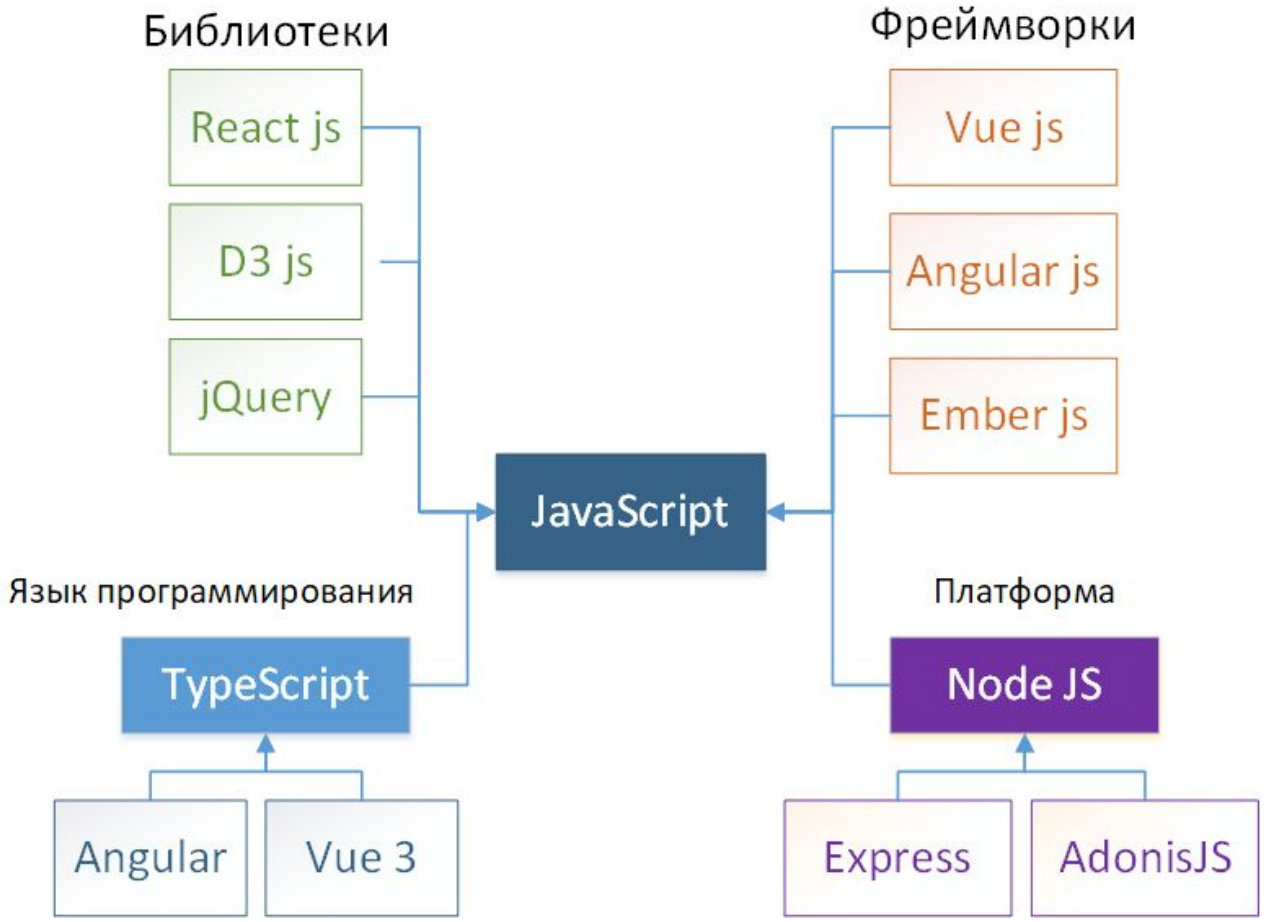
Web-программирование

Лекция 1. Основы Node.js

Технологии JavaScript



Основы Node.js

- **Node.js** представляет среду выполнения кода на JavaScript, которая построена на основе движка *JavaScript Chrome V8*, который позволяет транслировать вызовы на языке JavaScript в машинный код.
- **Node.js** прежде всего предназначен для создания *серверных приложений* на языке JavaScript, хотя также существуют проекты по написанию десктопных приложений (к примеру, фреймворк *Electron*, разработанный *GitHub*) и даже по созданию кода для микроконтроллеров.
- Но прежде всего мы говорим о Node.js, как о *платформе для создания веб-приложений*.
- Для разработки под *Node.js* достаточно простейшего текстового редактора.

Инструмент REPL

- Для загрузки перейдите на официальный сайт https://nodejs.org/en/.
- После установки на компьютер *Node.js* нам становится доступным инструмент **REPL**.
- **REPL (**Read Eval Print Loop**)** представляет возможность запуска выражений на языке JavaScript в *командной строке* (ОС Windows) или *терминале* (OS X, Linux).
- При запуске командной строки (cmd) или терминала можно ввести команду **node**.
- После ввода этой команды можно выполнять различные выражения на JavaScript.
- Есть несколько команд, которые позволят управлять данным режимом. К примеру, команда **«.exit»** позволит выйти из режима **REPL**.

Инструмент REPL

- Пример:

```
C:\>node
Welcome to Node.js v13.9.0.
Type ".help" for more information.
> var a = 10
undefined
> a + 5
15
> function f(x) { return x*x }
undefined
> f(13)
169
> console.log("Квадрат 11 = " + f(11));
Квадрат 11 = 121
undefined
>
```

- При ошибке в коде **REPL** укажет об этом:

```
> f1(11)
Uncaught ReferenceError: f1 is not defined
```

Инструмент REPL

Выполнение файла

- Вместо того чтобы вводить весь код напрямую в консоль, удобнее вынести его во внешний файл.
- Например, создадим файл *app.js* с кодом:

```
console.log("Hello World!")
```

- Он просто выводит текст в консоль. Выполним его командой **node**:

```
C:\>node C:\app.js
Hello World!
```

Модули

- Node.js использует **модульную систему**.
- Вся встроенная функциональность разбита на отдельные **пакеты** или **модули**.
- Модуль представляет блок кода, который может использоваться в других модулях.
- При необходимости мы можем подключать нужные нам модули.
- В документации (https://nodejs.org/api/) можно узнать о встроенных модулях Node.js и их функциональности.

