

# Разработка клиентских сценариев с использованием JavaScript и библиотеки jQuery

# Unit 8

JSON, Ajax

# Contents

Формат JSON .....	4
Синтаксис JSON .....	6
Объект JSON .....	9
Функция stringify() .....	9
Функция parse() .....	17
Ошибки при использовании JSON .....	21
Метод toJSON() .....	22
Синхронные и асинхронные запросы .....	26
Ajax .....	27
Объект XMLHttpRequest .....	29
События onload, onloadend, onerror .....	41
Запросы на удаленный сервер .....	44
Методы GET и POST .....	46
GET .....	48
POST .....	49
Объект FormData .....	53
Домашнее задание .....	56
Задание 1 .....	56
Задание 2 .....	58

# Формат JSON

Мы знаем, что в JavaScript мы можем производить манипуляции над объектами, создавать, изменять, передавать в функции, взаимодействовать с объектом в пределах программы. Однако, что если нам необходимо передать эти объекты по сети на серверную часть программы или получить их оттуда? Для этого нужно преобразовать объекты в текст, а потом этот текст отправить по сети. Чтобы это сделать, используется текстовый формат JSON.

**JSON** (*JavaScript Object Notation*) — текстовый формат для хранения и передачи данных по сети. JSON — это по сути представление объектов JavaScript в текстовом формате.

JSON был основан на JavaScript и введен в стандарте ECMA-262 в 1999 году. Его начали активно использовать вместе с технологией AJAX, вытеснив ранее использованный XML формат.

Далее показано как выглядит объект JavaScript и представление этого объекта в JSON.

Объект человека в JavaScript:

```
let person = {  
  firstName: "Andrey",  
  lastName: "Ivanov",  
  birthDate : "04.05.2000"  
}
```

А это представление объекта в JSON формате:

```
"{"  
  "firstName" : "Andrey",  
  "lastName" : "Ivanov",  
  "birthDate" : "04.05.2000"  
}"
```

Очень похоже, правда?

Преимуществом JSON является то, что он компактнее по сравнению с XML, соответственно меньше весит, а значит быстрее передается по сети.

Для передачи файла JSON используются AJAX-запросы. Если говорить вкратце, AJAX это технология, которая позволяет отправлять и принимать данные с сервера для дальнейшей работы с ними. Далее AJAX будет рассмотрен подробнее в этом уроке.

Стоит отметить, что хоть формат JSON и разработан на основе JavaScript, он используется не только в нем, но и во многих других языках программирования.

Сам формат универсален — он позволяет преобразовать объект в строку, а любой язык, который хочет работать с ним, решает, как преобразовать строку JSON обратно в объект на основании особенностей языка.

# Синтаксис JSON

Синтаксически JSON очень похож на объекты в JavaScript. Данные в нем представляются в виде пар ключ-значение в фигурных скобках.

```
{  
  "firstName" : "Andrey",  
  "lastName" : "Ivanov"  
}
```

Пары ключ-значение разделены двоеточием. Ключи всегда должны быть заключены в двойные кавычки, а одинарные или обратные кавычки приводят к ошибке.

Значения могут быть:

- Строкой, обязательно в двойных скобках;
- Числом;
- Логическим значением;
- Объектом, заключенным в { };
- Массивом, заключенным в [ ];
- `null`.

С примитивными значениями все просто:

```
{  
  "firstName" : "Andrey",  
  "lastName" : "Ivanov",  
  "age" : 20,  
  "isStudent" : true  
}
```

Здесь мы описываем студента, у которого есть имя, фамилия, возраст и является ли он студентом сейчас.

А вот как выглядят вложенные объекты. С ними все немного интереснее:

```
{
  "firstName" : "Andrey",
  "lastName" : "Ivanov",
  "age" : 20,
  "isStudent" : true,
  "contactInfo" : {
    "phone" : "098-556-33-41",
    "email" : "AndreyIvanov@gmail.com"
  }
}
```

В примере выше у студента есть свой вложенный объект — контактная информация.

Объект записывается в фигурные скобки, после которых идут его свойства и значения.

У студента, также, может быть перечисление всех дисциплин, которые он посещает. Их можно записать с помощью массива.

```
{
  "firstName" : "Andrey",
  "lastName" : "Ivanov",
  "age" : 20,
  "isStudent" : true,
  "contactInfo" : {
    "phone" : "098-556-33-41",
    "email" : "AndreyIvanov@gmail.com"
  },
  "disciplines" : [
    "Programming", "Machine engineering", "English"
  ]
}
```

Массивы заключаются в квадратные скобки, в которых перечисляются значения.

Так как JSON является универсальным форматом для передачи данных, то некоторые специфические значения в JavaScript не поддерживаются, а именно:

- Методы объекта;
- Свойства объекта, которые являются символьным типом (*Symbol*);
- Свойства объекта, которые являются *undefined*.



# Объект JSON

Для работы с JSON форматом в JavaScript есть объект **JSON**. Он предоставляет методы для конвертации JSON-строки в объект и наоборот. Кроме того, можно преобразовывать не только объекты, но и примитивные значения и массивы.

Объект JSON предоставляет две функции:

- **stringify()** — преобразует объекты в JSON;
- **parse()** — преобразует JSON в объект.

## Функция **stringify()**

Функция **stringify()** сериализует объекты в строку.

**Сериализация** — процесс преобразования объектов в определенный формат для сохранения данных или их передачи по сети с возможной последующей десериализацией.

**Десериализация** — процесс восстановления объекта из строки JSON.

Синтаксис функции **stringify()**:

```
JSON.stringify(value [, replacer[, space]])
```

Функция принимает объект, который нужно сериализовать в строку и два необязательных параметра (о них будет рассказано далее) и возвращает новую строку как результат выполнения.

Рассмотрим пример:

```
let person = {  
  firstName: "Andrey",
```

```

    lastName: "Ivanov",
    age: 20,
    isStudent: true,
    contactInfo: {
        "phone": "098-556-33-41",
        "email": "AndreyIvanov@gmail.com"
    },
    disciplines: [
        "Programming", "Machine engineering", "English"
    ]
}

let jsonPerson = JSON.stringify(person);
alert(jsonPerson);

```

Есть объект `person`, описывающий человека, далее он передается в функцию `JSON.stringify()`, которая возвращает строковое представление объекта.

На экране появится окно с таким содержанием:

```

{
  "firstName": "Andrey",
  "lastName": "Ivanov",
  "age": 20,
  "isStudent": true,
  "contactInfo": {
    "phone": "098-556-33-41",
    "email": "AndreyIvanov@gmail.com"
  },
  "disciplines": [
    "Programming",
    "Machine engineering",
    "English"
  ]
}

```

*Рисунок 1*

Стоит отметить, что порядок свойств объекта может отличаться от порядка в полученной строке.

Функции, значения `Symbol` или `undefined` не сериализуются, а будут просто пропущены.

Как видно в этом примере:

```
let badExample = {
  [Symbol("id")]: 1,
  property: undefined,
  Foo() {
    console.log("Hi!");
  }
}

let emptyStr = JSON.stringify(badExample);
alert(emptyStr);
```

В таком случае значение переменной `emptyStr` будет равно «`{ }`» (рис. 2).

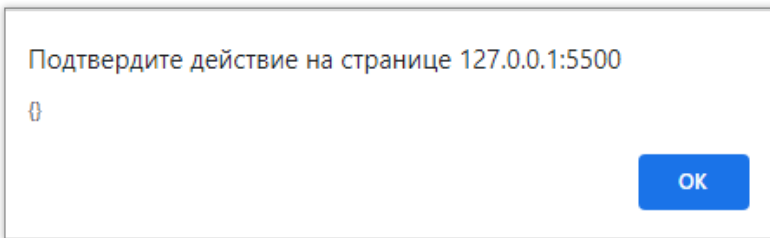


Рисунок 2

Важно знать, что в сериализованном объекте не должно быть циклических ссылок.

