 онлайн: <https://riptutorial.com/ru/node-js/topic/5934/node-js-c-es6>

# глава 17: Node.js с Oracle

## Examples

### Подключение к базе данных Oracle

Очень простой способ подключения к базе данных ORACLE - использовать модуль [oracledb](#). Этот модуль обрабатывает соединение между вашим приложением Node.js и сервером Oracle. Вы можете установить его, как и любой другой модуль:

```
npm install oracledb
```

Теперь вам нужно создать соединение ORACLE, которое вы можете позже запросить.

```
const oracledb = require('oracledb');

oracledb.getConnection(
  {
    user          : "oli",
    password      : "password",
    connectString : "ORACLE_DEV_DB_TNS_NAME"
  },
  connExecute
);
```

ConnectString «ORACLE\_DEV\_DB\_TNA\_NAME» может находиться в файле [tnsnames.org](#) в том же каталоге или где установлен ваш мгновенный клиент oracle.

Если на вашем компьютере разработки не установлен какой-либо мгновенный клиент oracle, вы можете следовать [instant client installation guide](#) по [instant client installation guide](#) для вашей операционной системы.

### Запрос объекта соединения без параметров

Теперь использование функции `connExecute-function` для выполнения запроса может быть использовано. У вас есть возможность получить результат запроса как объект или массив. Результат напечатан на `console.log`.

```
function connExecute(err, connection)
{
  if (err) {
    console.error(err.message);
    return;
  }
  sql = "select 'test' as c1, 'oracle' as c2 from dual";
  connection.execute(sql, {}, { outFormat: oracledb.OBJECT }, // or oracledb.ARRAY
    function(err, result)
    {
```

```

        if (err) {
            console.error(err.message);
            connRelease(connection);
            return;
        }
        console.log(result.metaData);
        console.log(result.rows);
        connRelease(connection);
    });
}

```

Поскольку мы использовали соединение без объединения, мы должны снова отправить наше соединение.

```

function connRelease(connection)
{
    connection.close(
        function(err) {
            if (err) {
                console.error(err.message);
            }
        });
}

```

**Выход для объекта будет**

```

[ { name: 'C1' }, { name: 'C2' } ]
[ { C1: 'test', C2: 'oracle' } ]

```

**и выход для массива будет**

```

[ { name: 'C1' }, { name: 'C2' } ]
[ [ 'test', 'oracle' ] ]

```

## Использование локального модуля для упрощения запросов

Чтобы упростить запрос из ORACLE-DB, вы можете вызвать свой запрос следующим образом:

```

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

const sql = "select 'test' as c1, 'oracle' as c2 from dual";
oracle.queryObject(sql, {}, {})
    .then(function(result) {
        console.log(result.rows[0]['C2']);
    })
    .catch(function(err) {
        next(err);
    });

```

Создание соединения и выполнение включено в этот файл oracle.js с содержимым следующим образом:

```

'use strict';
const oracledb = require('oracledb');

const oracleDbRelease = function(conn) {
  conn.release(function (err) {
    if (err)
      console.log(err.message);
  });
};

function queryArray(sql, bindParams, options) {
  options.isAutoCommit = false; // we only do SELECTs

  return new Promise(function(resolve, reject) {
    oracledb.getConnection(
      {
        user          : "oli",
        password      : "password",
        connectString : "ORACLE_DEV_DB_TNA_NAME"
      })
    .then(function(connection) {
      //console.log("sql log: " + sql + " params " + bindParams);
      connection.execute(sql, bindParams, options)
        .then(function(results) {
          resolve(results);
          process.nextTick(function() {
            oracleDbRelease(connection);
          });
        })
        .catch(function(err) {
          reject(err);

          process.nextTick(function() {
            oracleDbRelease(connection);
          });
        });
    })
    .catch(function(err) {
      reject(err);
    });
  });
}

function queryObject(sql, bindParams, options) {
  options['outFormat'] = oracledb.OBJECT; // default is oracledb.ARRAY
  return queryArray(sql, bindParams, options);
}

module.exports = queryArray;
module.exports.queryArray = queryArray;
module.exports.queryObject = queryObject;

```

Обратите внимание, что у вас есть оба метода `queryArray` и `queryObject` для вызова вашего объекта `oracle`.

Прочитайте Node.js с Oracle онлайн: <https://riptutorial.com/ru/node-js/topic/8248/node-js-c-oracle>

---

# глава 18: NodeJS Frameworks

## Examples

### Веб-серверные платформы

#### экспресс

```
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!');
});
```

#### Коа

```
var koa = require('koa');
var app = koa();

app.use(function *(next) {
  var start = new Date;
  yield next;
  var ms = new Date - start;
  console.log('%s %s - %s', this.method, this.url, ms);
});

app.use(function *(){
  this.body = 'Hello World';
});

app.listen(3000);
```

### Интерфейсы интерфейса командной строки

#### Commander.js

```
var program = require('commander');

program
  .version('0.0.1')

program
  .command('hi')
  .description('initialize project configuration')
```

```
.action(function(){
    console.log('Hi my Friend!!!');
});

program
    .command('bye [name]')
    .description('initialize project configuration')
    .action(function(name){
        console.log('Bye ' + name + '. It was good to see you!');
    });

program
    .command('*')
    .action(function(env){
        console.log('Enter a Valid command');
        terminate(true);
    });

program.parse(process.argv);
```

## Vorpal.js

```
const vorpal = require('vorpal')();

vorpal
    .command('foo', 'Outputs "bar".')
    .action(function(args, callback) {
        this.log('bar');
        callback();
    });

vorpal
    .delimiter('myapp$')
    .show();
```

Прочитайте NodeJS Frameworks онлайн: <https://riptutorial.com/ru/node-js/topic/6042/nodejs-frameworks>

---

# глава 19: NodeJS с Redis

## замечания

Мы рассмотрели основные и наиболее часто используемые операции в `node_redis`. Вы можете использовать этот модуль, чтобы использовать всю полноту Redis и создавать действительно сложные приложения Node.js. Вы можете создать много интересного с этой библиотекой, например, с сильным слоем кеширования, мощной системой обмена сообщениями Pub / Sub и т. Д. Чтобы узнать больше о библиотеке, ознакомьтесь с их [документацией](#).

## Examples

### Начиная

`node_redis`, как вы могли догадаться, является [клиентом Redis для Node.js](#). Вы можете установить его через npm, используя следующую команду.

```
npm install redis
```

После того, как вы установили модуль `node_redis`, вам хорошо идти. Давайте создадим простой файл `app.js` и посмотрим, как связаться с Redis от Node.js.

`app.js`

```
var redis = require('redis');
client = redis.createClient(); //creates a new client
```

По умолчанию `redis.createClient ()` будет использовать `127.0.0.1` и `6379` как имя хоста и порт соответственно. Если у вас есть другой хост / порт, вы можете предоставить их следующим образом:

```
var client = redis.createClient(port, host);
```

Теперь вы можете выполнить некоторые действия после установления соединения. В принципе, вам просто нужно прослушать события подключения, как показано ниже.

```
client.on('connect', function() {
  console.log('connected');
});
```

Итак, следующий фрагмент переходит в `app.js`:

```
var redis = require('redis');
var client = redis.createClient();

client.on('connect', function() {
  console.log('connected');
});
```

Теперь запустите приложение типа `node` в терминале. Перед запуском этого фрагмента убедитесь, что сервер `Redis` запущен и работает.

## Хранение пар ключей

Теперь, когда вы знаете, как подключиться к `Redis` от `Node.js`, давайте посмотрим, как хранить пары ключ-значение в хранилище `Redis`.

### Сохранение строк

Все команды `Redis` отображаются как различные функции на клиентском объекте. Чтобы сохранить простую строку, используйте следующий синтаксис:

```
client.set('framework', 'AngularJS');
```

Или же

```
client.set(['framework', 'AngularJS']);
```

В приведенных выше фрагментах хранится простая строка `AngularJS` против ключевой структуры. Вы должны заметить, что оба фрагмента делают то же самое. Единственное различие заключается в том, что первый передает переменное количество аргументов, а позже передает массив `args` функции `client.set()`. Вы также можете передать необязательный обратный вызов, чтобы получить уведомление, когда операция завершена:

```
client.set('framework', 'AngularJS', function(err, reply) {
  console.log(reply);
});
```

Если по какой-либо причине операция завершилась неудачей, аргумент `err` для обратного вызова представляет ошибку. Чтобы получить значение ключа, выполните следующие действия:

```
client.get('framework', function(err, reply) {
  console.log(reply);
});
```

`client.get()` позволяет получить ключ, сохраненный в `Redis`. Ключевое значение ключа можно получить через ответ аргумента обратного вызова. Если ключ не существует,



значение ответа будет пустым.

## Хранение хэша

Много раз хранение простых значений не решит вашу проблему. Вам нужно будет хранить хеши (объекты) в Redis. Для этого вы можете использовать `hmset()` следующим образом:

```
client.hmset('frameworks', 'javascript', 'AngularJS', 'css', 'Bootstrap', 'node', 'Express');

client.hgetall('frameworks', function(err, object) {
  console.log(object);
});
```

Вышеприведенный фрагмент хранит хэш в Redis, который отображает каждую технологию в свою структуру. Первый аргумент `hmset()` - это имя ключа. Последующие аргументы представляют пары ключ-значение. Аналогично, `hgetall()` используется для извлечения значения ключа. Если ключ найден, второй аргумент обратного вызова будет содержать значение, являющееся объектом.

Обратите внимание: Redis не поддерживает вложенные объекты. Все значения свойств объекта будут забиты в строки перед их сохранением. Вы также можете использовать следующий синтаксис для хранения объектов в Redis:

```
client.hmset('frameworks', {
  'javascript': 'AngularJS',
  'css': 'Bootstrap',
  'node': 'Express'
});
```

Дополнительный обратный вызов также может быть передан, чтобы знать, когда операция будет завершена.

Все функции (команды) можно вызывать с помощью прописных / строчных эквивалентов. Например, `client.hmset()` и `client.HMSET()` одинаковы. Сохранение списков

Если вы хотите сохранить список элементов, вы можете использовать списки Redis. Чтобы сохранить список, используйте следующий синтаксис:

```
client.rpush(['frameworks', 'angularjs', 'backbone'], function(err, reply) {
  console.log(reply); //prints 2
});
```

Вышеприведенный фрагмент создает список под названием `frameworks` и подталкивает к нему два элемента. Итак, длина списка теперь равна двум. Как вы можете видеть, я передал массив `args` для `rpush`. Первый элемент массива представляет собой имя ключа, в то время как остальные представляют элементы списка. Вы также можете использовать `lpush()` вместо `rpush()` для `rpush()` элементов влево.

Чтобы получить элементы списка, вы можете использовать `lrange()` следующим образом:

```
client.lrange('frameworks', 0, -1, function(err, reply) {
  console.log(reply); // ['angularjs', 'backbone']
});
```

Обратите внимание, что вы получаете все элементы списка, передавая `-1` в качестве третьего аргумента в `lrange()`. Если вы хотите подмножество списка, вы должны передать конечный индекс здесь.

## Хранение наборов

Наборы аналогичны спискам, но разница в том, что они не позволяют дублировать. Итак, если вы не хотите дублировать элементы в своем списке, вы можете использовать набор. Вот как мы можем изменить наш предыдущий фрагмент, чтобы использовать набор вместо списка.

```
client.sadd(['tags', 'angularjs', 'backbonejs', 'emberjs'], function(err, reply) {
  console.log(reply); // 3
});
```

Как вы можете видеть, `sadd()` создает новый набор с указанными элементами. Здесь длина набора равна трем. Чтобы получить элементы набора, используйте `smembers()` следующим образом:

```
client.smembers('tags', function(err, reply) {
  console.log(reply);
});
```

Этот фрагмент будет извлекать все элементы набора. Просто отметьте, что порядок не сохраняется при извлечении членов.

Это был список наиболее важных структур данных, найденных в каждом приложении Redis. Помимо строк, списков, наборов и хэшей вы можете хранить отсортированные наборы, `hyperLogLogs` и многое другое в Redis. Если вы хотите получить полный список команд и структур данных, посетите официальную документацию Redis. Помните, что почти каждая команда Redis отображается на объекте клиента, предлагаемом модулем `node_redis`.

## Некоторые более важные операции поддерживаются `node_redis`.

### Проверка наличия ключей

Иногда вам может потребоваться проверить, существует ли уже существующий ключ и действовать соответствующим образом. Для этого вы можете использовать функцию `exists()` как показано ниже:

```
client.exists('key', function(err, reply) {
  if (reply === 1) {
    console.log('exists');
  } else {
    console.log('doesn\'t exist');
  }
});
```

## Удаление и истечение ключей

Иногда вам нужно очистить некоторые ключи и повторно инициализировать их. Чтобы очистить ключи, вы можете использовать команду `del`, как показано ниже:

```
client.del('frameworks', function(err, reply) {
  console.log(reply);
});
```

Вы также можете указать срок действия существующего ключа следующим образом:

```
client.set('key1', 'val1');
client.expire('key1', 30);
```

Вышеприведенный фрагмент присваивает ключу времени истечения 30 секунд.

## Приращение и сокращение

Redis также поддерживает клавиши увеличения и уменьшения. Чтобы увеличить ключ, используйте функцию `incr()` как показано ниже:

```
client.set('key1', 10, function() {
  client.incr('key1', function(err, reply) {
    console.log(reply); // 11
  });
});
```

Функция `incr()` увеличивает значение ключа на 1. Если вам нужно увеличивать на другую величину, вы можете использовать `incrby()`. Аналогично, для уменьшения ключа вы можете использовать такие функции, как `decr()` и `decrby()`.

Прочитайте **NodeJS с Redis онлайн**: <https://riptutorial.com/ru/node-js/topic/7107/nodejs-c-redis>

---

# глава 20: nvm - Менеджер версий узлов

## замечания

URL-адреса, используемые в приведенных выше примерах, ссылаются на определенную версию диспетчера версий узлов. Скорее всего, последняя версия отличается от того, на что ссылаются. Чтобы установить nvm с использованием последней версии, [щелкните здесь](#), чтобы получить доступ к nvm на GitHub, который предоставит вам последние URL-адреса.

## Examples

### Установка NVM

Вы можете использовать `curl` :

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

Или вы можете использовать `wget` :

```
wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

### Проверьте версию NVM

Чтобы убедиться, что nvm установлен, выполните следующие действия:

```
command -v nvm
```

который должен выводить «nvm», если установка прошла успешно.

### Установка конкретной версии узла

Список доступных удаленных версий для установки

```
nvm ls-remote
```

### Установка удаленной версии

```
nvm install <version>
```

Например

```
nvm install 0.10.13
```

## Использование уже установленной версии узла

Чтобы просмотреть доступные локальные версии узла через NVM:

```
nvm ls
```

Например, если `nvm ls` возвращает:

```
$ nvm ls
  v4.3.0
  v5.5.0
```

Вы можете переключиться на `v5.5.0` с помощью:

```
nvm use v5.5.0
```

## Установите nvm на Mac OSX

# ПРОЦЕСС УСТАНОВКИ

Вы можете установить Node Version Manager с помощью `git`, `curl` или `wget`. Вы выполняете эти команды в **терминале** на **Mac OSX**.

**пример скручивания:**

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

**Пример wget:**

```
wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

## ИСПЫТАНИЕ, ЧТО NVM БЫЛО УСТАНОВЛЕНА

Чтобы проверить правильность установки `nvm`, закройте и снова откройте терминал и введите `nvm`. Если вы получите сообщение **nvm: command not found**, ваша ОС может не иметь необходимого файла **.bash\_profile**. В терминале введите `touch ~/.bash_profile` и снова запустите указанный выше сценарий установки.

Если вы все еще получаете **команду nvm: не найдена**, попробуйте следующее:

- В терминале введите `nano .bashrc`. Вы должны увидеть сценарий экспорта, почти идентичный следующему:

```
экспорт NVM_DIR = "/Users/johndoe/.nvm" [-s "$NVM_DIR/nvm.sh"] &&. «$NVM_DIR/nvm.sh»
```

- Скопируйте сценарий экспорта и удалите его из **.bashrc**
- Сохраните и закройте файл **.bashrc** (CTRL + O - Enter - CTRL + X)
- Затем введите `nano .bash_profile` чтобы открыть профиль Bash
- Вставьте скрипт экспорта, который вы скопировали в профиль Bash на новой строке.
- Сохранить и закрыть профиль Bash (CTRL + O - Enter - CTRL + X)
- Наконец, введите `nano .bashrc` чтобы повторно открыть файл **.bashrc**
- Вставьте следующую строку в файл:

```
источник ~/.nvm/nvm.sh
```

- Сохранить и закрыть (CTRL + O - Enter - CTRL + X)
- Перезапустите терминал и введите `nvm` чтобы проверить, работает ли он

## Установка псевдонима для версии узла

Если вы хотите установить некоторое имя псевдонима для установленной версии узла, выполните следующие действия:

```
nvm alias <name> <version>
```

Аналогичным образом, чтобы:

```
nvm unalias <name>
```

Правильная usecase будет, если вы хотите установить какую-либо другую версию, чем стабильную, как псевдоним по умолчанию. `default` aliased версии загружаются на консоль по умолчанию.

Подобно:

```
nvm alias default 5.0.1
```

Тогда каждый раз, когда **консоль** / **терминал** запускает 5.0.1, будет присутствовать по умолчанию.

Замечания:

```
nvm alias # lists all aliases created on nvm
```

## Запустите любую произвольную команду в оболочке с нужной версией узла

## Список всех установленных версий узлов

```
nvm ls
  v4.5.0
  v6.7.0
```

## Выполнить команду с использованием любой установленной версии узла

```
nvm run 4.5.0 --version or nvm exec 4.5.0 node --version
Running node v4.5.0 (npm v2.15.9)
v4.5.0
```

```
nvm run 6.7.0 --version or nvm exec 6.7.0 node --version
Running node v6.7.0 (npm v3.10.3)
v6.7.0
```

## Используя псевдоним

```
nvm run default --version or nvm exec default node --version
Running node v6.7.0 (npm v3.10.3)
v6.7.0
```

## Чтобы установить версию узла LTS

```
nvm install --lts
```

## Переключение версий

```
nvm use v4.5.0 or nvm use stable ( alias )
```

Прочитайте [nvm - Менеджер версий узлов онлайн: https://riptutorial.com/ru/node-js/topic/2823/nvm---менеджер-версий-узлов](https://riptutorial.com/ru/node-js/topic/2823/nvm---менеджер-версий-узлов)

# глава 21: OAuth 2.0

## Examples

### OAuth 2 с реализацией Redis - grant\_type: пароль

В этом примере я буду использовать oauth2 в rest api с базой данных redis

**Важно:** вам нужно будет установить базу данных redis на вашем компьютере, загрузить ее [здесь](#) для пользователей Linux и [отсюда](#), чтобы установить версию Windows, и мы будем использовать настольное приложение для менеджера redis, установите его [здесь](#) .

Теперь нам нужно настроить наш сервер node.js на использование базы данных redis.

- **Создание файла сервера: app.js**

```
var express = require('express'),
    bodyParser = require('body-parser'),
    oauthserver = require('oauth2-server'); // Would be: 'oauth2-server'

var app = express();

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

app.use(bodyParser.json());

app.oauth = oauthserver({
  model: require('./routes/Oauth2/model'),
  grants: ['password', 'refresh_token'],
  debug: true
});

// Handle token grant requests
app.all('/oauth/token', app.oauth.grant());

app.get('/secret', app.oauth.authorise(), function (req, res) {
  // Will require a valid access_token
  res.send('Secret area');
});

app.get('/public', function (req, res) {
  // Does not require an access_token
  res.send('Public area');
});

// Error handling
app.use(app.oauth.errorHandler());

app.listen(3000);
```



## • Создайте модель OAuth2 в маршрутах / OAuth2 / model.js

```
var model = module.exports,
    util = require('util'),
    redis = require('redis');

var db = redis.createClient();

var keys = {
  token: 'tokens:%s',
  client: 'clients:%s',
  refreshToken: 'refresh_tokens:%s',
  grantTypes: 'clients:%s:grant_types',
  user: 'users:%s'
};

model.getAccessToken = function (bearerToken, callback) {
  db.hgetAll(util.format(keys.token, bearerToken), function (err, token) {
    if (err) return callback(err);

    if (!token) return callback();

    callback(null, {
      accessToken: token.accessToken,
      clientId: token.clientId,
      expires: token.expires ? new Date(token.expires) : null,
      userId: token.userId
    });
  });
};

model.getClient = function (clientId, clientSecret, callback) {
  db.hgetAll(util.format(keys.client, clientId), function (err, client) {
    if (err) return callback(err);

    if (!client || client.clientSecret !== clientSecret) return callback();

    callback(null, {
      clientId: client.clientId,
      clientSecret: client.clientSecret
    });
  });
};

model.getRefreshToken = function (bearerToken, callback) {
  db.hgetAll(util.format(keys.refreshToken, bearerToken), function (err, token) {
    if (err) return callback(err);

    if (!token) return callback();

    callback(null, {
      refreshToken: token.accessToken,
      clientId: token.clientId,
      expires: token.expires ? new Date(token.expires) : null,
      userId: token.userId
    });
  });
};

model.grantTypeAllowed = function (clientId, grantType, callback) {
```

```

    db.sismember(util.format(keys.grantTypes, clientId), grantType, callback);
};

model.saveAccessToken = function (accessToken, clientId, expires, user, callback) {
    db.hmset(util.format(keys.token, accessToken), {
        accessToken: accessToken,
        clientId: clientId,
        expires: expires ? expires.toISOString() : null,
        userId: user.id
    }, callback);
};

model.saveRefreshToken = function (refreshToken, clientId, expires, user, callback) {
    db.hmset(util.format(keys.refreshToken, refreshToken), {
        refreshToken: refreshToken,
        clientId: clientId,
        expires: expires ? expires.toISOString() : null,
        userId: user.id
    }, callback);
};

model.getUser = function (username, password, callback) {
    db.hgetall(util.format(keys.user, username), function (err, user) {
        if (err) return callback(err);

        if (!user || password !== user.password) return callback();

        callback(null, {
            id: username
        });
    });
};
};
};

```

Вам нужно только установить redis на свой компьютер и запустить следующий файл узла

```

#!/usr/bin/env node

var db = require('redis').createClient();

db.multi()
  .hmset('users:username', {
    id: 'username',
    username: 'username',
    password: 'password'
  })
  .hmset('clients:client', {
    clientId: 'client',
    clientSecret: 'secret'
  })//clientId + clientSecret to base 64 will generate Y2xpZW50OnNlY3JldA==
  .sadd('clients:client:grant_types', [
    'password',
    'refresh_token'
  ])
  .exec(function (errs) {
    if (errs) {
      console.error(errs[0].message);
      return process.exit(1);
    }

    console.log('Client and user added successfully');
  });

```

```
process.exit();
});
```

**Примечание** . Этот файл установит учетные данные для вашего интерфейса, чтобы запросить токен. Таким образом, ваш запрос от

Образец базы данных redis после вызова вышеуказанного файла:

Redis Desktop Manager v.0.9.0

local::db0::users:username ✕

**HASH:** users:username

row	key	value
1	id	username
2	username	username
3	password	password

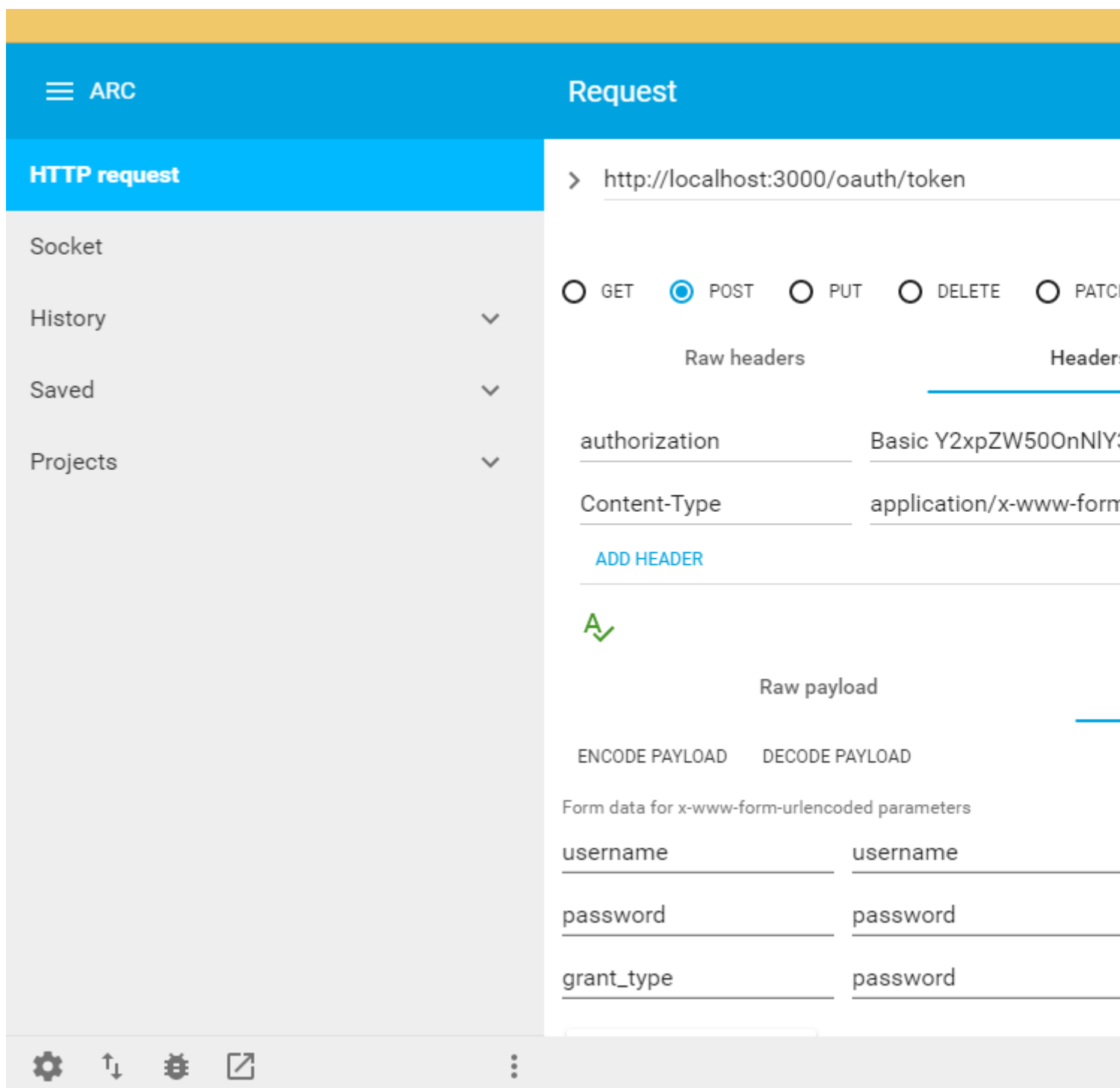
Key: size in bytes: 0

Value: size in bytes: 0

2017-03-28 18:16:02 : Connection: local > Response re  
2017-03-28 18:16:02 : Connection: local > [runComma  
2017-03-28 18:16:02 : Connection: local > Response re  
2017-03-28 18:16:02 : Connection: local > [runComma  
2017-03-28 18:16:02 : Connection: local > Response re

Import / Export    + Connect to Redis Server    System log ✕

Запрос будет выглядеть следующим образом:



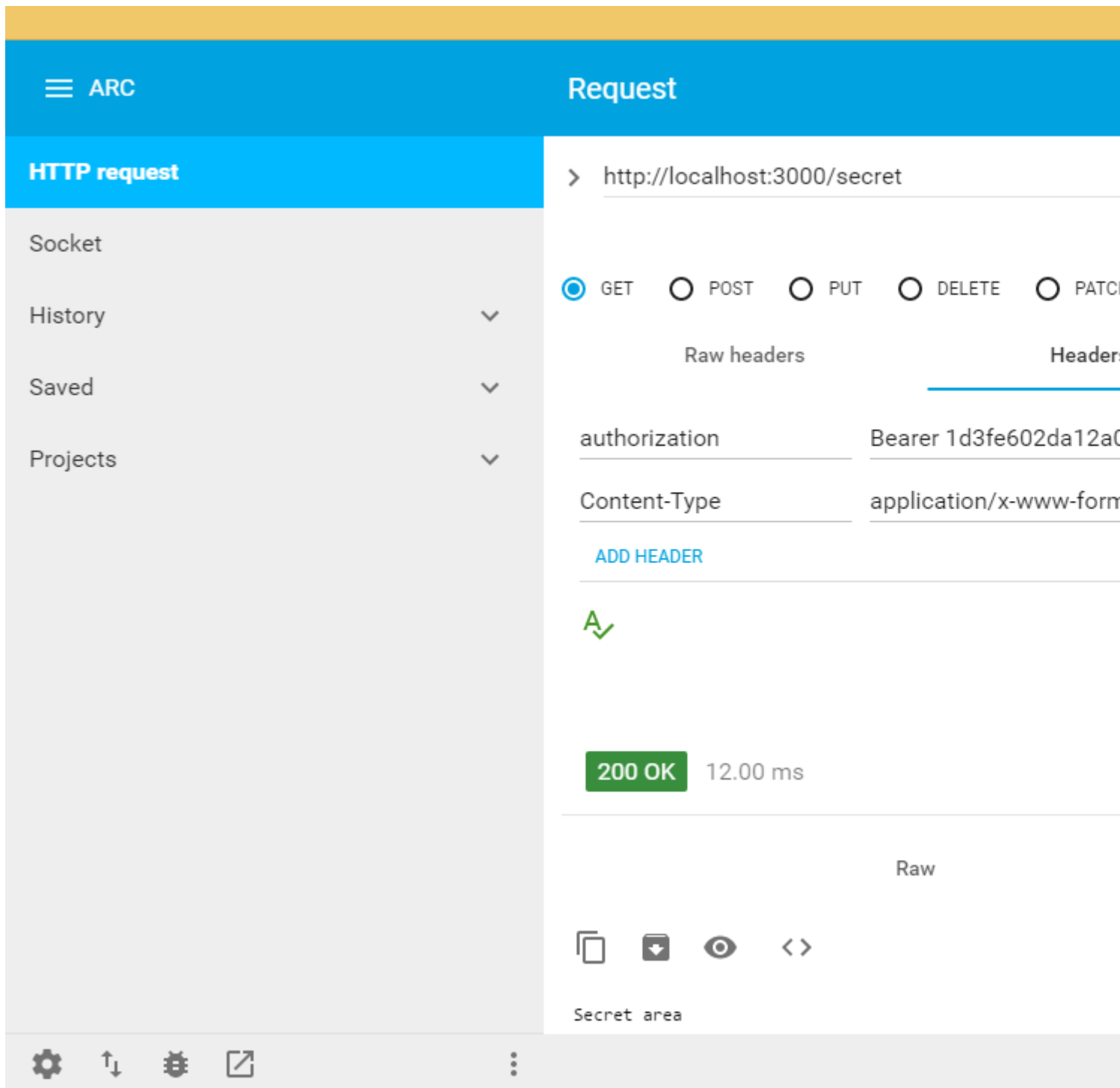
Заголовок:

1. авторизация: Basic, за которой следует пароль, когда вы впервые настроили redis:
  - a. clientId + secretId to base64
2. Форма данных:
  - имя пользователя: пользователь, запрашивающий токен
  - пароль: пароль пользователя

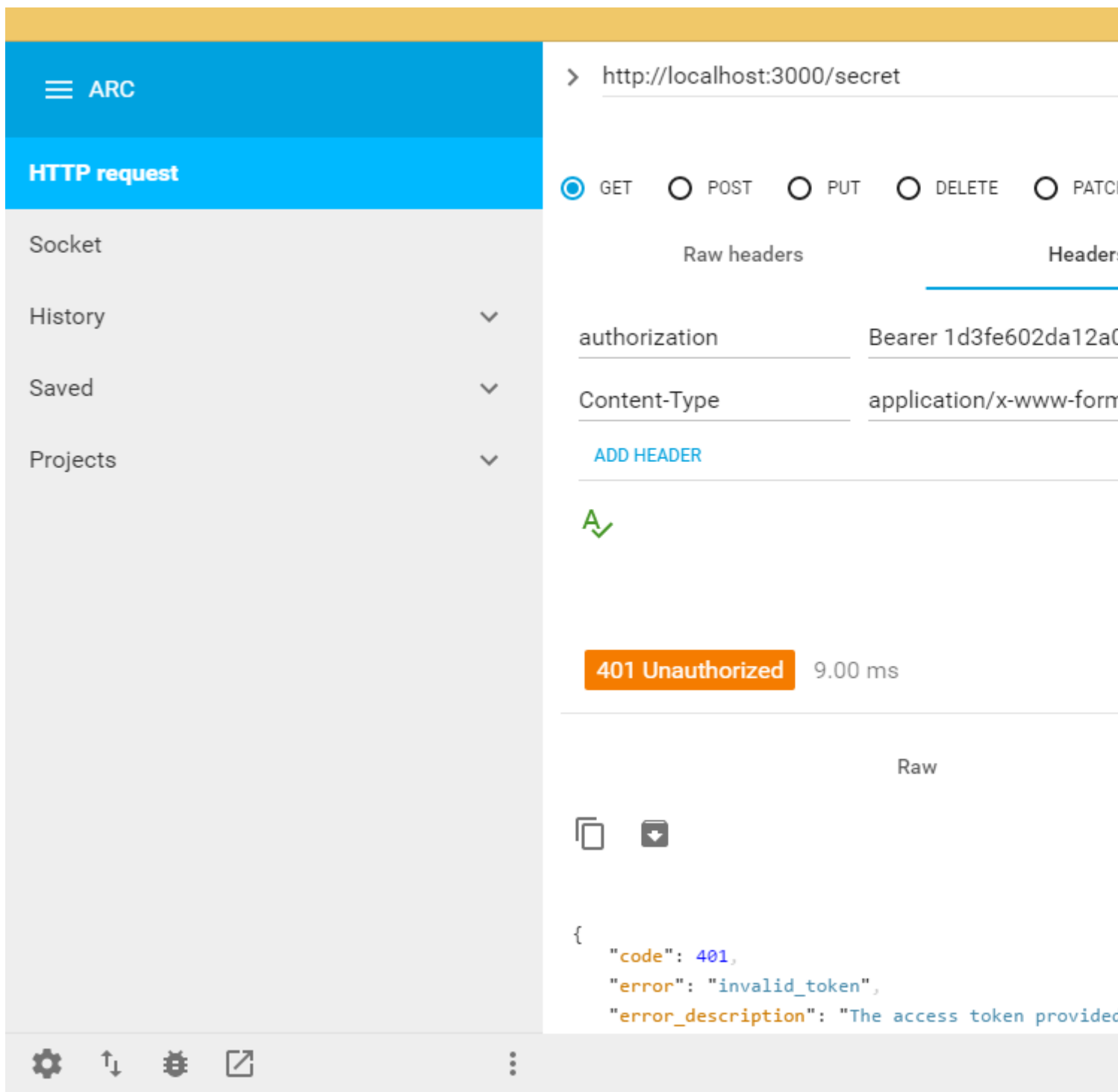
grant\_type: зависит от того, какие параметры вы хотите, я выбираю password, который принимает только имя пользователя и пароль, которые будут созданы в redis, Data on redis будет выглядеть следующим образом:

```
{
  "access_token": "1d3fe602da12a086ecb2b996fd7b7ae874120c4f",
  "token_type": "bearer", // Will be used to access api + access+token e.g. bearer
  1d3fe602da12a086ecb2b996fd7b7ae874120c4f
  "expires_in": 3600,
  "refresh_token": "b6ad56e5c9aba63c85d7e21b1514680bbf711450"
}
```

Поэтому нам нужно вызвать наш api и захватить некоторые защищенные данные с помощью нашего токена доступа, который мы только что создали, см. Ниже:



когда токен истекает, аri выдает ошибку, по которой истекает токен, и вы не можете иметь доступ к любому из вызовов аri, см. изображение ниже:



Давайте посмотрим, что делать, если токен истекает. Позвольте мне сначала объяснить вам это, если токен доступа истекает, refresh\_token существует в redis, ссылаясь на expired access\_token. Итак, нам нужно снова вызвать oauth / токен с refresh\_token grant\_type и установить авторизацию на Basic clientId: clientsecret (до базы 64!) и, наконец, отправить refresh\_token, это создаст новый access\_token с новыми данными истечения срока действия.

На следующем рисунке показано, как получить новый токен доступа:

The screenshot displays the ARC tool interface for configuring an HTTP request. The top bar shows the ARC logo and the title "Request". The left sidebar contains navigation options: "HTTP request" (selected), "Socket", "History", "Saved", and "Projects". The main area shows the request configuration for the URL "http://localhost:3000/oauth/token". The method is set to "POST". The "Raw headers" section includes "authorization: Basic Y2xpZW50OnNIY..." and "Content-Type: application/x-www-form...". The "Raw payload" section shows form data for "refresh\_token" (value: b6ad56e5c9aba63c85d7) and "grant\_type" (value: refresh\_token). A bottom bar contains icons for settings, navigation, and a menu.

## Надеюсь помочь!

Прочитайте OAuth 2.0 онлайн: <https://riptutorial.com/ru/node-js/topic/9566/oauth-2-0>



# глава 22: package.json

## замечания

Вы можете создать `package.json` с

```
npm init
```

который задаст вам основные факты о ваших проектах, включая [идентификатор лицензии](#)

## Examples

### Определение базового проекта

```
{
  "name": "my-project",
  "version": "0.0.1",
  "description": "This is a project.",
  "author": "Someone <someone@example.com>",
  "contributors": [{
    "name": "Someone Else",
    "email": "else@example.com"
  }],
  "keywords": ["improves", "searching"]
}
```

поле	Описание
название	<b>необходимое</b> поле для установки пакета. Нужно быть строчным, одно слово без пробелов. (Разрешены штрихи и подчеркивания)
версия	<b>обязательное</b> поле для версии пакета с использованием <a href="#">семантического управления версиями</a> .
описание	краткое описание проекта
автор	указывает автор пакета
авторы	массив объектов, по одному для каждого вкладчика
ключевые слова	массив строк, это поможет людям найти ваш пакет

## ЗАВИСИМОСТИ

```
"dependencies": {"module-name": "0.1.0"}
```

- **точный** : 0.1.0 установит эту конкретную версию модуля.
- **новейшая второстепенная версия** : ^0.1.0 установит новейшую второстепенную версию, например 0.2.0 , но не будет устанавливать модуль с более высокой основной версией, например 1.0.0
- **самый новый патч** : 0.1.x или ~0.1.0 установит новейшую версию патча, например 0.1.4 , но не будет устанавливать модуль с более крупной или младшей версией, например 0.2.0 или 1.0.0 .
- **wildcard** : \* будет установлена последняя версия модуля.
- **git** : следующее установит tarball из главной ветви git-репо. Также можно предоставить #sha , #tag или #branch :
  - **GitHub** : user/project или user/project#v1.0.0
  - **url** : git://gitlab.com/user/project.git или git://gitlab.com/user/project.git#develop
- **локальный путь** : file:../lib/project

После добавления их в файл package.json, используйте команду `npm install` в каталоге проекта в терминале.

## devDependencies

```
"devDependencies": {  
  "module-name": "0.1.0"  
}
```

Для зависимостей, необходимых только для разработки, например, для тестирования прокси-серверов ext. Эти dev-зависимости не будут установлены при запуске «npm install» в режиме производства.

## Сценарии

Вы можете определить сценарии, которые могут быть выполнены или запускаться до или после другого сценария.

```
{  
  "scripts": {  
    "pretest": "scripts/pretest.js",  
    "test": "scripts/test.js",  
    "posttest": "scripts/posttest.js"  
  }  
}
```

В этом случае вы можете выполнить скрипт, выполнив одну из следующих команд:

```
$ npm run-script test  
$ npm run test
```

```
$ npm test
$ npm t
```

## Предварительно определенные сценарии

Имя скрипта	Описание
prepublish	Запуск до публикации пакета.
публиковать, публиковать	Запуск после публикации пакета.
предустанавливать	Запустите перед установкой пакета.
установить, postinstall	Запустите после установки пакета.
preuninstall, удалить	Запустите перед удалением пакета.
postuninstall	Запуск после удаления пакета.
предисловие, версия	Запустите перед выпуском версию пакета.
postversion	Запустите после выпуска версию пакета.
предварительный тест, тест, посттест	Запуск командой <code>npm test</code>
престоп, стоп, постстоп	Запуск командой <code>npm stop</code>
предварительный, старт, poststart	Запуск командой <code>npm start</code>
prerestart, restart, postrestart	Запустить команду <code>npm restart</code>

## Пользовательские скрипты

Вы также можете определить свои собственные сценарии так же, как и с заранее определенными сценариями:

```
{
  "scripts": {
    "preci": "scripts/preci.js",
    "ci": "scripts/ci.js",
    "postci": "scripts/postci.js"
  }
}
```

В этом случае вы можете выполнить скрипт, выполнив одну из следующих команд:

```
$ npm run-script ci
$ npm run ci
```

Пользовательские скрипты также поддерживают сценарии *pre* и *post*, как показано в примере выше.

## Расширенное определение проекта

Некоторые дополнительные атрибуты анализируются веб-сайтом npm, например `repository`, `bugs` или `homepage` и показаны в инфобоксах для этих пакетов

```
{
  "main": "server.js",
  "repository": {
    "type": "git",
    "url": "git+https://github.com/<accountname>/<repositoryname>.git"
  },
  "bugs": {
    "url": "https://github.com/<accountname>/<repositoryname>/issues"
  },
  "homepage": "https://github.com/<accountname>/<repositoryname>#readme",
  "files": [
    "server.js", // source files
    "README.md", // additional files
    "lib" // folder with all included files
  ]
}
```

поле	Описание
главный	Вступительный скрипт для этого пакета. Этот скрипт возвращается, когда пользователь требует пакет.
хранилище	Местоположение и тип публичного хранилища
ошибки	Bugtracker для этого пакета (например, github)
домашняя страница	Домашняя страница для этого пакета или общий проект
файлы	Список файлов и папок, которые должны быть загружены, когда пользователь делает <code>npm install &lt;packagename&gt;</code>

## Изучение package.json

Файл `package.json`, обычно присутствующий в корне проекта, содержит метаданные о вашем приложении или модуле, а также список зависимостей для установки из npm при запуске `npm install`.

Для инициализации `package.json` введите `npm init` в командной строке.

Чтобы создать `package.json` со значениями по умолчанию, используйте:

```
npm init --yes
# or
npm init -y
```

Чтобы установить пакет и сохранить его в `package.json` используйте:

```
npm install {package name} --save
```

Вы также можете использовать сокращенную нотацию:

```
npm i -S {package name}
```

`--save` **NPM** `-S` для `--save` и `-D` to `--save-dev` для сохранения в ваших зависимостях производства или разработки соответственно.

Пакет будет отображаться в ваших зависимостях; если вы используете `--save-dev` вместо `--save`, пакет появится в ваших `devDependencies`.

Важные свойства `package.json`:

```
{
  "name": "module-name",
  "version": "10.3.1",
  "description": "An example module to illustrate the usage of a package.json",
  "author": "Your Name <your.name@example.org>",
  "contributors": [{
    "name": "Foo Bar",
    "email": "foo.bar@example.com"
  }],
  "bin": {
    "module-name": "./bin/module-name"
  },
  "scripts": {
    "test": "vows --spec --isolate",
    "start": "node index.js",
    "predeploy": "echo About to deploy",
    "postdeploy": "echo Deployed",
    "prepublish": "coffee --bare --compile --output lib/foo src/foo/*.coffee"
  },
  "main": "lib/foo.js",
  "repository": {
    "type": "git",
    "url": "https://github.com/username/repo"
  },
  "bugs": {
    "url": "https://github.com/username/issues"
  },
  "keywords": [
    "example"
  ],
}
```

```
"dependencies": {
  "express": "4.2.x"
},
"devDependencies": {
  "assume": "<1.0.0 || >=2.3.1 <2.4.5 || >=2.5.2 <3.0.0"
},
"peerDependencies": {
  "moment": ">2.0.0"
},
"preferGlobal": true,
"private": true,
"publishConfig": {
  "registry": "https://your-private-hosted-npm.registry.domain.com"
},
"subdomain": "foobar",
"analyze": true,
"license": "MIT",
"files": [
  "lib/foo.js"
]
}
```

Информация о некоторых важных свойствах:

name

Уникальное имя вашего пакета и должно быть внизу в нижнем регистре. Это свойство требуется, и ваш пакет не будет установлен без него.

1. Имя должно быть меньше или равно 214 символам.
2. Имя не может начинаться с точки или подчеркивания.
3. Новые пакеты не должны иметь заглавных букв в названии.

version

Версия пакета определяется [семантической](#) версией (semver). Что предполагает, что номер версии записывается как MAJOR.MINOR.PATCH, и вы увеличиваете:

1. ОСНОВНАЯ версия, когда вы делаете несовместимые изменения API
2. MINOR, когда вы добавляете функциональность обратно-совместимым образом
3. PATCH, когда вы делаете исправления ошибок с обратной совместимостью

description

Описание проекта. Постарайтесь держать его коротким и кратким.

author

Автор этого пакета.

bin

Объект, который используется для отображения двоичных скриптов из вашего пакета. Объект предполагает, что ключ - это имя бинарного скрипта, а значение - относительный путь к скрипту.

Это свойство используется пакетами, которые содержат интерфейс командной строки (CLI).

```
script
```

Объект, который предоставляет дополнительные команды прт. Объект предполагает, что ключ - это команда прт, а значение - путь к скрипту. Эти сценарии могут выполняться при запуске `npm run {command name}` или `npm run-script {command name}`.

Пакеты, содержащие интерфейс командной строки и устанавливаемые локально, могут быть вызваны без относительного пути. Поэтому вместо вызова `./node_modules/.bin/mocha` вы можете напрямую вызвать `mocha`.

```
main
```

Основная точка входа в ваш пакет. При вызове `require('{module name}')` в узле, это будет фактический файл, который требуется.

Настоятельно рекомендуется, чтобы основной файл не вызывал никаких побочных эффектов. Например, требующий основного файла не должен запускать HTTP-сервер или подключаться к базе данных. Вместо этого вы должны создать что-то вроде `exports.init = function () {...}` в своем основном скрипте.

```
keywords
```

Массив ключевых слов, которые описывают ваш пакет. Это поможет людям найти ваш пакет.

```
devDependencies
```

Это зависимости, которые предназначены только для разработки и тестирования вашего модуля. Зависимости будут установлены автоматически, если не `NODE_ENV=production` переменная `NODE_ENV=production` среды `NODE_ENV=production`. Если это так, вы все равно можете использовать эти пакеты, используя `npm install --dev`

```
peerDependencies
```

Если вы используете этот модуль, то `peerDependencies` отображает модули, которые вы должны установить рядом с этим. Например, `moment-timezone` должен устанавливаться рядом с `moment` потому что это плагин на мгновение, даже если он прямо не `require("moment")`.

```
preferGlobal
```

Свойство, указывающее, что эта страница предпочитает устанавливать глобально, используя `npm install -g {module-name}`. Это свойство используется пакетами, которые содержат интерфейс командной строки (CLI).

Во всех других ситуациях вы НЕ должны использовать это свойство.

```
publishConfig
```

`PublishConfig` - это объект со значениями конфигурации, которые будут использоваться для публикации модулей. Установленные значения конфигурации переопределяют вашу конфигурацию `npm` по умолчанию.

Наиболее распространенное использование `publishConfig` состоит в том, чтобы опубликовать ваш пакет в приватном реестре `npm`, чтобы у вас все еще были преимущества `npm`, но для частных пакетов. Это делается простым настройкой URL вашего частного `npm` как значения для ключа реестра.

```
files
```

Это массив всех файлов для включения в опубликованный пакет. Можно использовать путь к файлу или путь к папке. Будет включено все содержимое пути к папке. Это уменьшает общий размер вашего пакета, включая только правильные файлы для распространения. Это поле работает в сочетании с `.npmignore` правил `.npmignore`.

[Источник](#)

Прочитайте `package.json` онлайн: <https://riptutorial.com/ru/node-js/topic/1515/package-json>



---

# глава 23: passport.js

## Вступление

Паспорт является популярным модулем авторизации для узла. Простыми словами он обрабатывает все запросы авторизации на вашем приложении пользователями. Passport поддерживает более 300 стратегий, так что вы можете легко интегрировать логин с Facebook / Google или любой другой социальной сетью, используя его. Стратегия, которую мы обсудим здесь, это Local, где вы аутентифицируете пользователя, используя свою собственную базу данных зарегистрированных пользователей (используя имя пользователя и пароль).

## Examples

### Пример LocalStrategy in passport.js

```
var passport = require('passport');
var LocalStrategy = require('passport-local').Strategy;

passport.serializeUser(function(user, done) { //In serialize user you decide what to store in
the session. Here I'm storing the user id only.
  done(null, user.id);
});

passport.deserializeUser(function(id, done) { //Here you retrieve all the info of the user
from the session storage using the user id stored in the session earlier using serialize user.
  db.findById(id, function(err, user) {
    done(err, user);
  });
});

passport.use(new LocalStrategy(function(username, password, done) {
  db.findOne({'username':username},function(err, student){
    if(err)return done(err, {message:message}); //wrong roll_number or password;
    var pass_retrieved = student.pass_word;
    bcrypt.compare(password, pass_retrieved, function(err3, correct) {
      if(err3){
        message = [{"msg": "Incorrect Password!"}];
        return done(null,false, {message:message}); // wrong password
      }
      if(correct){
        return done(null,student);
      }
    });
  });
}));

app.use(session({ secret: 'super secret' })); //to make passport remember the user on other
pages too.(Read about session store. I used express-sessions.)
app.use(passport.initialize());
app.use(passport.session());
```

```
app.post('/',passport.authenticate('local',{successRedirect:'/users' failureRedirect: '/'}),
  function(req,res,next){
  });
```

Прочитайте passport.js онлайн: <https://riptutorial.com/ru/node-js/topic/8812/passport-js>

---

# глава 24: Readline

## Синтаксис

- `const readline = require('readline')`
- `readline.close ()`
- `readline.pause ()`
- `readline.prompt ([preserveCursor])`
- `readline.question` (запрос, обратный вызов)
- `readline.resume ()`
- `readline.setPrompt` (приглашение)
- `readline.write` (данные [, ключ])
- `readline.clearLine` (поток, каталог)
- `readline.clearScreenDown` (поток)
- `readline.createInterface` (варианты)
- `readline.cursorTo` (поток, x, y)
- `readline.emitKeypressEvents` (поток [, интерфейс])
- `readline.moveCursor` (поток, dx, dy)

## Examples

### Пошаговое чтение файлов

```
const fs = require('fs');
const readline = require('readline');

const rl = readline.createInterface({
  input: fs.createReadStream('text.txt')
});

// Each new line emits an event - every time the stream receives \r, \n, or \r\n
rl.on('line', (line) => {
  console.log(line);
});

rl.on('close', () => {
  console.log('Done reading file');
});
```

### Вызов пользовательского ввода через CLI

```
const readline = require('readline');

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});
```

```
rl.question('What is your name?', (name) => {  
  console.log(`Hello ${name}!`);  
  
  rl.close();  
});
```

Прочитайте Readline онлайн: <https://riptutorial.com/ru/node-js/topic/1431/readline>

# глава 25: Require ()

## Вступление

В этой документации основное внимание уделяется объяснению использования и инструкции `require()` которую **NodeJS** включает в свой язык.

Требовать - это импорт определенных файлов или пакетов, используемых с модулями NodeJS. Он используется для улучшения структуры кода и использования. `require()` используется для файлов, которые установлены локально, с прямым маршрутом из файла, который `require`.

## Синтаксис

- `module.exports = {testFunction: testFunction};`
- `var test_file = require ('./ testFile.js');` // У нас есть файл с именем `testFile`
- `test_file.testFunction (our_data);` // Пусть `testFile` имеет функцию `testFunction`

## замечания

Использование `require()` позволяет структурировать код таким образом, который аналогичен использованию Java **классами** и общедоступными методами. Если функция имеет значение `.export`, она может `require` в другом файле, который будет использоваться. Если файл не `.export 'ed`, его нельзя использовать в другом файле.

## Examples

### Начало `require ()` использование с функцией и файлом

Требовать - это утверждение, которое узел интерпретирует как, в некотором смысле, функцию `getter`. Например, скажем, у вас есть файл с именем `analysis.js`, и внутренняя часть вашего файла выглядит так:

```
function analyzeWeather(weather_data) {
  console.log('Weather information for ' + weather_data.time + ': ');
  console.log('Rainfall: ' + weather_data.precip);
  console.log('Temperature: ' + weather_data.temp);
  //More weather_data analysis/printing...
}
```

Этот файл содержит только метод, `analyzeWeather(weather_data)`. Если мы хотим использовать эту функцию, она должна быть либо использована внутри этого файла, либо скопирована в файл, который он хочет использовать. Однако Node включил очень

полезный инструмент, помогающий с организацией кода и файлов, которые являются [модулями](#) .

Чтобы использовать нашу функцию, мы должны сначала `export` функцию через инструкцию в начале. Наш новый файл выглядит так:

```
module.exports = {
  analyzeWeather: analyzeWeather
}
function analyzeWeather(weather_data) {
  console.log('Weather information for ' + weather_data.time + ': ');
  console.log('Rainfall: ' + weather_data.precip);
  console.log('Temperature: ' + weather_data.temp);
  //More weather_data analysis/printing...
}
```

С помощью этого небольшого оператора `module.exports` наша функция теперь готова к использованию вне файла. Все, что осталось сделать, это использовать `require()` .

Если `require` функция или файл, синтаксис очень похож. Это обычно делается в начале файла и установить его в `var` -й или `const` 's для использования в течение всего файла. Например, у нас есть еще один файл (на том же уровне, что и у `analyze.js` именем `handleWeather.js` который выглядит так,

```
const analysis = require('./analysis.js');

weather_data = {
  time: '01/01/2001',
  precip: 0.75,
  temp: 78,
  //More weather data...
};
analysis.analyzeWeather(weather_data);
```

В этом файле, мы используем `require()` , чтобы захватить наш `analysis.js` файл. При использовании мы просто вызываем переменную или константу, назначенные этому `require` и используем любую функцию внутри экспортируемой.

## Начало `require ()` использование с пакетом NPM

Узла `require` также очень полезно при использовании в сочетании с [пакетом НПМ](#) .

Скажем, к примеру, вы хотели бы использовать пакет НПМ `require` в файле с именем `getWeather.js` . После того, как [NPM установит](#) ваш пакет через вашу командную строку ( `git install request` ), вы готовы использовать его. `getWeather.js` ваш файл `getWeather.js` выглядит так,

```
var https = require('request');

//Construct your url variable...
https.get(url, function(error, response, body) {
```

```
if (error) {
  console.log(error);
} else {
  console.log('Response => ' + response);
  console.log('Body => ' + body);
}
});
```

Когда этот файл запущен, сначала `require «s»` (импорт) только что установленного пакета с `request` . Внутри файла `request` есть много функций, к которым у вас есть доступ, один из которых называется `get` . В следующих парах строки функция используется для выполнения [запроса HTTP GET](#) .

Прочитайте [Require \(\) онлайн: https://riptutorial.com/ru/node-js/topic/10742/require---](#)

---

# глава 26: Sequelize.js

## Examples

### Монтаж

Убедитесь, что у вас сначала установлены Node.js и npm. Затем установите sequelize.js с npm

```
npm install --save sequelize
```

Вам также потребуется установить поддерживаемые модули базы данных Node.js. Вам нужно только установить тот, который вы используете

Для MySQL и Mariadb

```
npm install --save mysql
```

Для PostgreSQL

```
npm install --save pg pg-hstore
```

Для SQLite

```
npm install --save sqlite
```

Для MSSQL

```
npm install --save tedious
```

После установки вы можете включить и создать новый экземпляр Sequelize.

### Синтаксис ES5

```
var Sequelize = require('sequelize');  
var sequelize = new Sequelize('database', 'username', 'password');
```

### ES6 stage-0 Вавилонский синтаксис

```
import Sequelize from 'sequelize';  
const sequelize = new Sequelize('database', 'username', 'password');
```

Теперь у вас есть экземпляр sequelize. Вы можете, если вы так склонены, называть это другим именем, таким как



```
var db = new Sequelize('database', 'username', 'password');
```

или же

```
var database = new Sequelize('database', 'username', 'password');
```

эта часть - ваша прерогатива. После установки вы можете использовать его внутри своего приложения в соответствии с документацией API

<http://docs.sequelizejs.com/en/v3/api/sequelize/>

Следующим шагом после установки будет [настройка вашей собственной модели](#)

## Определение моделей

Существует два способа определения моделей в sequelize; с `sequelize.define(...)` или `sequelize.import(...)`. Обе функции возвращают объект модели sequelize.

# 1. sequelize.define (modelName, attributes, [options])

Это путь, если вы хотите определить все свои модели в одном файле или если вы хотите иметь дополнительный контроль над определением вашей модели.

```
/* Initialize Sequelize */
const config = {
  username: "database username",
  password: "database password",
  database: "database name",
  host: "database's host URL",
  dialect: "mysql" // Other options are postgres, sqlite, mariadb and mssql.
}
var Sequelize = require("sequelize");
var sequelize = new Sequelize(config);

/* Define Models */
sequelize.define("MyModel", {
  name: Sequelize.STRING,
  comment: Sequelize.TEXT,
  date: {
    type: Sequelize.DATE,
    allowNull: false
  }
});
```

Для документации и других примеров ознакомьтесь с [документацией доклетов](#) или [документацией sequelize.com](#).

