

JS Level 3

Node.js



Node.js

Основной темой нашей сегодняшней лекции будет разговор о серверных сетевых приложениях и протоколах.



TCP/IP



Network

Мы изучили всю теорию JS, которая нам необходима, для того, чтобы создавать сетевые приложения.

Теперь нам нужно изучить теорию самих сетей, чтобы понимать, что мы создаём и для чего.



Network

Вы помните вот эту картинку:



На ней клиент и сервер соединены друг с другом посредством сети интернет и могут обмениваться данными.

Поскольку и клиент, и сервер могут быть написаны на разных языках и устроены по-разному, им нужно договориться о том, как они будут взаимодействовать.



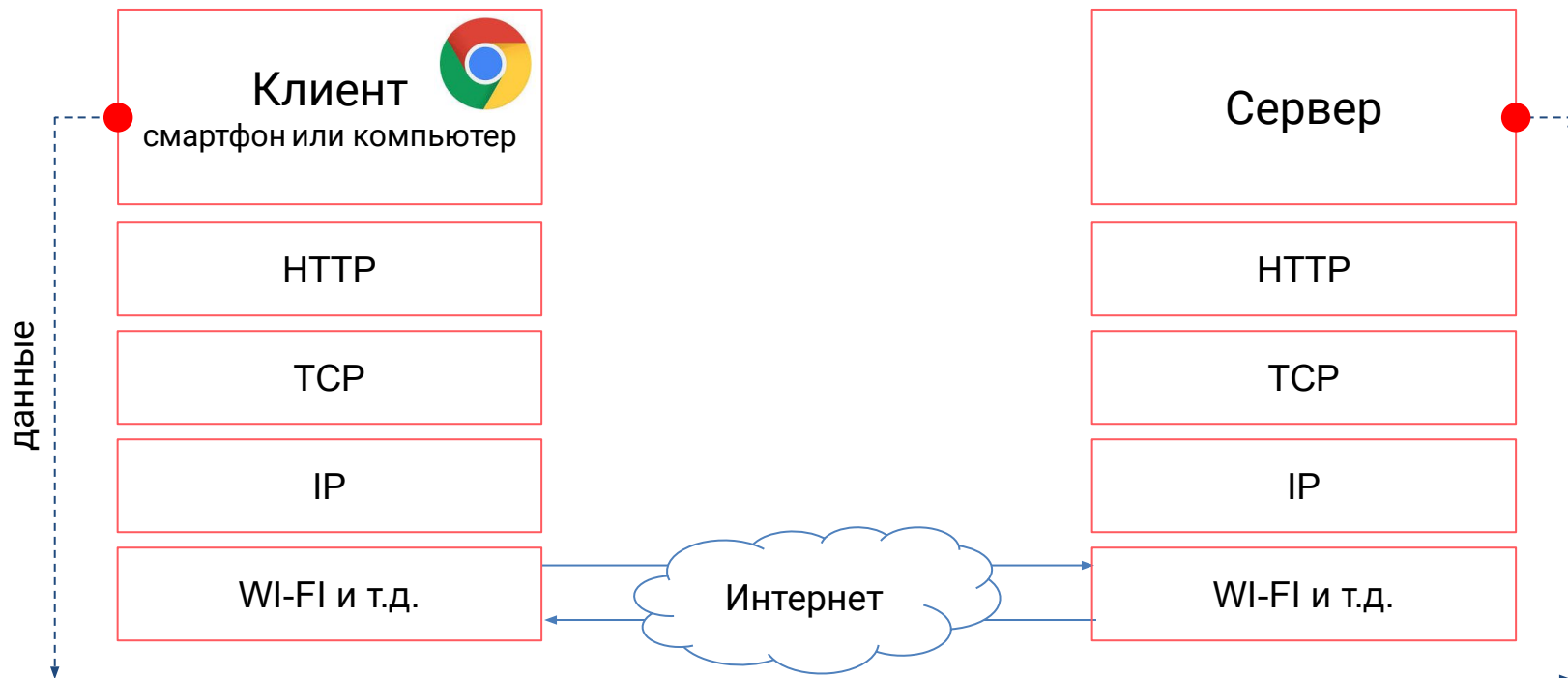
Protocol

Протокол – это набор правил, регламентирующих что-то. В нашем случае, протоколы будут регламентировать общение двух приложений (клиентского и серверного) по сети между собой.



TCP/IP

Самым распространённым стеком (набором) протоколов для сетевого взаимодействия является TCP/IP. Называется он стеком (структура данных, работающая по принципу первый пришёл – последний ушёл) потому, что они выстроены друг над другом так, что каждый конкретный уровень использует другой для работы:



TCP/IP

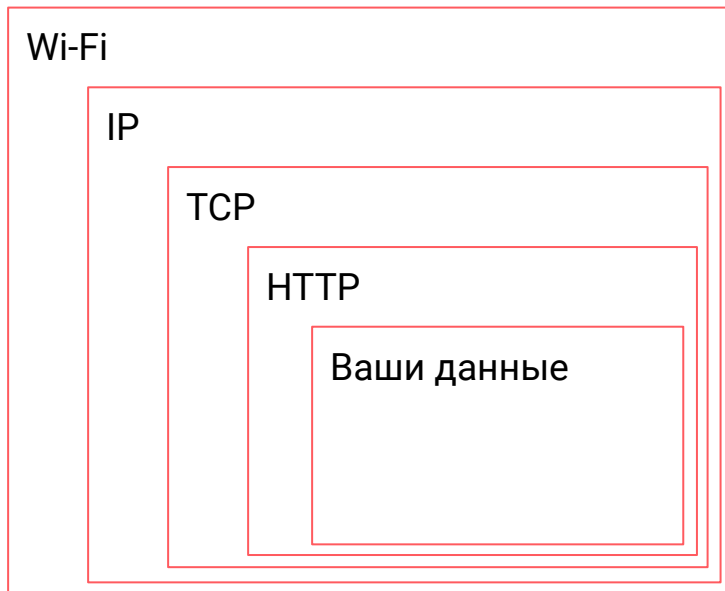
Как себе это представлять: если вы когда-либо отправляли бумажное письмо, то здесь устроено всё так же.

Вы (приложение) пишете письмо на листе бумаги. Дальше вступает в игру протокол (HTTP), который требует, чтобы вы упаковали ваш лист бумаги в конверт нужного формата и наклеили марки. После чего служба почты собирает все письма в мешки (протокол TCP) и отправляет. Отправлять она может поездом, самолётом или даже автомобильным транспортом. Но для отправки ей нужен адрес назначения – протокол IP.



TCP/IP

С точки зрения получателя всё происходит наоборот: поезд привозит мешки с письмами, письма вынимаются из мешка, почтальон относит письмо адресату (или уведомление), а адресат вскрывает письмо и уже видит лист бумаги с вашим текстом:



Network Access

Самый нижний уровень (Network Access) отвечает за то, как информация физически передаётся (в виде электрических сигналов или электромагнитного поля) по кабелям (например, Ethernet) или воздуху (например, Wi-Fi). За это несут ответственность сетевые карты компьютера и Wi-Fi адаптеры.