**Циклическая ссылка** — это ссылка, которая ссылается на другой объект, который в свою очередь ссылается на первый.

Давайте рассмотрим пример:

```
let Kate = {
  name : "Kate"
}

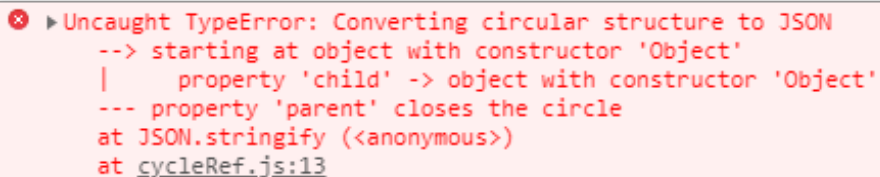
let Dev = {
  name: "Dev"
}

Kate.parent = Dev;
Dev.child = Kate;

let family = JSON.stringify(Dev);
console.log(family);
```

Здесь есть ссылка `Kate.parent` на объект `Dev`, а в `Dev.child` ссылка на `Kate`.

Если попробовать запустить этот пример, получим такую ошибку (рис. 3):



```
✖ ▶ Uncaught TypeError: Converting circular structure to JSON
    --> starting at object with constructor 'Object'
    |   property 'child' -> object with constructor 'Object'
    --- property 'parent' closes the circle
    at JSON.stringify (<anonymous>)
    at cycleRef.js:13
```

Рисунок 3

Функция `stringify()` может передать еще два необязательных параметра: `replacer` и `space`.

**Replacer** — параметр, который позволяет влиять на сериализацию объекта. Значением параметра может быть функция, массив или `null`, если параметр не нужен.

Функция в параметре `replace` используется, если нам необходимо определить значения, которые не будут включаться в сериализацию по определенному условию. Функция вызывается для каждого значения и может принимать два параметра: ключ и значение, которые и будут проверены, и, если они проходят проверку, то возвращается проверяемое значение, если нет, то `undefined`.

Рассмотрим на примере:

```
let person = {
  firstName: "Andrey",
  lastName: "Ivanov",
  age: 20,
  isStudent: true,
  contactInfo: {
    "phone": "098-556-33-41",
    "email": "AndreyIvanov@gmail.com"
  },
  disciplines: [
    "Programming", "Machine engineering", "English"
  ]
}

function checkAge(key, value) {
  if (key === "age" && value >= 18) {
    return undefined;
  }
  return value;
}

let jsonPerson2 = JSON.stringify(person, checkAge);
alert(jsonPerson2);
```

В примере объявлена функция `checkAge(key, value)`, которая будет проверять возраст. Если он выше требу-

емого, то будет возвращаться `undefined`, и это значение не будет сериализовано, если нет, вернется проверяемое значение, и оно будет включено в сериализацию.

На экране будет появляться такое значение `json-Person2`:

```
{
  "firstName": "Andrey",
  "lastName": "Ivanov",
  "isStudent": true,
  "contactInfo": {
    "phone": "098-556-33-41",
    "email": "AndreyIvanov@gmail.com"
  },
  "disciplines": [
    "Programming",
    "Machine engineering",
    "English"
  ]
}
```

Рисунок 4

Как мы видим, в строке отсутствует свойство `age`, так как оно не удовлетворяет условию в функции `checkAge()`. Если вторым аргументом в функцию `JSON.stringify(value, replacer)` передать массив, то значения массива будут сопоставлены со свойствами передаваемого объекта и если они совпадут то будут включены в преобразование, если нет, то пропущены.

```
let jsonPerson3 = JSON.stringify(person, ["firstName",
                                          "lastName"]);
alert(jsonPerson3);
```

Ожидается, что в `jsonPerson3` будут только те значения, которые есть в передаваемом массиве:

```
{
  "firstName": "Andrey",
  "lastName": "Ivanov"
}
```

Рисунок 5

И третий параметр в функции `JSON.stringify(value, replacer, space)` — это `space`, который принимает строку или число и позволяет придать более читаемый вид строке JSON, добавляя отступы.

Ниже показано как будет выглядеть JSON, если не передавать в функцию `stringify(person, null)` третьим параметром строку или число:

```
let person = {
  firstName: "Andrey",
  lastName: "Ivanov",
  age: 20,
  isStudent: true,
  contactInfo: {
    "phone": "098-556-33-41",
    "email": "AndreyIvanov@gmail.com"
  },
  disciplines: [
    "Programming", "Machine engineering", "English"
  ]
}

alert(JSON.stringify(person, null, 2));
```

```
{"firstName":"Andrey","lastName":"Ivanov","age":20,"isStudent":true,"contactInfo":{
  "phone":"098-556-33-41","email":"AndreyIvanov@gmail.com"},"disciplines":
  ["Programming","Machine engineering","English"]}
```

Рисунок 6

Строка (рис. 6) нечитабельна и непонятна.

В этом примере все точно так же, за исключением параметра `space` — `stringify(person, null, 2)`.

```
let person = {
  firstName: "Andrey",
  lastName: "Ivanov",
  age: 20,
  isStudent: true,
  contactInfo: {
    "phone": "098-556-33-41",
    "email": "AndreyIvanov@gmail.com"
  },
  disciplines: [
    "Programming", "Machine engineering", "English"
  ]
}

alert(JSON.stringify(person, null, 2));
```

Вот так будет выглядеть строка JSON (рис. 7):

```
{
  "firstName": "Andrey",
  "lastName": "Ivanov",
  "age": 20,
  "isStudent": true,
  "contactInfo": {
    "phone": "098-556-33-41",
    "email": "AndreyIvanov@gmail.com"
  },
  "disciplines": [
    "Programming",
    "Machine engineering",
    "English"
  ]
}
```

Рисунок 7



Как видно, при формировании строки JSON добавились отступы.

## Функция `parse()`

Чтобы строку JSON преобразовать в объект, нужно выполнить над ней метод парсинга. Парсинг, в данном случае, означает процесс десериализации строки в объект.

Функция `parse()` — функция, которая десериализует JSON строку и возвращает объект JavaScript.

Синтаксис:

```
JSON.parse(str, [reviver])
```

принимает строку с текстом, которая будет парситься, и возвращает готовый объект.

Здесь показана работа `JSON.parse()`:

```
let personStr = '{
  "firstName": "Andrey",
  "lastName": "Ivanov",
  "age": 20,
  "isStudent": true,
  "contactInfo": {
    "phone": "098-556-33-41",
    "email": "AndreyIvanov@gmail.com"
  },

  "disciplines": [
    "Programming",
    "Machine engineering",
    "English"
  ]
}'
```

```
let person = JSON.parse(personStr);  
  
alert(person.firstName)  
alert(person.contactInfo.phone);
```

В примере объявлена переменная `personStr`, содержащая строку JSON. После передаем ее в функцию `JSON.parse()`, которая создает и возвращает объект с данными, путем парсинга переданной строки.

Мы видим, что `person` является объектом и хранит те данные, которые были в строке `personStr`. Теперь мы можем взаимодействовать с ним как со всеми объектами JavaScript.

В нашем примере мы обращаемся к `firstName` и `contactInfo.phone` нашего объекта. На экране появятся окошки со значением «`Andrey`» и «`098-556-33-41`» соответственно (рис. 8).

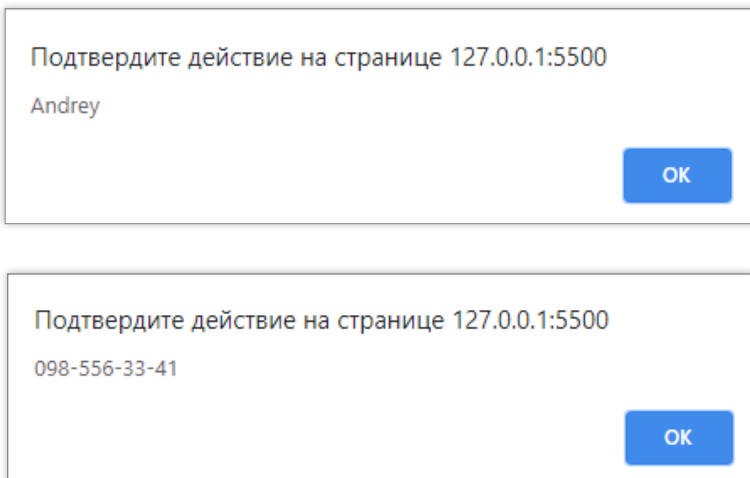


Рисунок 8

Также есть еще необязательный параметр `reviver`, с ним все практически то же, что и в функции `stringify()`, за исключением некоторого отличия. Это отличие заключается в том, что вторым параметром может быть только функция, которая будет отвечать за обработку значений. В этой функции можно определять значения, которые не будут присутствовать в новом объекте.

