ОНЛАЙН-ОБРАЗОВАНИЕ

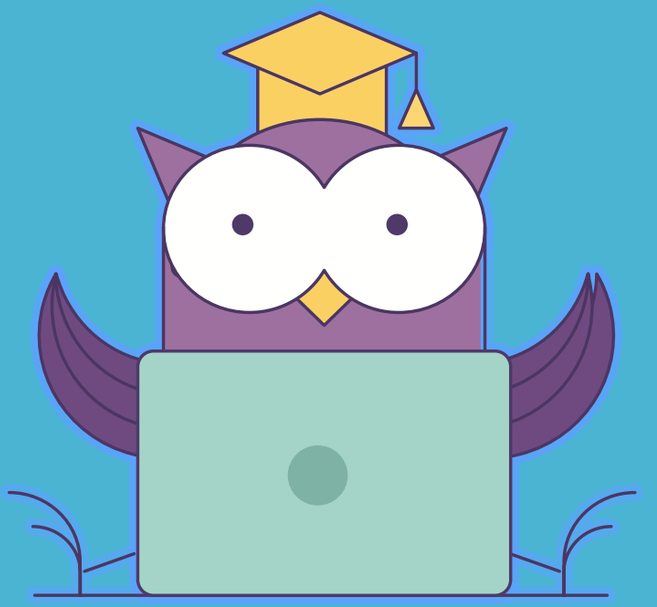
OTUS

Modern JavaScript Frameworks

Events, Timers, Event Loop

Александр Коржиков

Как меня слышно и видно?



> Напишите в чат

+ если все хорошо

– если есть проблемы со звуком или с видео

- Ма планируем **WebAssembly** на **13.05 - Node in Production**
- Ма планируем **offline** встречу на **16.05 или 17.05**

Темы предыдущего занятия

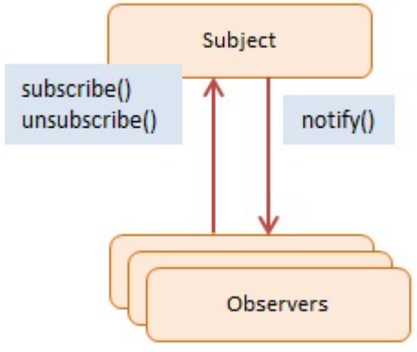
- Modules
  - Pattern
  - Classic
  - AMD
  - CommonJS
  - ES Modules
  - Native ES Modules - Gil Tayar

- Понимать **Event Loop** и особенности работы **Timers** в окружении **Node**

- Events
- Event Loop
- Timers

- The Node.js Event Loop, Timers, and process.nextTick()
- Jake Archibald - Tasks, microtasks, queues and schedules
- Event-driven programming
- The Node.js Event Loop: Not So Single Threaded

Observer определяет объект (**subject**), который может сообщать своим подписчикам (**subscribers, listeners**) об изменении своего состояния



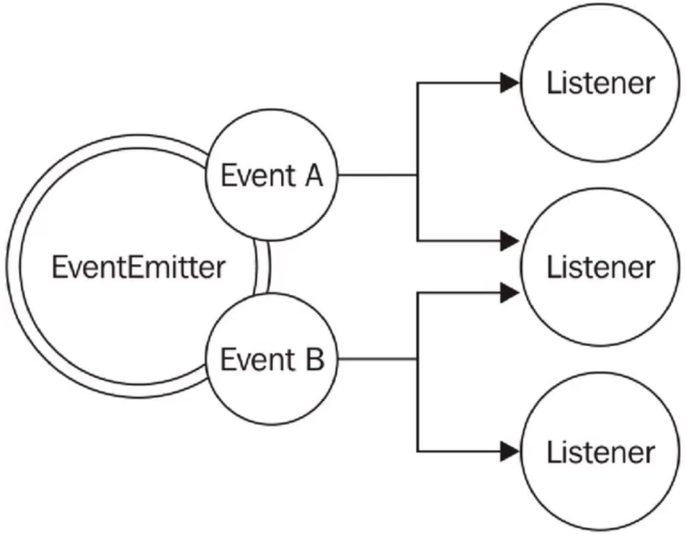
- **Observer** - подписка на события и уведомления
- Синхронное исполнение хэндлеров

```
const EventEmitter = require('events')
class MyEmitter extends EventEmitter {}
const myEmitter = new MyEmitter()
myEmitter.on('event', () => {
  console.log('an event occurred!')
})
myEmitter.emit('event')
console.log('after')
```

