**Работа с API. Введение**

Всё это время вы работали с JavaScript так: вешали обработчики, меняли страницу при наступлении какого-то события. Все данные при этом вы заранее добавляли в файл script.js или в разметку. Так, вы с вашим кодом были заперты внутри браузера.

В этой теме вы сможете вырваться на свободу — начать взаимодействовать с сервером. Вы научитесь посылать на сервер запросы и получать от него ответы. Сможете обеспечить сайт приличным UX: показывать пользователю ошибку, если у него отвалился интернет.

Умение общаться с сервером откроет перед вами новые горизонты веб-разработки. В интернете есть множество API, от которых можно получать данные. В этой теме вы научитесь взаимодействовать и с ними.

Так что, вперёд — навстречу к знанию, которое сделает вас по-настоящему свободным веб-разработчиком.

# Протокол HTTP

Первые прототипы интернета появились ещё в 60-е годы. Но чтобы стать популярным, интернету понадобилось ещё 30 лет. Дело в том, что не было интерфейса: каких-то правил, как интернетом пользоваться.

В 1991 году в лаборатории ЦЕРН учёный-физик Тим Бернерс-Ли со своей командой размышляли, как сделать работу с интернетом простой. Они пытались придумать правила, по которым можно было бы организовать коммуникацию между компьютерами в сети. Так появился проект всемирной паутины.

Так что же нужно было придумать, чтобы интернет обрёл свой современный облик?

Поскольку интернет — сеть компьютеров, которые обмениваются данными, эти данные необходимо как-то оформить. Нужно придумать, как представлять информацию в интернете.

Но мало представить информацию, нужно ещё научить компьютеры её отображать. То есть нужно какое-то программное обеспечение, которое позволит прочитать информацию в интернете и показать её пользователю.

Ну и наконец: информацией нужно как-то обмениваться: нужны правила, по которым компьютеры будут отправлять и получать документы.

В итоге Тим Бернерс-Ли вместе с командой придумали:

1. язык, на котором можно создавать документы в интернете — HTML;
2. программу для просмотра этих документов — браузер;
3. единые правила, по которым документы передаются — транспортный протокол HTTP.

С первыми двумя пунктами вы уже знакомы. В этой теме расскажем о третьем — протоколе передачи данных HTTP.

# Что такое транспортный протокол

Как мы уже сказали, интернет — это сеть из компьютеров, которые обмениваются друг с другом информацией. Чтобы это было возможным, нужно установить какие-то правила: в каком виде данные будут ходить от одного компьютера к другому.

Обмен данными в интернете построен на принципе «запрос — ответ»: браузер формирует запрос, сервер его анализирует и отправляет ответ. Правила, по которым нужно формулировать запросы и ответы, определяет транспортный протокол — HTTP.

Проведём аналогию: если вам нужно отправить письмо по почте, вы должны положить его в конверт, написать адрес по определённым правилам, наклеить марку и отнести на почту. Компьютер с сервером тоже обмениваются сообщениями — «письмами». Правила оформления этих писем — и есть протокол HTTP.

# Когда браузер делает запросы

HTTP расшифровывается как Hypertext Transfer Protocol. Он так называется, потому что был придуман для передачи HTML-документов. Но сейчас этот протокол используют для передачи через интернет любых данных.

Когда вы заходите на сайт, браузер отправляет HTTP-запрос на сервер. Тот в свою очередь формирует ответ — HTML-код нужной страницы. Из кода страницы браузер построит DOM-дерево и начнёт отрисовывать сайт.

В коде веб-страницы могут быть ссылки на CSS и JavaScript-файлы, шрифты, изображения. Чтобы загрузить их, браузер сделает соответствующие HTTP-запросы к нужным ресурсам. Когда браузер получит ответы на эти запросы, он сможет отрисовать сайт полностью. Также на сайте может быть JavaScript-код, который тоже будет посылать HTTP-запросы.

Все запросы браузера можно поделить на 4 категории:

1. запросы HTML-документов;
2. запросы изображений, CSS, JS и других файлов;
3. отправка форм встроенным способом с перезагрузкой страницы;
4. запросы из JavaScript-кода.

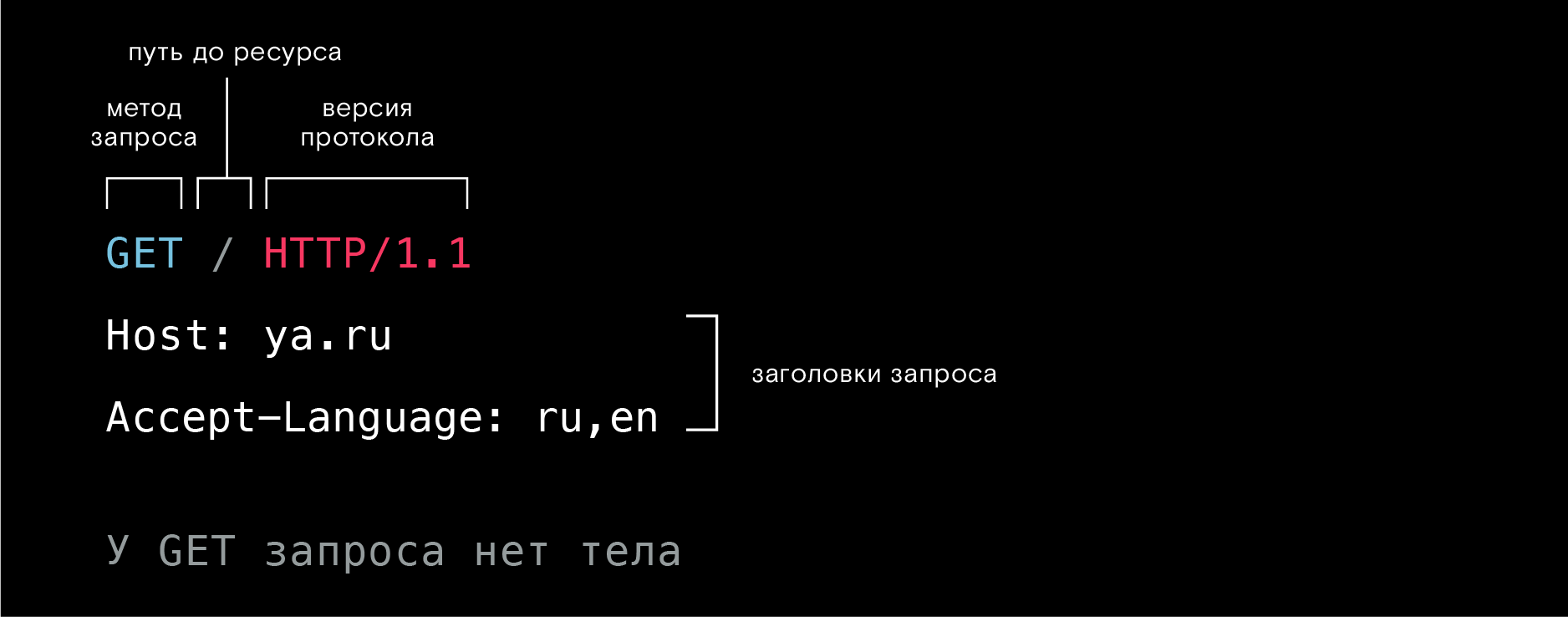
Запросы первых двух категорий браузер делает автоматически — вручную их формировать не нужно. С третьей категорией запросов мы познакомимся в восьмом уроке. А четвёртая категория — это запросы, которые делает разработчик из JavaScript кода. В этой теме мы будем работать в основном с ними.

# Формат запроса

Запрос, который формирует браузер, может включать в себя:

* HTTP-метод: он определяет операцию, которую нужно совершить. Есть несколько методов, самые популярные из них: GET и POST. Первый используют для того, чтобы получить данные с сервера, второй — чтобы их отправить.
* Путь до ресурса: это часть адреса без имени сайта. Например, в example.com/hello, путь — это /hello.
* Версию HTTP протокола, который используется для отправки запроса: например, HTTP/1.1.
* Заголовки запроса или headers: в них можно передать серверу дополнительную информацию.
* Тело запроса — есть не у всех запросов. Например, тело POST-запроса — это отправляемые данные.