TCP/IP

IP (Internet Protocol) отвечает за передачу данных между двумя компьютерами, подключенными к сети (будем называть их хостами), у которых есть адреса (их называют IP-адресами).

IP отвечает за то, чтобы попытаться доставить данные от отправителя до получателя (т.е. найти маршрут и передать сами данные). При этом IP не гарантирует, что данные будут доставлены.

TCP (Transmission Protocol) отвечает за доставку данных.



net

Давайте попробуем написать свой первый TCP-сервер. За работу на этом уровне отвечает [модуль net](#) стандартной библиотеки Node.js:

Class: `net.Server`

#

Added in: v0.1.90

- Extends: `<EventEmitter>`

This class is used to create a TCP or `IPC` server.

`new net.Server([options][, connectionListener])`

#

- `options` `<Object>` See `net.createServer([options][, connectionListener])`.
- `connectionListener` `<Function>` Automatically set as a listener for the `'connection'` event.
- Returns: `<net.Server>`

`net.Server` is an `EventEmitter` with the following events:

Что значит **class** и **extends**?



CLASSES



Функции

Чтобы разобраться с этим, нам придётся вернуться к функциям и повторить несколько ключевых вещей.

Итак, мы говорили, что функции – это специальный объект, который можно:

1. Вызывать (с помощью оператора `()`)
2. Передавать в другие функции (поскольку функция – это объект)
3. Хранить в свойствах объекта (тогда функцию называют методом)

Но есть ещё несколько, например, это вызов функции с оператором `new`. Давайте разбираться, для чего они нужны и как связаны друг с другом.



Сервер

Итак, мы хотим с вами создать сервер. Что такое сервер? Под этим термином обычно (в зависимости от контекста) понимают:

1. Компьютер, который находится (чаще всего) в сети Интернет
2. Приложение, которое работает на этом компьютере, занимаясь непрерывным обслуживанием запросов клиентов

Наша задача – написать приложение (компьютер мы явно "написать" не сможем).



Сервер

Пока мы умеем создавать только объекты, давайте так и сделаем:

```
JS main.js > ...  
1  function handler() {  
2    |   console.log('handle request');  
3  }  
4  
5  const server = {  
6    |   name: 'application server',  
7    |   handler: handler,  
8  };
```

Т.е. по нашей логике, сервер - это объект, у которого есть имя, и свойство, в котором лежит функция, выводящая в консоль строку 'handle request'.



Сервер

Теперь мы можем на этом объекте вызвать метод любым из способов, который вам нравится:

```
JS main.js > ...  
1  function handler() {  
2    |   console.log('handle request');  
3  }  
4  
5  const server = {  
6    |   name: 'application server',  
7    |   handler: handler,  
8  };  
9  
10 server.handler();  
11 server['handler']();
```



Операторы

Давайте немного поговорим об операторах, чтобы вы понимали, как и что работает.

В математике у нас есть операторы $+$ и $*$. Выражение: $11 + 5 * 2$ вычисляется следующим образом: сначала считается $5 * 2$ а затем $11 + 10$. Надеюсь, вы знаете почему? Потому что у оператора $*$ приоритет выше, чем у оператора $+$.

В JS у нас так же существуют операторы и они так же выстроены в определённом порядке исходя из их приоритета.



Описание	Оператор
member	. []
call / create instance	() ^{вызов} new
negation/increment	! ~ _{унарный} + _{унарный} ++ -- typeof void delete
multiply/divide	* / %
addition/subtraction	+ -
bitwise shift	<< >> >>>
relational	< <= > >= in instanceof
equality	== != === !==
bitwise-and	&
bitwise-xor	^
bitwise-or	
logical-and	&&
logical-or	
conditional	?:
assignment	= += -= *= /= %= <<= >>= >>>= &= ^= =
comma	,

приоритет

min

Примечание*: здесь приведён неполный список



Операторы

Давайте разберём вот это выражение: `server.handler();`

В нём есть целых два оператора: `.` Точка и `()` Вызов.

Смотрим по таблице:

Описание	Оператор
member	<code>.</code> <code>[]</code>
call / create instance	<code>()</code> <small>Вызов</small> <code>new</code>

У оператора `.` Точка приоритет выше, значит мы сначала вычисляем `server.handler` (а это будет функция `handler`), а затем уже выполняем оператор `()` Вызов (вызываем эту функцию). Соответственно, так же работает со всеми остальными выражениями.



Операторы

Как и в математике, чтобы изменить приоритет операторов, вы можете поставить `()` (не вызов функции, а просто круглые скобки для группировки выражений).

Теперь давайте разбираться дальше.



Сервер

Мы создали один сервер. А что если мы хотим создать их несколько? Тогда нам придётся продублировать наш код:

```
JS main.js > ...  
1  function handler() {  
2    |    console.log('handle request');  
3  }  
4  
5  const appServer = {  
6    |    name: 'application server',  
7    |    handler: handler,  
8  };  
9  
10 const photoServer = {  
11   |    name: 'photo server',  
12   |    handle: handler,  
13 };
```

Заметили ли вы ошибку?



Сервер

Мы совершенно "случайно" опечатались и назвали во втором случае поле **handle** вместо **handler**. А это значит, что при вызове мы получим вот такую ошибку:

```
10  const photoServer = {  
11    |    name: 'photo server',  
12    |    handle: handler,  
13  };  
14  
15  photoServer.handler();
```

```
c:\projects\server\main.js:15  
photoServer.handler();  
      ^
```

```
TypeError: photoServer.handler is not a function  
    at Object.<anonymous> (c:\projects\server\main.js:15:13)  
    at Module._compile (internal/modules/cjs/loader.js:778:30)  
    ... несколько строк пропущено ...  
    at bootstrapNodeJSCore (internal/bootstrap/node.js:622:3)  
Waiting for the debugger to disconnect...  
Process exited with code 1
```



Сервер

Давайте попробуем вывести с помощью `typeof` и `console.log` информацию о том, что мы получили:

```
15 console.log(photoServer.handler);  
16 console.log(typeof photoServer.handler);
```

В данном случае никакой ошибки нет: при попытке обращения к несуществующему свойству мы получаем `undefined`. А `undefined` – это не функция, поэтому использовать оператор вызова нельзя.



Сервер

Как бы так сделать, чтобы мы не ошибались так при создании объектов? По факту, нам нужно выполнить набор повторяющихся действий... А что, если само создание объекта поместить в функцию?

```
JS main.js > ...  
1  function handler() {  
2    |   console.log('handle request');  
3  }  
4  
5  function createServer(name, handler) {  
6    |   const object = {};  
7    |   object.name = name;  
8    |   object.handler = handler;  
9    |   return object;  
10 }  
11  
12 const appServer = createServer('application server', handler);  
13 const photoServer = createServer('photo server', handler);
```

Никакой магии - внутри функции создаётся объект и нам возвращается.