Специальные события

- **error** - произошла ошибка
- **newListener** - добавился новый подписчик
- **removeListener** - подписчик удалился

```
const EventEmitter = require('events')
class MyEmitter extends EventEmitter {}
const myEmitter = new MyEmitter()
myEmitter.emit('error')
```



- **on(), once(), prependListener()** - подписаться на события
- **emit()** - триггерить событие
- **removeListener()** - удалить подписку
- **listeners()** - вывести список хэндлеров

```
const EventEmitter = require('events')
const myEmitter = new EventEmitter()
myEmitter.on('event', (a, b) => {
  console.log(a, b, this)
  // prints a b {}
})

myEmitter.emit('event', 'a', 'b')
```

Что будет выведено в консоль?

```
const EventEmitter = require('events')

class WithLog extends EventEmitter {
  execute(taskFunc) {
    console.log('Before executing')
    this.emit('begin')
    taskFunc()
    this.emit('end')
    console.log('After executing')
  }
}

const withLog = new WithLog()

withLog.on('begin', () => console.log('About to execute'))
withLog.on('end', () => console.log('Done with execute'))

withLog.execute(() => console.log('*** Executing task ***'))
```

```
const EventEmitter = require('events')

class MyThing extends EventEmitter {
  constructor() {
    super()
    this.emit('thing1')
  }
}

const mt = new MyThing()
mt.on('thing1', function onThing1() {
  console.log('thing1')
})
```

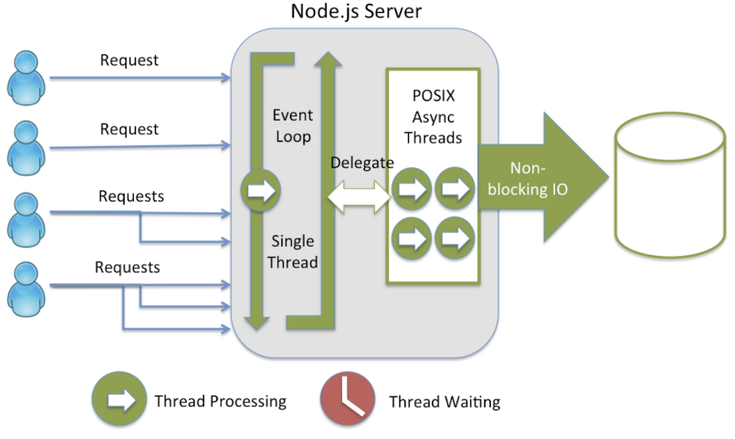
- Что не так с этим кодом?
- Что будет выведено в консоль?
- Как можно исправить?

Events Q&A

Событийно ориентированная парадигма - приложение подписывается на события и исполняют соответствующие функции обработчики

Вопрос

JavaScript однопоточный?



Что будет выведено в консоль?

```
console.log('script start')

setTimeout(function () {
  console.log('setTimeout')
}, 0)

Promise.resolve()
  .then(function () {
    console.log('promise1')
  })
  .then(function () {
    console.log('promise2')
  })

console.log('script end')
```

При старте **Node** инициализируется **Event Loop**, исполняется переданный скрипт вместе с синхронными вызовами (**API, setTimeout, process.nextTick**), после чего происходит работа **Event Loop**

```
const http = require('http')
const hostname = '127.0.0.1'
const port = 3000
const server = http.createServer((req, res) => {
  res.statusCode = 200
  res.setHeader('Content-Type', 'text/plain')
  res.end('Hello World\n')
})

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`)
})
```