Пример:

```
personStr = '{
  "firstName": "Andrey",
  "lastName": "Ivanov",
  "age": 20,
  "isStudent": true,
  "contactInfo": {
    "phone": "098-556-33-41",
    "email": "AndreyIvanov@gmail.com"
  },
  "disciplines": [
    "Programming",
    "Machine engineering",
    "English"
  ]
}'

function CheckIsStudent(key, value) {
  if (key === "isStudent" && value == true) {
    return undefined;
  }
  return value;
}

let person2 = JSON.parse(personStr, CheckIsStudent);
alert(person2.isStudent);
```

В примере показана функция `CheckIsStudent`, которая проверяет, студент ли этот человек, и, если да, исключает это свойство. И при попытке обратиться к `person2.isStudent`, нам вернется значение `undefined` (рис. 9).

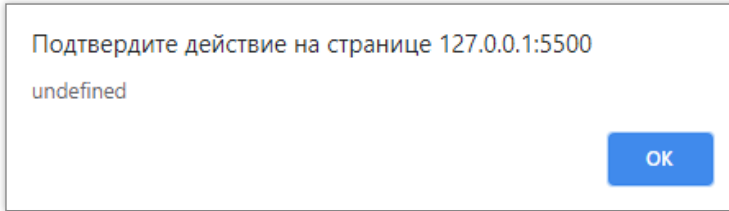


Рисунок 9

# Ошибки при использовании JSON

При написании JSON ошибки появятся в таких ситуациях:

- Когда имя свойства введено без кавычек `{property:"value"}`

✖ ▶ Uncaught SyntaxError: Unexpected token p in JSON at position 1 at JSON.parse (<anonymous>)

Рисунок 10

- Когда значение у которого должны быть кавычки, написано без них `{"property": value}`.

✖ ▶ Uncaught SyntaxError: Unexpected token v in JSON at position 14 at JSON.parse (<anonymous>)

Рисунок 11

- Когда используются одинарные или обратные кавычки вместо двойных `{'property': 'value'}`.

✖ ▶ Uncaught SyntaxError: Unexpected token ' in JSON at position 1 at JSON.parse (<anonymous>)

Рисунок 12

- Когда используется ключевое слово `new` `{property : new Date(2019,7,15)}`.

✖ ▶ Uncaught SyntaxError: Unexpected token p in JSON at position 2 at JSON.parse (<anonymous>)

Рисунок 13

## Метод toJSON()

Метод `toJSON()` может быть методом любого объекта. Он позволяет определить собственное представление объекта в JSON. Таким образом, можно заменить стандартное поведение сериализации на собственное. Функция `stringify()` будет вызывать метод объекта `toJSON()` и использовать возвращаемое значение, вместо того чтобы сериализовать переданный объект.

Применяется, если объект JavaScript может содержать некорректные значения, которые нарушают сериализацию.

Давайте рассмотрим это на примере:

```
let model = {  
  name: "BMW",  
  autopilot : undefined,  
}  
  
let car = {  
  color: "Black",  
  date : new Date(2019, 7, 21),  
  model  
}
```

У объекта `car` присутствует вложенный объект `model`, который состоит из названия модели и системы автопилота. Возможна ситуация, когда неизвестно, есть ли у машины автопилот. Это приведет к тому, что при сериализации свойство объекта `autopilot` не сериализуется (рис. 14).

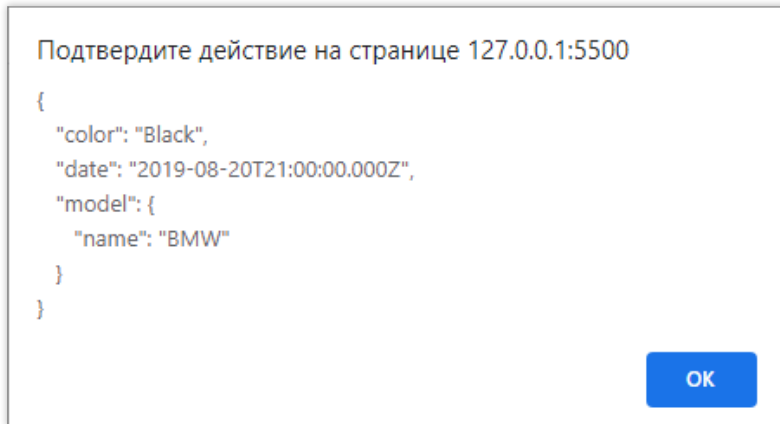


Рисунок 14

Чтобы это исправить, можно при определении метода `toJSON()` самостоятельно задать строковое представление объекта, и всем свойствам со значением `undefined` присвоить подходящее для сериализации значение.

Вот как будет это выглядеть:

```
let model = {
  name: "BMW",
  autopilot : undefined,
  toJSON() {
    let jsonStr = '{"name": "${this.name}",
                  "autopilot": '
    if(this.autopilot === undefined){
      jsonStr += '"Not"'
    }
    else{
      jsonStr += '"${this.autopilot}"'
    }
    return jsonStr;
  }
}
```

```
let car = {  
  color: "Black",  
  date : new Date(2019, 7, 21),  
  model  
  
}  
  
let carJSON = JSON.stringify(car;  
alert(carJSON);
```

В объекте `model` объявляется метод `toJSON()`. В нем создается переменная `jsonStr`, которая будет составлять строку JSON и возвращаться из метода как строковое представление объекта. В методе `toJSON()` проверяется свойство `autopilot`, значение которого `undefined`, и заменяется на строку «Not», что позволит сериализовать объект правильно.

Теперь на экране мы увидим, что объект `car` принял такой вид:

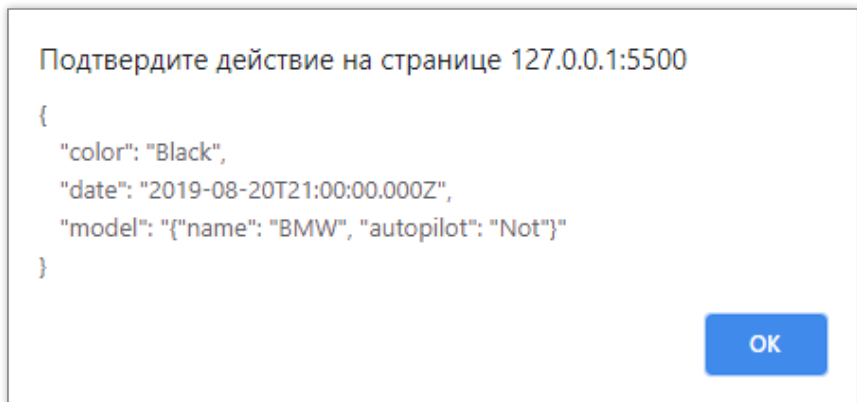


Рисунок 15



Теперь у объекта `model` свойство `autopilot` заменено на корректное. Также здесь можно заметить, что свойство `date` изменилось. Это из-за того, что объект `Date` имеет встроенный метод `toJSON()`, который возвращает подходящее для JSON значение даты.

# Синхронные и асинхронные запросы

Очень часто нам нужно получить или отправить какие-либо данные с сервера. Это делается с помощью сетевых запросов. Запросы можно разделить на синхронные и асинхронные.

**Синхронные запросы** — запросы, при отправке которых нужно дождаться ответа с сервера. Пока запрос обрабатывается на сервере, выполнение скрипта останавливается в ожидании ответа, и страница перестает откликаться на действие пользователя. Такие запросы применяются, когда нам необязательно, чтобы пользователь имел возможность взаимодействовать с интерфейсом страницы на время обработки запроса. В некоторых случаях запрос может быть отправлен синхронно, дабы не усложнять код. Например, при авторизации на сайте, пользователю не нужно выполнять другие действия на странице, и в этом случае можно использовать синхронный запрос.

**Асинхронные запросы** — запросы, которые позволяют не дожидаться ответа с сервера. Результат запроса будет обработан в момент, как только ответ будет принят с сервера. Скрипт продолжает работу, что позволяет пользователю продолжить работу на странице. Асинхронные запросы используют для работы с большим количеством данных, когда запрос будет долгим, и у пользователя должна быть возможность взаимодействовать со страницей.

# Ајах

Для отправки запросов используется технология Ајах.

**Ајах** — технология для взаимодействия с сервером без перезагрузки страницы.

Технология Ајах основана на JavaScript и XML и поддерживает асинхронность, что видно из ее аббревиатуры *Asynchronous JavaScript And XML*. В конце указан формат данных XML, но, несмотря на это, также можно использовать JSON или даже простой текст.