## 2. sequelize.import (путь)

Если ваши определения моделей разбиты на файл для каждого, тогда `import` - ваш друг. В файле, где вы инициализируете Sequelize, вам нужно вызвать импорт так:

```
/* Initialize Sequelize */
// Check previous code snippet for initialization

/* Define Models */
sequelize.import("./models/my_model.js"); // The path could be relative or absolute
```

Затем в файлах определения модели ваш код будет выглядеть примерно так:

```
module.exports = function(sequelize, DataTypes) {
  return sequelize.define("MyModel", {
    name: DataTypes.STRING,
    comment: DataTypes.TEXT,
    date: {
      type: DataTypes.DATE,
      allowNull: false
    }
  });
};
```

Для получения дополнительной информации о том, как использовать `import`, посмотрите пример [экспресс-демонстрации на GitHub](#).

Прочитайте [Sequelize.js онлайн](https://riptutorial.com/ru/node-js/topic/7705/sequelize-js): <https://riptutorial.com/ru/node-js/topic/7705/sequelize-js>

---

# глава 27: Автозагрузка изменений

## Examples

Автозагрузка изменений исходного кода с использованием `nodemon`

Пакет `nodemon` позволяет автоматически перезагружать вашу программу при изменении любого файла в исходном коде.

---

## Установка `nodemon` по всему миру

```
npm install -g nodemon (или npm i -g nodemon)
```

---

## Установка `nodemon` локально

Если вы не хотите устанавливать его на глобальном уровне

```
npm install --save-dev nodemon (или npm i -D nodemon)
```

---

## Использование `nodemon`

Запустите свою программу с помощью `nodemon entry.js` (или `nodemon entry`)

Это заменяет обычное использование `node entry.js` (или `node entry`).

Вы также можете добавить свой запуск `nodemon` в качестве сценария `npm`, что может быть полезно, если вы хотите предоставлять параметры и не печатать их каждый раз.

Добавьте `package.json`:

```
"scripts": {
  "start": "nodemon entry.js -devmode -something 1"
}
```

Таким образом, вы можете просто `npm start` с консоли.

## Browsersync

---

## обзор

[Browsersync](#) - это инструмент, который позволяет осуществлять просмотр в реальном времени и перезагрузку браузера. Он доступен как [пакет NPM](#) .

---

## Монтаж

Чтобы установить Browsersync, вам сначала необходимо установить [Node.js](#) и NPM. Для получения дополнительной информации см. Документацию SO по [установке и запуску Node.js](#).

После того, как ваш проект настроен, вы можете установить Browsersync с помощью следующей команды:

```
$ npm install browser-sync -D
```

Это установит `node_modules` в локальный каталог `node_modules` и сохранит его в зависимости от вашего разработчика.

Если вы хотите установить его глобально, используйте флаг `-g` вместо флага `-D` .

## Пользователи Windows

Если у вас возникли проблемы с установкой Browsersync в Windows, вам может потребоваться установить Visual Studio, чтобы вы могли получить доступ к средствам сборки для установки Browsersync. Затем вам нужно указать версию Visual Studio, которую вы используете, например:

```
$ npm install browser-sync --msvs_version=2013 -D
```

Эта команда указывает версию Visual Studio 2013 года.

---

## Основное использование

Чтобы автоматически перезагружать сайт, когда вы меняете файл JavaScript в своем проекте, используйте следующую команду:

```
$ browser-sync start --proxy "myproject.dev" --files "**/*.js"
```

Замените `myproject.dev` веб-адресом, который вы используете для доступа к вашему проекту. Browsersync выдаст альтернативный адрес, который можно использовать для доступа к вашему сайту через прокси.

# Расширенное использование

Помимо интерфейса командной строки, описанного выше, [Browsersync](#) также может использоваться с [Grunt.js](#) и [Gulp.js](#).

## Grunt.js

Использование с Grunt.js требует плагина, который можно установить так:

```
$ npm install grunt-browser-sync -D
```

Затем вы добавите эту строку в свой `gruntfile.js` :

```
grunt.loadNpmTasks('grunt-browser-sync');
```

## Gulp.js

Browsersync работает как модуль [CommonJS](#) , поэтому нет необходимости в плагине Gulp.js. Просто требуется модуль следующим образом:

```
var browserSync = require('browser-sync').create();
```

Теперь вы можете использовать [API Browsersync](#) для его настройки в соответствии с вашими потребностями.

---

## API

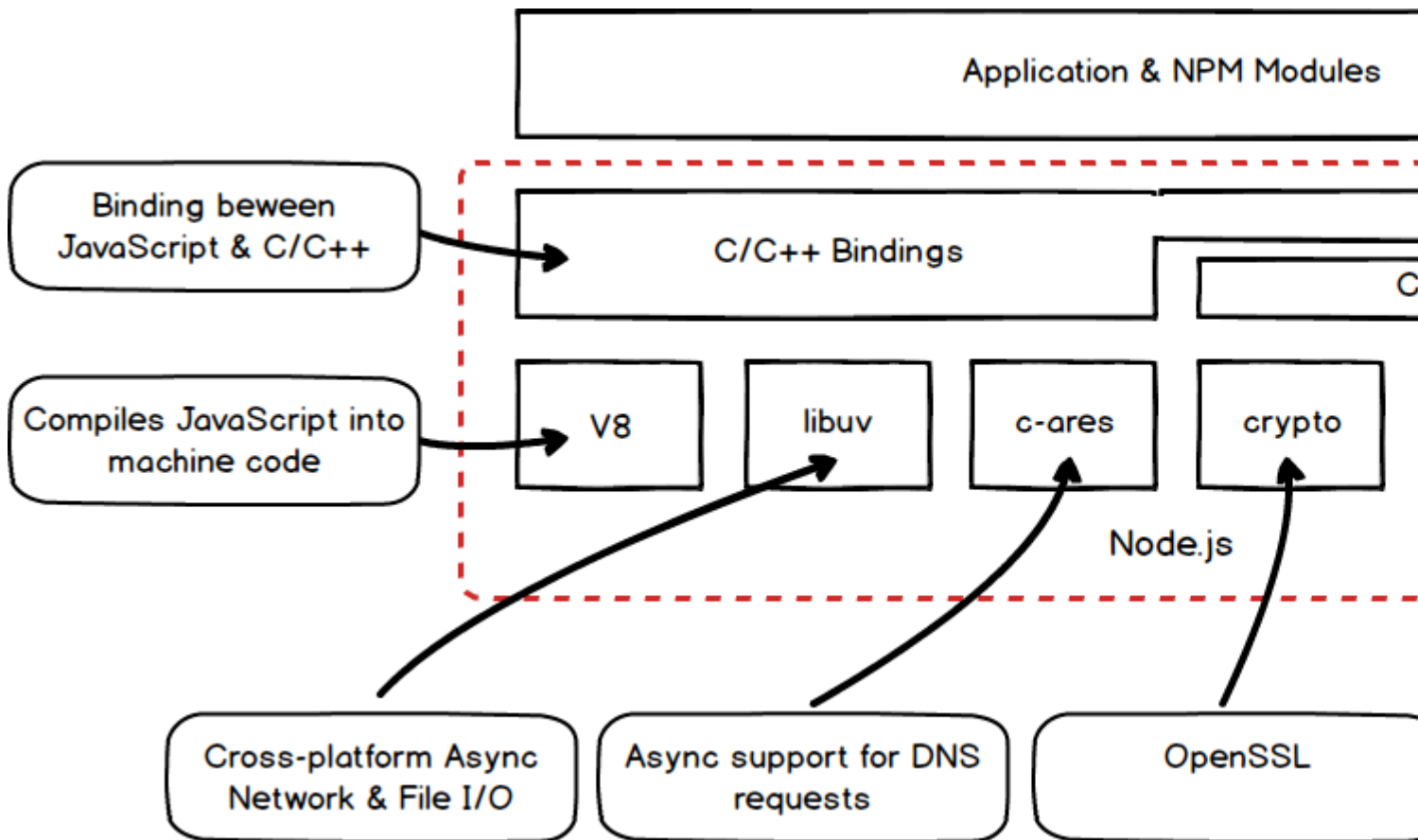
API Browsersync можно найти здесь: <https://browsersync.io/docs/api>

Прочитайте [Автозагрузка изменений онлайн](#): <https://riptutorial.com/ru/node-js/topic/1743/автозагрузка-изменений>

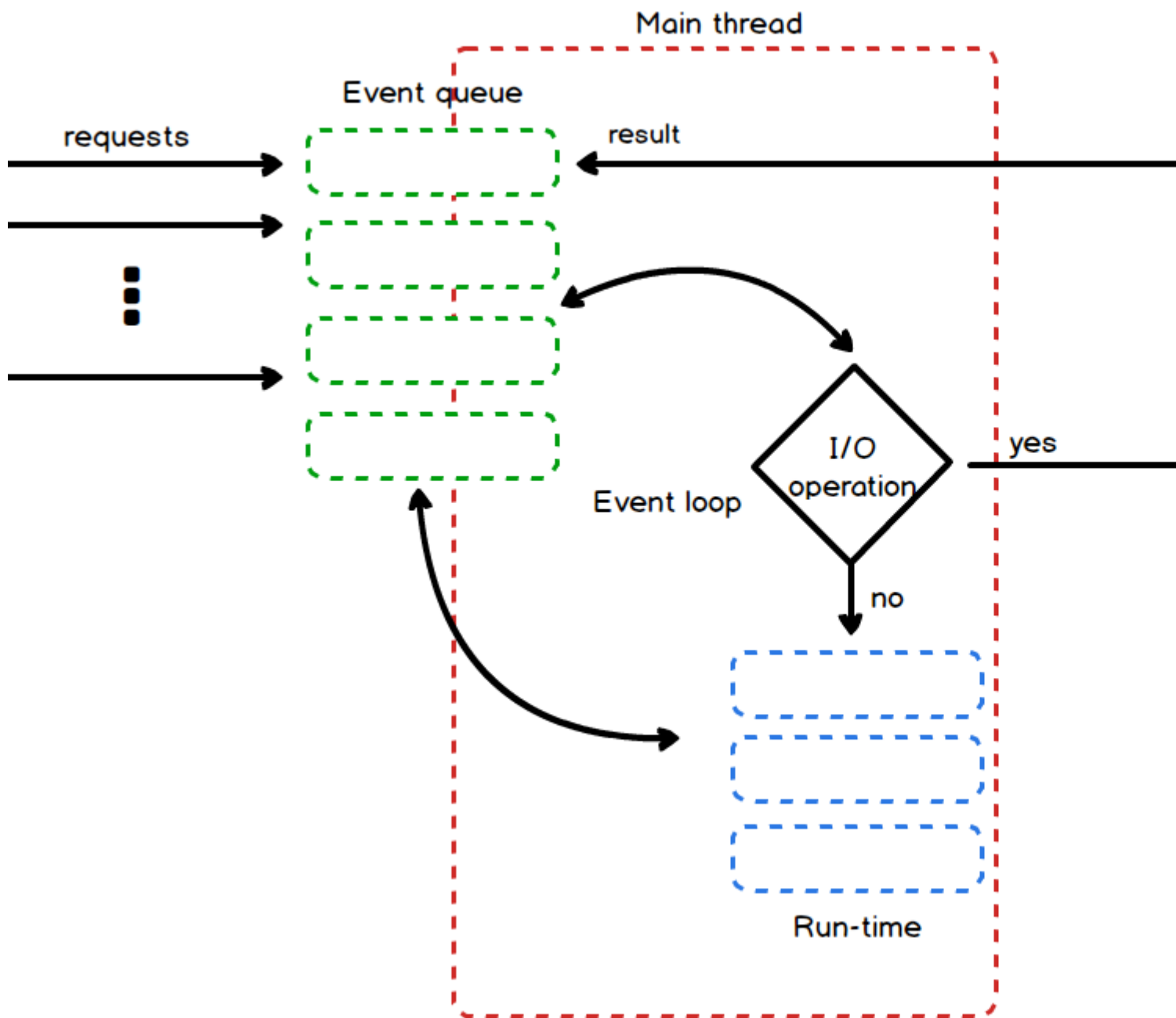
# глава 28: Архитектура и внутренние проекты Node.js

## Examples

Node.js - под капотом



Node.js - в движении



Прочитайте Архитектура и внутренние проекты Node.js онлайн:

<https://riptutorial.com/ru/node-js/topic/5892/архитектура-и-внутренние-проекты-node-js>

---

# глава 29: Асинхронное программирование

## Вступление

Узел - это язык программирования, где все может выполняться асинхронно. Ниже вы можете найти некоторые примеры и типичные вещи асинхронной работы.

## Синтаксис

- `doSomething ([args], function ([argsCB]) {/ * делать что-то, когда сделано * /});`
- `doSomething ([args], ([argsCB]) => {/ * делать что-то, когда сделано * /});`

## Examples

### Функции обратного вызова

---

## Функции обратного вызова в JavaScript

Функции обратного вызова распространены в JavaScript. Функции обратного вызова возможны в JavaScript, поскольку [функции являются первоклассными гражданами](#) .

## Синхронные обратные вызовы.

Функции обратного вызова могут быть синхронными или асинхронными. Поскольку функции асинхронного обратного вызова могут быть более сложными, это простой пример синхронной функции обратного вызова.

```
// a function that uses a callback named `cb` as a parameter
function getSyncMessage(cb) {
  cb("Hello World!");
}

console.log("Before getSyncMessage call");
// calling a function and sending in a callback function as an argument.
getSyncMessage(function(message) {
  console.log(message);
});
console.log("After getSyncMessage call");
```

Вывод для вышеуказанного кода:

```
> Before getSyncMessage call
> Hello World!
```



```
> After getSyncMessage call
```

Сначала мы рассмотрим, как выполняется вышеуказанный код. Это больше для тех, кто еще не понимает концепцию обратных вызовов, если вы уже понимаете, что можете пропустить этот абзац. Сначала код анализируется, и тогда первая интересная вещь, которая должна произойти, выполняется строка 6, которая выводит `Before getSyncMessage call` на консоль. Затем выполняется строка 8, которая вызывает функцию `getSyncMessage` отправляющую анонимную функцию в качестве аргумента для параметра `cb` в функции `getSyncMessage`. Выполнение теперь выполняется внутри функции `getSyncMessage` в строке 3, которая выполняет только что переданную функцию `cb`, этот вызов отправляет строку аргумента «Hello World» для параметра с именем `message` в переданной анонимной функции. Выполнение затем переходит к строке 9, в которой записывается `Hello World!` на консоль. Затем выполнение проходит через процесс выхода из [callstack](#) ( см. [Также](#) ), попадая в линию 10, затем линию 4, а затем обратно в строку 11.

Некоторая информация, которая должна знать о обратных вызовах в целом:

- Функция, которую вы отправляете функции в качестве обратного вызова, может быть вызвана нулевым временем, раз или несколько раз. Все зависит от реализации.
- Функция обратного вызова может быть вызвана синхронно или асинхронно и, возможно, синхронно и асинхронно.
- Подобно нормальным функциям, имена, которые вы даете параметрам для вашей функции, не важны, но порядок. Так, например, в строке 8 `message` параметра могло быть названо `statement`, `msg`, или если вы нонсенсируете что-то вроде `jellybean`. Поэтому вы должны знать, какие параметры отправлены в ваш обратный вызов, чтобы вы могли получить их в правильном порядке с именами.

## Асинхронные обратные вызовы.

Одно примечание о JavaScript - это синхронно по умолчанию, но в среде есть API-интерфейсы (браузер, Node.js и т. Д.), которые могли бы сделать его асинхронным (об этом [здесь больше](#) ).

Некоторые распространенные вещи, которые являются асинхронными в средах JavaScript, которые принимают обратные вызовы:

- События
- `SetTimeout`
- `setInterval`
- API выборки
- обещания

Также любая функция, которая использует одну из вышеперечисленных функций, может быть обернута функцией, которая выполняет обратный вызов, а обратный вызов будет

асинхронным обратным вызовом (хотя обертывание обещаний с помощью функции, которая принимает обратный вызов, скорее всего, будет рассматриваться как анти-шаблон есть более предпочтительные способы обработки обещаний).

Поэтому, учитывая эту информацию, мы можем построить асинхронную функцию, аналогичную описанной выше синхронной.

```
// a function that uses a callback named `cb` as a parameter
function getAsyncMessage(cb) {
  setTimeout(function () { cb("Hello World!") }, 1000);
}

console.log("Before getSyncMessage call");
// calling a function and sending in a callback function as an argument.
getAsyncMessage(function(message) {
  console.log(message);
});
console.log("After getSyncMessage call");
```

Что печатает на консоли следующее:

```
> Before getSyncMessage call
> After getSyncMessage call
// pauses for 1000 ms with no output
> Hello World!
```

Выполнение строки переходит к строкам 6 журналов «Перед вызовом `getSyncMessage`». Затем выполнение переходит к строке 8, вызывающей `getAsyncMessage` с обратным вызовом для параметра `cb`. Затем выполняется строка 3, которая вызывает `setTimeout` с обратным вызовом в качестве первого аргумента, а число 300 - вторым аргументом. `setTimeout` делает все, что он делает, и держится за этот обратный вызов, чтобы он мог вызывать его позже в 1000 миллисекунд, но после настройки таймаута и до того, как он приостанавливает 1000 миллисекунд, он передает выполнение обратно туда, где он остановился, поэтому он переходит в строку 4, затем строка 11, а затем пауза в течение 1 секунды и `setTimeout` затем вызывает функцию обратного вызова, которая возвращает выполнение в строку 3, где `getAsyncMessages` **ВЫЗОВ** `getAsyncMessages` **ВЫЗЫВАЕТСЯ** со значением «Hello World» для его `message` параметра, которое затем регистрируется в консоли в строке 9,

---

## Функции обратного вызова в Node.js

NodeJS имеет асинхронные обратные вызовы и обычно предоставляет два параметра для ваших функций, иногда условно называемых `err` и `data`. Пример с чтением файла.

```
const fs = require("fs");

fs.readFile("./test.txt", "utf8", function(err, data) {
```

```
if(err) {
    // handle the error
} else {
    // process the file text given with data
}
});
```

Это пример обратного вызова, который называется одним временем.

Это хорошая практика, чтобы обрабатывать ошибку так или иначе, даже если вы просто регистрируете ее или бросаете ее. Другое необязательно, если вы бросаете или возвращаете и можете удалить, чтобы уменьшить отступ, пока вы прекратите выполнение текущей функции в `if`, выполнив что-то вроде метания или возврата.

Хотя может быть общим, чтобы видеть `err`, `data` могут быть не всегда так, что ваши обратные вызовы будут использовать этот шаблон, лучше всего посмотреть документацию.

Другой пример обратного вызова происходит из экспресс-библиотеки (express 4.x):

```
// this code snippet was on http://expressjs.com/en/4x/api.html
const express = require('express');
const app = express();

// this app.get method takes a url route to watch for and a callback
// to call whenever that route is requested by a user.
app.get('/', function(req, res){
    res.send('hello world');
});

app.listen(3000);
```

В этом примере показан обратный вызов, который вызывается несколько раз. Обратный вызов предоставляется с двумя объектами в качестве параметров, названных здесь как `req` и `res` эти имена соответствуют запросу и ответу соответственно, и они предоставляют способы просмотра входящего запроса и настройки ответа, который будет отправлен пользователю.

Как вы можете видеть, существуют различные способы обратного вызова, которые можно использовать для выполнения синхронизации и асинхронного кода в JavaScript, и обратные вызовы очень повсеместны для всего JavaScript.

## Пример кода

**Вопрос:** Каков результат кода ниже и почему?

```
setTimeout(function() {
    console.log("A");
}, 1000);

setTimeout(function() {
    console.log("B");
```

```
}, 0);

getDataFromDatabase(function(err, data) {
  console.log("C");
  setTimeout(function() {
    console.log("D");
  }, 1000);
});

console.log("E");
```

**Результат:** это точно известно: EBAD . C неизвестен, когда он будет зарегистрирован.

**Объяснение:** Компилятор не остановится на `getDataFromDatabase` `setTimeout` и `getDataFromDatabase` . Таким образом, первая строка, которую он будет записывать, - E. Функции обратного вызова (*первый аргумент `setTimeout`* ) будут выполняться после установленного таймаута асинхронным способом!

**Подробнее:**

1. E не имеет `setTimeout`
2. B имеет заданный тайм-аут 0 миллисекунд
3. A имеет заданное время ожидания 1000 миллисекунд
4. D должен запросить базу данных после того, как она должна D ждать 1000 миллисекунд, чтобы она появилась после A
5. C неизвестен, поскольку он неизвестен, когда запрашиваются данные базы данных. Это может быть до или после A

## Обработка ошибок Async

# Попробуй поймать

Ошибки всегда должны выполняться. Если вы используете синхронное программирование, вы можете использовать `try catch` . Но это не работает, если вы работаете асинхронно!

Пример:

```
try {
  setTimeout(function() {
    throw new Error("I'm an uncaught error and will stop the server!");
  }, 100);
}
catch (ex) {
  console.error("This error will not be work in an asynchronous situation: " + ex);
}
```

Ошибки Async будут обрабатываться только внутри функции обратного вызова!

# Рабочие возможности

v0.8

## Обработчики событий

Первые версии Node.JS получили обработчик событий.

```
process.on("UncaughtException", function(err, data) {
  if (err) {
    // error handling
  }
});
```

v0.8

## Домены

Внутри домена ошибки выпускаются через излучатели событий. Используя это все ошибки, таймеры, методы обратного вызова неявно регистрируются только внутри домена. По ошибке, отправляйте сообщение об ошибке и не сбей приложения.

```
var domain = require("domain");
var d1 = domain.create();
var d2 = domain.create();

d1.run(function() {
  d2.add(setTimeout(function() {
    throw new Error("error on the timer of domain 2");
  }, 0));
});

d1.on("error", function(err) {
  console.log("error at domain 1: " + err);
});

d2.on("error", function(err) {
  console.log("error at domain 2: " + err);
});
```

## Обратный звонок

Адвокат обратного вызова (также пирамида гибели или эффекта бумеранга) возникает, когда вы устанавливаете слишком много функций обратного вызова внутри функции обратного вызова. Ниже приведен пример чтения файла (в ES6).

```
const fs = require('fs');
let filename = `${__dirname}/myfile.txt`;
```

```

fs.exists(filename, exists => {
  if (exists) {
    fs.stat(filename, (err, stats) => {
      if (err) {
        throw err;
      }
      if (stats.isFile()) {
        fs.readFile(filename, null, (err, data) => {
          if (err) {
            throw err;
          }
          console.log(data);
        });
      }
      else {
        throw new Error("This location contains not a file");
      }
    });
  }
  else {
    throw new Error("404: file not found");
  }
});

```

## Как избежать «Callback Hell»

Рекомендуется устанавливать не более двух функций обратного вызова. Это поможет вам сохранить читаемость кода и будет намного легче поддерживать в будущем. Если вам нужно вложить более двух обратных вызовов, попробуйте использовать [распределенные события](#).

Также существует библиотека с именем [async](#), которая помогает управлять обратными вызовами и их исполнением на прм. Это повышает читаемость кода обратного вызова и дает вам больше контроля над потоком кода обратного вызова, включая возможность запуска их параллельно или последовательно.

## Родные обещания

v6.0.0

Обещания - это инструмент для асинхронного программирования. В JavaScript обещание известно их `then` методы. Обещания состоят из двух основных состояний: «ожидающих» и «обоснованных». Как только обещание «улажено», оно не может вернуться к «ожиданию». Это означает, что обещания в основном хороши для событий, которые происходят только один раз. «Устоявшееся» государство имеет два состояния, «разрешенные» и «отклоненные». Вы можете создать новое обещание с использованием `new` ключевого слова и передать функцию в конструктор `new Promise(function (resolve, reject) {})`.

Функция, переданная в конструктор `Promise`, всегда получает первый и второй параметры, обычно называемые соответственно `resolve` и `reject`. Именование этих двух параметров является условным, но они обещают либо «разрешенное» состояние, либо «отклоненное»

состояние. Когда либо один из них называется обещанием, он переходит от «ожидającego» к «поселенному». `resolve` вызывается, когда требуемое действие, которое часто является асинхронным, выполняется, и `reject` используется, если действие было выполнено с ошибкой.

В приведенном ниже таймауте есть функция, которая возвращает Promise.

```
function timeout (ms) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      resolve("It was resolved!");
    }, ms)
  });
}

timeout(1000).then(function (dataFromPromise) {
  // logs "It was resolved!"
  console.log(dataFromPromise);
})

console.log("waiting...");
```

#### КОНСОЛЬНЫЙ ВЫХОД

```
waiting...
// << pauses for one second>>
It was resolved!
```

Когда вызывается время ожидания, функция, переданная конструктору Promise, выполняется без задержки. Затем выполняется метод `setTimeout`, и его обратный вызов активируется в следующих миллисекундах `ms`, в этом случае `ms=1000`. Поскольку обратный вызов `setTimeout` еще не запущен, функция тайм-аута возвращает управление вызывающей области. Цепочка `then` методы, которые затем сохраняются будет называться позже, когда / если обещание решено. Если бы здесь были методы `catch` они также были бы сохранены, но были бы уволены, когда / если обещание «отвергает».

Затем скрипт печатает «ожидание ...». Через секунду `setTimeout` вызывает свой обратный вызов, который вызывает функцию разрешения со строкой «Это было разрешено!». Затем эта строка передается в обратный вызов метода `then` и затем регистрируется пользователем.

В этом же смысле вы можете обернуть асинхронную функцию `setTimeout`, которая требует обратного вызова, вы можете обернуть любое сингулярное асинхронное действие с обещанием.

Подробнее о обещаниях в документации JavaScript [Promises](#).

Прочитайте Асинхронное программирование онлайн: <https://riptutorial.com/ru/node-js/topic/8813/асинхронное-программирование>

# глава 30: Асинхронный / Await

## Вступление

Async / await - это набор ключевых слов, который позволяет обрабатывать асинхронный код процедурным образом, не полагаясь на обратные вызовы ( *callback hell* ) или *bind-chaining* ( `.then().then().then()` ).

Это работает, используя ключевое слово `await` чтобы приостановить состояние функции `async`, до разрешения обещания, и используя ключевое слово `async` для объявления таких функций `async`, которые возвращают обещание.

Async / await доступен по `--harmony-async-await` `node.js` 8 по умолчанию или 7 с использованием флага `--harmony-async-await` .

## Examples

### Асинхронные функции с обработкой ошибок Try-Catch

Одной из лучших возможностей синтаксиса `async / await` является то, что стандартный стиль кодировки `try-catch` возможен, точно так же, как вы писали синхронный код.

```
const myFunc = async (req, res) => {
  try {
    const result = await somePromise();
  } catch (err) {
    // handle errors here
  }
};
```

Вот пример с Express и `prom-mysql`:

```
router.get('/flags/:id', async (req, res) => {

  try {

    const connection = await pool.createConnection();

    try {
      const sql = `SELECT f.id, f.width, f.height, f.code, f.filename
        FROM flags f
        WHERE f.id = ?
        LIMIT 1`;
      const flags = await connection.query(sql, req.params.id);
      if (flags.length === 0)
        return res.status(404).send({ message: 'flag not found' });

      return res.send({ flags[0] });
    }
  }
});
```



```
    } finally {
      pool.releaseConnection(connection);
    }

    } catch (err) {
      // handle errors here
    }
  });
```

## Сравнение обещаний с Async / Await

Функция, использующая обещания:

```
function myAsyncFunction() {
  return aFunctionThatReturnsAPromise()
    .then(result => doSomething(result))
    .catch(handleError);
}
```

Итак, вот когда Async / Await вводит действие, чтобы получить более чистую нашу функцию:

```
async function myAsyncFunction() {
  let result;

  try {
    result = await aFunctionThatReturnsAPromise();
  } catch (error) {
    handleError(error);
  }

  // doSomething is a sync function
  return doSomething(result);
}
```

Таким образом, ключевое слово `async` будет похоже на запись `return new Promise((resolve, reject) => {...})`.

И `await` похоже, чтобы получить ваш результат в `then` обратном вызове.

Здесь я оставляю довольно краткий gif, который не оставит никаких сомнений в виду, увидев его:

[GIF](#)

## Прогресс от обратных вызовов

В начале были обратные вызовы, и обратные вызовы были в порядке:

```
const getTemperature = (callback) => {
  http.get('www.temperature.com/current', (res) => {
```

```

    callback(res.data.temperature)
  })
}

const getAirPollution = (callback) => {
  http.get('www.pollution.com/current', (res) => {
    callback(res.data.pollution)
  });
}

getTemperature(function(temp) {
  getAirPollution(function(pollution) {
    console.log(`the temp is ${temp} and the pollution is ${pollution}.`)
    // The temp is 27 and the pollution is 0.5.
  })
})
})

```

Но было несколько **очень неприятных** проблем с обратными вызовами, поэтому мы все начали использовать обещания.

```

const getTemperature = () => {
  return new Promise((resolve, reject) => {
    http.get('www.temperature.com/current', (res) => {
      resolve(res.data.temperature)
    })
  })
}

const getAirPollution = () => {
  return new Promise((resolve, reject) => {
    http.get('www.pollution.com/current', (res) => {
      resolve(res.data.pollution)
    })
  })
}

getTemperature()
  .then(temp => console.log(`the temp is ${temp}`))
  .then(() => getAirPollution())
  .then(pollution => console.log(`and the pollution is ${pollution}`))
// the temp is 32
// and the pollution is 0.5

```

Это было немного лучше. Наконец, мы нашли `async / wait`. Который все еще использует обещания под капотом.

```

const temp = await getTemperature()
const pollution = await getAirPollution()

```

## Останавливает выполнение в ожидании

Если обещание ничего не возвращает, асинхронную задачу можно выполнить, используя `await`.

```

try{

```

```
await User.findByIdAndUpdate(user._id, {
  $push: {
    tokens: token
  }
}).exec()
} catch (e) {
  handleError(e)
}
```

Прочитайте Асинхронный / Await онлайн: <https://riptutorial.com/ru/node-js/topic/6729/>  
асинхронный---await

---

# глава 31: База данных (MongoDB с Mongoose)

## Examples

### Монгузское соединение

Убедитесь, что mongodb работает первым! `mongod --dbpath data/`

package.json

```
"dependencies": {
  "mongoose": "^4.5.5",
}
```

server.js (ECMA 6)

```
import mongoose from 'mongoose';

mongoose.connect('mongodb://localhost:27017/stackoverflow-example');
const db = mongoose.connection;
db.on('error', console.error.bind(console, 'DB connection error!'));
```

server.js (ECMA 5.1)

```
var mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/stackoverflow-example');
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'DB connection error!'));
```

## модель

Определите свою модель (ы):

app / models / user.js (ECMA 6)

```
import mongoose from 'mongoose';

const userSchema = new mongoose.Schema({
  name: String,
  password: String
});

const User = mongoose.model('User', userSchema);

export default User;
```

## app / model / user.js (ECMA 5.1)

```
var mongoose = require('mongoose');

var userSchema = new mongoose.Schema({
  name: String,
  password: String
});

var User = mongoose.model('User', userSchema);

module.exports = User
```

## Вставить данные

### ECMA 6:

```
const user = new User({
  name: 'Stack',
  password: 'Overflow',
});

user.save((err) => {
  if (err) throw err;

  console.log('User saved!');
});
```

### ECMA5.1:

```
var user = new User({
  name: 'Stack',
  password: 'Overflow',
});

user.save(function (err) {
  if (err) throw err;

  console.log('User saved!');
});
```

## Чтение данных

### ECMA6:

```
User.findOne({
  name: 'stack'
}, (err, user) => {
  if (err) throw err;

  if (!user) {
    console.log('No user was found');
  } else {
    console.log('User was found');
  }
});
```

```
});
```

## ECMA5.1:

```
User.findOne({
  name: 'stack'
}, function (err, user) {
  if (err) throw err;

  if (!user) {
    console.log('No user was found');
  } else {
    console.log('User was found');
  }
});
```

Прочитайте База данных (MongoDB с Mongoose) онлайн: <https://riptutorial.com/ru/node-js/topic/6411/база-данных--mongodb-с-mongoose->

# глава 32: Ввод / вывод файловой системы

## замечания

В Node.js операции с ресурсами, такие как ввод-вывод, выполняются *асинхронно*, но имеют *синхронный* аналог (например, существует `fs.readFile` а его аналог - `fs.readFileSync`). Поскольку Node является однопоточным, вы должны быть осторожны при использовании *синхронных* операций, поскольку они блокируют весь процесс.

Если процесс блокируется синхронной операцией, весь цикл выполнения (включая цикл событий) останавливается. Это означает, что другой асинхронный код, включая события и обработчики событий, не будет запускаться, и ваша программа будет продолжать ждать завершения одной операции блокировки.

Существуют соответствующие применения как для синхронных, так и для асинхронных операций, но необходимо принять меры к тому, чтобы они использовались надлежащим образом.

## Examples

### Запись в файл с использованием `writeFile` или `writeFileSync`

```
var fs = require('fs');

// Save the string "Hello world!" in a file called "hello.txt" in
// the directory "/tmp" using the default encoding (utf8).
// This operation will be completed in background and the callback
// will be called when it is either done or failed.
fs.writeFile('/tmp/hello.txt', 'Hello world!', function(err) {
  // If an error occurred, show it and return
  if(err) return console.error(err);
  // Successfully wrote to the file!
});

// Save binary data to a file called "binary.txt" in the current
// directory. Again, the operation will be completed in background.
var buffer = new Buffer([ 0x48, 0x65, 0x6c, 0x6c, 0x6f ]);
fs.writeFile('binary.txt', buffer, function(err) {
  // If an error occurred, show it and return
  if(err) return console.error(err);
  // Successfully wrote binary contents to the file!
});
```

`fs.writeFileSync` ведет себя аналогично `fs.writeFile`, но не выполняет обратный вызов, поскольку он завершает синхронно и, следовательно, блокирует основной поток. Большинство разработчиков node.js предпочитают асинхронные варианты, которые практически не задерживают выполнение программы.

*Примечание. Блокирование основного потока - это плохая практика в node.js. Синхронная функция должна использоваться только при отладке или когда другие параметры недоступны.*

```
// Write a string to another file and set the file mode to 0755
try {
  fs.writeFileSync('sync.txt', 'anni', { mode: 0o755 });
} catch(err) {
  // An error occurred
  console.error(err);
}
```

## Асинхронно читать из файлов

Используйте модуль файловой системы для всех операций с файлами:

```
const fs = require('fs');
```

## С кодировкой

В этом примере прочитайте `hello.txt` из каталога `/tmp`. Эта операция будет завершена в фоновом режиме, и обратный вызов будет выполняться при завершении или сбое:

```
fs.readFile('/tmp/hello.txt', { encoding: 'utf8' }, (err, content) => {
  // If an error occurred, output it and return
  if(err) return console.error(err);

  // No error occurred, content is a string
  console.log(content);
});
```

## Без кодирования

Прочитайте бинарный файл `binary.txt` из текущего каталога, асинхронно в фоновом режиме. Обратите внимание, что мы не устанавливаем параметр «encoding» - это предотвращает декодирование содержимого Node.js в строку:

```
fs.readFile('binary', (err, binaryContent) => {
  // If an error occurred, output it and return
  if(err) return console.error(err);

  // No error occurred, content is a Buffer, output it in
  // hexadecimal representation.
  console.log(content.toString('hex'));
});
```

## Относительные пути



Имейте в виду, что в общем случае ваш скрипт может запускаться с произвольным текущим рабочим каталогом. Чтобы обратиться к файлу относительно текущего скрипта, используйте `__dirname` или `__filename` :

```
fs.readFile(path.resolve(__dirname, 'someFile'), (err, binaryContent) => {
  //Rest of Function
})
```

## Список содержимого каталога с помощью `readdir` или `readdirSync`

```
const fs = require('fs');

// Read the contents of the directory /usr/local/bin asynchronously.
// The callback will be invoked once the operation has either completed
// or failed.
fs.readdir('/usr/local/bin', (err, files) => {
  // On error, show it and return
  if(err) return console.error(err);

  // files is an array containing the names of all entries
  // in the directory, excluding '.' (the directory itself)
  // and '..' (the parent directory).

  // Display directory entries
  console.log(files.join(' '));
});
```

Синхронный вариант доступен как `readdirSync` который блокирует основной поток и, следовательно, предотвращает одновременное выполнение асинхронного кода. Большинство разработчиков избегают синхронных функций ввода-вывода для повышения производительности.

```
let files;

try {
  files = fs.readdirSync('/var/tmp');
} catch(err) {
  // An error occurred
  console.error(err);
}
```

## Использование генератора

```
const fs = require('fs');

// Iterate through all items obtained via
// 'yield' statements
// A callback is passed to the generator function because it is required by
// the 'readdir' method
function run(gen) {
  var iter = gen((err, data) => {
    if (err) { iter.throw(err); }
  });
}
```

```
    return iter.next(data);
  });

  iter.next();
}

const dirPath = '/usr/local/bin';

// Execute the generator function
run(function* (resume) {
  // Emit the list of files in the directory from the generator
  var contents = yield fs.readdir(dirPath, resume);
  console.log(contents);
});
```

## Чтение из файла синхронно

Для любых операций с файлами вам понадобится модуль файловой системы:

```
const fs = require('fs');
```

## Чтение строки

`fs.readFileSync` ведет себя аналогично `fs.readFile`, но не выполняет обратный вызов, поскольку он завершает синхронно и поэтому блокирует основной поток. Большинство разработчиков `node.js` предпочитают асинхронные варианты, которые практически не задерживают выполнение программы.

Если указан параметр `encoding`, будет возвращена строка, иначе `Buffer` будет возвращен.

```
// Read a string from another file synchronously
let content;
try {
  content = fs.readFileSync('sync.txt', { encoding: 'utf8' });
} catch(err) {
  // An error occurred
  console.error(err);
}
```

## Удаление файла с помощью `unlink` или `unlinkSync`

Удалите файл асинхронно:

```
var fs = require('fs');

fs.unlink('/path/to/file.txt', function(err) {
  if (err) throw err;

  console.log('file deleted');
});
```

Вы также можете удалить его синхронно \*:

```
var fs = require('fs');

fs.unlinkSync('/path/to/file.txt');
console.log('file deleted');
```

\* избегайте синхронных методов, потому что они блокируют весь процесс до завершения выполнения.

## Чтение файла в буфер с использованием потоков

Хотя чтение содержимого из файла уже асинхронно с использованием `fs.readFile()`, иногда мы хотим получить данные в потоке по сравнению с простым обратным вызовом. Это позволяет нам передавать эти данные в другие местоположения или обрабатывать их, как это происходит в каждом случае одновременно.

```
const fs = require('fs');

// Store file data chunks in this array
let chunks = [];
// We can use this variable to store the final data
let fileBuffer;

// Read file into stream.Readable
let fileStream = fs.createReadStream('text.txt');

// An error occurred with the stream
fileStream.once('error', (err) => {
  // Be sure to handle this properly!
  console.error(err);
});

// File is done being read
fileStream.once('end', () => {
  // create the final data Buffer from data chunks;
  fileBuffer = Buffer.concat(chunks);

  // Of course, you can do anything else you need to here, like emit an event!
});

// Data is flushed from fileStream in chunks,
// this callback will be executed for each chunk
fileStream.on('data', (chunk) => {
  chunks.push(chunk); // push data chunk to array

  // We can perform actions on the partial data we have so far!
});
```

## Проверка разрешений файла или каталога

`fs.access()` определяет, существует ли путь и какие права доступа пользователю к файлу или каталогу на этом пути. `fs.access` не возвращает результат скорее, если он не

возвращает ошибку, путь существует, и у пользователя есть необходимые разрешения.

Режимы разрешений доступны как свойство объекта `fs`, `fs.constants`

- `fs.constants.F_OK` - Имеет разрешения на чтение / запись / выполнение (если не `fs.constants.F_OK` режим, это значение по умолчанию)
- `fs.constants.R_OK` - имеет разрешения на чтение
- `fs.constants.W_OK` - имеет разрешения на запись
- `fs.constants.X_OK` - имеет разрешения на выполнение (работает так же, как `fs.constants.F_OK` в Windows)

---

## Асинхронный

```
var fs = require('fs');
var path = '/path/to/check';

// checks execute permission
fs.access(path, fs.constants.X_OK, (err) => {
  if (err) {
    console.log("%s doesn't exist", path);
  } else {
    console.log('can execute %s', path);
  }
});

// Check if we have read/write permissions
// When specifying multiple permission modes
// each mode is separated by a pipe : `|`
fs.access(path, fs.constants.R_OK | fs.constants.W_OK, (err) => {
  if (err) {
    console.log("%s doesn't exist", path);
  } else {
    console.log('can read/write %s', path);
  }
});
```

---

## Синхронно

`fs.access` также имеет синхронную версию `fs.accessSync`. При использовании `fs.accessSync` вы должны заключить его в блок `try / catch`.

```
// Check write permission
try {
  fs.accessSync(path, fs.constants.W_OK);
  console.log('can write %s', path);
}
catch (err) {
  console.log("%s doesn't exist", path);
}
```

## Избегание условий гонки при создании или использовании существующего каталога

Из-за асинхронного характера узла, создания или использования каталога сначала:

1. проверяя его существование с помощью `fs.stat()`, тогда
2. создавая или используя его в зависимости от результатов проверки существования,

может привести к состоянию **гонки**, если папка создается между временем проверки и временем создания. Метод ниже обортывает `fs.mkdir()` и `fs.mkdirSync()` в `fs.mkdirSync()` ошибках, которые пропускают исключение, если его код `EEXIST` (уже существует). Если ошибка - это что-то еще, например `EPERM` (permission denied), пропустите или передайте ошибку, как это делают нативные функции.

### Асинхронная версия с `fs.mkdir()`

```
var fs = require('fs');

function mkdir (dirPath, callback) {
  fs.mkdir(dirPath, (err) => {
    callback(err && err.code !== 'EEXIST' ? err : null);
  });
}

mkdir('./existingDir', (err) => {

  if (err)
    return console.error(err.code);

  // Do something with `./existingDir` here

});
```

### Синхронная версия с `fs.mkdirSync()`

```
function mkdirSync (dirPath) {
  try {
    fs.mkdirSync(dirPath);
  } catch(e) {
    if ( e.code !== 'EEXIST' ) throw e;
  }
}

mkdirSync('./existing-dir');
// Do something with `./existing-dir` now
```

## Проверка наличия файла или каталога

## Асинхронный

```
var fs = require('fs');

fs.stat('path/to/file', function(err) {
  if (!err) {
    console.log('file or directory exists');
  }
  else if (err.code === 'ENOENT') {
    console.log('file or directory does not exist');
  }
});
```

## Синхронно

здесь мы должны обернуть вызов функции в блок `try/catch` для обработки ошибки.

```
var fs = require('fs');

try {
  fs.statSync('path/to/file');
  console.log('file or directory exists');
}
catch (err) {
  if (err.code === 'ENOENT') {
    console.log('file or directory does not exist');
  }
}
```

## Клонирование файла с использованием потоков

Эта программа иллюстрирует, как можно скопировать файл с помощью считываемых и записываемых потоков с помощью функций `createReadStream()` и `createWriteStream()` предоставляемых модулем файловой системы.

```
//Require the file System module
var fs = require('fs');

/*
  Create readable stream to file in current directory (__dirname) named 'node.txt'
  Use utf8 encoding
  Read the data in 16-kilobyte chunks
*/
var readable = fs.createReadStream(__dirname + '/node.txt', { encoding: 'utf8', highWaterMark:
16 * 1024 });

// create writable stream
var writable = fs.createWriteStream(__dirname + '/nodeCopy.txt');

// Write each chunk of data to the writable stream
readable.on('data', function(chunk) {
  writable.write(chunk);
});
```

## Копирование файлов по потокам трубопроводов

Эта программа копирует файл с помощью считываемого и записываемого потока с помощью функции `pipe()` предоставляемой классом потока

```
// require the file system module
var fs = require('fs');

/*
  Create readable stream to file in current directory named 'node.txt'
  Use utf8 encoding
  Read the data in 16-kilobyte chunks
*/
var readable = fs.createReadStream(__dirname + '/node.txt', { encoding: 'utf8', highWaterMark:
16 * 1024 });

// create writable stream
var writable = fs.createWriteStream(__dirname + '/nodePipe.txt');

// use pipe to copy readable to writable
readable.pipe(writable);
```

## Изменение содержимого текстового файла

Пример. Он заменит слово `email` на `name` в текстовом файле `index.txt` с простой `RegExp` `replace(/email/gim, 'name')`

```
var fs = require('fs');

fs.readFile('index.txt', 'utf-8', function(err, data) {
  if (err) throw err;

  var newValue = data.replace(/email/gim, 'name');

  fs.writeFile('index.txt', newValue, 'utf-8', function(err, data) {
    if (err) throw err;
    console.log('Done!');
  })
})
```

## Определение количества строк текстового файла

### app.js

```
const readline = require('readline');
const fs = require('fs');

var file = 'path.to.file';
var linesCount = 0;
var rl = readline.createInterface({
  input: fs.createReadStream(file),
  output: process.stdout,
  terminal: false
});
rl.on('line', function (line) {
```

```
    linesCount++; // on each linebreak, add +1 to 'linesCount'
  });
  rl.on('close', function () {
    console.log(linesCount); // print the result when the 'close' event is called
  });
```

Использование:

узловое приложение

## Чтение файла по строкам

### app.js

```
const readline = require('readline');
const fs = require('fs');

var file = 'path.to.file';
var rl = readline.createInterface({
  input: fs.createReadStream(file),
  output: process.stdout,
  terminal: false
});

rl.on('line', function (line) {
  console.log(line) // print the content of the line on each linebreak
});
```

Использование:

узловое приложение

Прочитайте Ввод / вывод файловой системы онлайн: <https://riptutorial.com/ru/node-js/topic/489/ввод---вывод-файловой-системы>



# глава 33: Веб-приложения с Express

## Вступление

Express - это минимальная и гибкая инфраструктура веб-приложений Node.js, обеспечивающая надежный набор функций для создания веб-приложений.

Официальный сайт Express - [expressjs.com](https://expressjs.com) . Источник можно найти [на GitHub](#) .

## Синтаксис

- `app.get` (путь [, промежуточное ПО], обратный вызов [, обратный вызов ...])
- `app.put` (путь [, промежуточное ПО], обратный вызов [, обратный вызов ...])
- `app.post` (путь [, промежуточное ПО], обратный вызов [, обратный вызов ...])
- `app['delete']` (путь [, промежуточное ПО], обратный вызов [, обратный вызов ...])
- `app.use` (путь [, промежуточное ПО], обратный вызов [, обратный вызов ...])
- `app.use` (обратный вызов)

## параметры

параметр	подробности
<code>path</code>	Определяет участок пути или URL-адрес, который обрабатывает данный обратный вызов.
<code>middleware</code>	Одна или несколько функций, которые будут вызываться перед обратным вызовом. По сути, это цепочка нескольких функций <code>callback</code> . Полезно для более конкретной обработки, например авторизации или обработки ошибок.
<code>callback</code>	Функция, которая будет использоваться для обработки запросов к указанному <code>path</code> . Он будет называться как <code>callback(request, response, next)</code> , где <code>request</code> , <code>response</code> и <code>next</code> описаны ниже.
<code>request</code> <i>обратного вызова</i>	Объект, инкапсулирующий данные о HTTP-запросе, вызываемые для обратного вызова.
<code>response</code>	Объект, который используется для указания того, как сервер должен отвечать на запрос.
<code>next</code>	Обратный вызов, который передает управление на следующий

параметр	подробности
	соответствующий маршрут. Он принимает необязательный объект ошибки.

## Examples

### Начиная

Сначала вам нужно создать каталог, получить доступ к нему в своей оболочке и установить Express с помощью [npm](#), выполнив `npm install express --save`

Создайте файл и назовите его `app.js` и добавьте следующий код, который создает новый Express-сервер и добавляет к нему одну конечную точку (`/ping`) с `app.get` метода `app.get` :

```
const express = require('express');

const app = express();

app.get('/ping', (request, response) => {
  response.send('pong');
});

app.listen(8080, 'localhost');
```

Для запуска скрипта используйте следующую команду в своей оболочке:

```
> node app.js
```

Приложение будет принимать соединения на локальном порту 8080. Если аргумент `hostname` в `app.listen` опущен, сервер будет принимать соединения на IP-адресе компьютера, а также на `localhost`. Если значение порта равно 0, операционная система назначит доступный порт.

Как только ваш скрипт запущен, вы можете протестировать его в оболочке, чтобы подтвердить, что вы получаете ожидаемый ответ «понг» с сервера:

```
> curl http://localhost:8080/ping
pong
```

Вы также можете открыть веб-браузер, перейти к URL-адресу <http://localhost:8080/ping> для просмотра вывода

### Базовая маршрутизация

Сначала создайте экспресс-приложение:

```
const express = require('express');
const app = express();
```

Затем вы можете определить маршруты следующим образом:

```
app.get('/someUri', function (req, res, next) {})
```

Эта структура работает для всех HTTP-методов и ожидает путь как первый аргумент и обработчик для этого пути, который получает объекты запроса и ответа. Итак, для базовых HTTP-методов это маршруты

```
// GET www.domain.com/myPath
app.get('/myPath', function (req, res, next) {})

// POST www.domain.com/myPath
app.post('/myPath', function (req, res, next) {})

// PUT www.domain.com/myPath
app.put('/myPath', function (req, res, next) {})

// DELETE www.domain.com/myPath
app.delete('/myPath', function (req, res, next) {})
```

Вы можете проверить полный список поддерживаемых глаголов [здесь](#) . Если вы хотите определить одно и то же поведение для маршрута и всех методов HTTP, вы можете использовать:

```
app.all('/myPath', function (req, res, next) {})
```

или же

```
app.use('/myPath', function (req, res, next) {})
```

или же

```
app.use('*', function (req, res, next) {})

// * wildcard will route for all paths
```

Вы можете связать определения маршрута по одному пути

```
app.route('/myPath')
  .get(function (req, res, next) {})
  .post(function (req, res, next) {})
  .put(function (req, res, next) {})
```

Вы также можете добавлять функции к любому HTTP-методу. Они будут выполняться перед окончательным обратным вызовом и принимать параметры (req, res, next) в качестве аргументов.

```
// GET www.domain.com/myPath
app.get('/myPath', myFunction, function (req, res, next) {})
```

Ваши заключительные обратные вызовы могут быть сохранены во внешнем файле, чтобы избежать слишком большого количества кода в одном файле:

```
// other.js
exports.doSomething = function(req, res, next) { /* do some stuff */};
```

А затем в файле, содержащем ваши маршруты:

```
const other = require('./other.js');
app.get('/someUri', myFunction, other.doSomething);
```

Это сделает ваш код намного чище.

## Получение информации из запроса

Чтобы получить информацию с запрашивающего URL-адреса (обратите внимание, что `req` является объектом запроса в функции обработчика маршрутов). Рассмотрим это определение маршрута `/settings/:user_id` и этот конкретный пример `/settings/32135?field=name`

```
// get the full path
req.originalUrl // => /settings/32135?field=name

// get the user_id param
req.params.user_id // => 32135

// get the query value of the field
req.query.field // => 'name'
```

Вы также можете получить заголовки запроса, например

```
req.get('Content-Type')
// "text/plain"
```

Чтобы упростить получение другой информации, вы можете использовать `middlewares`. Например, чтобы получить информацию о теле запроса, вы можете использовать промежуточное программное обеспечение `body-parser`, которое преобразует необработанный модуль запроса в удобный формат.

```
var app = require('express')();
var bodyParser = require('body-parser');

app.use(bodyParser.json()); // for parsing application/json
app.use(bodyParser.urlencoded({ extended: true })); // for parsing application/x-www-form-urlencoded
```

Теперь предположим, что запрос такой

```
PUT /settings/32135
{
  "name": "Peter"
}
```

Вы можете получить доступ к опубликованному имени, как это

```
req.body.name
// "Peter"
```

Аналогичным образом вы можете получить доступ к файлам cookie из запроса, вам также понадобится промежуточное программное обеспечение, такое как [cookie-parser](#)

```
req.cookies.name
```

## Модульное экспресс-приложение

Чтобы сделать экспресс-веб-приложение модульным заводом-изготовителем маршрутизаторов:

Модуль:

```
// greet.js
const express = require('express');

module.exports = function(options = {}) { // Router factory
  const router = express.Router();

  router.get('/greet', (req, res, next) => {
    res.end(options.greeting);
  });

  return router;
};
```

Заявка:

```
// app.js
const express = require('express');
const greetMiddleware = require('./greet.js');

express()
  .use('/api/v1/', greetMiddleware({ greeting: 'Hello world' }))
  .listen(8080);
```

Это сделает ваше приложение модульным, настраиваемым и ваш код повторно использовать.

При доступе к `http://<hostname>:8080/api/v1/greet` вывод будет `Hello world`

## Более сложный пример

Пример с услугами, которые показывают преимущества фабрики промежуточного ПО.

Модуль:

```
// greet.js
const express = require('express');

module.exports = function(options = {}) { // Router factory
  const router = express.Router();
  // Get controller
  const {service} = options;

  router.get('/greet', (req, res, next) => {
    res.end(
      service.createGreeting(req.query.name || 'Stranger')
    );
  });

  return router;
};
```

Заявка:

```
// app.js
const express = require('express');
const greetMiddleware = require('./greet.js');

class GreetingService {
  constructor(greeting = 'Hello') {
    this.greeting = greeting;
  }

  createGreeting(name) {
    return `${this.greeting}, ${name}!`;
  }
}

express()
  .use('/api/v1/service1', greetMiddleware({
    service: new GreetingService('Hello'),
  }))
  .use('/api/v1/service2', greetMiddleware({
    service: new GreetingService('Hi'),
  }))
  .listen(8080);
```

При доступе к `http://<hostname>:8080/api/v1/service1/greet?name=World` вывод будет `Hello, World` и доступ к `http://<hostname>:8080/api/v1/service2/greet?name=World` выход будет `Hi, World`.

## Использование механизма шаблонов

# Использование механизма шаблонов

Следующий код установит Jade в качестве механизма шаблонов. (Примечание: Джейд был переименован в pug состоянию на декабрь 2015 года.)

```
const express = require('express'); //Imports the express module
const app = express(); //Creates an instance of the express module

const PORT = 3000; //Randomly chosen port

app.set('view engine','jade'); //Sets jade as the View Engine / Template Engine
app.set('views','src/views'); //Sets the directory where all the views (.jade files) are
stored.

//Creates a Root Route
app.get('/',function(req, res){
  res.render('index'); //renders the index.jade file into html and returns as a response.
The render function optionally takes the data to pass to the view.
});

//Starts the Express server with a callback
app.listen(PORT, function(err) {
  if (!err) {
    console.log('Server is running at port', PORT);
  } else {
    console.log(JSON.stringify(err));
  }
});
```

Аналогичным образом могут использоваться и другие шаблонные двигатели, такие как Handlebars ( hbs ) или ejs . Не забудьте также npm install механизм шаблонов. Для Handlebars мы используем пакет hbs , для Jade у нас есть пакет jade и для EJS у нас есть пакет ejs .

## Пример шаблона EJS

С помощью EJS (например, других экспресс-шаблонов) вы можете запускать код сервера и получать доступ к своим переменным сервера из HTML.

В EJS это делается с помощью « <% » в качестве начального тега и « %> » в качестве конечного тега, переменные, переданные в качестве параметров рендеринга, могут быть доступны с помощью <%=var\_name%>

Например, если у вас есть массив поставок в коде сервера вы можете

```
<h1><%= title %></h1>
<ul>
<% for(var i=0; i<supplies.length; i++) { %>
  <li>
    <a href='supplies/<%= supplies[i] %>'>
      <%= supplies[i] %>
    </a>
  </li>
```

```
<% } %>
```

Как вы можете видеть в этом примере каждый раз, когда вы переключаетесь между кодом на стороне сервера и HTML, вам нужно закрыть текущий тег EJS и открыть новый позже, здесь мы хотели создать `li` внутри команды `for` поэтому нам нужно было закрыть наш тег EJS в конце `for` и создайте новый тег только для фигурных скобок

другой пример

если мы хотим, чтобы входная версия по умолчанию была переменной с серверной стороны, мы используем `<%=`

например:

```
Message:<br>
<input type="text" value="<%= message %>" name="message" required>
```

Здесь переменная сообщения, переданная со стороны вашего сервера, будет значением по умолчанию для вашего ввода, обратите внимание, что если вы не передали переменную сообщения со своей серверной части, EJS выдаст исключение. Вы можете передавать параметры с помощью `res.render('index', {message: message});` (для файла `ejs`, называемого `index.ejs`).

В тегах EJS вы также можете использовать `if`, `while` или любую другую команду javascript, которую вы хотите.

## API JSON с ExpressJS

```
var express = require('express');
var cors = require('cors'); // Use cors module for enable Cross-origin resource sharing

var app = express();
app.use(cors()); // for all routes

var port = process.env.PORT || 8080;

app.get('/', function(req, res) {
  var info = {
    'string_value': 'StackOverflow',
    'number_value': 8476
  }
  res.json(info);

  // or
  /* res.send(JSON.stringify({
    string_value: 'StackOverflow',
    number_value: 8476
  })); */

  //you can add a status code to the json response
  /* res.status(200).json(info) */
})

app.listen(port, function() {
```



```
console.log('Node.js listening on port ' + port)
})
```

В `http://localhost:8080/` **выходной объект**

```
{
  string_value: "StackOverflow",
  number_value: 8476
}
```

## Обслуживание статических файлов

При создании веб-сервера с помощью Express часто требуется использовать комбинацию динамического содержимого и статических файлов.

