

# DAY 1

12

6

 $\angle$ 

9

# План

---

1. Клиент-серверное взаимодействие
2. HTTP
3. Express
4. React SSR
5. Morgan, Nodemon

# Клиент и сервер

# Клиент и сервер

---

**Сервер** — программа, способная принимать и обрабатывать запросы от других программ.

**Клиент** — программа, которая отправляет запросы серверу и получает от него ответы.

# Клиент и сервер: работа по протоколу

---

**Протокол** — набор правил, по которым сервер и клиент обмениваются данными и “понимают” друг друга.

Клиент и сервер могут взаимодействовать друг с другом в рамках:

- одного компьютера,
- локальной сети или
- или целого интернета.

# Клиент и сервер

---



Клиент



Сервер

# Клиент и сервер

---



Клиент

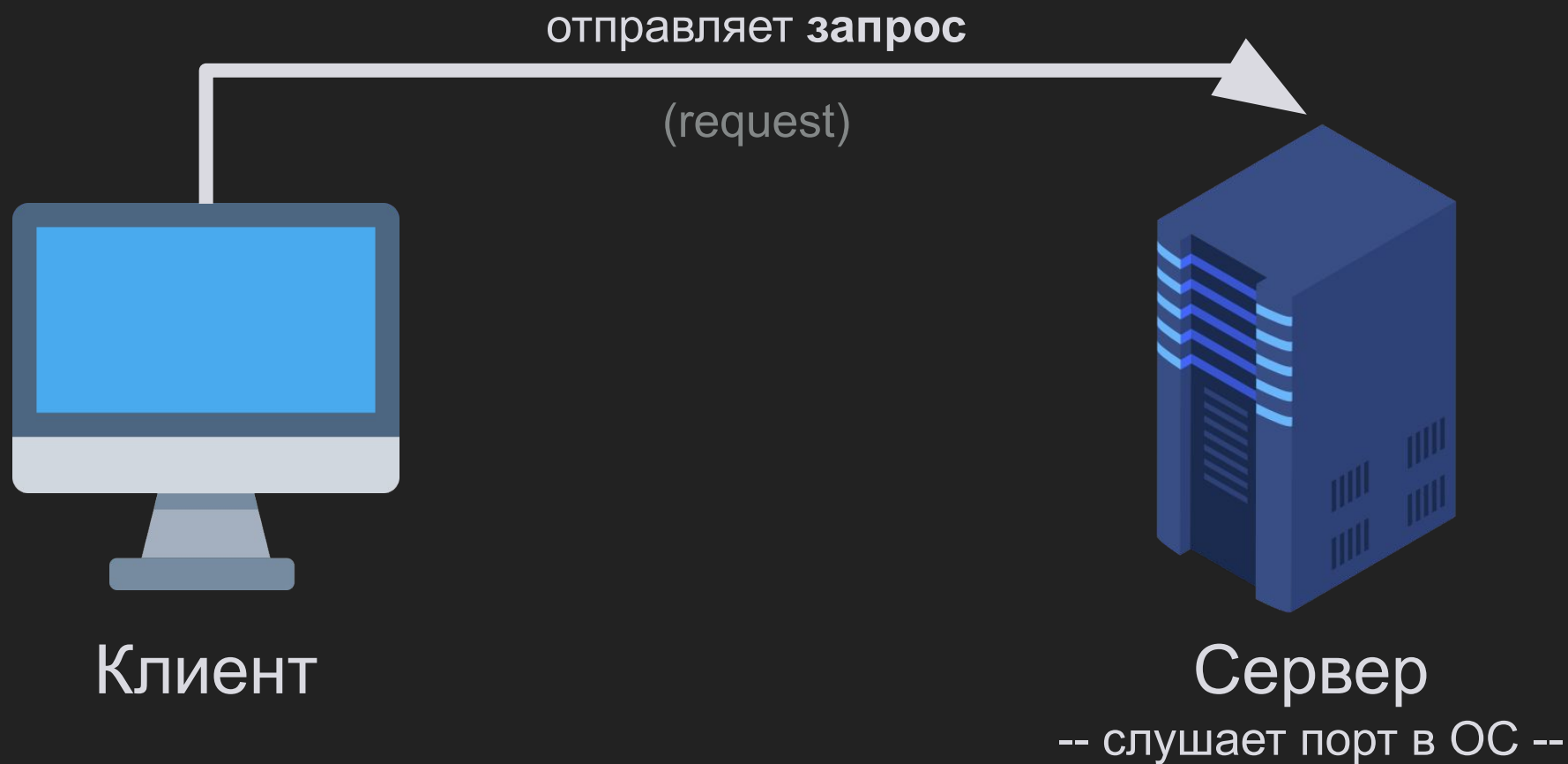


Сервер

-- слушает порт в ОС --

# Клиент и сервер

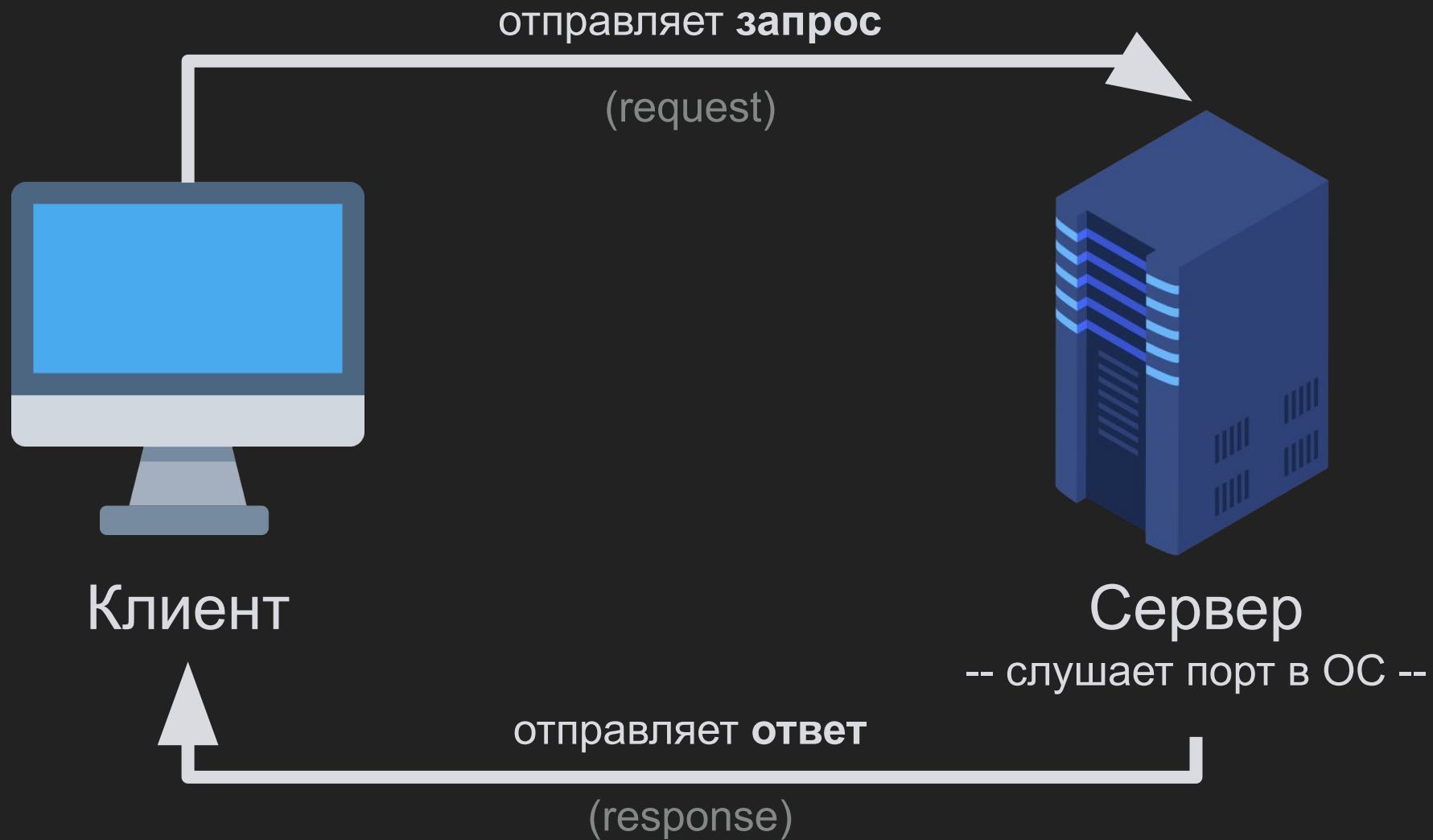
---





# Клиент и сервер

---



# HTTP

# HTTP

---

HyperText Transfer Protocol, HTTP — текстовый протокол, разработанный для передачи HTML.

Сейчас используется для передачи любой информации между веб-сервером и браузером.

Вы пользуетесь HTTP каждый раз, когда открываете любой сайт в интернете.

# HTTP

---

Пока мы будем работать только с версией **HTTP/1.1**

Версии 0.9, 1.0, 1.1 — текстовые протоколы.

Версии HTTP/2 и HTTP/3 (готовится к выходу) — бинарные.

# HTTP в TCP/IP

---

**HTTP** входит в состав прикладного уровня стека **TCP/IP**  
(сетевая модель передачи данных)

На стеке протоколов **TCP/IP** построено всё взаимодействие пользователей в IP-сетях.

# HTTP в TCP/IP, ports

---

На прикладном уровне (Application layer) **TCP/IP** работает большинство сетевых приложений.

Эти программы имеют свои собственные протоколы обмена информацией и дефолтное значение портов, например:

- интернет браузер для протокола **HTTP/HTTPS** (порты: 80/443),
- FTP-клиент для протокола FTP (порты: 20/21),
- почтовая программа для протокола SMTP (порт: 25),
- SSH для безопасного соединения с удалённой машиной (порт: 22)

# HTTP: URI, URL, URN

---

**HTTP-ресурс** (веб-ресурс) — любой ресурс, доступный в WWW

**URI** — унифицированный идентификатор ресурса

**URL** — унифицированный определитель местонахождения ресурса

**URN** — унифицированное имя ресурса

Каждый URL - это URI, но не наоборот.

Любой URI, в котором указан протокол - это URL (http, ftp, mailto и др.)

# HTTP: URI, URL, URN. Пример

---

Имя и адрес ресурса в сети, включает в себя URL и URN

**URI:** `https://somesite.ru/images/logo.png`

Адрес ресурса в сети, определяет местонахождение и способ обращения к нему

**URL** — `https://somesite.ru`

Имя ресурса в сети, определяет только название ресурса

**URN** — `images/logo.png`



# HTTP / URL

---

## Анатомия URL

`http://example.com:8080/some/path?showResp=1&path=no`



Протокол

Хост

Порт

Path

Query

# HTTP / Запрос

---

## HTTP-запрос (request)

Сообщение, отправленное клиентом на сервер.

Состоит из:

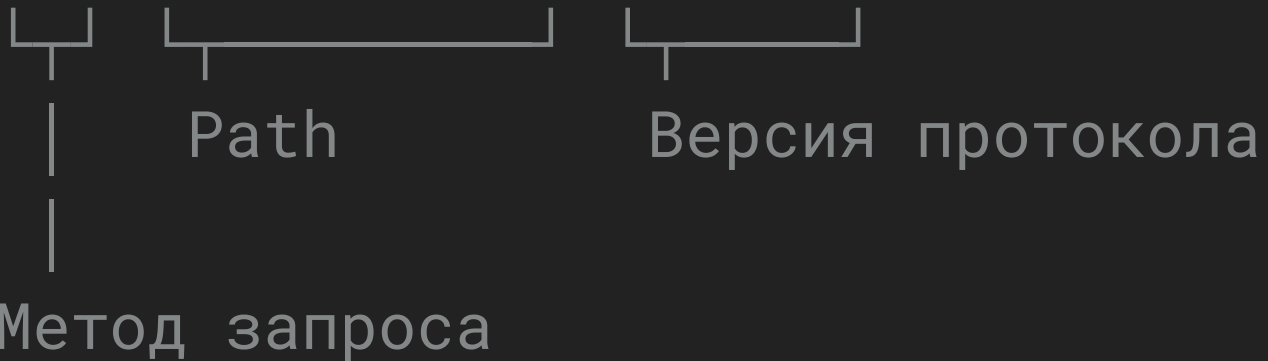
1. стартовой строки
2. заголовков
3. тела запроса

# HTTP / Запрос

---

## Стартовая строка

GET /about.html HTTP/1.1



# HTTP / Запрос

---

## HTTP-метод (HTTP-глагол)

Желаемое действие над веб-ресурсом, которое запрашивает клиент - прочитать, записать, удалить и т. д.

Реализация этого действия зависит от веб-сервера.

# HTTP / Запрос

---

У каждого из методов есть назначение.

- GET — получить данные (Read)
- POST — отправить данные (Create)
- PUT — изменить данные (Update)
- DELETE — удалить данные (Delete)

Список HTTP-методов: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>

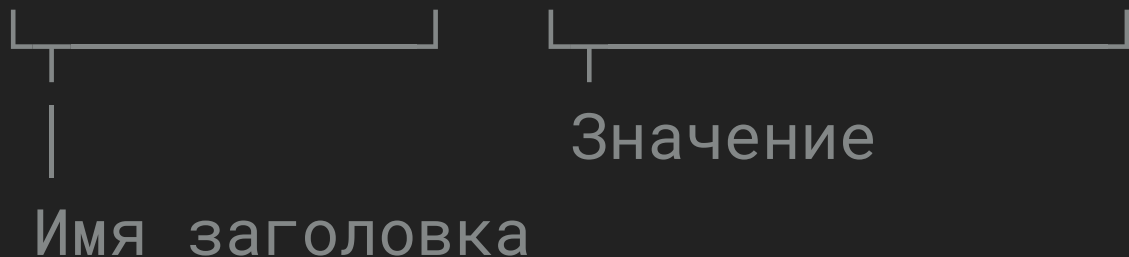
# HTTP / Заголовки

---

## Заголовки (HTTP headers)

Мета-информация о запросе или ответе.

Content-Type: `application/json`



# HTTP / Тело сообщения

---

## Тело сообщения

Может быть пустым, либо содержать произвольные данные.

Всегда отделяется пустой строкой.

# HTTP / Запрос

---

## Структура запроса

GET /students HTTP/1.1 — стартовая строка

Host: school.example

Accept-Language: ru-ru

User-Agent: Mozilla/5.0

┌  
├ заголовки  
└

(пустая строка)

(тело запроса — пустое или с данными)



# HTTP / Ответ

---

## HTTP-ответ (response)

Сообщение, отправленное сервером в ответ на запрос клиента.

Что будет, если оставить запрос без ответа?

# HTTP / Ответ

---

## HTTP-ответ (response)

Состоит из:

1. строки статуса (стартовой строки),
2. заголовков,
3. тела ответа.

# HTTP / Ответ

---

## Строка статуса

HTTP/1.1 200 OK

└──┬──┘

└┘

└┘

|

|

Текст кода ответа

|

Код ответа

Версия протокола

# HTTP / Ответ

---

**Код ответа (HTTP-код)**

Описывает результат запроса.

Например, код 404 — данные не найдены.

**Коды ответа** поделены на группы по смыслу:

**1\*\*** информационные коды

(например, смена протокола с HTTP на WebSocket)

**2\*\*** всё хорошо, запрос выполнен успешно

**3\*\*** переадресация, сервер просит перейти на другую страницу

**Коды ответа** поделены на группы по смыслу:

**4\*\*** ошибка со стороны клиента

(запрошен несуществующий адрес, переданы некорректные данные и т. д.)

**5\*\*** ошибка со стороны сервера

(сервер не смог обратиться к базе данных или другая внутренняя ошибка)

## Примеры кодов ответа

- 200 — всё ок, запрос выполнен успешно
- 201 — объект создан успешно
- 403 — доступ к данным запрещён
- 404 — запрошенные данные не найдены

# HTTP / Ответ

---

## Структура ответа

`HTTP/1.1 200 OK` — строка статуса (стартовая строка)

`X-Powered-By: Express` ⊥

`Content-Type: application/json` ⊥ заголовки

(пустая строка)

`{"students": [{"id": 1, "name": "Пётр Петров", "phone": "70001112233"}]}`

⊥ тело ответа



# HTTP / Ответ

---

## Структура ответа, пример №2

HTTP/1.1 200 OK — строка статуса (стартовая строка)

X-Powered-By: Express ↴

Content-Type: text/html ⊥ заголовки

(пустая строка)

<!doctype html><html><head>...

↳ тело ответа

# HTTP / Типы контента

---

## MIME type

Описание типа данных, передаваемых по сети. Например:

- `text/plain` — простой текст
- `text/html` — HTML-документ
- `application/json` — данные в JSON-формате
- `application/xml` — XML-документ
- `multipart/form-data` — бинарные данные

# HTTP / Типы контента

---

Для описания типа передаваемых и принимаемых данных используются заголовки `Content-Type` и `Accept` соответственно.

# Express

# Express: установка, базовый сервер

---

Установка: `npm install express`

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

```
const app = express();
```

```
app.get('/', function (req, res) {  
    res.send('Hello, world');  
});
```

```
app.listen(3000, () => { console.log('Server started') });
```

# Express: логирование запросов

---

Установка: `npm install morgan`

```
const express = require('express');  
const morgan = require("morgan");  
const app = express();  
  
app.use(morgan("dev"));  
  
// ...
```

# Express: чтение x-www-form-urlencoded, json

---

```
app.use(express.urlencoded({ extended: true })); // читать данные из тела запросов
app.use(express.json()); // читать JSON-данные из тела запросов
```

```
app.get('/test', function (req, res) {
  const data = req.query;
  res.send(JSON.stringify(data));
});
```

```
app.post('/test', function (req, res) {
  const data = req.body;
  res.send(JSON.stringify(req.body));
});
```

# Express: ВИДЫ ОТВЕТОВ

---

```
res.send(text)           // послать текст с кодом 200 + завершить ответ
res.json({ user: 'tobi' }) // послать json с кодом 200 + завершить ответ
res.end()                // завершить ответ
res.status(403)          // установить код, но НЕ завершить ответ

// установить код 500, послать json + завершить ответ
res.status(500).json({ error: 'message' })

res.status(404).end()     // установить код + завершить ответ
res.redirect('/other-route') // переадресовать клиента + завершить ответ
res.render(view, {data})  // рендер шаблона + послать html + завершить ответ
```



# React SSR

# React SSR в тезисах

---

- React — JS-библиотека для отрисовки пользовательских интерфейсов (user interfaces, UI)
- React SSR (Server Side Rendering) - рендеринг html на сервере с помощью React
- позволяет вставлять данные в заранее подготовленные HTML-шаблоны
- JSX — позволяет писать HTML прямо в js-файлах
- шаблонизаторы отделяют View от всего остального
- слой View в MVC

# React SSR: необходимое для работы

---

Установка Babel и React:

```
npm install @babel/core @babel/preset-env @babel/preset-react  
@babel/register react react-dom
```

**Babel** позволяет подключать jsx файлы (JavaScript, в котором можно писать HTML). Чтобы подключить Babel в главном js-файле сверху нужно добавить строчку:

```
require( '@babel/register' );
```

В корень проекта нужно добавить файл “.babelrc” с таким содержимым:

```
{  
  "presets": [ "@babel/preset-env", "@babel/preset-react" ]  
}
```

# React SSR: компонент обёртка Layout

---

Layout — главный компонент, в который будем вставлять остальные.

```
// views/Layout.jsx
const React = require('react');

module.exports = function Layout({ title, children }) {
  return (
    <html lang="en">
      <head>
        <title>{title}</title>
        <link rel="stylesheet" href="style.css" />
        <script src="script.js" />
      </head>
      <body>{children}</body>
    </html>
  );
};
```

Diagram illustrating the props passed to the Layout component:

```
graph TD
    A["{ title, children }"] --> B["Props"]
```

# React SSR: тонкости HTML

---

```
// views/Home.jsx
const React = require('react');
const Layout = require('./Layout');

module.exports = function Home({ title, name }) {
  return (
    <Layout title={title}>      _____ все теги должны быть закрыты
      <input type="text" /><br />
      <h1 className="title" style={{color: "red"}}>Hello, {name}</h1>
    </Layout>      _____
  );              className вместо class      style задаётся как объект,
};                                                  а не как строка
```

# React SSR: пример ответа для GET запроса

---

```
// app.js
const ReactDOMServer = require('react-dom/server');
const React = require('react');
const Home = require('./views/Home');

// Отображаем главную страницу с использованием компонента "Home"
app.get('/', (req, res) => {
  // создаём React-элемент на основе React-компонента
  const home = React.createElement(Home, {
    title: 'My site',
    name: 'John',
  });
  // рендерим элемент и получаем HTML (в виде строки)
  const html = ReactDOMServer.renderToStaticMarkup(home);
  // отправляем первую строку нашего HTML-документа
  res.write('<!DOCTYPE html>');
  // отправляем отрендеренный HTML и закрываем соединение
  res.end(html);
});
```



# React SSR: условный рендеринг

---

Что можно делать:

```
// условный рендеринг (в зависимости от условия)
// if
{author && <h1>{author.firstName} {author.lastName}</h1>}
// if/else
{author
  ? <h1>{author.firstName} {author.lastName}</h1>
  : <h1>Unknown author</h1>}
```



# React SSR: рендер списка

---

Что можно делать:

```
// ...если people - это массив
```

```
{ people: [{id: 1, name: "A", age: 15}, {id: 2, name: "B", age: 21}] }
```

```
// использовать map, чтобы отрендерить массив
```

```
<ul>  
  {people.map((person) => (  
    <li key={person.id}>Name:{person.name}, age: {person.age}</li>  
  )))  
</ul>
```

key - уникальный ключ элемента массива  
(если его нет, можно использовать index)

# Morgan, Nodemon

# Morgan

---

Выводит в консоль сервера информацию о входящих запросах и ответах на них.

Установка:

```
npm install morgan
```

Подключение:

```
app.use(morgan("dev"));
```

# Nodemon

---

Автоматически перезапускает сервер при изменении файлов для удобства разработки.

Установка: `npm i -D nodemon`

Настройка отслеживания файлов: `--ext js,json,jsx`

Вместо `node server.js` пишем `nodemon server.js`

# Документация

---

- <https://expressjs.com/en/api.html#res>
- <https://ru.reactjs.org/>