Ајах решает проблему с перезагрузкой страницы. Раньше каждый раз, когда нужно было отобразить новую информацию, страница перезагружалась, даже если изменения затронули небольшую ее часть. Например, в интернет магазине, для того чтобы отобразить еще некоторое количество товаров, перезагружалась вся страница. Из-за этого пользователь был вынужден ждать, пока запрос отправится на сервер, будет там принят, обработан и отправлен ответ, после чего сгенерируется страница с новыми данными. Все действия требовали времени, что влияло на отзывчивость страницы и в итоге повышало вероятность ухода пользователя со страницы.

С появлением Ајах эта проблема пропала, обновление стало затрагивать только нужную часть вместо всей страницы, таким образом сайты стали отзывчивее.

Запросы Ајах применяются в таких задачах, как:

- Отправка данных из формы;
- Для написания почтовой системы или чатов, чтобы принимать сообщения без перезагрузки страницы;

- Когда нужно подгружать данные постепенно, а не все сразу, например, загрузка товаров, комментариев, картинок;
- Поисковые системы используют автозаполнение поисковой строки, предлагая наиболее популярные запросы.

Перед тем как узнать, как использовать запросы Ajax, необходимо разобраться, куда они отправляются и откуда приходит ответ на них.

Выше уже было упомянуто такое понятие как сервер.

**Сервер** — это компьютер или программное обеспечение, которое обеспечивает взаимодействие пользователя и страницы сайта, принимает запросы, обрабатывает их, отправляет ответы на эти запросы, хранит данные, обеспечивает защиту этих данных и многое другое.

Для того чтобы понять, как работает сервер, приведем пример из жизни. Вы приходите в пекарню и просите продавца дать вам определенный хлеб, продавец находит запрашиваемый вами хлеб и отдает его вам. Вы, в данном случае, являетесь клиентом, требуемый хлеб — запросом, пекарь и пекарня — это, сервер который принял ваш запрос и отправил обратный ответ в виде хлебushка.

Сервер можно разделить на удаленный и локальный сервер.

**Удаленный сервер** — специальный мощный компьютер, предназначенный для больших нагрузок, он, как правило, находится в специализированном месте и доступ к нему осуществляется по сети.

URL-адрес выглядит примерно так: <https://www.google.com>.

**Локальный сервер** — может быть вашим собственным компьютером. Чтобы создать такой сервер нужно установить некоторое программное обеспечение, но об этом вы узнаете из курса PHP.

Локальные сервера, как правило, располагаются по адресу: <http://127.0.0.1:5500>.

**127.0.0.1** — это зарезервированное доменное имя для локальной сети. Такая запись эквивалентна **localhost**. Когда в адресе указывается **127.0.0.1** или **localhost**, это значит, что компьютер по сети обращается к самому себе.

В конце адреса указано число **5500** — это порт. Он нужен для идентификации приложения, к которому идет обращение. Ведь на сервере может располагаться несколько сайтов или программ, и, чтобы определить к кому именно был отправлен запрос, используются порты. У каждого приложения свой порт и это не обязательно 5500, порты могут быть в диапазоне от **0** до **65535**, главное, чтобы он не был занят другим приложением.

Почти все примеры, которые здесь приведены, были запущены на локальном сервере.

Давайте теперь рассмотрим, как работать с Ajax запросами.

## Объект XMLHttpRequest

Для использования Ajax в JavaScript есть специальный объект.

В старых версиях Internet Explorer (IE5 и IE6) по историческим причинам используется объект **ActiveXObject**.

```
let request = ActiveXObject("Microsoft.XMLHTTP");
```

Все современные браузеры имеет встроенный объект `XMLHttpRequest`:

```
let request = new XMLHttpRequest();
```

Ниже можно увидеть, как правильно создавать объект для запросов. Для начала стоит проверить, поддерживает ли браузер `XMLHttpRequest`, если да, тогда создать `XMLHttpRequest`, если нет — `ActiveXObject`.

```
let request;
if(window.XMLHttpRequest){
    request = new XMLHttpRequest();
}
else{
    request = new ActiveXObject("Microsoft.XMLHTTP");
}
```

Это кроссбраузерное решение, которое гарантирует, что запрос будет отправлен со всех браузеров.

Для того чтобы послать запрос, нужно использовать методы `open()` и `send()`.

Метод `open()` инициализирует запрос для дальнейшей отправки. Синтаксис:

```
open(method, URL, [async, user, password])
```

- `method` — метод HTTP запроса. Обычно это «GET» или «POST» которые будут описаны далее, также может быть «PUT», «DELETE», и т.д.
- `URL` — адрес сервера, куда будет отправлен запрос.

Когда в URL указывается просто путь к файлу, например: “\data\file.txt”, что равносильно записи “http://127.0.0.1:5500\data\file.txt”, а это значит что файл располагается локально, и запрос отправляется на локальный сервер.

Когда посылается запрос на удаленный сервер, в URL-адресе указывается тип протокола [http](#) или [https](#), домен сайта и адрес самого сервера. Например, <https://www.google.com>.

Далее идут необязательные параметры:

- **async** — указывает, будет ли запрос асинхронным или нет. По умолчанию значение этого параметра **true**, это значит, что запрос будет асинхронным.
- **user** и **password** — имя пользователя и пароль, если для запроса нужна аутентификация.

Метод [open\(\)](#) только инициализирует запрос а не отправляет его, как может сначала показаться. Чтобы отправить запрос, используется метод [send\(\)](#).

После того как запрос будет отправлен, на сервере произойдет его обработка, и после этого вернется ответ. Чтобы получить ответ с сервера, нужно подписаться на событие [onreadystatechange](#) у объекта запроса. В этом событии можно проверить состояние запроса через специальное свойство [readyState](#), которое содержит число, обозначающее, на какой стадии находится запрос. В таблице 1 показаны все значения [readyState](#) и их описание.

Событие [onreadystatechange](#) срабатывает каждый раз, когда [readyState](#) изменяется. Чтобы получить ответ, нужно дождаться, когда значение [readyState](#) будет равно 4. Ответ сервера на запрос можно получить через свойство [response](#).

Таблица 1

Значение readyState	Описание
0	Запрос не инициализирован
1	Запрос инициализирован
2	Запрос отправлен
3	Запрос обрабатывается на сервере
4	Запрос завершен, получен ответ с сервера

Для того чтобы все встало на свои места, давайте рассмотрим отправку запроса на примере:

```
let request;

if(window.XMLHttpRequest){
    request = new XMLHttpRequest();
}
else{
    request = new ActiveXObject("Microsoft.XMLHTTP");
}

request.open("GET", "text.txt");
request.onreadystatechange = function(){
    console.log("readyState = " + request.readyState);

    if(request.readyState === 4){
        alert(request.response);
    }
}

request.send();
```

Для начала проверяем, поддерживается ли объект **XMLHttpRequest** браузером, если да, создаем его, если нет, то создаем объект **ActiveXObject**.



После создания объекта нужно проинициализировать запрос с помощью метода `open()`. В `open()` передается строка со значением метода отправки, а также путь к файлу, содержащему данные, которые нужно получить. Запрос отправляется на локальный сервер, это видно по пути к файлу.

Далее подписываемся на событие `onreadystatechange`, чтобы получить пришедшие данные, в данном случае это анонимная функция. Каждый раз, когда будет изменяться свойство `readyState`, будет вызываться функция обработчик, в ней в консоль разработчика выводится текущее состояние `readyState`.



```
readyState = 2
readyState = 3
readyState = 4
```

Рисунок 16

Из скриншота (рис. 16) видно, какие этапы проходит запрос. Так как запрос уже был создан, инициализирован и отправлен, его состояние равняется `2` на момент, когда сработало событие `onreadystatechange` в первый раз. После отправки `onreadystatechange` срабатывает второй раз, и состояние запроса теперь `3`, а это значит, что запрос был принят на сервере и сейчас обрабатывается. И когда `onreadystatechange` вызывается в третий раз, `readyState` становится равным `4`.

Также нужно проверить свойство `status`, которое указывает, успешно ли завершился запрос. Если зна-

чение `status` равно `200`, то запрос завершился успешно, если какое-либо другое значение, то произошла ошибка. Например, значение `404` означает, что ресурс не найден.