Модули

- Для загрузки модулей применяется функция **require()**.
- К примеру, для получения и обработки запроса был необходим модуль **http**.
- Создадим файл *app.js*:

```
const http = require("http");
http.createServer(function(request,response){
    response.end("Hello NodeJS!");
}).listen(3000, "127.0.0.1",function(){
    console.log("Сервер запущен на порту 3000");
});
```

```
C:\>node app
Сервер запущен на порту 3000
```

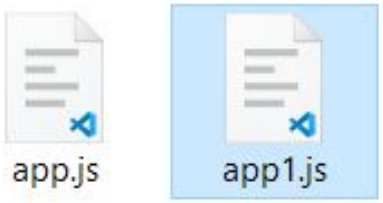
Модули

- Подобным образом мы можем загружать и использовать другие ***встроенные модули***.
- Например, используем модуль **os**, который предоставляет информацию об окружении и ОС.

```
const os = require("os");
```

Пользовательские модули

- При необходимости можно создавать *свои* модули.
- Но в отличие от встроенных модулей для подключения своих модулей надо передать в функцию **require** относительный путь с именем файла.
- К примеру, если файл «*app1.js*» располагается рядом с запускаемым файл



- Для его исполнения можно написать:

```
const a = require("./app1");
```

Пользовательские модули

Передача данных из модуля

- Если мы описываем переменные или функции внутри модуля, то извне они недоступны.
- Чтобы они стали доступны, необходимо определить их в объекте **module.exports**.
- Объект **module.exports** – это то, что возвращает функция ***require***() при получении модуля.
- Объект **module** представляет ссылку на текущий модуль, а его свойство **exports** определяет все свойства и методы модуля, которые могут быть экспортированы и использованы в других модулях.

Пользовательские модули

- Например, создан файл модуля ***moduleTime.js***:

```
let currentDate = new Date();
module.exports.date = currentDate;
```

- Подключим его в файле ***app.js***:

```
const time1 = require("./moduleTime");
console.log(time1.date); //выводим дату
```

```
C:\>node app
2022-01-24T11:11:46.662Z
```

- Сразу стоит отметить, что подключаемые модули кэшируются. Если несколько раз получить модуль в разные константы, то все они будут указывать на один и тот же объект.

Пользовательские модули

Передача конструктора из модуля

- Кроме определения простейших функций или свойств в модуле могут определяться сложные объекты или функции конструкторов.
- Пример:
- Создадим файл модуля ***user.js***:

```
function User(name, age){
    this.name = name;
    this.age = age;
    this.displayInfo = function(){
        console.log(`Имя: ${this.name}  Возраст: ${this.age}`);
    }
}
module.exports = User;
```

Пользовательские модули

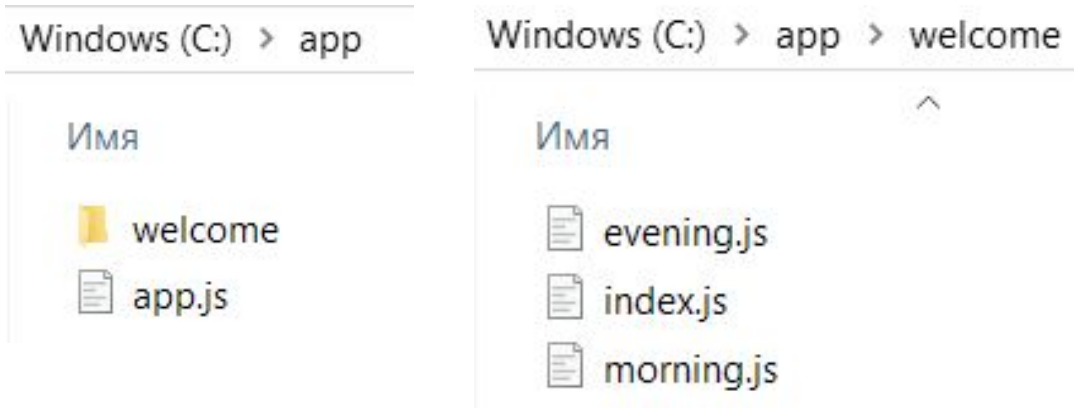
- Весь модуль ***user*** указывает на определенную функцию конструктора.
- Файл ***app.js***:

```
const User = require("./user.js");
let person = new User("Tom", 32);
person.displayInfo();
```

```
C:\>node app
Имя: Tom  Возраст: 32
```

Структура модулей

- Нередко модули приложения образуют какие-то отдельные *наборы* или *области*. Такие наборы модулей лучше помещать в отдельные каталоги.
- Например, у нас есть в каталоге приложения подкаталог ***welcome,*** а в нем три файла: *index.js, morning.js, evening.js.* В итоге общая структура проекта:



Структура модулей

- Файл ***morning.js***:

```
module.exports = "Доброе утро";
```

- Файл ***evening.js***:

```
module.exports = "Добрый вечер";
```

- Файл ***index.js***:

```
const morning = require("./morning");
const evening = require("./evening");
module.exports = {
    getMorningMessage : function(){
        console.log(morning);
    },
    getEveningMessage : function(){
        console.log(evening);
    }
}
```

Структура модулей

- Теперь используем модуль ***index.js*** в файле ***app.js***:

```
const welcome = require("./welcome");
welcome.getMorningMessage();
welcome.getEveningMessage();
```

- Так как файла ***welcome.js*** нет, но есть каталог *welcome*, который содержит файл с именем ***index.js***, то можно обращаться к модулю по имени каталога.

```
C:\app>node app
Доброе утро
Добрый вечер
```

Объект global и глобальные переменные

- Node.js предоставляет специальный объект **global**, который предоставляет доступ к глобальным переменным и функциям. Примерным аналогом данного объекта в JavaScript является объект **window**.
- Для примера создадим следующий модуль ***greeting.js***:

```
let currentDate = new Date();
global.date = currentDate;
module.exports.getMessage = function(){
    let hour = currentDate.getHours();
    if(hour >16) return "Добрый вечер, " + global.name;
    else if(hour >10) return "Добрый день, " + name;
    else return "Доброе утро, " + name;
}
```

Объект global и глобальные переменные

- Определим файл приложения ***app.js***:

```
const greeting = require("./greeting");
global.name = "Tom";
global.console.log(date);
console.log(greeting.getMessage());
```

- Здесь устанавливаем глобальную переменную *name*, которую мы получаем в модуле ***greeting****.js* и также выводим на консоль глобальную переменную *date*.

```
C:\app>node app
2022-01-24T11:30:44.972Z
Добрый вечер, Tom
```

Передача параметров приложению

- При запуске приложения из терминала/командной строки можно передавать ему параметры.
- Для получения параметров в коде приложения применяется массив **process.argv**.
- **Первый элемент** этого массива всегда указывает на путь к файлу *node.exe*, который вызывает приложение.
- **Второй элемент** массив всегда указывает на путь к файлу приложения, который выполняется.