Вот как выглядит запрос страницы ya.ru:

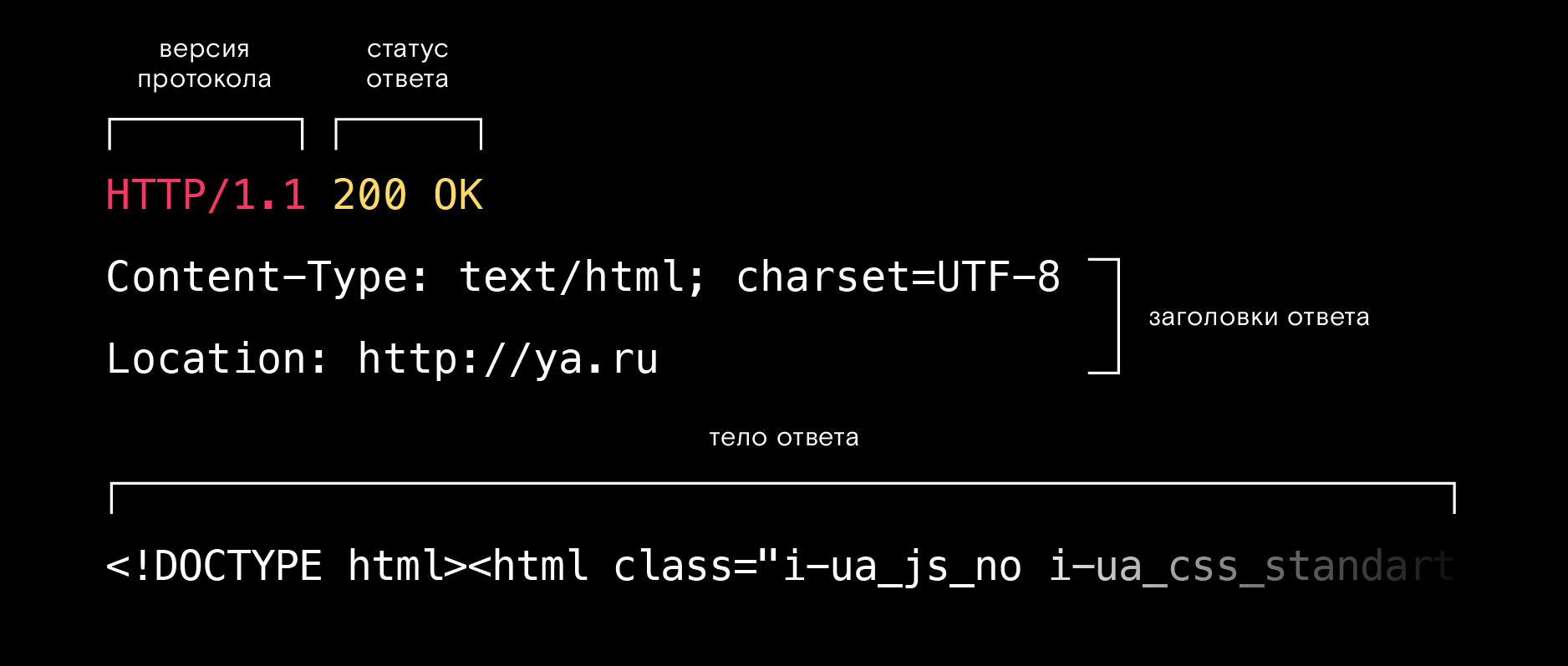


# Формат ответа

Ответ может включать:

* Версию HTTP протокола.
* Код и сообщение ответа. Например «200 OK», если всё в порядке, или «404 Not Found», если запрашиваемый ресурс не найден.
* Заголовки, содержащие дополнительную информацию для браузера.
* Тело ответа. Например, при заходе на сайт yandex.ru в теле ответа будет HTML-код этой страницы.

При запросе страницы ya.ru ответ выглядит так:



Таким образом, браузер общается с сервером через запросы. Некоторые из них происходят автоматически, другие — можно настроить самостоятельно. Чтобы сделать это, нужно понимать структуру HTTP-запросов. В следующем уроке расскажем, как генерировать запросы к серверу из JavaScript-кода.

# Запросы из JavaScript

В прошлом уроке мы говорили об HTTP-запросах. Раньше их можно было делать только автоматически: вы нажимаете на ссылку, браузер отправляет запрос страницы на сервер, получает ответ и отрисовывает сайт.

Со временем это стало неудобно. Например, вы хотите реализовать на сайте чат. При каждой отправке сообщения нужно отрисовывать сайт заново. К тому же, чтобы получить сообщения, придётся обновлять страницу вручную.

Короче, понадобилось отправлять запросы из кода. Для этого был разработан специальный API внутри браузера — fetch.

## Метод fetch

Метод fetch создаёт запрос на сервер и возвращает его ответ. На вход fetch принимает два аргумента. Первый — обязательный — URL запрашиваемого ресурса:

Скопировать кодJAVASCRIPT

fetch('https://example.com');

Будьте внимательны и не оставляйте пробелов в конце при указании URL-адреса.

Второй аргумент — необязательный. Это объект опций. Чаще всего нужны опции method, headers и body: они отвечают за метод запроса, его заголовки и тело:

Скопировать кодJAVASCRIPT

fetch('https://example.com/users', {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify({

username: 'ivan'

})

});

Метод fetch асинхронный. Когда вы его вызываете, он создаёт промис, а когда получает ответ — переводит промис в нужный статус. Ответ от сервера при этом записывается в промис, так что его можно использовать через then и catch:

Скопировать кодJAVASCRIPT

fetch('https://example.com')

.then((res) => {

console.log(res); *// если всё хорошо, получили ответ*

})

.catch((err) => {

console.log('Ошибка. Запрос не выполнен');

});

Если запрос выполнен успешно, сработает обработчик then. Если же что-то пошло не так, — например, отвалился интернет — сработает catch.

Вас мог напугать метод JSON.stringify или непонятное свойство headers. Если это так — не пугайтесь. Скоро ознакомитесь и с тем, и с тем.

# Формат JSON

В прошлом уроке мы рассказывали о методе fetch: он позволяет отправить запрос на сервер из JavaScript-кода. В этом уроке расскажем, в каком виде от сервера приходит ответ, и что с этим делать.

После того, как мы отправили запрос, его должен обработать сервер и вернуть нам ответ. Логика работы сервера может быть описана на языках Java, Python или C++, а не JavaScript. Так что сервер не может вернуть нам данные внутри массива или объекта — этих структур просто нет у сервера.

Поэтому для передачи структурированных данных были разработаны специальные форматы. Самый распространённый из них — JSON**.**

JSON расшифровывается как JavaScript Object Notation.

Вот как выглядят данные в формате JSON:

Скопировать кодJSON

[

{

"title": "Вектора",

"artist": "OZORA"

},

{

"title": "Страшно",

"artist": "Shortparis"

},

{

"title": "Ariadna",

"artist": "Kedr Livansky"

}

]

Поскольку данные записаны в строго определённой форме, их можно преобразовать в строку и передать в HTTP-запросе. В этом и заключается смысл формата JSON: он позволяет собирать данные в объект, а затем преобразовывать этот объект в строку для передачи в запросе. Затем эту строку превращают обратно в объект.

Обратите внимание, что ключи взяты в двойные кавычки — в JSON это обязательно. Значениями ключей могут быть строки, числа, булевы значения, null, объекты и массивы:

Скопировать кодJSON

{

"name": "Иван",

"age": 30,

"hasDog": true,

"hasCat": false,

"dog": {

"name": "Бонни",

"age": 6

},

"cat": null

}

На верхнем уровне всегда должен быть объект или массив. JSON не может содержать функций, переменных или комментариев.

## Методы JSON.stringify и JSON.parse

Как мы уже сказали, объект JSON нужно преобразовывать в строку и обратно. Для этого есть специальные методы: JSON.stringify и JSON.parse.

Метод JSON.stringify делает из объекта строку JSON:

Скопировать кодJAVASCRIPT

const songs = [

{

title: 'Вектора',

artist: 'OZORA'

},

{

title: 'Страшно',

artist: 'Shortparis'

},

{

title: 'Ariadna',

artist: 'Kedr Livansky'

}

];

const songsJSON = JSON.stringify(songs);

console.log(songsJSON);

console.log(typeof songsJSON); *// "string"*

Метод JSON.parse делает обратное — преобразовывает JSON-строку в объект JavaScript:

Скопировать кодJAVASCRIPT

const songs = [

{

title: 'Вектора',

artist: 'OZORA'

},

{

title: 'Страшно',

artist: 'Shortparis'

},

{

title: 'Ariadna',

artist: 'Kedr Livansky'

}

];

const songsJSON = JSON.stringify(songs);

console.log(typeof songsJSON); *// "string"*

const songsObject = JSON.parse(songsJSON);

console.log(typeof songsObject); *// "object"*

console.log(songsObject[0].title); *// "Вектора"*

Методу JSON.parse нельзя передать какую угодно строку:

Скопировать кодJAVASCRIPT

JSON.parse('Привет'); *// SyntaxError: Unexpected token П in JSON at position 0*

Строка должна быть JSON-совместимой, то есть:

* на верхнем уровне должен быть объект или массив;
* ключи и значения-строки должны быть в двойных кавычках;
* кроме строк JSON может содержать числовые, булевы значения или null.

## Метод res.json

Метод fetch возвращает объект ответа:

Скопировать кодJAVASCRIPT

fetch('https://praktikum.yandex.ru')

.then((res) => {

console.log(res); *// В консоли окажется объект ответа от сервера*

});

Мы познакомимся с ним подробнее, но сейчас нас интересует один его метод — json. Метод json читает ответ от сервера в формате json и возвращает промис. Из этого промиса потом можно доставать нужные нам данные.

Поскольку метод json асинхронный, то и использовать его нужно таким образом:

Скопировать кодJAVASCRIPT

fetch('https://praktikum.yandex.ru')

.then((res) => {

return res.json(); *// возвращаем результат работы метода и идём в следующий then*

})

.then((data) => {

console.log(data.user.name); *// если мы попали в этот then, data — это объект*

})

.catch((err) => {

console.log('Ошибка. Запрос не выполнен');

});

# Важно

Ещё раз: json — асинхронный метод. Такой код некорректен:

Скопировать кодJAVASCRIPT

fetch('https://praktikum.yandex.ru')

.then((res) => {

const data = res.json();

console.log(data.user.name); *// data — это промис, она ещё не готова*

})

.catch((err) => {

console.log('Ошибка. Запрос не выполнен');

});

Пример:

Методами JSON.stringify и JSON.parse создайте глубокую копию объекта user.

const user = {

name: 'Иван',

age: 30,

hasDog: true,

dog: {

name: 'Бонни',

age: 6

}

};

const userDeepCopy = JSON.parse(JSON.stringify(user));

console.log(user === userDeepCopy); // false

console.log(user.dog === userDeepCopy.dog); // false

console.log(user.name === userDeepCopy.name); // true

console.log(user.dog.name === userDeepCopy.dog.name); // true

Пример:

Одна цитата быстро наскучит. Сделаем так, чтобы она обновлялась при нажатии на кнопку. Для этого:

* создайте функцию updateQuote и поместите туда весь код, который подгружает данные и добавляет их в DOM;
* повесьте на кнопку обработчик клика: updateQuote должна срабатывать при нажатии на эту кнопку.

const quoteElement = document.querySelector('.quote');

function updateQuote () {

fetch('https://api.kanye.rest')

.then((res) => {

return res.json()

})

.then((data) => {

quoteElement.textContent = data.quote

})

}

const btn = document.querySelector('.header\_\_btn');

btn.addEventListener('click', updateQuote);

# HTTP-запрос

Вы уже знаете о HTTP-запросе как о способе получать данные сервера. В этом уроке разберёмся, из чего состоит HTTP-запрос и как эти запросы делать. Для нас в запросе важны три вещи: HTTP-метод, заголовки и тело запроса. Разберём всё по порядку.

# Методы HTTP

Метод HTTP-запроса определяет, что нужно сделать с запрашиваемым ресурсом. Разным целям служат разные методы: одни нужны для получения данных, другие — для отправки, третьи — для удаления. Все методы разбирать не будем — остановимся только на основных.

# GET

GET — самый распространённый метод. Данные обычно получают именно этим методом. Если метод не прописать явно, fetch будет отправлять запросы методом GET:

Скопировать кодJAVASCRIPT

*// это GET запрос к ресурсу "https://example.com"*

fetch('https://example.com')

.then((res) => {

return res.json();

})

.then((data) => {

console.log(data.user.name); *// если мы попали в этот then, data — это объект*

})

.catch((err) => {

console.log('Ошибка. Запрос не выполнен: ', err);

});

# POST

POST — второй по распространённости метод. Его используют для отправки данных на сервер. Чтобы послать запрос методом POST, нужно указать этот метод в объекте опций — втором аргументе метода fetch:

Скопировать кодJAVASCRIPT

*// POST запрос к ресурсу "https://example.com"*

fetch('https://example.com', {

method: 'POST'

});

Поскольку метод POST отправляет данные, эти данные нужно как-то хранить в запросе. Для этого их нужно перевести в формат JSON и записать в свойство body объекта опций:

Скопировать кодJAVASCRIPT

fetch('https://example.com', {

method: 'POST',

body: JSON.stringify({

name: 'Иван',

age: 30

})

});

На самом деле данные можно отправлять не только в формате JSON — есть и другие форматы. Поэтому серверу нужно как-то понять, что за данные ему пришли. За это отвечает так называемый MIME Type, или Media Type. Его прописывают в заголовке запроса Content-Type.

Много английских непонятных слов — сейчас со всеми разберёмся.

# MIME Type

Чтобы сервер мог обработать данные, он должен понимать, чтó мы ему отправили. Для этого в вебе есть своя система указания типов — MIME Types. По этой системе типы указывают строкой, состоящей из типа и подтипа документа, разделённых слешем:

Скопировать код

type/subtype

MIME типов десятки — с большинством из них вам работать не придётся. Поэтому вот самые распространённые:

Скопировать код

text/plain

text/html

text/javascript

text/css

image/jpeg

image/png

audio/mpeg

audio/ogg

video/mp4

application/json

application/octet-stream

application/x-www-form-urlencoded

multipart/form-data

Из многих названий понятно, что представляют собой данные этого типа. text/plain — это просто текст, image/jpeg — картинка в формате jpg.

Для отправки данных формы обычно используют один из трёх типов:

* application/json — для передачи данных в формате JSON;
* application/x-www-form-urlencoded — формат, который кодирует поля формы так, чтобы их можно было отправить в URL;
* multipart/form-data — для отправки файлов на сервер. Подойдёт, если среди прочего вы отправляете через форму картинку.

[Статья о MIME-типах на MDN.](https://developer.mozilla.org/ru/docs/Web/HTTP/Basics_of_HTTP/MIME_types)

# Тело и заголовки запроса

MIME тип посылают в специальном заголовке Content-Type. Заголовки передаются в свойстве headers второго аргумента метода fetch. Чтобы послать в теле запроса данные в JSON, нужно передать в заголовках тип application/json.

Некоторые заголовки, например Content-Length (размер пересылаемых данных), браузер установит запросу самостоятельно.

Заголовки не чувствительны к регистру. Например, это одинаковые заголовки:

Скопировать код

Content-Type: application/json

content-type: application/json

# Как передать данные на сервер

Мы говорили, что отправлять данные на сервер можно в теле запроса — свойстве body. Но это не единственный способ. Дело в том, что обработка нашего запроса — забота сервера. Поэтому только реализация сервера определяет, в каком виде данные должны поступать.

# В параметрах запроса

Можно отправлять в параметрах запроса. Их перечисляют прямо в URL после вопросительного знака:

Скопировать кодJAVASCRIPT

fetch('https://example.com/images/random?type=portrait', {

method: 'GET'

});

Имя параметра и его значение указывают друг за другом через знак равенства: name=value. Такие пары имя атрибута=значение разделяют амперсандами &:

Скопировать кодJAVASCRIPT

fetch('https://example.com/images/random?type=portrait&name=DorianGrey', {

method: 'GET'

});

# Прямо в URL

Ещё один способ передать данные — указать их в URL. Например, если картинке присвоен идентификатор можно запросить его в URL и получить картинку:

Скопировать кодJAVASCRIPT

fetch('https://example.com/images/ewfh23d832jnf2903', { *// ewfh23d832jnf2903 — идентификатор картинки*

method: 'GET'

});

# Другие методы запросов

Раньше все запросы к серверу делали методами GET и POST. Но со временем появились новые методы. Они немного отличаются от старых и служат конкретным целям.