Дублирование кода

Если внимательно посмотреть на этот код, то можно увидеть, что какие бы подобные конструкции не создавали, всегда будет повторяться две строки:

```
+ 5  function createServer(name, handler) {  
6      const object = {};  
7      object.name = name;  
8      object.handler = handler;  
+ 9      return object;  
10 }
```

Например, создадим функцию, которая будет отвечать за создание постов:

```
+ 12 function createPost(id, name, content) {  
13     const object = {};  
14     object.id = id;  
15     object.name = name;  
16     object.content = content;  
+ 17     return object;  
18 }
```



Функция-конструктор

В JS сделали следующую вещь: сказали, что если мы вызываем функцию вместе с оператором `new`, то:

1. Нам в подарок внутри функции дадут имя `this`, в котором будет пустой объект (то, что мы раньше писали `const object = {}`)
2. За нас автоматически выполнят `return this` (то, что мы раньше писали `return object`)

JS main.js > ...

```
1 function handler() {  
2   console.log('handle request');  
3 }  
4  
5 function Server(name, handler) {  
6   this.name = name;  
7   this.handler = handler;  
8 }  
9
```

Такие функции называют функции-конструкторы и по соглашению присваивают им имена с большой буквы

```
10 const appServer = new Server('application server', handler);  
11 const photoServer = new Server('photo server', handler);  
12  
13 appServer.handler();  
14 photoServer.handler();
```



Функция-конструктор

Интересно, а что будет, если мы запустим нашу функцию-конструктор без **new**, давайте проверим в отладчике:



The screenshot shows a JavaScript debugger interface. On the left, the 'VARIABLES' panel is expanded, showing the 'Local: Server' scope. It displays the following information:

- `handler`: `f handler() {\r\n ...`
- `name`: `'application server'`
- `this`: `global`

Below this, the 'Global' scope is also visible. On the right, the 'main.js' file is open, showing the following code:

```
1 function handler() {  
2     console.log('handle request');  
3 }  
4  
5 function Server(name, handler) {  
6     this.name = name;  
7     this.handler = handler;  
8 }  
9  
10 Server('application server', handler);
```

The line `this.name = name;` on line 6 is highlighted in yellow, indicating it is the current line of execution. A yellow arrow points to the start of line 6, indicating the current position in the code.

Т.е. в данном случае **this** – это **global**. И при таком "случайном" вызове мы в глобальный объект запишем свойства.



global

```
JS main.js > ...  
1  function handler() {  
2    |    console.log('handle request');  
3  }  
4  
5  function Server(name, handler) {  
6    |    this.name = name;  
7    |    this.handler = handler;  
8  }  
9  
10 Server('application server', handler);  
11  
12 console.log(global.name); // application server
```



global

На самом деле, ситуация ещё хуже: мы можем писать имена без `const/let`, и создавать таким образом свойства в `global`:

```
14  property = 'it works';  
15  console.log(global.property); // it works
```

Как-то это не совсем хорошо и почему это до сих пор не исправили?



global

Вы как разработчики должны понимать следующую вещь: если вашим продуктом пользуются миллиарды людей, нельзя просто так взять и всё переделать.

JS используется не только в Node.js, но ещё и в браузерах. Поэтому если это просто "выключить", то мало кто себе представляет последствия: сколько сайтов перестанут работать. Обратите внимание, не у всех сайтов есть программисты, которые поддерживают их целыми днями.



use strict

В JS придумали другой выход – специальную инструкцию `'use strict'`, которая переводит JS в строгий режим. `'use strict'` можно писать либо в самом верху js-файла, либо внутри функции (тогда в строгом режиме работает только тело функции). Данный строгий режим вместо глобального `this` подставляет `undefined` при вызове без `new`:

JS main.js > ...

```
1  'use strict';
2
3  function handler() {
4      console.log('handle request');
5  }
6
7  function Server(name, handler) {
8      this.name = name;
9      this.handler = handler;
10 }
11
12 Server('application server', handler);
```

c:\projects\server\main.js:8

this.name = name;
^

TypeError: Cannot set property 'name' of undefined



use strict

То же самое и в другом случае:

```
JS main.js
1  'use strict';
2
3  property = 'it works';
4  console.log(global.property); // ReferenceError
```

```
c:\projects\server\main.js:3
property = 'it works';
      ^
```

```
ReferenceError: property is not defined
    at Object.<anonymous> (c:\projects\server\main.js:3:10)
```

Использование **'use strict'** обязательно, бот будет проверять наличие этой инструкции в каждом JS-файле.



Code Style

Вы можете спросить: но если сам язык позволяет так писать, почему бот и мы (проверяющие) требуем писать по-другому?

Всё дело в том, что не все возможности являются безопасными и хорошими практиками (вы достаточно часто можете слышать выражения Good Parts, Bad/Evil Parts). В JS всё ровно так же.

Поэтому большие компании, например Airbnb, собирают [целые руководства по тому, как нужно писать код на JS](#). И мы, в целом, им будем следовать.



Code Style

Принятый стиль кодирования в компании называют Code Style. Соответственно, если вы не следуете ему, то ваш код просто не принимают в качестве работы.

Бот поступает ровно таким же образом.



ESM

Если вы подключаете модули в режиме ESM, то они автоматически выполняются в строгом режиме.



Контекст вызова

Итак, давайте смотреть дальше. Мы с вами выяснили, что когда вызываешь функцию с помощью оператора `new`, то внутри функции `this` равен пустому объекту.

`this` – это свойство контекста вызова. Контекст вызова может быть:

- `global` (вне функции), тогда `this` указывает на глобальный объект
- `function` (внутри функции), тогда `this` может быть разным (сейчас это разберём)
- `eval` (связан с работой функции `eval`, которая позволяет строку выполнять как код, но является небезопасной, поэтому нами не рассматривается)

Важно: внутри стрелочных функций нет своего контекста вызова, она использует вышестоящий (т.е. тот, в котором была написана стрелочная функция).



Контекст вызова

Мы выяснили, что когда мы вызываем функцию с **new**, то **this** указывает на новый объект. Давайте рассмотрим, какие ещё есть сценарии.

```
JS main.js > ...
1  'use strict';
2
3  function handler() {
4    console.log(this);
5  }
6
7  function Server(name, handler) {
8    this.name = name;
9    this.handler = handler;
10 }
11
12 const appServer = new Server('application server', handler);
13 appServer.handler();
```

→ Server { name: 'application server', handler: [Function: handler] }



Контекст вызова

Что это значит, это значит, что если вы вызываете какую-то функцию в виде `object.function()`, то внутри функции `this` будет указывать на тот объект, что стоял до точки:

```
13 appServer.handler();
```

Т.е. внутри `handler this` будет указывать на `appServer`. Но здесь тоже не всё так просто – это будет работать только если вы вызываете именно в таком виде.

Примечание*: в JS есть возможность создать функцию с уже "привязанным" контекстом, который не будет зависеть от того, как мы вызываем функцию.