Передача параметров приложению

- К примеру, определим следующий файл ***app.js***:

```
let nodePath = process.argv[0]; // путь к node.exe
let appPath = process.argv[1];  // путь к app.js
let name = process.argv[2]; // параметр приложения 1
let age = process.argv[3];  // параметр приложения 2
console.log("nodePath: " + nodePath);
console.log("appPath: " + appPath);
console.log("name: " + name);
console.log("age: " + age);
```

Передача параметров приложению

- Запуск приложения без параметров и с ними:

```
C:\app>node app
nodePath: C:\Program Files\nodejs\node.exe
appPath: C:\app\app
name: undefined
age: undefined

C:\app>node app Tom 23
nodePath: C:\Program Files\nodejs\node.exe
appPath: C:\app\app
name: Tom
age: 23
```

Установка модулей

- Кроме встроенных и пользовательских модулей Node.js существует огромное количество различных библиотек, фреймворков и утилит, созданных сторонними производителями и которые можно использовать в проекте, например, *express*, *grunt*, *gulp* и т.д.
- Чтобы удобнее было работать со всеми сторонними решениями, они распространяются в виде **пакетов**.
- Для автоматизации установки и обновления пакетов в Node.js используется пакетный менеджер **NPM** (Node Package Manager), который устанавливается вместе с Node.js. Но можно обновить установленную версию до последней.

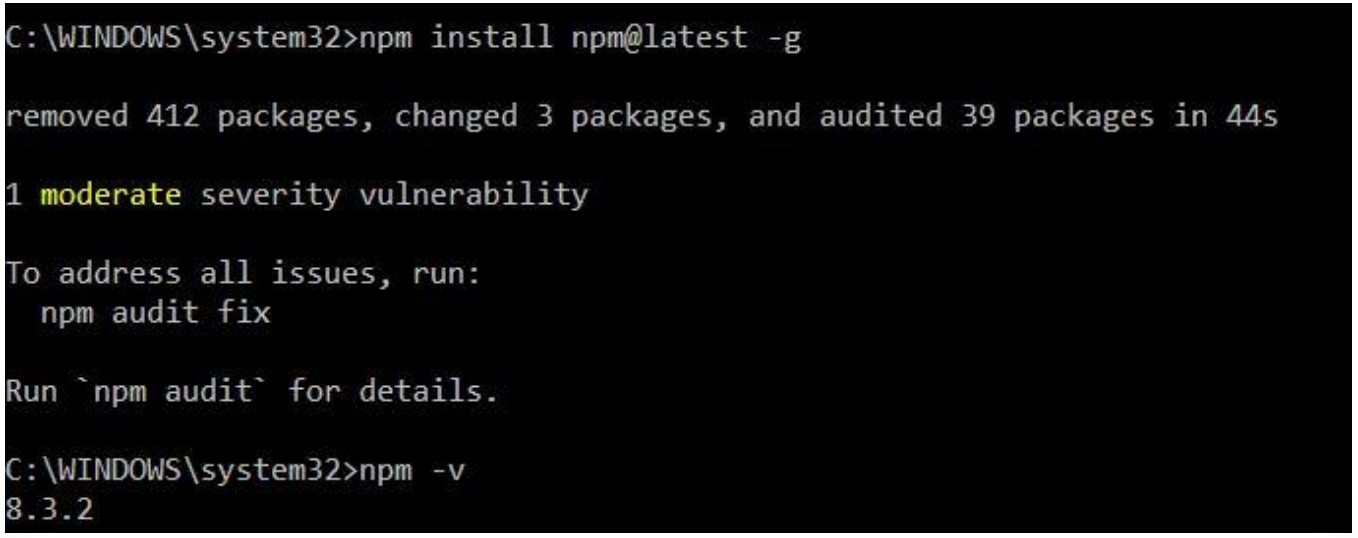
Установка модулей

- Можно обновить установленную версию **NPM** до последней:

```
npm install npm@latest -g
```

- Может потребоваться запустить командную строку от имени администратора.
- Чтобы узнать текущую версию **NPM**:

```
npm -v
```



Установка модулей

Файл package.json

- Для более удобного управления конфигурацией и пакетами приложения в **NPM** применяется файл конфигурации **package.json**.
- К примеру, у нас имеется каталог проекта ***modulesapp***. Добавим в него файл ***package.json***:

```
{
  "name": "modulesapp",
  "version": "1.0.0"
}
```

- Здесь определены только две секции: имя проекта – *modulesapp* и его версия - 1.0.0. Это минимально необходимое определение файла *package.json*.

Установка модулей

- Далее для примера установим в проект пакет ***Express***.
- Для установки функциональности *Express* в проект вначале перейдем в каталог проекта, затем введем команду:

```
npm install express
```

```
C:\>cd modulesapp

C:\modulesapp>npm install express

added 50 packages, and audited 51 packages in 11s

found 0 vulnerabilities
```

Установка модулей

- После установки *Express* в папке проекта *modulesapp* появится подпапка ***node_modules***, в которой будут храниться все установленные внешние модули.
- В файл *package.json* также добавится информация о данном модуле:

```
{
  "name": "modulesapp",
  "version": "1.0.0",
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

- Информация обо всех добавляемых пакетах, которые используются при работе приложения, добавляется в секцию **dependencies**.

Установка модулей

- Можно устанавливать множество пакетов одной командой.
- В этом случае мы можем определить все необходимые пакеты в файле *package.json* и потом одной командой их установить. К примеру:

```
{
  "name": "modulesapp",
  "version": "1.0.0",
  "dependencies": {
    "express": "^4.17.1",
    "react": "^16.9.0",
    "react-dom": "^16.9.0"
  }
}
```

- Затем для загрузки всех пакетов выполнить команду

```
npm install
```

Установка модулей

- Для удаления пакетов используется команда *npm uninstall*. Например:

```
npm uninstall express
```

- Если нам надо удалить не один пакет, а несколько, то мы можем удалить их определение из файла *package.json* и ввести команду *npm install*, и удаленые из *package.js* пакеты также будут удалены из папки *node_modules*.

Команды NPM

- NPM позволяет определять в файле *package.json* команды, которые выполняют определенные действия.
- Например, определим следующий файл ***app.js***:

```
let name = process.argv[2]; // параметр приложения 1
let age = process.argv[3]; // параметр приложения 2
console.log("name: " + name + ", age: " + age);
```

Команды NPM

- Определим следующий файл ***package.json***:

```
{
    "name": "modulesapp",
    "version": "1.0.0",
  ▷ Debug
    "scripts" : {
      "start" : "node app.js",
      "dev" : "node app.js Tom 26"
    }
}
```

- Здесь добавлена секция **scripts**, которая определяет две команды.
- Названия и количество команд могут быть произвольными.

Команды NPM

- Но надо учитывать, что есть некоторые предопределенные названия для команд, например, ***start***, ***test***, ***run*** и т.д.

- И для их выполнения в терминале/командной строке надо выполнить команду в каталоге приложения:

```
npm [название_команды]
```

- Команды с остальными названиями (например, «*dev*») запускаются так:

```
npm run [название_команды]
```

Команды NPM

- Например, последовательно выполним обе команды:

```
C:\modulesapp>npm start

> modulesapp@1.0.0 start
> node app.js

name: undefined
age: undefined

C:\modulesapp>npm run dev

> modulesapp@1.0.0 dev
> node app.js Tom 26

name: Tom
age: 26
```

Асинхронность в Node.js

- Асинхронность представляет возможность одновременно выполнять сразу несколько задач.
- Например, допустим в файле приложения ***app.js*** у нас расположен следующий синхронный код:

```
function display(data){
    console.log(data);
}
console.log("Начало работы программы");
display("Обработка данных...");
console.log("Завершение работы программы");
```

```
C:\>node app
Начало работы программы
Обработка данных...
Завершение работы программы
```

Асинхронность в Node.js

- Для рассмотрения асинхронности изменим код файла ***app.js*** следующим образом:

```
function display(data){
    setTimeout(function(){
        console.log(data);
    }, 0);
}
console.log("Начало работы программы");
display("Обработка данных...")
console.log("Завершение работы программы");
```

- Теперь вывод сообщения «*Обработка данных…*» будет выполняться асинхронно с остальным кодом.
- Для того используется функция **setTimeout()**.

Асинхронность в Node.js

- Результат:

```
C:\>node app
Начало работы программы
Завершение работы программы
Обработка данных...
```

- Несмотря на то, что в *setTimeout* передается промежуток **0**, фактическое выполнение функции *display* завершается после всех остальных функций, которые определены в программе. В итоге исполнение кода на функции *display* не блокируется, а идет дальше.
- Такое происходит из-за того, что функции обратного вызова в асинхронных функциях помещаются в специальную **очередь**, и начинают выполняться **после** того, как все остальные синхронные вызовы в приложении завершат свою работу.

События

- Подавляющее большинство функционала Node.js применяет **асинхронную событийную архитектуру**, которая использует специальные объекты – **эмиттеры** для генерации различных событий, которые обрабатываются специальными функциями – **обработчиками или слушателями событий**.
- Весь необходимый функционал сосредоточен в модуле **events**.
- Объекты, которые генерируют события, – экземпляры класса **EventEmitter**.
- С помощью функции **eventEmitter.on**() к определенному событию по имени цепляется функция обработчика. Причем для одного события можно указать множество обработчиков.

События

- Для примера определим следующий файл *app.js*:

```
const Emitter = require("events");
let emitter = new Emitter();
let eventName = "event1";
emitter.on(eventName, function(){ // обработчик 1
    console.log("Hello all!");
});
emitter.on(eventName, function(){ // обработчик 2
    console.log("Привет!");
});
emitter.emit(eventName); // генерирует событие по имени
```

- Для генерации события выполняется функция **emitter.emit()**.

```
C:\modulesapp>node app
Hello all!
Привет!
```

События

Передача параметров событию

- При вызове события в качестве второго параметра в функцию **emit** можно передавать некоторый объект, который передается в функцию обработчика события:

```
const Emitter = require("events");
let emitter = new Emitter();
let eventName = "event1";
emitter.on(eventName, function(data){
    console.log(data);
});
emitter.emit(eventName, "Привет мир!");
```

```
C:\app>node app
Привет мир!
```

События

Наследование от EventEmitter

- В приложении мы можем оперировать сложными объектами, для которых также можно определять события, но для этого их надо связать с объектом EventEmitter:

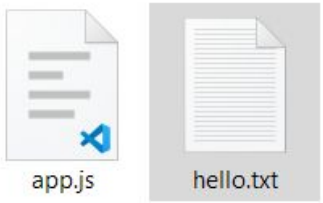
```
const EventEmitter = require("events");
let eventName = "event1";
class User extends EventEmitter { // наследование функционала
    sayHi(data) {
        this.emit(eventName, data);
    }
}
let user = new User();
// добавляем обработчик события "event1"
user.on(eventName, function(data){
    console.log(data);
});
user.sayHi("Hi");
```

```
C:\modulesapp>node app
Hi
```

Работа с файловой системой. Модуль fs

Чтение из файла

- Допустим, в одной папке с файлом приложения *app.js* расположен текстовый файл *hello.txt* с простейшим текстом, например «`Hello World`».



- Для чтения файла в синхронном варианте применяется функция **fs.readFileSync()**:

```
let fileContent = fs.readFileSync("hello.txt", "utf8");
```

- На выходе получаем считанный текст.

Работа с файловой системой. Модуль fs

- Для асинхронного чтения файла применяется функция **fs.readFile()**:

```
fs.readFile("hello.txt", "utf8", function(error,data){ });
```

- Третий параметр здесь – функция обратного вызова, которая выполняется после завершения чтения:
- Первый параметр этой функции – *информация об ошибке*, второй – *считанные данные*.

Работа с файловой системой. Модуль fs

- Для чтения файла определим в файле *app.js* следующий код:

```
const fs = require("fs");
// асинхронное чтение
fs.readFile("hello.txt", "utf8", function(error,data){
  console.log("Асинхронное чтение файла");
  if(error) throw error; // если возникла ошибка
  console.log(data);  // выводим считанные данные
});
// синхронное чтение
console.log("Синхронное чтение файла")
let fileContent = fs.readFileSync("hello.txt", "utf8");
console.log(fileContent);
```