Например, у вас могут быть `index.html` и `script.js`, которые являются статическими файлами, хранящимися в файловой системе.

Обычно для использования статических файлов используется папка с именем «public». В этом случае структура папок может выглядеть так:

```
project root
├─ server.js
├─ package.json
└─ public
   └─ index.html
   └─ script.js
```

Вот как настроить Express для обслуживания статических файлов:

```
const express = require('express');
const app = express();

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

Примечание: как только папка настроена, `index.html`, `script.js` и все файлы в «общедоступной» папке будут доступны на корневом пути (вы не должны указывать `/public/` в URL-адресе). Это связано с тем, что экспресс ищет файлы, относящиеся к установленной статической папке. Вы можете указать *префикс виртуального пути*, как показано ниже:

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

сделает ресурсы доступными в `/static/` префикс.

## Несколько папок

Одновременно можно определить несколько папок:

```
app.use(express.static('public'));
app.use(express.static('images'));
app.use(express.static('files'));
```

При обслуживании ресурсов Express проверит папку в порядке определения. В случае файлов с тем же именем будет обслуживаться тот, который находится в первой соответствующей папке.

## Именованные маршруты в стиле Django

Одна из больших проблем заключается в том, что ценные именованные маршруты не поддерживаются Express из коробки. Решение заключается в установке поддерживаемого стороннего пакета, например [express-reverse](#) :

```
npm install express-reverse
```

Подключите его в свой проект:

```
var app = require('express')();
require('express-reverse')(app);
```

Затем используйте его так:

```
app.get('test', '/hello', function(req, res) {
  res.end('hello');
});
```

Недостатком этого подхода является то , что вы не можете использовать `route` модуль Express , как показано в [Advanced использования маршрутизатора](#) . Обходным путем является передача вашего `app` в качестве параметра для вашего маршрутизатора:

```
require('./middlewares/routing')(app);
```

И используйте его как:

```
module.exports = (app) => {
  app.get('test', '/hello', function(req, res) {
    res.end('hello');
  });
};
```

Теперь вы можете понять это, как определить функции, чтобы объединить его с указанными пользовательскими пространствами имен и указать на соответствующие контроллеры.

## Обработка ошибок

## Основная обработка ошибок

По умолчанию Express будет искать представление «error» в каталоге `/views` для рендеринга. Просто создайте представление «ошибка» и поместите его в каталог представлений для обработки ошибок. Ошибки записываются с сообщением об ошибке, статусом и трассировкой стека, например:

*просмотров / error.pug*

```
html
  body
    h1= message
    h2= error.status
    p= error.stack
```

## Расширенная обработка ошибок

Определите функции промежуточного программного обеспечения обработки ошибок в самом конце стека функций промежуточного программного обеспечения. У них есть четыре аргумента вместо трех (`err`, `req`, `res`, `next`):

*app.js*

```
// catch 404 and forward to error handler
app.use(function(req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;

  //pass error to the next matching route.
  next(err);
});

// handle error, print stacktrace
app.use(function(err, req, res, next) {
  res.status(err.status || 500);

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

Вы можете определить несколько функций промежуточного программного обеспечения обработки ошибок, как и обычные функции промежуточного программного обеспечения.

## Использование промежуточного программного обеспечения и следующего обратного вызова

Express передает `next` обратный вызов каждому обработчику маршрута и функции промежуточного программного обеспечения, которые могут использоваться для разрыва логики для отдельных маршрутов через несколько обработчиков. Вызов `next()` без

аргументов говорит `express` для продолжения следующего соответствующего промежуточного программного обеспечения или обработчика маршрута. Вызов `next(err)` с ошибкой вызовет любое промежуточное программное обеспечение обработчика ошибок. Вызов `next('route')` будет обходить любое последующее промежуточное ПО на текущем маршруте и перейти к следующему согласованному маршруту. Это позволяет разделить логику домена на повторно используемые компоненты, которые являются автономными, более простыми для тестирования и проще поддерживать и изменять.

## Несколько совпадающих маршрутов

Запросы в `/api/foo` или `/api/bar` будут запускать начальный обработчик для поиска члена, а затем передать управление фактическому обработчику для каждого маршрута.

```
app.get('/api', function(req, res, next) {
  // Both /api/foo and /api/bar will run this
  lookupMember(function(err, member) {
    if (err) return next(err);
    req.member = member;
    next();
  });
});

app.get('/api/foo', function(req, res, next) {
  // Only /api/foo will run this
  doSomethingWithMember(req.member);
});

app.get('/api/bar', function(req, res, next) {
  // Only /api/bar will run this
  doSomethingDifferentWithMember(req.member);
});
```

## Обработчик ошибок

Обработчики ошибок являются промежуточным программным обеспечением с `function(err, req, res, next)` подписи `function(err, req, res, next)`. Их можно настроить на один маршрут (например, `app.get('/foo', function(err, req, res, next) {`), но, как правило, достаточно одного обработчика ошибок, который отображает страницу с ошибкой.

```
app.get('/foo', function(req, res, next) {
  doSomethingAsync(function(err, data) {
    if (err) return next(err);
    renderPage(data);
  });
});

// In the case that doSomethingAsync return an error, this special
// error handler middleware will be called with the error as the
// first parameter.
app.use(function(err, req, res, next) {
  renderErrorPage(err);
});
```

## Промежуточное

Каждая из вышеперечисленных функций фактически является функцией промежуточного программного обеспечения, которая запускается всякий раз, когда запрос соответствует определенному маршруту, но любое количество промежуточных функций может быть определено на одном маршруте. Это позволяет определить промежуточное программное обеспечение в отдельных файлах и общую логику для повторного использования на нескольких маршрутах.

```
app.get('/bananas', function(req, res, next) {
  getMember(function(err, member) {
    if (err) return next(err);
    // If there's no member, don't try to look
    // up data. Just go render the page now.
    if (!member) return next('route');
    // Otherwise, call the next middleware and fetch
    // the member's data.
    req.member = member;
    next();
  });
}, function(req, res, next) {
  getMemberData(req.member, function(err, data) {
    if (err) return next(err);
    // If this member has no data, don't bother
    // parsing it. Just go render the page now.
    if (!data) return next('route');
    // Otherwise, call the next middleware and parse
    // the member's data. THEN render the page.
    req.member.data = data;
    next();
  });
}, function(req, res, next) {
  req.member.parsedData = parseMemberData(req.member.data);
  next();
});

app.get('/bananas', function(req, res, next) {
  renderBananas(req.member);
});
```

В этом примере каждая промежуточная функция будет либо в собственном файле, либо в переменной в другом месте файла, чтобы его можно было повторно использовать в других маршрутах.

## Обработка ошибок

Основные документы можно найти [здесь](#)

```
app.get('/path/:id(\\d+)', function (req, res, next) { // please note: "next" is passed
  if (req.params.id == 0) // validate param
    return next(new Error('Id is 0')); // go to first Error handler, see below

  // Catch error on sync operation
  var data;
```

```

try {
  data = JSON.parse('/file.json');
} catch (err) {
  return next(err);
}

// If some critical error then stop application
if (!data)
  throw new Error('Smth wrong');

// If you need send extra info to Error handler
// then send custom error (see Appendix B)
if (smth)
  next(new MyError('smth wrong', arg1, arg2))

// Finish request by res.render or res.end
res.status(200).end('OK');
});

// Be sure: order of app.use have matter
// Error handler
app.use(function(err, req, res, next) {
  if (smth-check, e.g. req.url != 'POST')
    return next(err); // go-to Error handler 2.

  console.log(req.url, err.message);

  if (req.xhr) // if req via ajax then send json else render error-page
    res.json(err);
  else
    res.render('error.html', {error: err.message});
});

// Error handler 2
app.use(function(err, req, res, next) {
  // do smth here e.g. check that error is MyError
  if (err instanceof MyError) {
    console.log(err.message, err.arg1, err.arg2);
  }
  ...
  res.end();
});

```

## Приложение

```

// "In Express, 404 responses are not the result of an error,
// so the error-handler middleware will not capture them."
// You can change it.
app.use(function(req, res, next) {
  next(new Error(404));
});

```

## Приложение B

```

// How to define custom error
var util = require('util');
...
function MyError(message, arg1, arg2) {

```

```
this.message = message;
this.arg1 = arg1;
this.arg2 = arg2;
Error.captureStackTrace(this, MyError);
}
util.inherits(MyError, Error);
MyError.prototype.name = 'MyError';
```

## Крюк: как выполнить код перед любым req и после любого res

`app.use()` и промежуточное ПО можно использовать для «раньше», а комбинацию событий **закрытия** и **завершения** можно использовать для «после».

```
app.use(function (req, res, next) {
  function afterResponse() {
    res.removeListener('finish', afterResponse);
    res.removeListener('close', afterResponse);

    // actions after response
  }
  res.on('finish', afterResponse);
  res.on('close', afterResponse);

  // action before request
  // eventually calling `next()`
  next();
});
...
app.use(app.router);
```

Примером этого является промежуточное программное обеспечение **логгера**, которое будет добавляться к журналу после ответа по умолчанию.

Просто убедитесь, что это «промежуточное программное обеспечение» используется до приложения `app.router` поскольку порядок имеет значение.

---

Оригинальный пост [здесь](#)

## Обработка запросов POST

Так же, как вы обрабатываете запросы на получение в Express с помощью метода `app.get`, вы можете использовать метод `app.post` для обработки почтовых запросов.

Но прежде чем вы сможете обрабатывать запросы POST, вам нужно будет использовать промежуточное программное обеспечение `body-parser`. Он просто анализирует тело `POST`, `PUT`, `DELETE` и других запросов.

`Body-Parser` ПО `Body-Parser` анализирует тело запроса и превращает его в объект, доступный **В** `req.body`

```

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

const express = require('express');

const app = express();

// Parses the body for POST, PUT, DELETE, etc.
app.use(bodyParser.json());

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

app.post('/post-data-here', function(req, res, next){

    console.log(req.body); // req.body contains the parsed body of the request.

});

app.listen(8080, 'localhost');

```

## Настройка файлов cookie с помощью cookie-parser

Ниже приведен пример установки и чтения файлов cookie с использованием модуля [cookie-parser](#) :

```

var express = require('express');
var cookieParser = require('cookie-parser'); // module for parsing cookies
var app = express();
app.use(cookieParser());

app.get('/setcookie', function(req, res){
    // setting cookies
    res.cookie('username', 'john doe', { maxAge: 900000, httpOnly: true });
    return res.send('Cookie has been set');
});

app.get('/getcookie', function(req, res) {
    var username = req.cookies['username'];
    if (username) {
        return res.send(username);
    }

    return res.send('No cookie found');
});

app.listen(3000);

```

## Пользовательское промежуточное ПО в Express

В Express вы можете определить middlewares, которые могут использоваться для проверки запросов или установки некоторых заголовков в ответ.

```
app.use(function(req, res, next){ }); // signature
```

### пример



Следующий код добавляет `user` к объекту запроса и передает управление следующему соответствующему маршруту.

```
var express = require('express');
var app = express();

//each request will pass through it
app.use(function(req, res, next){
  req.user = 'testuser';
  next();    // it will pass the control to next matching route
});

app.get('/', function(req, res){
  var user = req.user;
  console.log(user); // testuser
  return res.send(user);
});

app.listen(3000);
```

## Обработка ошибок в Express

В Express вы можете определить унифицированный обработчик ошибок для обработки ошибок, возникающих в приложении. Определите тогда обработчик в конце всех маршрутов и логического кода.

### пример

```
var express = require('express');
var app = express();

//GET /names/john
app.get('/names/:name', function(req, res, next){
  if (req.params.name == 'john'){
    return res.send('Valid Name');
  } else{
    next(new Error('Not valid name'));    //pass to error handler
  }
});

//error handler
app.use(function(err, req, res, next){
  console.log(err.stack);    // e.g., Not valid name
  return res.status(500).send('Internal Server Occured');
});

app.listen(3000);
```

## Добавление промежуточного ПО

Функции промежуточного программного обеспечения - это функции, которые имеют доступ к объекту запроса (`req`), объекту ответа (`res`) и следующей функции промежуточного программного обеспечения в цикле запроса-ответа приложения.

Функции промежуточного программного обеспечения могут выполнять любой код, вносить изменения в объекты `res` и `req`, завершать цикл ответа и вызывать следующее промежуточное программное обеспечение.

Очень распространенным примером промежуточного программного обеспечения является модуль `cors`. Чтобы добавить поддержку CORS, просто установите ее, установите ее и поместите эту строку:

```
app.use(cors());
```

перед любыми маршрутизаторами или функциями маршрутизации.

## Привет, мир

Здесь мы создаем базовый сервер приветствия с помощью Express. Маршруты:

- '/'
- «/ Вики»

А для отдыха даст «404», т.е. страница не найдена.

```
'use strict';

const port = process.env.PORT || 3000;

var app = require('express')();
app.listen(port);

app.get('/', (req, res) => res.send('HelloWorld!'));
app.get('/wiki', (req, res) => res.send('This is wiki page.'));
app.use((req, res) => res.send('404-PageNotFound'));
```

**Примечание.** Мы поместили маршрут 404 в качестве последнего маршрута в качестве экспресс-стеков в порядке и обрабатываем их для каждого запроса последовательно.

Прочитайте [Веб-приложения с Express онлайн: https://riptutorial.com/ru/node-js/topic/483/веб-приложения-с-express](https://riptutorial.com/ru/node-js/topic/483/веб-приложения-с-express)

---

# глава 34: Взаимодействие с консолью

## Синтаксис

- `console.log` ([данные] [, ...])
- `console.error` ([данные] [, ...])
- `console.time` (ярлык)
- `console.timeEnd` (ярлык)

## Examples

### логирование

---

## Консольный модуль

Подобно среде браузера JavaScript `node.js` предоставляет **консольный** модуль, который обеспечивает простые возможности ведения журнала и отладки.

Наиболее важные методы, предоставляемые консольным модулем, - `console.log`, `console.error` и `console.time`. Но есть несколько других, таких как `console.info`.

## `console.log`

Параметры будут напечатаны на стандартный вывод (`stdout`) с новой строкой.

```
console.log('Hello World');
```

```
> console.log('Hello World')
Hello World
```

## `console.error`

Параметры будут напечатаны на стандартную ошибку (`stderr`) с новой строкой.

```
console.error('Oh, sorry, there is an error.');
```

```
> console.error("Oh, sorry, error");
Oh, sorry, error
```

## `console.time`, `console.timeEnd`

`console.time` запускает таймер с уникальной вкладкой, которая может использоваться для вычисления продолжительности операции. Когда вы вызываете `console.timeEnd` с тем же

ярлыком, таймер останавливается, и он печатает прошедшее время в миллисекундах на `stdout`.

```
> console.time("label");  
undefined  
> console.timeEnd("label");  
label: 9297.320ms
```

---

## Технологический модуль

Можно использовать модуль **процесса** для записи **непосредственно** в стандартный вывод консоли. Поэтому существует метод `process.stdout.write`. В отличие от `console.log` этот метод не добавляет новую строку перед вашим выходом.

Итак, в следующем примере метод вызывается два раза, но новая строка не добавляется между их выходами.

```
> process.stdout.write("123");process.stdout.write("456");  
123456true
```

---

## форматирование

Можно использовать **терминальные (управляющие) коды** для выдачи определенных команд, например, переключения цветов или позиционирования курсора.

```
> console.log("\033[31mThis will be red");  
This will be red
```

## генеральный

эффект	Код
Сброс	\033[0m
HiColor	\033[1m
подчеркивание	\033[4m
обратный	\033[7m

## Цвет шрифта

эффект	Код
черный	\033[30m

эффект	Код
красный	\033[31m
зеленый	\033[32m
желтый	\033[33m
синий	\033[34m
фуксин	\033[35m
Сyan	\033[36m
белый	\033[37m

## Фоновые цвета

эффект	Код
черный	\033[40m
красный	\033[41m
зеленый	\033[42m
желтый	\033[43m
синий	\033[44m
фуксин	\033[45m
Сyan	\033[46m
белый	\033[47m

Прочитайте Взаимодействие с консолью онлайн: <https://riptutorial.com/ru/node-js/topic/5935/взаимодействие-с-консолью>

---

# глава 35: Внедрение зависимости

## Examples

### Зачем использовать инъекции зависимостей

1. Быстрый процесс разработки
2. Развязка
3. Ед.

#### Быстрый процесс разработки

При использовании разработчика узла инъекций зависимости может ускорить процесс разработки, поскольку после DI меньше конфликтов кода и легко управляется всем модулем.

#### Развязка

Модули становятся меньше пары, тогда их легко поддерживать.

#### Ед.

Закодированные зависимости могут передавать их в модуль, а затем легко записывать модульные тесты для каждого модуля.

Прочитайте Внедрение зависимости онлайн: <https://riptutorial.com/ru/node-js/topic/7681/внедрение-зависимости>

# глава 36: Всплывающие уведомления

## Вступление

Поэтому, если вы хотите сделать уведомление о веб-приложении, я предлагаю вам использовать структуру Push.js или SoneSignal для веб-приложения.

Push - это самый быстрый способ запустить и запустить с помощью Javascript-уведомлений. Довольно новое дополнение к официальной спецификации, Notification API позволяет современным браузерам, таким как Chrome, Safari, Firefox и IE 9+, отправлять уведомления на рабочий стол пользователя.

Вам нужно будет использовать Socket.io и некоторые бэкэнд-рамки, я буду использовать Express для этого примера.

## параметры

модуль / рамки	описание
Node.js / экспресс	Простая базовая инфраструктура для приложения Node.js, очень проста в использовании и чрезвычайно эффективна
Socket.io	Socket.IO обеспечивает двунаправленную связь на основе событий в режиме реального времени. Он работает на каждой платформе, в браузере или устройстве, одинаково ориентируясь на надежность и скорость.
Push.js	Самая универсальная в мире настольная система уведомлений
OneSignal	Еще одна форма с push-уведомлений для устройств Apple
Firebase	Firebase - это мобильная платформа Google, которая помогает быстро разрабатывать высококачественные приложения и развивать свой бизнес.

## Examples

### Веб-уведомление

Во-первых, вам нужно будет установить модуль [Push.js](#).

```
$ npm install push.js --save
```

Или импортируйте его в свое интерфейсное приложение через [CDN](#)

```
<script src="./push.min.js"></script> <!-- CDN link -->
```

После того, как вы закончите с этим, вам должно быть хорошо идти. Вот как это должно выглядеть, если вы хотите сделать простое уведомление:

```
Push.create('Hello World!')
```

Я предполагаю, что вы знаете, как настроить [Socket.io](#) с вашим приложением. Вот пример кода моего бэкэнд-приложения с выражением:

```
var app = require('express')();
var server = require('http').Server(app);
var io = require('socket.io')(server);

server.listen(80);

app.get('/', function (req, res) {
  res.sendFile(__dirname + '/index.html');
});

io.on('connection', function (socket) {

  socket.emit('pushNotification', { success: true, msg: 'hello' });

});
```

После того, как ваш сервер настроен, вы сможете перейти к интерфейсу. Теперь все, что нам нужно сделать, это импортировать [Socket.io CDN](#) и добавить этот код в файл *index.html* :

```
<script src="./socket.io.js"></script> <!-- CDN link -->
<script>
  var socket = io.connect('http://localhost');
  socket.on('pushNotification', function (data) {
    console.log(data);
    Push.create("Hello world!", {
      body: data.msg, //this should print "hello"
      icon: '/icon.png',
      timeout: 4000,
      onClick: function () {
        window.focus();
        this.close();
      }
    });
  });
</script>
```

Теперь вы можете показать свое уведомление, это также работает на любом устройстве Android, и если вы хотите использовать облачную [службу Firebase](#) , вы можете



использовать его с этим модулем. [Вот](#) ссылка на этот пример, написанный Ником (создателем от Push.js)

## яблоко

Имейте в виду, что это не будет работать на устройствах Apple (я их не тестировал), но если вы хотите сделать push-уведомления, проверьте плагин [OneSignal](#) .

Прочитайте [Всплывающие уведомления онлайн](#): <https://riptutorial.com/ru/node-js/topic/10892/всплывающие-уведомления>

# глава 37: Выполнение файлов или команд с помощью дочерних процессов

## Синтаксис

- `child_process.exec` (команда [, options] [, обратный вызов])
- `child_process.execFile` (файл [, args] [, options] [, callback])
- `child_process.fork` (modulePath [, args] [, options])
- `child_process.spawn` (команда [, args] [, options])
- `child_process.execFileSync` (файл [, args] [, options])
- `child_process.execSync` (команда [, options])
- `child_process.spawnSync` (команда [, args] [, options])

## замечания

При работе с дочерними процессами все асинхронные методы возвращают экземпляр `ChildProcess`, в то время как все синхронные версии возвращают вывод того, что было запущено. Как и другие синхронные операции в Node.js, если возникает ошибка, она *будет* бросать.

## Examples

### Создание нового процесса для выполнения команды

Чтобы создать новый процесс, в котором вам нужен *небуферизованный* вывод (например, длительные процессы, которые могут печатать выходные данные в течение определенного периода времени, а не сразу печатать и `child_process.spawn()`), используйте

```
child_process.spawn() .
```

Этот метод генерирует новый процесс, используя заданную команду и массив аргументов. Возвращаемое значение представляет собой экземпляр `ChildProcess`, который, в свою очередь, предоставляет свойства `stdout` и `stderr`. Оба эти потока являются экземплярами `stream.Readable`.

Следующий код эквивалентен использованию команды `ls -lh /usr`.

```
const spawn = require('child_process').spawn;
const ls = spawn('ls', ['-lh', '/usr']);

ls.stdout.on('data', (data) => {
  console.log(`stdout: ${data}`);
});
```

```
ls.stderr.on('data', (data) => {
  console.log(`stderr: ${data}`);
});

ls.on('close', (code) => {
  console.log(`child process exited with code ${code}`);
});
```

Другая команда примера:

```
zip -0vr "archive" ./image.png
```

Может быть написано как:

```
spawn('zip', ['-0vr', '"archive"', './image.png']);
```

## Истечение оболочки для выполнения команды

Чтобы запустить команду в оболочке, в которой требуется буферизованный вывод (т. `child_process.exec` Это не поток), используйте `child_process.exec`. Например, если вы хотите запустить команду `cat *.js file | wc -l`, без параметров, это будет выглядеть так:

```
const exec = require('child_process').exec;
exec('cat *.js file | wc -l', (err, stdout, stderr) => {
  if (err) {
    console.error(`exec error: ${err}`);
    return;
  }

  console.log(`stdout: ${stdout}`);
  console.log(`stderr: ${stderr}`);
});
```

Функция принимает до трех параметров:

```
child_process.exec(command[, options][, callback]);
```

Параметр команды является строкой и требуется, в то время как объект опций и обратный вызов являются необязательными. Если объект `options` не указан, `exec` будет использовать следующее по умолчанию:

```
{
  encoding: 'utf8',
  timeout: 0,
  maxBuffer: 200*1024,
  killSignal: 'SIGTERM',
  cwd: null,
  env: null
}
```

Объект `options` также поддерживает параметр `shell` по умолчанию `/bin/sh` в UNIX и `cmd.exe`

в Windows, параметр `uid` для установки идентификатора пользователя процесса и параметр `gid` для идентификатора группы.

Обратный вызов, который вызывается при выполнении команды, вызывается с тремя аргументами (`err`, `stdout`, `stderr`). Если команда выполняется успешно, значение `err` будет равно `null`, в противном случае это будет экземпляр `Error`, а `err.code` будет кодом выхода процесса, а `err.signal` - сигналом, который был отправлен для его завершения.

Аргументы `stdout` и `stderr` являются результатом команды. Он декодируется с кодировкой, указанной в объекте `options` (по умолчанию: `string`), но в противном случае может быть возвращен как объект `Buffer`.

Существует также синхронная версия `exec`, которая является `execSync`. Синхронная версия не выполняет обратный вызов и возвращает `stdout` вместо экземпляра `ChildProcess`. Если синхронная версия обнаруживает ошибку, он *будет* бросать и остановить вашу программу. Это выглядит так:

```
const execSync = require('child_process').execSync;
const stdout = execSync('cat *.js file | wc -l');
console.log(`stdout: ${stdout}`);
```

## Инициирование процесса запуска исполняемого файла

Если вы хотите запустить файл, например исполняемый файл, используйте `child_process.execFile`. Вместо того, чтобы `child_process.exec` такую оболочку, как `child_process.exec`, она будет напрямую создавать новый процесс, который немного эффективнее, чем запуск команды. Функция может быть использована следующим образом:

```
const execFile = require('child_process').execFile;
const child = execFile('node', ['--version'], (err, stdout, stderr) => {
  if (err) {
    throw err;
  }

  console.log(stdout);
});
```

В отличие от `child_process.exec`, эта функция будет принимать до четырех параметров, где второй параметр представляет собой массив аргументов, которые вы хотите предоставить исполняемому файлу:

```
child_process.execFile(file[, args][, options][, callback]);
```

В противном случае параметры и формат обратного вызова в противном случае идентичны `child_process.exec`. То же самое касается синхронной версии функции:

```
const execFileSync = require('child_process').execFileSync;
const stdout = execFileSync('node', ['--version']);
console.log(stdout);
```

Прочитайте [Выполнение файлов или команд с помощью дочерних процессов онлайн](https://riptutorial.com/ru/node-js/topic/2726/выполнение-файлов-или-команд-с-помощью-дочерних-процессов):  
<https://riptutorial.com/ru/node-js/topic/2726/выполнение-файлов-или-команд-с-помощью-дочерних-процессов>

---

# глава 38: Доставить HTML или любой другой файл

## Синтаксис

- `response.sendFile (имя_файла, параметры, функция (err) {});`

## Examples

### Доставить HTML по указанному пути

Вот как создать экспресс-сервер и обслуживать `index.html` по умолчанию (пустой путь `/`) и `page1.html` для пути `/page1`.

## Структура папок

```
project root
|  server.js
|  ___views
|     index.html
|     page1.html
```

## server.js

```
var express = require('express');
var path = require('path');
var app = express();

// deliver index.html if no file is requested
app.get("/", function (request, response) {
  response.sendFile(path.join(__dirname, 'views/index.html'));
});

// deliver page1.html if page1 is requested
app.get('/page1', function(request, response) {
  response.sendFile(path.join(__dirname, 'views', 'page1.html', function(error) {
    if (error) {
      // do something in case of error
      console.log(err);
      response.end(JSON.stringify({error:"page not found"}));
    }
  }));
});

app.listen(8080);
```

Обратите внимание, что `sendFile()` просто передает статический файл в качестве ответа, не предлагая его изменять. Если вы обслуживаете HTML-файл и хотите включить с ним динамические данные, вам нужно будет использовать механизм *шаблонов*, такой как Pug, Mustache или EJS.

Прочитайте [Доставить HTML или любой другой файл онлайн](https://riptutorial.com/ru/node-js/topic/6538/доставить-html-или-любой-другой-файл): <https://riptutorial.com/ru/node-js/topic/6538/доставить-html-или-любой-другой-файл>

---

# глава 39: Запуск node.js как службы

## Вступление

В отличие от многих веб-серверов, Node не устанавливается как служба из коробки. Но в производстве лучше использовать его как *dæmon*, управляемый системой *init*.

## Examples

### Node.js как системный *dæmon*

*systemd* является *де-факто* *init*-системой в большинстве дистрибутивов Linux. После того, как узел был настроен для работы с *systemd*, для его управления можно использовать *service* команду.

Прежде всего, ему нужен файл конфигурации, давайте его создадим. Для дистрибутивов на основе Debian это будет в `/etc/systemd/system/node.service`

```
[Unit]
Description=My super nodejs app

[Service]
# set the working directory to have consistent relative paths
WorkingDirectory=/var/www/app

# start the server file (file is relative to WorkingDirectory here)
ExecStart=/usr/bin/node serverCluster.js

# if process crashes, always try to restart
Restart=always

# let 500ms between the crash and the restart
RestartSec=500ms

# send log to syslog here (it doesn't compete with other log config in the app itself)
StandardOutput=syslog
StandardError=syslog

# nodejs process name in syslog
SyslogIdentifier=nodejs

# user and group starting the app
User=www-data
Group=www-data

# set the environment (dev, prod...)
Environment=NODE_ENV=production

[Install]
# start node at multi user system level (= sysVinit runlevel 3)
```



```
WantedBy=multi-user.target
```

Теперь можно запускать, останавливать и перезапускать приложение с помощью:

```
service node start  
service node stop  
service node restart
```

Чтобы сообщить `systemd` автоматически запускать узел при загрузке, просто введите:

```
systemctl enable node .
```

Вот и все, узел теперь работает как `dæmon`.

Прочитайте [Запуск node.js как службы онлайн: https://riptutorial.com/ru/node-js/topic/9258/запуск-node-js-как-службы](https://riptutorial.com/ru/node-js/topic/9258/запуск-node-js-как-службы)

# глава 40: Защита приложений Node.js

## Examples

### Предотвращение подделки подпроса (CSRF)

**CSRF** - это атака, которая заставляет конечного пользователя выполнять нежелательные действия в веб-приложении, в котором он / она сейчас аутентифицирован.

Это может произойти из-за того, что файлы cookie отправляются с каждым запросом на сайт - даже если эти запросы поступают с другого сайта.

Мы можем использовать модуль `csrf` для создания токена csrf и проверки его.

#### пример

```
var express = require('express')
var cookieParser = require('cookie-parser') //for cookie parsing
var csrf = require('csrf') //csrf module
var bodyParser = require('body-parser') //for body parsing

// setup route middlewares
var csrfProtection = csrf({ cookie: true })
var parseForm = bodyParser.urlencoded({ extended: false })

// create express app
var app = express()

// parse cookies
app.use(cookieParser())

app.get('/form', csrfProtection, function(req, res) {
  // generate and pass the csrfToken to the view
  res.render('send', { csrfToken: req.csrfToken() })
})

app.post('/process', parseForm, csrfProtection, function(req, res) {
  res.send('data is being processed')
})
```

Таким образом, когда мы обращаемся к `GET /form`, он передает представление csrf `csrfToken` в представление.

Теперь в представлении установите значение `csrfToken` как значение скрытого поля ввода с именем `_csrf`.

например, для шаблонов `handlebar`

```
<form action="/process" method="POST">
  <input type="hidden" name="_csrf" value="{{csrfToken}}">
```

```
Name: <input type="text" name="name">
<button type="submit">Submit</button>
</form>
```

например, для jade шаблонов

```
form(action="/process" method="post")
  input (type="hidden", name="_csrf", value=csrfToken)

  span Name:
    input (type="text", name="name", required=true)
  br

  input (type="submit")
```

например, для шаблонов ejs

```
<form action="/process" method="POST">
  <input type="hidden" name="_csrf" value="<%= csrfToken %>">
  Name: <input type="text" name="name">
  <button type="submit">Submit</button>
</form>
```

## SSL / TLS в Node.js

Если вы решите обрабатывать SSL / TLS в приложении Node.js, считайте, что вы также несете ответственность за поддержку предотвращения атак SSL / TLS на этом этапе. Во многих архитектурах сервер-клиент SSL / TLS заканчивается на обратном прокси, как для уменьшения сложности приложений, так и для уменьшения объема конфигурации безопасности.

Если ваше приложение Node.js должно обрабатывать SSL / TLS, его можно защитить, загрузив файлы ключей и сертификатов.

Если вашему поставщику сертификатов требуется цепочка сертификатов (CA), его можно добавить в опцию `ca` в качестве массива. Цепочка с несколькими записями в одном файле должна быть разделена на несколько файлов и введена в том же порядке в массив, что и Node.js в настоящее время не поддерживает несколько записей `ca` в одном файле. Пример приведен в приведенном ниже коде для файлов `1_ca.crt` и `2_ca.crt`. Если массив `ca` необходим и не установлен правильно, клиентские браузеры могут отображать сообщения, которые не могут подтвердить подлинность сертификата.

### пример

```
const https = require('https');
const fs = require('fs');

const options = {
  key: fs.readFileSync('privatekey.pem'),
  cert: fs.readFileSync('certificate.pem'),
```

```
ca: [fs.readFileSync('1_ca.crt'), fs.readFileSync('2_ca.crt')]
};

https.createServer(options, (req, res) => {
  res.writeHead(200);
  res.end('hello world\n');
}).listen(8000);
```

## Использование HTTPS

Минимальная настройка для HTTPS-сервера в Node.js будет примерно такой:

```
const https = require('https');
const fs = require('fs');

const httpsOptions = {
  key: fs.readFileSync('path/to/server-key.pem'),
  cert: fs.readFileSync('path/to/server-crt.pem')
};

const app = function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}

https.createServer(httpsOptions, app).listen(4433);
```

Если вы также хотите поддерживать HTTP-запросы, вам нужно сделать только эту небольшую модификацию:

```
const http = require('http');
const https = require('https');
const fs = require('fs');

const httpsOptions = {
  key: fs.readFileSync('path/to/server-key.pem'),
  cert: fs.readFileSync('path/to/server-crt.pem')
};

const app = function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}

http.createServer(app).listen(8888);
https.createServer(httpsOptions, app).listen(4433);
```

## Настройка сервера HTTPS

После установки в вашей системе узла node.js, просто следуйте приведенной ниже процедуре, чтобы получить базовый веб-сервер с поддержкой как HTTP, так и HTTPS!

## Шаг 1. Создание центра сертификации

1. создайте папку, в которой вы хотите сохранить свой ключ и сертификат:

```
mkdir conf
```

---

2. перейдите в этот каталог:

```
cd conf
```

---

3. возьмите этот файл `ca.cnf` для использования в качестве ярлыка конфигурации:

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/ca.cnf
```

---

4. создайте новый центр сертификации, используя эту конфигурацию:

```
openssl req -new -x509 -days 9999 -config ca.cnf -keyout ca-key.pem -out ca-cert.pem
```

---

5. теперь, когда у нас есть наш `ca-key.pem` сертификации в `ca-key.pem` и `ca-cert.pem`, давайте сгенерируем закрытый ключ для сервера:

```
openssl genrsa -out key.pem 4096
```

---

6. захватите этот файл `server.cnf` для использования в качестве ярлыка конфигурации:

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/server.cnf
```

---

7. сгенерировать запрос подписи сертификата с использованием этой конфигурации:

```
openssl req -new -config server.cnf -key key.pem -out csr.pem
```

---

8. Подпишите запрос:

```
openssl x509 -req -extfile server.cnf -days 999 -passin "pass:password" -in csr.pem -CA ca-cert.pem -CAkey ca-key.pem -CAcreateserial -out cert.pem
```

---

## Шаг 2. Установите сертификат как корневой сертификат.

1. скопируйте свой сертификат в папку корневых сертификатов:

```
sudo cp ca-crt.pem /usr/local/share/ca-certificates/ca-crt.pem
```

---

2. обновить магазин CA:

```
sudo update-ca-certificates
```

## Безопасное приложение express.js 3

Конфигурация для безопасного подключения с помощью express.js (Начиная с версии 3):

```
var fs = require('fs');
var http = require('http');
var https = require('https');
var privateKey = fs.readFileSync('sslcert/server.key', 'utf8');
var certificate = fs.readFileSync('sslcert/server.crt', 'utf8');

// Define your key and cert

var credentials = {key: privateKey, cert: certificate};
var express = require('express');
var app = express();

// your express configuration here

var httpServer = http.createServer(app);
var httpsServer = https.createServer(credentials, app);

// Using port 8080 for http and 8443 for https

httpServer.listen(8080);
httpsServer.listen(8443);
```

Таким образом, вы предоставляете прямое промежуточное ПО на собственный сервер http / https

Если вы хотите, чтобы ваше приложение работало на портах ниже 1024, вам нужно будет использовать команду `sudo` (не рекомендуется) или использовать обратный прокси (например, `nginx`, `haproxy`).

Прочитайте [Защита приложений Node.js онлайн: https://riptutorial.com/ru/node-js/topic/3473/защита-приложений-node-js](https://riptutorial.com/ru/node-js/topic/3473/защита-приложений-node-js)

---

# глава 41: Избегайте обратного вызова ада

## Examples

### Асинхронный модуль

---

Источник доступен для загрузки из GitHub. Кроме того, вы можете установить с помощью npm:

```
$ npm install --save async
```

Также, используя Bower:

```
$ bower install async
```

Пример:

```
var async = require("async");
async.parallel([
  function(callback) { ... },
  function(callback) { ... }
], function(err, results) {
  // optional callback
});
```

### Асинхронный модуль

К счастью, библиотеки, такие как Async.js, существуют, чтобы попытаться сдержать проблему. Async добавляет тонкий слой функций поверх вашего кода, но может значительно снизить сложность, избегая разворота обратного вызова.

В Async существует много вспомогательных методов, которые могут использоваться в разных ситуациях, таких как серия, параллель, водопад и т. Д. Каждая функция имеет конкретный прецедент, поэтому потребуется некоторое время, чтобы узнать, какой из них поможет в каких ситуациях.

Как и Async, как ничто, это не идеально. Его очень легко увлечь, объединяя ряды, параллельно, навсегда и т. Д., После чего вы вернетесь туда, где вы начали с грязного кода. Будьте осторожны, преждевременно оптимизируйте. Просто потому, что несколько асинхронных задач могут выполняться параллельно, не всегда означает, что они должны. В действительности, поскольку Node является только однопоточным, одновременные запущенные задачи при использовании Async практически не имеют прироста производительности.

Источник доступен для загрузки с <https://github.com/caolan/async>. Кроме того, вы можете

установить с помощью npm:

```
$ npm install --save async
```

Также, используя Bower:

```
$ bower install async
```

Водопад Асинка Пример:

```
var fs = require('fs');
var async = require('async');

var myFile = '/tmp/test';

async.waterfall([
  function(callback) {
    fs.readFile(myFile, 'utf8', callback);
  },
  function(txt, callback) {
    txt = txt + '\nAppended something!';
    fs.writeFile(myFile, txt, callback);
  }
], function (err, result) {
  if(err) return console.log(err);
  console.log('Appended text!');
});
```

Прочитайте **Избегайте обратного вызова ада онлайн**: <https://riptutorial.com/ru/node-js/topic/10045/избегайте-обратного-вызова-ада>



# глава 42: Излучатели событий

## замечания

Когда событие «срабатывает» (что означает то же самое, что «публикация события» или «излучение события»), каждый слушатель будет называться синхронно ( **источник** ) вместе с любыми сопроводительными данными, которые были переданы в `emit()` , по сколько аргументов вы передадите:

```
myDog.on('bark', (howLoud, howLong, howIntense) => {
  // handle the event
})
myDog.emit('bark', 'loudly', '5 seconds long', 'fiercely')
```

Слушатели будут вызваны в том порядке, в котором они были зарегистрированы:

```
myDog.on('urinate', () => console.log('My first thought was "Oh-no"'))
myDog.on('urinate', () => console.log('My second thought was "Not my lawn :)"))
myDog.emit('urinate')
// The console.logs will happen in the right order because they were registered in that order.
```

Но если вам нужно сначала прослушивать, прежде чем все остальные слушатели уже добавлены, вы можете использовать `prependListener()` следующим образом:

```
myDog.prependListener('urinate', () => console.log('This happens before my first and second thoughts, even though it was registered after them'))
```

Если вам нужно прослушать событие, но вы хотите только один раз услышать об этом, вы можете использовать `once` вместо `on` или `prependOnceListener` вместо `prependListener` . После того, как событие будет запущено, а слушатель будет вызван, слушатель автоматически будет удален и не будет вызываться снова при следующем запуске события.

Наконец, если вы хотите удалить всех слушателей и начать все сначала, не стесняйтесь делать это:

```
myDog.removeAllListeners()
```

## Examples

### HTTP-аналитика через событие-эмитент

В коде сервера HTTP (например, `server.js` ):

```
const EventEmitter = require('events')
```

```

const serverEvents = new EventEmitter()

// Set up an HTTP server
const http = require('http')
const httpServer = http.createServer((request, response) => {
  // Handler the request...
  // Then emit an event about what happened
  serverEvents.emit('request', request.method, request.url)
});

// Expose the event emitter
module.exports = serverEvents

```

В коде супервизора (например, `supervisor.js`):

```

const server = require('./server.js')
// Since the server exported an event emitter, we can listen to it for changes:
server.on('request', (method, url) => {
  console.log(`Got a request: ${method} ${url}`)
})

```

Всякий раз, когда сервер получает запрос, он выдает событие, называемое `request` который прослушивает супервизор, а затем супервизор может реагировать на событие.

## ОСНОВЫ

Излучатели событий встроены в узел и предназначены для pub-sub, где *издатель* будет генерировать события, на которые могут слушать и реагировать *абоненты*. В узле jargon издатели называются *Event Emitters*, и они излучают события, а абоненты называются *слушателями*, и они реагируют на события.

```

// Require events to start using them
const EventEmitter = require('events').EventEmitter;
// Dogs have events to publish, or emit
class Dog extends EventEmitter {};
class Food {};

let myDog = new Dog();

// When myDog is chewing, run the following function
myDog.on('chew', (item) => {
  if (item instanceof Food) {
    console.log('Good dog');
  } else {
    console.log(`Time to buy another ${item}`);
  }
});

myDog.emit('chew', 'shoe'); // Will result in console.log('Time to buy another shoe')
const bacon = new Food();
myDog.emit('chew', bacon); // Will result in console.log('Good dog')

```

В приведенном выше примере собака является издателем / `EventEmitter`, а функцией, которая проверяет элемент, является абонент / слушатель. Вы также можете сделать

больше слушателей:

```
myDog.on('bark', () => {
  console.log('WHO'S AT THE DOOR?');
  // Panic
});
```

Также могут быть несколько слушателей для одного события и даже удалить прослушиватели:

```
myDog.on('chew', takeADeepBreathe);
myDog.on('chew', calmDown);
// Undo the previous line with the next one:
myDog.removeListener('chew', calmDown);
```

Если вы хотите прослушивать событие только один раз, вы можете использовать:

```
myDog.once('chew', pet);
```

Который автоматически удалит слушателя без условий гонки.

## Получите имена событий, на которые подписаны

Функция **EventEmitter.eventNames ()** вернет массив, содержащий имена зарегистрированных в настоящее время событий.

```
const EventEmitter = require("events");
class MyEmitter extends EventEmitter{}

var emitter = new MyEmitter();

emitter
.on("message", function(){ //listen for message event
  console.log("a message was emitted!");
})
.on("message", function(){ //listen for message event
  console.log("this is not the right message");
})
.on("data", function(){ //listen for data event
  console.log("a data just occurred!!");
});

console.log(emitter.eventNames()); //=> ["message","data"]
emitter.removeAllListeners("data");//=> removeAllListeners to data event
console.log(emitter.eventNames()); //=> ["message"]
```

[Запуск в RunKit](#)

Получите количество зарегистрированных слушателей для прослушивания определенного события

Функция `Emitter.listenerCount (eventName)` вернет количество слушателей, которые в настоящее время прослушивают событие, предоставленное в качестве аргумента

```
const EventEmitter = require("events");
class MyEmitter extends EventEmitter{}
var emitter = new MyEmitter();

emitter
.on("data", ()=>{ // add listener for data event
  console.log("data event emitter");
});

console.log(emitter.listenerCount("data")) // => 1
console.log(emitter.listenerCount("message")) // => 0

emitter.on("message", function mListener(){ //add listener for message event
  console.log("message event emitted");
});
console.log(emitter.listenerCount("data")) // => 1
console.log(emitter.listenerCount("message")) // => 1

emitter.once("data", (stuff)=>{ //add another listener for data event
  console.log(`Tell me my ${stuff}`);
})

console.log(emitter.listenerCount("data")) // => 2
console.log(emitter.listenerCount("message")) // => 1
```

Прочитайте Излучатели событий онлайн: <https://riptutorial.com/ru/node-js/topic/1623/излучатели-событий>

---

# глава 43: Изящное завершение

## Examples

### Изящное завершение работы - SIGTERM

Используя **server.close ()** и **process.exit ()**, мы можем поймать исключение сервера и сделать изящное завершение работы.

```
var http = require('http');

var server = http.createServer(function (req, res) {
  setTimeout(function () { //simulate a long request
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
  }, 4000);
}).listen(9090, function (err) {
  console.log('listening http://localhost:9090/');
  console.log('pid is ' + process.pid);
});

process.on('SIGTERM', function () {
  server.close(function () {
    process.exit(0);
  });
});
```

Прочитайте Изящное завершение онлайн: <https://riptutorial.com/ru/node-js/topic/5996/изящное-завершение>

---

# глава 44: Интеграция MongoDB для Node.js / Express.js

## Вступление

MongoDB является одной из самых популярных баз данных NoSQL, благодаря помощи стека MEAN. Взаимодействие с базами данных Mongo из Express-приложения выполняется быстро и просто, как только вы понимаете синтаксис типа «произвольный». Мы будем использовать Мангуста, чтобы помочь нам.

## замечания

Более подробную информацию можно найти здесь: <http://mongoosejs.com/docs/guide.html>

## Examples

### Установка MongoDB

```
npm install --save mongodb
npm install --save mongoose //A simple wrapper for ease of development
```

В вашем файле сервера (обычно называемом index.js или server.js)

```
const express = require('express');
const mongodb = require('mongodb');
const mongoose = require('mongoose');
const mongoConnectionString = 'http://localhost/database name';

mongoose.connect(mongoConnectionString, (err) => {
  if (err) {
    console.log('Could not connect to the database');
  }
});
```

### Создание модели Мангуста

```
const Schema = mongoose.Schema;
const ObjectId = Schema.Types.ObjectId;

const Article = new Schema({
  title: {
    type: String,
    unique: true,
    required: [true, 'Article must have title']
  },
  author: {
```

```
    type: ObjectId,
    ref: 'User'
  }
});

module.exports = mongoose.model('Article', Article);
```

Давайте рассмотрим это. MongoDB и Mongoose используют JSON (фактически BSON, но это не имеет значения здесь) в качестве формата данных. Вверху, я установил несколько переменных, чтобы уменьшить набор.

Я создаю `new Schema` и назначаю ее константе. Это простой JSON, и каждый атрибут - это еще один объект со свойствами, которые помогают обеспечить более согласованную схему. Уникальные ссылки вставляют в базу данных новые экземпляры, очевидно, уникальные. Это отлично подходит для предотвращения создания пользователем нескольких учетных записей в службе.

Требуется другое, объявленное как массив. Первый элемент - это логическое значение, а второе сообщение об ошибке должно быть вставлено или обновлено.

ObjectIds используются для отношений между моделями. Примерами могут быть «У пользователей много комментариев». Вместо ObjectId могут использоваться другие атрибуты. Строки, такие как имя пользователя, являются одним из примеров.

Наконец, экспорт модели для использования с вашими маршрутами API обеспечивает доступ к вашей схеме.

## Запрос вашей базы данных Mongo

Простой запрос GET. Предположим, что модель из приведенного выше примера находится в файле `./db/models/Article.js`.

```
const express = require('express');
const Articles = require('./db/models/Article');

module.exports = function (app) {
  const routes = express.Router();

  routes.get('/articles', (req, res) => {
    Articles.find().limit(5).lean().exec((err, doc) => {
      if (doc.length > 0) {
        res.send({ data: doc });
      } else {
        res.send({ success: false, message: 'No documents retrieved' });
      }
    });
  });
});

app.use('/api', routes);
};
```

Теперь мы можем получить данные из нашей базы данных, отправив HTTP-запрос этой конечной точке. Несколько ключевых моментов:

1. `Limit` делает именно то, на что похоже. Я получаю только 5 документов.
2. `Lean` удаляет некоторые вещи из сырой BSON, уменьшая сложность и накладные расходы. Не требуется. Но полезно.
3. При использовании `find` вместо `findOne` убедитесь, что `doc.length` больше 0. Это происходит потому, что `find` всегда возвращает массив, поэтому пустой массив не будет обрабатывать вашу ошибку, если он не проверяется на длину
4. Мне лично нравится отправлять сообщение об ошибке в этом формате. Измените его в соответствии с вашими потребностями. То же самое для возвращаемого документа.
5. Код в этом примере написан в предположении, что вы поместили его в другой файл, а не непосредственно на экспресс-сервер. Чтобы вызвать это на сервере, включите эти строки в код сервера:

```
const app = express();
require('./path/to/this/file')(app) //
```

Прочитайте [Интеграция MongoDB для Node.js / Express.js онлайн](https://riptutorial.com/ru/node-js/topic/9020/интеграция-mongodb-для-node-js---express-js):

<https://riptutorial.com/ru/node-js/topic/9020/интеграция-mongodb-для-node-js---express-js>



---

# глава 45: Интеграция MSSQL

## Вступление

Для интеграции любой базы данных с nodejs вам нужен пакет драйверов, или вы можете назвать его модулем прт, который предоставит вам базовый API для подключения к базе данных и выполнения взаимодействия. То же самое верно и для базы данных mssql, здесь мы будем интегрировать mssql с nodejs и выполнить некоторые базовые запросы в таблицах SQL.

## замечания

Мы предположили, что у нас будет локальный экземпляр сервера базы данных mssql, работающего на локальной машине. Вы можете сослаться на [этот документ](#), чтобы сделать то же самое.

Также убедитесь, что вы создали соответствующий пользователь с добавленными привилегиями.

## Examples

### Подключение через SQL через. Модуль mssql npm

Мы начнем с создания простого узла с базовой структурой и последующего соединения с локальной базой данных SQL Server и выполнения некоторых запросов в этой базе данных.

**Шаг 1.** Создайте каталог / папку по названию проекта, который вы намерены создать. Инициализируйте приложение узла с помощью команды `npm init`, которая создаст `package.json` в текущем каталоге.

```
mkdir mySqlApp
//folder created
cd mwSqlApp
//change to newly created directory
npm init
//answer all the question ..
npm install
//This will complete quickly since we have not added any packages to our app.
```

**Шаг 2.** Теперь мы создадим файл `App.js` в этом каталоге и установим некоторые пакеты, которые нам понадобятся для подключения к sql db.

```
sudo gedit App.js
//This will create App.js file , you can use your fav. text editor :)
```

```
npm install --save mssql
//This will install the mssql package to you app
```

**Шаг 3:** Теперь мы добавим базовую конфигурационную переменную в наше приложение, которое будет использоваться модулем mssql для установления соединения.

```
console.log("Hello world, This is an app to connect to sql server.");
var config = {
  "user": "myusername", //default is sa
  "password": "yourStrong(!)Password",
  "server": "localhost", // for local machine
  "database": "staging", // name of database
  "options": {
    "encrypt": true
  }
}

sql.connect(config, err => {
  if(err){
    throw err ;
  }
  console.log("Connection Successful !");

  new sql.Request().query('select 1 as number', (err, result) => {
    //handle err
    console.dir(result)
    // This example uses callbacks strategy for getting results.
  })
});

sql.on('error', err => {
  // ... error handler
  console.log("Sql database connection error " ,err);
})
```

**Шаг 4:** Это самый простой шаг, когда мы запускаем приложение, и приложение будет подключаться к серверу sql и распечатывать некоторые простые результаты.

```
node App.js
// Output :
// Hello world, This is an app to connect to sql server.
// Connection Successful !
// 1
```

Для использования обещаний или асинхронизации для выполнения запроса обратитесь к официальным документам пакета mssql:

- [обещания](#)
- [Асинхронный / Await](#)

Прочитайте [Интеграция MSSQL онлайн: https://riptutorial.com/ru/node-js/topic/9884/интеграция-mssql](https://riptutorial.com/ru/node-js/topic/9884/интеграция-mssql)

# глава 46: Интеграция MySQL

## Вступление

В этом разделе вы узнаете, как интегрироваться с Node.js с помощью инструмента управления базами данных MySQL. Вы узнаете различные способы подключения и взаимодействия с данными, находящимися в mysql, с помощью программы и скрипта nodejs.

## Examples

### Запрос объекта соединения с параметрами

Когда вы хотите использовать созданный пользователем контент в SQL, он выполняется с параметрами. Например, для поиска пользователя с именем `aminadav` вы должны:

```
var username = 'aminadav';
var querystring = 'SELECT name, email from users where name = ?';
connection.query(querystring, [username], function(err, rows, fields) {
  if (err) throw err;
  if (rows.length) {
    rows.forEach(function(row) {
      console.log(row.name, 'email address is', row.email);
    });
  } else {
    console.log('There were no results.');
```

### Использование пула соединений

#### а. Выполнение нескольких запросов одновременно

Все запросы в соединении MySQL выполняются один за другим. Это означает, что если вы хотите сделать 10 запросов, и каждый запрос занимает 2 секунды, для завершения всего выполнения потребуется 20 секунд. Решение состоит в том, чтобы создать 10 соединений и запустить каждый запрос в другом соединении. Это можно сделать автоматически, используя пул соединений

```
var pool = mysql.createPool({
  connectionLimit : 10,
  host             : 'example.org',
  user            : 'bobby',
  password       : 'pass',
  database       : 'schema'
});

for(var i=0;i<10;i++){
```

```
pool.query('SELECT ` as example', function(err, rows, fields) {
  if (err) throw err;
  console.log(rows[0].example); //Show 1
});
}
```

Он будет запускать все 10 запросов параллельно.

Когда вы используете `pool` вам больше не нужно подключение. Вы можете напрямую запросить пул. Модуль MySQL будет искать следующее бесплатное соединение для выполнения вашего запроса.

## б. Достижение многоуровневости на сервере базы данных с различными базами данных, размещенными на нем.

В настоящее время многозадачность является общим требованием к корпоративному приложению, и создание пула соединений для каждой базы данных на сервере базы данных не рекомендуется. поэтому вместо этого мы можем создать пул соединений с сервером базы данных и затем переключать их между базами данных, размещенными на сервере базы данных по требованию.

Предположим, что наше приложение имеет разные базы данных для каждой фирмы, размещенной на сервере базы данных. Мы подключаемся к соответствующей базе данных фирмы, когда пользователь обращается к приложению. Вот пример того, как это сделать: -

```
var pool = mysql.createPool({
  connectionLimit : 10,
  host             : 'example.org',
  user            : 'bobby',
  password       : 'pass'
});

pool.getConnection(function(err, connection){
  if(err){
    return cb(err);
  }
  connection.changeUser({database : "firm1"});
  connection.query("SELECT * from history", function(err, data){
    connection.release();
    cb(err, data);
  });
});
```

Позвольте мне разбить пример:

При определении конфигурации пула я не дал имя базы данных, но дал только сервер базы данных, т. Е.

```
{
  connectionLimit : 10,
  host             : 'example.org',
  user            : 'bobby',
  password       : 'pass'
}
```

поэтому, когда мы хотим использовать конкретную базу данных на сервере базы данных, мы запрашиваем подключение к базе данных, используя -

```
connection.changeUser({database : "firm1"});
```

вы можете сослаться на официальную документацию [здесь](#)

## Подключение к MySQL

Одним из самых простых способов подключения к MySQL является использование модуля `mysql`. Этот модуль обрабатывает соединение между приложением Node.js и сервером MySQL. Вы можете установить его, как и любой другой модуль:

```
npm install --save mysql
```

Теперь вам нужно создать соединение `mysql`, которое вы можете позже запросить.

```
const mysql      = require('mysql');
const connection = mysql.createConnection({
  host      : 'localhost',
  user     : 'me',
  password : 'secret',
  database : 'database_schema'
});

connection.connect();

// Execute some query statements
// I.e. SELECT * FROM FOO

connection.end();
```

В следующем примере вы узнаете, как запрашивать объект `connection`.

## Запрос объекта соединения без параметров

Вы отправляете запрос в виде строки и в ответном обратном вызове ответ получен. Обратный вызов дает вам `error`, массив `rows` и полей. Каждая строка содержит весь столбец возвращенной таблицы. Ниже приведен фрагмент следующего объяснения.

```
connection.query('SELECT name,email from users', function(err, rows, fields) {
  if (err) throw err;
```

```
console.log('There are:', rows.length, ' users');
console.log('First user name is:',rows[0].name)
});
```

## Запустить несколько запросов с одним соединением из пула

Могут быть ситуации, когда вы настроили пул соединений MySQL, но у вас есть несколько запросов, которые вы хотели бы запустить последовательно:

```
SELECT 1;
SELECT 2;
```

Вы можете просто запустить, используя `pool.query` как показано в другом месте , однако, если у вас есть только одно бесплатное соединение в пуле, вы должны подождать, пока соединение станет доступным, прежде чем вы сможете запустить второй запрос.

Однако вы можете сохранить активное соединение из пула и запустить столько запросов, сколько хотите использовать одно соединение, используя `pool.getConnection` :

```
pool.getConnection(function (err, conn) {
  if (err) return callback(err);

  conn.query('SELECT 1 AS seq', function (err, rows) {
    if (err) throw err;

    conn.query('SELECT 2 AS seq', function (err, rows) {
      if (err) throw err;

      conn.release();
      callback();
    });
  });
});
```

**Примечание.** Вы должны помнить о том, чтобы `release` соединение, иначе есть еще одно соединение MySQL, доступное для остальной части пула!

Для получения дополнительной информации об объединении соединений MySQL [ознакомьтесь с документами MySQL](#) .