Контекст вызова

Если вы вызовете просто `handler();` – то `this` будет `undefined`. Но есть и похитрее трюк:

```
JS main.js > ...
1  'use strict';
2
3  function handler() {
4    |   console.log(this);
5  }
6
7  function Server(name, handler) {
8    |   this.name = name;
9    |   this.handler = handler;
10 }
11
12 const appServer = new Server('application server', handler);
13 const func = appServer.handler; // не вызываем, просто кладём в переменную
14 func(); // вызываем - this = undefined
```

Т.е. важно именно то, как вы вызываете функцию.



Advanced

Это продвинутая секция, вы её можете пропустить. Существуют дополнительные три функции, которые позволяют изменять контекст:

- `bind`
- `call`
- `apply`

`bind` позволяет вам создать новую функцию с "привязанным" `this`:

```
12  const appServer = new Server('application server', handler);
13  const func = handler.bind(appServer);
14  func(); // вызываем - this = appServer
```

`call` и `apply` позволяют вызвать функцию, передав туда объект, который будет выступать в роли `this`:

```
12  const appServer = new Server('application server', handler);
13  handler.call(appServer); // this = appServer
```



Advanced

Кроме того, в стандартной библиотеке, вы постоянно будете видеть аргумент под названием **thisArg**, который позволяет указать другой **this**:

Syntax

```
arr.forEach(callback(currentValue [, index [, array]]), thisArg)
```

Parameters

callback

Function to execute on each element. It accepts between one and three arguments:

currentValue

The current element being processed in the array.

index | Optional

The index **currentValue** in the array.

array | Optional

The array **forEach()** was called upon.

thisArg | Optional

Value to use as **this** when executing **callback**.



Важно

С **this** нужно обязательно разобраться, поскольку его очень любят спрашивать на собеседованиях.



Прототипы

С темой функций-конструкторов тесно связана тема прототипов. Мы уже сталкивались с ними в документации:

```
Array.prototype.forEach()
```

```
Array.prototype.includes()
```

```
Array.prototype.indexOf()
```

```
Array.prototype.join()
```

Во-первых, давайте разберёмся, почему **Array** написано с большой буквы. Скорее всего, вы уже догадались, что это функция-конструктор.

Поэтому массивы можно создавать как через синтаксис **const items = [1, 2, 3];** так и через **const items = new Array(1, 2, 3);**



Прототипы

Во-вторых, посмотрим на то, что такое **prototype**:

```
✓ Server: f Server(name, handler) {\r\n ...  
  > get arguments: f ()  
  > set arguments: f ()  
  > get caller: f ()  
  > set caller: f ()  
  length: 2  
  name: 'Server'  
✓ prototype: {constructor: f}  
  > constructor: f Server(name, handler) ...  
  > __proto__: Object  
    [[FunctionLocation]]: @ c:\projects\se...  
  > [[Scopes]]: Scopes[1]  
  > __proto__: function () { [native code]...
```

```
7 function Server(name, handler) {  
8     this.name = name;  
9     this.handler = handler;  
10 }  
11  
12 console.log(Server.prototype);  
13
```

Т.е. у функции **Server** есть поле **prototype**, в котором хранится объект с полем **constructor** и **__proto__**.



Прототипы

В новых версиях Node.js вместо `__proto__` будет написано `[[Prototype]]`:

```
> prototype: {constructor: f}
  [[FunctionLocation]]:
> [[Prototype]]: f ()
> [[Scopes]]: Scopes[1]
> get __proto__: f __proto__()
> set __proto__: f __proto__()
> this: Object
```



Прототипы

Напоминаем, что функции – это тоже объекты, а значит в них можно класть свойства.



Прототипы

В JS объекты выстраиваются в цепочки с помощью свойства `__proto__` (два нижних подчёркивания говорят о том, что это служебное свойство и пользоваться им не стоит):

```
✓ appServer: Server {name: 'application s...  
  > handler: f handler() {\r\n    console...  
    name: 'application server'  
  }  
  > __proto__: Object  
    > constructor: f Server(name, handler) ...  
    > __proto__: Object
```

```
12 console.log(Server.prototype);  
13  
14 const appServer = new Server('application server', handler);  
15 console.log(appServer.__proto__);
```



Прототипы

А теперь ещё раз внимательно сравним:


```

✓ Server: f Server(name, handler) {\r\n ...
  > get arguments: f ()
  > set arguments: f ()
  > get caller: f ()
  > set caller: f ()
  length: 2
  name: 'Server'
  ✓ prototype: {constructor: f}
    > constructor: f Server(name, handler) ...
    > __proto__: Object
      [[FunctionLocation]]: @ c:\projects\se...
    > [[Scopes]]: Scopes[1]
    > __proto__: function () { [native code]...
  
```



```

✓ appServer: Server {name: 'application s...
  > handler: f handler() {\r\n    console...
    name: 'application server'
  ✓ __proto__: Object
    > constructor: f Server(name, handler) ...
    > __proto__: Object
  
```



Не кажется ли вам, что объекты, хранящиеся в выделенных свойствах очень похожи?



Прототипы

Если их сравнить, то получится **true** – это значит, что на самом деле, один и тот же объект хранится в этих двух свойствах:

```
JS main.js > ...
1  'use strict';
2
3  function handler() {
4    |   console.log(this);
5  }
6
7  function Server(name, handler) {
8    |   this.name = name;
9    |   this.handler = handler;
10 }
11
12 const appServer = new Server('application server', handler);
13
14 console.log(Server.prototype === appServer.__proto__); // true
```



===

Оператор `===` в данном случае проверяет, что оба свойства указывают на один и тот же объект.

Что это значит? Это значит, что при создании объекта через функцию-конструктор, в его свойство `__proto__` кладётся тот объект, который лежал в функции-конструкторе в свойстве `prototype`.

Но что это даёт?



Server

Возвращаясь к нашему серверу: получается мы можем вынести в прототип все свойства, которые должны быть одинаковыми для всех объектов, созданных с помощью этой функции-конструктора. Мы можем поместить туда функцию **shutdown** (выключения сервера - пусть все сервера выключаются одинаково):

```
JS main.js > ...
1  'use strict';
2
3  function handler() {
4    |   console.log(this);
5  }
6
7  function shutdown() {
8    |   console.log('shutdown');
9  }
10
11 function Server(name, handler) {
12   |   this.name = name;
13   |   this.handler = handler;
14 }
15
16 Server.prototype.shutdown = shutdown;
17
18 const appServer = new Server('application server', handler);
19 appServer.shutdown();
```



Server

Теперь внимательно посмотрим: и `shutdown`, и `handler` мы использовали всего по одному разу. Стоило ли их объявлять отдельно?

В JS у нас есть два понятия:

1. `function declaration` – это объявление функции в том виде, в котором мы объявляли раньше
2. `function expression` – объявление функции в виде выражения (которое выполняется и возвращает в результате объект-функцию).



Function Declaration

Когда вы объявляете подобным образом функцию, в конце объявления точку с запятой не ставят и функция "всплывает" на самый верх файла. Что значит всплывает?



Function Declaration

JS main.js > ...

```
1  'use strict';
2
3  Server.prototype.shutdown = shutdown;
4
5  const appServer = new Server('application server', handler);
6  appServer.shutdown();
7
8  function handler() {
9    |   console.log(this);
10 }
11
12 function shutdown() {
13   |   console.log('shutdown');
14 }
15
16 function Server(name, handler) {
17   |   this.name = name;
18   |   this.handler = handler;
19 }
```

Это будет работать,
потому что JS соберёт
все функции и поместит
их в верх файла (т.е. всё
будет так, как было до
этого).



Function Expression

Объявление функции в виде выражения выглядит иначе, при этом ничего никуда не всплывает, поэтом порядок важен:

```
JS main.js > ...
1  'use strict';
2
3  const handler = function() {
4    |   console.log(this);
5  };
6
7  const shutdown = function () {
8    |   console.log('shutdown');
9  };
10
11 const Server = function(name, handler) {
12   |   this.name = name;
13   |   this.handler = handler;
14 };
15 Server.prototype.shutdown = shutdown;
16
17 const appServer = new Server('application server', handler);
18 appServer.shutdown();
```

Т.е. мы просто сохранили наши функции в виде переменных, а потом использовали эти переменные. Но если мы используем переменные (вместо которых подставляются значения), можем ли мы использовать сами значения?



Anonymous Function

Функции, у которых нет имени, называют анонимными функциями (на предыдущем слайде тоже были анонимные функции):

```
JS main.js > ...
1  'use strict';
2
3  const Server = function(name, handler) {
4    |   this.name = name;
5    |   this.handler = handler;
6  };
7  Server.prototype.shutdown = function() {
8    |   console.log('shutdown');
9  };
10
11 const appServer = new Server('application server', function() {
12 |   console.log(this);
13 });
14 appServer.shutdown();
```

Обратите внимание, мы используем функциональные выражения и в качестве свойств, и в качестве аргументов.



Class

Получилось неплохо, но всё-таки вот эта часть (выделенная) с функцией-конструктором и prototype не воспринимается как единое целое:

```
JS main.js > ...
1  'use strict';
2
3  const Server = function(name, handler) {
4      this.name = name;
5      this.handler = handler;
6  };
7  Server.prototype.shutdown = function() {
8      console.log('shutdown');
9  };
10
11 const appServer = new Server('application server', function() {
12     console.log(this);
13 });
14 appServer.shutdown();
```

Хотелось бы, чтобы они как-то визуально и логически были связаны.



Class

Для этой цели в современных версиях языка придумали ключевое слово **class**, которое является "синтаксическим сахаром" (удобной записью) предыдущей конструкции + содержит ряд ограничений:

```
JS main.js > ...
1  'use strict';
2
3  class Server {
4      constructor(name, handler) { // аналог функции-конструктора
5          this.name = name;
6          this.handler = handler;
7      }
8
9      shutdown() { // аналог prototype.shutdown
10         console.log('shutdown');
11     }
12 }
13
14 const appServer = new Server('application server', function () {
15     console.log(this);
16 });
17 appServer.shutdown();
```



Class

Что же за ограничения есть? Например, функцию-конструктор мы можем вызывать как с **new**, так и без **new**, написав нечто вроде:

```
JS main.js > ...
1  'use strict';
2
3  function Server(name, handler) {
4      if (this === undefined) {
5          // called without new
6          return
7      }
8
9      // called with new
10 }
```

И вызывая вот так: **Server('application server', function() { ... });** - без **new**.

Кроме того, вся функция "всплывёт" в начало файла.



Class

В случае же классов:

1. Без `new` вызывать нельзя
2. Никуда не всплывают (считайте, что объявлены как `const`)
3. Весь код внутри класса в `strict mode`
4. И ряд других ограничений



NET



net

Теперь, когда мы разобрались с классами, можем двигаться дальше:

Class: `net.Server`

Added in: v0.1.90

- Extends: `<EventEmitter>`

This class is used to create a TCP or `IPC` server.

`new net.Server([options][, connectionListener])`

- `options` `<Object>` See `net.createServer([options][, connectionListener])`.
- `connectionListener` `<Function>` Automatically set as a listener for the `'connection'` event.
- Returns: `<net.Server>`

`net.Server` is an `EventEmitter` with the following events:



net

extends означает, что где-то в цепочке прототипов будет объект того типа, который указан в **extends**. В данном случае – **EventEmitter**. А это значит, что все его методы будут нам доступны.

В JS мы могли бы записать это следующим образом:

```
class Server extends EventEmitter { ... }
```

добавляет **EventEmitter** в цепочку прототипов



net

Помимо класса, у нас есть вспомогательная функция, которая, фактически, создаёт объект нужного класса:

```
net.createServer([options][, connectionListener]) #
```

► History

- `options` `<Object>`
 - `allowHalfOpen` `<boolean>` If set to `false`, then the socket will automatically end the writable side when the readable side ends. **Default:** `false`.
 - `highWaterMark` `<number>` Optionally overrides all `net.Socket`'s `readableHighWaterMark` and `writableHighWaterMark`. **Default:** See `stream.getDefaultHighWaterMark()`.
 - `pauseOnConnect` `<boolean>` Indicates whether the socket should be paused on incoming connections. **Default:** `false`.
 - `noDelay` `<boolean>` If set to `true`, it disables the use of Nagle's algorithm immediately after a new incoming connection is received. **Default:** `false`.
 - `keepAlive` `<boolean>` If set to `true`, it enables keep-alive functionality on the socket immediately after a new incoming connection is received, similarly on what is done in `socket.setKeepAlive([enable][, initialDelay])`. **Default:** `false`.
 - `keepAliveInitialDelay` `<number>` If set to a positive number, it sets the initial delay before the first keepalive probe is sent on an idle socket. **Default:** `0`.
- `connectionListener` `<Function>` Automatically set as a listener for the `'connection'` event.
- Returns: `<net.Server>`



net

Давайте разбираться:

JS main.js X

JS main.js > ...

```
1  'use strict';
2
3  const net = require('node:net'); ← будет вызвана, когда кто-то подключится
4
5  const server = new net.Server(function(socket) {
6    |   console.log('somebody connected');
7  });
8
9  server.listen(9999); ← запуск на определённом порту
```



net

В рамках протокола TCP устанавливается соединение между двумя участниками: клиентом и сервером. Соединение представляет из себя пару сокетов.

Сокет – это такая абстракция (представьте себе трубу, в которую участники могут записывать данные, а также читать из неё). После установления соединения, упрощённо можем считать, что клиент и сервер могут обмениваться потоком байт, пока один из них не закроет соединение.



net

Аналогия: когда вы кому-то звоните, то телефоны устанавливают соединение друг с другом. Но для того, чтобы позвонить, вам нужно две вещи:

1. Знать, кому вы звоните – номер телефона (это в нашей схеме IP адрес)
2. Чтобы человек, которому вы звоните, взял трубку

При этом сформируется пара сокетов. Т.е. вы будете говорить в микрофон, который на вашем устройстве, а слышать вас ваш собеседник будет в своём динамике (хотя между вашим микрофоном и его динамиком могут быть сотни километров). Соответственно, сокет есть на вашем устройстве (это микрофон + динамик), и на устройстве вашего собеседника (его микрофон + его динамик).

Когда соединение установлено, вы разговариваете непрерывным потоком (и не думаете ни о каких номерах телефона, гудках и т.д.) так же, как если бы общались вживую.



net

JS main.js X

JS main.js > ...

```
1  'use strict';
2
3  const net = require('node:net');
4
5  const server = new net.Server(function(socket) {
6    |   console.log('somebody connected');
7  });
8
9  server.listen(9999);
```

Так вот то, что мы написали, отвечает за обработку звонка (строки 5-7). А строка 9 отвечает за ожидание и приём звонка. Т.е. сервер должен сидеть и ждать, пока кто-то позвонит, а затем передавать "звонок" в функцию 5-7.



net

Q: что это за 9999?

A: в отличие от телефона, у которого чаще всего один владелец, на компьютере работает много приложений. Поэтому помимо IP-адреса (который относится ко всему компьютеру) приложениям выделяются специальные номера, которые называются портами. Вот 9999 это номер порта.



net

Но хватит обсуждать, давайте запустим наше приложение и посмотрим, что произойдёт. Первое, что мы увидим, это то, что приложение не завершается, а продолжает работать – всё верно, потому что это сервер, он должен постоянно работать, ожидая подключения клиентов.

Вы можете остановить его, нажав на значок стоп:



EADDRINUSE

Если вдруг, при запуске вы видите вот такую ошибку:

```
events.js:287
    throw er; // Unhandled 'error' event
    ^
Error: listen EADDRINUSE: address already in use :::9999
```

Это значит, что вы либо второй раз пытаетесь запустить сервер (а его нужно запускать только один раз), либо какая-то программа на вашем компьютере уже работает на по этому адресу (кроме того, сервер может блокировать антивирус, поэтому посмотрите в настройках антивируса).



EADDRINUSE

Чтобы узнать, какая программа уже работает по этому адресу, выполните следующие действия (в Windows):

1. Вбейте в терминале VS Code команду **resmon**
2. Перейдите на вкладку Сеть и найдите ID процесса, который занимает порт **9999**:

Монитор ресурсов

Файл Монитор Справка

Обзор ЦП Память Диск Сеть

Процессы с сетевой активностью

Сетевая активность 1 кбит/с - сетевой ввод-вывод Использование сети: 0%

TCP-подключения

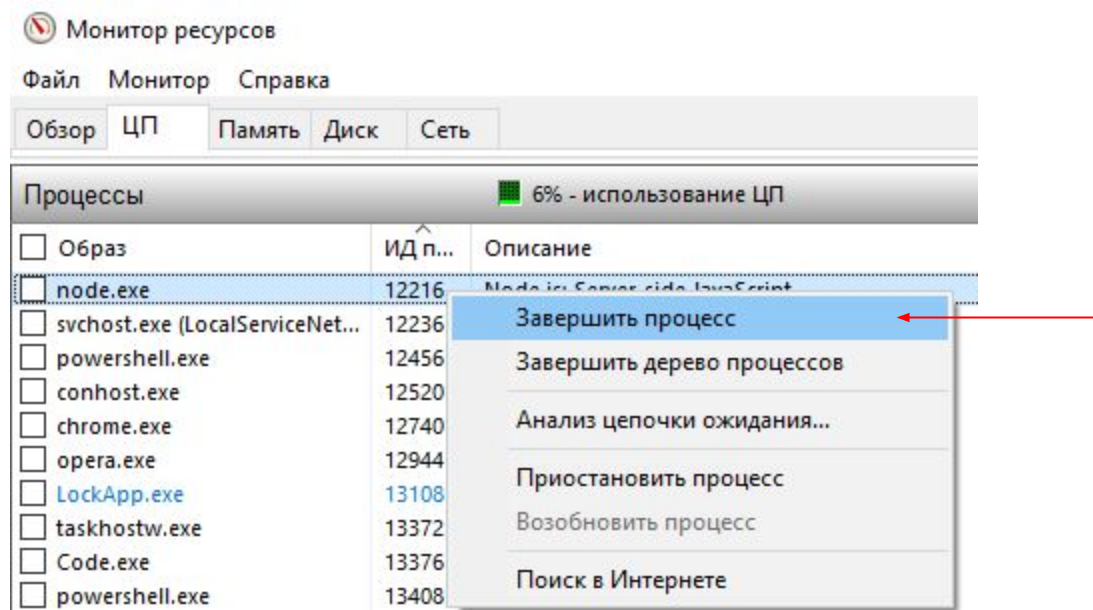
Прослушиваемые порты

Образ	ИД процесса	Адрес	Порт	Протокол	Состояние бр...
goland64.exe	19744	Петлевой адрес в IPv4	6943	TCP	Разрешен, не ...
node.exe	12216	IPv6 не задан	9999	TCP	Не разрешен, ...



EADDRINUSE

3. Перейдите на вкладку ЦП, найдите этот процесс по идентификатору, щёлкните правой кнопкой мыши нажмите **Завершить процесс**:



EADDRINUSE

В Mac OS выполните команду:

```
lsof -nP -iTCP:9999 | grep LISTEN
```

Вы получите идентификатор процесса примерно вот в таком виде:

```
node 62441 ...
```

После чего этот процесс можно убить командой:

```
sudo kill -9 62441
```



EADDRINUSE

В Linux выполните команду:

```
fuser -n tcp 9999
```

Вы получите идентификатор процесса примерно вот в таком виде:

```
9999/tcp: 62441
```

После чего этот процесс можно убить командой:

```
sudo kill -9 62441
```

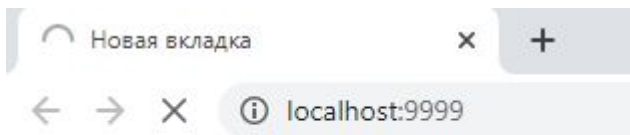


localhost

Сервер наш запустился, осталось понять, как к нему получить доступ.

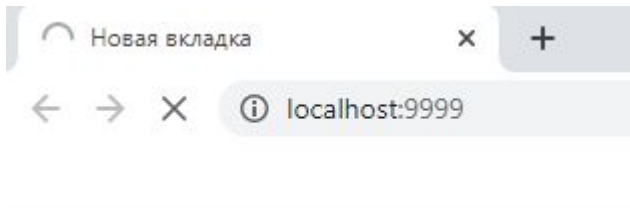
У нашего компьютера может быть много IP-адресов, но один есть точно – 127.0.0.1 или localhost. Это специальный адрес, который всегда есть у вашего компьютера, даже если он не подключен к сети. Он позволяет сетевым приложениям работать на вашем компьютере, и программировать их, даже если вы сейчас не подключены к сети Интернет (или какой-либо другой сети).

Открываем браузер и вбиваем в адресную строку: **localhost:9999**



localhost

Всё будет выглядеть так, как будто браузер что-то грузит (хотя на самом деле ничего он не грузит):



Но если мы откроем консоль сервера, то увидим сообщение somebody connected:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
C:\Program Files\nodejs\node.exe c:\projects\server\main.js
Debugger listening on ws://127.0.0.1:52326/a7311762-48e5-4a22-9690-0009570879a0
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
somebody connected
```

Это значит, что браузер действительно соединился с нашим сервером и Node.js вызвал наш callback на подключение.



localhost

Как же нам прочитать те данные, что передаёт нам клиент (браузер) и ответить ему? Давайте разбираться: нам в callback при подключении клиента приходит объект класса **Socket** (когда мы говорим объект класса, это значит этот объект был создан с помощью функции-конструктора или класса):

```
const server = new net.Server(function(socket) {  
  |   console.log('somebody connected');  
});
```

Этот [класс описан в документации](#) и у него есть куча свойств, среди которых есть [write](#):

socket.write(data[, encoding][, callback])

Added in: v0.1.90

- **data** <string> | <Buffer> | <Uint8Array>
- **encoding** <string> Only used when data is **string**. Default: **utf8**.
- **callback** <Function>
- Returns: <boolean>



localhost

Давайте попробуем. Браузер умеет обрабатывать HTML, вот его мы ему и подсунем: создадим файл `index.html`, с помощью сокращения `! + Tab` сформируем базовую разметку и отдадим браузеру:

```
<> index.html > ...
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <title>Document</title>
7  </head>
8  <body>
9      <h1>Hello from Node.js!</h1> ← это мы дописали сами
10 </body>
11 </html>
```



localhost

```
JS main.js > ...
1  'use strict';
2
3  const net = require('net');
4  const fs = require('fs');
5
6  const html = fs.readFileSync('index.html', 'utf8');
7
8  const server = new net.Server(function(socket) {
9    |   console.log('somebody connected');
10   |   socket.write(html);
11   | });
12
13  server.listen(9999);
```

Важно: не забудьте перезапустить сервер и обновить страницу в браузере.



localhost

И всё вроде должно работать, но мы получим следующую страницу:



Страница недоступна

Сайт **localhost** отправил недействительный ответ.

ERR_INVALID_HTTP_RESPONSE

Перезагрузить

Это значит, что браузеру не понравилось, что мы ему ответили.

Но почему? Мы же отдали ему HTML-страницу.



HTTP



HTTP

Всё дело в том, что браузер работает по протоколу HTTP. А протокол, как вы помните – это набор правил. Соответственно, если мы работаем не по правилам, то нас не понимают. Об этом нам и сообщает браузер, говоря "invalid response".

Сам этот протокол имеет несколько версий, нас будет интересовать пока только версия 1.1: <https://tools.ietf.org/html/rfc2616>.

HTTP работает поверх TCP, т.е. фактически, использует TCP как транспорт для доставки сообщений.



HTTP

Давайте попробуем сами реализовать протокол HTTP поверх TCP. Что значит реализовать протокол? Это значит передавать байты по правилам, которые требует данный протокол.

Сейчас наш с вами `net.Server` – это TCP-сервер, который умеет читать и писать байты. Вот на базе него мы всё и сделаем.



Messages

В протоколе говорится о том, что клиент и сервер взаимодействуют в формате передачи сообщений:

4.1 Message Types

HTTP messages consist of requests from client to server and responses from server to client.

HTTP-message = Request | Response ; HTTP/1.1 messages

Сообщения бывают двух типов (символ | – означает или):

1. Запрос (**Request**) от клиента
2. Ответ (**Response**) от сервера.



Messages

5 Request

A request message from a client to a server includes, within the first line of that message, the method to be applied to the resource, the identifier of the resource, and the protocol version in use.

```
Request      = Request-Line           ; Section 5.1
               *(( general-header      ; Section 4.5
                 | request-header      ; Section 5.3
                 | entity-header ) CRLF) ; Section 7.1
               CRLF
               [ message-body ]       ; Section 4.3
```

уже знакомые вам CRLF →

6 Response

After receiving and interpreting a request message, a server responds with an HTTP response message.

```
Response     = Status-Line           ; Section 6.1
               *(( general-header      ; Section 4.5
                 | response-header     ; Section 6.2
                 | entity-header ) CRLF) ; Section 7.1
               CRLF
               [ message-body ]       ; Section 7.2
```

уже знакомые вам CRLF →



HTTP 1.1

HTTP 1.1 представляет собой текстовый протокол - это значит, что байты, передаваемые на уровне этого протокола, можно интерпретировать как текст.

Пока нас не интересует сам запрос (мы будем разбираться с ним на следующей лекции). Нас интересует ответ.

Ответ состоит из трёх частей:

- Status Line (как завершился запрос – успешно или нет)
- Headers (мета-данные – набор заголовков, разделённых CRLF)
- Тело ответа (непосредственно, сам контент, если есть)



HTTP 1.1

Q: зачем нужны три части, если браузер всегда показывает только HTML-страничку?

A: они нужны для передачи служебных данных (например, информации о кодировке страницы, последнем обновлении и т.д.).

Status Line должна выглядеть следующим образом:

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

Где:

- HTTP-Version – HTTP/1.1
- SP – пробел
- Status-Code – числовой код, например, **200** – если всё хорошо, **404** – страница не найдена
- Reason-Phrase – фраза, например, **OK** или **Not Found**



HTTP 1.1

Статус-коды делятся на 5 больших групп:

- 100-199 – Info
- 200-299 – Success
- 300-399 – Redirection
- 400-499 – Client Error
- 500-599 – Server Error

Мы пока будем отдавать код **200** с фразой OK.



HTTP 1.1

Помимо **Status Line** нам нужно отдавать ещё заголовки, которые описывают тип контента, который мы отдаём, его размер (чтобы клиент понимал, сколько ему нужно читать) и т.д.



HTTP 1.1

В итоге мы получим примерно вот такое приложение:

JS main.js > ...

поскольку читаем файл всего один раз

до старта приложения, можем

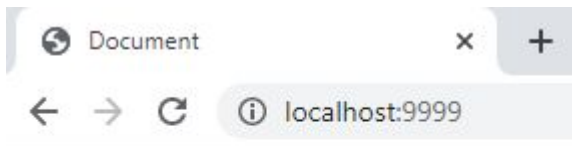
использовать синхронную версию

```
1  'use strict';
2
3  const net = require('node:net');
4  const fs = require('node:fs');
5
6  const html = fs.readFileSync('index.html', 'utf-8');
7
8  const server = new net.Server(function(socket) {
9    console.log('somebody connected');
10    // status line
11    socket.write('HTTP/1.1 200 OK\r\n');
12    // headers
13    socket.write('Content-Type: text/html; charset=utf-8\r\n');
14    socket.write('Content-Length: ' + html.length + '\r\n');
15    socket.write('Connection: close\r\n');
16    socket.write('\r\n');
17    // body
18    socket.write(html);
19  });
20
21  server.listen(9999);
```



HTTP 1.1

Перезапустим сервер и обновив страничку в браузере увидим:



Hello from Node.js!

Т.е. мы действительно реализовали протокол HTTP (это слишком громко сказано, мы сделали небольшую заглушку), который позволяет в браузер выдавать одну html-страницу.



http

Естественно, это очень не продуктивно – работать на уровне байт и вручную писать строки заголовков, **CRLF** и т.д.

Поскольку одна из основных сфер применения Node.js – это веб (а именно http-сервера), то в нём давно уже реализован модуль [http](#), который позволяет нам делать всё гораздо проще.

Давайте смотреть: класс **http.Server** наследуется от **net.Server**:

Class: http.Server

#

Added in: v0.1.17

- Extends: `<net.Server>`

Поэтому, в принципе, мы можем писать всё то же самое. Но это несколько не улучшит ситуацию.



EventEmitter

Пришла пора немного поговорить про **EventEmitter**: это специальный класс, который предоставляет метод [on](#) (не только его, конечно):

```
emitter.on(eventName, listener)
```

Added in: v0.1.101

- `eventName` `<string> | <symbol>` The name of the event.
- `listener` `<Function>` The callback function
- Returns: `<EventEmitter>`

Этот метод предоставляет возможность подписаться на события, происходящие в жизни объекта. Например, у нашего сервера **net.Server** таким событием было подключение клиента:

```
Event: 'connection'
```

Added in: v0.1.90

- `<net.Socket>` The connection object

Emitted when a new connection is made. `socket` is an instance of `net.Socket`.



EventEmitter

И тот callback, который мы клали в функцию-конструктор, на самом деле, записывался именно в это событие.

```
new net.Server([options][, connectionListener])
```

- `options` `<Object>` See `net.createServer([options][, connectionListener])`.
- `connectionListener` `<Function>` Automatically set as a listener for the `'connection'` event. ←
- Returns: `<net.Server>`

`net.Server` is an `EventEmitter` with the following events:



EventEmitter

Здесь важно понять:

- в браузере сам браузер определяет, какие события и на каких элементах доступны (мы можем вещать listener'ы на эти события), но также мы можем создавать и собственные события и отправлять их с помощью метода [dispatchEvent](#) – т.е. ключевым интерфейсом является **EventTarget**
- в Node.js у нас нет **EventTarget**'а, но есть [EventEmitter](#) – нам достаточно отнаследоваться от него и мы сможем:
 1. Генерировать события
 2. Подписываться на них



EventEmitter

JS index.js > ...

```
1  const EventEmitter = require('node:events');
2
3  class Interval extends EventEmitter {
4      constructor() {
5          super();
6          setInterval(() => this.emit('tick', 'some value'), 1000);
7      }
8  }
9
10 const interval = new Interval();
11 interval.on('tick', value => {
12     console.log(`tick: ${value}`);
13 });
```

Примерно так внутри "могли" бы быть устроены `net.Server` и `http.Server` (естественно, у них нет никакого `setInterval`): `this.emit(event, args)` – позволяет сгенерировать событие типа `event` и передать обработчикам аргументы (всё это попадёт на `callback` с 11-ой строки).



http.Server

У `http.Server` есть событие `request`, которое как раз возникает, когда нам приходит http-запрос (а не подключение клиента):

Event: 'request'

Added in: v0.1.0

- `request` `<http.IncomingMessage>`
 - `response` `<http.ServerResponse>`
- ← аргументы, которые будут приходить в `callback`
для метода `on`

Emitted each time there is a request. There may be multiple requests per connection (in the case of HTTP Keep-Alive connections).



http.Server

Соответственно и подписаться мы можем именно на него:

```
JS main.js > ...
1  'use strict';
2
3  const net = require('node:net');
4  const fs = require('node:fs');
5
6  const html = fs.readFileSync('index.html', 'utf-8');
7
8  const server = new http.Server();
9  server.on('request', function(request, response) {
10     response.setHeader('Content-Type', 'text/html; charset=utf-8');
11     response.end(html);
12 });
13
14 server.listen(9999);
```

Обратите внимание, мы не выставяем статус-код 200, не расставляем **CRLF** - всё будет сделано за нас:

- **setHeader** выставит лишь заголовок **Content-Type**
- **end** отправит ответ (заодно посчитав его размер)



http.Server

В онлайн-руководствах, вы чаще увидите использование функции `http.createServer`, которая за один вызов сделает то, что мы сделали за несколько:

JS index.js > ...

```
1  'use strict';
2
3  const net = require('node:net');
4  const fs = require('node:fs');
5
6  const html = fs.readFileSync('index.html', 'utf-8');
7
8  const server = http.createServer((request, response) => {
9    response.setHeader('Content-Type', 'text/html; charset=utf-8');
10   response.end(html);
11 });
12
13 server.listen(9999);
```



ИТОГИ



Итоги

В этой лекции мы обсудили достаточно много важных моментов:

1. Разобрались с контекстом вызова
2. Поговорили о функциях-конструкторах, прототипах
3. Поговорили о TCP/IP и начали рассматривать HTTP



ДОМАШНЕЕ ЗАДАНИЕ



ДЗ №1: Constructor

Создайте функцию-конструктор для объектов типа **Post**.

Функция-конструктор должна принимать следующие аргументы:

1. id
2. Название
3. Контент

Добавьте через прототип метод **preview**, который возвращает первое предложение из поля контент. Конец предложения определяется по символу **.** (точка).



ДЗ №2: Classes

Переделайте предыдущую задачу на базе классов **Post**.

Конструктор класса должен принимать следующие аргументы:

1. id
2. Название
3. Контент

Метод **preview** должен быть определён внутри класса.



ДЗ №3: Headers Echo

Создайте http-сервер, который работает на порту 9999 (как на лекции) и занимается следующей задачей:

1. Анализирует аргумент коллбека `request`, а именно его свойство `headers`
2. В ответ присылает все значения из `headers` в формате:
`name: value\n`
`name: value\n`
3. `Content-Type` должен быть выставлен как `text/plain`.



ДЗ №4: Image Service

Создайте http-сервер, который работает на порту 9999 (как на лекции) и занимается следующей задачей:

1. Анализирует аргумент коллбека `request`, а именно его свойство `url`
2. Ищет изображение с расширением `png` или `jpg`, если `url` содержит в конце файл с расширением `png` или `jpg`
3. Если файл найден – то отдаёт файл (так же, как `html`, только `Content-Type` для `png` должен быть `image/png`, а для `jpg` - `image/jpeg`)
4. Если файл не найден, то отдаёт статус-код 404 (см. метод `writeHead`)

В каталог к серверу положите две картинки `image.png` и `image.jpg` (любые, но не очень большие).



ДЗ №4: Image Service

Как это работает: запускаете сервер, вбиваете в адресную строку адреса:

- <http://localhost:9999/image.jpg> (в браузере должна отобразиться картинка)
- <http://localhost:9999/image.png> (в браузере должна отобразиться картинка)
- <http://localhost:9999/404.png> (должен быть статус код 404), как и на любые другие URL

Важно: возможно, вам придётся почитать документацию на модуль **fs**.



Спасибо за внимание

alif skills

2023г.