```
C:\modulesapp>node app
Синхронное чтение файла
Hello World
Асинхронное чтение файла
Hello World
```

Работа с файловой системой. Модуль fs

Запись файла

- Для записи файла в синхронном варианте используется функция **fs.writeFileSync**(), которая в качестве параметра принимает путь к файлу и записываемые данные:

```
fs.writeFileSync("hello.txt", "Привет мир!")
```

- Также для записи файла можно использовать асинхронную функцию **fs.writeFile()**:

```
fs.writeFile("hello.txt", "Привет МИР!")
```

- В качестве вспомогательного параметра в функцию может передаваться функция обратного вызова, которая выполняется после завершения записи.

Работа с файловой системой. Модуль fs

- Пример:

```
const fs = require("fs");
fs.writeFile("hello.txt", "Hello мир!", function(error){
    if(error) throw error; // если возникла ошибка
    console.log("Асинхронная запись файла завершена. Содержимое файла:");
    // проверяем записанное
    let data = fs.readFileSync("hello.txt", "utf8");
    console.log(data);  // выводим считанные данные
});
```

```
C:\modulesapp>node app
Асинхронная запись файла завершена. Содержимое файла:
Hello мир!
```

- Данные методы полностью перезаписывают файл.
- Если надо осуществить дозапись, то применяются методы **fs.appendFile**() / **fs.appendFileSync**().

Работа с файловой системой. Модуль fs

Удаление файла

- Для удаления файла в синхронном варианте используется функция **fs.unlinkSync**(), которая в качестве параметра принимает путь к удаляемому файлу:

```
fs.unlinkSync("hello.txt")
```

- Также для удаления файла можно использовать асинхронную функцию **fs.unlink**(), которая принимает путь к файлу и функцию, вызываемую при завершении удаления.

Потоки данных

- Объект **Stream** представляет поток данных.
- Потоки бывают различных типов, среди которых можно выделить **потоки для чтения** и **потоки для записи**.
- К примеру, при создании сервера используются

```
const http = require("http");
http.createServer(function(request, response){}).listen(3000);
```

- Параметры **request** и **response**, которые передаются в функцию, и с помощью которых мы можем получать данные о запросе и управлять ответом, как раз представляют собой потоки *для чтения* и *для записи* соответственно.

Потоки данных

Используя потоки чтения и записи, можно считывать и записывать информацию в файл.

Поток записи

- Для создания потока для записи применяется метод **fs.createWriteStream()**, в который передается название файла.
- Запись данных производится с помощью метода **write**(), в который передаются данные.
- Для окончания записи вызывается метод **end**().

Потоки данных

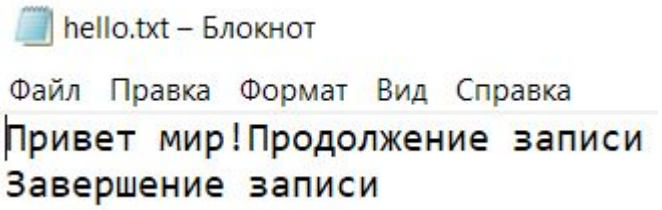
Поток чтения

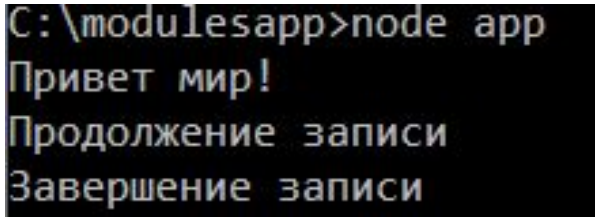
- Для создания потока для чтения используется метод **fs.createReadStream**(), в который также передается название файла.
- Сам поток разбивается на ряд кусков или *чанков* (*chunk*).
- При считывании каждого такого куска, «автоматически» возникает событие ***data***.

Потоки данных

- Пример:

```
const fs = require("fs");
// поток записи
let writeableStream = fs.createWriteStream("hello.txt");
writeableStream.write("Привет мир!");
writeableStream.write("Продолжение записи \n");
writeableStream.end("Завершение записи");
// поток чтения
let readableStream = fs.createReadStream("hello.txt", "utf8");
readableStream.on("data", function(chunk){
    // выводим чанки для просмотра
    console.log(chunk);
});
```





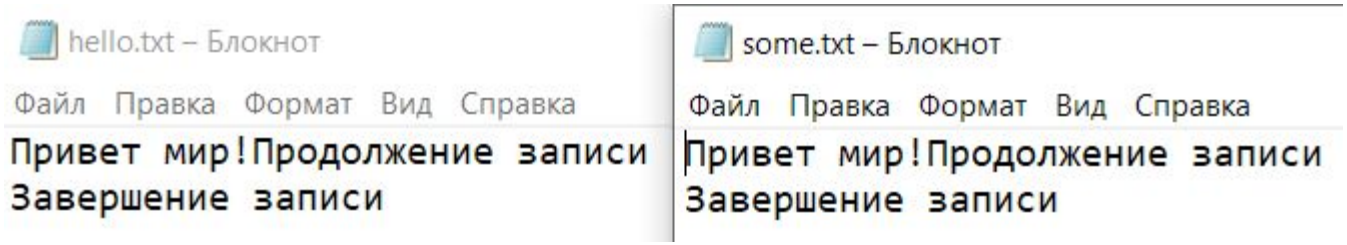
Канал Pipe

- **Pipe** – это канал, который связывает поток для чтения и поток для записи и позволяет сразу считать из потока чтения в поток записи.
- Рассмотрим пример с копированием данных из одного файла в другой.
- Эта задача является довольно распространенной, и в этом случае каналы (pipe) позволяют нам сократить объем кода.

Канал Pipe

- Скопируем содержимое файла *hello.txt* в новый файл *some.txt*:

```
const fs = require("fs");
let readableStream = fs.createReadStream("hello.txt", "utf8");
let writeableStream = fs.createWriteStream("some.txt");
readableStream.pipe(writeableStream);
```



- У потока чтения вызывается метод **pipe**(), в который передается поток для записи.

Клиент-серверная архитектура приложения

- Протокол *HTTP* предоставляет набор методов для указания целей запроса, отправляемого *серверу*.
- Эти методы основаны на дисциплине ссылок, где для указания ресурса, к которому должен быть применен данный метод, используется универсальный *идентификатор ресурсов* (Universal Resource Identifier, ***URI***) в виде *местонахождения ресурса* (Universal Resource Locator, ***URL***) или в виде его *универсального имени* (Universal Resource Name, ***URN***).
- Протокол HTTP реализует **принцип запрос/ответ**.

Клиент-серверная архитектура приложения

- Запрашивающая *программа–клиент* инициирует взаимодействие с отвечающей программой–*сервером*, и посылает **запрос**, содержащий:
  - метод доступа;
  - адрес URL;
  - версию протокола;
  - сообщение с информацией о типе передаваемых данных, информацией о *клиенте*, пославшем запрос, и, возможно, с содержательной частью (телом) сообщения.
- **Ответ** *сервера* содержит:
  - строку состояния, в которую входит версия протокола и *код возврата* (успех или ошибка);
  - сообщение, в которое входит информация *сервера*, метаинформация и *тело сообщения*.
- Соединение, как правило, открывает клиент, а сервер после отправки ответа инициирует его разрыв.

Создание сервера

- Для работы с сервером и протоколом HTTP в Node.js используется модуль **http**.
- Чтобы создать сервер, следует вызвать метод **http.createServer()**:

```
const http = require("http");
http.createServer().listen(3000);
```

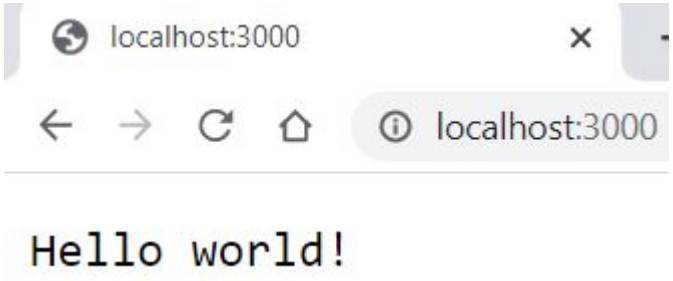
- Данный метод возвращает объект **http.Server**.
- Чтобы сервер мог прослушивать и обрабатывать входящие подключения, у объекта сервера необходимо вызвать метод **listen**().

Создание сервера

- Для обработки подключений можно передать особую функцию:

```
const http = require("http");
http.createServer(function(request, response){
    response.end("Hello world!");
}).listen(3000);
```

```
C:\app>node app
```



- Эта функция принимает два параметра:
  - **request** – хранит информацию о запросе;
  - **response** – управляет отправкой ответа.

Создание сервера

Request

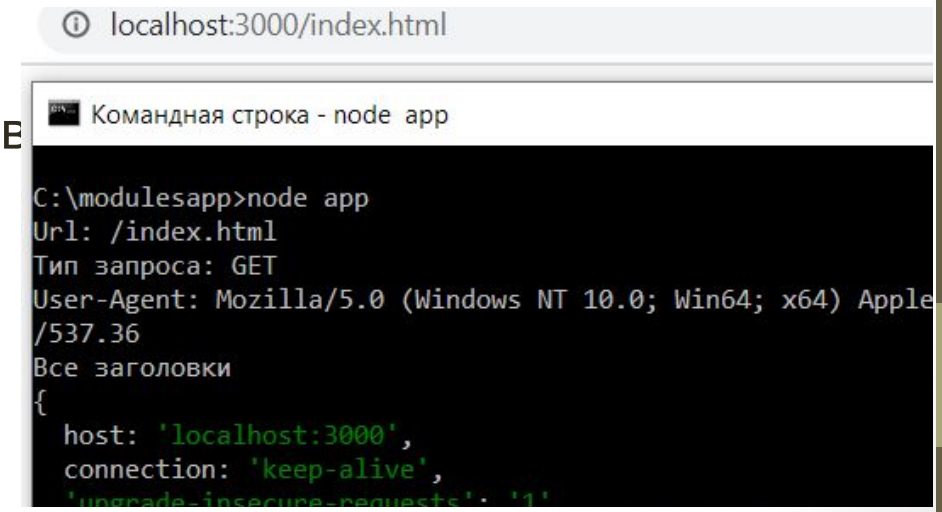
- Параметр **request** позволяет получить информацию о запросе и представляет объект **http.IncomingMessage**.
- Отметим некоторые основные свойства этого объекта:
  - *headers*: возвращает заголовки запроса;
  - *method*: тип запроса (GET, POST, DELETE, PUT);
  - *url*: представляет запрошенный адрес.

Создание сервера

- Например, определим следующий файл *app.js*:

```
var http = require("http");
http.createServer(function(request, response){
    console.log("Url: " + request.url);
    console.log("Тип запроса: " + request.method);
    console.log("User-Agent: " + request.headers["user-agent"]);
    console.log("Все заголовки");
    console.log(request.headers);
    response.end();
}).listen(3000);
```

- Запустим его и обратимся в браузере по адресу *http://localhost:3000/index.html*:



Создание сервера

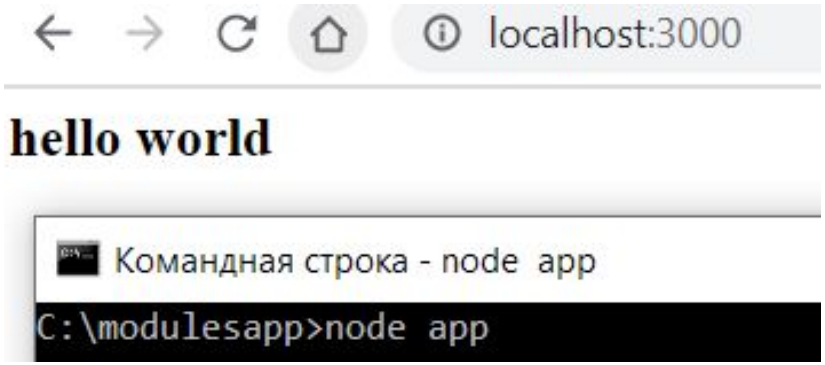
Response

- Параметр **response** управляет отправкой ответа и представляет объект **http.ServerResponse**.
- Среди его функциональности можно выделить следующие методы:
  - *statusCode*: устанавливает статусный код ответа;
  - *statusMessage*: устанавливает сообщение, отправляемое вместе со статусным кодом;
  - *setHeader*(name, value): добавляет в ответ один заголовок;
  - *write*: пишет в поток ответа некоторое содержимое;
  - *writeHead*: добавляет в ответ статусный код и набор заголовков;
  - *end*: сигнализирует серверу, что заголовки и тело ответа установлены, в итоге ответ отсылается клиента. Данный метод должен вызываться в каждом запросе.

Создание сервера

- Например, изменим файл *app.js* следующим образом:

```
const http = require("http");
http.createServer(function(request, response){
    response.setHeader("UserId", 12);
    response.setHeader("Content-Type", "text/html; charset=utf-8;");
    response.write("<h2> hello world </h2>");
    response.end();
}).listen(3000);
```



Маршрутизация

- Node.js не имеет **встроенной** системы *маршрутизации адресов запросов к серверу*.
- Обычно она реализуется с помощью специальных фреймворков типа **Express**.

- Однако если необходимо разграничить простейшую обработку небольшого количества маршрутов, то вполне можно использовать для этого свойство *url* объекта **request**.

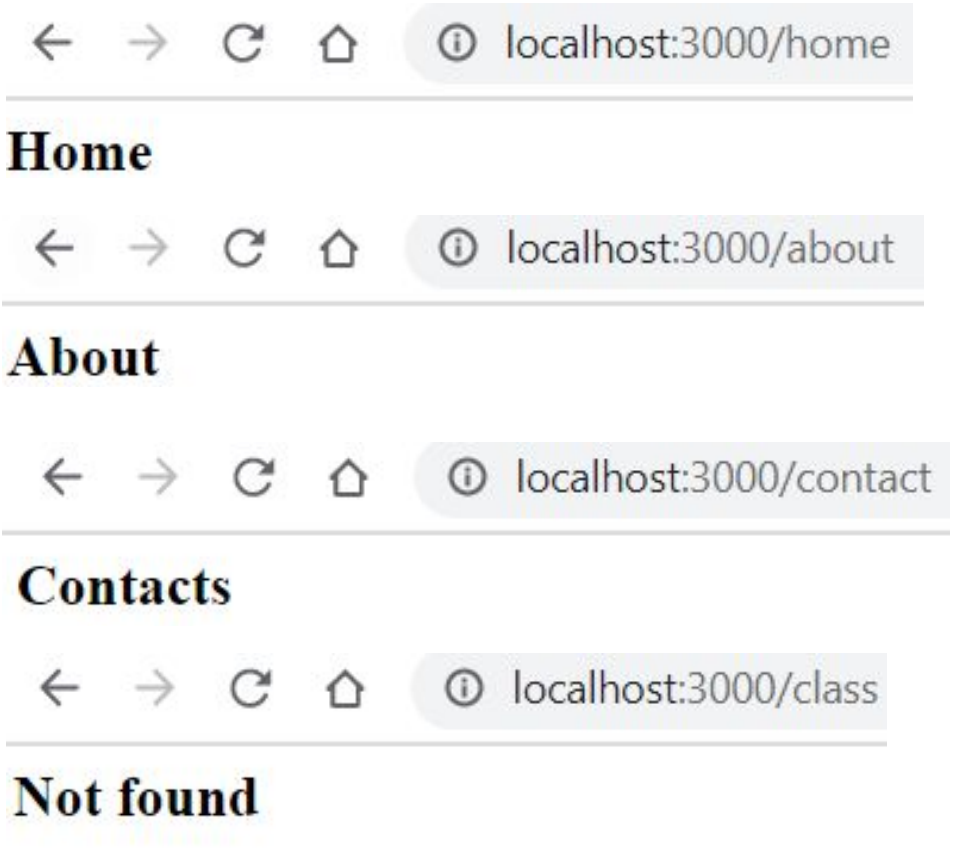
Маршрутизация

- Например:

```
const http = require("http");
http.createServer(function(request, response){
    response.setHeader("Content-Type", "text/html; charset=utf-8;");
    if(request.url === "/home" || request.url === "/"){ // маршрут 1
        response.write("<h2>Home</h2>");
    }
    else if(request.url == "/about"){ // маршрут 2
        response.write("<h2>About</h2>");
    }
    else if(request.url == "/contact"){ // маршрут 3
        response.write("<h2>Contacts</h2>");
    }
    else{
        response.write("<h2>Not found</h2>"); // маршрут не обнаружен
    }
    response.end();
}).listen(3000);
```

Маршрутизация

- В данном случае обрабатываются три маршрута и неправильный ввод маршрута.

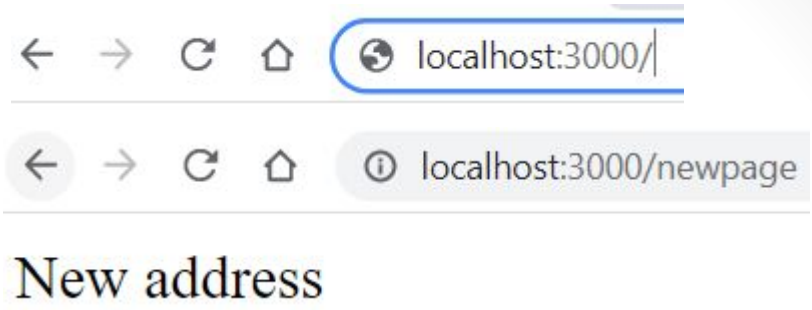


Переадресация

- Переадресация предполагает отправку **статусного кода**:
  - 301 (*постоянная переадресация*)
  - 302 (*временная переадресация*)
- А также заголовка **Location**, который указывает на новый адрес.

Переадресация

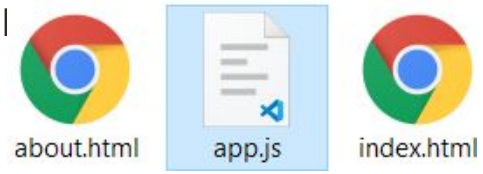
- Например:




```
const http = require("http");
http.createServer(function(request, response){
    response.setHeader("Content-Type", "text/html; charset=utf-8;");
    if(request.url === "/"){
        response.statusCode = 302; // временная переадресация
        response.setHeader("Location", "/newpage");
    }
    else if(request.url == "/newpage"){
        response.write("New address");
    }
    else{
        response.statusCode = 404; // адрес не найден
        response.write("Not Found");
    }
    response.end();
}).listen(3000);
```

Отправка файлов

- Отправка статических файлов – частая задача в работе веб-приложения.
- Пусть в каталоге проекта у нас будут три фа



index.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Главная</title>
    <meta charset="utf-8" />
</head>
<body>
    <h1>Главная</h1>
</body>
<html>
```

about.html

```
<!DOCTYPE html>
<html>
<head>
    <title>О сайте</title>
    <meta charset="utf-8" />
</head>
<body>
    <h1>О сайте</h1>
</body>
<html>
```

- Наша задача будет заключаться в том, чтобы отправить их содержимое пользователю.