## Вернуть запрос при возникновении ошибки

Вы можете присоединить запрос, выполняемый к вашему объекту `err` когда возникает ошибка:

```
var q = mysql.query('SELECT `name` FROM `pokedex` WHERE `id` = ?', [ 25 ], function (err, result) {
  if (err) {
    // Table 'test.pokedex' doesn't exist
    err.query = q.sql; // SELECT `name` FROM `pokedex` WHERE `id` = 25
  }
});
```

```
    callback(err);
  }
  else {
    callback(null, result);
  }
});
```

## Пул подключения экспорта

```
// db.js

const mysql = require('mysql');

const pool = mysql.createPool({
  connectionLimit : 10,
  host             : 'example.org',
  user            : 'bob',
  password        : 'secret',
  database        : 'my_db'
});

module.export = {
  getConnection: (callback) => {
    return pool.getConnection(callback);
  }
}
```

```
// app.js

const db = require('./db');

db.getConnection((err, conn) => {
  conn.query('SELECT something from sometable', (error, results, fields) => {
    // get the results
    conn.release();
  });
});
```

Прочитайте Интеграция MySQL онлайн: <https://riptutorial.com/ru/node-js/topic/1406/интеграция-mysql>

---

# глава 47: Интеграция PostgreSQL

## Examples

### Подключиться к PostgreSQL

Использование модуля `npm PostgreSQL` .

установить зависимость от `npm`

```
npm install pg --save
```

Теперь вам нужно создать соединение PostgreSQL, которое вы можете позже запросить.

Предположим, вы `Database_Name = студенты`, `Host = localhost` и `DB_User = postgres`

```
var pg = require("pg")
var connectionString = "pg://postgres:postgres@localhost:5432/students";
var client = new pg.Client(connectionString);
client.connect();
```

### Запрос с объектом подключения

Если вы хотите использовать объект подключения для базы данных запросов, вы можете использовать этот пример кода.

```
var queryString = "SELECT name, age FROM students " ;
var query = client.query(queryString);

query.on("row", (row, result)=> {
  result.addRow(row);
});

query.on("end", function (result) {
  //LOGIC
});
```

Прочитайте [Интеграция PostgreSQL онлайн: https://riptutorial.com/ru/node-js/topic/7706/интеграция-postgresql](https://riptutorial.com/ru/node-js/topic/7706/интеграция-postgresql)



---

# глава 48: Интеграция Кассандры

## Examples

### Привет, мир

Для доступа к Cassandra `cassandra-driver` модуль `cassandra-driver` из DataStax можно использовать. Он поддерживает все функции и может быть легко настроен.

```
const cassandra = require("cassandra-driver");
const clientOptions = {
  contactPoints: ["host1", "host2"],
  keyspace: "test"
};

const client = new cassandra.Client(clientOptions);

const query = "SELECT hello FROM world WHERE name = ?";
client.execute(query, ["John"], (err, results) => {
  if (err) {
    return console.error(err);
  }

  console.log(results.rows);
});
```

Прочитайте Интеграция Кассандры онлайн: <https://riptutorial.com/ru/node-js/topic/5949/интеграция-кассандры>

# глава 49: Интеграция паспорта

## замечания

Пароль **всегда** должен быть хэширован. Простым способом защиты паролей с помощью **NodeJS** будет использование модуля **bcrypt-nodejs** .

## Examples

### Начиная

**Паспорт** должен быть инициализирован с использованием `passport.initialize()` промежуточного программного обеспечения. Чтобы использовать сеансы входа в систему, требуется промежуточное программное обеспечение `passport.session()` .

Обратите внимание, что должны быть определены методы `passport.serialize()` и `passport.deserializeUser()` . **Паспорт** будет сериализовать и десериализовать пользовательские экземпляры на сеанс и с него

```
const express = require('express');
const session = require('express-session');
const passport = require('passport');
const cookieParser = require('cookie-parser');
const app = express();

// Required to read cookies
app.use(cookieParser());

passport.serializeUser(function(user, next) {
  // Serialize the user in the session
  next(null, user);
});

passport.deserializeUser(function(user, next) {
  // Use the previously serialized user
  next(null, user);
});

// Configuring express-session middleware
app.use(session({
  secret: 'The cake is a lie',
  resave: true,
  saveUninitialized: true
}));

// Initializing passport
app.use(passport.initialize());
app.use(passport.session());

// Starting express server on port 3000
app.listen(3000);
```

## Локальная проверка

**Паспорт-локальный** модуль используется для реализации локальной аутентификации.

Этот модуль позволяет выполнять аутентификацию с использованием имени пользователя и пароля в ваших приложениях Node.js.

Регистрация пользователя:

```
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;

// A named strategy is used since two local strategy are used :
// one for the registration and the other to sign-in
passport.use('localSignup', new LocalStrategy({
  // Overriding defaults expected parameters,
  // which are 'username' and 'password'
  usernameField: 'email',
  passwordField: 'password',
  passReqToCallback: true // allows us to pass back the entire request to the callback
}),
function(req, email, password, next) {
  // Check in database if user is already registered
  findUserByEmail(email, function(user) {
    // If email already exists, abort registration process and
    // pass 'false' to the callback
    if (user) return next(null, false);
    // Else, we create the user
    else {
      // Password must be hashed !
      let newUser = createUser(email, password);

      newUser.save(function() {
        // Pass the user to the callback
        return next(null, newUser);
      });
    }
  });
});
```

Вход в систему пользователя:

```
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;

passport.use('localSignin', new LocalStrategy({
  usernameField : 'email',
  passwordField : 'password',
}),
function(email, password, next) {
  // Find the user
  findUserByEmail(email, function(user) {
    // If user is not found, abort signing in process
    // Custom messages can be provided in the verify callback
    // to give the user more details concerning the failed authentication
    if (!user)
      return next(null, false, {message: 'This e-mail address is not associated with any
```

```

account.});
  // Else, we check if password is valid
  else {
    // If password is not correct, abort signing in process
    if (!isPasswordValid(password)) return next(null, false);
    // Else, pass the user to callback
    else return next(null, user);
  }
});
});

```

## Создание маршрутов:

```

// ...
app.use(passport.initialize());
app.use(passport.session());

// Sign-in route
// Passport strategies are middlewares
app.post('/login', passport.authenticate('localSignin', {
  successRedirect: '/me',
  failureRedirect: '/login'
}));

// Sign-up route
app.post('/register', passport.authenticate('localSignup', {
  successRedirect: '/',
  failureRedirect: '/signup'
}));

// Call req.logout() to log out
app.get('/logout', function(req, res) {
  req.logout();
  res.redirect('/');
});

app.listen(3000);

```

## Проверка подлинности Facebook

Модуль **passport-facebook** используется для реализации аутентификации **Facebook**. В этом примере, если пользователь не существует при входе в систему, он создается.

### Стратегия реализации:

```

const passport = require('passport');
const FacebookStrategy = require('passport-facebook').Strategy;

// Strategy is named 'facebook' by default
passport.use({
  clientID: 'yourclientid',
  clientSecret: 'yourclientsecret',
  callbackURL: '/auth/facebook/callback'
},
// Facebook will send a token and user's profile
function(token, refreshToken, profile, next) {

```

```

// Check in database if user is already registered
findUserByFacebookId(profile.id, function(user) {
  // If user exists, returns his data to callback
  if (user) return next(null, user);
  // Else, we create the user
  else {
    let newUser = createUserFromFacebook(profile, token);

    newUser.save(function() {
      // Pass the user to the callback
      return next(null, newUser);
    });
  }
});
});

```

## Создание маршрутов:

```

// ...
app.use(passport.initialize());
app.use(passport.session());

// Authentication route
app.get('/auth/facebook', passport.authenticate('facebook', {
  // Ask Facebook for more permissions
  scope : 'email'
}));

// Called after Facebook has authenticated the user
app.get('/auth/facebook/callback',
  passport.authenticate('facebook', {
    successRedirect : '/me',
    failureRedirect : '/'
}));

//...

app.listen(3000);

```

## Простая аутентификация пользователя и пароля

В ваших маршрутах / index.js

Здесь `user` является моделью для `userSchema`

```

router.post('/login', function(req, res, next) {
  if (!req.body.username || !req.body.password) {
    return res.status(400).json({
      message: 'Please fill out all fields'
    });
  }

  passport.authenticate('local', function(err, user, info) {
    if (err) {
      console.log("ERROR : " + err);
    }
  }

```

```

        return next(err);
    }

    if(user) {
        console.log("User Exists!");
        //All the data of the user can be accessed by user.x
        res.json({"success" : true});
        return;
    } else {
        res.json({"success" : false});
        console.log("Error" + errorResponse());
        return;
    }
})(req, res, next);
});

```

## Аутентификация Google Passport

У нас есть простой модуль, доступный в версии для npm для прочтения звуков имени **паспорта-google-oauth20**

Рассмотрим следующий пример. В этом примере создана папка, а именно config с файлом passport.js и google.js в корневом каталоге. В ваш app.js входят следующие

```

var express = require('express');
var session = require('express-session');
var passport = require('./config/passport'); // path where the passport file placed
var app = express();
passport(app);

```

// другой код для инициализации сервера, ошибка

В файле passport.js в папке config укажите следующий код

```

var passport = require ('passport'),
google = require('./google'),
User = require('../model/user'); // User is the mongoose model

module.exports = function(app){
    app.use(passport.initialize());
    app.use(passport.session());
    passport.serializeUser(function(user, done){
        done(null, user);
    });
    passport.deserializeUser(function (user, done) {
        done(null, user);
    });
    google();
};

```

В файл google.js в той же папке конфигурации входят следующие

```

var passport = require('passport'),
GoogleStrategy = require('passport-google-oauth20').Strategy,

```

```

User = require('../model/user');
module.exports = function () {
passport.use(new GoogleStrategy({
  clientID: 'CLIENT ID',
  clientSecret: 'CLIENT SECRET',
  callbackURL: "http://localhost:3000/auth/google/callback"
}),
function(accessToken, refreshToken, profile, cb) {
  User.findOne({ googleId : profile.id }, function (err, user) {
    if(err){
      return cb(err, false, {message : err});
    }else {
      if (user != '' && user != null) {
        return cb(null, user, {message : "User "});
      } else {
        var username = profile.displayName.split(' ');
        var userData = new User({
          name : profile.displayName,
          username : username[0],
          password : username[0],
          facebookId : '',
          googleId : profile.id,
        });
        // send email to user just in case required to send the newly created
        // credentials to user for future login without using google login
        userData.save(function (err, newuser) {
          if (err) {
            return cb(null, false, {message : err + " !!! Please try again"});
          }else{
            return cb(null, newuser);
          }
        });
      }
    }
  });
});
};

```

Здесь, в этом примере, если пользователь не находится в БД, тогда создается новый пользователь в БД для локальной ссылки, используя имя поля googleId в пользовательской модели.

Прочитайте Интеграция паспорта онлайн: <https://riptutorial.com/ru/node-js/topic/7666/интеграция-паспорта>

---

# глава 50: Использование Browserify для разрешения «требуемой» ошибки с помощью браузеров

## Examples

### Пример - file.js

В этом примере у нас есть файл с именем **file.js**.

Предположим, что вам нужно проанализировать URL-адрес с помощью JavaScript и модуля запросов NodeJS.

Для этого вам нужно всего лишь вставить следующий файл в свой файл:

```
const querystring = require('querystring');
var ref = querystring.parse("foo=bar&abc=xyz&abc=123");
```

---

## Что делает этот фрагмент?

Итак, во-первых, мы создаем модуль `querystring`, который предоставляет утилиты для синтаксического анализа и форматирования строк запроса URL. Доступ к нему можно получить, используя:

```
const querystring = require('querystring');
```

Затем мы анализируем URL-адрес, используя метод `.parse()`. Он анализирует строку запроса URL (`str`) в набор пар ключ и значение.

Например, строка запроса `'foo=bar&abc=xyz&abc=123'` анализируется на:

```
{ foo: 'bar', abc: ['xyz', '123'] }
```

К сожалению, у браузеров не установлен метод `require`, но Node.js.

---

## Установка браузера

С помощью Browserify вы можете писать код, который *требует* использования так же, как и в Node. Итак, как вы это решаете? Это просто.



1. Первый установочный узел, который поставляется с npm. Затем выполните:

```
npm install -g browserify
```

2. Перейдите в каталог , в котором ваш file.js является и установить наш модуль с НПМ строки запроса:

```
npm install querystring
```

**Примечание.** Если вы не изменяетесь в конкретном каталоге, команда завершится неудачно, так как не сможет найти файл, содержащий модуль.

3. Теперь рекурсивно свяжите все необходимые модули, начиная с file.js, в один файл с именем bundle.js (или как вам нравится его называть) с помощью **команды браузера** :

```
browserify file.js -o bundle.js
```

Browserify анализирует синтаксическое дерево Abstract для запросов `require()`, чтобы пересечь весь график зависимостей вашего

4. Наконец Заверните один тег в свой html, и все готово!

```
<script src="bundle.js"></script>
```

Случается, что вы получаете комбинацию старого .js-файла ( **file.js** ) и вашего вновь созданного файла **bundle.js** . Эти два файла объединяются в один файл.

---

## Важный

Имейте в виду, что если вы хотите внести какие-либо изменения в файл file.js и не повлияете на поведение вашей программы. **Ваши изменения вступят в силу только если вы отредактируете вновь созданный файл bundle.js**

---

## Что это значит?

Это означает, что если вы хотите редактировать **file.js** по каким-либо причинам, изменения не будут иметь никаких эффектов. Вам действительно нужно отредактировать **bundle.js**, так как это слияние **bundle.js** и **file.js**.

Прочитайте [Использование Browserify для разрешения «требуемой» ошибки с помощью браузеров онлайн](https://riptutorial.com/ru/node-js/topic/7123/использование-browserify-для-разрешения-требуемой-ошибки-с-помощью-браузеров): <https://riptutorial.com/ru/node-js/topic/7123/использование-browserify-для-разрешения-требуемой-ошибки-с-помощью-браузеров>

---

# глава 51: Использование IISNode для размещения веб-приложений Node.js в IIS

## замечания

---

## Виртуальный каталог / Вложенное приложение с представлениями Pitfall

Если вы собираетесь использовать Express для визуализации представлений с помощью механизма просмотра, вам нужно передать значение `virtualDirPath` в свои представления

```
`res.render('index', { virtualDirPath: virtualDirPath });`
```

Причина этого заключается в том, чтобы ваши гиперссылки на другие представления, размещаемые вашим приложением и статическими путями ресурсов, знали, где находится сайт, без необходимости изменять все представления после развертывания. Это одна из наиболее раздражающих и утомительных ошибок использования виртуальных каталогов с IISNode.

---

## Версии

Все приведенные выше примеры работают с

- Express v4.x
- IIS 7.x / 8.x
- Socket.io v1.3.x или выше

## Examples

### Начиная

[IISNode](#) позволяет размещать веб-приложения Node.js в IIS 7/8, как и приложение .NET. Конечно, вы можете самостоятельно разместить процесс `node.exe` в Windows, но зачем это делать, когда вы можете просто запустить приложение в IIS.

IISNode будет обрабатывать масштабирование по нескольким ядрам, управлять процессом `node.exe` и автоматически перерабатывать ваше приложение IIS всякий раз, когда ваше приложение обновляется, чтобы назвать несколько его [преимуществ](#) .

# Требования

У IISNode есть несколько требований, прежде чем вы сможете разместить приложение Node.js в IIS.

1. Node.js должен быть установлен на хост IIS, 32-разрядный или 64-разрядный, либо поддерживаются.
2. IISNode установлен [x86](#) или [x64](#) , это должно соответствовать битности вашего хоста IIS.
3. Модуль [URL-Rewrite Microsoft для IIS](#) установлен на вашем хосте IIS.
  - Это ключ, иначе запросы к вашему Node.js-приложению не будут функционировать должным образом.
4. `Web.config` в корневой папке вашего приложения Node.js.
5. IISNode через файл `iisnode.yml` или элемент `<iisnode>` в вашем `Web.config` .

## Основной пример Hello World с использованием Express

Чтобы этот пример работал, вам нужно создать приложение IIS 7/8 на хосте IIS и добавить каталог, содержащий веб-приложение Node.js в качестве физического каталога.

Убедитесь, что идентификатор пула приложений / приложений может получить доступ к установке Node.js. В этом примере используется 64-разрядная установка Node.js.

---

## Project Structure

Это основная структура проекта веб-приложения IISNode / Node.js. Он выглядит почти идентично любому веб-приложению, отличному от IISNode, за исключением добавления

`Web.config` .

```
- /app_root
- package.json
- server.js
- Web.config
```

---

## server.js - Экспресс-приложение

```
const express = require('express');
const server = express();

// We need to get the port that IISNode passes into us
// using the PORT environment variable, if it isn't set use a default value
const port = process.env.PORT || 3000;

// Setup a route at the index of our app
server.get('/', (req, res) => {
```

```
    return res.status(200).send('Hello World');
  });

  server.listen(port, () => {
    console.log(`Listening on ${port}`);
  });
```

## Конфигурация и Web.config

Web.config аналогичен любому другому IIS Web.config за исключением следующих двух вещей: URL `<rewrite><rules>` и IISNode `<handler>`. Оба эти элемента являются `<system.webServer>` элементами элемента `<system.webServer>`.

### конфигурация

Вы можете настроить IISNode с помощью `iisnode.yml` файл или путем добавления `<iisnode>` элемент в качестве дочернего `<system.webServer>` в вашем Web.config. Обе эти конфигурации могут использоваться совместно друг с другом, однако в этом случае Web.config должен будет указать файл `iisnode.yml` и любые конфликты конфигурации будут `iisnode.yml` файла `iisnode.yml`. Это переопределение конфигурации не может произойти наоборот.

### Обработчик IISNode

Чтобы IIS знал, что `server.js` содержит наше веб-приложение Node.js, нам нужно явно сказать об этом. Мы можем сделать это, добавив IISNode `<handler>` к элементу `<handlers>`.

```
<handlers>
  <add name="iisnode" path="server.js" verb="*" modules="iisnode"/>
</handlers>
```

### Правила перезаписи URL-адресов

Последняя часть конфигурации гарантирует, что трафик, предназначенный для нашего приложения Node.js, входящего в наш экземпляр IIS, направляется в IISNode. Без правил перезаписи URL нам нужно будет посетить наше приложение, перейдя по `http://<host>/server.js` и еще хуже, при попытке запросить ресурс, предоставленный `server.js` вы получите 404. Вот почему переписывание URL-адресов необходимо для веб-приложений IISNode.

```
<rewrite>
  <rules>
    <!-- First we consider whether the incoming URL matches a physical file in the /public
    folder -->
    <rule name="StaticContent" patternSyntax="Wildcard">
```

```
<action type="Rewrite" url="public/{R:0}" logRewrittenUrl="true"/>
<conditions>
  <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true"/>
</conditions>
<match url="*.*/>
</rule>

<!-- All other URLs are mapped to the Node.js application entry point -->
<rule name="DynamicContent">
  <conditions>
    <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="True"/>
  </conditions>
  <action type="Rewrite" url="server.js"/>
</rule>
</rules>
</rewrite>
```

Это рабочий файл `Web.config` для этого примера , установка для 64-разрядной установки Node.js.

Вот и все, теперь зайдите на свой сайт IIS и посмотрите, как работает ваше приложение Node.js.

## Использование виртуального каталога IIS или вложенного приложения через

Использование виртуального каталога или вложенного приложения в IIS является распространенным сценарием и, скорее всего, тем, что вы захотите использовать при использовании IISNode.

IISNode не обеспечивает прямую поддержку виртуальных каталогов или вложенных приложений через конфигурацию, поэтому для достижения этой цели нам нужно воспользоваться функцией IISNode, которая не является частью конфигурации и гораздо менее известна. Все дочерние `<appSettings>` с `Web.config` добавляются в объект `process.env` как свойства с помощью ключа `appSetting`.

Позволяет создать виртуальный каталог в нашем `<appSettings>`

```
<appSettings>
  <add key="virtualDirPath" value="/foo" />
</appSettings>
```

В нашем приложении Node.js мы можем получить доступ к настройке `virtualDirPath`

```
console.log(process.env.virtualDirPath); // prints /foo
```

Теперь, когда мы можем использовать элемент `<appSettings>` для конфигурации, давайте воспользуемся этим и использовать его в нашем коде сервера.

```
// Access the virtualDirPath appSettings and give it a default value of '/'
// in the event that it doesn't exist or isn't set
var virtualDirPath = process.env.virtualDirPath || '/';

// We also want to make sure that our virtualDirPath
// always starts with a forward slash
if (!virtualDirPath.startsWith('/', 0))
    virtualDirPath = '/' + virtualDirPath;

// Setup a route at the index of our app
server.get(virtualDirPath, (req, res) => {
    return res.status(200).send('Hello World');
});
```

## Мы также можем использовать virtualDirPath с нашими статическими ресурсами

```
// Public Directory
server.use(express.static(path.join(virtualDirPath, 'public')));
// Bower
server.use('/bower_components', express.static(path.join(virtualDirPath,
'bower_components')));
```

## Давайте все вместе

```
const express = require('express');
const server = express();

const port = process.env.PORT || 3000;

// Access the virtualDirPath appSettings and give it a default value of '/'
// in the event that it doesn't exist or isn't set
var virtualDirPath = process.env.virtualDirPath || '/';

// We also want to make sure that our virtualDirPath
// always starts with a forward slash
if (!virtualDirPath.startsWith('/', 0))
    virtualDirPath = '/' + virtualDirPath;

// Public Directory
server.use(express.static(path.join(virtualDirPath, 'public')));
// Bower
server.use('/bower_components', express.static(path.join(virtualDirPath,
'bower_components')));

// Setup a route at the index of our app
server.get(virtualDirPath, (req, res) => {
    return res.status(200).send('Hello World');
});

server.listen(port, () => {
    console.log(`Listening on ${port}`);
});
```

## Использование Socket.io с IISNode

Чтобы получить Socket.io, работающий с IISNode, единственные изменения, необходимые

при отсутствии виртуального каталога / вложенного приложения, находятся в `Web.config`.

Поскольку Socket.io отправляет запросы, начинающиеся с `/socket.io`, IISNode должен связываться с IIS, чтобы они также обрабатывались IISNode и не были только статическими файловыми запросами или другим трафиком. Для этого требуется другой `<handler>` чем стандартные приложения IISNode.

```
<handlers>
  <add name="iisnode-socketio" path="server.js" verb="*" modules="iisnode" />
</handlers>
```

В дополнение к изменениям в `<handlers>` нам также необходимо добавить дополнительное правило перезаписи URL. Правило `rewrite` отправляет весь трафик `/socket.io` в наш файл сервера, на котором запущен сервер Socket.io.

```
<rule name="SocketIO" patternSyntax="ECMAScript">
  <match url="socket.io.+"/>
  <action type="Rewrite" url="server.js"/>
</rule>
```

Если вы используете IIS 8, вам нужно отключить параметр `webSockets` в вашем `Web.config` в дополнение к добавлению вышеперечисленного и переписать правила. Это не нужно в IIS 7, так как нет поддержки `webSocket`.

```
<webSocket enabled="false" />
```

Прочитайте [Использование IISNode для размещения веб-приложений Node.js в IIS онлайн: https://riptutorial.com/ru/node-js/topic/6003/использование-iisnode-для-размещения-веб-приложений-node-js-в-iis](https://riptutorial.com/ru/node-js/topic/6003/использование-iisnode-для-размещения-веб-приложений-node-js-в-iis)

---

# глава 52: Использование WebSocket с Node.JS

## Examples

### Установка WebSocket

Есть несколько способов установить WebSocket для вашего проекта. Вот несколько примеров:

```
npm install --save ws
```

или внутри вашего пакета.json, используя:

```
"dependencies": {  
  "ws": "*"   
},
```

### Добавление WebSocket в файл

Чтобы добавить ws в ваш файл, просто используйте:

```
var ws = require('ws');
```

### Использование WebSocket и WebSocket Server

Чтобы открыть новый WebSocket, просто добавьте что-то вроде:

```
var WebSocket = require("ws");  
var ws = new WebSocket("ws://host:8080/OptionalPathName");  
// Continue on with your code...
```

Чтобы открыть сервер, используйте:

```
var WebSocketServer = require("ws").Server;  
var wss = new WebSocketServer({port: 8080, path: "OptionalPathName"});
```

### Пример простого сервера WebSocket

```
var WebSocketServer = require('ws').Server  
, wss = new WebSocketServer({ port: 8080 }); // If you want to add a path as well, use path:  
"PathName"  
  
wss.on('connection', function connection(ws) {
```



```
ws.on('message', function incoming(message) {
  console.log('received: %s', message);
});

ws.send('something');
});
```

Прочитайте [Использование WebSocket с Node.JS онлайн: https://riptutorial.com/ru/node-js/topic/6106/использование-websocket-c-node-js](https://riptutorial.com/ru/node-js/topic/6106/использование-websocket-c-node-js)

# глава 53: Использование потоков

## параметры

параметр	Определение
Считываемый поток	тип потока, где данные могут быть считаны из
Считываемый поток	тип потока, в котором данные могут быть записаны в
Дуплексный поток	тип потока, который доступен как для чтения, так и для записи
Преобразовать поток	тип дуплексного потока, который может преобразовывать данные по мере их чтения, а затем записывать

## Examples

### Чтение данных из TextFile с потоками

I/O в узле является асинхронным, поэтому взаимодействие с диском и сетью связано с передачей обратных вызовов к функциям. Возможно, у вас возникнет соблазн написать код, который обслуживает файл с диска следующим образом:

```
var http = require('http');
var fs = require('fs');

var server = http.createServer(function (req, res) {
  fs.readFile(__dirname + '/data.txt', function (err, data) {
    res.end(data);
  });
});
server.listen(8000);
```

Этот код работает, но он громоздкий и буферизирует весь файл data.txt в памяти для каждого запроса, прежде чем записывать результат обратно клиентам. Если data.txt очень большой, ваша программа может начать есть много памяти, поскольку она обслуживает множество пользователей одновременно, особенно для пользователей с медленными соединениями.

У пользователей плохой опыт, потому что пользователям придется ждать, пока весь файл будет буферизован в память на вашем сервере, прежде чем они смогут начать получать какое-либо содержимое.

К счастью, оба аргумента (`req`, `res`) - это потоки, что означает, что мы можем записать это гораздо лучше, используя `fs.createReadStream ()` вместо `fs.readFile ()`:

```
var http = require('http');
var fs = require('fs');

var server = http.createServer(function (req, res) {
  var stream = fs.createReadStream(__dirname + '/data.txt');
  stream.pipe(res);
});
server.listen(8000);
```

Здесь `.pipe ()` заботится о прослушивании событий «data» и «end» из `fs.createReadStream ()`. Этот код не только более чистый, но теперь файл `data.txt` будет записываться клиентам на один фрагмент одновременно сразу же после их получения с диска.

## Трубопроводы

Считываемые потоки могут быть «подключены по каналам» или подключены к записываемым потокам. Это делает поток данных из потока источника в поток назначения без особых усилий.

```
var fs = require('fs')

var readable = fs.createReadStream('file1.txt')
var writable = fs.createWriteStream('file2.txt')

readable.pipe(writable) // returns writable
```

Когда записываемые потоки также являются читаемыми потоками, т. Е. Когда они являются *дуплексными* потоками, вы можете продолжать передавать их другим записываемым потокам.

```
var zlib = require('zlib')

fs.createReadStream('style.css')
  .pipe(zlib.createGzip()) // The returned object, zlib.Gzip, is a duplex stream.
  .pipe(fs.createWriteStream('style.css.gz'))
```

Считываемые потоки также могут быть переданы в несколько потоков.

```
var readable = fs.createReadStream('source.css')
readable.pipe(zlib.createGzip()).pipe(fs.createWriteStream('output.css.gz'))
readable.pipe(fs.createWriteStream('output.css'))
```

Обратите внимание, что перед любыми потоками данных вы должны синхронно (в то же время) подключаться к выходным потокам. Несоблюдение этого требования может привести к потоку неполных данных.

Также обратите внимание, что объекты потока могут вызывать `error`; обязательно

ответственно обрабатывайте эти события на *каждом* потоке, если необходимо:

```
var readable = fs.createReadStream('file3.txt')
var writable = fs.createWriteStream('file4.txt')
readable.pipe(writable)
readable.on('error', console.error)
writable.on('error', console.error)
```

## Создание собственного читаемого / записываемого потока

Мы увидим, что объекты потока возвращаются модулями типа fs и т. Д., Но что, если мы хотим создать собственный поточный объект.

Чтобы создать объект Stream, нам нужно использовать модуль потока, предоставляемый NodeJs

```
var fs = require("fs");
var stream = require("stream").Writable;

/*
 * Implementing the write function in writable stream class.
 * This is the function which will be used when other stream is piped into this
 * writable stream.
 */
stream.prototype._write = function(chunk, data){
  console.log(data);
}

var customStream = new stream();

fs.createReadStream("aml.js").pipe(customStream);
```

Это даст нам собственный пользовательский поток, доступный для записи. мы можем реализовать что-либо в функции *\_write* . Выше метод работает в версии NodeJs 4.xx, но в NodeJs 6.x **ES6** введены классы, поэтому синтаксис изменился. Ниже приведен код для версии 6.x NodeJs

```
const Writable = require('stream').Writable;

class MyWritable extends Writable {
  constructor(options) {
    super(options);
  }

  _write(chunk, encoding, callback) {
    console.log(chunk);
  }
}
```

## Почему потоки?

Давайте рассмотрим следующие два примера для чтения содержимого файла:

Первый, который использует метод `asunc` для чтения файла и предоставляет функцию обратного вызова, которая вызывается после полного чтения файла в памяти:

```
fs.readFile(`${__dirname}/utils.js`, (err, data) => {
  if (err) {
    handleError(err);
  } else {
    console.log(data.toString());
  }
})
```

А вторая, которая использует `streams` для чтения содержимого файла, по частям:

```
var fileStream = fs.createReadStream(`${__dirname}/file`);
var fileContent = '';
fileStream.on('data', data => {
  fileContent += data.toString();
})

fileStream.on('end', () => {
  console.log(fileContent);
})

fileStream.on('error', err => {
  handleError(err)
})
```

Стоит отметить, что оба примера делают то **же самое** . В чем же тогда разница?

- Первый из них короче и выглядит более элегантно
- Второй позволяет вам обрабатывать файл **во время** его чтения (!)

Когда файлы, с которыми вы имеете дело, малы, тогда нет реального эффекта при использовании `streams` , но что происходит, когда файл большой? (настолько большой, что требуется 10 секунд, чтобы прочитать его в памяти)

Без `streams` вы будете ждать, ничего не делая абсолютно ничего (если только ваш процесс не делает других вещей), до тех пор, пока не пройдет 10 секунд, и файл не будет **полностью прочитан** , и только тогда вы сможете начать обработку файла.

С `streams` вы получаете содержимое файла по частям, **прямо, когда они доступны**, - и это позволяет обрабатывать файл **во время** его чтения.

---

В приведенном выше примере не показано, как `streams` могут использоваться для работы, которые невозможно выполнить при переходе на режим обратного вызова, поэтому рассмотрим другой пример:

Я хотел бы загрузить `gzip` файл, разархивировать его и сохранить его содержимое на диск. Учитывая `url` файла, это то, что нужно сделать:

- Загрузите файл
- Разархивируйте файл
- Сохранить на диске

Вот [маленький файл] [1], который хранится в моем хранилище s3 . Следующий код делает это в обратном вызове.

```
var startTime = Date.now()
s3.getObject({Bucket: 'some-bucket', Key: 'tweets.gz'}, (err, data) => {
  // here, the whole file was downloaded

  zlib.gunzip(data.Body, (err, data) => {
    // here, the whole file was unzipped

    fs.writeFile(`${__dirname}/tweets.json`, data, err => {
      if (err) console.error(err)

      // here, the whole file was written to disk
      var endTime = Date.now()
      console.log(`${endTime - startTime} milliseconds`) // 1339 milliseconds
    })
  })
})

// 1339 milliseconds
```

Вот как это выглядит, используя streams :

```
s3.getObject({Bucket: 'some-bucket', Key: 'tweets.gz'}).createReadStream()
  .pipe(zlib.createGunzip())
  .pipe(fs.createWriteStream(`${__dirname}/tweets.json`));

// 1204 milliseconds
```

Да, это не быстрее при работе с небольшими файлами - проверенный вес файлов 80КВ . Тестирование этого файла в более 71МВ файле, 71МВ gzipped ( 382МВ распаковкой), показывает, что версия streams намного быстрее

- Потребовалось 20925 миллисекунд, чтобы загрузить 71МВ , разархивировать его, а затем записать 382МВ на диск - **используя метод обратного вызова** .
- Для сравнения потребовалось 13434 миллисекунды, чтобы сделать то же самое при использовании версии streams (на 35% быстрее, для не очень большого файла)

Прочитайте [Использование потоков онлайн: https://riptutorial.com/ru/node-js/topic/2974/использование-потоков](https://riptutorial.com/ru/node-js/topic/2974/использование-потоков)

# глава 54: Использовать Случаи Node.js

## Examples

### HTTP-сервер

```
const http = require('http');

console.log('Starting server...');
var config = {
  port: 80,
  contentType: 'application/json; charset=utf-8'
};
// JSON-API server on port 80

var server = http.createServer();
server.listen(config.port);
server.on('error', (err) => {
  if (err.code === 'EADDRINUSE') console.error('Port ' + config.port + ' is already in use');
  else console.error(err.message);
});
server.on('request', (request, res) => {
  var remoteAddress = request.headers['x-forwarded-for'] ||
  request.connection.remoteAddress; // Client address
  console.log(remoteAddress + ' ' + request.method + ' ' + request.url);

  var out = {};
  // Here you can change output according to `request.url`
  out.test = request.url;
  res.writeHead(200, {
    'Content-Type': config.contentType
  });
  res.end(JSON.stringify(out));
});
server.on('listening', () => {
  c.info('Server is available: http://localhost:' + config.port);
});
```

### Консоль с командной строкой

```
const process = require('process');
const rl = require('readline').createInterface(process.stdin, process.stdout);

rl.pause();
console.log('Something long is happening here...');

var cliConfig = {
  promptPrefix: ' > '
}

/*
  Commands recognition
  BEGIN
*/
```

```

var commands = {
  eval: function(arg) { // Try typing in console: eval 2 * 10 ^ 3 + 2 ^ 4
    arg = arg.join(' ');
    try { console.log(eval(arg)); }
    catch (e) { console.log(e); }
  },
  exit: function(arg) {
    process.exit();
  }
};
rl.on('line', (str) => {
  rl.pause();
  var arg = str.trim().match(/([\^"]+)|("(?:[\^"\\\]|\\.)+")/g); // Applying regular expression
  for removing all spaces except for what between double quotes:
  http://stackoverflow.com/a/14540319/2396907
  if (arg) {
    for (let n in arg) {
      arg[n] = arg[n].replace(/^\\"|\\"$/g, '');
    }
    var commandName = arg[0];
    var command = commands[commandName];
    if (command) {
      arg.shift();
      command(arg);
    }
    else console.log('Command "' + commandName + '" doesn\'t exist');
  }
  rl.prompt();
});
/*
  END OF
  Commands recognition
*/

rl.setPrompt(cliConfig.promptPrefix);
rl.prompt();

```

Прочитайте [Использовать Случай Node.js онлайн: https://riptutorial.com/ru/node-js/topic/7703/использовать-случай-node-js](https://riptutorial.com/ru/node-js/topic/7703/использовать-случай-node-js)



---

# глава 55: История Nodejs

## Вступление

Здесь мы обсудим историю Node.js, информацию о версии и текущий статус.

## Examples

Ключевые события в каждом году

---

### 2009

- 3 марта . Проект был назван «узлом»,
- 1 октября: [первый очень ранний предварительный просмотр npm](#), пакет Node менеджер
- 8 ноября: [Ryan Dahl \(создатель Node.js\) Оригинальная дискуссия Node.js на JSConf 2009](#)

---

### 2010

- Экспресс: инфраструктура веб-разработки Node.js
- Начальный выпуск Socket.io
- 28 апреля: [Экспериментальная поддержка Node.js на Heroku](#)
- 28 июля: [Google Tech Talk от Ryan Dahl на Node.js](#)
- 20 августа: [выпущен Node.js 0.2.0](#)

---

### 2011

- 31 марта: руководство Node.js
- 1 мая: [npm 1.0: выпущено](#)
- 1 мая: [AMA Райана Даля на Reddit](#)
- 10 июля: [завершена Книга Нода Новичка, введение в Node.js.](#)
  - Полный учебник Node.js для начинающих.
- 16 августа: [LinkedIn использует Node.js](#)
  - LinkedIn запустила полностью переработанное мобильное приложение с новыми функциями и новыми деталями под капотом.
- 5 октября: [Райан Дал рассказывает об истории Node.js и почему он ее создал](#)
- 5 декабря: [Node.js в производстве в Uber](#)

- Инженер-программист Uber Curtis Chambers объясняет, почему его компания полностью переработала свое приложение с помощью Node.js для повышения эффективности и улучшения взаимодействия с партнерами и клиентами.

---

## 2012

- 30 января: [Создатель Node.js Райан Дал отходит от ежедневного Узла](#)
- 25 июня: [Node.js v0.8.0 \[stable\] отсутствует](#)
- 20 декабря: [Napi, рама Node.js](#) выпущена

---

## 2013

- 30 апреля: [MEAN Stack: MongoDB, ExpressJS, AngularJS и Node.js](#)
- 17 мая: [Как мы создали первое приложение Node.js eBay](#)
- 15 ноября: [PayPal выпускает Kraken, инфраструктуру Node.js](#)
- 22 ноября: [Утечка памяти Node.js в Walmart](#)
  - Лаборатории Eran Hammer из Wal-Mart пришли к основной команде Node.js, жалуясь на утечку памяти, которую он отслеживал в течение нескольких месяцев.
- 19 декабря: [Коа - веб-среда для Node.js](#)

---

## 2014

- 15 января: [ТД Фонтен принимает проект Node](#)
- 23 октября: [Консультативный совет Node.js](#)
  - Joyent и несколько членов сообщества Node.js объявили о предложении для Консультативного совета Node.js в качестве следующего шага к полностью открытой модели управления проектом Node.js с открытым исходным кодом.
- 19 ноября: [Node.js в пламенных графиках - Netflix](#)
- 28 ноября: [IO.js - Evented I / O для V8 Javascript](#)

---

## 2015

### Q1

- 14 января: [IO.js 1.0.0](#)
- 10 февраля: [Joyent движется к созданию Node.js Foundation](#)
  - Joyent, IBM, Microsoft, PayPal, Fidelity, SAP и Linux Foundation объединяют усилия для поддержки сообщества Node.js с нейтральным и открытым управлением

- [Предложение 27th Febraury : IO.js и Node.js](#)

## Q2

- 14 апреля: [npm Частные модули](#)
- 28 мая: [Узел ведущего TJ Фонтен уходит и оставляет Joyent](#)
- 13 мая: [Node.js и io.js сливаются под Фондом Узла](#)

## Q3

- 2 августа: [отслеживание и отладка отслеживания производительности Trace - Node.js](#)
  - Trace - это визуализированный инструмент мониторинга микросервиса, который дает вам все показатели, необходимые для работы с микросервисами.
- 13 августа: [4.0 - новый 1.0](#)

## Q4

- 12 октября: [Node v4.2.0, первая версия Long Term Support](#)
- 8 декабря: [Apigee, RisingStack и Yahoo присоединяются к Node.js Foundation](#)
- 8 и 9 декабря: [Узел Интерактивный](#)
  - Первая ежегодная конференция Node.js от Node.js Foundation

---

# 2016

## Q1

- 10 февраля: [Экспресс становится инкубированным проектом](#)
- 23 марта: [инцидент с левым](#)
- 29 марта: [Google Cloud Platform присоединяется к Node.js Foundation](#)

## Q2

- 26 апреля: [npm имеет 210 000 пользователей](#)

## Q3

- 18 июля: [CJ Silverio становится техническим директором npm](#)
- 1 августа: [Trace, решение отладки Node.js становится общедоступным](#)
- 15 сентября: [первый Node Interactive в Европе](#)

## Q4

- 11 октября: [выпущен менеджер пакетов пряжи](#)
- 18 октября: [Node.js 6 становится версией LTS](#)

### *Ссылка*

1. «История Node.js на временной шкале» [Online]. Доступно: [<https://blog.risingstack.com/history-of-node-js>]

Прочитайте [История Nodejs онлайн](https://riptutorial.com/ru/node-js/topic/8653/история-nodejs): <https://riptutorial.com/ru/node-js/topic/8653/история-nodejs>

---

# глава 56: Как загружаются модули

## Examples

### Глобальный режим

Если вы установили узел с использованием каталога по умолчанию, а в глобальном режиме, NPM устанавливает пакеты в `/usr/local/lib/node_modules`. Если вы введете следующее в оболочке, NPM будет искать, загружать и устанавливать последнюю версию пакета с именем `saх` внутри каталога `/usr/local/lib/node_modules/express`:

```
$ npm install -g express
```

Убедитесь, что у вас есть достаточные права доступа к папке. Эти модули будут доступны для всех процессов узла, которые будут работать на этом компьютере

В локальном режиме установки. Npm загрузит и установит модули в текущие рабочие папки, создав новую папку `node_modules` например, если вы находитесь в `/home/user/apps/my_app` создана новая папка, называемая `node_modules` `/home/user/apps/my_app/node_modules` если его еще не существует

### Загрузочные модули

Когда мы ссылаемся на модуль в коде, узел сначала ищет папку `node_module` внутри указанной папки в требуемом заявлении. Если имя модуля не является относительным и не является основным модулем, Node будет пытаться найти его внутри папки `node_modules` в текущем каталог. Например, если вы выполните следующее, Node попытается найти файл `./node_modules/myModule.js`:

```
var myModule = require('myModule.js');
```

Если Node не сможет найти файл, он будет выглядеть внутри родительской папки с именем `../node_modules/myModule.js`. Если он не сработает снова, он попробует родительскую папку и продолжит спускаться до тех пор, пока не достигнет корня или не найдет необходимый модуль.

Вы также можете опустить расширение `.js` если хотите, и в этом случае узел добавит расширение `.js` и будет искать файл.

---

## Загрузка модуля папок

Вы можете использовать путь к папке для загрузки модуля следующим образом:

```
var myModule = require('./myModuleDir');
```

Если вы это сделаете, узел будет искать внутри этой папки. Узел предположит, что эта папка представляет собой пакет и попытается найти определение пакета. Это определение пакета должно быть файлом с именем `package.json`. Если в этой папке не содержится файл определения пакета с именем `package.json`, точка ввода пакета будет принимать значение по умолчанию `index.js`, а в этом случае Node будет искать файл по пути `./myModuleDir/index.js`.

Последнее, если модуль не найден ни в одной из папок, это папка установки глобального модуля.

Прочитайте Как загружаются модули онлайн: <https://riptutorial.com/ru/node-js/topic/7738/как-загружаются-модули>

---

# глава 57: Кластерный модуль

## Синтаксис

- `const cluster = require("cluster")`
- `cluster.fork()`
- `cluster.isMaster`
- `cluster.isWorker`
- `cluster.schedulingPolicy`
- `cluster.setupMaster` (настройки)
- `cluster.settings`
- `cluster.worker` // у рабочего
- `cluster.workers` // в master

## замечания

Обратите внимание, что `cluster.fork()` дочерний процесс, который начинает выполнять текущий скрипт с самого начала, в отличие от системного вызова `fork()` в C, который клонирует текущий процесс и продолжается из инструкции после системного вызова как в родительском, так и в дочерний процесс.

Документация Node.js содержит более полное руководство по кластерам [здесь](#)

## Examples

### Привет, мир

Это ваш `cluster.js`:

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  // Fork workers.
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('exit', (worker, code, signal) => {
    console.log(`worker ${worker.process.pid} died`);
  });
} else {
  // Workers can share any TCP connection
  // In this case it is an HTTP server
  require('./server.js')();
}
```

Это ваш главный `server.js` :

```
const http = require('http');

function startServer() {
  const server = http.createServer((req, res) => {
    res.writeHead(200);
    res.end('Hello Http');
  });

  server.listen(3000);
}

if(!module.parent) {
  // Start server if file is run directly
  startServer();
} else {
  // Export server, if file is referenced via cluster
  module.exports = startServer;
}
```

В этом примере мы размещаем базовый веб-сервер, однако мы разворачиваем рабочих (дочерние процессы) с помощью встроенного **кластерного** модуля. Количество обработчиков процессов зависит от количества доступных ядер процессора. Это позволяет приложению Node.js использовать многоядерные процессоры, поскольку один экземпляр Node.js работает в одном потоке. Теперь приложение будет передавать порт 8000 во все процессы. Погрузки автоматически распределяются между работниками, используя метод Round-Robin по умолчанию.

## Пример кластера

Один экземпляр `Node.js` работает в одном потоке. Чтобы использовать многоядерные системы, приложение может быть запущено в кластере процессов Node.js для обработки нагрузки.

Модуль `cluster` позволяет вам легко создавать дочерние процессы, которые используют общий доступ к портам сервера.

В следующем примере создайте дочерний процесс `worker` в основном процессе, который обрабатывает нагрузку через несколько ядер.

### пример

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length; //number of CPUs

if (cluster.isMaster) {
  // Fork workers.
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork(); //creating child process
  }
}
```



```
//on exit of cluster
cluster.on('exit', (worker, code, signal) => {
  if (signal) {
    console.log(`worker was killed by signal: ${signal}`);
  } else if (code !== 0) {
    console.log(`worker exited with error code: ${code}`);
  } else {
    console.log('worker success!');
  }
});
} else {
  // Workers can share any TCP connection
  // In this case it is an HTTP server
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end('hello world\n');
  }).listen(3000);
}
```

Прочитайте Кластерный модуль онлайн: <https://riptutorial.com/ru/node-js/topic/2817/кластерный-модуль>

---

# глава 58: Код Node.js для STDIN и STDOUT без использования какой-либо библиотеки

## Вступление

Это простая программа в node.js, которая принимает данные от пользователя и печатает его на консоли.

Объект **процесса** является глобальным, который предоставляет информацию о текущем процессе Node.js и контролирует его. Как глобальный, он всегда доступен для приложений Node.js без использования require ().

## Examples

### программа

Свойство **process.stdin** возвращает Readable поток, эквивалентный или связанный с stdin.

Свойство **process.stdout** возвращает поток Writable, эквивалентный или связанный с stdout.

```
process.stdin.resume()
console.log('Enter the data to be displayed ');
process.stdin.on('data', function(data) { process.stdout.write(data) })
```

Прочитайте Код Node.js для STDIN и STDOUT без использования какой-либо библиотеки онлайн: <https://riptutorial.com/ru/node-js/topic/8961/код-node-js-для-stdin-и-stdout-без-использования-какой-либо-библиотеки>

# глава 59: Локализация узла JS

## Вступление

Его очень легко поддерживать локализацию nodejs express

## Examples

### с использованием модуля i18n для поддержки локализации в приложении node js

Легкий простой модуль перевода с динамическим хранилищем json. Поддерживает простые приложения vanilla node.js и должен работать с любыми фреймворками (например, выразить, рестифицировать и, возможно, больше), предоставляя метод app.use (), передающий объекты res и req. Использует общий синтаксис \_\_ ('...') в приложении и шаблонах. Сохраняет языковые файлы в json-файлах, совместимых с форматом webtranslateit json. Добавляет новые строки «на лету», когда они впервые используются в вашем приложении. Никакого дополнительного анализа не требуется.

express + i18n-node + cookieParser и избегать проблем с параллелизмом

```
// usual requirements
var express = require('express'),
    i18n = require('i18n'),
    app = module.exports = express();

i18n.configure({
  // setup some locales - other locales default to en silently
  locales: ['en', 'ru', 'de'],

  // sets a custom cookie name to parse locale settings from
  cookie: 'yourcookienamе',

  // where to store json files - defaults to './locales'
  directory: __dirname + '/locales'
});

app.configure(function () {
  // you will need to use cookieParser to expose cookies to req.cookies
  app.use(express.cookieParser());

  // i18n init parses req for language headers, cookies, etc.
  app.use(i18n.init);
});

// serving homepage
app.get('/', function (req, res) {
  res.send(res.__('Hello World'));
});
```

```
// starting server
if (!module.parent) {
  app.listen(3000);
}
```

Прочитайте Локализация узла JS онлайн: <https://riptutorial.com/ru/node-js/topic/9594/локализация-узла-js>

# глава 60: Маршрутизация ајах-запросов с помощью Express.JS

## Examples

### Простая реализация AJAX

У вас должен быть базовый шаблон экспресс-генератора

В `app.js` добавьте (вы можете добавить его куда угодно после `var app = express.app()`):

```
app.post(function(req, res, next){
  next();
});
```

Теперь в файле `index.js` (или соответствующем совпадении) добавьте:

```
router.get('/ajax', function(req, res){
  res.render('ajax', {title: 'An Ajax Example', quote: "AJAX is great!"});
});
router.post('/ajax', function(req, res){
  res.render('ajax', {title: 'An Ajax Example', quote: req.body.quote});
});
```

Создайте `ajax.jade` / `ajax.pug` или `ajax.ejs` каталоге `/views`, добавьте:

Для Jade / PugJS:

```
extends layout
script(src="http://code.jquery.com/jquery-3.1.0.min.js")
script(src="/magic.js")
h1 Quote: !{quote}
form(method="post" id="changeQuote")
  input(type="text", placeholder="Set quote of the day", name="quote")
  input(type="submit", value="Save")
```

Для EJS:

```
<script src="http://code.jquery.com/jquery-3.1.0.min.js"></script>
<script src="/magic.js"></script>
<h1>Quote: <%=quote%> </h1>
<form method="post" id="changeQuote">
  <input type="text" placeholder="Set quote of the day" name="quote"/>
  <input type="submit" value="Save">
</form>
```

Теперь создайте файл в `/public` названный `magic.js`

```
$(document).ready(function(){
  $("#form#changeQuote").on('submit', function(e){
    e.preventDefault();
    var data = $('input[name=quote]').val();
    $.ajax({
      type: 'post',
      url: '/ajax',
      data: data,
      dataType: 'text'
    })
    .done(function(data){
      $('h1').html(data.quote);
    });
  });
});
```

И вот оно! Когда вы нажмете «Сохранить цитату», она изменится!

Прочитайте [Маршрутизация ajax-запросов с помощью Express.JS онлайн:](https://riptutorial.com/ru/node-js/topic/6738/маршрутизация-ajax-запросов-с-помощью-express-js)

<https://riptutorial.com/ru/node-js/topic/6738/маршрутизация-ajax-запросов-с-помощью-express-js>

---

# глава 61: Маршрутизация NodeJs

## Вступление

Как настроить базовый веб-сервер Express под узлом js и исследовать маршрутизатор Express.

## замечания

Наконец, с помощью Express Router вы можете использовать средство маршрутизации в своем приложении и его легко реализовать.

## Examples

### Экспресс-трафик веб-сервера

#### Создание Express Web-сервера

Экспресс-сервер оказался удобным, и он проникает во многие пользователи и сообщества. Он становится популярным.

Позволяет создать Express Server. Для управления пакетами и гибкости для зависимостей Мы будем использовать NPM (диспетчер пакетов узлов).

1. Перейдите в каталог Project и создайте файл package.json. **package.json** {"name": "expressRouter", "version": "0.0.1", "scripts": {"start": "node Server.js"}, "dependencies": {"express": "^ 4.12.3 "}}
2. Сохраните файл и установите экспресс-зависимость, используя следующую команду `npm install` . Это создаст node\_modules в каталоге проекта вместе с необходимой зависимостью.
3. Давайте создадим Express Web Server. Перейдите в каталог проекта и создайте файл server.js. **server.js**

```
var express = require ("express"); var app = express ();
```

```
// Создание объекта Router ()
```

```
var router = express.Router ();
```

```
// Предоставляем все маршруты здесь, это для Домашней страницы.
```

```
router.get ("/", function (req, res) {
```

```
res.json({"message" : "Hello World"});
```

```
});
```

```
app.use ( "/ API", маршрутизатор);
```

```
// Прослушать этот порт
```

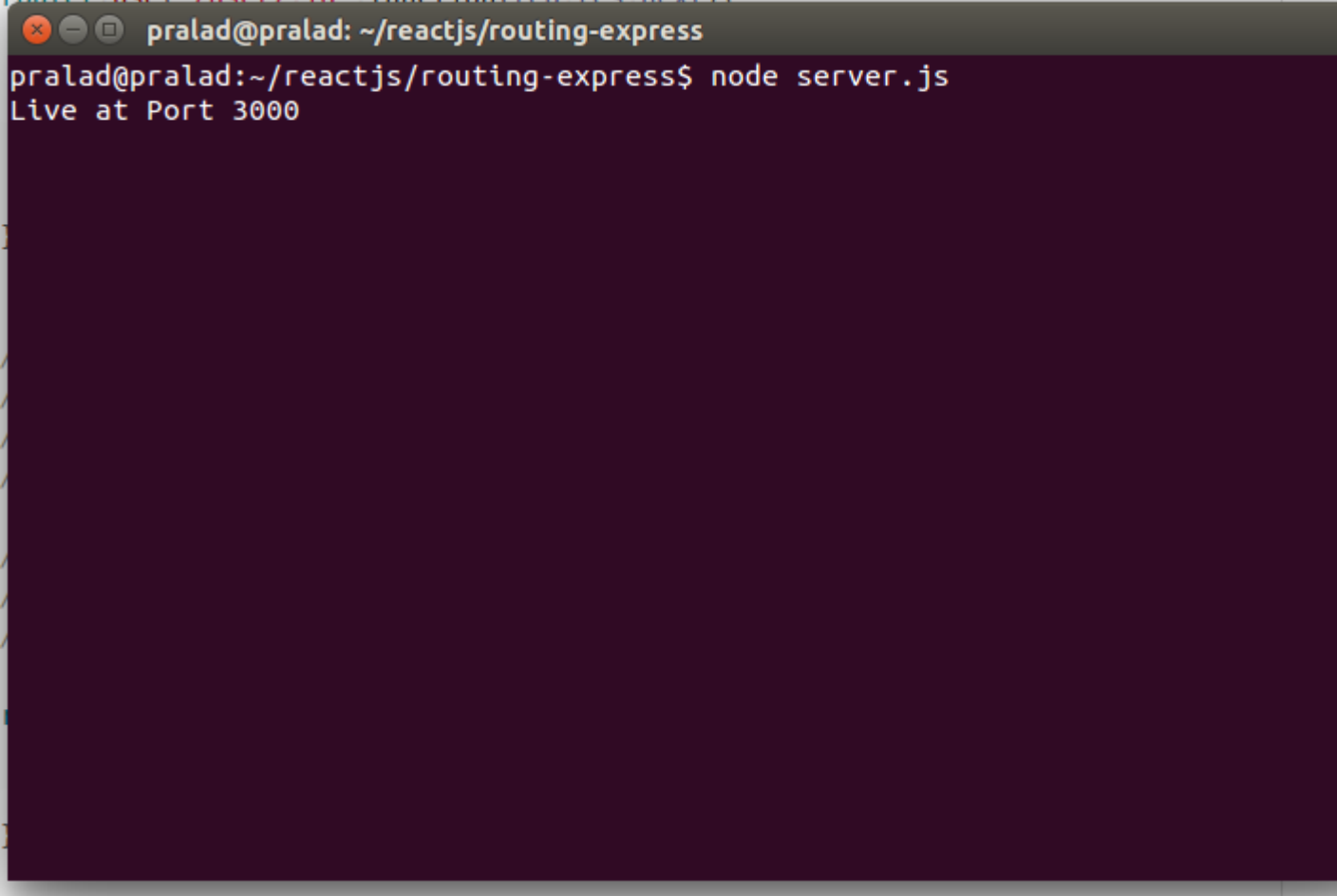
```
app.listen (3000, function () {console.log («Live at Port 3000»)});
```

For more detail on setting node server you can see [\[here\]](#)[1].

4. Запустите сервер, набрав следующую команду.

```
node server.js
```

Если сервер работает успешно, вы увидите что-то вроде этого.



```
pralad@pralad: ~/reactjs/routing-express  
pralad@pralad:~/reactjs/routing-express$ node server.js  
Live at Port 3000
```

5. Теперь перейдите в браузер или почтальон и сделайте запрос

[HTTP: // локальный: 3000 / API /](http://localhost:3000/API/)

Выход будет





Вот и все, основные из Express Routing.

Теперь давайте обработаем GET, POST и т. Д.

Измените файл `server.js` yours

```
var express = require("express");
var app = express();

//Creating Router() object

var router = express.Router();

// Router middleware, mentioned it before defining routes.

router.use(function(req, res, next) {
  console.log("/" + req.method);
  next();
});

// Provide all routes here, this is for Home page.

router.get("/", function(req, res) {
  res.json({ "message" : "Hello World" });
});

app.use("/api", router);

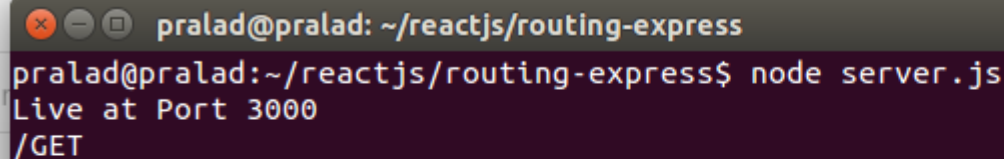
app.listen(3000, function() {
  console.log("Live at Port 3000");
});
```

```
});
```

Теперь, если вы перезапустите сервер и сделаете запрос

```
http://localhost:3000/api/
```

Вы увидите что-то вроде



```
pralad@pralad: ~/reactjs/routing-express
pralad@pralad:~/reactjs/routing-express$ node server.js
Live at Port 3000
/GET
```

### ***Доступ к параметру в маршрутизации***

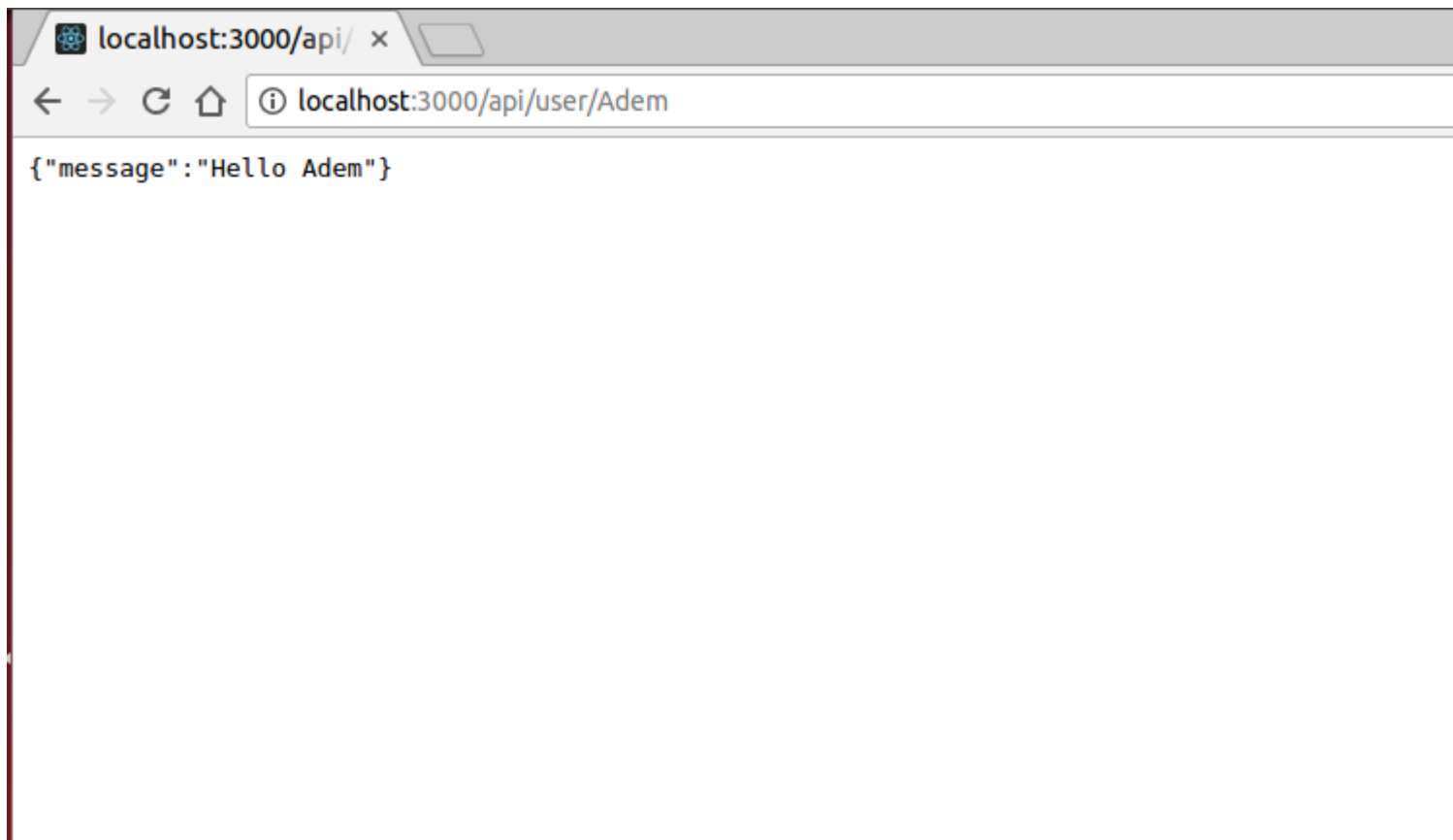
Вы также можете получить доступ к параметру из url, например

<http://example.com/api/:name/> . Таким образом, параметром имени может быть доступ.

Добавьте следующий код в свой сервер server.js

```
router.get("/user/:id", function(req, res) {
  res.json({"message" : "Hello "+req.params.id});
});
```

Теперь перезагрузите сервер и перейдите к [ <http://localhost:3000/api/user/Adem>] [4] ,  
выход будет похож на



Прочитайте Маршрутизация NodeJs онлайн: <https://riptutorial.com/ru/node-js/topic/9846/маршрутизация-nodejs>

---

# глава 62: Менеджер пакетов пряжи

## Вступление

[Пряжа](#) - это менеджер пакетов для Node.js, аналогичный npm. Несмотря на то, что они разделяют много общего, есть несколько ключевых различий между пряжей и npm.

## Examples

### Установка пряжи

В этом примере объясняются различные методы установки пряжи для вашей ОС.

---

## Macos

### Homebrew

```
brew update  
brew install yarn
```

### MacPorts

```
sudo port install yarn
```

## Добавление пряжи в вашу PATH

Добавьте в свой предпочтительный профиль оболочки ( `.profile` , `.bashrc` , `.zshrc` т. Д.).

```
export PATH="$PATH:`yarn global bin`"
```

---

## Windows

### МОНТАЖНИК

Во-первых, установите Node.js, если он еще не установлен.

Загрузите программу установки пряжи в `.msi` с [сайта Yarn](#) .

## ШОКОЛАДНЫМ

```
choco install yarn
```

# Linux

## Debian / Ubuntu

Убедитесь, что Node.js установлен для вашего дистрибутива или выполните следующие

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -  
sudo apt-get install -y nodejs
```

### Настройка репозитория YarnPkg

```
curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | sudo apt-key add -  
echo "deb https://dl.yarnpkg.com/debian/ stable main" | sudo tee  
/etc/apt/sources.list.d/yarn.list
```

### Установить пряжу

```
sudo apt-get update && sudo apt-get install yarn
```

## CentOS / Fedora / RHEL

Установите Node.js, если он еще не установлен.

```
curl --silent --location https://rpm.nodesource.com/setup_6.x | bash -
```

### Установить пряжу

```
sudo wget https://dl.yarnpkg.com/rpm/yarn.repo -O /etc/yum.repos.d/yarn.repo  
sudo yum install yarn
```

## арочный

Установите пряжу через AUR.

Пример использования yaourt:

```
yaourt -S yarn
```

## В единственном числе

```
sudo eopkg install yarn
```

## Все дистрибутивы

Добавьте в свой предпочтительный профиль оболочки ( `.profile` , `.bashrc` , `.zshrc` т. Д.).

```
export PATH="$PATH:`yarn global bin`"
```

---

## Альтернативный способ установки

### Сценарий оболочки

```
curl -o- -L https://yarnpkg.com/install.sh | bash
```

или указать версию для установки

```
curl -o- -L https://yarnpkg.com/install.sh | bash -s -- --version [version]
```

### тарболл

```
cd /opt
wget https://yarnpkg.com/latest.tar.gz
tar zvxf latest.tar.gz
```

## НПМ

Если у вас уже установлен npm, просто запустите

```
npm install -g yarn
```

---

## Post Install

Проверьте установленную версию пряжи, выполнив

```
yarn --version
```

### Создание базового пакета

Команда `yarn init` проведет вас через создание файла `package.json` чтобы настроить некоторую информацию о вашем пакете. Это похоже на команду `npm init` в `npm`.

Создайте и перейдите к новому каталогу для хранения вашего пакета, а затем запустите `yarn init`

```
mkdir my-package && cd my-package
yarn init
```

Ответьте на вопросы, которые следуют в CLI

```
question name (my-package): my-package
question version (1.0.0):
question description: A test package
question entry point (index.js):
question repository url:
question author: StackOverflow Documentation
question license (MIT):
success Saved package.json
[] Done in 27.31s.
```

Это создаст файл `package.json` подобный следующему

```
{
  "name": "my-package",
  "version": "1.0.0",
  "description": "A test package",
  "main": "index.js",
  "author": "StackOverflow Documentation",
  "license": "MIT"
}
```

Теперь попробуйте добавить зависимость. Основной синтаксис для этого - `yarn add [package-name]`

Для установки ExpressJS выполните следующие действия:

```
yarn add express
```

Это добавит раздел `dependencies` к вашему `package.json` и добавит ExpressJS

```
"dependencies": {
  "express": "^4.15.2"
}
```

## Установить пакет с пряжей

Пряжа использует тот же реестр, что и `npm`. Это означает, что каждый пакет, доступный на `npm`, одинаковый для пряжи.

Чтобы установить пакет, запустите `yarn add package`.

Если вам нужна определенная версия пакета, вы можете использовать `yarn add package@version`.

Если версия, которую вам нужно установить, была помечена, вы можете использовать `yarn add package@tag`.

Прочитайте Менеджер пакетов пряжи онлайн: <https://riptutorial.com/ru/node-js/topic/9441/менеджер-пакетов-пряжи>



---

# глава 63: Многопоточность

## Вступление

Node.js был спроектирован как однопоточный. Поэтому для всех практических целей приложения, запускаемые с помощью Node, будут работать в одном потоке.

Однако сам Node.js запускается многопоточно. Операции ввода-вывода и т.п. будут выполняться из пула потоков. Кроме того, любой экземпляр приложения-узла будет работать в другом потоке, поэтому для запуска многопоточных приложений запускается несколько экземпляров.

## замечания

Понимание цикла [событий](#) важно понять, как и почему использовать несколько потоков.

## Examples

### кластер

Модуль `cluster` позволяет запускать одно и то же приложение несколько раз.

Кластеризация желательна, когда разные экземпляры имеют один и тот же поток выполнения и не зависят друг от друга. В этом случае у вас есть один мастер, который может запускать вилки и вилки (или дети). Дети работают самостоятельно и имеют свое пространство в Раме и Event Loop.

Настройка кластеров может быть полезной для веб-сайтов / API. Любой поток может обслуживать любого клиента, так как он не зависит от других потоков. База данных (например, Redis) будет использоваться для совместного использования Cookies, поскольку **переменные не могут использоваться совместно!** между нитями.

```
// runs in each instance
var cluster = require('cluster');
var numCPUs = require('os').cpus().length;

console.log('I am always called');

if (cluster.isMaster) {
  // runs only once (within the master);
  console.log('I am the master, launching workers!');
  for(var i = 0; i < numCPUs; i++) cluster.fork();
} else {
  // runs in each fork
  console.log('I am a fork!');
```

```
// here one could start, as an example, a web server  
  
}  
  
console.log('I am always called as well');
```

## Детский процесс

Процессы Child - это путь, когда вы хотите запускать процессы независимо друг от друга с инициализацией и проблемами. Подобно forks в кластерах, `child_process` работает в своем потоке, но, в отличие от forks, у него есть способ связаться со своим родителем.

Связь идет в обоих направлениях, поэтому родитель и ребенок могут прослушивать сообщения и отправлять сообщения.

### Родитель (../parent.js)

```
var child_process = require('child_process');  
console.log('[Parent]', 'inititalize');  
  
var child1 = child_process.fork(__dirname + '/child');  
child1.on('message', function(msg) {  
    console.log('[Parent]', 'Answer from child: ', msg);  
});  
  
// one can send as many messages as one want  
child1.send('Hello'); // Hello to you too :)  
child1.send('Hello'); // Hello to you too :)  
  
// one can also have multiple children  
var child2 = child_process.fork(__dirname + '/child');
```

### Ребенок (../child.js)

```
// here would one initialize this child  
// this will be executed only once  
console.log('[Child]', 'inititalize');  
  
// here one listens for new tasks from the parent  
process.on('message', function(messageFromParent) {  
  
    //do some intense work here  
    console.log('[Child]', 'Child doing some intense work');  
  
    if(messageFromParent == 'Hello') process.send('Hello to you too :)');  
    else process.send('what?');  
  
})
```

Рядом с сообщением можно прослушивать **многие события**, такие как «ошибка», «подключено» или «отключиться».

Запуск дочернего процесса имеет определенную стоимость, связанную с ним. Можно было

бы создать как можно меньше из них.

Прочитайте Многопоточность онлайн: <https://riptutorial.com/ru/node-js/topic/10592/>  
МНОГОПОТОЧНОСТЬ

# глава 64: Монгодская интеграция

## Синтаксис

- `дб. collection.insertOne ( document , options (w, wtimeout, j, serializeFuntions, forceServerObjectId, bypassDocumentValidation) , обратный вызов )`
- `дб. коллекция.insertMany ( [документы] , параметры (w, wtimeout, j, serializeFuntions, forceServerObjectId, bypassDocumentValidation) , обратный вызов )`
- `дб. коллекция.find ( запрос )`
- `дб. коллекция.updateOne ( фильтр , обновление , опции (upsert, w, wtimeout, j, bypassDocumentValidation) , обратный вызов )`
- `дб. коллекция.updateMany ( фильтр , обновление , опции (upsert, w, wtimeout, j) , обратный вызов )`
- `дб. коллекция.deleteOne ( фильтр , опции (upsert, w, wtimeout, j) , обратный вызов )`
- `дб. коллекция.deleteMany ( фильтр , опции (upsert, w, wtimeout, j) , обратный вызов )`

## параметры

параметр	подробности
документ	Объект javascript, представляющий документ
документы	Массив документов
запрос	Объект, определяющий поисковый запрос
фильтр	Объект, определяющий поисковый запрос
Перезвоните	Функция, вызываемая при выполнении операции
опции	(необязательно) Дополнительные настройки (по умолчанию: null)
вес	(необязательно)
wtimeout	(необязательно) Тайм-аут ожидания записи. (по умолчанию: null)
J	(необязательно) Укажите проблему записи журнала (по умолчанию: false)
upsert	(необязательно) Операция обновления (по умолчанию: false)

параметр	подробности
МНОГО	(необязательно) Обновление одного / всех документов (по умолчанию: <i>false</i> )
serializeFunctions	(необязательно) Сериализовать функции на любом объекте (по умолчанию: <i>false</i> )
forceServerObjectId	(необязательно) Принудительный сервер для назначения значений <code>_id</code> вместо драйвера (по умолчанию: <i>false</i> )
bypassDocumentValidation	(необязательно) Разрешить драйверу обходить проверку схемы в MongoDB 3.2 или выше (по умолчанию: <i>false</i> )

## Examples

### Подключиться к MongoDB

Подключитесь к MongoDB, напечатайте 'Connected!' и закройте соединение.

```
const MongoClient = require('mongodb').MongoClient;

var url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function(err, db) { // MongoClient method 'connect'
  if (err) throw new Error(err);
  console.log("Connected!");
  db.close(); // Don't forget to close the connection when you are done
});
```

### Метод MongoClient `connect()`

`MongoClient.connect ( url , options , callback )`

аргументация	Тип	Описание
url	строка	Строка, указывающая сервер ip / hostname, порт и база данных
options	объект	(необязательно) Дополнительные настройки (по умолчанию: <i>null</i> )
callback	функция	Функция, вызываемая при попытке подключения

Функция `callback` принимает два аргумента

- `err` : Ошибка - при возникновении ошибки аргумент `err` будет определен
- `db` : объект - экземпляр MongoDB

## Вставить документ

Вставьте документ под названием «myFirstDocument» и установите 2 свойства, `greetings` и `farewell`

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  db.collection('myCollection').insertOne({ // Insert method 'insertOne'
    "myFirstDocument": {
      "greetings": "Hellu",
      "farewell": "Bye"
    }
  }, function (err, result) {
    if (err) throw new Error(err);
    console.log("Inserted a document into the myCollection collection!");
    db.close(); // Don't forget to close the connection when you are done
  });
});
```

## Метод `insertOne()`

`db.collection ( collection ) .insertOne ( документ , параметры , обратный вызов )`

аргументация	Тип	Описание
<code>collection</code>	строка	Строка, указывающая коллекцию
<code>document</code>	объект	Документ, который необходимо вставить в сборник
<code>options</code>	объект	(необязательно) Дополнительные настройки (по умолчанию: <code>null</code> )
<code>callback</code>	функция	Функция, вызываемая при выполнении операции вставки

Функция `callback` принимает два аргумента

- `err` : Ошибка - при возникновении ошибки аргумент `err` будет определен
- `result` : объект - объект, содержащий сведения о операции вставки

## Читать коллекцию

Получите все документы в коллекции «myCollection» и распечатайте их на консоли.

```

const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  var cursor = db.collection('myCollection').find(); // Read method 'find'
  cursor.each(function (err, doc) {
    if (err) throw new Error(err);
    if (doc != null) {
      console.log(doc); // Print all documents
    } else {
      db.close(); // Don't forget to close the connection when you are done
    }
  });
});

```

## Метод `find()`

`db.collection ( collection ) .find ( )`

аргументация	Тип	Описание
<code>collection</code>	строка	Строка, указывающая коллекцию

## Обновить документ

Найдите документ с свойством `{ greetings: 'Hellu' }` и измените его на `{ greetings: 'Whut?' }`

```

const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  db.collection('myCollection').updateOne({ // Update method 'updateOne'
    greetings: "Hellu" },
    { $set: { greetings: "Whut?" } },
    function (err, result) {
      if (err) throw new Error(err);
      db.close(); // Don't forget to close the connection when you are done
    });
});

```

## Метод `updateOne()`

`db.collection ( collection ) .updateOne ( фильтр , обновление , опции . обратный вызов )`

параметр	Тип	Описание
filter	объект	Определяет критерий выбора
update	объект	Указывает, какие изменения применяются
options	объект	<i>(необязательно)</i> Дополнительные настройки <i>(по умолчанию: null)</i>
callback	функция	Функция, вызываемая при выполнении операции

Функция `callback` принимает два аргумента

- `err` : Ошибка - при возникновении ошибки аргумент `err` будет определен
- `db` : объект - экземпляр MongoDB

## Удаление документа

Удалить документ с помощью свойства `{ greetings: 'Whut?' }`

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  db.collection('myCollection').deleteOne(// Delete method 'deleteOne'
    { greetings: "Whut?" },
    function (err, result) {
      if (err) throw new Error(err);
      db.close(); // Don't forget to close the connection when you are done
    });
});
```

## Метод `deleteOne()`

`db.collection ( collection ) .deleteOne ( фильтр , параметры , обратный вызов )`

параметр	Тип	Описание
filter	объект	Документ, определяющий критерий выбора
options	объект	<i>(необязательно)</i> Дополнительные настройки <i>(по умолчанию: null)</i>
callback	функция	Функция, вызываемая при выполнении операции

Функция `callback` принимает два аргумента



- `err` : Ошибка - при возникновении ошибки аргумент `err` будет определен
- `db` : объект - экземпляр MongoDB

## Удаление нескольких документов

Удалите ВСЕ документы с «прощальным» свойством, установленным в «okay».

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  db.collection('myCollection').deleteMany(// MongoDB delete method 'deleteMany'
    { farewell: "okay" }, // Delete ALL documents with the property 'farewell: okay'
    function (err, result) {
      if (err) throw new Error(err);
      db.close(); // Don't forget to close the connection when you are done
    });
});
```

## Метод коллекции `deleteMany()`

`db.collection ( collection ) .deleteMany ( фильтр , параметры , обратный вызов )`

параметр	Тип	Описание
<code>filter</code>	документ	Документ, определяющий критерий выбора
<code>options</code>	объект	(необязательно) Дополнительные настройки (по умолчанию: <i>null</i> )
<code>callback</code>	функция	Функция, вызываемая при выполнении операции

Функция `callback` принимает два аргумента

- `err` : Ошибка - при возникновении ошибки аргумент `err` будет определен
- `db` : объект - экземпляр MongoDB

## Простое подключение

```
MongoDB.connect('mongodb://localhost:27017/databaseName', function(error, database) {
  if(error) return console.log(error);
  const collection = database.collection('collectionName');
  collection.insert({key: 'value'}, function(error, result) {
    console.log(error, result);
  });
});
```

## Простое соединение, используя обещания

```
const MongoDB = require('mongodb');

MongoDB.connect('mongodb://localhost:27017/databaseName')
  .then(function(database) {
    const collection = database.collection('collectionName');
    return collection.insert({key: 'value'});
  })
  .then(function(result) {
    console.log(result);
  });
` ``
```

Прочитайте Монгодская интеграция онлайн: <https://riptutorial.com/ru/node-js/topic/5002/монгодская-интеграция>

---

# глава 65: Монгузская библиотека

## Examples

### Подключение к MongoDB Использование Mongoose

Во-первых, установите Mongoose с:

```
npm install mongoose
```

Затем добавьте его в `server.js` как зависимости:

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
```

Затем создайте схему базы данных и имя коллекции:

```
var schemaName = new Schema({
  request: String,
  time: Number
}, {
  collection: 'collectionName'
});
```

Создайте модель и подключитесь к базе данных:

```
var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');
```

Затем запустите MongoDB и запустите `server.js` используя `node server.js`

Чтобы проверить, успешно ли мы подключены к базе данных, мы можем использовать `open` события, `error` из объекта `mongoose.connection`.

```
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'connection error:'));
db.once('open', function() {
  // we're connected!
});
```

### Сохранение данных в MongoDB с использованием маршрутов Mongoose и Express.js

---

## Настроить

Сначала установите необходимые пакеты с помощью:

```
npm install express cors mongoose
```

## Код

Затем добавьте зависимости к файлу `server.js`, создайте схему базы данных и имя коллекции, создайте сервер Express.js и подключитесь к MongoDB:

```
var express = require('express');
var cors = require('cors'); // We will use CORS to enable cross origin domain requests.
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var app = express();

var schemaName = new Schema({
  request: String,
  time: Number
}, {
  collection: 'collectionName'
});

var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');

var port = process.env.PORT || 8080;
app.listen(port, function() {
  console.log('Node.js listening on port ' + port);
});
```

Теперь добавьте маршруты Express.js, которые мы будем использовать для записи данных:

```
app.get('/save/:query', cors(), function(req, res) {
  var query = req.params.query;

  var savedata = new Model({
    'request': query,
    'time': Math.floor(Date.now() / 1000) // Time of save the data in unix timestamp
format
  }).save(function(err, result) {
    if (err) throw err;

    if(result) {
      res.json(result)
    }
  })
});
```

Здесь переменная `query` будет параметром `<query>` из входящего HTTP-запроса, который будет сохранен в MongoDB:

```
var savedata = new Model({
```

```
'request': query,  
//...
```

Если при попытке записи в MongoDB возникает ошибка, вы получите сообщение об ошибке на консоли. Если все будет успешным, вы увидите сохраненные данные в формате JSON на странице.

```
//...  
}).save(function(err, result) {  
  if (err) throw err;  
  
  if(result) {  
    res.json(result)  
  }  
})  
//...
```

Теперь вам нужно запустить MongoDB и запустить файл `server.js` используя `node server.js`

---

## ИСПОЛЬЗОВАНИЕ

Чтобы использовать это для сохранения данных, перейдите к следующему URL-адресу в своем браузере:

```
http://localhost:8080/save/<query>
```

Где `<query>` - новый запрос, который вы хотите сохранить.

Пример:

```
http://localhost:8080/save/JavaScript%20is%20Awesome
```

Выход в формате JSON:

```
{  
  __v: 0,  
  request: "JavaScript is Awesome",  
  time: 1469411348,  
  _id: "57957014b93bc8640f2c78c4"  
}
```

Поиск данных в MongoDB Использование маршрутов Mongoose и Express.js

---

## Настроить

Сначала установите необходимые пакеты с помощью:

```
npm install express cors mongoose
```

## Код

Затем добавьте зависимости к `server.js`, создайте схему базы данных и имя коллекции, создайте сервер Express.js и подключитесь к MongoDB:

```
var express = require('express');
var cors = require('cors'); // We will use CORS to enable cross origin domain requests.
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var app = express();

var schemaName = new Schema({
  request: String,
  time: Number
}, {
  collection: 'collectionName'
});

var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');

var port = process.env.PORT || 8080;
app.listen(port, function() {
  console.log('Node.js listening on port ' + port);
});
```

Теперь добавьте маршруты Express.js, которые мы будем использовать для запроса данных:

```
app.get('/find/:query', cors(), function(req, res) {
  var query = req.params.query;

  Model.find({
    'request': query
  }, function(err, result) {
    if (err) throw err;
    if (result) {
      res.json(result)
    } else {
      res.send(JSON.stringify({
        error : 'Error'
      }))
    }
  })
})
```

Предположим, что следующие документы находятся в коллекции в модели:

```
{
  "_id" : ObjectId("578abe97522ad414b8eeb55a"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710551
}
{
  "_id" : ObjectId("578abe9b522ad414b8eeb55b"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710555
}
{
  "_id" : ObjectId("578abea0522ad414b8eeb55c"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710560
}
```

И цель состоит в том, чтобы найти и отобразить все документы, содержащие "JavaScript is Awesome" ПОД КЛЮЧОМ "request" .

Для этого запустите MongoDB и запустите `server.js` с `node server.js` :

---

## ИСПОЛЬЗОВАНИЕ

Чтобы использовать это для поиска данных, перейдите по следующему URL-адресу в браузере:

```
http://localhost:8080/find/<query>
```

Где `<query>` - поисковый запрос.

Пример:

```
http://localhost:8080/find/JavaScript%20is%20Awesome
```

Выход:

```
[{
  _id: "578abe97522ad414b8eeb55a",
  request: "JavaScript is Awesome",
  time: 1468710551,
  __v: 0
},
{
  _id: "578abe9b522ad414b8eeb55b",
  request: "JavaScript is Awesome",
  time: 1468710555,
  __v: 0
},
{
  _id: "578abea0522ad414b8eeb55c",
  request: "JavaScript is Awesome",
  time: 1468710560,
```

```
  __v: 0
}]
```

## Поиск данных в MongoDB Использование маршрутов Mongoose, Express.js и \$ text Operator

### Настроить

Сначала установите необходимые пакеты с помощью:

```
npm install express cors mongoose
```

### Код

Затем добавьте зависимости к `server.js`, создайте схему базы данных и имя коллекции, создайте сервер Express.js и подключитесь к MongoDB:

```
var express = require('express');
var cors = require('cors'); // We will use CORS to enable cross origin domain requests.
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var app = express();

var schemaName = new Schema({
  request: String,
  time: Number
}, {
  collection: 'collectionName'
});

var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');

var port = process.env.PORT || 8080;
app.listen(port, function() {
  console.log('Node.js listening on port ' + port);
});
```

Теперь добавьте маршруты Express.js, которые мы будем использовать для запроса данных:

```
app.get('/find/:query', cors(), function(req, res) {
  var query = req.params.query;

  Model.find({
    'request': query
  }, function(err, result) {
    if (err) throw err;
```



```

    if (result) {
      res.json(result)
    } else {
      res.send(JSON.stringify({
        error : 'Error'
      }))
    }
  })
})
})

```

Предположим, что следующие документы находятся в коллекции в модели:

```

{
  "_id" : ObjectId("578abe97522ad414b8eeb55a"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710551
}
{
  "_id" : ObjectId("578abe9b522ad414b8eeb55b"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710555
}
{
  "_id" : ObjectId("578abea0522ad414b8eeb55c"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710560
}

```

И цель состоит в том, чтобы найти и отобразить все документы, содержащие только слово "JavaScript" под ключом "request".

Для этого сначала создайте *текстовый индекс* для "request" в коллекции. Для этого добавьте следующий код в `server.js`:

```

schemaName.index({ request: 'text' });

```

И замените:

```

Model.find({
  'request': query
}, function(err, result) {

```

С:

```

Model.find({
  $text: {
    $search: query
  }
}, function(err, result) {

```

Здесь мы используем `$text` и `$search` MongoDB-операторы для поиска всех документов в коллекции `collectionName` которое содержит хотя бы одно слово из указанного запроса на поиск.

---

# ИСПОЛЬЗОВАНИЕ

Чтобы использовать это для поиска данных, перейдите по следующему URL-адресу в браузере:

```
http://localhost:8080/find/<query>
```

Где `<query>` - поисковый запрос.

Пример:

```
http://localhost:8080/find/JavaScript
```

Выход:

```
[{
  _id: "578abe97522ad414b8eeb55a",
  request: "JavaScript is Awesome",
  time: 1468710551,
  __v: 0
},
{
  _id: "578abe9b522ad414b8eeb55b",
  request: "JavaScript is Awesome",
  time: 1468710555,
  __v: 0
},
{
  _id: "578abea0522ad414b8eeb55c",
  request: "JavaScript is Awesome",
  time: 1468710560,
  __v: 0
}]
```

## Индексы в моделях.

MongoDB поддерживает вторичные индексы. В Mongoose мы определяем эти индексы в нашей схеме. Определение индексов на уровне схемы необходимо, когда нам нужно создавать составные индексы.

## Соединение Mongoose

```
var strConnection = 'mongodb://localhost:27017/dbName';
var db = mongoose.createConnection(strConnection)
```

## Создание базовой схемы

```
var Schema = require('mongoose').Schema;
var usersSchema = new Schema({
```

```

username: {
  type: String,
  required: true,
  unique: true
},
email: {
  type: String,
  required: true
},
password: {
  type: String,
  required: true
},
created: {
  type: Date,
  default: Date.now
}
});

var usersModel = db.model('users', usersSchema);
module.exports = usersModel;

```

По умолчанию mongoose добавляет два новых поля в нашу модель, даже если они не определены в модели. Эти поля:

### **Я бы**

Mongoose присваивает каждой из ваших схем поле `_id` по умолчанию, если он не передается в конструктор схемы. Назначенный тип - `ObjectId`, который совпадает с поведением MongoDB по умолчанию. Если вы не хотите, чтобы `_id` добавили в вашу схему вообще, вы можете отключить ее, используя эту опцию.

```

var usersSchema = new Schema({
  username: {
    type: String,
    required: true,
    unique: true
  }, {
    _id: false
});

```

### **\_\_v или versionKey**

VersionKey - это свойство, установленное для каждого документа при первом создании Mongoose. Это значение ключа содержит внутреннюю ревизию документа. Имя этого свойства документа настраивается.

Вы можете легко отключить это поле в конфигурации модели:

```

var usersSchema = new Schema({
  username: {
    type: String,
    required: true,
    unique: true

```

```
  }, {  
    versionKey: false  
  });
```

## Составные индексы

Мы можем создать еще один индекс, помимо того, что создает Mongoose.

```
usersSchema.index({username: 1 });  
usersSchema.index({email: 1 });
```

В этом случае наша модель имеет еще два индекса: один для имени пользователя поля и другой для поля электронной почты. Но мы можем создавать составные индексы.

```
usersSchema.index({username: 1, email: 1 });
```

## Влияние производительности индекса

По умолчанию mongoose всегда вызывает `makeIndex` для каждого индекса последовательно и генерирует событие «index» на модели, когда все вызовы `secureIndex` преуспели или когда произошла ошибка.

В MongoDB `securityIndex` устарел с версии 3.0.0, теперь является псевдонимом для `createIndex`.

Рекомендуется отключить поведение, установив для параметра `autoIndex` вашей схемы значение `false` или глобально на соединение, установив опцию `config.autoIndex` равным `false`.

```
usersSchema.set('autoIndex', false);
```

## Полезные функции Мангуста

Mongoose содержит некоторые встроенные функции, которые основаны на стандартном `find()`.

```
doc.find({'some.value':5}, function(err, docs) {  
  //returns array docs  
});  
  
doc.findOne({'some.value':5}, function(err, doc) {  
  //returns document doc  
});  
  
doc.findById(obj._id, function(err, doc) {  
  //returns document doc  
});
```

## найти данные в mongodb, используя обещания

---

# Настроить

Сначала установите необходимые пакеты с помощью:

```
npm install express cors mongoose
```

---

# Код

Затем добавьте зависимости к `server.js`, создайте схему базы данных и имя коллекции, создайте сервер Express.js и подключитесь к MongoDB:

```
var express = require('express');
var cors = require('cors'); // We will use CORS to enable cross origin domain requests.
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var app = express();

var schemaName = new Schema({
  request: String,
  time: Number
}, {
  collection: 'collectionName'
});

var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');

var port = process.env.PORT || 8080;
app.listen(port, function() {
  console.log('Node.js listening on port ' + port);
});

app.use(function(err, req, res, next) {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});

app.use(function(req, res, next) {
  res.status(404).send('Sorry cant find that!');
});
```

Теперь добавьте маршруты Express.js, которые мы будем использовать для запроса данных:

```
app.get('/find/:query', cors(), function(req, res, next) {
  var query = req.params.query;

  Model.find({
    'request': query
  })
```

```
.exec() //remember to add exec, queries have a .then attribute but aren't promises
.then(function(result) {
  if (result) {
    res.json(result)
  } else {
    next() //pass to 404 handler
  }
})
.catch(next) //pass to error handler
})
```

Предположим, что следующие документы находятся в коллекции в модели:

```
{
  "_id" : ObjectId("578abe97522ad414b8eeb55a"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710551
}
{
  "_id" : ObjectId("578abe9b522ad414b8eeb55b"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710555
}
{
  "_id" : ObjectId("578abea0522ad414b8eeb55c"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710560
}
```

И цель состоит в том, чтобы найти и отобразить все документы, содержащие "JavaScript is Awesome" ПОД КЛЮЧОМ "request" .

Для этого запустите MongoDB и запустите `server.js` с `node server.js` :

---

## ИСПОЛЬЗОВАНИЕ

Чтобы использовать это для поиска данных, перейдите по следующему URL-адресу в браузере:

```
http://localhost:8080/find/<query>
```

Где `<query>` - поисковый запрос.

Пример:

```
http://localhost:8080/find/JavaScript%20is%20Awesome
```

Выход:

```
[{
  "_id": "578abe97522ad414b8eeb55a",
```

```
  request: "JavaScript is Awesome",
  time: 1468710551,
  __v: 0
},
{
  _id: "578abe9b522ad414b8eeb55b",
  request: "JavaScript is Awesome",
  time: 1468710555,
  __v: 0
},
{
  _id: "578abea0522ad414b8eeb55c",
  request: "JavaScript is Awesome",
  time: 1468710560,
  __v: 0
}]
```

Прочитайте Монгузская библиотека онлайн: <https://riptutorial.com/ru/node-js/topic/3486/>  
монгузская-библиотека

# глава 66: мотыга

## Examples

### Добавьте новые расширения, требующие ()

Вы можете добавить новые расширения для `require()` путем расширения `require.extensions`.

Пример XML :

```
// Add .xml for require()
require.extensions['.xml'] = (module, filename) => {
  const fs = require('fs')
  const xml2js = require('xml2js')

  module.exports = (callback) => {
    // Read required file.
    fs.readFile(filename, 'utf8', (err, data) => {
      if (err) {
        callback(err)
        return
      }
      // Parse it.
      xml2js.parseString(data, (err, result) => {
        callback(null, result)
      })
    })
  })
}
```

Если содержимое `hello.xml` следующему:

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
  <bar>baz</bar>
  <qux />
</foo>
```

Вы можете прочитать и проанализировать его с помощью `require()` :

```
require('./hello')((err, xml) {
  if (err)
    throw err;
  console.log(err);
})
```

Он печатает `{ foo: { bar: [ 'baz' ], qux: [ ' ' ] } }`.

Прочитайте мотыга онлайн: <https://riptutorial.com/ru/node-js/topic/6645/мотыга>



---

# глава 67: Начало работы с профилированием узлов

## Вступление

Цель этого сообщения - начать работу с профилирующим узлом applicationjs и понять, как это сделать, чтобы зафиксировать ошибку или утечку памяти. Приложение, выполняющее nodejs, является ничем иным, как процессами v8 engine, которые во многом совпадают с веб-сайтом, запущенным в браузере, и мы можем в основном захватить все показатели, связанные с процессом веб-сайта для приложения-узла.

Инструментом моего предпочтения является chrome devtools или chrome inspector в сочетании с инспектором узлов.

## замечания

Инспектор узлов не может присоединиться к процессу debug узла, иногда в этом случае вы не сможете получить контрольную точку debug в devtools. Повторите обновление вкладки devtools несколько раз и подождите несколько секунд, чтобы увидеть, находится ли она в режиме отладки.

Если не перезапустить инспектор узлов из командной строки.

## Examples

### Профилирование простого узла

**Шаг 1** : Установите пакет инспектора узлов, используя npm глобально на вашей машине

```
$ npm install -g node-inspector
```

**Шаг 2.** Запустите сервер узла-инспектора.

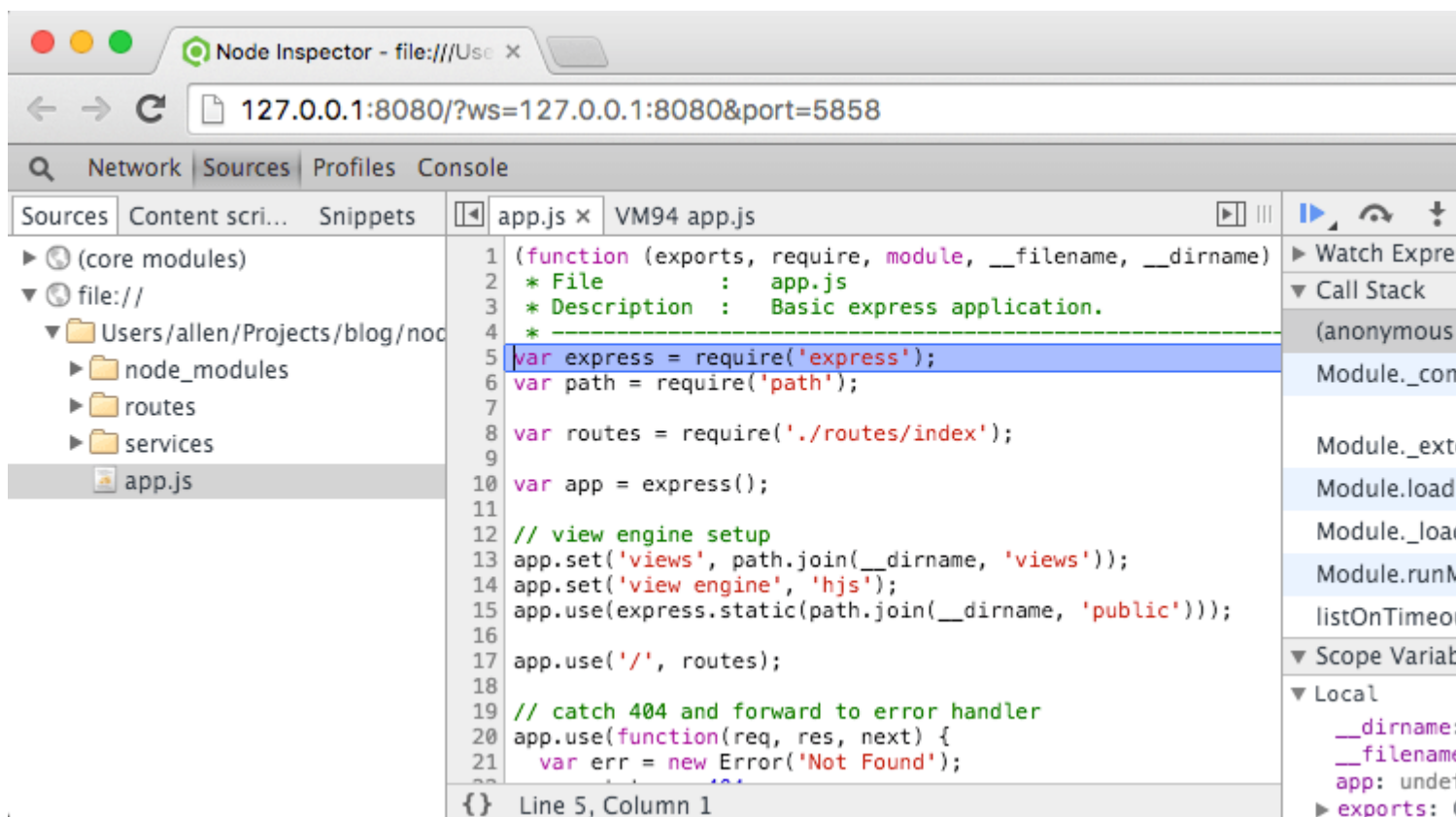
```
$ node-inspector
```

**Шаг 3.** Запустите отладку приложения узла.

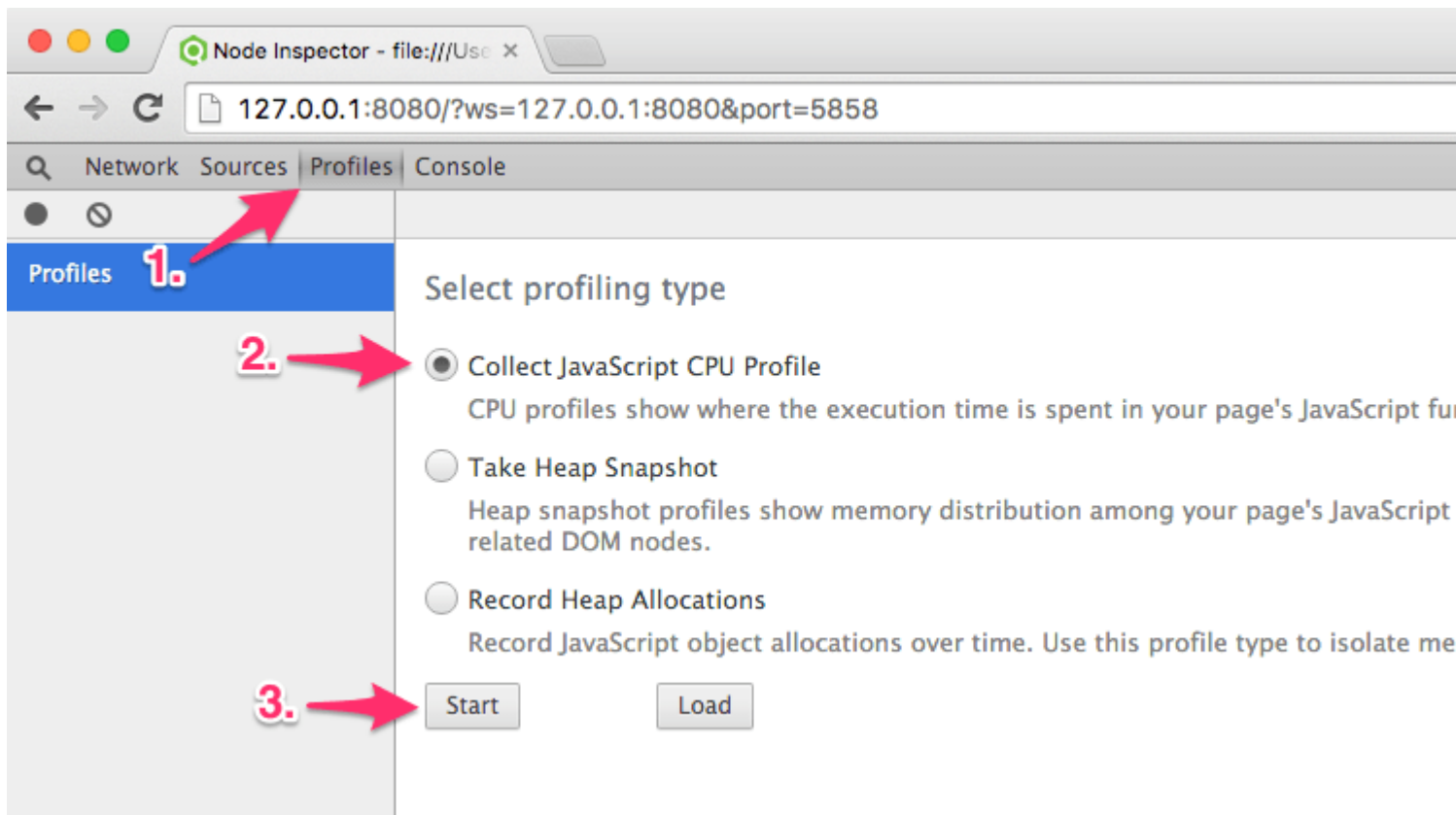
```
$ node --debug-brk your/short/node/script.js
```

**Шаг 4** : Откройте <http://127.0.0.1:8080/?port=5858> в браузере Chrome. И вы увидите интерфейс инструментов chrom-dev с исходным кодом приложения nodejs на левой панели.

И поскольку мы использовали опцию отладки при отладке приложения, выполнение кода останавливается в первой строке кода.



**Шаг 5** : Это легкая часть, на которой вы переключаетесь на вкладку профилирования и начинаете профилирование приложения. В случае, если вы хотите получить профиль для определенного метода или потока, убедитесь, что выполнение кода разломано перед тем, как этот фрагмент кода будет выполнен.



**Шаг 6** : После того, как вы зарегистрировали свой профиль процессора или кучу дампа / моментального снимка или распределение кучи, вы можете просмотреть результаты в том же окне или сохранить их на локальном диске для последующего анализа или сравнения с другими профилями.

Вы можете использовать эти статьи, чтобы узнать, как читать профили:

- [Чтение профилей процессора](#)
- [Профилировщик процессора Chrome и куратор](#)

Прочитайте [Начало работы с профилированием узлов онлайн](https://riptutorial.com/ru/node-js/topic/9347/начало-работы-с-профилированием-узлов): <https://riptutorial.com/ru/node-js/topic/9347/начало-работы-с-профилированием-узлов>

---

# глава 68: НПМ

## Вступление

Диспетчер пакетов узлов (npm) предоставляет следующие две основные функции: онлайн-репозитории для пакетов / модулей node.js, которые можно найти на [search.npmjs.org](https://search.npmjs.org).  
Утилита командной строки для установки пакетов Node.js, управления версиями и управления зависимостями пакетов Node.js.

## Синтаксис

- npm <command>, где <command> является одним из следующих:
  - [Добавить пользователя](#)
  - [Добавить пользователя](#)
  - [apihelp](#)
  - [автор](#)
  - [бункер](#)
  - [ошибки](#)
  - [с](#)
  - [кэш](#)
  - [завершение](#)
  - [конфиг](#)
  - [DDP](#)
  - [дедупликации](#)
  - [протестовать](#)
  - [документы](#)
  - [редактировать](#)
  - [проводить исследования](#)
  - [Часто задаваемые вопросы](#)
  - [находить](#)
  - [найти-простофиля](#)
  - [получить](#)
  - [Помогите](#)
  - [помощь-поиск](#)
  - [Главная](#)
  - [я](#)
  - [устанавливать](#)
  - [Информация](#)
  - [В ЭТОМ](#)
  - [isntall](#)
  - [проблемы](#)

- ля
- [ссылка на сайт](#)
- [список](#)
- Л.Л.
- пер
- авторизоваться
- Ls
- устаревший
- [владелец](#)
- пак
- префикс
- [чернослив](#)
- [публиковать](#)
- р
- [Р.Б.](#)
- [перестраивать](#)
- Удалить
- [Сделки РЕПО](#)
- [запустить снова](#)
- [комната](#)
- [корень](#)
- [выполнения сценария](#)
- s
- [се](#)
- [поиск](#)
- [задавать](#)
- шоу
- Упаковочная
- [звезда](#)
- [звезды](#)
- Начните
- [стоп](#)
- [подмодуль](#)
- [тег](#)
- [тестовое задание](#)
- TST
- ООН
- [деинсталляция](#)
- [разъединить](#)
- [Отменить публикацию](#)
- [снять пометку](#)
- [вверх](#)
- [Обновить](#)

- [v](#)
- [версия](#)
- [Посмотреть](#)
- [кто я](#)

## параметры

параметр	пример
<a href="#">доступ</a>	<code>npm publish --access=public</code>
<a href="#">бункер</a>	<code>npm bin -g</code>
<a href="#">редактировать</a>	<code>npm edit connect</code>
<a href="#">Помогите</a>	<code>npm help init</code>
<a href="#">в этом</a>	<code>npm init</code>
<a href="#">устанавливать</a>	<code>npm install</code>
<a href="#">ссылка на сайт</a>	<code>npm link</code>
<a href="#">чернослив</a>	<code>npm prune</code>
<a href="#">публиковать</a>	<code>npm publish ./</code>
<a href="#">запустить снова</a>	<code>npm restart</code>
<a href="#">Начните</a>	<code>npm start</code>
<a href="#">стоп</a>	<code>npm start</code>
<a href="#">Обновить</a>	<code>npm update</code>
<a href="#">версия</a>	<code>npm version</code>

## Examples

### Установка пакетов

## Вступление

Пакет - это термин, используемый npm для обозначения инструментов, которые

разработчики могут использовать для своих проектов. Это включает в себя все: от библиотек и фреймворков, таких как jQuery и AngularJS, до задач, таких как Gulp.js. Пакеты появятся в папке, обычно называемой `node_modules`, которая также будет содержать файл `package.json`. Этот файл содержит информацию обо всех пакетах, включая любые зависимости, которые являются дополнительными модулями, необходимыми для использования определенного пакета.

Npm использует командную строку для установки и управления пакетами, поэтому пользователи, пытающиеся использовать npm, должны быть знакомы с базовыми командами в своей операционной системе, то есть: перемещаться по каталогам, а также видеть содержимое каталогов.

---

## Установка NPM

Обратите внимание, что для установки пакетов необходимо установить NPM.

Рекомендуемый способ установки NPM - использовать один из установщиков со [страницы загрузки Node.js](#). Вы можете проверить, установлено ли у вас уже установленное node.js, выполнив команду `npm -v` или `npm version`.

После установки NPM с помощью установщика Node.js обязательно проверьте наличие обновлений. Это связано с тем, что NPM обновляется чаще, чем установщик Node.js. Чтобы проверить наличие обновлений, выполните следующую команду:

```
npm install npm@latest -g
```

---

## Как установить пакеты

Для установки одного или нескольких пакетов используйте следующее:

```
npm install <package-name>
# or
npm i <package-name>...

# e.g. to install lodash and express
npm install lodash express
```

**Примечание**. Это установит пакет в каталог, в который в настоящее время входит командная строка, поэтому важно проверить, выбран ли соответствующий каталог

Если у вас уже есть файл `package.json` в вашем текущем рабочем каталоге, и в нем

определены зависимости, то `npm install` автоматически разрешит и установит все зависимости, перечисленные в файле. Вы также можете использовать сокращенную версию команды `npm install` которая: `npm i`

Если вы хотите установить определенную версию пакета, используйте:

```
npm install <name>@<version>

# e.g. to install version 4.11.1 of the package lodash
npm install lodash@4.11.1
```

Если вы хотите установить версию, которая соответствует определенному диапазону версий, используйте:

```
npm install <name>@<version range>

# e.g. to install a version which matches "version >= 4.10.1" and "version < 4.11.1"
# of the package lodash
npm install lodash@">=4.10.1 <4.11.1"
```

Если вы хотите установить последнюю версию, используйте:

```
npm install <name>@latest
```

Вышеупомянутые команды будут искать пакеты в центральном репозитории `npm` на [npmjs.com](https://www.npmjs.com). Если вы не хотите устанавливать из реестра `npm`, поддерживаются другие параметры, такие как:

```
# packages distributed as a tarball
npm install <tarball file>
npm install <tarball url>

# packages available locally
npm install <local path>

# packages available as a git repository
npm install <git remote url>

# packages available on GitHub
npm install <username>/<repository>

# packages available as gist (need a package.json)
npm install gist:<gist-id>

# packages from a specific repository
npm install --registry=http://myreg.mycompany.com <package name>

# packages from a related group of packages
# See npm scope
npm install @<scope>/<name>(@<version>)

# Scoping is useful for separating private packages hosted on private registry from
# public ones by setting registry for specific scope
npm config set @mycompany:registry http://myreg.mycompany.com
```



```
npm install @mycompany/<package name>
```

Обычно модули устанавливаются локально в папке с именем `node_modules`, которая может быть найдена в текущем рабочем каталоге. Это каталог `require()` будет использовать для загрузки модулей, чтобы сделать их доступными для вас.

Если вы уже создали файл `package.json`, вы можете использовать `--save` (стенографию `-S`) или один из ее вариантов, чтобы автоматически добавить установленный пакет к вашему `package.json` в качестве зависимости. Если кто-то еще установит ваш пакет, `npm` автоматически прочтает зависимости из файла `package.json` и установит перечисленные версии. Обратите внимание, что вы все равно можете добавлять и управлять своими зависимостями, редактируя файл позже, поэтому обычно рекомендуется отслеживать зависимости, например:

```
npm install --save <name> # Install dependencies
# or
npm install -S <name> # shortcut version --save
# or
npm i -S <name>
```

Чтобы устанавливать пакеты и сохранять их только в том случае, если они необходимы для разработки, а не для их запуска, а не если они необходимы для запуска приложения, выполните следующую команду:

```
npm install --save-dev <name> # Install dependencies for development purposes
# or
npm install -D <name> # shortcut version --save-dev
# or
npm i -D <name>
```

---

## Установка зависимостей

Некоторые модули не только предоставляют библиотеку для вас, но также предоставляют один или несколько двоичных файлов, которые предназначены для использования в командной строке. Хотя вы все равно можете устанавливать эти пакеты локально, часто рекомендуется устанавливать их в глобальном масштабе, чтобы инструменты командной строки могли быть включены. В этом случае `npm` автоматически свяжет двоичные файлы с соответствующими путями (например, `/usr/local/bin/<name>`), чтобы их можно было использовать из командной строки. Чтобы установить пакет по всему миру, используйте:

```
npm install --global <name>
# or
npm install -g <name>
# or
npm i -g <name>
```

```
# e.g. to install the grunt command line tool
npm install -g grunt-cli
```

Если вы хотите просмотреть список всех установленных пакетов и связанных с ними версий в текущей рабочей области, используйте:

```
npm list
npm list <name>
```

Добавление необязательного аргумента имени может проверить версию определенного пакета.

**Примечание.** Если вы сталкиваетесь с проблемами разрешений при попытке установить модуль `npm` во всем мире, не поддавайтесь искушению выпустить `sudo npm install -g ...` для решения проблемы. Предоставление сторонних скриптов для работы с вашей системой с повышенными привилегиями опасно. Проблема с разрешением может означать, что у вас возникла проблема с тем, как была установлена сама `npm`. Если вы заинтересованы в установке узла в изолированной среде, вы можете попробовать использовать [nvm](#).

Если у вас есть инструменты для сборки или другие зависимости только для разработки (например, Grunt), вы можете не захотеть связать их с приложением, которое вы развертываете. Если это так, вы хотите, чтобы он был зависимым от разработки, который указан в `package.json` под `devDependencies`. Чтобы установить пакет как зависимость от разработки, используйте `--save-dev` (или `-D`).

```
npm install --save-dev <name> // Install development dependencies which is not included in
production
# or
npm install -D <name>
```

Вы увидите, что пакет затем добавляется в `devDependencies` вашего `package.json`.

Чтобы установить зависимости загруженного / клонированного проекта `node.js`, вы можете просто использовать

```
npm install
# or
npm i
```

`npm` автоматически считывает зависимости от `package.json` и устанавливает их.

---

## NPM за прокси-сервером

Если ваш интернет-доступ через прокси-сервер, вам может потребоваться изменить команды установки `npm`, которые будут обращаться к удаленным репозиториям. `npm`

использует файл конфигурации, который может быть обновлен через командную строку:

```
npm config set
```

Вы можете найти свои настройки прокси на панели настроек вашего браузера. После того, как вы получили настройки прокси-сервера (URL-адрес сервера, порт, имя пользователя и пароль); вам необходимо настроить конфигурацию npm следующим образом.

```
$ npm config set proxy http://<username>:<password>@<proxy-server-url>:<port>
$ npm config set https-proxy http://<username>:<password>@<proxy-server-url>:<port>
```

username , password , поля port являются необязательными. Как только вы их установили, ваша npm install , npm i -g и т. Д. npm i -g работать правильно.

## Области и репозитории

```
# Set the repository for the scope "myscope"
npm config set @myscope:registry http://registry.corporation.com

# Login at a repository and associate it with the scope "myscope"
npm adduser --registry=http://registry.corporation.com --scope=@myscope

# Install a package "mylib" from the scope "myscope"
npm install @myscope/mylib
```

Если имя вашего собственного пакета начинается с @myscope а область «myscope» связана с другим репозиторием, npm publish будет загружать ваш пакет в этот репозиторий.

Вы также можете сохранить эти настройки в файле .npmrc :

```
@myscope:registry=http://registry.corporation.com
//registry.corporation.com/:_authToken=xxxxxxxx-xxxx-xxxx-xxxxxxxxxxxxxxxx
```

Это полезно при автоматизации сборки на CI-сервере fe

## Удаление пакетов

Чтобы удалить один или несколько локально установленных пакетов, используйте:

```
npm uninstall <package name>
```

Команда удаления для npm содержит пять псевдонимов, которые также можно использовать:

```
npm remove <package name>
npm rm <package name>
npm r <package name>
```

```
npm unlink <package name>
npm un <package name>
```

Если вы хотите удалить пакет из файла `package.json` как часть удаления, используйте флаг `--save` (стенограмма: `-s`):

```
npm uninstall --save <package name>
npm uninstall -S <package name>
```

Для зависимостей разработки используйте флаг `--save-dev` (стенограмма: `-D`):

```
npm uninstall --save-dev <package name>
npm uninstall -D <package name>
```

Для дополнительной зависимости используйте флаг `--save-optional` (стенограмма: `-O`):

```
npm uninstall --save-optional <package name>
npm uninstall -O <package name>
```

Для пакетов, которые установлены глобально, используйте флаг `--global` (стенограмма: `-g`):

```
npm uninstall -g <package name>
```

## Основные семантические версии

Перед публикацией пакета вам нужно его выпустить. npm поддерживает [семантическое управление версиями](#), это означает, что есть **патч**, **младший** и **основной** релиз.

Например, если ваш пакет находится в версии 1.2.3 для изменения версии, вам необходимо:

1. Патч релиз: `npm version patch => 1.2.4`
2. младший выпуск: `npm version minor => 1.3.0`
3. основной выпуск: `npm version major => 2.0.0`

Вы также можете указать версию напрямую:

```
npm version 3.1.4 => 3.1.4
```

Когда вы устанавливаете версию пакета с помощью одной из приведенных выше команд npm, npm изменяет поле версии файла `package.json`, фиксирует его, а также создает новый тег Git с версией с префиксом «v», как если бы вы Выпустили команду:

```
git tag v3.1.4
```

В отличие от других менеджеров пакетов, таких как Bower, реестр npm не полагается на

теги Git, которые создаются для каждой версии. Но, если вам нравится использовать теги, вы должны помнить о том, чтобы нажимать вновь созданный тег после нападения на версию пакета:

```
git push origin master (нажать на изменение package.json)
```

```
git push origin v3.1.4 (нажать новый тег)
```

Или вы можете сделать это одним махом:

```
git push origin master --tags
```

## Настройка конфигурации пакета

Конфигурации пакетов Node.js содержатся в файле `package.json` который вы можете найти в корне каждого проекта. Вы можете настроить новый файл конфигурации, вызвав:

```
npm init
```

Это попытается прочитать текущую рабочую директорию для информации репозитория Git (если она существует) и переменных среды, чтобы попытаться автозаполнить некоторые из значений `placeholder` для вас. В противном случае он предоставит диалоговое окно ввода для основных параметров.

Если вы хотите создать `package.json` со значениями по умолчанию, используйте:

```
npm init --yes
# or
npm init -y
```

Если вы создаете `package.json` для проекта, который вы не собираетесь публиковать в виде пакета npm (т. Е. Исключительно для округления ваших зависимостей), вы можете передать это намерение в файле `package.json` :

1. При желании для `private` публикации запрещается публикация `private` собственности.
2. Необязательно установите для свойства `license` значение «НЕИЗВЕСТНОЕ», чтобы лишить других права использовать ваш пакет.

Чтобы установить пакет и автоматически сохранить его в `package.json` , используйте:

```
npm install --save <package>
```

Пакет и связанные метаданные (например, версия пакета) будут отображаться в ваших зависимостях. Если вы сохраните, если в качестве зависимости развития (используя `--save-dev` ) пакет вместо этого появится в ваших `devDependencies` .

С помощью этого `bare-bones package.json` вы будете сталкиваться с предупреждающими

сообщениями при установке или обновлении пакетов, сообщая вам, что вам не хватает описания и поля репозитория. Хотя безопасно игнорировать эти сообщения, вы можете избавиться от них, открыв `package.json` в любом текстовом редакторе и добавив следующие строки к объекту JSON:

```
[...]  
"description": "No description",  
"repository": {  
  "private": true  
},  
[...]
```

## Публикация пакета

Во-первых, убедитесь, что вы настроили свой пакет (как указано в [разделе «Настройка конфигурации пакета»](#)). Затем вы должны войти в `npmjs`.

Если у вас уже есть пользователь `npm`

```
npm login
```

Если у вас нет пользователя

```
npm adduser
```

Чтобы проверить, что ваш пользователь зарегистрирован в текущем клиенте

```
npm config ls
```

После этого, когда ваш пакет готов к публикации, используйте

```
npm publish
```

И все готово.

Если вам нужно опубликовать новую версию, убедитесь, что вы обновили версию своего пакета, как указано в [Basic semantic versioning](#). В противном случае `npm` не позволит вам опубликовать пакет.

```
{  
  name: "package-name",  
  version: "1.0.4"  
}
```

## Запуск скриптов

Вы можете определить скрипты в вашем `package.json`, например:

```
{
  "name": "your-package",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "author": "",
  "license": "ISC",
  "dependencies": {},
  "devDependencies": {},
  "scripts": {
    "echo": "echo hello!"
  }
}
```

Чтобы запустить `echo` скрипт, запустите `npm run echo` из командной строки. Произвольные сценарии, такие как `echo` выше, должны выполняться с `npm run <script name>`. `npm` также имеет ряд официальных сценариев, которые он запускает на определенных этапах жизни пакета (например, `preinstall`). См. [Здесь](#) полный обзор того, как `npm` обрабатывает поля сценария.

Скрипты `npm` используются чаще всего для таких вещей, как запуск сервера, построение проекта и запуск тестов. Вот более реалистичный пример:

```
"scripts": {
  "test": "mocha tests",
  "start": "pm2 start index.js"
}
```

В записях `scripts` командной строки, такие как `mocha` будут работать, если они установлены либо глобально, либо локально. Если запись в командной строке не существует в системе PATH, `npm` также проверит ваши локально установленные пакеты.

Если ваши скрипты становятся очень длинными, их можно разбить на части, например:

```
"scripts": {
  "very-complex-command": "npm run chain-1 && npm run chain-2",
  "chain-1": "webpack",
  "chain-2": "node app.js"
}
```

## Удаление посторонних пакетов

Для удаления посторонних пакетов (пакетов, которые установлены, но не в списке зависимостей) выполните следующую команду:

```
npm prune
```

Чтобы удалить все пакеты `dev` добавьте флаг `--production`:

```
npm prune --production
```

[Подробнее об этом](#)

## Список установленных пакетов

Чтобы создать список (древовидное представление) установленных пакетов, используйте

```
npm list
```

**ls** , **la** и **ll** - это псевдонимы команды **списка** . Команды **la** и **ll** показывают расширенную информацию, такую как описание и репозиторий.

### Опции

Формат ответа можно изменить, передав параметры.

```
npm list --json
```

- **json** - отображает информацию в формате json
- **long** - показывает расширенную информацию
- **parseable** - показывает список синтаксического анализа вместо дерева
- **global** - показывает глобально установленные пакеты
- **depth** - максимальная глубина отображения дерева зависимостей
- **dev / development** - показывает devDependencies
- **prod / production** - Показывает зависимости

Если вы хотите, вы также можете перейти на домашнюю страницу пакета.

```
npm home <package name>
```

## Обновление npm и пакетов

Поскольку сам npm является модулем Node.js, он может быть обновлен с использованием самого себя.

*Если ОС - это Windows, необходимо запустить командную строку в качестве администратора*

```
npm install -g npm@latest
```

Если вы хотите проверить обновленные версии, вы можете:

```
npm outdated
```



Чтобы обновить конкретный пакет:

```
npm update <package name>
```

Это обновит пакет до последней версии в соответствии с ограничениями в `package.json`

Если вы также хотите заблокировать обновленную версию в `package.json`:

```
npm update <package name> --save
```

## Блокировка модулей для определенных версий

По умолчанию npm устанавливает последнюю доступную версию модулей в соответствии с [семантической версией](#) каждой зависимости. Это может быть проблематично, если автор модуля не придерживается `semver` и, например, вносит изменения в обновление модуля.

Чтобы заблокировать каждую версию зависимостей (и версии их зависимостей и т. Д.) Для конкретной версии, установленной локально в папке `node_modules`, используйте

```
npm shrinkwrap
```

Затем будет создан `npm-shrinkwrap.json` вместе с вашим `package.json` которым перечислены конкретные версии зависимостей.

## Настройка для общедоступных пакетов

Вы можете использовать `npm install -g` для установки пакета «глобально». Обычно это делается для установки исполняемого файла, который вы можете добавить к своему пути для запуска. Например:

```
npm install -g gulp-cli
```

Если вы обновите свой путь, вы можете вызвать `gulp` напрямую.

Во многих ОС `npm install -g` попытается записать в каталог, который ваш пользователь может не написать, например, `/usr/bin`. Вы **не** должны использовать `sudo npm install` в этом случае, так как существует вероятность угрозы выполнения произвольных скриптов с помощью `sudo` и пользователь `root` может создавать каталоги в вашем доме, которые вы не можете записать, что затрудняет будущие установки.

Вы можете указать `npm` где устанавливать глобальные модули через ваш конфигурационный файл `~/.npmrc`. Это называется `prefix` который можно просмотреть с `npm prefix`.

```
prefix=~/.npm-global-modules
```

Это будет использовать префикс всякий раз, когда вы запускаете `npm install -g`. Вы также можете использовать `npm install --prefix ~/.npm-global-modules` чтобы установить префикс при установке. Если префикс совпадает с вашей конфигурацией, вам не нужно использовать `-g`.

Чтобы использовать глобально установленный модуль, он должен быть на вашем пути:

```
export PATH=$PATH:~/.npm-global-modules/bin
```

Теперь, когда вы запускаете `npm install -g gulp-cli` вы сможете использовать `gulp`.

**Примечание.** Когда вы `npm install` (без `-g`), префикс будет каталогом с `package.json` или текущим каталогом, если нигде не найдено в иерархии. Это также создает каталог `node_modules/.bin` который имеет исполняемые файлы. Если вы хотите использовать исполняемый файл, специфичный для проекта, нет необходимости использовать `npm install -g`. Вы можете использовать его в `node_modules/.bin`.

## Связывание проектов для более быстрой отладки и разработки

Построение зависимостей проекта иногда может быть утомительной задачей. Вместо публикации версии пакета для NPM и установки зависимости для проверки изменений используйте `npm link`. `npm link` создает символическую ссылку, поэтому последний код может быть протестирован в локальной среде. Это упрощает тестирование глобальных инструментов и зависимостей проектов, позволяя запустить последний код перед выпуском опубликованной версии.

## Текст справки

```
NAME
  npm-link - Symlink a package folder

SYNOPSIS
  npm link (in package dir)
  npm link [<@scope>/]<pkg>[@<version>]

alias: npm ln
```

## Шаги для связывания зависимостей проекта

При создании ссылки зависимостей обратите внимание, что имя пакета - это то, что будет указано в родительском проекте.

1. CD в каталог зависимостей (например: `cd ../my-dep`)
2. `npm link`

3. CD в проект, который будет использовать зависимость

4. `npm link my-dep` или если namespaced `npm link @namespace/my-dep`

## Шаги по связыванию глобального инструмента

1. CD в каталог проекта (например: `cd eslint-watch` )

2. `npm link`

3. Использовать инструмент

4. `esw --quiet`

## Проблемы, которые могут возникнуть

Связывание проектов иногда может вызывать проблемы, если уже установлена зависимость или глобальный инструмент. `npm uninstall (-g) <pkg>` а затем запуск `npm link` обычно разрешает любые проблемы, которые могут возникнуть.

Прочитайте НПМ онлайн: <https://riptutorial.com/ru/node-js/topic/482/нпм>

# глава 69: Обещания Bluebird

## Examples

### Преобразование библиотеки узлов в Promises

```
const Promise = require('bluebird'),
      fs = require('fs')

Promise.promisifyAll(fs)

// now you can use promise based methods on 'fs' with the Async suffix
fs.readFileAsync('file.txt').then(contents => {
  console.log(contents)
}).catch(err => {
  console.error('error reading', err)
})
```

### Функциональные обещания

#### Пример карты:

```
Promise.resolve([ 1, 2, 3 ]).map(el => {
  return Promise.resolve(el * el) // return some async operation in real world
})
```

#### Пример фильтра:

```
Promise.resolve([ 1, 2, 3 ]).filter(el => {
  return Promise.resolve(el % 2 === 0) // return some async operation in real world
}).then(console.log)
```

#### Пример сокращения:

```
Promise.resolve([ 1, 2, 3 ]).reduce((prev, curr) => {
  return Promise.resolve(prev + curr) // return some async operation in real world
}).then(console.log)
```

### Корутины (генераторы)

```
const promiseReturningFunction = Promise.coroutine(function* (file) {
  const data = yield fs.readFileAsync(file) // this returns a Promise and resolves to the file contents

  return data.toString().toUpperCase()
})

promiseReturningFunction('file.txt').then(console.log)
```

## Автоматическое удаление ресурсов (Promise.using)

```
function somethingThatReturnsADisposableResource() {
  return getSomeResourceAsync(...).disposer(resource => {
    resource.dispose()
  })
}

Promise.using(somethingThatReturnsADisposableResource(), resource => {
  // use the resource here, the disposer will automatically close it when Promise.using exits
})
```

## Выполнение в серии

```
Promise.resolve([1, 2, 3])
  .mapSeries(el => Promise.resolve(el * el)) // in real world, use Promise returning async
  function
  .then(console.log)
```

Прочитайте Обещания Bluebird онлайн: <https://riptutorial.com/ru/node-js/topic/6728/обещания-bluebird>

# глава 70: Обработка запроса POST в Node.js

## замечания

Node.js использует [потoki](#) для обработки входящих данных.

Цитируя из документов,

Поток представляет собой абстрактный интерфейс для работы с потоковыми данными в Node.js. Модуль потока предоставляет базовый API, который упрощает сбор объектов, реализующих интерфейс потока.

Чтобы обрабатывать запрос в запросе POST, используйте объект `request`, который является читаемым потоком. Потоки данных испускаются как события `data` в объекте `request`.

```
request.on('data', chunk => {
  buffer += chunk;
});
request.on('end', () => {
  // POST request body is now available as `buffer`
});
```

Просто создайте пустую строку буфера и добавьте данные буфера, полученные по событиям `data`.

## НОТА

1. Буферные данные, полученные в событиях `data` имеют тип [Buffer](#)
2. Создайте новую строку буфера для сбора буферизованных данных из событий данных **для каждого запроса**, т. Е. Создайте `buffer` строку внутри обработчика запроса.

## Examples

### Пример node.js-сервера, который просто обрабатывает запросы POST

```
'use strict';

const http = require('http');

const PORT = 8080;
const server = http.createServer((request, response) => {
  let buffer = '';
```

```
request.on('data', chunk => {
  buffer += chunk;
});
request.on('end', () => {
  const responseString = `Received string ${buffer}`;
  console.log(`Responding with: ${responseString}`);
  response.writeHead(200, "Content-Type: text/plain");
  response.end(responseString);
});
}).listen(PORT, () => {
  console.log(`Listening on ${PORT}`);
});
```

Прочитайте [Обработка запроса POST в Node.js онлайн: https://riptutorial.com/ru/node-js/topic/5676/обработка-запроса-post-в-node-js](https://riptutorial.com/ru/node-js/topic/5676/обработка-запроса-post-в-node-js)

# глава 71: Обработка исключений

## Examples

### Исправление обработки в Node.Js

Node.js имеет 3 основных способа обработки исключений / ошибок:

1. блок **try- catch**
2. **ошибка** в качестве первого аргумента для `callback`
3. `emit` событие **ошибки** с помощью `eventEmitter`

**try-catch** используется для обнаружения исключений, возникающих при выполнении синхронного кода. Если вызывающий (или вызываемый абонент, ...) использовал `try / catch`, то они могут поймать ошибку. Если ни один из вызывающих абонентов не имеет попыток, чем программа выйдет из строя.

Если использование `try-catch` в `async`-операции и исключение было выбрано из обратного вызова метода `async`, то оно не попадет в `try-catch`. Чтобы поймать исключение из обратного вызова асинхронной операции, предпочтительно использовать *обещания*.

Пример, чтобы лучше понять его

```
// ** Example - 1 **
function doSomeSynchronousOperation(req, res) {
  if(req.body.username === ''){
    throw new Error('User Name cannot be empty!');
  }
  return true;
}

// calling the method above
try {
  // synchronous code
  doSomeSynchronousOperation(req, res)
} catch(e) {
  //exception handled here
  console.log(e.message);
}

// ** Example - 2 **
function doSomeAsynchronousOperation(req, res, cb) {
  // imitating async operation
  return setTimeout(function(){
    cb(null, []);
  },1000);
}

try {
  // asynchronous code
  doSomeAsynchronousOperation(req, res, function(err, rs){
    throw new Error("async operation exception");
  })
}
```



```
} catch(e) {
  // Exception will not get handled here
  console.log(e.message);
}
// The exception is unhandled and hence will cause application to break
```

**обратные вызовы** в основном используются в Node.js, поскольку обратный вызов выполняет асинхронное событие. Пользователь передает вам функцию (обратный вызов), и вы вызываете ее когда-нибудь позже, когда асинхронная операция завершается. Обычная модель заключается в том, что обратный вызов вызывается как *обратный вызов (ошибка, результат)*, где только один из ошибок и результатов не равен нулю, в зависимости от того, была ли операция успешной или неудачной.

```
function doSomeAsynchronousOperation(req, res, callback) {
  setTimeout(function() {
    return callback(new Error('User Name cannot be empty'));
  }, 1000);
  return true;
}

doSomeAsynchronousOperation(req, res, function(err, result) {
  if (err) {
    //exception handled here
    console.log(err.message);
  }

  //do some stuff with valid data
});
```

**emit** Для более сложных случаев вместо использования обратного вызова сама функция может возвращать объект `EventEmitter`, и ожидается, что вызывающий абонент будет прослушивать события ошибки на эмиттере.

```
const EventEmitter = require('events');

function doSomeAsynchronousOperation(req, res) {
  let myEvent = new EventEmitter();

  // runs asynchronously
  setTimeout(function() {
    myEvent.emit('error', new Error('User Name cannot be empty'));
  }, 1000);

  return myEvent;
}

// Invoke the function
let event = doSomeAsynchronousOperation(req, res);

event.on('error', function(err) {
  console.log(err);
});

event.on('done', function(result) {
  console.log(result); // true
```

```
});
```

## Управление незатронутыми исключениями

Поскольку Node.js работает на одном процессе, неработающие исключения - это проблема, о которой следует знать при разработке приложений.

## Без исключения

Большинство людей позволяют серверу (узлам) node.js незаметно проглатывать ошибки.

- Тишина обработки исключения

```
process.on('uncaughtException', function (err) {  
  console.log(err);  
});
```

**Это плохо**, он будет работать, но:

- Корневая причина остается неизвестной, так как это не будет способствовать разрешению того, что вызвало исключение (ошибка).
- В случае, если соединение с базой данных (пул) закрывается по какой-либо причине, это приведет к постоянному распространению ошибок, что означает, что сервер будет работать, но он не будет повторно подключаться к db.

---

## Возврат в начальное состояние

В случае «uncaughtException» хорошо перезапустить сервер и вернуть его в **исходное состояние**, где мы знаем, что он будет работать. Исключение регистрируется, приложение завершается, но поскольку он будет запущен в контейнере, который будет следить за тем, чтобы сервер работал, мы достигнем перезапуска сервера (вернемся к исходному рабочему состоянию).

- Установка вечно (или другого инструмента CLI, чтобы убедиться, что сервер узла работает непрерывно)

```
npm install forever -g
```

- Запуск сервера навсегда

```
forever start app.js
```

Причина, почему она началась и почему мы используем вечно, - это то, что

сервер будет **завершен** навсегда, и процесс снова запустит сервер.

- **Перезапуск сервера**

```
process.on('uncaughtException', function (err) {
  console.log(err);

  // some logging mechanism
  // ....

  process.exit(1); // terminates process
});
```

---

На стороне заметки был также способ обработки исключений с **кластерами и доменами** .

Домены больше не рекомендуют [здесь](#) .

## Ошибки и обещания

Обещания обрабатывают ошибки по-разному для синхронного или обратного кода.

```
const p = new Promise(function (resolve, reject) {
  reject(new Error('Oops'));
});

// anything that is `reject`ed inside a promise will be available through catch
// while a promise is rejected, `.then` will not be called
p
  .then(() => {
    console.log("won't be called");
  })
  .catch(e => {
    console.log(e.message); // output: Oops
  })
  // once the error is caught, execution flow resumes
  .then(() => {
    console.log('hello!'); // output: hello!
  });
```

в настоящее время ошибки, сбрасываемые в обещание, которое не попало, приводят к проглатыванию ошибки, что может затруднить отслеживание ошибки. Это можно [решить](#), используя инструменты linting, такие как [eslint](#), или гарантируя, что вы всегда имеете предложение `catch` .

Это поведение устарело [в узле 8](#) в пользу прекращения процесса узла.

Прочитайте [Обработка исключений онлайн: https://riptutorial.com/ru/node-js/topic/2819/обработка-исключений](https://riptutorial.com/ru/node-js/topic/2819/обработка-исключений)

# глава 72: Обратный звонок для обещания

## Examples

### Продвижение обратного вызова

Обратный вызов на основе:

```
db.notification.email.find({subject: 'promisify callback'}, (error, result) => {
  if (error) {
    console.log(error);
  }

  // normal code here
});
```

Это использует метод `promisifyAll` от `bluebird`, чтобы обезопасить код, основанный на обратном вызове, как описано выше. `bluebird` выполнит обеими версиями всех методов в объекте, эти имена, основанные на методах обещания, добавили к ним `Async`:

```
let email = bluebird.promisifyAll(db.notification.email);

email.findAsync({subject: 'promisify callback'}).then(result => {

  // normal code here
})
.catch(console.error);
```

Если нужно прокомментировать только определенные методы, просто используйте его обещание:

```
let find = bluebird.promisify(db.notification.email.find);

find({locationId: 168}).then(result => {

  // normal code here
});
.catch(console.error);
```

Существуют некоторые библиотеки (например, `MassiveJS`), которые не могут быть обещаны, если непосредственный объект метода не передается во втором параметре. В этом случае просто передайте непосредственный объект метода, который должен быть пролонгирован по второму параметру и заключен в свойство контекста.

```
let find = bluebird.promisify(db.notification.email.find, { context: db.notification.email });

find({locationId: 168}).then(result => {

  // normal code here
```

```
});  
.catch(console.error);
```

## Вручную пролонгировать обратный вызов

Иногда может потребоваться ручная проработка функции обратного вызова. Это может быть в случае, когда обратный вызов не соответствует стандартным формам [ошибок](#) или требуется дополнительная логика для того, чтобы обещать:

Пример с [fs.exists](#) (путь, обратный вызов) :

```
var fs = require('fs');  
  
var existsAsync = function(path) {  
  return new Promise(function(resolve, reject) {  
    fs.exists(path, function(exists) {  
      // exists is a boolean  
      if (exists) {  
        // Resolve successfully  
        resolve();  
      } else {  
        // Reject with error  
        reject(new Error('path does not exist'));  
      }  
    });  
  });  
  
  // Use as a promise now  
  existsAsync('/path/to/some/file').then(function() {  
    console.log('file exists!');  
  }).catch(function(err) {  
    // file does not exist  
    console.error(err);  
  });  
};
```

## setTimeout promisified

```
function wait(ms) {  
  return new Promise(function (resolve, reject) {  
    setTimeout(resolve, ms)  
  })  
}
```

Прочитайте [Обратный звонок для обещания онлайн](https://riptutorial.com/ru/node-js/topic/2346/обратный-звонок-для-обещания): <https://riptutorial.com/ru/node-js/topic/2346/обратный-звонок-для-обещания>

---

# глава 73: Основы проектирования Node.js

## Examples

### Философия Node.js

#### Малый ядро , малый модуль : -

Создавайте небольшие и единичные модули не только с точки зрения размера кода, но также и с точки зрения объема, который служит единой цели

```
a - "Small is beautiful"
b - "Make each program do one thing well."
```

#### Схема реактора

The Reactor Pattern - это сердце асинхронного характера `node.js`. Допустим, что система будет реализована как однопоточный процесс с серией генераторов событий и обработчиков событий с помощью цикла событий, который работает непрерывно.

#### Неблокирующий механизм ввода / вывода Node.js - libuv -

**Шаблон наблюдателя** (EventEmitter) поддерживает список иждивенцев / наблюдателей и уведомляет их

```
var events = require('events');
var eventEmitter = new events.EventEmitter();

var ringBell = function ringBell()
{
  console.log('tring tring tring');
}
eventEmitter.on('doorOpen', ringBell);

eventEmitter.emit('doorOpen');
```

Прочитайте Основы проектирования Node.js онлайн: <https://riptutorial.com/ru/node-js/topic/6274/основы-проектирования-node-js>

---

# глава 74: Отладка приложения Node.js

## Examples

Отладчик Core node.js и инспектор узлов

---

## Использование основного отладчика

Node.js предоставляет сборку в неграфической утилите для отладки. Чтобы начать сборку в отладчике, запустите приложение с помощью этой команды:

```
node debug filename.js
```

Рассмотрим следующее простое приложение Node.js, содержащееся в `debugDemo.js`

```
'use strict';

function addTwoNumber(a, b){
  // function returns the sum of the two numbers
  debugger
  return a + b;
}

var result = addTwoNumber(5, 9);
console.log(result);
```

`debugger` **КЛЮЧЕВЫХ СЛОВ** ОСТАНОВИТ ОТЛАДЧИК В ЭТОЙ ТОЧКЕ КОДА.

## Ссылка на команду

### 1. Шагая

```
cont, c - Continue execution
next, n - Step next
step, s - Step in
out, o - Step out
```

### 2. Контрольные точки

```
setBreakpoint(), sb() - Set breakpoint on current line
setBreakpoint(line), sb(line) - Set breakpoint on specific line
```

Для отладки приведенного выше кода выполните следующую команду

```
node debug debugDemo.js
```

После выполнения вышеперечисленных команд вы увидите следующий вывод. Чтобы выйти из интерфейса отладчика, введите `process.exit()`

```
ankuranand:~/workspace/nodejs/nodejsDebugging $ node debug debugDemo.js
< Debugger listening on port 5858
debug> . ok
break in debugDemo.js:3
  1 // A Demo Code Showing the basic capabilities of the nodejs  debugging module
  2
> 3 'use strict';
  4
  5 function addTwoNumber(a, b){
debug> n
break in debugDemo.js:11
  9 }
 10
>11 let result = addTwoNumber(5, 9);
 12 console.log(result);
 13
debug> c
break in debugDemo.js:7
  5 function addTwoNumber(a, b){
  6 // function returns the sum of the two numbers
> 7 debugger
  8   return a + b;
  9 }
debug> c
< 14
debug> process.exit()
ankuranand:~/workspace/nodejs/nodejsDebugging $
```

Используйте команду `watch(expression)` чтобы добавить переменную или выражение, значение которого вы хотите посмотреть и `restart` чтобы перезапустить приложение и отладить его.

Используйте `repl` для ввода кода в интерактивном режиме. Режим `repl` имеет тот же контекст, что и строка, которую вы отлаживаете. Это позволяет вам просматривать содержимое переменных и проверять строки кода. Нажмите `Ctrl+C` чтобы оставить отладочную замену.

---

## Использование встроенного инспектора узлов

v6.3.0

Вы можете запустить узел, [встроенный в инспектор v8!](#) Плагин [узла-инспектора](#) больше не нужен.



Просто передайте флаг инспектора, и вам будет предоставлен URL-адрес инспектора

```
node --inspect server.js
```

---

## Использование указателя узла

Установите инспектор узлов:

```
npm install -g node-inspector
```

Запустите приложение с помощью команды node-debug:

```
node-debug filename.js
```

После этого нажмите в Chrome:

```
http://localhost:8080/debug?port=5858
```

Иногда порт 8080 может быть недоступен на вашем компьютере. Вы можете получить следующую ошибку:

Не удается запустить сервер на 0.0.0.0:8080. Ошибка: прослушивание EACCES.

В этом случае запустите инспектор узлов на другом порту, используя следующую команду.

```
$node-inspector --web-port=6500
```

Вы увидите что-то вроде этого:

```
1 // A Demo Code Showing the basic capabilities of the nodejs debugging module
2
3 'use strict';
4
5 function addTwoNumber(a, b){
6 // function returns the sum of the two numbers
7   return a + b;
8 }
9
10 var result = addTwoNumber(5, 9);
11 console.log(result);
12
```

10:1 JavaScript Spaces: 4

Прочитайте Отладка приложения Node.js онлайн: <https://riptutorial.com/ru/node-js/topic/5900/отладка-приложения-node-js>

---

# глава 75: Отправить веб-уведомление

## Examples

### Отправить веб-уведомление с помощью GCM (Google Cloud Messaging System)

Такой пример знает широкое распространение среди **PWA** (Progressive Web Applications), и в этом примере мы собираемся отправить простое **бэкэнд**- уведомление с помощью **NodeJS** и **ES6**

1. Установка модуля Node-GCM: `npm install node-gcm`
2. Установите Socket.io: `npm install socket.io`
3. Создайте приложение GCM Enabled с помощью [Google Консоли](#).
4. Grabe ваш идентификатор приложения GCM (он нам понадобится позже)
5. Grabe ваш секретный код приложения GCM.
6. Откройте свой любимый редактор кода и добавьте следующий код:

```
'use strict';

const express = require('express');
const app = express();
const gcm = require('node-gcm');
app.io = require('socket.io')();

// [*] Configuring our GCM Channel.
const sender = new gcm.Sender('Project Secret');
const regTokens = [];
let message = new gcm.Message({
  data: {
    key1: 'msg1'
  }
});

// [*] Configuring our static files.
app.use(express.static('public/'));

// [*] Configuring Routes.
app.get('/', (req, res) => {
  res.sendFile(__dirname + '/public/index.html');
});

// [*] Configuring our Socket Connection.
app.io.on('connection', socket => {
  console.log('we have a new connection ...');
  socket.on('new_user', (reg_id) => {
```

```

    // [*] Adding our user notification registration token to our list typically
    // hidden in a secret place.
    if (regTokens.indexOf(reg_id) === -1) {
        regTokens.push(reg_id);

        // [*] Sending our push messages
        sender.send(message, {
            registrationTokens: regTokens
        }, (err, response) => {
            if (err) console.error('err', err);
            else console.log(response);
        });
    }
})
});

module.exports = app

```

PS: Я использую здесь специальный хак, чтобы Socket.io работал с Express, потому что просто он не работает за пределами коробки.

Теперь создайте файл **.json** и назовите его: **Manifest.json** , откройте его и **пропустите** следующее:

```

{
  "name": "Application Name",
  "gcm_sender_id": "GCM Project ID"
}

```

Закройте его и сохраните в своем каталоге **ROOT** приложения.

PS: файл Manifest.json должен находиться в корневом каталоге, иначе он не будет работать.

В приведенном выше коде я делаю следующее:

1. Я настроил и отправил обычную страницу index.html, которая также будет использовать socket.io.
2. Я слушаю событие **соединения**, запущенное из **front-end** ака моей **страницы index.html** (он будет запущен, как только новый клиент успешно подключится к нашей предварительно определенной ссылке)
3. Я отправляю специальный токен в качестве **регистрационного токена** из моего index.html через событие socket.io **new\_user** , таким токеном будет наш уникальный пользовательский код доступа, и каждый код генерируется обычно из поддерживающего браузера **API веб-уведомлений** (подробнее [Вот](#) .
4. Я просто использую модуль **node-gcm** для отправки моего уведомления, которое будет обработано и показано позже, используя **Service Workers** `.

Это с точки зрения **NodeJS** . в других примерах я покажу, как мы можем отправлять пользовательские данные, значки ..etc в нашем push-сообщении.

PS: здесь вы можете найти полную рабочую демоверсию .

Прочитайте [Отправить веб-уведомление онлайн: https://riptutorial.com/ru/node-js/topic/6333/отправить-веб-уведомление](https://riptutorial.com/ru/node-js/topic/6333/отправить-веб-уведомление)

# глава 76: Отправка потока файлов клиенту

## Examples

### Использование fs И pipe Для потоковой передачи статических файлов с сервера

Хорошая услуга VOD (Video On Demand) должна начинаться с основ. Допустим, у вас есть каталог на вашем сервере, который не является общедоступным, но через какой-то портал или рауwall вы хотите разрешить пользователям доступ к вашим медиа.

```
var movie = path.resolve('./public/' + req.params.filename);

fs.stat(movie, function (err, stats) {

  var range = req.headers.range;

  if (!range) {

    return res.sendStatus(416);

  }

  //Chunk logic here
  var positions = range.replace(/bytes=/, "").split("-");
  var start = parseInt(positions[0], 10);
  var total = stats.size;
  var end = positions[1] ? parseInt(positions[1], 10) : total - 1;
  var chunksize = (end - start) + 1;

  res.writeHead(206, {

    'Transfer-Encoding': 'chunked',

    "Content-Range": "bytes " + start + "-" + end + "/" + total,

    "Accept-Ranges": "bytes",

    "Content-Length": chunksize,

    "Content-Type": mime.lookup(req.params.filename)

  });

  var stream = fs.createReadStream(movie, { start: start, end: end, autoClose: true
})

  .on('end', function () {

    console.log('Stream Done');

  })

  .on("error", function (err) {
```

```
        res.end(err);
    })
    .pipe(res, { end: true });
});
```

Вышеприведенный фрагмент - это основной план того, как вы хотите передать свое видео клиенту. Логика блоков зависит от множества факторов, включая сетевой трафик и задержку. Важно балансировать размер патрона и количество.

Наконец, вызов `.pipe` позволяет узлу.js знать, что соединение открыто с сервером и при необходимости посылать дополнительные куски.

## Потоковая передача с использованием fluent-ffmpeg

Вы также можете использовать `fluent-ffmpeg` для преобразования файлов `.mp4` в файлы `.flv` или другие типы:

```
res.contentType ('FLV');
```

```
var pathToMovie = './public/' + req.params.filename;
var proc = ffmpeg(pathToMovie)
    .preset('flashvideo')
    .on('end', function () {
        console.log('Stream Done');
    })
    .on('error', function (err) {
        console.log('an error happened: ' + err.message);
        res.send(err.message);
    })
    .pipe(res, { end: true });
```

Прочитайте Отправка потока файлов клиенту онлайн: <https://riptutorial.com/ru/node-js/topic/6994/отправка-потока-файлов-клиенту>

---

# глава 77: по металлу

## Examples

### Создайте простой блог

Предполагая, что у вас установлены и доступны узлы и npm, создайте папку проекта с допустимым `package.json`. Установите необходимые зависимости:

```
npm install --save-dev metalsmith metalsmith-in-place handlebars
```

Создайте файл с именем `build.js` в корне вашей папки проекта, содержащий следующее:

```
var metalsmith = require('metalsmith');
var handlebars = require('handlebars');
var inPlace = require('metalsmith-in-place');

Metalsmith(__dirname)
  .use(inPlace('handlebars'))
  .build(function(err) {
    if (err) throw err;
    console.log('Build finished!');
  });
```

Создайте папку под названием `src` в корне вашей папки проекта. Создайте `index.html` в `src`, содержащий следующее:

```
---
title: My awesome blog
---
<h1>{{ title }}</h1>
```

Запуск `node build.js` теперь будет создавать все файлы в `src`. После выполнения этой команды у вас будет `index.html` в вашей папке сборки со следующим содержимым:

```
<h1>My awesome blog</h1>
```

Прочитайте по металлу онлайн: <https://riptutorial.com/ru/node-js/topic/6111/по-металлу>



# глава 78: Подключиться к MongoDB

## Вступление

MongoDB - это бесплатная и открытая кросс-платформенная документарно-ориентированная программа баз данных. Классифицируется как программа базы данных NoSQL, MongoDB использует JSON-подобные документы со схемами.

Для получения дополнительной информации перейдите на [страницу](https://www.mongodb.com/) <https://www.mongodb.com/>

## Синтаксис

- `MongoClient.connect ('mongodb: //127.0.0.1: 27017 / crud', function (err, db) { // здесь делать что-то здесь});`

## Examples

### Простой пример подключения mongoDB от Node.JS

```
MongoClient.connect ('mongodb://localhost:27017/myNewDB', function (err, db) {
  if(err)
    console.log("Unable to connect DB. Error: " + err)
  else
    console.log('Connected to DB');

  db.close();
});
```

myNewDB - это имя БД, если оно не существует в базе данных, оно автоматически создаст ЭТОТ ВЫЗОВ.

### Простой способ подключения mongoDB с ядром Node.JS

```
var MongoClient = require('mongodb').MongoClient;

//connection with mongoDB
MongoClient.connect ("mongodb://localhost:27017/MyDb", function (err, db) {
  //check the connection
  if(err){
    console.log("connection failed.");
  }else{
    console.log("successfully connected to mongoDB.");
  }
});
```

Прочитайте Подключиться к Mongoddb онлайн: <https://riptutorial.com/ru/node-js/topic/6280/>

[подключиться-к-mongodb](#)

# глава 79: Постоянно поддерживать приложение узла

## Examples

### Использовать PM2 в качестве менеджера процессов

PM2 позволяет запускать скрипты nodejs навсегда. В случае сбоя вашего приложения PM2 также перезапустит его для вас.

Установите PM2 глобально для управления экземплярами nodejs

```
npm install pm2 -g
```

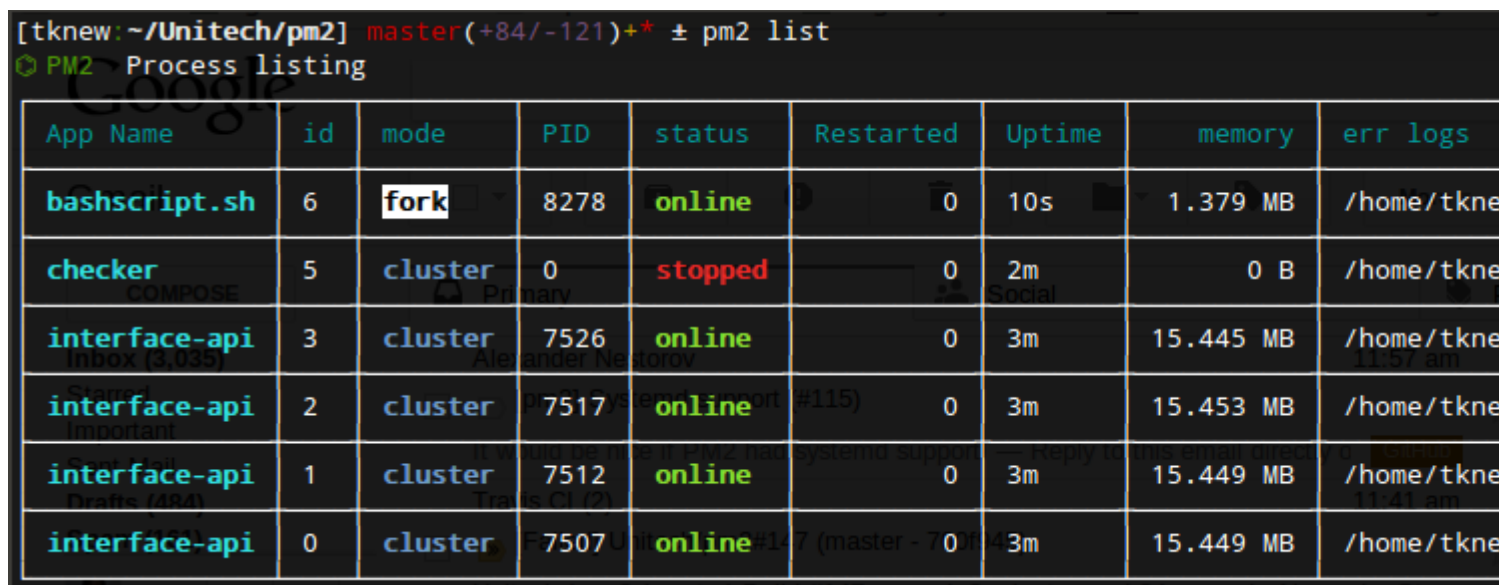
Перейдите в каталог, в котором находится ваш скрипт nodejs, и выполните следующую команду каждый раз, когда вы хотите запустить экземпляр nodejs, который будет контролироваться pm2:

```
pm2 start server.js --name "app1"
```

## Полезные команды для мониторинга процесса

1. Список всех экземпляров nodejs, управляемых pm2

```
pm2 list
```



```
[tknew: ~/Unitech/pm2] master(+84/-121)+* ± pm2 list
PM2 Process listing
```

App Name	id	mode	PID	status	Restarted	Uptime	memory	err logs
bashscript.sh	6	fork	8278	online	0	10s	1.379 MB	/home/tkne
checker	5	cluster	0	stopped	0	2m	0 B	/home/tkne
interface-api	3	cluster	7526	online	0	3m	15.445 MB	/home/tkne
interface-api	2	cluster	7517	online	0	3m	15.453 MB	/home/tkne
interface-api	1	cluster	7512	online	0	3m	15.449 MB	/home/tkne
interface-api	0	cluster	7507	online	0	43m	15.449 MB	/home/tkne

2. Остановить конкретный экземпляр nodejs

```
pm2 stop <instance named>
```

### 3. Удалить конкретный экземпляр nodejs

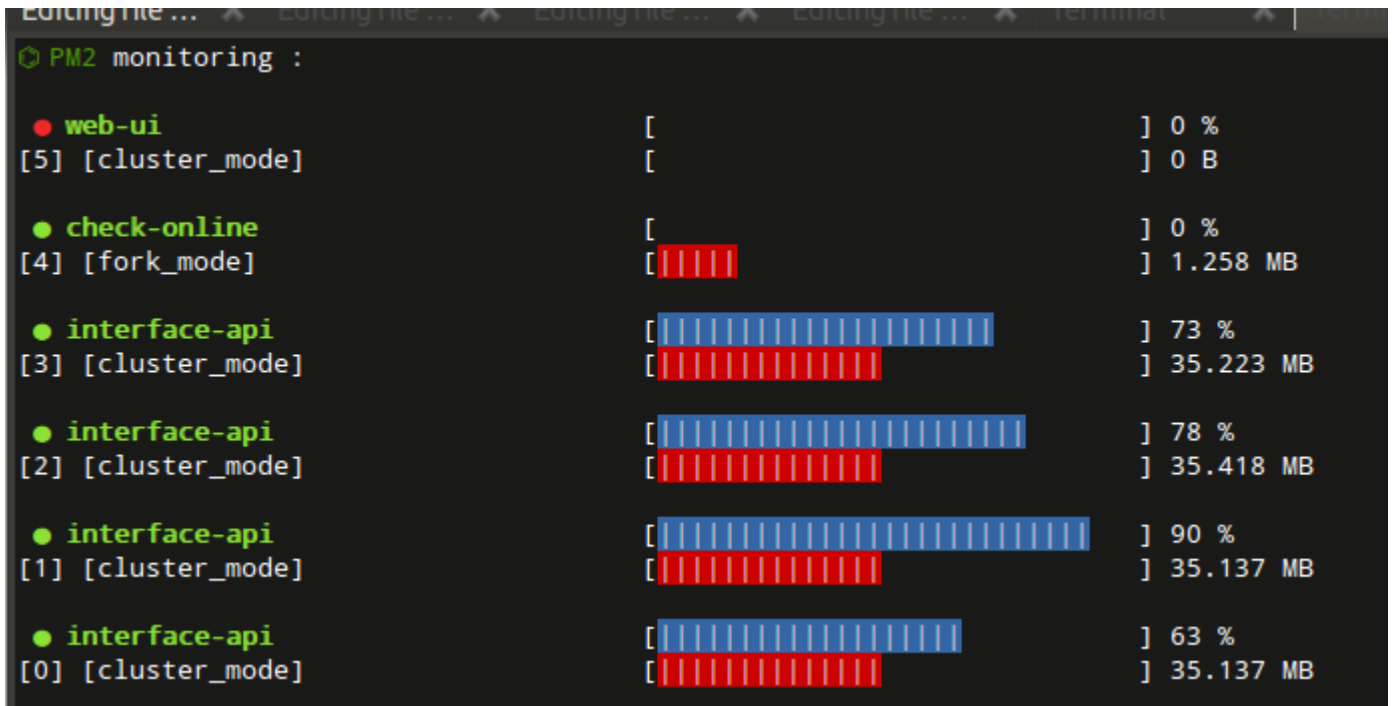
```
pm2 delete <instance name>
```

### 4. Перезапустить конкретный экземпляр nodejs

```
pm2 restart <instance name>
```

### 5. Мониторинг всех экземпляров nodejs

```
pm2 monit
```



### 6. Остановка pm2

```
pm2 kill
```

### 7. В отличие от перезапуска, которая убивает и перезапускает процесс, перезагрузка достигает 0-секундного времени перезагрузки

```
pm2 reload <instance name>
```

### 8. Просмотр журналов

```
pm2 logs <instance_name>
```

## Запуск и остановка демон Forever

Чтобы начать процесс:

```
$ forever start index.js  
warn: --minUptime not set. Defaulting to: 1000ms  
warn: --spinSleepTime not set. Your script will exit if it does not stay up for at least
```

```
1000ms
info:    Forever processing file: index.js
```

## Список запущенных экземпляров Forever:

```
$ forever list
info:    Forever processes running

|data: | index | uid | command          | script          |forever pid|id  | logfile
|uptime |         |         |         |         |         |         |         |
|-----|-----|-----|-----|-----|-----|-----|-----|
---|-----|
|data: | [0]   | f4Kt | /usr/bin/nodejs | src/index.js |2131   |
2146 | /root/.forever/f4Kt.log | 0:0:0:11.485 |
```

## Остановите первый процесс:

```
$ forever stop 0
$ forever stop 2146
$ forever stop --uid f4Kt
$ forever stop --pidFile 2131
```

## Непрерывный запуск с nohup

Альтернативой навсегда для Linux является nohup.

Чтобы запустить экземпляр nohup

1. cd в папку `app.js` или `www`
2. запустите `nohup nodejs app.js &`

Чтобы убить процесс

1. запустить `ps -ef|grep nodejs`
2. `kill -9 <the process number>`

## Технологическое соглашение с Forever

### Монтаж

```
npm install forever -g
cd /node/project/directory
```

### Обычай

```
forever start app.js
```

Прочитайте Постоянно поддерживать приложение узла онлайн:

<https://riptutorial.com/ru/node-js/topic/2820/постоянно-поддерживать-приложение-узла>

# глава 80: Проблемы с производительностью

## Examples

### Обработка длинных запросов с помощью узла

Поскольку Node является однопоточным, существует необходимость обходного пути, если речь идет о долгосрочных расчетах.

**Примечание:** это пример «готов к запуску». Просто не забудьте получить jQuery и установить необходимые модули.

#### Основная логика этого примера:

1. Клиент отправляет запрос на сервер.
2. Сервер запускает процедуру в отдельном экземпляре узла и отправляет немедленный ответ обратно с соответствующим идентификатором задачи.
3. Клиент постоянно отправляет проверки на сервер для обновления статуса данного идентификатора задачи.

#### Структура проекта:

```
project
├── package.json
├── index.html
├── js
│   ├── main.js
│   └── jquery-1.12.0.min.js
└── srv
    ├── app.js
    ├── models
    │   └── task.js
    └── tasks
        └── data-processor.js
```

#### app.js:

```
var express      = require('express');
var app          = express();
var http         = require('http').Server(app);
var mongoose     = require('mongoose');
var bodyParser   = require('body-parser');

var childProcess= require('child_process');
```

```

var Task          = require('./models/task');

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

app.use(express.static(__dirname + '/../'));

app.get('/', function(request, response){
    response.render('index.html');
});

//route for the request itself
app.post('/long-running-request', function(request, response){
    //create new task item for status tracking
    var t = new Task({ status: 'Starting ...' });

    t.save(function(err, task){
        //create new instance of node for running separate task in another thread
        taskProcessor = childProcess.fork('./srv/tasks/data-processor.js');

        //process the messages coming from the task processor
        taskProcessor.on('message', function(msg){
            task.status = msg.status;
            task.save();
        }).bind(this));

        //remove previously opened node instance when we finished
        taskProcessor.on('close', function(msg){
            this.kill();
        });

        //send some params to our separate task
        var params = {
            message: 'Hello from main thread'
        };

        taskProcessor.send(params);
        response.status(200).json(task);
    });
});

//route to check is the request is finished the calculations
app.post('/is-ready', function(request, response){
    Task
        .findById(request.body.id)
        .exec(function(err, task){
            response.status(200).json(task);
        });
});

mongoose.connect('mongodb://localhost/test');
http.listen('1234');

```

## task.js:

```

var mongoose      = require('mongoose');

var taskSchema = mongoose.Schema({
    status: {
        type: String
    }
});

```



```

    }
  });

mongoose.model('Task', taskSchema);

module.exports = mongoose.model('Task');

```

## Данные-processor.js:

```

process.on('message', function(msg){
  init = function(){
    processData(msg.message);
  }.bind(this)();

  function processData(message){
    //send status update to the main app
    process.send({ status: 'We have started processing your data.' });

    //long calculations ..
    setTimeout(function(){
      process.send({ status: 'Done!' });

      //notify node, that we are done with this task
      process.disconnect();
    }, 5000);
  }
});

process.on('uncaughtException',function(err){
  console.log("Error happened: " + err.message + "\n" + err.stack + ".\n");
  console.log("Gracefully finish the routine.");
});

```

## index.html:

```

<!DOCTYPE html>
<html>
  <head>
    <script src="./js/jquery-1.12.0.min.js"></script>
    <script src="./js/main.js"></script>
  </head>
  <body>
    <p>Example of processing long-running node requests.</p>
    <button id="go" type="button">Run</button>

    <br />

    <p>Log:</p>
    <textarea id="log" rows="20" cols="50"></textarea>
  </body>
</html>

```

## main.js:

```

$(document).on('ready', function(){

  $('#go').on('click', function(e){

```

```

//clear log
$("#log").val('');

$.post("/long-running-request", {some_params: 'params' })
  .done(function(task){
    $("#log").val( $("#log").val() + '\n' + task.status);

    //function for tracking the status of the task
    function updateStatus(){
      $.post("/is-ready", {id: task._id })
        .done(function(response){
          $("#log").val( $("#log").val() + '\n' + response.status);

          if(response.status != 'Done!'){
            checkTaskTimeout = setTimeout(updateStatus, 500);
          }
        });
    }

    //start checking the task
    var checkTaskTimeout = setTimeout(updateStatus, 100);
  });
});
});

```

### package.json:

```

{
  "name": "nodeProcessor",
  "dependencies": {
    "body-parser": "^1.15.2",
    "express": "^4.14.0",
    "html": "0.0.10",
    "mongoose": "^4.5.5"
  }
}

```

**Отказ от ответственности:** этот пример призван дать вам базовую идею. Чтобы использовать его в производственной среде, ему нужны улучшения.

Прочитайте Проблемы с производительностью онлайн: <https://riptutorial.com/ru/node-js/topic/6325/проблемы-с-производительностью>

---

# глава 81: Проверка подлинности Windows в узле node.js

## замечания

Существует несколько других APIS Active Directory, таких как [activedirectory2](#) и [adldap](#) .

## Examples

### Использование активированного справочника

Пример, приведенный ниже, взят из полных документов, доступных [здесь \(GitHub\)](#) или [здесь \(NPM\)](#) .

---

## Монтаж

```
npm install --save activedirectory
```

---

## ИСПОЛЬЗОВАНИЕ

```
// Initialize
var ActiveDirectory = require('activedirectory');
var config = {
  url: 'ldap://dc.domain.com',
  baseDN: 'dc=domain,dc=com'
};
var ad = new ActiveDirectory(config);
var username = 'john.smith@domain.com';
var password = 'password';
// Authenticate
ad.authenticate(username, password, function(err, auth) {
  if (err) {
    console.log('ERROR: '+JSON.stringify(err));
    return;
  }
  if (auth) {
    console.log('Authenticated!');
  }
  else {
    console.log('Authentication failed!');
  }
});
```

Прочитайте [Проверка подлинности Windows в узле node.js онлайн](#):



---

# глава 82: Производительность Node.js

## Examples

### Event Loop

---

## Пример операции блокировки

```
let loop = (i, max) => {
  while (i < max) i++
  return i
}

// This operation will block Node.js
// Because, it's CPU-bound
// You should be careful about this kind of code
loop(0, 1e+12)
```

---

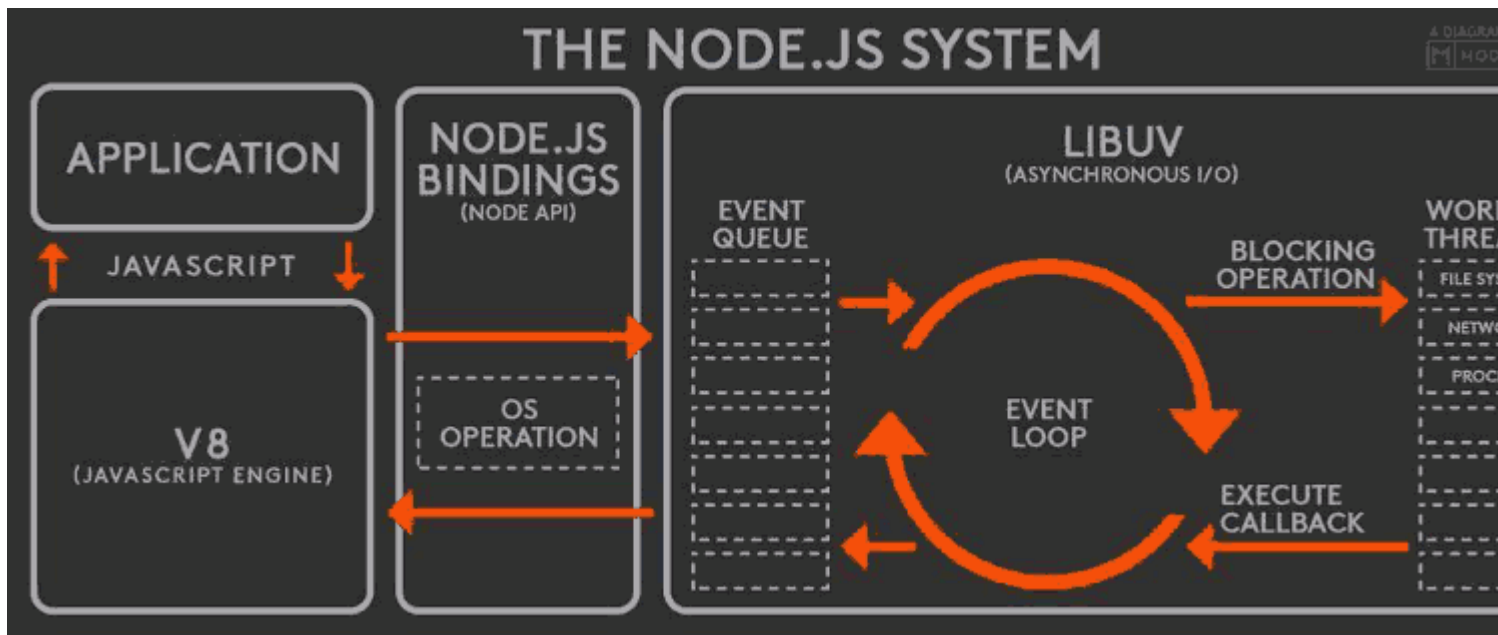
## Пример неблокирующей операции ввода-вывода

```
let i = 0

const step = max => {
  while (i < max) i++
  console.log('i = %d', i)
}

const tick = max => process.nextTick(step, max)

// this will postpone tick run step's while-loop to event loop cycles
// any other IO-bound operation (like filesystem reading) can take place
// in parallel
tick(1e+6)
tick(1e+7)
console.log('this will output before all of tick operations. i = %d', i)
console.log('because tick operations will be postponed')
tick(1e+8)
```



В более простых выражениях Event Loop представляет собой однопоточный механизм очереди, который выполняет код с привязкой к процессору до конца его выполнения и код с привязкой к IO неблокирующим образом.

Однако Node.js под ковром использует многопоточность для некоторых своих операций через библиотеку [libuv](#).

## Рекомендации по производительности

- Неблокирующие операции не будут блокировать очередь и не будут влиять на производительность цикла.
- Однако операции с привязкой к ЦП будут блокировать очередь, поэтому вы должны быть осторожны, чтобы не выполнять операции с ЦП в коде Node.js.

Node.js неблокирует IO, потому что он выгружает работу в ядро операционной системы, и когда операция ввода-вывода поставляет данные ( в виде события ), она будет уведомлять ваш код с вашими входящими обратными вызовами.

### Увеличить maxSockets

## ОСНОВЫ

```
require('http').globalAgent.maxSockets = 25

// You can change 25 to Infinity or to a different value by experimenting
```

Node.js по умолчанию использует `maxSockets = Infinity` в то же время (начиная с [v0.12.0](#)). До Node v0.12.0 по умолчанию был `maxSockets = 5` (см. [V0.11.0](#)). Таким образом, после более

чем 5 запросов они будут поставлены в очередь. Если вы хотите параллелизм, увеличьте это число.

---

## Настройка собственного агента

http API использует « [Глобальный агент](#) ». Вы можете предоставить своего агента. Как это:

```
const http = require('http')
const myGloriousAgent = new http.Agent({ keepAlive: true })
myGloriousAgent.maxSockets = Infinity

http.request({ ..., agent: myGloriousAgent }, ...)
```

---

## Полностью отключить сокет

```
const http = require('http')
const options = {.....}

options.agent = false

const request = http.request(options)
```

---

## Ловушки

- Вы должны сделать то же самое для `https` API, если хотите, чтобы те же эффекты
- Помните, что, например, [AWS](#) будет использовать 50 вместо `Infinity`.

### Включить gzip

```
const http = require('http')
const fs = require('fs')
const zlib = require('zlib')

http.createServer((request, response) => {
  const stream = fs.createReadStream('index.html')
  const acceptsEncoding = request.headers['accept-encoding']

  let encoder = {
    hasEncoder : false,
    contentEncoding: {},
    createEncoder : () => throw 'There is no encoder'
  }

  if (!acceptsEncoding) {
    acceptsEncoding = ''
```

```
}

if (acceptsEncoding.match(/\bdeflate\b/)) {
  encoder = {
    hasEncoder      : true,
    contentEncoding: { 'content-encoding': 'deflate' },
    createEncoder   : zlib.createDeflate
  }
} else if (acceptsEncoding.match(/\bgzip\b/)) {
  encoder = {
    hasEncoder      : true,
    contentEncoding: { 'content-encoding': 'gzip' },
    createEncoder   : zlib.createGzip
  }
}

response.writeHead(200, encoder.contentEncoding)

if (encoder.hasEncoder) {
  stream = stream.pipe(encoder.createEncoder())
}

stream.pipe(response)

}).listen(1337)
```

Прочитайте Производительность Node.js онлайн: <https://riptutorial.com/ru/node-js/topic/9410/производительность-node-js>



# глава 83: Простой API CRUD на основе REST

## Examples

### REST API для CRUD в Express 3+

```
var express = require("express"),
    bodyParser = require("body-parser"),
    server = express();

//body parser for parsing request body
server.use(bodyParser.json());
server.use(bodyParser.urlencoded({ extended: true }));

//temporary store for `item` in memory
var itemStore = [];

//GET all items
server.get('/item', function (req, res) {
  res.json(itemStore);
});

//GET the item with specified id
server.get('/item/:id', function (req, res) {
  res.json(itemStore[req.params.id]);
});

//POST new item
server.post('/item', function (req, res) {
  itemStore.push(req.body);
  res.json(req.body);
});

//PUT edited item in-place of item with specified id
server.put('/item/:id', function (req, res) {
  itemStore[req.params.id] = req.body;
  res.json(req.body);
});

//DELETE item with specified id
server.delete('/item/:id', function (req, res) {
  itemStore.splice(req.params.id, 1);
  res.json(req.body);
});

//START SERVER
server.listen(3000, function () {
  console.log("Server running");
});
```

Прочитайте Простой API CRUD на основе REST онлайн: <https://riptutorial.com/ru/node-js/topic/5850/простой-апи-crud-на-основе-rest>

# глава 84: Пул соединений Mysql

## Examples

### Использование пула соединений без базы данных

Достижение многопоточности на сервере базы данных с несколькими базами данных, размещенными на нем.

В настоящее время многоуровневость является общим требованием к корпоративному приложению, и создание пула соединений для каждой базы данных на сервере базы данных не рекомендуется. поэтому вместо этого мы можем создать пул соединений с сервером базы данных и переключиться между базами данных, размещенными на сервере базы данных по требованию.

Предположим, что наше приложение имеет разные базы данных для каждой фирмы, размещенной на сервере базы данных. Мы подключаемся к соответствующей базе данных фирмы, когда пользователь обращается к приложению. вот пример того, как это сделать: -

```
var pool = mysql.createPool({
  connectionLimit : 10,
  host            : 'example.org',
  user           : 'bobby',
  password       : 'pass'
});

pool.getConnection(function(err, connection){
  if(err){
    return cb(err);
  }
  connection.changeUser({database : "firm1"});
  connection.query("SELECT * from history", function(err, data){
    connection.release();
    cb(err, data);
  });
});
```

Позвольте мне разбить пример:

При определении конфигурации пула я не дал имя базы данных, но дал только сервер базы данных, т. Е.

```
{
  connectionLimit : 10,
  host            : 'example.org',
  user           : 'bobby',
  password       : 'pass'
}
```

поэтому, когда мы хотим использовать конкретную базу данных на сервере базы данных, мы запрашиваем подключение к базе данных, используя: -

```
connection.changeUser({database : "firm1"});
```

вы можете сослаться на официальную документацию [здесь](#)

Прочитайте Пул соединений Mysql онлайн: <https://riptutorial.com/ru/node-js/topic/6353/пул-соединений-mysql>

# глава 85: Разбор аргументов командной строки

## Examples

### Передача действий (глагол) и значений

```
const options = require("commander");

options
  .option("-v, --verbose", "Be verbose");

options
  .command("convert")
  .alias("c")
  .description("Converts input file to output file")
  .option("-i, --in-file <file_name>", "Input file")
  .option("-o, --out-file <file_name>", "Output file")
  .action(doConvert);

options.parse(process.argv);

if (!options.args.length) options.help();

function doConvert(options) {
  //do something with options.inFile and options.outFile
};
```

### Передача логических переключателей

```
const options = require("commander");

options
  .option("-v, --verbose")
  .parse(process.argv);

if (options.verbose) {
  console.log("Let's make some noise!");
}
```

Прочитайте Разбор аргументов командной строки онлайн: <https://riptutorial.com/ru/node-js/topic/6174/разбор-аргументов-командной-строки>

---

# глава 86: Развертывание приложений Node.js в производстве

## Examples

### Настройка NODE\_ENV = "производство"

Производственные развертывания будут разными способами, но стандартное соглашение при развертывании в производстве - это определить переменную среды, называемую `NODE_ENV` и установить ее значение в «*production*» .

---

## Флаги времени выполнения

Любой код, запущенный в вашем приложении (включая внешние модули), может проверить значение `NODE_ENV` :

```
if(process.env.NODE_ENV === 'production') {
  // We are running in production mode
} else {
  // We are running in development mode
}
```

---

## ЗАВИСИМОСТИ

Когда для переменной среды `NODE_ENV` установлено значение «*production*», все `devDependencies` в вашем файле `package.json` будут полностью игнорироваться при запуске `npm install` . Вы также можете обеспечить это с помощью флага `--production` :

```
npm install --production
```

Для установки `NODE_ENV` вы можете использовать любой из этих методов

### метод 1: установите NODE\_ENV для всех приложений узла

Windows:

```
set NODE_ENV=production
```

Linux или другая система на основе unix:

```
export NODE_ENV=production
```

Это устанавливает `NODE_ENV` для текущего сеанса `bash`, поэтому все приложения запускаются после того, как этот оператор будет иметь `NODE_ENV` установленный для `production`.

## метод 2: установите `NODE_ENV` для текущего приложения

```
NODE_ENV=production node app.js
```

Это установит только `NODE_ENV` для текущего приложения. Это помогает, когда мы хотим протестировать наши приложения в разных средах.

## метод 3: создать файл `.env` и использовать его

Здесь используется идея, описанная [здесь](#). См. Это сообщение для более подробного объяснения.

В основном вы создаете файл `.env` и запускаете некоторый скрипт `bash`, чтобы установить их в среде.

Чтобы избежать написания сценария `bash`, пакет `env-cmd` можно использовать для загрузки переменных среды, определенных в файле `.env`.

```
env-cmd .env node app.js
```

## метод 4: используйте пакет `cross-env`

Этот [пакет](#) позволяет устанавливать переменные среды для каждой платформы.

После установки с помощью `npm` вы можете просто добавить его в свой сценарий развертывания в `package.json` следующим образом:

```
"build:deploy": "cross-env NODE_ENV=production webpack"
```

## Управление приложением с диспетчером процессов

Хорошая практика для запуска приложений NodeJS, контролируемых менеджерами процессов. Менеджер процессов помогает поддерживать работоспособность приложения навсегда, перезагружается при сбое, перезагружается без простоя и упрощает администрирование. Самые мощные из них (например, [PM2](#)) имеют встроенный балансировщик нагрузки. `PM2` также позволяет управлять ведением журнала приложений, мониторингом и кластеризацией.

## Менеджер процессов `PM2`

Установка `PM2`:

```
npm install pm2 -g
```

Процесс можно запустить в режиме кластера с интегрированным балансировщиком нагрузки для распространения нагрузки между процессами:

```
pm2 start app.js -i 0 --name "api" ( -i - указать количество процессов для появления. Если это 0, то номер процесса будет основан на подсчете количества ядер процессора)
```

Имея несколько пользователей в производстве, он должен иметь единственную точку для PM2. Поэтому команда pm2 должна иметь префикс местоположения (для конфигурации PM2), иначе он будет порождать новый процесс pm2 для каждого пользователя с конфигурацией в соответствующем домашнем каталоге. И это будет непоследовательно.

Использование: `PM2_HOME=/etc/.pm2 pm2 start app.js`

## Развертывание с использованием PM2

PM2 - это менеджер производственных процессов для приложений `Node.js`, который позволяет сохранять приложения навсегда и перезагружать их без простоя. PM2 также позволяет управлять ведением журнала приложений, мониторингом и кластеризацией.

Установите pm2 глобально.

```
npm install -g pm2
```

Затем запустите приложение `node.js`, используя PM2.

```
pm2 start server.js --name "my-app"
```

```
$ pm2 start app.js --name my-app
[PM2] restartProcessId process id 0
```

App name	id	mode	pid	status	restart	uptime	memory	watching
my-app	0	fork	64029	online	1	0s	17.816 MB	disabled

Use the ``pm2 show <id|name>`` command to get more details about an app.

Следующие команды полезны при работе с PM2 .

Список всех запущенных процессов:

```
pm2 list
```

Остановить приложение:

```
pm2 stop my-app
```

Перезапустите приложение:

```
pm2 restart my-app
```

Чтобы просмотреть подробную информацию о приложении:

```
pm2 show my-app
```

Чтобы удалить приложение из реестра PM2:

```
pm2 delete my-app
```

## Развертывание с помощью диспетчера процессов

Диспетчер процессов обычно используется в производстве для развертывания приложения `nodedjs`. Основными функциями диспетчера процессов являются перезапуск сервера, если он сбой, проверка потребления ресурсов, улучшение производительности во время выполнения, мониторинг и т. Д.

Некоторые из популярных менеджеров процессов, созданных сообществом узлов, навсегда, `pm2` и т. Д.

## Forever

`forever` - инструмент интерфейса командной строки для обеспечения непрерывного выполнения заданного сценария. `forever` «с простой интерфейс делает его идеальным для запуска небольших развертываний `Node.js` приложений и скриптов.

`forever` контролирует ваш процесс и перезапускает его, если он сработает.

Установите `forever` глобально.

```
$ npm install -g forever
```

Запустить приложение:

```
$ forever start server.js
```

Это запустит сервер и дает идентификатор для процесса (начинается с 0).

Перезапустить приложение:

```
$ forever restart 0
```



Здесь 0 - идентификатор сервера.

Остановить применение:

```
$ forever stop 0
```

Подобно перезагрузке, 0 является идентификатором сервера. Вы также можете указать идентификатор процесса или имя сценария вместо идентификатора, заданного вечным.

Дополнительные команды: <https://www.npmjs.com/package/forever>

## Использование разных свойств / конфигурации для разных сред, таких как dev, qa, настройка и т. Д.

Крупномасштабные приложения часто нуждаются в разных свойствах при работе в разных средах. мы можем добиться этого, передав аргументы в приложение NodeJS и используя тот же аргумент в процессе узла для загрузки файла свойств среды.

Предположим, у нас есть два файла свойств для разных условий.

- dev.json

```
{
  "PORT": 3000,
  "DB": {
    "host": "localhost",
    "user": "bob",
    "password": "12345"
  }
}
```

- qa.json

```
{
  "PORT": 3001,
  "DB": {
    "host": "where_db_is_hosted",
    "user": "bob",
    "password": "54321"
  }
}
```

Следующий код в приложении будет экспортировать соответствующий файл свойств, который мы хотим использовать.

```
process.argv.forEach(function (val) {
```

```
var arg = val.split("=");
if (arg.length > 0) {
  if (arg[0] === 'env') {
    var env = require('./' + arg[1] + '.json');
    exports.prop = env;
  }
}
});
```

Мы приводим аргументы для приложения, как показано ниже.

```
node app.js env=dev
```

если мы используем диспетчер процессов как *навсегда*, чем это так просто, как

```
forever start app.js env=dev
```

## Воспользовавшись кластерами

Один экземпляр Node.js работает в одном потоке. Чтобы воспользоваться преимуществами многоядерных систем, пользователь иногда захочет запустить кластер из процессов Node.js для обработки нагрузки.

```
var cluster = require('cluster');

var numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  // In real life, you'd probably use more than just 2 workers,
  // and perhaps not put the master and worker in the same file.
  //
  // You can also of course get a bit fancier about logging, and
  // implement whatever custom logic you need to prevent DoS
  // attacks and other bad behavior.
  //
  // See the options in the cluster documentation.
  //
  // The important thing is that the master does very little,
  // increasing our resilience to unexpected errors.
  console.log('your server is working on ' + numCPUs + ' cores');

  for (var i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('disconnect', function(worker) {
    console.error('disconnect!');
    //clearTimeout(timeout);
    cluster.fork();
  });
} else {
  require('./app.js');
```

```
}
```

Прочитайте Развертывание приложений Node.js в производстве онлайн:  
<https://riptutorial.com/ru/node-js/topic/2975/развертывание-приложений-node-js-в-производстве>

---

# глава 87: Развертывание приложения Node.js без простоя.

## Examples

### Развертывание с использованием PM2 без простоя.

ecosystem.json

```
{
  "name": "app-name",
  "script": "server",
  "exec_mode": "cluster",
  "instances": 0,
  "wait_ready": true
  "listen_timeout": 10000,
  "kill_timeout": 5000,
}
```

wait\_ready

Вместо перезагрузки, ожидающего прослушивания, подождите `process.send('ready');`

listen\_timeout

Время в мс перед тем, как заставить перезагружать, если приложение не прослушивает.

kill\_timeout

Время в мс перед отправкой окончательного SIGKILL.

server.js

```
const http = require('http');
const express = require('express');

const app = express();
const server = http.Server(app);
const port = 80;

server.listen(port, function() {
  process.send('ready');
});

process.on('SIGINT', function() {
  server.close(function() {
    process.exit(0);
  });
});
```

Возможно, вам придется подождать, пока ваше приложение установит соединения с вашими DB / кэшами / рабочими / независимо. PM2 должен ждать, прежде чем рассматривать ваше приложение как онлайн. Для этого вам нужно предоставить `wait_ready: true` в файле процесса. Это заставит PM2 прослушать это событие. В вашем приложении вам нужно будет добавить `process.send('ready');` когда вы хотите, чтобы ваше приложение считалось готовым.

Когда процесс останавливается / перезапускается PM2, некоторые системные сигналы отправляются в ваш процесс в определенном порядке.

Сначала сигнал `SIGINT` отправляется в ваши процессы, сигнализируя, что вы можете узнать, что ваш процесс будет остановлен. Если ваше приложение не выйдет за пределы до 1.6s (настраивается), он получит сигнал `SIGKILL` чтобы заставить процесс выйти. Поэтому, если вашему приложению необходимо очистить какие-либо состояния или задания, вы можете поймать сигнал `SIGINT` чтобы подготовить свое приложение к выходу.

Прочитайте [Развертывание приложения Node.js без простоя. онлайн:](https://riptutorial.com/ru/node-js/topic/9752/развертывание-приложения-node-js-без-простоя-онлайн)

[https://riptutorial.com/ru/node-js/topic/9752/развертывание-приложения-node-js-без-простоя-](https://riptutorial.com/ru/node-js/topic/9752/развертывание-приложения-node-js-без-простоя-онлайн)

---

# глава 88: Рамки шаблонов

## Examples

### Nunjucks

Серверный движок с наследованием блоков, автоматическим кэшированием, макросами, асинхронным управлением и т. Д. Сильно вдохновленный jinja2, очень похожий на Twig (php).

Документы - <http://mozilla.github.io/nunjucks/>

Установить - `npm i nunjucks`

Основное использование с помощью [Express](#) ниже.

#### *app.js*

```
var express = require ('express');
var nunjucks = require('nunjucks');

var app = express();
app.use(express.static('/public'));

// Apply nunjucks and add custom filter and function (for example).
var env = nunjucks.configure(['views/'], { // set folders with templates
  autoescape: true,
  express: app
});
env.addFilter('myFilter', function(obj, arg1, arg2) {
  console.log('myFilter', obj, arg1, arg2);
  // Do smth with obj
  return obj;
});
env.addGlobal('myFunc', function(obj, arg1) {
  console.log('myFunc', obj, arg1);
  // Do smth with obj
  return obj;
});

app.get('/', function(req, res){
  res.render('index.html', {title: 'Main page'});
});

app.get('/foo', function(req, res){
  res.locals.smthVar = 'This is Sparta!';
  res.render('foo.html', {title: 'Foo page'});
});

app.listen(3000, function() {
  console.log('Example app listening on port 3000...');
});
```

*/views/index.html*

```
<html>
<head>
  <title>Nunjucks example</title>
</head>
<body>
  {% block content %}
  {{title}}
  {% endblock %}
</body>
</html>
```

*/views/foo.html*

```
{% extends "index.html" %}

{# This is comment #}
{% block content %}
  <h1>{{title}}</h1>
  {# apply custom function and next build-in and custom filters #}
  {{ myFunc(smithVar) | lower | myFilter(5, 'abc') }}
{% endblock %}
```

Прочитайте Рамки шаблонов онлайн: <https://riptutorial.com/ru/node-js/topic/5885/рамки-шаблонов>

---

# глава 89: Руководство для начинающих NodeJS

## Examples

Привет, мир !

Вставьте следующий код в имя файла `helloworld.js`

```
console.log("Hello World");
```

Сохраните файл и выполните его через Node.js:

```
node helloworld.js
```

Прочитайте [Руководство для начинающих NodeJS онлайн](https://riptutorial.com/ru/node-js/topic/7693/руководство-для-начинающих-nodejs): <https://riptutorial.com/ru/node-js/topic/7693/руководство-для-начинающих-nodejs>



# глава 90: Связь Socket.io

## Examples

"Привет, мир!" с сообщениями сокета.

Устанавливать узловые модули

```
npm install express
npm install socket.io
```

Сервер Node.js

```
const express = require('express');
const app = express();
const server = app.listen(3000, console.log("Socket.io Hello World server started!"));
const io = require('socket.io')(server);

io.on('connection', (socket) => {
  //console.log("Client connected!");
  socket.on('message-from-client-to-server', (msg) => {
    console.log(msg);
  })
  socket.emit('message-from-server-to-client', 'Hello World!');
});
```

Клиент браузера

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Hello World with Socket.io</title>
  </head>
  <body>
    <script src="https://cdn.socket.io/socket.io-1.4.5.js"></script>
    <script>
      var socket = io("http://localhost:3000");
      socket.on("message-from-server-to-client", function(msg) {
        document.getElementById('message').innerHTML = msg;
      });
      socket.emit('message-from-client-to-server', 'Hello World!');
    </script>
    <p>Socket.io Hello World client started!</p>
    <p id="message"></p>
  </body>
</html>
```

Прочитайте Связь Socket.io онлайн: <https://riptutorial.com/ru/node-js/topic/4261/связь-socket-io>

---

# глава 91: Связь клиент-сервер

## Examples

/ в Экспресс, jQuery и Jade

```
//'client.jade'  
  
//a button is placed down; similar in HTML  
button(type='button', id='send_by_button') Modify data  
  
#modify Lorem ipsum Sender  
  
//loading jQuery; it can be done from an online source as well  
script(src='./js/jquery-2.2.0.min.js')  
  
//AJAX request using jQuery  
script  
  $(function () {  
    $('#send_by_button').click(function (e) {  
      e.preventDefault();  
  
      //test: the text within brackets should appear when clicking on said button  
      //window.alert('You clicked on me. - jQuery');  
  
      //a variable and a JSON initialized in the code  
      var predeclared = "Katamori";  
      var data = {  
        Title: "Name_SenderTest",  
        Nick: predeclared,  
        FirstName: "Zoltan",  
        Surname: "Schmidt"  
      };  
  
      //an AJAX request with given parameters  
      $.ajax({  
        type: 'POST',  
        data: JSON.stringify(data),  
        contentType: 'application/json',  
        url: 'http://localhost:7776/domaintest',  
  
        //on success, received data is used as 'data' function input  
        success: function (data) {  
          window.alert('Request sent; data received.');  
          var jsonstr = JSON.stringify(data);  
          var jsonobj = JSON.parse(jsonstr);  
  
          //if the 'nick' member of the JSON does not equal to the predeclared  
          string (as it was initialized), then the backend script was executed, meaning that  
          communication has been established  
          if(data.Nick != predeclared){  
            document.getElementById("modify").innerHTML = "JSON changed!\n" +  
            jsonstr;  
          }  
        }  
      });  
    }  
  });
```

```

        }
    });
});
});

//'domaintest_route.js'

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

//an Express router listening to GET requests - in this case, it's empty, meaning that nothing
is displayed when you reach 'localhost/domaintest'
router.get('/', function(req, res, next) {
});

//same for POST requests - notice, how the AJAX request above was defined as POST
router.post('/', function(req, res) {
    res.setHeader('Content-Type', 'application/json');

    //content generated here
    var some_json = {
        Title: "Test",
        Item: "Crate"
    };

    var result = JSON.stringify(some_json);

    //content got 'client.jade'
    var sent_data = req.body;
    sent_data.Nick = "ttony33";

    res.send(sent_data);

});

module.exports = router;

```

// на основе лично используемого gist: <https://gist.github.com/Katamori/5c9850f02e4baf6e9896>

Прочитайте Связь клиент-сервер онлайн: <https://riptutorial.com/ru/node-js/topic/6222/связь-клиент-сервер>

# глава 92: Сервер узла без рамки

## замечания

Хотя у [узла](#) есть много возможностей, которые помогут вам запустить ваш сервер, в основном:

[Экспресс](#) : наиболее используемая структура

[Total](#) : структура ALL-IN-ONE UNITY, которая имеет все и не зависит от какой-либо другой структуры или модуля.

Но всегда есть ни один размер, который подходит всем, поэтому разработчику, возможно, придется создавать собственный сервер без какой-либо другой зависимости.

Если приложение, к которому я обращался через внешний сервер, может быть проблемой, [CORS](#) может быть проблемой, код, который следует избегать, был предоставлен.

## Examples

### Сервер узлов без структуры

```
var http = require('http');
var fs = require('fs');
var path = require('path');

http.createServer(function (request, response) {
  console.log('request ', request.url);

  var filePath = '.' + request.url;
  if (filePath == './')
    filePath = './index.html';

  var extname = String(path.extname(filePath)).toLowerCase();
  var contentType = 'text/html';
  var mimeTypes = {
    '.html': 'text/html',
    '.js': 'text/javascript',
    '.css': 'text/css',
    '.json': 'application/json',
    '.png': 'image/png',
    '.jpg': 'image/jpeg',
    '.gif': 'image/gif',
    '.wav': 'audio/wav',
    '.mp4': 'video/mp4',
    '.woff': 'application/font-woff',
    '.ttf': 'application/font-ttf',
    '.eot': 'application/vnd.ms-fontobject',
    '.otf': 'application/font-otf',
    '.svg': 'application/image/svg+xml'
  };
});
```

```

contentType = mimeTypes[extname] || 'application/octet-stream';

fs.readFile(filePath, function(error, content) {
  if (error) {
    if(error.code == 'ENOENT'){
      fs.readFile('./404.html', function(error, content) {
        response.writeHead(200, { 'Content-Type': contentType });
        response.end(content, 'utf-8');
      });
    }
    else {
      response.writeHead(500);
      response.end('Sorry, check with the site admin for error: '+error.code+' ..\n');
      response.end();
    }
  }
  else {
    response.writeHead(200, { 'Content-Type': contentType });
    response.end(content, 'utf-8');
  }
});

}).listen(8125);
console.log('Server running at http://127.0.0.1:8125/');

```

## Преодоление проблем CORS

```

// Website you wish to allow to connect to
response.setHeader('Access-Control-Allow-Origin', '*');

// Request methods you wish to allow
response.setHeader('Access-Control-Allow-Methods', 'GET, POST, OPTIONS, PUT, PATCH, DELETE');

// Request headers you wish to allow
response.setHeader('Access-Control-Allow-Headers', 'X-Requested-With,content-type');

// Set to true if you need the website to include cookies in the requests sent
// to the API (e.g. in case you use sessions)
response.setHeader('Access-Control-Allow-Credentials', true);

```

Прочитайте Сервер узла без рамки онлайн: <https://riptutorial.com/ru/node-js/topic/5910/сервер-узла-без-рамки>

# глава 93: Синхронное и асинхронное программирование в узлах

## Examples

### Использование `async`

Пакет `async` предоставляет функции для асинхронного кода.

Используя функцию `auto`, вы можете определить асинхронные отношения между двумя или более функциями:

```
var async = require('async');

async.auto({
  get_data: function(callback) {
    console.log('in get_data');
    // async code to get some data
    callback(null, 'data', 'converted to array');
  },
  make_folder: function(callback) {
    console.log('in make_folder');
    // async code to create a directory to store a file in
    // this is run at the same time as getting the data
    callback(null, 'folder');
  },
  write_file: ['get_data', 'make_folder', function(results, callback) {
    console.log('in write_file', JSON.stringify(results));
    // once there is some data and the directory exists,
    // write the data to a file in the directory
    callback(null, 'filename');
  }],
  email_link: ['write_file', function(results, callback) {
    console.log('in email_link', JSON.stringify(results));
    // once the file is written let's email a link to it...
    // results.write_file contains the filename returned by write_file.
    callback(null, {'file':results.write_file, 'email':'user@example.com'});
  }],
}, function(err, results) {
  console.log('err = ', err);
  console.log('results = ', results);
});
```

Этот код можно было бы сделать синхронно, просто позвонив `get_data`, `make_folder`, `write_file` и `email_link` в правильном порядке. `Async` отслеживает результаты для вас, и если произошла ошибка (первый параметр `callback` равен `null`), он останавливает выполнение других функций.

Прочитайте [Синхронное и асинхронное программирование в узлах онлайн](https://riptutorial.com/ru/node-js/topic/8287/синхронное-и-асинхронное-программирование-в-узлах):

<https://riptutorial.com/ru/node-js/topic/8287/синхронное-и-асинхронное-программирование-в-узлах>

узлах

# глава 94: Создание API с помощью Node.js

## Examples

### GET api с помощью Express

Node.js apis можно легко создать в веб-среде Express .

В следующем примере создается простой GET api для перечисления всех пользователей.

#### пример

```
var express = require('express');
var app = express();

var users = [{
  id: 1,
  name: "John Doe",
  age : 23,
  email: "john@doe.com"
}];

// GET /api/users
app.get('/api/users', function(req, res){
  return res.json(users);    //return response as JSON
});

app.listen('3000', function(){
  console.log('Server listening on port 3000');
});
```

### POST api с помощью Express

В следующем примере создайте POST api с помощью Express . Этот пример похож на пример GET за исключением использования body-parser который анализирует данные post и добавляет его в req.body .

#### пример

```
var express = require('express');
var app = express();
// for parsing the body in POST request
var bodyParser = require('body-parser');

var users = [{
  id: 1,
  name: "John Doe",
  age : 23,
  email: "john@doe.com"
}];
```



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

// GET /api/users
app.get('/api/users', function(req, res){
  return res.json(users);
});

/* POST /api/users
  {
    "user": {
      "id": 3,
      "name": "Test User",
      "age" : 20,
      "email": "test@test.com"
    }
  }
*/
app.post('/api/users', function (req, res) {
  var user = req.body.user;
  users.push(user);

  return res.send('User has been added successfully');
});

app.listen('3000', function(){
  console.log('Server listening on port 3000');
});
```

Прочитайте Создание API с помощью Node.js онлайн: <https://riptutorial.com/ru/node-js/topic/5991/создание-апи-с-помощью-node-js>

---

# глава 95: Создание библиотеки Node.js, которая поддерживает обе обещания и первые обратные вызовы

## Вступление

Многим нравится работать с обещаниями и / или синтаксисом `async / await`, но при написании модуля было бы полезно, чтобы некоторые программисты также поддерживали классические методы стиля обратного вызова. Вместо того, чтобы создавать два модуля или два набора функций или иметь программиста, обещающего ваш модуль, ваш модуль может поддерживать оба метода программирования на одном, используя `asCallback ()` или `b's nodeify ()`.

## Examples

### Пример модуля и соответствующей программы с помощью Bluebird

#### math.js

```
'use strict';

const Promise = require('bluebird');

module.exports = {

  // example of a callback-only method
  callbackSum: function(a, b, callback) {
    if (typeof a !== 'number')
      return callback(new Error('"a" must be a number'));
    if (typeof b !== 'number')
      return callback(new Error('"b" must be a number'));

    return callback(null, a + b);
  },

  // example of a promise-only method
  promiseSum: function(a, b) {
    return new Promise(function(resolve, reject) {
      if (typeof a !== 'number')
        return reject(new Error('"a" must be a number'));
      if (typeof b !== 'number')
        return reject(new Error('"b" must be a number'));
      resolve(a + b);
    });
  },

  // a method that can be used as a promise or with callbacks
  sum: function(a, b, callback) {
```

```

return new Promise(function(resolve, reject) {
  if (typeof a !== 'number')
    return reject(new Error('"a" must be a number'));
  if (typeof b !== 'number')
    return reject(new Error('"b" must be a number'));
  resolve(a + b);
}).asCallback(callback);
},
);

```

## index.js

```

'use strict';

const math = require('./math');

// classic callbacks

math.callbackSum(1, 3, function(err, result) {
  if (err)
    console.log('Test 1: ' + err);
  else
    console.log('Test 1: the answer is ' + result);
});

math.callbackSum(1, 'd', function(err, result) {
  if (err)
    console.log('Test 2: ' + err);
  else
    console.log('Test 2: the answer is ' + result);
});

// promises

math.promiseSum(2, 5)
  .then(function(result) {
    console.log('Test 3: the answer is ' + result);
  })
  .catch(function(err) {
    console.log('Test 3: ' + err);
  });

math.promiseSum(1)
  .then(function(result) {
    console.log('Test 4: the answer is ' + result);
  })
  .catch(function(err) {
    console.log('Test 4: ' + err);
  });

// promise/callback method used like a promise

math.sum(8, 2)
  .then(function(result) {
    console.log('Test 5: the answer is ' + result);
  })

```

```
.catch(function(err) {
  console.log('Test 5: ' + err);
});

// promise/callback method used with callbacks

math.sum(7, 11, function(err, result) {
  if (err)
    console.log('Test 6: ' + err);
  else
    console.log('Test 6: the answer is ' + result);
});

// promise/callback method used like a promise with async/await syntax

(async () => {

  try {
    let x = await math.sum(6, 3);
    console.log('Test 7a: ' + x);

    let y = await math.sum(4, 's');
    console.log('Test 7b: ' + y);

  } catch(err) {
    console.log(err.message);
  }

})();
```

Прочитайте [Создание библиотеки Node.js, которая поддерживает обе обещания и первые обратные вызовы онлайн: https://riptutorial.com/ru/node-js/topic/9874/создание-библиотеки-node-js--которая-поддерживает-обе-обещания-и-первые-обратные-вызовы](https://riptutorial.com/ru/node-js/topic/9874/создание-библиотеки-node-js--которая-поддерживает-обе-обещания-и-первые-обратные-вызовы)

---

# глава 96: Сокеты TCP

## Examples

### Простой TCP-сервер

```
// Include Nodejs' net module.
const Net = require('net');
// The port on which the server is listening.
const port = 8080;

// Use net.createServer() in your code. This is just for illustration purpose.
// Create a new TCP server.
const server = new Net.Server();
// The server listens to a socket for a client to make a connection request.
// Think of a socket as an end point.
server.listen(port, function() {
  console.log(`Server listening for connection requests on socket localhost:${port}`.);
});

// When a client requests a connection with the server, the server creates a new
// socket dedicated to that client.
server.on('connection', function(socket) {
  console.log('A new connection has been established.');
```

```
  // Now that a TCP connection has been established, the server can send data to
  // the client by writing to its socket.
  socket.write('Hello, client.');
```

```
  // The server can also receive data from the client by reading from its socket.
  socket.on('data', function(chunk) {
    console.log(`Data received from client: ${chunk.toString}`.);
  });

  // When the client requests to end the TCP connection with the server, the server
  // ends the connection.
  socket.on('end', function() {
    console.log('Closing connection with the client');
```

```
  });

  // Don't forget to catch error, for your own sake.
  socket.on('error', function(err) {
    console.log(`Error: ${err}`.);
  });
});
```

### Простой клиент TCP

```
// Include Nodejs' net module.
const Net = require('net');
// The port number and hostname of the server.
const port = 8080;
const host = 'localhost';
```

```
// Create a new TCP client.
const client = new Net.Socket();
// Send a connection request to the server.
client.connect({ port: port, host: host }, function() {
  // If there is no error, the server has accepted the request and created a new
  // socket dedicated to us.
  console.log('TCP connection established with the server.');
```

```
  // The client can now send data to the server by writing to its socket.
  client.write('Hello, server.');
```

```
});

// The client can also receive data from the server by reading from its socket.
client.on('data', function(chunk) {
  console.log(`Data received from the server: ${chunk.toString()}.`);

  // Request an end to the connection after the data has been received.
  client.end();
});

client.on('end', function() {
  console.log('Requested an end to the TCP connection');
```

```
});
```

Прочитайте Сокеты TCP онлайн: <https://riptutorial.com/ru/node-js/topic/6545/сокеты-tcp>

---

# глава 97: Сообщение Arduino с nodeJs

## Вступление

Способ показать, как Node.Js может общаться с Arduino Uno.

## Examples

### Связь узла Js с Arduino через последовательный порт

## Код узла js

Образец для запуска этой темы - сервер Node.js, связывающийся с Arduino через serialport.

```
npm install express --save
npm install serialport --save
```

### Пример app.js:

```
const express = require('express');
const app = express();
var SerialPort = require("serialport");

var port = 3000;

var arduinoCOMPort = "COM3";

var arduinoSerialPort = new SerialPort(arduinoCOMPort, {
  baudrate: 9600
});

arduinoSerialPort.on('open',function() {
  console.log('Serial Port ' + arduinoCOMPort + ' is opened.');
```

```
});

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

  return res.send('Working!');

})

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

  var action = req.params.action || req.param('action');

  if(action == 'led'){
    arduinoSerialPort.write("w");
    return res.send('Led light is on!');
  }
  if(action == 'off') {
    arduinoSerialPort.write("t");
```

```
        return res.send("Led light is off!");
    }

    return res.send('Action: ' + action);
});

app.listen(port, function () {
    console.log('Example app listening on port http://0.0.0.0:' + port + '!');
});
```

Запуск образца экспресс-сервера:

```
node app.js
```

## Код Arduino

```
// the setup function runs once when you press reset or power the board
void setup() {
    // initialize digital pin LED_BUILTIN as an output.

    Serial.begin(9600); // Begen listening on port 9600 for serial

    pinMode(LED_BUILTIN, OUTPUT);

    digitalWrite(LED_BUILTIN, LOW);
}

// the loop function runs over and over again forever
void loop() {

    if(Serial.available() > 0) // Read from serial port
    {
        char ReaderFromNode; // Store current character
        ReaderFromNode = (char) Serial.read();
        convertToState(ReaderFromNode); // Convert character to state
    }
    delay(1000);
}

void convertToState(char chr) {
    if(chr=='o'){
        digitalWrite(LED_BUILTIN, HIGH);
        delay(100);
    }
    if(chr=='f'){
        digitalWrite(LED_BUILTIN, LOW);
        delay(100);
    }
}
```

## Запуск

1. Подключите ардуино к вашей машине.



## 2. Запустить сервер

Контролируйте сборку во главе с узлом js express server.

Чтобы включить светодиод:

```
http://0.0.0.0:3000/led
```

Чтобы выключить светодиод:

```
http://0.0.0.0:3000/off
```

Прочитайте Сообщение Arduino с nodeJs онлайн: <https://riptutorial.com/ru/node-js/topic/10509/сообщение-arduino-с-nodejs>

# глава 98: Среда

## Examples

### Доступ к переменным среды

Свойство `process.env` возвращает объект, содержащий пользовательскую среду.

Он возвращает объект, подобный этому:

```
{
  TERM: 'xterm-256color',
  SHELL: '/usr/local/bin/bash',
  USER: 'maciej',
  PATH: '~/.bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin',
  PWD: '/Users/maciej',
  EDITOR: 'vim',
  SHLVL: '1',
  HOME: '/Users/maciej',
  LOGNAME: 'maciej',
  _: '/usr/local/bin/node'
}
```

```
process.env.HOME // '/Users/maciej'
```

Если вы установите переменную среды `FOO` в `foobar`, она будет доступна с помощью:

```
process.env.FOO // 'foobar'
```

### аргументы командной строки `process.argv`

[process.argv](#) - это массив, содержащий аргументы командной строки. Первым элементом будет `node`, второй элемент будет именем файла JavaScript. Следующие элементы будут любыми дополнительными аргументами командной строки.

#### Пример кода:

Выводная сумма всех аргументов командной строки

`index.js`

```
var sum = 0;
for (i = 2; i < process.argv.length; i++) {
  sum += Number(process.argv[i]);
}

console.log(sum);
```

## Использование Eхарле:

```
node index.js 2 5 6 7
```

Выход будет 20

## Краткое описание кода:

Здесь цикл `for (i = 2; i < process.argv.length; i++)` начинается с 2, потому что первые два элемента в массиве `process.argv` **всегда** являются `['path/to/node.exe', 'path/to/js/file', ...]`

Преобразование в число `Number(process.argv[i])` поскольку элементы в массиве `process.argv` **всегда** являются строками

## Использование разных свойств / конфигурации для разных сред, таких как dev, qa, настройка и т. Д.

Крупномасштабные приложения часто нуждаются в разных свойствах при работе в разных средах. мы можем добиться этого, передав аргументы в приложение NodeJs и используя тот же аргумент в процессе узла для загрузки файла свойств среды.

Предположим, у нас есть два файла свойств для разных условий.

- dev.json

```
{
  PORT : 3000,
  DB : {
    host : "localhost",
    user : "bob",
    password : "12345"
  }
}
```

- qa.json

```
{
  PORT : 3001,
  DB : {
    host : "where_db_is_hosted",
    user : "bob",
    password : "54321"
  }
}
```

Следующий код в приложении будет экспортировать соответствующий файл свойств, который мы хотим использовать.

Предположим, что код находится в `environment.js`

```
process.argv.forEach(function (val, index, array) {
  var arg = val.split("=");
  if (arg.length > 0) {
    if (arg[0] === 'env') {
      var env = require('./' + arg[1] + '.json');
      module.exports = env;
    }
  }
});
```

Мы приводим аргументы для приложения, как показано ниже.

```
node app.js env=dev
```

если мы используем диспетчер процессов как *навсегда*, чем это так просто, как

```
forever start app.js env=dev
```

## Как использовать файл конфигурации

```
var env= require("environment.js");
```

## Загрузка свойств среды из файла свойств

- Установщик свойств:

```
npm install properties-reader --save
```

- Создайте **каталог env** для хранения ваших файлов свойств:

```
mkdir env
```

- Создать **environment.js** :

```
process.argv.forEach(function (val, index, array) {
  var arg = val.split("=");
  if (arg.length > 0) {
    if (arg[0] === 'env') {
      var env = require('./env/' + arg[1] + '.properties');
      module.exports = env;
    }
  }
});
```

- **Файл СВОЙСТВ свойства `development.properties` :**

```
# Dev properties
[main]
# Application port to run the node server
app.port=8080

[database]
# Database connection to mysql
mysql.host=localhost
mysql.port=2500
...
```

- **Пример использования загруженных свойств:**

```
var environment = require('./environments');
var PropertiesReader = require('properties-reader');
var properties = new PropertiesReader(environment);

var someVal = properties.get('main.app.port');
```

- **Запуск экспресс-сервера**

```
npm start env=development
```

или же

```
npm start env=production
```

Прочитайте Среда онлайн: <https://riptutorial.com/ru/node-js/topic/2340/среда>

---

# глава 99: Структура Route-Controller-Service для ExpressJS

## Examples

### Модель-Маршруты-Контроллеры-Структура каталогов услуг

```
|—models
|   |—user.model.js
|—routes
|   |—user.route.js
|—services
|   |—user.service.js
|—controllers
|   |—user.controller.js
```

Для модульной структуры кода логику следует разделить на эти каталоги и файлы.

**Модели** - определение схемы модели

**Маршруты** - API маршрутизирует карты контроллерам

**Контроллеры** . Контроллеры обрабатывают всю логику за проверкой параметров запроса, запроса, отправки ответов с правильными кодами.

**Сервисы** . Службы содержат запросы к базе данных и возвращающие объекты или метательные ошибки

Этот кодер закончит писать больше кодов. Но в конце коды будут намного более удобными и разделенными.

### Структура кода-контроллеров-контроллеров-служб

---

## user.model.js

```
var mongoose = require('mongoose')

const UserSchema = new mongoose.Schema({
  name: String
})

const User = mongoose.model('User', UserSchema)

module.exports = User;
```

## user.routes.js

```
var express = require('express');
var router = express.Router();

var UserController = require('../controllers/user.controller')

router.get('/', UserController.getUsers)

module.exports = router;
```

## user.controllers.js

```
var UserService = require('../services/user.service')

exports.getUsers = async function (req, res, next) {
  // Validate request parameters, queries using express-validator

  var page = req.params.page ? req.params.page : 1;
  var limit = req.params.limit ? req.params.limit : 10;
  try {
    var users = await UserService.getUsers({}, page, limit)
    return res.status(200).json({ status: 200, data: users, message: "Succesfully Users Retrieved" });
  } catch (e) {
    return res.status(400).json({ status: 400, message: e.message });
  }
}
```

## user.services.js

```
var User = require('../models/user.model')

exports.getUsers = async function (query, page, limit) {

  try {
    var users = await User.find(query)
    return users;
  } catch (e) {
    // Log Errors
    throw Error('Error while Paginating Users')
  }
}
```

Прочитайте Структура Route-Controller-Service для ExpressJS онлайн:

<https://riptutorial.com/ru/node-js/topic/10785/структура-route-controller-service-для-expressjs>

---

# глава 100: Структура проекта

## Вступление

На структуру проекта `nodejs` влияют личные предпочтения, используемая архитектура проекта и стратегия внедрения модулей. Также на основе событий, основанной на дуге, которая использует механизм создания динамического модуля. Чтобы иметь структуру MVC, необходимо выделить исходный код на стороне сервера и на стороне клиента, поскольку код на стороне клиента, вероятно, будет сведен к минимуму и отправлен в браузер и будет общедоступным по своему основному характеру. И серверная или серверная часть предоставит API для выполнения операций CRUD

## замечания

В приведенном выше проекте используются модули браузера и `vue.js` в качестве базового представления приложений и библиотек минимизации. Таким образом, структура проекта может незначительно измениться на основе используемой вами структуры `mvc`. Например, в каталоге сборки публично должен быть указан весь код `mvc`. У вас может быть задача, которая сделает это за вас.

## Examples

### Простое приложение `nodejs` с MVC и API

- Первое важное различие заключается в динамически созданных каталогах, которые будут использоваться для хостинга и исходных каталогов.
- У исходных каталогов будет файл конфигурации или папка в зависимости от объема конфигурации, который у вас может быть. Это включает конфигурацию среды и конфигурацию бизнес-логики, которую вы можете поместить в каталог конфигурации.

```
|-- Config
  |-- config.json
  |-- appConfig
    |-- pets.config
    |-- payment.config
```

- Теперь наиболее важные каталоги, в которых мы различаем серверную часть / бэкэнд и интерфейсные модули. Сервер 2-х *серверов* и *webapp* представляют собой бэкэнд и интерфейс, которые мы можем выбрать для размещения в исходной директории. `src`.

Вы можете использовать разные имена в соответствии с личным выбором



для сервера или webapp в зависимости от того, что имеет смысл для вас. Удостоверьтесь, что вы не хотите делать это слишком долго или сложным, поскольку это в конечном итоге внутренняя структура проекта.

- Внутри каталога *сервера* вы можете иметь контроллер, App.js / index.js, который будет вашим основным файлом nodejs и начальной точкой. Сервер dir. также может иметь *dto-dir*, который содержит все объекты передачи данных, которые будут использоваться контроллерами API.

```
|-- server
  |-- dto
    |-- pet.js
    |-- payment.js
  |-- controller
    |-- PetsController.js
    |-- PaymentController.js
  |-- App.js
```

- Каталог webapp можно разделить на две основные части *public* и *mvc*, на это снова влияет стратегия построения, которую вы хотите использовать. Мы используем [browserfy](#) для сборки MVC-части webapp и минимизации содержимого из простой директории *mvc*.

| - Охота и рыбалка

- Теперь общий каталог может содержать все статические ресурсы, изображения, css (вы также можете иметь файлы saas) и, самое главное, файлы HTML.

```
|-- public
  |-- build // will contain minified scripts (mvc)
  |-- images
    |-- mouse.jpg
    |-- cat.jpg
  |-- styles
    |-- style.css
  |-- views
    |-- petStore.html
    |-- paymentGateway.html
    |-- header.html
    |-- footer.html
  |-- index.html
```

- Каталог *mvc* будет содержать интерфейсную логику, включая *модели*, *контроллеры представлений* и любые другие модули *utils*, которые могут вам понадобиться как часть пользовательского интерфейса. Также index.js или shell.js в зависимости от того, какой из них будет принадлежать, вы также являетесь частью этого каталога.

```
|-- mvc
  |-- controllers
    |-- Dashborad.js
    |-- Help.js
```

```
|-- Login.js
|-- utils
|-- index.js
```

Итак, в заключение вся структура проекта будет выглядеть ниже. И простая задача сборки, такая как `gulp browserify`, минимизирует сценарии `mvc` и публикует в *общедоступном* каталоге. Затем мы можем предоставить этот **общий** каталог как статический ресурс через `express.use (satic ('public'))` api.

```
|-- node_modules
|-- src
  |-- server
    |-- controller
    |-- App.js // node app
  |-- webapp
    |-- public
      |-- styles
      |-- images
      |-- index.html
    |-- mvc
      |-- controller
      |-- shell.js // mvc shell
|-- config
|-- Readme.md
|-- .gitignore
|-- package.json
```

Прочитайте Структура проекта онлайн: <https://riptutorial.com/ru/node-js/topic/9935/структура-проекта>

---

# глава 101: Структуры модульного тестирования

## Examples

### Мокко синхронный

```
describe('Suite Name', function() {
  describe('#method()', function() {
    it('should run without an error', function() {
      expect([ 1, 2, 3 ].length).to.be.equal(3)
    })
  })
})
```

### Моча асинхронный (обратный вызов)

```
var expect = require("chai").expect;
describe('Suite Name', function() {
  describe('#method()', function() {
    it('should run without an error', function(done) {
      testSomething(err => {
        expect(err).to.not.be.equal(null)
        done()
      })
    })
  })
})
```

### Моха асинхронный (обещание)

```
describe('Suite Name', function() {
  describe('#method()', function() {
    it('should run without an error', function() {
      return doSomething().then(result => {
        expect(result).to.be.equal('hello world')
      })
    })
  })
})
```

### Mocha Asynchronous (асинхронный / ждущий)

```
const { expect } = require('chai')

describe('Suite Name', function() {
  describe('#method()', function() {
    it('should run without an error', async function() {
```

```
    const result = await answerToTheUltimateQuestion()
    expect(result).toBe.equal(42)
  })
})
})
```

Прочитайте Структуры модульного тестирования онлайн: <https://riptutorial.com/ru/node-js/topic/6731/структуры-модульного-тестирования>

---

# глава 102: Удаление Node.js

## Examples

### Полностью удалить Node.js на Mac OSX

В терминале в операционной системе Mac введите следующие две команды:

```
lsbom -f -l -s -pf /var/db/receipts/org.nodejs.pkg.bom | while read f; do sudo rm /usr/local/${f}; done

sudo rm -rf /usr/local/lib/node /usr/local/lib/node_modules /var/db/receipts/org.nodejs.*
```

### Удаление Node.js в Windows

Чтобы удалить Node.js в Windows, используйте «Установка и удаление программ»:

1. Откройте « Add or Remove Programs в меню «Пуск».
2. Найдите Node.js

#### Windows 10:

3. Нажмите Node.js.
4. Нажмите «Удалить».
5. Нажмите кнопку «Удалить».

#### Windows 7-8.1:

3. Нажмите кнопку «Удалить» в разделе «Node.js».

Прочитайте Удаление Node.js онлайн: <https://riptutorial.com/ru/node-js/topic/2821/удаление-node-js>

# глава 103: Удаленная отладка в Node.JS

## Examples

### Конфигурация запуска NodeJS

Чтобы настроить удаленную отладку узла, просто запустите процесс узла с помощью флага `--debug` . Вы можете добавить порт, на котором должен работать отладчик, с помощью `--debug=<port>` .

Когда процесс узла запускается, вы должны увидеть сообщение

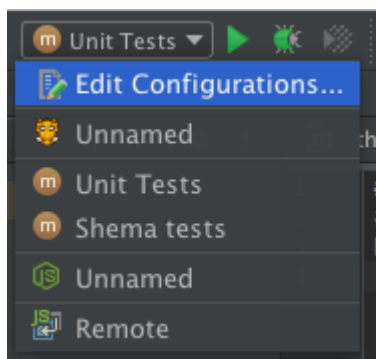
```
Debugger listening on port <port>
```

Что скажет вам, что все хорошо.

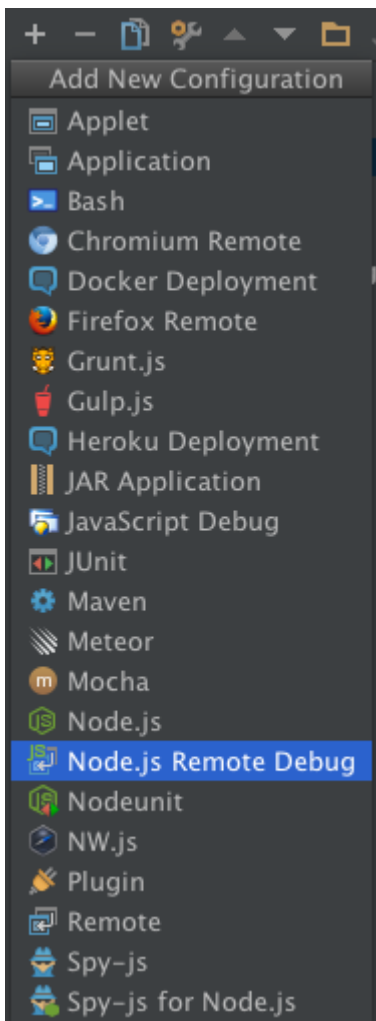
Затем вы настраиваете удаленную цель отладки в своей конкретной среде IDE.

### Настройка IntelliJ / Webstorm

1. Убедитесь, что подключен плагин NodeJS
2. Выберите конфигурацию прогона (экран)



3. Выберите + > **Node.js Удаленная отладка**



4. Убедитесь, что вы ввели указанный выше порт, а также правильный хост

A screenshot of the configuration form for 'Node.js Remote Debug'. The form has a dark background and contains the following fields: 'Name' with the value 'Remote', 'Host' with the value '127.0.0.1', and 'Port' with the value '5859'. There are also two checkboxes: 'Share' (unchecked) and 'Single instance only' (checked).

Как только они настроены, просто запустите цель отладки, как обычно, и она остановится на ваших контрольных точках.

## Использовать прокси для отладки через порт в Linux

Если вы запускаете приложение в Linux, используйте прокси для отладки через порт, например:

```
socat TCP-LISTEN:9958,fork TCP:127.0.0.1:5858 &
```

Затем используйте порт 9958 для удаленной отладки.

Прочитайте Удаленная отладка в Node.JS онлайн: <https://riptutorial.com/ru/node-js/topic/6335/>

[удаленная-отладка-в-node-js](#)



---

# глава 104: Управление ошибками Node.js

## Вступление

Мы узнаем, как создавать объекты `Error` и как бросать и обрабатывать ошибки в Node.js

Будущие изменения, связанные с передовыми методами обработки ошибок.

## Examples

### Создание объекта `Error`

#### новая ошибка (сообщение)

Создает новый объект ошибки, где для `message` значения задано свойство `message` созданного объекта. Обычно аргументы `message` передаются в конструктор ошибок в виде строки. Однако, если аргументом `message` является объект не строка, то конструктор ошибок вызывает метод `.toString()` переданного объекта и устанавливает это значение в свойство `message` созданного объекта ошибки.

```
var err = new Error("The error message");
console.log(err.message); //prints: The error message
console.log(err);
//output
//Error: The error message
//    at ...
```

Каждый объект ошибки имеет трассировку стека. Трассировка стека содержит информацию об ошибке и показывает, где произошла ошибка (приведенный выше вывод показывает стек ошибок). После создания объекта ошибки система фиксирует трассировку стека ошибки в текущей строке. Чтобы получить трассировку стека, используйте свойство стека для любого созданного объекта ошибки. Ниже две строки идентичны:

```
console.log(err);
console.log(err.stack);
```

#### Ошибка выброса

Ошибка броска означает исключение, если какое-либо исключение не обрабатывается, тогда сервер узла будет аварийно завершен.

Следующая строка вызывает ошибку:

```
throw new Error("Some error occurred");
```

или же

```
var err = new Error("Some error occurred");
throw err;
```

или же

```
throw "Some error occurred";
```

Последний пример (бросание строк) не является хорошей практикой и не рекомендуется (всегда бросайте ошибки, являющиеся экземплярами объекта Error).

Обратите внимание, что если вы `throw` ошибку в свою, то система выйдет из строя на этой строке (если нет обработчиков исключений), после этой строки никакой код не будет выполнен.

```
var a = 5;
var err = new Error("Some error message");
throw err; //this will print the error stack and node server will stop
a++; //this line will never be executed
console.log(a); //and this one also
```

Но в этом примере:

```
var a = 5;
var err = new Error("Some error message");
console.log(err); //this will print the error stack
a++;
console.log(a); //this line will be executed and will print 6
```

## попробуйте ... `catch` block

`try ... catch` block предназначен для обработки исключений, помните, что исключение означает сброшенную ошибку, а не ошибку.

```
try {
  var a = 1;
  b++; //this will cause an error because b is undefined
  console.log(b); //this line will not be executed
} catch (error) {
  console.log(error); //here we handle the error caused in the try block
}
```

В блоке `try` `b++` выдается ошибка и эта ошибка передается блоку `catch` который можно обрабатывать там или даже может быть вызвана той же ошибкой в блоке `catch` или сделать небольшую модификацию, а затем выбросить. Давайте посмотрим следующий пример.

```
try {
```

```
var a = 1;
b++;
console.log(b);
} catch (error) {
  error.message = "b variable is undefined, so the undefined can't be incremented"
  throw error;
}
```

В приведенном выше примере мы изменили свойство `message` объекта `error` и затем выбросили измененную `error`.

Вы можете выполнить любую ошибку в блоке `try` и обработать ее в блоке `catch`:

```
try {
  var a = 1;
  throw new Error("Some error message");
  console.log(a); //this line will not be executed;
} catch (error) {
  console.log(error); //will be the above thrown error
}
```

Прочитайте [Управление ошибками Node.js онлайн: https://riptutorial.com/ru/node-js/topic/8590/управление-ошибками-node-js](https://riptutorial.com/ru/node-js/topic/8590/управление-ошибками-node-js)

# глава 105: Усовершенствованный дизайн API: лучшие практики

## Examples

### Обработка ошибок: получить все ресурсы

Как вы обрабатываете ошибки, а не регистрируете их на консоли?

Плохой способ:

```
Router.route('/')
  .get((req, res) => {
    Request.find((err, r) => {
      if(err){
        console.log(err)
      } else {
        res.json(r)
      }
    })
  })
  .post((req, res) => {
    const request = new Request({
      type: req.body.type,
      info: req.body.info
    });
    request.info.user = req.user._id;
    console.log("ABOUT TO SAVE REQUEST", request);
    request.save((err, r) => {
      if (err) {
        res.json({ message: 'there was an error saving your r' });
      } else {
        res.json(r);
      }
    });
  });
});
```

Лучший путь:

```
Router.route('/')
  .get((req, res) => {
    Request.find((err, r) => {
      if(err){
        console.log(err)
      } else {
        return next(err)
      }
    })
  })
  .post((req, res) => {
    const request = new Request({
      type: req.body.type,
```

```
    info: req.body.info
  });
  request.info.user = req.user._id;
  console.log("ABOUT TO SAVE REQUEST", request);
  request.save((err, r) => {
    if (err) {
      return next(err)
    } else {
      res.json(r);
    }
  });
});
```

Прочитайте Усовершенствованный дизайн API: лучшие практики онлайн:

<https://riptutorial.com/ru/node-js/topic/6490/усовершенствованный-дизайн-апи--лучшие-практики>

---

# глава 106: Установка Node.js

## Examples

### Установите Node.js на Ubuntu

---

## Использование диспетчера пакетов apt

```
sudo apt-get update
sudo apt-get install nodejs
sudo apt-get install npm
sudo ln -s /usr/bin/nodejs /usr/bin/node

# the node & npm versions in apt are outdated. This is how you can update them:
sudo npm install -g npm
sudo npm install -g n
sudo n stable # (or lts, or a specific version)
```

---

## Использование последней версии конкретной версии (например, LTS 6.x) непосредственно из nodeourse

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
apt-get install -y nodejs
```

Кроме того, для правильного способа установки глобальных модулей npm, установите для них личный каталог (исключает необходимость в sudo и позволяет избежать ошибок EACCES):

```
mkdir ~/.npm-global
echo "export PATH=~/.npm-global/bin:$PATH" >> ~/.profile
source ~/.profile
npm config set prefix '~/.npm-global'
```

### Установка Node.js в Windows

#### Стандартная установка

Все файлы Node.js, инсталляторы и исходные файлы можно загрузить [здесь](#) .

Вы можете загрузить только время выполнения `node.exe` или использовать установщик

Windows ( `.msi` ), который также установит `npm` , рекомендуемый менеджер пакетов для Node.js и настроит пути.

## Установка менеджером пакетов

Вы также можете установить менеджер пакетов [Chocolatey](#) (Software Management Automation).

```
# choco install nodejs.install
```

Более подробная информация о текущей версии, вы можете найти в хранилище шоколадного [здесь](#) .

## Использование Node Version Manager (nvm)

[Node Version Manager](#) , иначе известный как `nvm` , является скриптом `bash` , который упрощает управление несколькими версиями Node.js.

Чтобы установить `nvm` , используйте прилагаемый сценарий установки:

```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

Для окон есть пакет `nvm-windows` с установщиком. На [этой](#) странице [GitHub](#) есть информация для установки и использования пакета `nvm-windows`.

После установки `nvm` запустите «`nvm on`» из командной строки. Это позволяет `nvm` управлять версиями узлов.

*Примечание.* Возможно, вам потребуется перезапустить терминал, чтобы он распознал недавно установленную команду `nvm` .

Затем установите последнюю версию узла:

```
$ nvm install node
```

Вы также можете установить определенную версию узла, передав основные, второстепенные и / или патч-версии:

```
$ nvm install 6  
$ nvm install 4.2
```

Чтобы просмотреть версии, доступные для установки:

```
$ nvm ls-remote
```

Затем вы можете переключать версии, передавая версию так же, как и при установке:

```
$ nvm use 5
```

Вы можете установить определенную версию Узел, который вы установили, для **версии по умолчанию** , указав:

```
$ nvm alias default 4.2
```

Чтобы отобразить список версий узлов, установленных на вашем компьютере, введите:

```
$ nvm ls
```

Чтобы использовать версии узлов конкретного проекта, вы можете сохранить версию в файле `.nvmrc`. Таким образом, работа с другим проектом будет менее подвержена ошибкам после извлечения из своего репозитория.

```
$ echo "4.2" > .nvmrc
$ nvm use
Found '/path/to/project/.nvmrc' with version <4.2>
Now using node v4.2 (npm v3.7.3)
```

Когда Node установлен через `nvm`, нам не нужно использовать `sudo` для установки глобальных пакетов, поскольку они установлены в домашней папке. Таким образом, `npm i -g http-server` работает без каких-либо ошибок разрешения.

## Установить Node.js из источника с помощью диспетчера пакетов APT

### Предпосылки

```
sudo apt-get install build-essential
sudo apt-get install python

[optional]
sudo apt-get install git
```

### Получить источник и построить

```
cd ~
git clone https://github.com/nodejs/node.git
```

### ИЛИ Для последней версии LTS Node.js 6.10.2

```
cd ~
wget https://nodejs.org/dist/v6.3.0/node-v6.10.2.tar.gz
tar -xzf node-v6.10.2.tar.gz
```

Перейдите в исходный каталог, например, в `cd ~/node-v6.10.2`



```
./configure
make
sudo make install
```

## Установка Node.js на Mac с помощью диспетчера пакетов

### Homebrew

Вы можете установить Node.js с помощью диспетчера пакетов [Homebrew](#) .

Начните с обновления варева:

```
brew update
```

Вам может потребоваться изменить разрешения или пути. Лучше всего запустить это, прежде чем продолжить:

```
brew doctor
```

Затем вы можете установить Node.js, запустив:

```
brew install node
```

После установки Node.js вы можете проверить версию, установленную при запуске:

```
node -v
```

### MacPorts

Вы также можете установить node.js через [Macports](#) .

Сначала обновите его, чтобы убедиться, что ссылки на последние пакеты:

```
sudo port selfupdate
```

Затем установите nodejs и npm

```
sudo port install nodejs npm
```

Теперь вы можете запускать узел через CLI напрямую, вызывая `node` . Кроме того, вы можете проверить свою текущую версию узла с помощью

```
node -v
```

## Установка с помощью MacOS X Installer

Вы можете найти установщиков на [странице загрузки Node.js](#). Обычно Node.js рекомендует две версии Node, версию LTS (долгосрочную поддержку) и текущую версию (последняя версия). Если вы новичок в узле, просто зайдите в LTS, а затем нажмите кнопку `Macintosh Installer`, чтобы загрузить пакет.

Если вы хотите найти другие выпуски NodeJS, перейдите [сюда](#), выберите выпуск и нажмите «Загрузить». На странице загрузки найдите файл с расширением `.pkg`.

После того, как вы скачали файл (с расширением `.pkg` ofcourse), дважды щелкните его, чтобы установить. Установщик, упакованный с `Node.js` и `npm`, по умолчанию, пакет будет устанавливать оба, но вы можете настроить, какой из них установить, нажав кнопку `customize` на шаге «`Installation Type`». Помимо этого, следуйте инструкциям по установке, это довольно просто.

---

## Проверьте, установлен ли узел

Открытый `terminal` (если вы не знаете, как открыть терминал, посмотрите на это [wikihow](#)). Затем в терминальном типе `node --version` введите. Ваш терминал будет выглядеть так, если установлен узел:

```
$ node --version
v7.2.1
```

Версия `v7.2.1` - это ваша версия Node.js, если вы не получили команду сообщения `command not found: node` вместо этого, значит, есть проблема с вашей установкой.

## Установка Node.js на малиновый PI

Чтобы установить v6.x обновление пакетов

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
```

Использование диспетчера пакетов apt

```
sudo apt-get install -y nodejs
```

## Установка с помощью диспетчера версий узлов под Fish Shell с помощью моей рыбы!

[Node Version Manager](#) (`nvm`) значительно упрощает управление версиями Node.js, их установку и устраняет необходимость использования `sudo` при работе с пакетами

(например, `npm install ...`). [Fish Shell](#) (`fish`) » - это интеллектуальная и удобная оболочка командной строки для OS X, Linux и остальной семьи », которая является популярной альтернативой программистам для обычных оболочек, таких как `bash`. Наконец, [Oh My Fish](#) (`omf`) позволяет настраивать и устанавливать пакеты в оболочке Fish.

**В этом руководстве предполагается, что вы уже используете Fish в качестве оболочки .**

## Установить nvm

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.4/install.sh | bash
```

## Установите Oh My Fish

```
curl -L https://github.com/oh-my-fish/oh-my-fish/raw/master/bin/install | fish
```

(Примечание: вам будет предложено перезагрузить терминал в этот момент. Идите вперед и сделайте это сейчас.)

## Установите плагин-nvm для Oh My Fish

Мы установим [plugin-nvm](#) через Oh My Fish, чтобы выявить возможности `nvm` в оболочке Fish:

```
omf install nvm
```

## Установка Node.js с помощью диспетчера версий узлов

Теперь вы готовы использовать `nvm`. Вы можете установить и использовать версию Node.js по своему вкусу. Некоторые примеры:

- Установите последнюю версию узла: `nvm install node`
- Установка 6.3.1: `nvm install 6.3.1`
- Список установленных версий: `nvm ls`
- Переключиться на ранее установленное 4.3.1: `nvm use 4.3.1`

## Итоговые заметки

Помните еще раз, что нам больше не нужно `sudo` при работе с Node.js, используя этот метод! Варианты узлов, пакеты и т. Д. Устанавливаются в вашем домашнем каталоге.

## Установите Node.js из источника на Centos, RHEL и Fedora

### Предпосылки

- мерзавец
- `clang` и `clang++` 3.4 ^ или `gcc` и `g++` 4.8 ^
- Python 2.6 или 2.7
- GNU Make 3.81 ^

## Получить источник

### Node.js v6.x LTS

```
git clone -b v6.x https://github.com/nodejs/node.git
```

### Node.js v7.x

```
git clone -b v7.x https://github.com/nodejs/node.git
```

## строить

```
cd node
./configure
make -jX
su -c make install
```

*X - количество процессорных ядер, значительно ускоряет сборку*

## Очистка [Дополнительно]

```
cd
rm -rf node
```

## Установка Node.js с n

Во-первых, есть действительно хорошая оболочка для настройки n в вашей системе. Просто беги:

```
curl -L https://git.io/n-install | bash
```

для установки n . Затем установите двоичные файлы различными способами:

### самый последний

```
n latest
```

### стабильный

```
n stable
```

### LTS

```
n lts
```

### Любая другая версия

```
n <version>
```

например, n 4.4.7

Если эта версия уже установлена, эта команда активирует эту версию.

## Переключение версий

n сам создаст список выбора установленных двоичных файлов. Используйте вверх и вниз, чтобы найти тот, который вы хотите, и Enter, чтобы активировать его.

Прочитайте [Установка Node.js онлайн: https://riptutorial.com/ru/node-js/topic/1294/установка-node-js](https://riptutorial.com/ru/node-js/topic/1294/установка-node-js)

# глава 107: Файл загружен

## Examples

### Загрузка одного файла с использованием multer

#### Запомни

- создать папку для загрузки ( uploads в пример).
- установить multer `npm i -S multer`

#### server.js :

```
var express = require("express");
var multer = require('multer');
var app = express();
var fs = require('fs');

app.get('/', function(req, res) {
    res.sendFile(__dirname + "/index.html");
});

var storage = multer.diskStorage({
    destination: function (req, file, callback) {
        fs.mkdir('./uploads', function(err) {
            if(err) {
                console.log(err.stack)
            } else {
                callback(null, './uploads');
            }
        })
    },
    filename: function (req, file, callback) {
        callback(null, file.fieldname + '-' + Date.now());
    }
});

app.post('/api/file', function(req, res) {
    var upload = multer({ storage : storage}).single('userFile');
    upload(req, res, function(err) {
        if(err) {
            return res.end("Error uploading file.");
        }
        res.end("File is uploaded");
    });
});

app.listen(3000, function() {
    console.log("Working on port 3000");
});
```

#### index.html :

```
<form id      = "uploadForm"
  enctype     = "multipart/form-data"
  action      = "/api/file"
  method      = "post"
>
<input type="file" name="userFile" />
<input type="submit" value="Upload File" name="submit">
</form>
```

---

## Замечания:

Чтобы загрузить файл с расширением, вы можете использовать встроенную библиотеку Node.js [path](#)

Для этого просто нужно `server.js` path к файлу `server.js` :

```
var path = require('path');
```

и изменение:

```
callback(null, file.fieldname + '-' + Date.now());
```

добавив расширение файла следующим образом:

```
callback(null, file.fieldname + '-' + Date.now() + path.extname(file.originalname));
```

---

## Как фильтровать загрузку по расширению:

В этом примере посмотрите, как загружать файлы, чтобы разрешить только определенные расширения.

Например, только расширения изображений. Просто добавьте в `var upload = multer({ storage : storage}).single('userFile');` условие `fileFilter`

```
var upload = multer({
  storage: storage,
  fileFilter: function (req, file, callback) {
    var ext = path.extname(file.originalname);
    if(ext !== '.png' && ext !== '.jpg' && ext !== '.gif' && ext !== '.jpeg') {
      return callback(new Error('Only images are allowed'))
    }
    callback(null, true)
  }
}).single('userFile');
```

Теперь вы можете загружать только файлы изображений с расширениями `png`, `jpg`, `gif` или `jpeg`

## Использование грозного модуля

Установить модуль и прочитать [документы](#)

```
npm i formidable@latest
```

### Пример сервера на порту 8080

```
var formidable = require('formidable'),
    http = require('http'),
    util = require('util');

http.createServer(function(req, res) {
  if (req.url == '/upload' && req.method.toLowerCase() == 'post') {
    // parse a file upload
    var form = new formidable.IncomingForm();

    form.parse(req, function(err, fields, files) {
      if (err)
        do-smth; // process error

      // Copy file from temporary place
      // var fs = require('fs');
      // fs.rename(file.path, <targetPath>, function (err) { ... });

      // Send result on client
      res.writeHead(200, {'content-type': 'text/plain'});
      res.write('received upload:\n\n');
      res.end(util.inspect({fields: fields, files: files}));
    });

    return;
  }

  // show a file upload form
  res.writeHead(200, {'content-type': 'text/html'});
  res.end(
    '<form action="/upload" enctype="multipart/form-data" method="post">'+
    '<input type="text" name="title"><br>'+
    '<input type="file" name="upload" multiple="multiple"><br>'+
    '<input type="submit" value="Upload">'+
    '</form>';
  );
}).listen(8080);
```

Прочитайте [Файл загружен онлайн: https://riptutorial.com/ru/node-js/topic/4080/файл-загружен](https://riptutorial.com/ru/node-js/topic/4080/файл-загружен)



# глава 108: Хороший стиль кодирования

## замечания

Я бы рекомендовал начинающему начинать с этого стиля кодирования. И если кто-то может предложить лучший способ (ps, я выбрал эту технику и эффективно работаю для меня в приложении, используемом более чем 100 000 пользователей), не стесняйтесь принимать какие-либо предложения. ТИА.

## Examples

### Основная программа регистрации

В этом примере будет объяснено разделение кода **node.js** на разные **модули / папки** для лучшей совместимости. Следуя этому методу, разработчикам проще понять код, поскольку он может напрямую ссылаться на соответствующий файл, а не на весь код. Основное использование - это когда вы работаете в команде, а новый разработчик присоединяется к более позднему этапу, ему будет легче прилипнуть к самому коду.

**index.js** : - Этот файл будет управлять подключением к серверу.

```
//Import Libraries
var express = require('express'),
    session = require('express-session'),
    mongoose = require('mongoose'),
    request = require('request');

//Import custom modules
var userRoutes = require('./app/routes/userRoutes');
var config = require('./app/config/config');

//Connect to Mongo DB
mongoose.connect(config.getDBString());

//Create a new Express application and Configure it
var app = express();

//Configure Routes
app.use(config.API_PATH, userRoutes());

//Start the server
app.listen(config.PORT);
console.log('Server started at - '+ config.URL+ ":" +config.PORT);
```

**config.js** : -Этот файл будет управлять всеми параметрами, связанными с конфигурацией, которые останутся **такими** же на всем протяжении.

```
var config = {
```

```

VERSION: 1,
BUILD: 1,
URL: 'http://127.0.0.1',
API_PATH : '/api',
PORT : process.env.PORT || 8080,
DB : {
  //MongoDB configuration
  HOST : 'localhost',
  PORT : '27017',
  DATABASE : 'db'
},
/*
 * Get DB Connection String for connecting to MongoDB database
 */
getDBString : function(){
  return 'mongodb://' + this.DB.HOST + ':' + this.DB.PORT + '/' + this.DB.DATABASE;
},
/*
 * Get the http URL
 */
getHTTPOurl : function(){
  return 'http://' + this.URL + ":" + this.PORT;
}

module.exports = config;

```

## user.js : - Файл модели, в котором определена схема

```

var mongoose = require('mongoose');
var Schema = mongoose.Schema;

//Schema for User
var UserSchema = new Schema({
  name: {
    type: String,
    // required: true
  },
  email: {
    type: String
  },
  password: {
    type: String,
    //required: true
  },
  dob: {
    type: Date,
    //required: true
  },
  gender: {
    type: String, // Male/Female
    // required: true
  }
});

//Define the model for User
var User;
if(mongoose.models.User)
  User = mongoose.model('User');

```

```

else
  User = mongoose.model('User', UserSchema);

//Export the User Model
module.exports = User;

```

## UserController : - Этот файл содержит функцию для пользователя signUp

```

var User = require('../models/user');
var crypto = require('crypto');

//Controller for User
var UserController = {

  //Create a User
  create: function(req, res){
    var repassword = req.body.repassword;
    var password = req.body.password;
    var userEmail = req.body.email;

    //Check if the email address already exists
    User.find({"email": userEmail}, function(err, usr){
      if(usr.length > 0){
        //Email Exists

        res.json('Email already exists');
        return;
      }
      else
      {
        //New Email

        //Check for same passwords
        if(password != repassword){
          res.json('Passwords does not match');
          return;
        }

        //Generate Password hash based on sha1
        var shasum = crypto.createHash('sha1');
        shasum.update(req.body.password);
        var passwordHash = shasum.digest('hex');

        //Create User
        var user = new User();
        user.name = req.body.name;
        user.email = req.body.email;
        user.password = passwordHash;
        user.dob = Date.parse(req.body.dob) || "";
        user.gender = req.body.gender;

        //Validate the User
        user.validate(function(err){
          if(err){
            res.json(err);
            return;
          }
          else{
            //Finally save the User
            user.save(function(err){
              if(err)

```

```

        {
            res.json(err);
            return;
        }

        //Remove Password before sending User details
        user.password = undefined;
        res.json(user);
        return;
    });
}
});
}
});
}

module.exports = UserController;

```

### userRoutes.js : - Этот маршрут для userController

```

var express = require('express');
var UserController = require('../controllers/userController');

//Routes for User
var UserRoutes = function(app)
{
    var router = express.Router();

    router.route('/users')
        .post(UserController.create);

    return router;
}

module.exports = UserRoutes;

```

Вышеприведенный пример может показаться слишком большим, но если новичок в node.js с небольшой смесью экспресс-знаний попытается пройти через это, это будет легко и полезно.

Прочитайте Хороший стиль кодирования онлайн: <https://riptutorial.com/ru/node-js/topic/6489/хороший-стиль-кодирования>

---

# глава 109: хрюкать

## замечания

### Дальнейшее чтение:

Руководство по [установке Grunt](#) содержит подробную информацию об установке конкретных, производственных или разрабатываемых версий Grunt и grunt-cli.

Руководство по [настройке задач](#) содержит подробное объяснение того, как настраивать задачи, цели, параметры и файлы внутри файла Grunt, а также объяснять шаблоны, шаблоны глобусов и импортировать внешние данные.

В [руководстве](#) «[Создание задач](#)» перечислены различия между типами задач Grunt и показаны некоторые примеры задач и конфигураций.

## Examples

### Введение в GruntJs

Grunt - это бегун для задач JavaScript, используемый для автоматизации повторяющихся задач, таких как минификация, компиляция, модульное тестирование, листинг и т. Д.

Чтобы начать работу, вы захотите установить интерфейс командной строки Grunt (CLI) по всему миру.

```
npm install -g grunt-cli
```

**Подготовка нового проекта Grunt:** типичная настройка включает в себя добавление двух файлов в ваш проект: package.json и Gruntfile.

package.json: Этот файл используется npm для хранения метаданных для проектов, опубликованных как модули npm. Вы перечислите grunt и плагины Grunt, необходимые вашему проекту как devDependencies в этом файле.

Gruntfile: этот файл называется Gruntfile.js и используется для настройки или определения задач и загрузки плагинов Grunt.

Example package.json:

```
{
  "name": "my-project-name",
  "version": "0.1.0",
  "devDependencies": {
    "grunt": "~0.4.5",
```

```
"grunt-contrib-jshint": "~0.10.0",
"grunt-contrib-nodeunit": "~0.4.1",
"grunt-contrib-uglify": "~0.5.0"
}
}
```

## Пример файла grunt:

```
module.exports = function(grunt) {

  // Project configuration.
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    uglify: {
      options: {
        banner: '/*! <%= pkg.name %> <%= grunt.template.today("yyyy-mm-dd") %> */\n'
      },
      build: {
        src: 'src/<%= pkg.name %>.js',
        dest: 'build/<%= pkg.name %>.min.js'
      }
    }
  });

  // Load the plugin that provides the "uglify" task.
  grunt.loadNpmTasks('grunt-contrib-uglify');

  // Default task(s).
  grunt.registerTask('default', ['uglify']);

};
```

## Установка gruntplugins

### Добавление зависимостей

Чтобы использовать `gruntplugin`, вам сначала нужно добавить его в качестве зависимости от вашего проекта. Давайте используем плагин `jshint` в качестве примера.

```
npm install grunt-contrib-jshint --save-dev
```

Опция `--save-dev` используется для добавления плагина в `package.json`, таким образом, плагин всегда устанавливается после `npm install`.

### Загрузка плагина

Вы можете загрузить свой плагин в файле `loadNpmTasks` с помощью `loadNpmTasks`.

```
grunt.loadNpmTasks('grunt-contrib-jshint');
```

### Настройка задачи

Вы настраиваете задачу в файле `grunt`, добавляя свойство `jshint` к объекту, переданному в

```
grunt.initConfig .
```

```
grunt.initConfig({  
  jshint: {  
    all: ['Gruntfile.js', 'lib/**/*.js', 'test/**/*.js']  
  }  
});
```

Не забывайте, что вы можете использовать другие свойства для других плагинов, которые вы используете.

## Выполнение задачи

Чтобы просто запустить задачу с помощью плагина, вы можете использовать командную строку.

```
grunt jshint
```

Или вы можете добавить `jshint` в другую задачу.

```
grunt.registerTask('default', ['jshint']);
```

Задача по умолчанию выполняется с помощью команды `grunt` в терминале без каких-либо параметров.

Прочитайте хрюкать онлайн: <https://riptutorial.com/ru/node-js/topic/6059/хрюкать>

---

# глава 110: Экспорт и импорт модуля в node.js

## Examples

### Использование простого модуля в node.js

Что такое модуль node.js ( [ссылка на статью](#) ):

Модуль инкапсулирует связанный код в единую единицу кода. При создании модуля это можно интерпретировать как перемещение всех связанных функций в файл.

Теперь давайте посмотрим пример. Представьте, что все файлы находятся в одном каталоге:

Файл: printer.js

```
"use strict";

exports.printHelloWorld = function () {
  console.log("Hello World!!!");
}
```

Другой способ использования модулей:

Файл animals.js

```
"use strict";

module.exports = {
  lion: function() {
    console.log("ROAARR!!!");
  }
};
```

Файл: app.js

Запустите этот файл, перейдя в ваш каталог и набрав: `node app.js`

```
"use strict";

//require('./path/to/module.js') node which module to load
var printer = require('./printer');
var animals = require('./animals');

printer.printHelloWorld(); //prints "Hello World!!!"
```



```
animals.lion(); //prints "ROAARR!!!"
```

## Использование импорта в ES6

Node.js построен против современных версий V8. Поддерживая новейшие версии этого движка, мы гарантируем, что новые функции из спецификации JavaScript ECMA-262 будут доведены до разработчиков Node.js своевременно, а также продолжительные улучшения производительности и стабильности.

Все функции ECMAScript 2015 (ES6) разделены на три группы для доставки, постановки и выполнения функций:

Все функции доставки, которые V8 считает стабильными, включаются по умолчанию на Node.js и HE требуют какого-либо флага времени выполнения. Поэтапные функции, которые являются почти завершенными функциями, которые не считают стабильными командой V8, требуют флага времени выполнения: `--harmony`. В процессе выполнения функции могут быть активированы индивидуально по их соответствующему флагом гармонии, хотя это крайне не рекомендуется, если только для целей тестирования. Примечание. Эти флаги отображаются V8 и могут изменяться без уведомления об устаревании.

В настоящее время ES6 поддерживает операторы импорта изначально [см здесь](#)

Поэтому, если у нас есть файл `fun.js` ...

```
export default function say(what){
  console.log(what);
}

export function sayLoud(whoot) {
  say(whoot.toUpperCase());
}
```

... и если был еще один файл с именем `app.js` где мы хотим использовать наши ранее определенные функции, существует три способа их импорта.

### Импорт по умолчанию

```
import say from './fun';
say('Hello Stack Overflow!!'); // Output: Hello Stack Overflow!!
```

Импортирует функцию `say()` поскольку она помечена как экспорт по умолчанию в исходный файл ( `export default ...` )

### Именованный импорт

```
import { sayLoud } from './fun';
```

```
sayLoud('JS modules are awesome.');// Output: JS MODULES ARE AWESOME.
```

Именованный импорт позволяет нам импортировать именно те части модуля, которые нам действительно нужны. Мы делаем это, явно называя их. В нашем случае, назвав `sayLoud` в фигурных скобках в инструкции `import`.

## Вложенный импорт

```
import * as i from './fun';  
i.say('What?');// Output: What?  
i.sayLoud('Whoot!');// Output: WHOOT!
```

Если мы хотим иметь все это, это путь. Используя синтаксис `* as i` мы имеем оператор `import` предоставляет нам объект `i` который содержит весь экспорт нашего модуля `fun` как соответствующие им свойства.

## пути

Имейте в виду, что вы должны явно указывать пути импорта как *относительные пути*, даже если импортируемый файл находится в том же каталоге, что и файл, в который вы импортируете, используя `./`. Импорт из неподписанных дорожек

```
import express from 'express';
```

будут просматриваться в локальных и глобальных папках `node_modules` и будут `node_modules` ошибку, если не найдены соответствующие модули.

## Экспорт с использованием синтаксиса ES6

Это эквивалентно [другому примеру](#), но вместо этого используется ES6.

```
export function printHelloWorld() {  
  console.log("Hello World!!!");  
}
```

Прочитайте Экспорт и импорт модуля в node.js онлайн: <https://riptutorial.com/ru/node-js/topic/1173/экспорт-и-импорт-модуля-в-node-js>

---

# глава 111: Экспорт и потребление модулей

## замечания

Хотя все в Node.js обычно выполняется асинхронно, `require()` не является одной из этих вещей. Так как модули на практике нужно загружать только один раз, это операция блокировки и должна использоваться должным образом.

Модули кэшируются после первого их загрузки. Если вы хотите редактировать модуль в разработке, вам нужно будет удалить его запись в кеше модуля, чтобы использовать новые изменения. При этом, даже если модуль очищен от кеша модуля, сам модуль не является сборкой мусора, поэтому следует проявлять осторожность при его использовании в производственных средах.

## Examples

### Загрузка и использование модуля

Модуль может быть «импортирован», иначе «требуется» функцией `require()`. Например, чтобы загрузить модуль `http` который поставляется с Node.js, можно использовать следующее:

```
const http = require('http');
```

Помимо модулей, поставляемых вместе со средой выполнения, вам также могут потребоваться модули, установленные вами из числа npm, например `express`. Если вы уже установили `express` в своей системе через `npm install express`, вы можете просто написать:

```
const express = require('express');
```

Вы также можете включить модули, которые вы написали сами, как часть вашего приложения. В этом случае включить файл с именем `lib.js` в том же каталоге, что и текущий файл:

```
const mylib = require('./lib');
```

Обратите внимание, что вы можете опустить расширение, и предполагается, что `.js`. После загрузки модуля переменная заполняется объектом, который содержит методы и свойства, опубликованные из требуемого файла. Полный пример:

```
const http = require('http');  
  
// The `http` module has the property `STATUS_CODES`
```

```
console.log(http.STATUS_CODES[404]); // outputs 'Not Found'

// Also contains `createServer()`
http.createServer(function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write('<html><body>Module Test</body></html>');
  res.end();
}).listen(80);
```

## Создание модуля hello-world.js

Узел предоставляет интерфейс `module.exports` для `module.exports` функций и переменных другим файлам. Самый простой способ сделать это - экспортировать только один объект (функцию или переменную), как показано в первом примере.

### привет-world.js

```
module.exports = function(subject) {
  console.log('Hello ' + subject);
};
```

Если мы не хотим, чтобы весь экспорт был единственным объектом, мы можем экспортировать функции и переменные в качестве свойств объекта `exports`. Эти три следующих примера демонстрируют это несколько по-разному:

- `hello-venus.js`: определение функции выполняется отдельно, затем добавляется как **СВОЙСТВО** `module.exports`
- `hello-jupiter.js`: определения функций непосредственно помещаются как значение **СВОЙСТВ** `module.exports`
- `hello-mars.js`: определение функции напрямую объявляется как **свойство** `exports` которое является короткой версией `module.exports`

### привет-venus.js

```
function hello(subject) {
  console.log('Venus says Hello ' + subject);
}

module.exports = {
  hello: hello
};
```

### привет-jupiter.js

```
module.exports = {
  hello: function(subject) {
    console.log('Jupiter says hello ' + subject);
  },

  bye: function(subject) {
    console.log('Jupiter says goodbye ' + subject);
  }
};
```

```
    }  
};
```

## привет-mars.js

```
exports.hello = function(subject) {  
    console.log('Mars says Hello ' + subject);  
};
```

## Модуль загрузки с именем каталога

У нас есть каталог с именем `hello` который включает в себя следующие файлы:

### index.js

```
// hello/index.js  
module.exports = function(){  
    console.log('Hej');  
};
```

### main.js

```
// hello/main.js  
// We can include the other files we've defined by using the `require()` method  
var hw = require('./hello-world.js'),  
    hm = require('./hello-mars.js'),  
    hv = require('./hello-venus.js'),  
    hj = require('./hello-jupiter.js'),  
    hu = require('./index.js');  
  
// Because we assigned our function to the entire `module.exports` object, we  
// can use it directly  
hw('World!'); // outputs "Hello World!"  
  
// In this case, we assigned our function to the `hello` property of exports, so we must  
// use that here too  
hm.hello('Solar System!'); // outputs "Mars says Hello Solar System!"  
  
// The result of assigning module.exports at once is the same as in hello-world.js  
hv.hello('Milky Way!'); // outputs "Venus says Hello Milky Way!"  
  
hj.hello('Universe!'); // outputs "Jupiter says hello Universe!"  
hj.bye('Universe!'); // outputs "Jupiter says goodbye Universe!"  
  
hu(); //output 'hej'
```

## Недействительность кэша модуля

В процессе разработки вы можете обнаружить, что использование `require()` на одном модуле несколько раз всегда возвращает тот же модуль, даже если вы внесли изменения в этот файл. Это связано с тем, что при первом загрузке модулей кэшируются, и любые последующие загрузки модулей загружаются из кеша.

Чтобы обойти эту проблему, вам нужно будет `delete` запись в кеше. Например, если вы загрузили модуль:

```
var a = require('./a');
```

Затем вы можете удалить запись кэша:

```
var rpath = require.resolve('./a.js');
delete require.cache[rpath];
```

Затем снова потребуется модуль:

```
var a = require('./a');
```

Обратите внимание, что это не рекомендуется в процессе производства, поскольку `delete` удаляет ссылку только на загруженный модуль, а не на загруженные данные. Модуль не собирает мусор, поэтому неправильное использование этой функции может привести к утечке памяти.

## Создание собственных модулей

Вы также можете ссылаться на объект для публичного экспорта и непрерывного добавления методов к этому объекту:

```
const auth = module.exports = {}
const config = require('../config')
const request = require('request')

auth.email = function (data, callback) {
  // Authenticate with an email address
}

auth.facebook = function (data, callback) {
  // Authenticate with a Facebook account
}

auth.twitter = function (data, callback) {
  // Authenticate with a Twitter account
}

auth.slack = function (data, callback) {
  // Authenticate with a Slack account
}

auth.stack_overflow = function (data, callback) {
  // Authenticate with a Stack Overflow account
}
```

Чтобы использовать любой из них, просто требуется модуль, как обычно:

```
const auth = require('./auth')
```

```
module.exports = function (req, res, next) {
  auth.facebook(req.body, function (err, user) {
    if (err) return next(err)

    req.user = user
    next()
  })
}
```

## Каждый модуль вводится только один раз

NodeJS выполняет модуль только в первый раз, когда вы его требуете. Любые дополнительные функции требуют повторного использования одного и того же объекта, поэтому в противном случае не будет выполняться код в модуле. Кроме того, Node кэширует модули в первый раз, когда они загружаются с использованием `require`. Это уменьшает количество чтения файлов и помогает ускорить работу приложения.

`myModule.js`

```
console.log(123) ;
exports.var1 = 4 ;
```

`index.js`

```
var a=require('./myModule') ; // Output 123
var b=require('./myModule') ; // No output
console.log(a.var1) ; // Output 4
console.log(b.var1) ; // Output 4
a.var2 = 5 ;
console.log(b.var2) ; // Output 5
```

## Загрузка модуля из `node_modules`

Модулям может `require` д без использования относительных путей, помещая их в специальный каталог `node_modules` .

Например, чтобы `require` модуль под названием `foo` из файла `index.js` , вы можете использовать следующую структуру каталогов:

```
index.js
├─ node_modules
│  └─ foo
│     └─ foo.js
└─ package.json
```

Модули должны быть размещены внутри каталога вместе с файлом `package.json` . `main` поле файла `package.json` должно указывать на точку входа для вашего модуля - это файл, который импортируется, когда пользователи этого `require('your-module')` . `main` значения по умолчанию - `index.js` если они не `index.js` . Кроме того, вы можете ссылаться на файлы

относительно вашего модуля, просто добавив относительный путь к `require` вызова:

```
require('your-module/path/to/file') .
```

`node_modules` также может `require` `d` от `node_modules` до иерархии файловой системы. Если у нас есть следующая структура каталогов:

```
my-project
├─ node_modules
│  ├─ foo    // the foo module
│  └─ ...
├─ baz    // the baz module
│  └─ node_modules
│     └─ bar    // the bar module
```

мы сможем `require` модуль `foo` из любого файла в `bar` используя `require('foo')` .

Обратите внимание, что узел будет соответствовать только самому модулю, находящемуся ближе всего к файлу в иерархии файловой системы, начиная с (текущий каталог файла / `node_modules`). Узел сопоставляет каталоги таким образом до корня файловой системы.

Вы можете либо установить новые модули из реестра npm, либо в другие реестры npm или создать свои собственные.

## Папка как модуль

Модули можно разделить на многие `.js`-файлы в одной папке. Пример в папке `my_module` :

### function\_one.js

```
module.exports = function() {
  return 1;
}
```

### function\_two.js

```
module.exports = function() {
  return 2;
}
```

### index.js

```
exports.f_one = require('./function_one.js');
exports.f_two = require('./function_two.js');
```

Модуль, подобный этому, используется, ссылаясь на него по имени папки:

```
var split_module = require('./my_module');
```

Обратите внимание, что если вам это необходимо, опустив `./` или любое указание пути к



папке из аргумента функции `require`, Node попытается загрузить модуль из папки `node_modules` .

В качестве альтернативы вы можете создать в той же папке файл `package.json` с ЭТИМ содержимым:

```
{
  "name": "my_module",
  "main": "./your_main_entry_point.js"
}
```

Таким образом, вам не нужно указывать основной файл модуля «index».

Прочитайте Экспорт и потребление модулей онлайн: <https://riptutorial.com/ru/node-js/topic/547/экспорт-и-потребление-модулей>

# кредиты

S. No	Главы	Contributors
1	Начало работы с Node.js	4444, Abdelaziz Mokhnache, Abhishek Jain, Adam, Aeolingamenfel, Alessandro Trinca Tornidor, Aljoscha Meyer, Amila Sampath, Ankit Gomkale, Ankur Anand, arcs, Aule, B Thuy, baranskistad, Bundit J., Chandra Sekhar, Chezzwizz, Christopher Ronning, Community, Craig Ayre, David Gatti, Djizeus, Florian Hämmerle, Franck Dernoncourt, ganesshkumar, George Aidonidis, Harangue, hexacyanide, Iain Reid, Inanc Gumus, Jason, Jasper, Jeremy Banks, John Slegers, JohnnyCoder, Joshua Kleveter, KolesnichenkoDS, krishgopinath, Léo Martin, Majid, Marek Skiba, Matt Bush, Meinkraft, Michael Irigoyen, Mikhail, Milan Laslop, ndugger, Nick, olegzhermal, Peter Mortensen, RamenChef, Reborn, Rishikesh Chandra, Shabin Hashim, Shiven, Sibeesh Venu, sigfried, SteveLacy, Susanne Oberhauser, thefourtheye, theunexpected1, Tomás Cañibano, user2314737, Volodymyr Sichka, xam, zurfyx
2	async.js	David Knipe, devnull69, DrakaSAN, F. Kauder, jerry, Isampaio, Shriganesh Kolhe, Sky, walid
3	CLI	Ze Rubeus
4	csv парсер в узле js	aisflat439
5	ECMAScript 2015 (ES6) с Node.js	David Xu, Florian Hämmerle, Osama Bari
6	Eventloop	Kelum Senanayake
7	HTTP	Ahmed Metwally
8	Koa Framework v2	David Xu
9	Lodash	M1kstur
10	Loopback - разъем на основе REST	Roopesh
11	N-API,	Parham Alvani
12	Node.js (express.js) с угловым.js Пример	sigfried

кода		
13	Node.js v6 Новые возможности и улучшения	<a href="#">creyD</a> , <a href="#">DominicValenciana</a> , <a href="#">KlwntSingh</a>
14	Node.JS и MongoDB.	<a href="#">midnightsyntax</a> , <a href="#">RamenChef</a> , <a href="#">Satyam S</a>
15	Node.js с CORS	<a href="#">Buzut</a>
16	Node.JS с ES6	<a href="#">Inanc Gumus</a> , <a href="#">xam</a> , <a href="#">ymz</a> , <a href="#">zurfyx</a>
17	Node.js с Oracle	<a href="#">oliolioli</a>
18	NodeJS Frameworks	<a href="#">dthree</a>
19	NodeJS с Redis	<a href="#">evalsocket</a>
20	nvm - Менеджер версий узлов	<a href="#">cyanbeam</a> , <a href="#">guleria</a> , <a href="#">John Vincent Jardin</a> , <a href="#">Luis González</a> , <a href="#">pranspach</a> , <a href="#">Shog9</a> , <a href="#">Tushar Gupta</a>
21	OAuth 2.0	<a href="#">tyehia</a>
22	package.json	<a href="#">Ankur Anand</a> , <a href="#">Asaf Manassen</a> , <a href="#">Chance Snow</a> , <a href="#">efeder</a> , <a href="#">Eric Smekens</a> , <a href="#">Florian Hämmerle</a> , <a href="#">Jaylem Chaudhari</a> , <a href="#">Kornel lauriys</a> , <a href="#">mezzode</a> , <a href="#">OzW</a> , <a href="#">RamenChef</a> , <a href="#">Robbie</a> , <a href="#">Shabin Hashim</a> , <a href="#">Simplans</a> , <a href="#">SteveLacy</a> , <a href="#">Sven 31415</a> , <a href="#">Tomás Cañibano</a> , <a href="#">user6939352</a> , <a href="#">V1P3R</a> , <a href="#">victorkohl</a>
23	passport.js	<a href="#">Red</a>
24	Readline	<a href="#">4444</a> , <a href="#">Craig Ayre</a> , <a href="#">Florian Hämmerle</a> , <a href="#">peteb</a>
25	Require ()	<a href="#">Philip Cornelius Glover</a>
26	Sequelize.js	<a href="#">Fikra</a> , <a href="#">Niroshan Ranapathi</a> , <a href="#">xam</a>
27	Автозагрузка изменений	<a href="#">ch4nd4n</a> , <a href="#">Dean Rather</a> , <a href="#">Jonas S</a> , <a href="#">Joshua Kleveter</a> , <a href="#">Nivesh</a> , <a href="#">Sanketh Katta</a> , <a href="#">zurfyx</a>
28	Архитектура и внутренние проекты Node.js	<a href="#">Ivan Hristov</a>
29	Асинхронное программирование	<a href="#">Ala Eddine JEBALI</a> , <a href="#">cyanbeam</a> , <a href="#">Florian Hämmerle</a> , <a href="#">H. Pauwelyn</a> , <a href="#">John</a> , <a href="#">Marek Skiba</a> , <a href="#">Native Coder</a> , <a href="#">omgimanagerd</a> , <a href="#">slowdeath007</a>
30	Асинхронный / Await	<a href="#">Cami Rodriguez</a> , <a href="#">Cody G.</a> , <a href="#">cyanbeam</a> , <a href="#">Dave</a> , <a href="#">David Xu</a> , <a href="#">Dom</a>

<a href="#">Vinyard</a> , <a href="#">m_callens</a> , <a href="#">Manuel</a> , <a href="#">nomanbinhusein</a> , <a href="#">Toni Villena</a>		
31	База данных ( MongoDB с Mongoose)	<a href="#">zurfyx</a>
32	Ввод / вывод файловой системы	<a href="#">4444</a> , <a href="#">Accepted Answer</a> , <a href="#">Aeolingamenfel</a> , <a href="#">Christophe Marois</a> , <a href="#">Craig Ayre</a> , <a href="#">DrakaSAN</a> , <a href="#">Duly Kinsky</a> , <a href="#">Florian Hämmerle</a> , <a href="#">gnerkus</a> , <a href="#">Harshal Bhamare</a> , <a href="#">hexacyanide</a> , <a href="#">jakerella</a> , <a href="#">Julien CROUZET</a> , <a href="#">Louis Barranqueiro</a> , <a href="#">midnightsyntax</a> , <a href="#">Mikhail</a> , <a href="#">peteb</a> , <a href="#">Shiven</a> , <a href="#">still_learning</a> , <a href="#">Tim Jones</a> , <a href="#">Tropic</a> , <a href="#">Vsevolod Goloviznin</a> , <a href="#">Zanon</a>
33	Веб-приложения с Express	<a href="#">Aikon Mogwai</a> , <a href="#">Alex Logan</a> , <a href="#">alexi2</a> , <a href="#">Andres C. Viesca</a> , <a href="#">Aph</a> , <a href="#">Asaf Manassen</a> , <a href="#">Batsu</a> , <a href="#">bekce</a> , <a href="#">brianmearns</a> , <a href="#">Community</a> , <a href="#">Craig Ayre</a> , <a href="#">Daniel Verem</a> , <a href="#">devnull69</a> , <a href="#">Everettss</a> , <a href="#">Florian Hämmerle</a> , <a href="#">H. Pauwelyn</a> , <a href="#">Inanc Gumus</a> , <a href="#">jemiloi</a> , <a href="#">Kid Binary</a> , <a href="#">kunerd</a> , <a href="#">Marek Skiba</a> , <a href="#">Mikhail</a> , <a href="#">Mohit Gangrade</a> , <a href="#">Mukesh Sharma</a> , <a href="#">Naeem Shaikh</a> , <a href="#">Niklas</a> , <a href="#">Nivesh</a> , <a href="#">noob</a> , <a href="#">Ojen</a> , <a href="#">Pasha Rumkin</a> , <a href="#">Paul</a> , <a href="#">Rafal Wiliński</a> , <a href="#">Shabin Hashim</a> , <a href="#">SteveLacy</a> , <a href="#">tandrewnichols</a> , <a href="#">Taylor Ackley</a> , <a href="#">themole</a> , <a href="#">tverdohleb</a> , <a href="#">Vsevolod Goloviznin</a> , <a href="#">xims</a> , <a href="#">Yerko Palma</a>
34	Взаимодействие с консолью	<a href="#">ScientiaEtVeritas</a>
35	Внедрение зависимости	<a href="#">Niroshan Ranapathi</a>
36	Всплывающие уведомления	<a href="#">Mario Rozic</a>
37	Выполнение файлов или команд с помощью дочерних процессов	<a href="#">guleria</a> , <a href="#">hexacyanide</a> , <a href="#">iSkore</a>
38	Доставить HTML или любой другой файл	<a href="#">Himani Agrawal</a> , <a href="#">RamenChef</a> , <a href="#">user2314737</a>
39	Запуск node.js как службы	<a href="#">Buzut</a>
40	Защита приложений Node.js	<a href="#">akinjide</a> , <a href="#">devnull69</a> , <a href="#">Florian Hämmerle</a> , <a href="#">John Slegers</a> , <a href="#">Mukesh Sharma</a> , <a href="#">Pauly Garcia</a> , <a href="#">Peter G</a> , <a href="#">pranspach</a> , <a href="#">RamenChef</a> , <a href="#">Simplans</a>
41	Избегайте обратного вызова ада	<a href="#">tyehia</a>

42	Излучатели событий	<a href="#">DrakaSAN</a> , <a href="#">Duly Kinsky</a> , <a href="#">Florian Hämmerle</a> , <a href="#">jamescostian</a> , <a href="#">MindlessRanger</a> , <a href="#">Mothman</a>
43	Изящное завершение	<a href="#">RamenChef</a> , <a href="#">Sathish</a>
44	Интеграция MongoDB для Node.js / Express.js	<a href="#">William Carron</a>
45	Интеграция MSSQL	<a href="#">damitj07</a>
46	Интеграция MySQL	<a href="#">Aminadav</a> , <a href="#">Andrés Encarnación</a> , <a href="#">Florian Hämmerle</a> , <a href="#">Ivan Schwarz</a> , <a href="#">jdrydn</a> , <a href="#">JohnnyCoder</a> , <a href="#">Kapil Vats</a> , <a href="#">KlwntSingh</a> , <a href="#">Marek Skiba</a> , <a href="#">Rafael Gadotti Bachovas</a> , <a href="#">RamenChef</a> , <a href="#">Simplans</a> , <a href="#">Sorangwala Abbasali</a> , <a href="#">surjikal</a>
47	Интеграция PostgreSQL	<a href="#">Niroshan Ranapathi</a>
48	Интеграция Кассандры	<a href="#">Vsevolod Goloviznin</a>
49	Интеграция паспорта	<a href="#">Ankit Rana</a> , <a href="#">Community</a> , <a href="#">Léo Martin</a> , <a href="#">M. A. Cordeiro</a> , <a href="#">Rupali Pemare</a> , <a href="#">shikhar bansal</a>
50	Использование Browserfiy для разрешения «требуемой» ошибки с помощью браузеров	<a href="#">Big Dude</a>
51	Использование IISNode для размещения веб-приложений Node.js в IIS	<a href="#">peteb</a>
52	Использование WebSocket с Node.JS	<a href="#">Rowan Harley</a>
53	Использование потоков	<a href="#">cyanbeam</a> , <a href="#">Duly Kinsky</a> , <a href="#">efeder</a> , <a href="#">johni</a> , <a href="#">KlwntSingh</a> , <a href="#">Max</a> , <a href="#">Ze Rubeus</a>
54	Использовать Случаи Node.js	<a href="#">vintproykt</a>
55	История Nodejs	<a href="#">Kelum Senanayake</a>

56	Как загружаются модули	<a href="#">RamenChef</a> , <a href="#">umesh</a>
57	Кластерный модуль	<a href="#">Benjamin</a> , <a href="#">Florian Hämmerle</a> , <a href="#">Kid Binary</a> , <a href="#">MayorMonty</a> , <a href="#">Mukesh Sharma</a> , <a href="#">riyadhalmur</a> , <a href="#">Vsevolod Goloviznin</a>
58	Код Node.js для STDIN и STDOUT без использования какой-либо библиотеки	<a href="#">Syam Pradeep</a>
59	Локализация узла JS	<a href="#">Osama Bari</a>
60	Маршрутизация аях-запросов с помощью Express.JS	<a href="#">RamenChef</a> , <a href="#">SynapseTech</a>
61	Маршрутизация NodeJs	<a href="#">parlad neupane</a>
62	Менеджер пакетов пряжи	<a href="#">Andrew Brooke</a> , <a href="#">skiilaa</a>
63	Многопоточность	<a href="#">arcs</a>
64	Монгодская интеграция	<a href="#">cyanbeam</a> , <a href="#">FabianCook</a> , <a href="#">midnightsyntax</a>
65	Монгузская библиотека	<a href="#">Alex Logan</a> , <a href="#">manuerumx</a> , <a href="#">Mikhail</a> , <a href="#">Naeem Shaikh</a> , <a href="#">Qiong Wu</a> , <a href="#">Simplans</a> , <a href="#">Will</a>
66	мотыга	<a href="#">signal</a>
67	Начало работы с профилированием узлов	<a href="#">damitj07</a>
68	НПМ	<a href="#">Abhishek Jain</a> , <a href="#">AJS</a> , <a href="#">Amreesh Tyagi</a> , <a href="#">Ankur Anand</a> , <a href="#">Asaf Manassen</a> , <a href="#">Ates Goral</a> , <a href="#">ccnokes</a> , <a href="#">CD..</a> , <a href="#">Cristian Cavalli</a> , <a href="#">David G.</a> , <a href="#">DrakaSAN</a> , <a href="#">Eric Fortin</a> , <a href="#">Everettss</a> , <a href="#">Explosion Pills</a> , <a href="#">Florian Hämmerle</a> , <a href="#">George Bailey</a> , <a href="#">hexacyanide</a> , <a href="#">HungryCoder</a> , <a href="#">Ionică Bizău</a> , <a href="#">James Taylor</a> , <a href="#">João Andrade</a> , <a href="#">John Slegers</a> , <a href="#">Jojodmo</a> , <a href="#">Josh</a> , <a href="#">Kid Binary</a> , <a href="#">Loufyloof</a> , <a href="#">m02ph3u5</a> , <a href="#">Matt</a> , <a href="#">Matthew Harwood</a> , <a href="#">Mehdi El Fadil</a> , <a href="#">Mikhail</a> , <a href="#">Mindsers</a> , <a href="#">Nick</a> , <a href="#">notgiorgi</a> , <a href="#">num8er</a> , <a href="#">oscarm</a> , <a href="#">Pete TNT</a> , <a href="#">Philipp Flenker</a> , <a href="#">Pieter Herroelen</a> , <a href="#">Pyloid</a> , <a href="#">QoP</a> , <a href="#">Quill</a> , <a href="#">Rafal Wiliński</a> , <a href="#">RamenChef</a> , <a href="#">Ratan Kumar</a> , <a href="#">RationalDev</a> , <a href="#">rdegges</a> ,

		<a href="#">refaelos</a> , <a href="#">Rizowski</a> , <a href="#">Shiven</a> , <a href="#">Skanda</a> , <a href="#">Sorangwala Abbasali</a> , <a href="#">still_learning</a> , <a href="#">subbu</a> , <a href="#">the12</a> , <a href="#">tlo</a> , <a href="#">Un3qual</a> , <a href="#">uzaif</a> , <a href="#">VladNeacsu</a> , <a href="#">Vsevolod Goloviznin</a> , <a href="#">Wasabi Fan</a> , <a href="#">Yerko Palma</a>
69	Обещания Bluebird	<a href="#">David Xu</a>
70	Обработка запроса POST в Node.js	<a href="#">Manas Jayanth</a>
71	Обработка исключений	<a href="#">KlwntSingh</a> , <a href="#">Nivesh</a> , <a href="#">riyadhainur</a> , <a href="#">sBanda</a> , <a href="#">sjmarshy</a> , <a href="#">topheman</a>
72	Обратный звонок для обещания	<a href="#">Clement JACOB</a> , <a href="#">Michael Buen</a> , <a href="#">Sanketh Katta</a>
73	Основы проектирования Node.js	<a href="#">Ankur Anand</a> , <a href="#">pietrovismara</a>
74	Отладка приложения Node.js	<a href="#">4444</a> , <a href="#">Alister Norris</a> , <a href="#">Ankur Anand</a> , <a href="#">H. Pauwelyn</a> , <a href="#">Matthew Shanley</a>
75	Отправить веб-уведомление	<a href="#">Housseem Yahiaoui</a>
76	Отправка потока файлов клиенту	<a href="#">Beshoy Hanna</a>
77	по металлу	<a href="#">RamenChef</a> , <a href="#">vsjn3290ckjnaoij2jikndckjb</a>
78	Подключиться к MongoDB	<a href="#">FabianCook</a> , <a href="#">Nainesh Raval</a> , <a href="#">Shriganesh Kolhe</a>
79	Постоянно поддерживать приложение узла	<a href="#">Alex Logan</a> , <a href="#">Bearington</a> , <a href="#">cyanbeam</a> , <a href="#">Himani Agrawal</a> , <a href="#">Mikhail</a> , <a href="#">mscdex</a> , <a href="#">optimus</a> , <a href="#">pietrovismara</a> , <a href="#">RamenChef</a> , <a href="#">Sameer Srivastava</a> , <a href="#">somebody</a> , <a href="#">Taylor Swanson</a>
80	Проблемы с производительностью	<a href="#">Antenka</a> , <a href="#">SteveLacy</a>
81	Проверка подлинности Windows в узле node.js	<a href="#">CJ Harries</a>
82	Производительность Node.js	<a href="#">Florian Hämmerle</a> , <a href="#">Inanc Gumus</a>
83	Простой API CRUD на	<a href="#">Iceman</a>

	основе REST	
84	Пул соединений Mysql	<a href="#">KlwntSingh</a>
85	Разбор аргументов командной строки	<a href="#">yrtimiD</a>
86	Развертывание приложений Node.js в производстве	<a href="#">Apidcloud</a> , <a href="#">Brett Jackson</a> , <a href="#">Community</a> , <a href="#">Cristian Boariu</a> , <a href="#">duncanhall</a> , <a href="#">Florian Hämmerle</a> , <a href="#">guleria</a> , <a href="#">haykam</a> , <a href="#">KlwntSingh</a> , <a href="#">Mad Scientist</a> , <a href="#">MatthieuLemoine</a> , <a href="#">Mukesh Sharma</a> , <a href="#">raghu</a> , <a href="#">sjmarshy</a> , <a href="#">tverdohleb</a> , <a href="#">tyehia</a>
87	Развертывание приложения Node.js без простоя.	<a href="#">gentlejo</a>
88	Рамки шаблонов	<a href="#">Aikon Mogwai</a>
89	Руководство для начинающих NodeJS	<a href="#">Niroshan Ranapathi</a>
90	Связь Socket.io	<a href="#">Forivin</a> , <a href="#">N.J.Dawson</a>
91	Связь клиент-сервер	<a href="#">Zoltán Schmidt</a>
92	Сервер узла без рамки	<a href="#">Hasan A Yousef</a> , <a href="#">Taylor Ackley</a>
93	Синхронное и асинхронное программирование в узлах	<a href="#">Craig Ayre</a> , <a href="#">Veger</a>
94	Создание API с помощью Node.js	<a href="#">Mukesh Sharma</a>
95	Создание библиотеки Node.js, которая поддерживает обе обещания и первые обратные вызовы	<a href="#">Dave</a>
96	Сокеты TCP	<a href="#">B Thuy</a>
97	Сообщение Arduino с nodeJs	<a href="#">sBanda</a>
98	Среда	<a href="#">Chris</a> , <a href="#">Freddie Coleman</a> , <a href="#">KlwntSingh</a> , <a href="#">Louis Barranqueiro</a> ,



Mikhail, sBanda		
99	Структура Route-Controller-Service для ExpressJS	<a href="#">nomanbinhusein</a>
100	Структура проекта	<a href="#">damitj07</a>
101	Структуры модульного тестирования	<a href="#">David Xu</a> , <a href="#">Florian Hämmerle</a> , <a href="#">skiilaa</a>
102	Удаление Node.js	<a href="#">John Vincent Jardin</a> , <a href="#">RamenChef</a> , <a href="#">snuggles08</a> , <a href="#">Trevor Clarke</a>
103	Удаленная отладка в Node.JS	<a href="#">Rick</a> , <a href="#">VooVoo</a>
104	Управление ошибками Node.js	<a href="#">Karlen</a>
105	Усовершенствованный дизайн API: лучшие практики	<a href="#">fresh5447</a> , <a href="#">nilakantha singh deo</a>
106	Установка Node.js	<a href="#">Alister Norris</a> , <a href="#">Aminadav</a> , <a href="#">Anh Cao</a> , <a href="#">asherbar</a> , <a href="#">Batsu</a> , <a href="#">Buzut</a> , <a href="#">Chance Snow</a> , <a href="#">Chezzwizz</a> , <a href="#">Dmitriy Borisov</a> , <a href="#">Florian Hämmerle</a> , <a href="#">GilZ</a> , <a href="#">guleria</a> , <a href="#">hexacyanide</a> , <a href="#">HungryCoder</a> , <a href="#">Inanc Gumus</a> , <a href="#">Jacek Labuda</a> , <a href="#">John Vincent Jardin</a> , <a href="#">Josh</a> , <a href="#">KahWee Teng</a> , <a href="#">Maciej Rostański</a> , <a href="#">mmhyamin</a> , <a href="#">Naing Lin Aung</a> , <a href="#">NuSkooler</a> , <a href="#">Shabin Hashim</a> , <a href="#">Siddharth Srivastva</a> , <a href="#">Sveratum</a> , <a href="#">tandrewnichols</a> , <a href="#">user2314737</a> , <a href="#">user6939352</a> , <a href="#">V1P3R</a> , <a href="#">victorkohl</a>
107	Файл загружен	<a href="#">Aikon Mogwai</a> , <a href="#">Iceman</a> , <a href="#">Mikhail</a> , <a href="#">walid</a>
108	Хороший стиль кодирования	<a href="#">Ajitej Kaushik</a> , <a href="#">RamenChef</a>
109	хрюкать	<a href="#">Naeem Shaikh</a> , <a href="#">Waterscroll</a>
110	Экспорт и импорт модуля в node.js	<a href="#">AndrewLeonardi</a> , <a href="#">Bharat</a> , <a href="#">commonSenseCode</a> , <a href="#">James Billingham</a> , <a href="#">Oliver</a> , <a href="#">sharif.io</a> , <a href="#">Shog9</a>
111	Экспорт и потребление модулей	<a href="#">Aminadav</a> , <a href="#">Craig Ayre</a> , <a href="#">cyanbeam</a> , <a href="#">devnull69</a> , <a href="#">DrakaSAN</a> , <a href="#">Fenton</a> , <a href="#">Florian Hämmerle</a> , <a href="#">hexacyanide</a> , <a href="#">Jason</a> , <a href="#">jdrydn</a> , <a href="#">LoufyLouf</a> , <a href="#">Louis Barranqueiro</a> , <a href="#">m02ph3u5</a> , <a href="#">Marek Skiba</a> , <a href="#">MrWhiteNerdy</a> , <a href="#">MSB</a> , <a href="#">Pedro Otero</a> , <a href="#">Shabin Hashim</a> , <a href="#">tkone</a> , <a href="#">uzaif</a>