И если свойства `readyState` равно `4`, а `status 200`, то информация выводится на экран.

Вот что появится на экране в случае успеха (рис. 17).

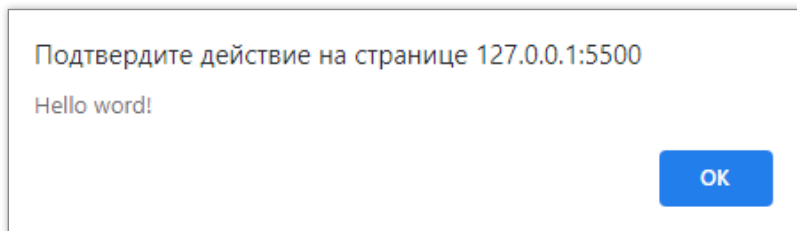


Рисунок 17

Объект `XMLHttpRequest` также имеет свойства:

- `statusText`, как и `status`, описывает, как завершился запрос, но в текстовом формате. Значение `200` в `statusText` будет означать «OK», значение `404` будет «Not Found».
- `responseType` — указывает, ответ какого типа нам пришел, также можно указать тип ожидаемого ответа.

Вот все возможные типы:

- «`""`» — строка, это значение по умолчанию;
- «`text`» — строка;
- «`arraybuffer`» и «`blob`» — для бинарных данных;
- «`document`» — XML-документ;
- «`json`» — JSON, значение будет автоматически десериализовано.

При использовании `responseType` нужно убедиться, что сервер вернет ответ нужного формата. Если ответ будет неподходящего формата, значение `response` будет равно `null`.

Пример, в котором `responseType` настроен на формат JSON:

```
let request;

if (window.XMLHttpRequest) {
    request = new XMLHttpRequest();
} else {
    request = new ActiveXObject("Microsoft.XMLHTTP");
}

request.open("GET", "person.json");
request.responseType = "json";
request.onreadystatechange = function() {
    if (request.readyState ===
        4 && request.status === 200) {
        let person = request.response;
        console.log(person);
    }
}

request.send();
```

В это примере для свойства `responseType` указывается тип «`json`», с этим значением строка JSON будет автоматически десериализована в объект. Далее отправляется запрос к файлу `person.json`. При получении ответа в свойстве `response` будет уже десериализованный из JSON объект. Это можно увидеть, открыв консоль разработчика (рис. 18).

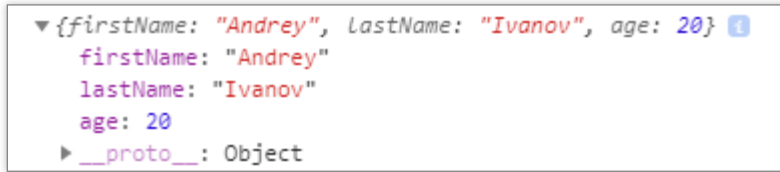


Рисунок 18

Следующий пример показывает, что будет, если формат ответа будет отличаться от указанного в **response-Type**:

```

let request;

if(window.XMLHttpRequest){
    request = new XMLHttpRequest();
}
else{
    request = new ActiveXObject("Microsoft.XMLHTTP");
}

request.open("GET", "text.txt");
request.responseType = "json";
request.onreadystatechange = function(){
    if(request.readyState ==
        4 && request.status == 200){
        alert(request.response);
    }
}

request.send();

```

Выше видно, что в ответе ожидаются данные в формате JSON, но фактически запрашиваемые данные в текстовом формате. В таком случае **response** будет равно **null**, и на экране появится сообщение:

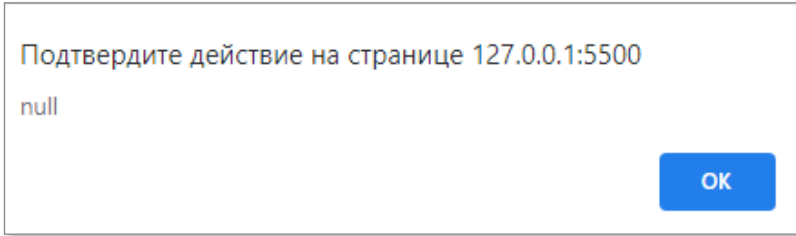


Рисунок 19

`responseText` — ответ будет в виде строки. Значением `responseType` должно быть либо значением по умолчанию, либо `"text"`, иначе будет ошибка.

В примере ниже показано, как использовать `responseText` с форматом JSON:

```
let request;

if(window.XMLHttpRequest){
    request = new XMLHttpRequest();
}
else{
    request = new ActiveXObject("Microsoft.XMLHTTP");
}

request.open("GET", "person.json");

request.onreadystatechange = function(){
    if(request.readyState ===
        4 && request.status == 200){
        alert(request.responseText);
    }
}

request.send();
```

Запрашивается файл с расширением JSON, и при этом `responseType` содержит значение по умолчанию, то есть в ответ придет строка с JSON. В результате `responseText` будет содержать JSON строку, которую в дальнейшем можно сериализовать в объект.

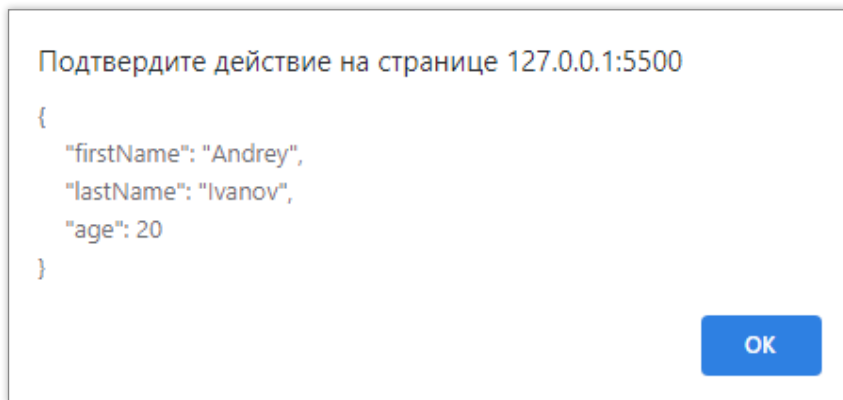


Рисунок 20

Здесь видно, что значением `responseText` является строка JSON.

`ResponseXML` — ответ будет в формате XML.

Вот как используется свойство `responseXML`:

```
let request;

if(window.XMLHttpRequest){
    request = new XMLHttpRequest();
}

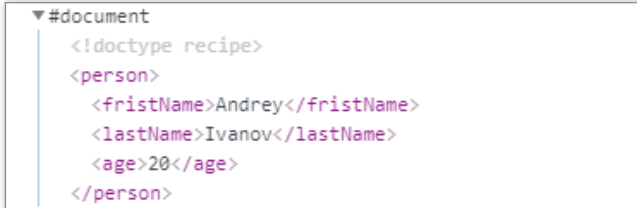
else{
    request = new ActiveXObject("Microsoft.XMLHTTP");
}
```

```

request.open("GET", "person.xml");
request.onreadystatechange = function(){
    if(request.readyState === 4 && request.status == 200){
        console.log(request.responseXML);
    }
}
request.send();

```

Здесь происходит запрос к файлу `person.xml`, свойство `responseType` имеет значение «`document`», так как оно определяется на основе полученного ответа, и мы явно не указали, какой формат ожидается. При получении ответа, к данным можно обращаться через свойство `responseXML`. В консоли разработчика будет такой результат:



```

▼ #document
  <!doctype recipe>
  <person>
    <fristName>Andrey</fristName>
    <lastName>Ivanov</lastName>
    <age>20</age>
  </person>

```

Рисунок 21

Здесь видно, что данные пришли в формате XML.

Рассмотрим ситуацию, когда формат ожидаемого ответа является некорректным:

```

let request;

if(window.XMLHttpRequest){
    request = new XMLHttpRequest();
}

```

```

else{
    request = new XMLHttpRequest("Microsoft.XMLHTTP");
}

request.responseType = "text";

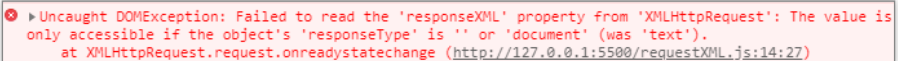
request.open("GET", "person.xml");
request.onreadystatechange = function(){
    if(request.readyState === 4 && request.status == 200){
        console.log(request.responseXML);
    }
}

request.send();

```

Выше можно увидеть, что `responseType` присваивается значение «`text`», но запрос к XML файлу. И при обращении к свойству `responseXML` произойдет ошибка, так как формат неподходящего типа.