Первый способ отправки файлов

- Для этого может применяться метод **fs.createReadStream**().
- Затем с помощью метода **pipe()** мы можем связать считанные файлы с потоком записи, то есть объектом *response*.

Первый способ отправки файлов

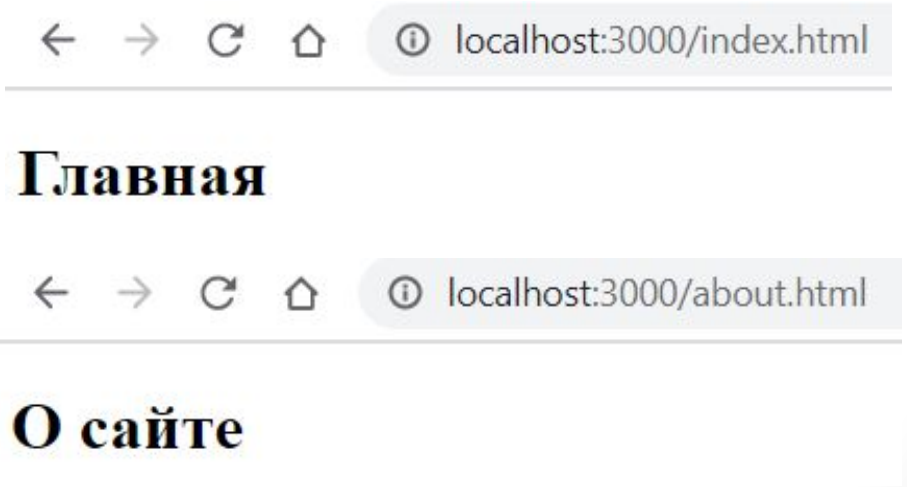
- Файл app.js :

```
const http = require("http");
const fs = require("fs");
http.createServer(function(request, response){
    console.log(`Запрошенный адрес: ${request.url}`);
    const filePath = request.url.substr(1); // получаем путь после слеша
    // смотрим, есть ли такой файл
    fs.access(filePath, fs.constants.R_OK, err => {
        if(err){ // если произошла ошибка, отправляем статусный код 404
            response.statusCode = 404;
            response.end("Resourse not found!");
        }
        else{ // если нет, связываем поток файла и ответа
            fs.createReadStream(filePath).pipe(response);
        }
      });
}).listen(3000, function(){
    console.log("Server started at 3000");
});
```

Первый способ отправки файлов

- Метод *fs.createReadStream()* создает поток для чтения – объект *fs.ReadStream*.
- Для получения данных из потока вызывается метод **pipe**(), в который передается объект интерфейса *stream.Writable* или поток для записи (объект *http.ServerResponse*).
- Запустим приложение и в браузере обратимся по

```
C:\app>node app
Server started at 3000
Запрошенный адрес: /
Запрошенный адрес: /index.html
Запрошенный адрес: /about.html
```



Первый способ отправки файлов

- В данном случае отправляются файлы *html*, но подобным образом можно отправлять разные файлы.
- Например, создадим новый файл *styles.css* со следующим содержимым:

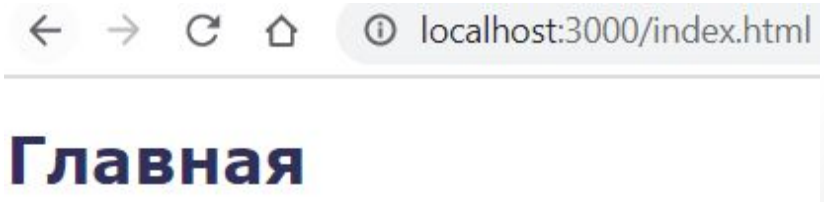
```
body{
    font-family: Verdana;
    color:rgb(48, 48, 92);
}
```

- Добавим стили на станице *index.html* в <*head*>:

```
<link href="public/styles.css" rel="stylesheet" type="text/css">
```

- И затем обратимся к *index.html*:

```
Server started at 3000
Запрошенный адрес: /index.html
Запрошенный адрес: /public/styles.css
```



Второй способ отправки файлов

- Второй способ представляет чтение данных с помощью функции **fs.readFile()** и отправка с помощью метода **response.end()**:

```
const http = require("http");
const fs = require("fs");
http.createServer(function(request, response){
    console.log(`Запрошенный адрес: ${request.url}`);
    const filePath = request.url.substr(1); // получаем путь после слеша
    fs.readFile(filePath, function(error, data){
        if(error){
            response.statusCode = 404;
            response.end("Resourse not found!");
        }
        else{
            response.end(data);
        }
    });
}).listen(3000, function(){
    console.log("Server started at 3000");
});
```

Шаблоны

- Вместо статичного содержимого можно применять **шаблоны**, вместо которых в файл будет вставляться какой-то определенный текст.
- Например, изменим файл *index.html* следующим образом:

```
<!DOCTYPE html>
<html>
<head>
    <title>Главная</title>
    <meta charset="utf-8" />
</head>
<body>
    <h1>{header}</h1>
    <p>{message}</p>
</body>
<html>
```

- Вместо конкретного содержимого здесь определены прейсхолдеры *{**header**}* и *{**message**}*, вместо которых может вставляться любой текст.

Шаблоны

- Изменим файл *app.js*:

```
const http = require("http");
const fs = require("fs");
http.createServer(function(request, response){
    fs.readFile("index.html", "utf8", function(error, data){
        let message = "Изучаем Node.js";
        let header = "Главная страница";
        data = data.toString(); // преобразовывает в строку
        data = data.replace("{header}", header)
                   .replace("{message}", message);
        response.end(data);
    })
}).listen(3000);
```

Шаблоны

- Здесь получаем содержимое файла и проводим его дополнительную обработку, заменяя плейсхолдеры на конкретный текст с помощью метода **data.replace**().
- Перед этим преобразовываем **data** в строку.

- При обращении к приложению мы получим полноценную *html*-страницу: