ОНЛАЙН-ОБРАЗОВАНИЕ

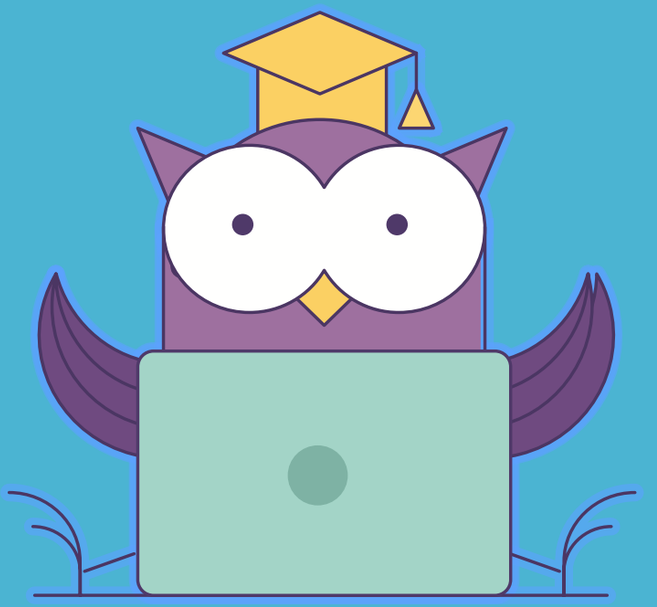
OTUS

Modern JavaScript Frameworks

Streams, Errors

Александр Коржиков

Как меня слышно и видно?



> Напишите в чат

+ если все хорошо

– если есть проблемы со звуком или с видео

Темы предыдущего занятия

- Events
- Event Loop
- Timers

Цели

- Использовать классы, объекты и функции стандартной библиотеки модуля **Streams**
- Работать с ошибками при написании серверного **JavaScript** кода

- Streams
- Errors

- Streams API
- Backpressuring in Streams
- Why I don't use Node's core 'stream' module - Rod Vagg

Абстрактный интерфейс для работы с потоками данных

```
const stream = require('stream')
```

- Что является потоком в **Node**?
- Для чего нужны потоки?

- Объекты **request**, **response** HTTP сервера

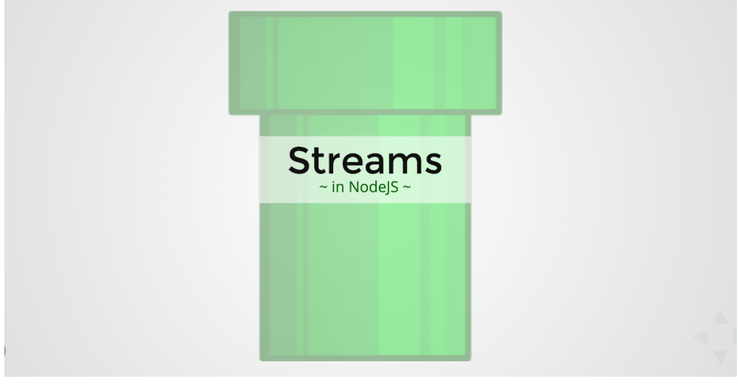
```
const server = http.createServer((req, res) => {
  res.statusCode = 200
  res.end()
})
```

- Стандартные потоки ввода-вывода **process.stdin**, **process.stdout**

```
process.stdin.setEncoding('utf8')
process.stdin.on('readable', () => {
  const chunk = process.stdin.read()
  // ...
})

process.stdout.write('end')
// console.log('end')
```

- Readable - для чтения
- Writable - для записи
- Duplex - комбинация
- Transform - модификация данных
- PassThrough - простейший Transform



Streams являются **EventEmitter**

- **on('data')**
- **on('readable')**
- **read()** для чтения

```
let body = []
request.on('data', (chunk) => {
  body.push(chunk)
})
.on('end', () => {
  body = Buffer.concat(body).toString()
})
```

Является **EventEmitter**

- **write()**, **end()** для Writable

```
response.write('<html>')
response.write('<body>')
response.write('<h1>Hello, World!</h1>')
response.write('</body>')
response.write('</html>')
response.end()
```

```
const http = require('http')
http.createServer((request, response) => {
  const { headers, method, url } = request
  let body = []

  request.on('data', (chunk) => {
    body.push(chunk)
  })
  .on('end', () => {
    body = Buffer.concat(body).toString()
    response.statusCode = 200
    response.setHeader('Content-Type', 'application/json')
    const responseBody = { headers, method, url, body }
    response.write(JSON.stringify(responseBody))
    response.end()
  })
})
.listen(8080)
```

- Что можно добавить?

Задача

Использовать стандартные потоки **process.stdin**, **process.stdout** для ввода имени из командной строки

```
node index.js

enter your name
Alex
your name is Alex
```

```
http.createServer((request, response) => {
  request.on('data', () => {
    /* ... */
  })
  .on('end', () => {
    response.write(/* ... */)
    response.end()
  })
})
```

- **Buffer, String** стандартно
- **Object** опционально с **objectMode**

```
const stream = require('stream')
const readable = (function() {
  const data = []
  const $ = new stream.Readable({ objectMode: true, read() {} })
  $.push({ a: 1 })
  return $
})()

readable.on('data', (data) => {
  console.log(data)
})
```

- **new Writable()** - конструктор
- **class MyWritable extends Writable** - класс

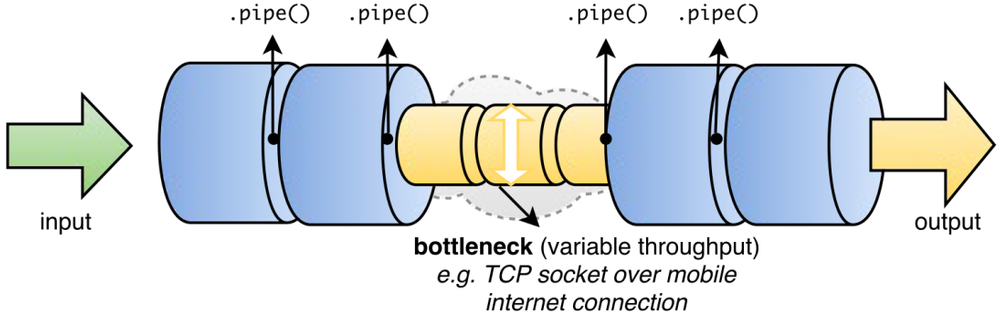
```
const stream = require('stream')
const writable = (function(){
  const data = []
  const $ = new stream.Writable({
    write(chunk, encoding, callback) {
      data.push(chunk.toString())
      callback()
    }
  })
  return $
})()

writable.write('some data')
writable.end('done writing data')
writable.on('finish', () => {
  console.log('All writes are now complete.')
})
```

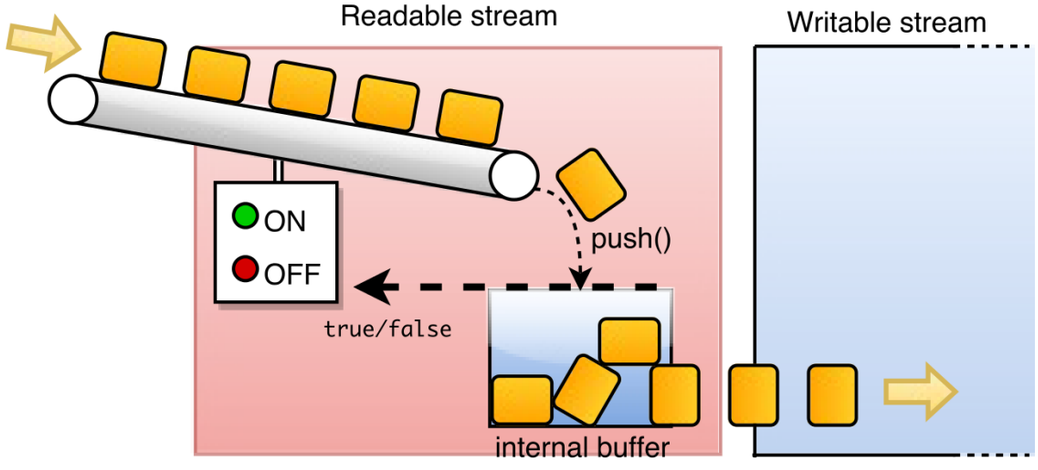
Readable.pipe() связывает поток, передавая все данные в **Writable**

```
readable.on('data', (chunk) => {
  writable.write(chunk)
})

readable.on('end', () => {
  writable.end()
})
// ->
readable.pipe(writable)
```



- **highWaterMark** - размер буфера
- **writable.write() - true / false**
- **drain** - событие продолжения



```
// node/stream/read.js

const { Readable } = require('stream')
const WORD = '1234567890'

var read = new Readable();

while(true) {
  new Array(100)
    .fill('')
    .map(() => read.push(WORD))
    .forEach((err) => console.log(err))
}
```

Использовать стандартные потоки и **pipe** для вывода вводимых данных

```
node index.js

abc
abc
123
123
```

Streams Q&A

- Стандартные типы ошибок JavaScript
- Системные ошибки при использовании API
- Assertion Errors
- User Errors

Вопрос - Стандартные ошибки

OTUS

Что к чему относится?

1. EvalError
2. SyntaxError
3. RangeError
4. ReferenceError
5. TypeError
6. URIError

```
/*a*/ var a = undefinedVariable
/*b*/ throw new EvalError('error')
/*c*/ decodeURIComponent('%')
/*d*/ eval('hoo bar')
/*e*/ undefined.not()
/*f*/ [].length = 'Wat?'
```

Какие способы обработки ошибок вы знаете?

```
require('fs')
 .readdir('not exist', () => {
   throw new Error('test')
  })
```

- try / catch / finally
- callback(err, res)
- on('error')
- Promise.reject()

```
try {
  require('not exist')
}
catch(e) {
  debugger
}
finally {
  console.log('go on')
}
```

Что здесь не так?

```
try {
  require('fs')
    .readdir('not exist', () => {
      throw new Error('test')
    })
} catch(e) {
  console.log('error')
}
```

Конструктор

- new Error(message)

Свойства

- error.message
- error.code - строка константа **E_ERROR_TITLE**
- error.stack с Error.captureStackTrace(error)

EventEmitter

- 'error' - всегда определять обработчик

Process

- 'uncaughtException' - возможность последнего слова
- 'unhandledRejection' - необработанные Promise

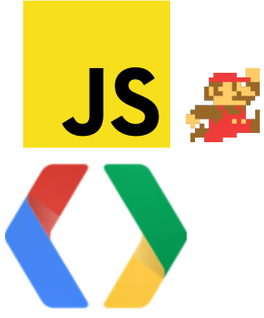
С помощью **process.on('uncaughtException')** перехватить и залогировать собственный тип ошибки (без использования конструктора **Error**, но включая стэк)

```
node error.js
Error
./errors/error.js:2:9)
    at emitOne (events.js:116:13)
    at process.emit (events.js:211:7)
    at process._fatalException (bootstrap_node.js:374:26)
```

- DO use Promises / async / await with catch()
- DO use Error class
- DO NOT throw 'strings or something'
- DO use central error handling and logging
- DO NOT continue with unknown 'uncaughtException'
- DO 'unhandledRejection' handling

Errors Q&A

- Работали с классами, объектами и функциями стандартной библиотеки модуля **Streams**
- Разобрали примеры обработки ошибок при написании серверного **JavaScript** кода


JS


node
JS
npm
express


TS

Написать приложение для работы с потоками:

- **Readable**, генерирующий случайные числа,
- **Transformable**, добавляющий случайное число к первому и
- **Writable**, выводящий в консоль данные

Данные должны "течь" readable > transformable > writable

Используйте **highWaterMark** для примера ограничения внутреннего буффера

Спасибо за внимание!

Пожалуйста, пройдите опрос

в личном кабинете