Вот такую ошибку можно увидеть в консоли разработчика:



Uncaught DOMException: Failed to read the 'responseXML' property from 'XMLHttpRequest': The value is only accessible if the object's 'responseType' is '' or 'document' (was 'text').  
 at XMLHttpRequest.request.onreadystatechange (http://127.0.0.1:5500/requestXML.js:14:27)

Рисунок 22



## События onload, onloadend, onerror

Все примеры выше использовали событие `onreadystatechange` для получения результата, однако, есть и другой, более удобный способ это сделать.

Есть событие `onload`, которое будет вызываться только тогда, когда все данные будут загружены.

Вот как это выглядит на примере:

```
let request;

if(window.XMLHttpRequest) {
    request = new XMLHttpRequest();
}
else{
    request = new ActiveXObject("Microsoft.XMLHTTP");
}

request.open("GET", "person.json");

request.onload = function(){
    if(request.status === 200){
        alert(request.response);
    }
}

request.send();
```

Стоит подписаться на событие `onload`, и оно срабатывает только тогда, когда запрос завершился, таким образом мы избавляемся от проверки свойства `readyState`. На экране будет «*Hello world!*» (рис. 23).

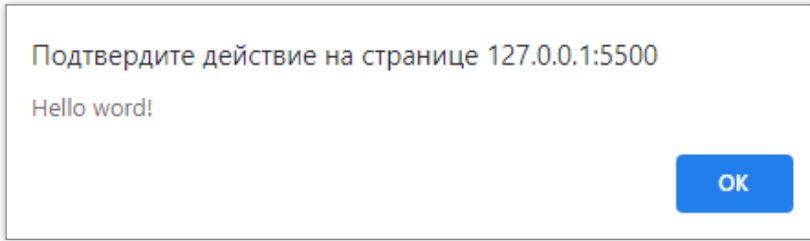


Рисунок 23

**XMLHttpRequest** имеет также и другие события:

- **onerror** — вызывается, когда запрос завершился некорректно.
- **onprogress** — периодически вызывается во время загрузки ответа. Можно использовать, чтобы выводить информацию о прогрессе загрузки.

Вот как будет выглядеть запрос, использующий эти события:

```
let request;
if(window.XMLHttpRequest){
    request = new XMLHttpRequest();
}
else{
    request = new ActiveXObject("Microsoft.XMLHTTP");
}

request.open("GET", "tex.txt");
request.onloadend = function(event){
    alert('Загружено ${event.loaded}');
}

request.onerror = function(){
    alert("Ошибка");
}
```

```
request.onload = function(){  
    if(request.status === 404){  
        alert(request.response);  
    }  
}  
  
request.send();
```

Однако, также нужно помнить, что не все старые браузеры поддерживают вышеперечисленные события, поэтому для полной надежности стоит использовать событие `onreadystatechange`, если вы хотите, чтобы AJAX запросы работали во всех браузерах.

## Запросы на удаленный сервер

Запросы на удаленный сервер отправляются по такому же принципу, как и на локальный.

Ниже приведен пример, где запрос будет отправлен на сервер, который хранит текущую погоду указанного города:

```
let request;
if(window.XMLHttpRequest){
    request = new XMLHttpRequest();
}
else{
    request = new ActiveXObject("Microsoft.XMLHTTP");
}

request.open("GET",
            "http://api.openweathermap.org/
            data/2.5/weather?q=Kiev&units=
            metric&APPID=b03a2c
            fad336d11bd9140ffd92074504");

request.onload = function(){
    if(request.status === 200){
        alert(request.response);
    }
}

request.send();
```

При инициализации запроса, в методе `open()` URL-адрес указан на удаленный сервер с погодой : «<http://api.openweathermap.org/data/2.5/weather?q=Kiev&APPID=b03a2cfad336d11bd9140ffd92074504>».

Из него видно, что обращение происходит на сервер, расположенный не локально, а по определенному адресу. Часть «<http://api.openweathermap.org/data/2.5/weather?>» это URL сервера, за ним идет параметры запроса «[?q=Kiev](#)» — это город, погода которого нужна, а «[APPID=b03a2cfad336d11bd9140ffd92074504](#)» ключ, необходимый для запроса. Его можно получить, зарегистрировавшись на сайте <https://openweathermap.org>. После обработки запроса придет такой ответ с погодой для Киева:

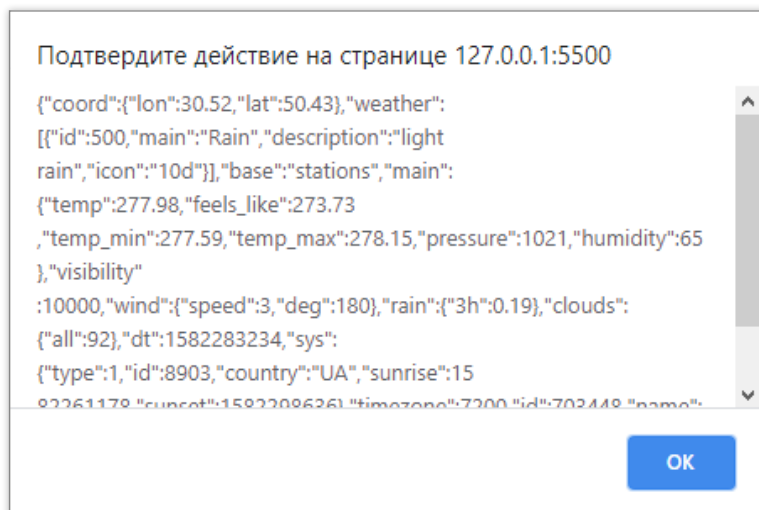


Рисунок 24

Пример выше использует специальное API, которое доступно по такому адресу <https://openweathermap.org/api>. На этом сайте есть документация для работы в нем.

# Методы GET и POST

Теперь пришло время разобраться, что такое HTTP-методы **GET** и **POST**. Только сначала давайте вкратце поговорим о HTTP.

**HTTP** — протокол, который используется для передачи данных по сети между клиентом и сервером. Клиентами могут быть разные устройства.

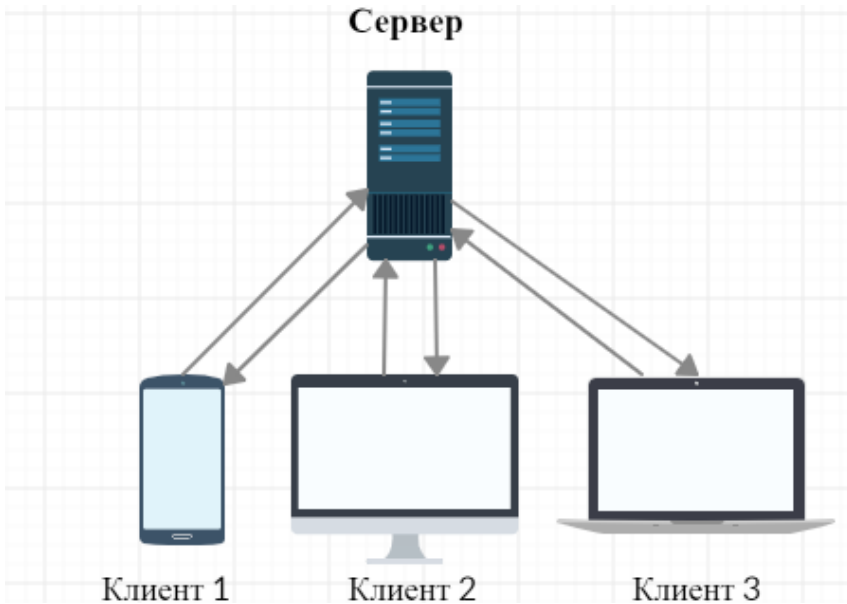


Рисунок 25

Передача данных происходит через запросы и ответы. Клиенты посылают запросы серверу, который, в свою очередь, обрабатывает их и посылает ответы клиентам, которые к нему обращаются.

Запрос, как правило, состоит из трех частей:

1. **Стартовая строка** — определяет в себе метод запроса, [url](#)-адрес ресурса, к которому отправляется запрос, и версию протокола.
2. **Заголовок** — хранит различную служебную информацию, например, кодировка сообщения, адрес, с которого был отправлен запрос, тип ожидаемого ответа и т.д.
3. **Тело** — передаваемые данные.

Вот как выглядит [http](#)-запрос при обращении на сайт:

```
GET /info.html HTTP/1.1
Host: mywebsite.ua
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/80.0.3987.122 Safari/537.36
Accept: text/html
```

Первой строкой идет стартовая строка, в ней указан метод [GET](#), данные, которые нужно получить, и версия протокола.

После идет набор заголовков, где [mywebsite.ua](#) — это хост, на котором расположен сайт, [User-Agent](#) — информация о клиенте, то есть о браузере и операционной системе, [Accept](#) — тип ожидаемого ответа от сервера. Это не все возможные заголовки, на самом деле, их намного больше, но в этом уроке перечислять их все мы не будем.

Теперь, когда понятно, что такое HTTP протокол можно подробнее поговорить про методы запроса.

Методы HTTP указывают, какие действия нужно выполнить с данными. Наиболее популярными являются [GET](#) и [POST](#).

## GET

Метод **GET** используют для получения данных с сервера. Раньше существовал только метод **GET** и использовался как для получения данных, так и для их отправки.

Передача данных осуществляется через адресную строку браузера. Для этого нужно вручную сформировать адресную строку, в которой будут все нужны данные. Вот пример того, как это выглядит:

```
http://url?параметр1=значение1&параметр2=значение2
```

Здесь видно, что **url** и передаваемые параметры разделяются символом '?', а параметры разделены между собой '&'.

Символы '?' и '&' зарезервированы и передавать их как значения параметров нельзя. Вместо этого нужно передавать не сам символ, а его кодовое значение, чтобы это сделать, нужно использовать символ '%', а потом кодовое значение нужного символа. Например, знак '?' будет иметь такой вид: %3F, а '&' будет %26.

**GET** метод имеет некоторые недостатки, а именно:

- Некоторые браузеры могут не корректно воспринимать кириллицу в адресной строке.
- Ограниченный размер передаваемых данных. Максимальный размер запроса **GET** — 1024 символа, а если размер запроса превышает максимум, то удаляется все, что идет после 1024 символа.

Из-за значительных недостатков метод **GET** используют только для получения данных с сервера.



## POST

Метод **POST** используется для отправки данных на сервер. В отличие от GET-запросов, данные POST-запросов передаются не в адресной строке браузера, а в теле самого запроса. Такие запросы позволяют передавать большое количество данных.

До этого момента все наши запросы отправлялись и запрашивали данные методом **GET**. Теперь рассмотрим, как отправить данные с помощью метода **POST**.

Здесь приведена **html** форма, в которой есть имя пользователя, его номер телефона и кнопка «Перезвоните мне.», при нажатии на которую пользователю должен перезвонить оператор. При нажатии на кнопку данные должны отправиться на сервер, и там происходит их обработка, в результате которой оператор получит задачу перезвонить этому пользователю.

```
<form id="form" action="">
  <input type="text" name="name" id="name-inp" >
  <input type="text" name="phone" id="phone-inp" >
  <input type="submit" id=""submit-btn">
</form>
```

Здесь код JavaScript, который отправляет данные. Ниже подробно описан алгоритм отправки.

```
<script>
let subbliBtn = document.getElementById("submit-btn");
subbliBtn.onclick = function () {
  let nameValue = document.getElementById("name-inp").
    value;
```

```

let phonValue = document.getElementById("phone-inp").
    value;
let data = "name=" +
    encodeURIComponent(nameValue) +
    "&phone=" +
    encodeURIComponent(phonValue);
let request;

if (window.XMLHttpRequest) {
    request = new XMLHttpRequest();
}
else {
    request = new ActiveXObject("Microsoft.XMLHTTP");
}

request.open("POST", "server.php");

request.onreadystatechange = function(){
    if (request.readyState ==
        4 && request.status == 200) {
        alert("Здравствуйте " +
            nameValue +
            "! Мы перезвоним вам через 1 минуту" );
    }
}

request.setRequestHeader('Content-Type',
    'application/x-www-form-urlencoded');
request.send(data);
}

</script>

```

Для начала находим кнопку по `id submit-btm`, при нажатии на которую введенные данные должны отправиться на сервер. Подписываемся на событие `onclick`, назначая ей обработчик. В этом обработчике формируем дан-

ные. Находим вводимые данные и формируем их в одну строку для отправки.

Здесь есть особенности: данные в строке представляются в виде пары ключ-значение, то есть сначала должно быть название параметра после это знак '=' и далее значение этого параметра.

Также каждое значение, введенное пользователем, должно быть преобразовано в строку в формате UTF-8. Это нужно дабы избежать проблем, связанных с зарезервированными символами. Для кодирования данных в JavaScript есть функция `encodeURIComponent()`.

И последнее, что нужно знать о формировании строки данных, это то, что каждую пару параметр-значение нужно разделять знаком '&'.

После этого создаем объект запроса, настраиваем методом `open()`, в котором нужно указать метод запроса, в нашем случае это метод `POST`, и путь к серверу, который будет принимать и обрабатывать запрос.

Далее подписываемся на `onreadystatechange`.

Обязательной частью `POST` запроса является установка заголовка `Content-Type`. Он необходим для того, чтобы указать, с помощью какой кодировки зашифрованы данные, чтобы сервер знал, как расшифровать пришедший запрос. Для этого используется метод `setRequestHeader()`. В нем нужно указать название заголовка, а именно `Content-Type`, и его значение «`application/x-www-form-urlencoded`».

Есть два стандарта кодирования данных: `multipart/form-data` и `text-plain`. В них данные передаются через строку-разделитель.

Отправляется запрос методом `send()` с параметром `data`, который является данными для отправки.

Сервер принимает запрос и выполняет нужные действия с пришедшими данными, например, добавляет их в базу данных. В нашем примере мы отправляли имя и номер телефона пользователя для того, чтобы ему позже перезвонил оператор. Чтобы пользователю было понятнее, что же произойдет дальше, выводится сообщение о том, что ему перезвонят через одну минуту. Это происходит в обработчике события `onreadystatechange`, когда запрос пришел без ошибок.

Этот пример работает на локальном сервере, при прохождении курса РНР вы научитесь создавать локальный сервер на своем компьютере.

Вот такое сообщение увидит пользователь, если запрос завершился без ошибок (рис. 26).

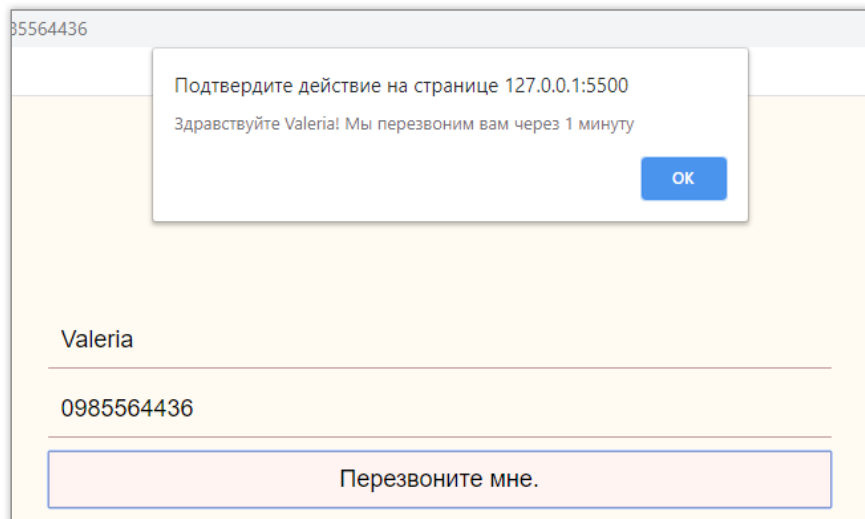


Рисунок 26

# Объект FormData

Для форматирования отправляемых данных в строку есть еще один, более удобный способ, а именно объект **FormData**.

**FormData** — позволяет формировать данные из форм в пары ключ-значение автоматически. Давайте рассмотрим тот же пример, что и выше, только с использованием объекта **FormData**:

```
let subbliBtn = document.getElementById("submit-btm");
subbliBtn.onclick = function () {
    let form = document.getElementById("form");
    let formData = new FormData(form)
    formData.append("date", new Date().toLocaleString());
    let request;

    if (window.XMLHttpRequest) {
        request = new XMLHttpRequest();
    }
    else {
        request = new ActiveXObject("Microsoft.XMLHTTP");
    }
    request.open("POST", "server.php");
    request.onreadystatechange = function () {
        if (request.readyState ==
            4 && request.status == 200) {
            alert("Здравствуйтесь " + nameValue +
                "! Мы перезвоним вам через 1 минуту");
        }
    }

    request.setRequestHeader('Content-Type',
                              'multipart/form-data');
    request.send(formData);
}
```

Этот пример очень похож на предыдущий, за исключением особенностей связанных с объектом `FormData`. В обработчике события `onclick` сначала создается объект `FormData`, в его конструктор передается DOM объект формы, на основе которой и будет сгенерирована строка с данными. Далее вызывается метод `append` в объекте `formData`, который выполняет добавление дополнительного параметра, указав ключ как «`date`» и само значение даты, например, для того чтобы оператор знал, когда была отправлена заявка на звонок.

Еще одно значительное изменение касается заголовка запроса, а именно при указании «`Content-Type`» значение должно быть «`multipart/form-data`», это нужно для того чтобы сервер правильно раскодировал данные. И в методе `open()` нужно передать объект `formData`.

Также `FormData` имеет такие методы:

- `set(key, value)` — добавляет новую пару ключ-значение, если такого ключа не существует, а если такой ключ существует, изменяет его значение;
- `append(key, value)` — добавляет новую пару ключ-значение, если такой ключ существует, а ключ-значение добавляется в конец;
- `delete(key)` — удаляет данные по ключу;
- `get(key)` — возвращает данные по ключу;
- `has(key)` — проверяет, существуют ли данные с заданным ключом.

Последнее, о чем будет сказано, это как передавать данные в формате JSON.

Рассмотрим это на примере:

```
let person = {
  firstName: "Andrey",
  lastName: "Ivanov",
  age: 20
}

let request;

if (window.XMLHttpRequest) {
  request = new XMLHttpRequest();
}
else {
  request = new ActiveXObject("Microsoft.XMLHTTP");
}

let jsonPerson = JSON.stringify(person);

request.open("POST", "server.php");
request.setRequestHeader('Content-Type',
                        'application/json');
request.send(jsonPerson);
```

Например, есть объект «человек», нужно передать данные о нем по сети. Чтобы это сделать, объект нужно сериализовать, а также при указании заголовка запроса в методе `setRequestHeader()` значение для `Content-Type` нужно установить «`application/json`».

# Домашнее задание

## Задание 1

Необходимо создать сайт с прогнозом погоды на сегодня.

Чтобы получить прогноз погоды, воспользуйтесь сайтом <https://openweathermap.org>. Для начала нужно зарегистрироваться на сайте по ссылке [https://home.openweathermap.org/users/sign up](https://home.openweathermap.org/users/sign_up) и получить ключ для дальнейшей работы. На странице <https://openweathermap.org/current> есть подробная документация как работать с API.

Создайте html-страницу с текущей погодой. Пользователь вводит название города в поисковую строку и при нажатии на кнопку поиска отображается погода.

На странице должны быть два блока.

Первый блок — с текущей погодой. Он должен иметь такую информацию:

- Город;
- Дата;
- Описание погоды;
- Иконка;
- Текущая температура;
- Минимальная температура;
- Максимальная температура;
- Скорость ветра.



Второй блок должен отображать почасовую погоду и иметь такие данные:

- Время;
- Иконка;
- Описание погоды;
- Текущая температура;
- Скорость ветра.

Пример, как должен выглядеть сайт:

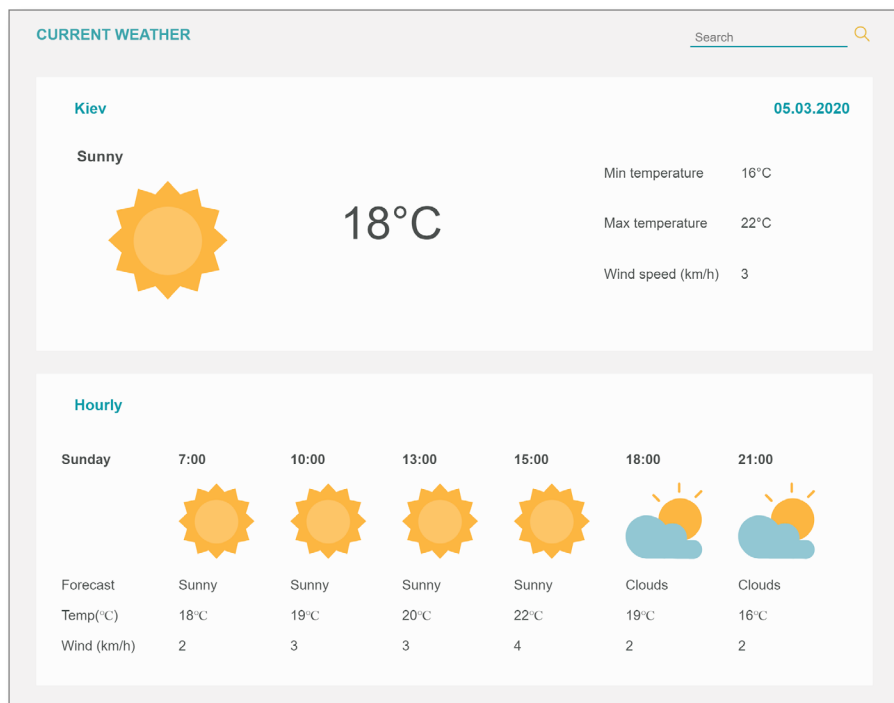
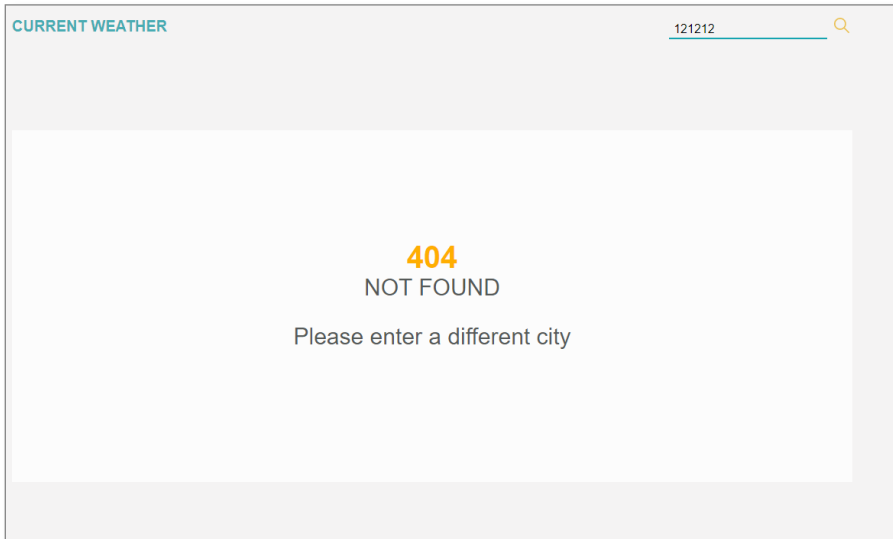


Рисунок 27

Если пользователь неверно ввел название города, то должно появиться такое уведомление:

*Рисунок 28*

## Задание 2

Необходимо создать сайт, на котором можно узнать информацию о пользователе GitHub. Для этого используйте GitHub API <https://api.github.com/users/>. Он возвращает информацию о пользователе по его логину. То есть чтобы получить нужные данные используйте такой запрос <https://api.github.com/users/userLogin>. `UserLogin` должен быть логином необходимого пользователя.

Должна быть возможность указать логин пользователя. При нажатии кнопки поиска должен выполняться запрос к API и отобразится такая информация на экране:


- Фото;
- Имя;
- Логин;

- Ссылка на GitHub пользователя;
- Ссылка на блог;
- Город;
- Почта;
- Количество подписчиков и подписок.

Если какой-то информации о пользователе нет, необходимо указать, что таких данных нет.

Пример страницы:

Search login



**Url to github:**
<https://github.com/mojombo>

**Blog:**
<http://tom.preston-werner.com>

**City:**

San Francisco

**Email:**

No Email

**Followers:** 21820
 **Following** 11

**Name:**

Tom Preston-Werner

**Login:**

mojombo

Рисунок 29