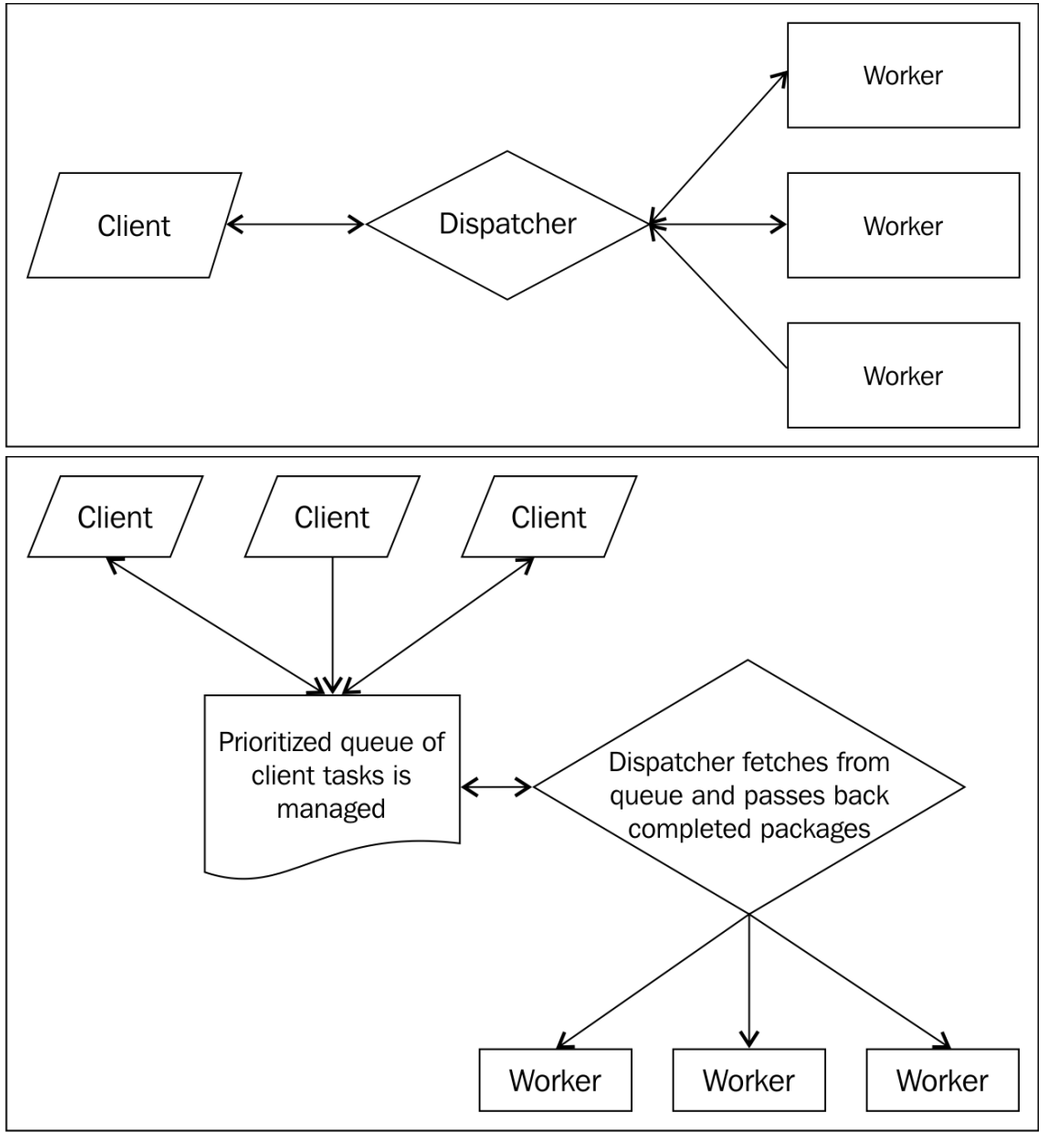
- В чем преимущество?

- Файл доступен для чтения
- Время ожидания таймера закончилось
- Что еще?

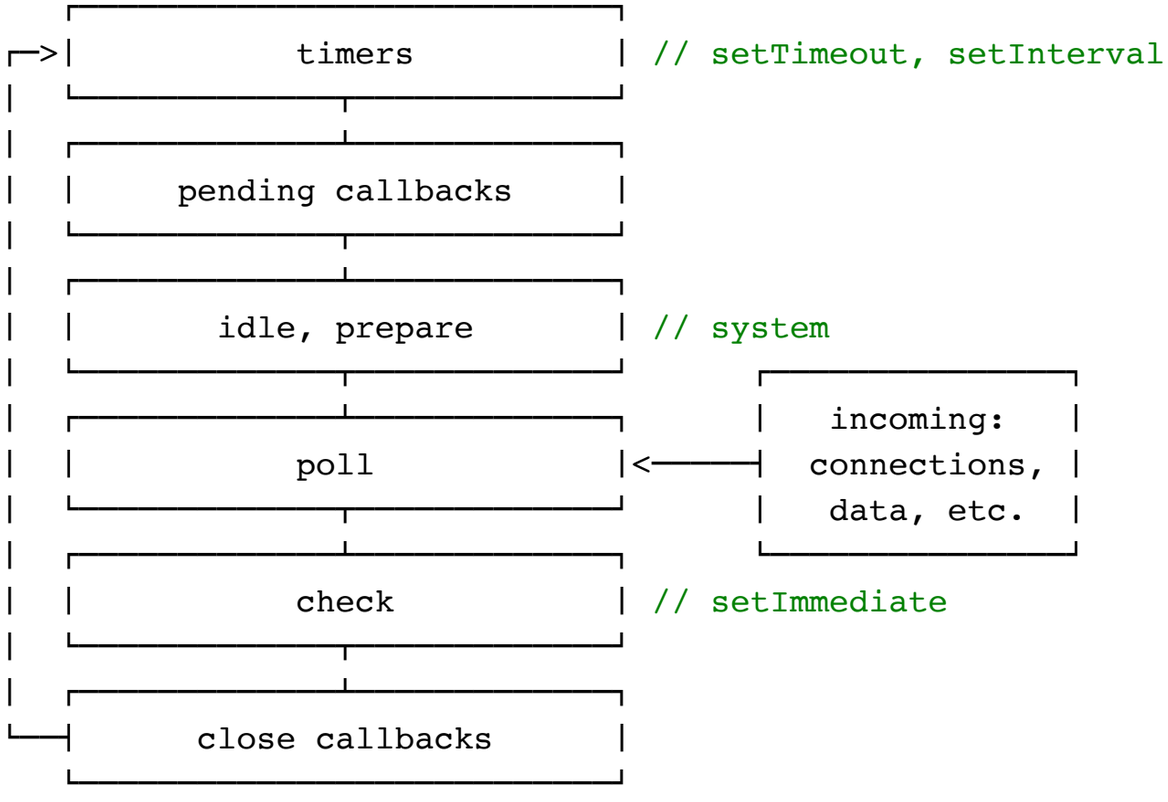
```
while there are still events to process:
  e = get the next event if there is a callback associated with e:
  call the callback

while (queue.waitForMessage()) {
  queue.processNextMessage()
}
```

© libuv

```
let stop = false
setTimeout(() => {
  stop = true
}, 1000)
while (stop === false) {}
```



- **Reactor** - ожидание доступа к ресурсу без блокировки программы
- *most worker threads spend their time waiting—for more instructions, a sub-task to complete*


timers
// setTimeout, setInterval
pending callbacks
idle, prepare
// system
poll
incoming:
connections,
data, etc.
check
// setImmediate
close callbacks

The algorithm

1. Is still something to do? Is Event Loop alive?
2. Update time - now()
3. Run timers if they are due
4. Run pending callbacks - callbacks from previous operations that have completed, f.e. to write on a **TCP** socket
5. Run pending handlers
6. Prepare handlers - always running before **I/O**
7. Block for **I/O** and poll from it
8. Check like prepare but after **I/O**
9. Close callbacks for objects disposal

- **setTimeout(), setInterval()** - как обычно
- **setImmediate()** - специальный таймер, работает в фазе **poll**

```
const timeoutObj = setTimeout(() => {
  console.log('timeout')
}, 1500)

const immediateObj = setImmediate(() => {
  console.log('immediate')
})

const intervalObj = setInterval(() => {
  console.log('interval')
}, 500)

clearImmediate(immediateObj)
```

Cancel timers

- **clearTimeout(), clearImmediate(), clearInterval()**
- **ref(), unref()** - установка / отмена таймеров по ссылке

```
const timerObj = setTimeout(() => {
  console.log('will i run?')
})

clearTimeout(timerObj)

timerObj.unref()

setImmediate(() => {
  timerObj.ref()
})
```

"Наиболее быстрое" исполнение асинхронных операций

в конце фаз Event Loop

```
console.log('start')

process.nextTick(() => {
  console.log('nextTick')
})

console.log('scheduled')

// start
// scheduled
// nextTick
```

Как это продемонстрировать?

- Node.js
- Alexander Lobashev, RaiffeisenBank, 2018, t.me/alobashev
- РИТ++ 2018

Что будет выведено в консоль?

```
setTimeout(() => {
  console.log('timeout')
}, 0)
setImmediate(() => {
  console.log('immediate')
})
```

А здесь?

```
const fs = require('fs')
fs.readFile(__filename, () => {
  setTimeout(() => { console.log('timeout') }, 0)
  setImmediate(() => { console.log('immediate') })
})
```

Что будет выведено в консоль?

```
const fs = require('fs')
setTimeout(() => console.log('timeout'))
setImmediate(() => console.log('immediate'))
fs.readFile('./events.js', () => console.log('fs'))
```

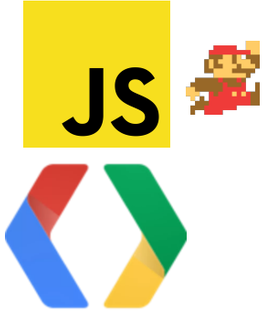
А здесь?

```
const fs = require('fs')
fs.readFile('./events.js', () => {
  console.log('fs')
  setTimeout(() => console.log('timeout'))
  setImmediate(() => console.log('immediate'))
})
```

Что будет выведено в консоль?

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

setTimeout(() => console.log('timeout out'))
setImmediate(() => console.log('immediate out'))

fs.readFile('./events.js', (err, data) => {
  console.log('fs')
  process.nextTick(() => console.log('next in'))
  setTimeout(() => console.log('timeout in'))
  setImmediate(() => console.log('immediate in'))
})

const next = () => {
  console.log('next')
  process.nextTick(next)
}

process.nextTick(next)
```

- Познакомились с понятием Event Loop
- Разобрали особенности работы Events, Timers и nextTick


JS


node
JS
npm
express


TS

Напишите скрипт **tree** для вывода списка файлов и папок файловой системы.

```
foo/
├── bar/
│   ├── bar1.txt
│   ├── bar2.txt
│   └── baz/
├── f1.txt
└── f2.txt
```

```
{
  "files": [
    "foo/f1.txt",
    "foo/f2.txt",
    "foo/bar/bar1.txt",
    "foo/bar/bar2.txt"
  ],
  "dirs": [
    "foo",
    "foo/bar",
    "foo/bar/baz"
  ]
}
```

Спасибо за внимание!

Пожалуйста, пройдите опрос

в личном кабинете

- Все ли темы были понятны? (да - нет)
- Легкий материал или нет? (1 просто - 10 сложно)