* PUT предназначен для полного обновления указанного ресурса. Например, если в каталоге книг вы решили заменить одну книгу другой;
* PATCH — для частичного обновления ресурса. Например, при обновлении профиля пользователя;
* DELETE — для удаления ресурса с сервера.

# Резюме

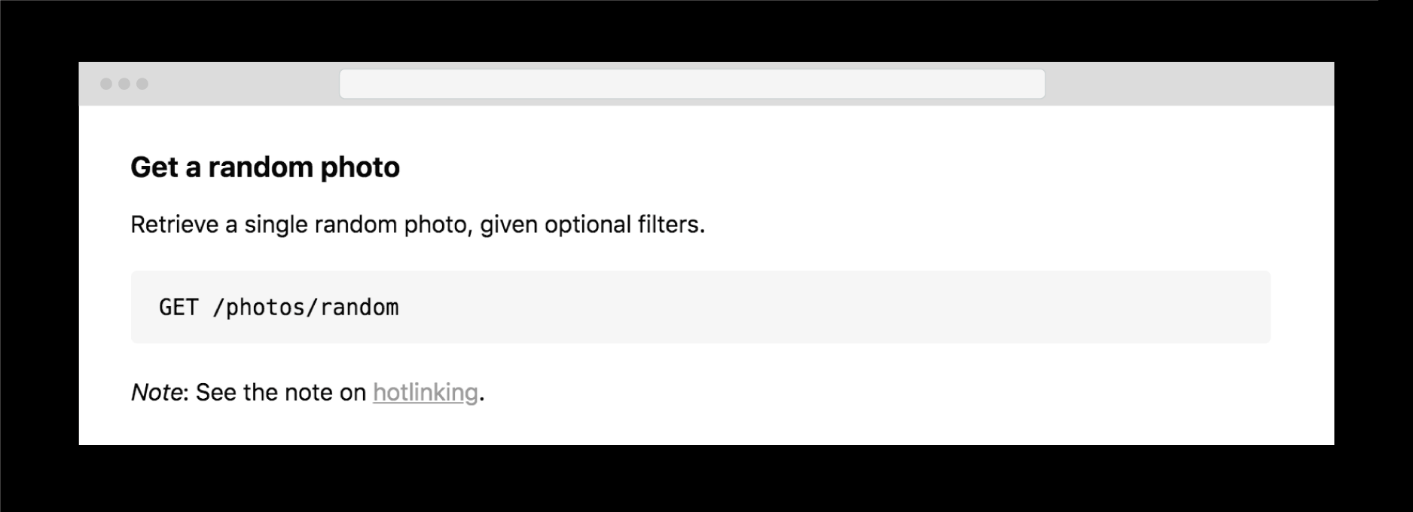
Основные HTTP-методы — это GET, POST, PUT, PATCH и DELETE.

Есть три основных способа передавать данные на сервер:

* в теле запроса;
* в параметрах запроса;
* в URL.

Какой из этих способов выбрать, зависит от реализации сервера. Поэтому если не знаете — идите к бэкендеру, который программировал сервер для вашего сайта. А если отправляете запросы в API — читайте его документацию.

К примеру, вот документация API сервиса бесплатных фотографий: <https://unsplash.com/documentation>.



Здесь сказано: чтобы получить случайную фотографию, отправьте GET запрос на URL unsplash.com/photos/random

Пример:

// создаёт разметку для поста

function createPostMarkup(post) {

return `

<div class="post">

<p class="post\_\_title">${post.title}</p>

<p class="post\_\_text">${post.body}</p>

</div>

`;

}

// вставляет разметку в DOM

function addPostToDOM(container, markup) {

container.insertAdjacentHTML('afterbegin', markup);

}

const postsContainer = document.querySelector('.container');

function getPosts() {

fetch('https://jsonplaceholder.typicode.com/posts')

.then(res => res.json())

.then((posts) => {

posts.forEach((post) => {

addPostToDOM(postsContainer, createPostMarkup(post))

})

})

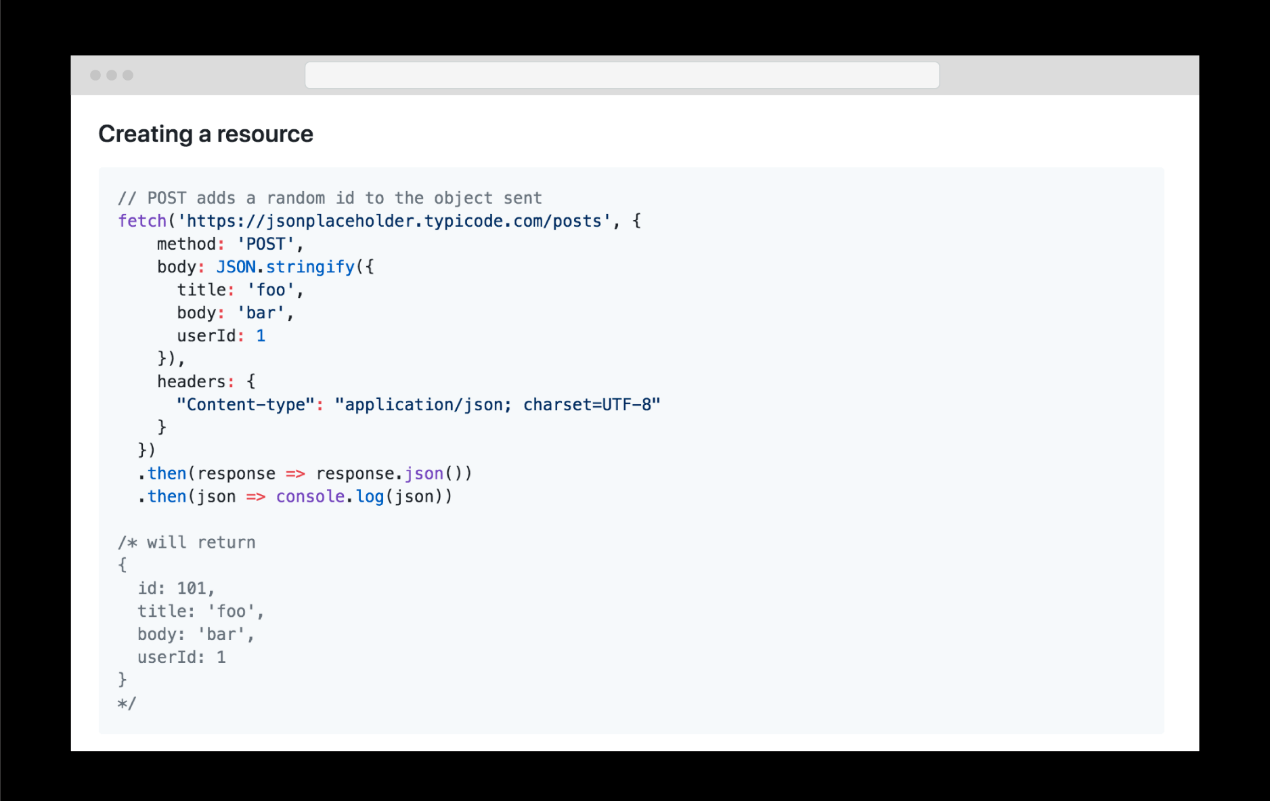
}

getPosts();

Пример:

Добавим пост. Иронично, что для этого нам понадобится метод POST. Чтобы понять, какой метод использовать и как, нужно посмотреть в документации к API. Ну или спросить у разработчика API, если документации нет.

Нам повезло — у сервиса JSONplaceholder есть [документация](https://jsonplaceholder.typicode.com/), даже с [примерами запросов](https://github.com/typicode/jsonplaceholder#creating-a-resource):



Здесь показано:

* как сделать POST-запрос, чтобы создать пост;
* что такой запрос вернёт.

POST-запрос к JSONplaceholder вернёт объект с постом. Это то, что нужно: после отправки запроса нам нужно отрисовать пост на странице. Для этого можно будет взять его прямо из ответа.

Напишите функцию createPost. В ней должен быть вызов fetch, который:

* делает POST-запрос по адресу <https://jsonplaceholder.typicode.com/posts>;
* с телом — JSON с двумя свойствами title и body;
* со свойством headers с единственным заголовком: 'Content-Type': 'application/json; charset=UTF-8';

Добавьте fetch два обработчика then:

* чтобы разобрать результат методом json;
* чтобы затем вывести этот результат в консоль.

// создаёт разметку для поста

function createPostMarkup(post) {

return `

<div class="post">

<p class="post\_\_title">${post.title}</p>

<p class="post\_\_text">${post.body}</p>

</div>

`;

}

// вставляет разметку в DOM

function addPostToDOM(container, markup) {

container.insertAdjacentHTML('afterbegin', markup);

}

function createPost(newPost) {

fetch('https://jsonplaceholder.typicode.com/posts', {

method: 'POST',

body: JSON.stringify({

title: newPost.title,

body: newPost.body

}),

headers: {

'Content-Type': 'application/json; charset=UTF-8'

}

})

.then(res => res.json())

.then((post) => {

addPostToDOM(document.querySelector('.container'), createPostMarkup(post)) })

}

// обработчик сабмита формы

document.forms.post.addEventListener('submit', function (event) {

event.preventDefault();

const { title, text } = event.currentTarget.elements;

createPost({

title: title.value,

body: text.value

});

});

# Ответ

Вы уже знаете, что от сервера приходит объект ответа. Чтобы получить из него данные, нужно вызвать встроенный метод json:

Скопировать кодJAVASCRIPT

fetch('https://api.kanye.rest')

.then(res => res.json())

.then((data) => {

console.log(data);

});

Но есть у объекта ответа также другие методы и свойства. О них сейчас и пойдёт речь.

# Статус

Наверняка с вами такое случалось: заходите на сайт, а вместо нужной страницы на экране большое число 404. Эти три цифры — код статуса ответа.

У ответа всегда есть статус — он может быть отклонён, перенаправлен или вернуться с ошибкой. Каждому статусу запроса соответствует уникальный код. 404 — один из таких кодов.

По коду статуса можно понять, что произошло с запросом. Если код начинается с:

* 2 — значит, запрос прошёл успешно;
* 3 — запрос был перенаправлен;
* 4 — с запросом что-то не так: ресурс не найден или у вас нет к нему доступа;
* 5 — на сервере произошла какая-то ошибка.

Вот самые распространённые статусы — с ними вы столкнётесь скорее всего:

* 200 OK;
* 401 Unauthorized;
* 403 Forbidden;
* 404 Not Found;
* 500 Internal Server Error.

После числа идёт сообщение статуса — оно объясняет, что произошло. Код и сообщения статуса хранятся в свойствах объекта ответа status и statusText:

Скопировать кодJAVASCRIPT

fetch('https://api.kanye.rest')

.then(res => {

console.log(res.status, res.statusText); *// 200 OK*

});

Для удобства также доступно булево свойство ok. Оно хранит в себе true, если ответ успешный (начинается с 2), и false — в любом другом случае:

Скопировать кодJAVASCRIPT

fetch('https://api.kanye.rest')

.then(res => {

console.log(res.ok); *// true*

});

По этому свойству можно определить, прошёл ли запрос успешно, и принять решение, что делать дальше:

Скопировать кодJAVASCRIPT

const quoteElement = document.querySelector('div.quote');

fetch('https://api.kanye.rest')

.then((res) => {

if (res.ok) {

return res.json();

}

*/\* отклоняем промис, чтобы перейти*

*в блок catch, если сервер вернул ошибку \*/*

return Promise.reject(`Что-то пошло не так: ${res.status}`);

})

.then((data) => {

quoteElement.textContent = data.quote;

})

.catch((err) => {

console.log(err); *// "Что-то пошло не так: ..."*

});

# Заголовки ответа

Подобно запросам, ответы могут иметь заголовки. В них содержится дополнительная информация от сервера.

# Как получить данные из заголовков

Такой код выведет в консоль пустой объект, хотя ответ имеет заголовки:

Скопировать кодJAVASCRIPT

fetch('https://api.kanye.rest')

.then((res) => {

console.log(res.headers); *// Headers {}*

});

Дело вот в чём. Заголовки нечувствительны к регистру, а ключи JS-объекта — чувствительны. Будь res.headers обычным объектом, было бы нельзя понять, какому ключу должен соответствовать, например, заголовок Content-Type:

Скопировать кодJAVASCRIPT

res.headers['Content-Type']; *// такому?*

res.headers['content-type']; *// или такому?*

res.headers['CONTENT-TYPE']; *// а, может, такому?*

Поэтому для работы с заголовками есть специальные методы. Чтобы получить значение заголовка, есть метод get:

Скопировать кодJAVASCRIPT

fetch('https://api.kanye.rest')

.then((res) => {

if (res.headers.get('Content-Type').contains('application/json')) {

return res.json();

}

});

У объекта заголовков есть и [другие методы](https://developer.mozilla.org/en-US/docs/Web/API/Headers).

Единственное отличие заголовков ответа от заголовков запроса: первые можно только читать, но нельзя редактировать.

# Тело ответа

У ответа почти всегда есть тело. Если мы запрашиваем веб-страницу, тело ответа — HTML-код этой страницы. Телом ответа бывает просто текст, картинка или JSON-строка. Чтобы разобрать тело ответа, есть специальные методы:

* res.json — разбирает JSON в объект, этот метод вы уже знаете;
* res.text — разбирает тело как текст;
* res.blob — разбирает тело ответа как бинарные данные: это нужно при получении файлов (изображений, видео, pdf-документов).

В основном мы будем использовать метод json, чтобы читать тело запроса как JSON-объект.

Все методы для чтения тела ответа работают асинхронно:

Скопировать кодJAVASCRIPT

*/\* методы разбора тела возвращают промис*

*их нужно использовать асинхронно \*/*

fetch('https://api.kanye.rest')

.then(res => res.json())

.then((result) => {

console.log(result);

});

fetch('https://api.kanye.rest')

.then(res => res.text())

.then((result) => {

console.log(result);

});

Логичный вопрос: если мы уже получили ответ от сервера, почему его нужно обрабатывать асинхронно? Почему его нельзя сразу посмотреть?

Скопировать кодJAVASCRIPT

*// почему нельзя было сделать так? удобней ведь*

fetch('https://api.kanye.rest')

.then(res => {

const json = res.json();

console.log(json);

});

Дело в том, что сервер разбивает данные на небольшие порции, когда отправляет нам ответ. Если от сервера пришёл ответ, это ещё не значит, что пришли все данные. Поэтому сначала мы обрабатываем приход ответа, а затем — данных. Отсюда и асинхронный код.

Можно провести аналогию с телефонным разговором: если наш собеседник снял трубку, для нас это ответ — соединение успешно установлено. Но данные мы получим, только когда собеседник нам что-то скажет.

# Инструменты: вкладка Network

Вы научились посылать запросы, получать ответы, читать их тела и заголовки. Но как анализировать полную информацию о запросах и ответах? Неужели всякий раз вставлять console.log?

Скопировать кодJAVASCRIPT

fetch('https://api.kanye.rest')

.then(res => {

console.log(res.status); *// какой у ответа статус?*

console.log(res.headers.get('Content-Type')); *// а Content-Type какой?*

return res.json();

})

.then((data) => {

console.log(data); *// а что пришло в теле?*

});

Конечно, можно и так, но неудобно. А чтобы было удобно, нужен инструмент дебаггинга. К счастью, он есть: вкладка Network в инструментах разработчика представляет собой средство анализа запросов и ответов.

# Что такое сетевой мониторинг

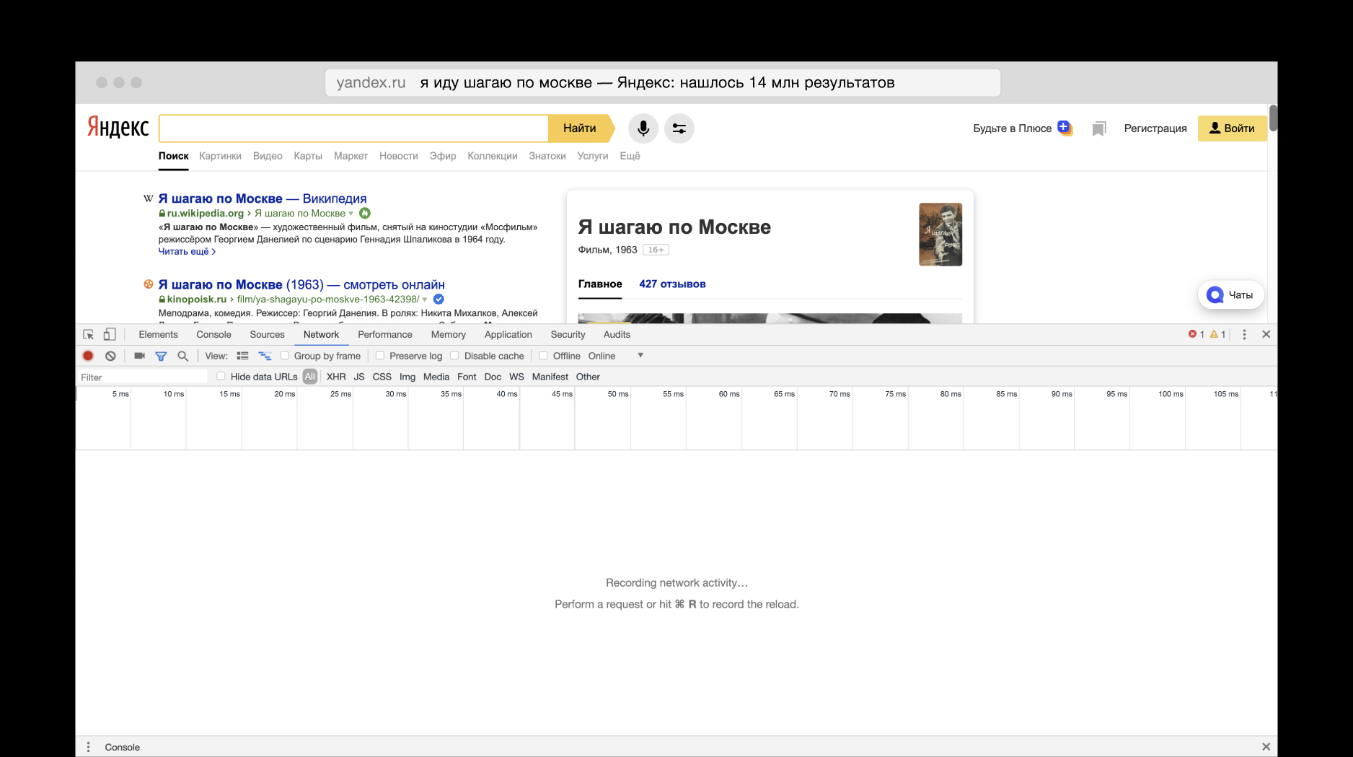
Network Monitor — инструмент для слежки за запросами, которые происходят в рамках сессии работы с сайтом.

Мы в примерах используем [Яндекс.Браузер](https://browser.yandex.ru/" \t "_blank). Если вы пользуетесь Firefox или Opera, команды и вкладки могут отличаться.

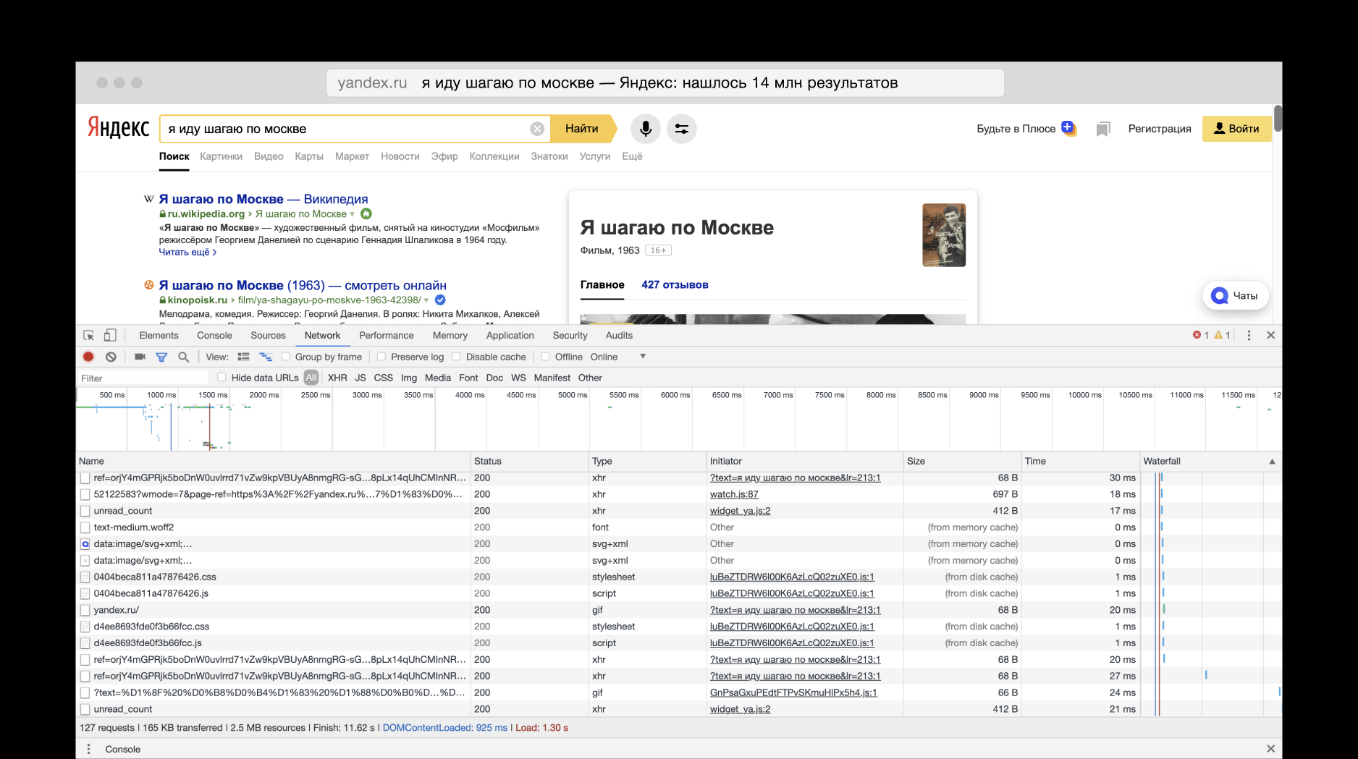
### Как открыть вкладку Network

Для этого нужно открыть инструмент разработчика сочетанием клавиш Command+Option+I (macOS) или Control+Shift+I (Windows, Linux). Таким сочетанием клавиш вы попадёте сразу в вкладку Network. Но если этого не случилось, найдите вкладку Network и перейдите на неё.

Вы увидите такой экран:



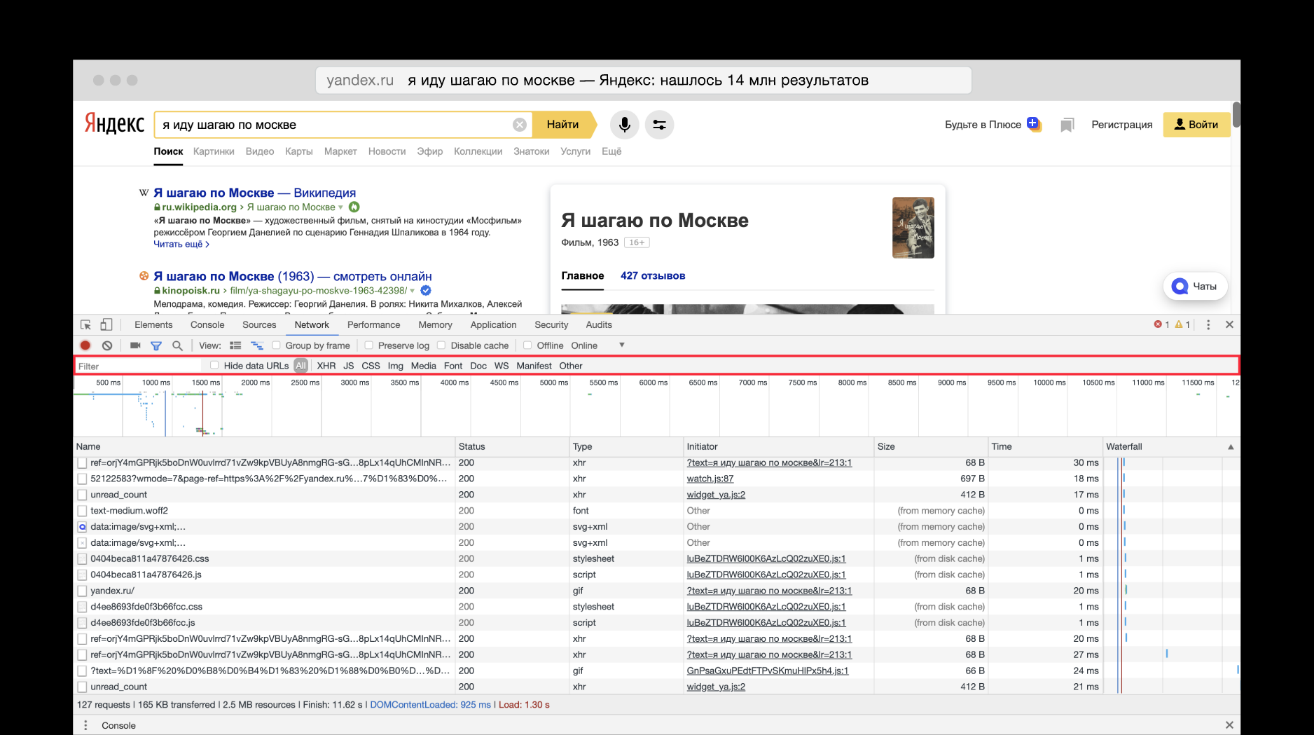
Рассмотрим, что расположено на основном экране. Вкладка Network состоит из трёх основных частей. Перезагрузите страницу с открытым инструментом:



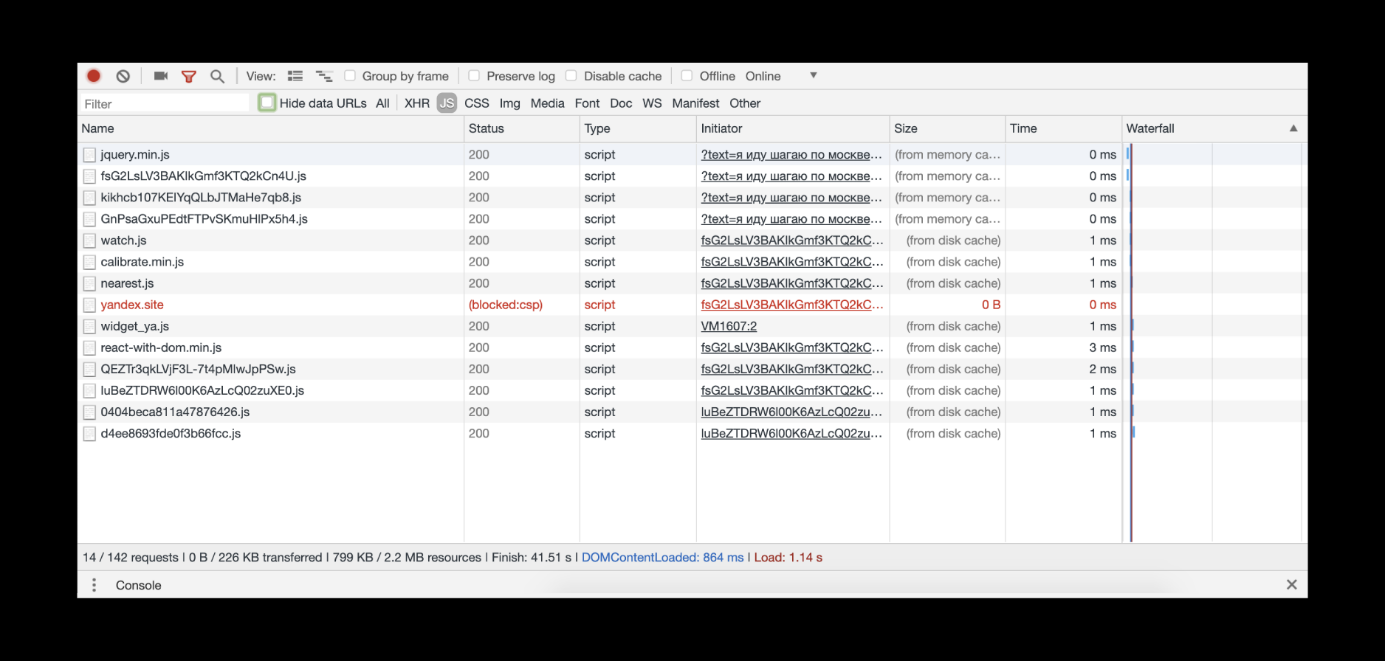
Перед вами все HTTP-запросы, которые делает страница. Здесь есть запросы HTML-документов, CSS- и JS-файлов, картинок, а также fetch-запросы из JavaScript.

## Фильтрация запросов

В таком количестве запросов сложно найти тот, что нас интересует. Но запросы можно фильтровать по типу. Для этого на выделенной панели нужно кликнуть на тип запроса:



Например, если кликнуть на «JS», браузер отобразит только запросы JavaScript файлов:



Нас в основном будет интересовать вкладка XHR. Именно там появляются запросы сделанные методом fetch из JavaScript кода.

# Внутренности запроса

Откройте инструменты разработчика в новом окне, иначе запрос не сработает. Затем скопируйте в консоль такой код и нажмите Enter:

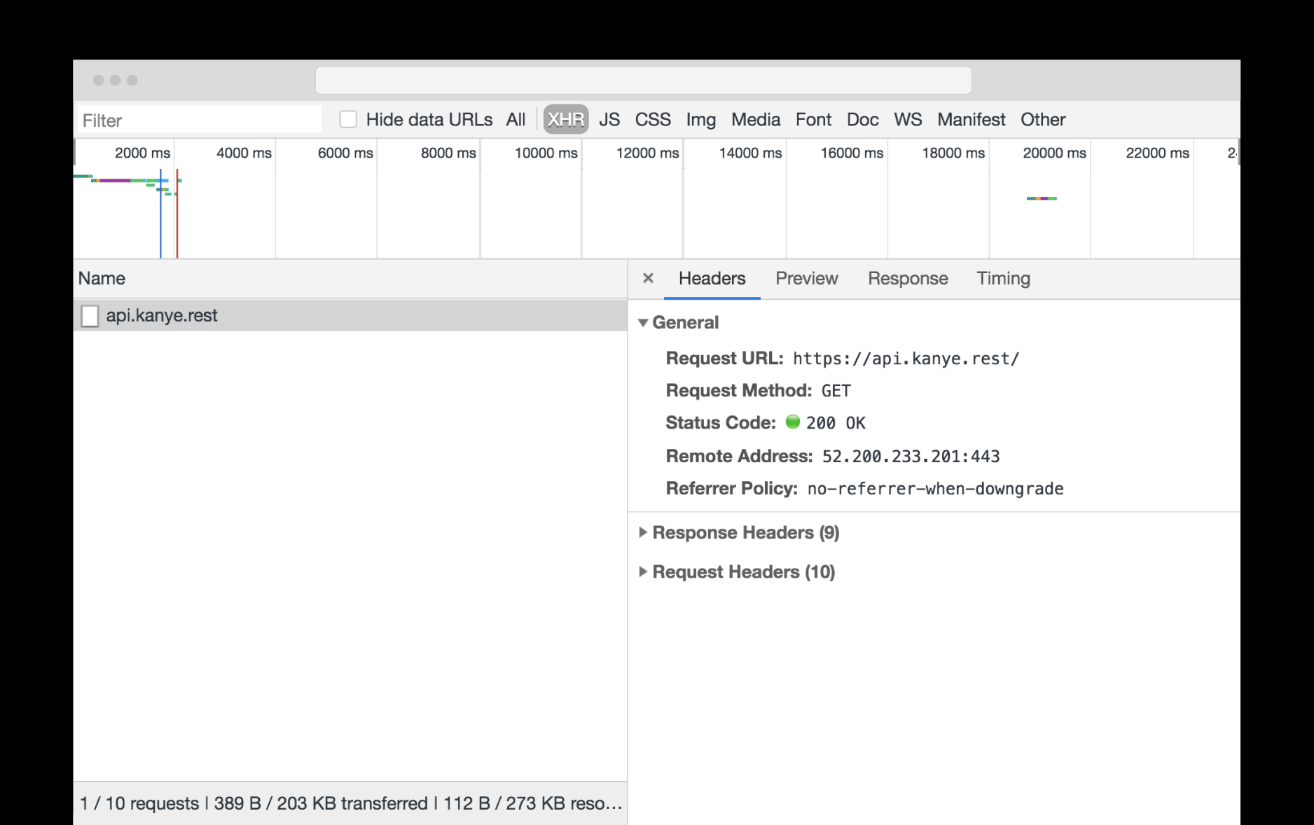
Скопировать кодJAVASCRIPT

fetch('https://api.kanye.rest');

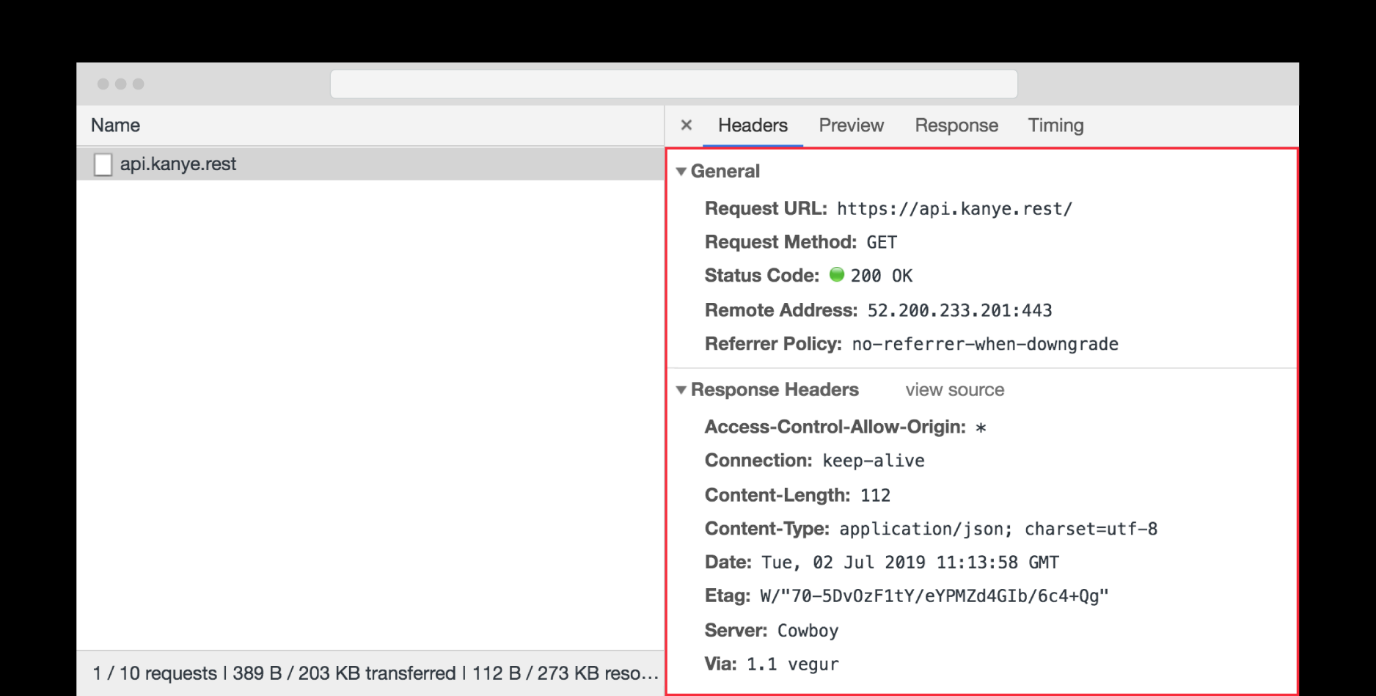
Затем:

* перейдите во вкладку Network;
* отфильтруйте XHR запросы;
* найдите в списке запрос [api.kanye.rest](http://api.kanye.rest/" \t "_blank) и кликните на него.

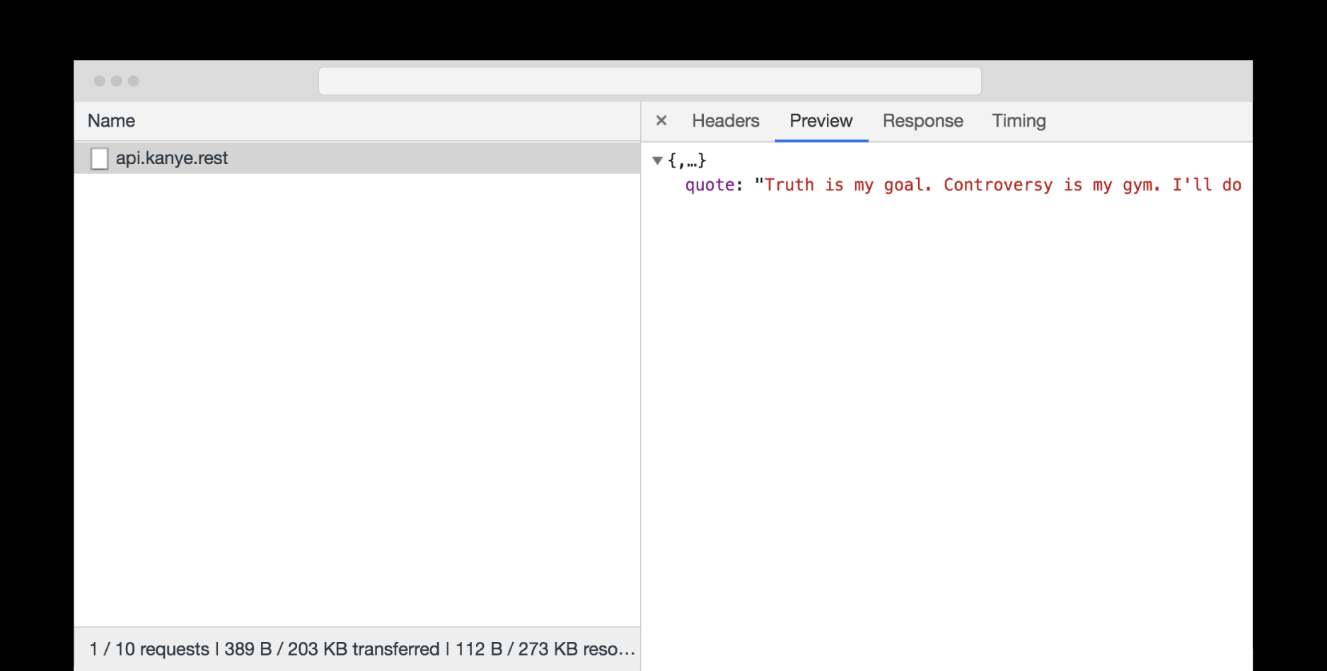
Вы увидите внутренности запроса:



Справа основная информация о запросе (его URL, метод, статус ответа), а также заголовки запроса и ответа:



Кликнув на вкладку «Preview», вы увидите тело запроса:



Часто в теле ответа приходит JSON с множеством разных полей. Вкладка «Preview» позволяет быстро проанализировать структуру и найти нужную информацию.

Во вкладке Network множество инструментов. С их помощью можно:

* анализировать внутренности запросов;
* смотреть, какие файлы приходят с сервера и как они влияют на время загрузки сайта;
* включать режим «медленного интернета» и наблюдать, как сайт загружается.

В этом уроке вы познакомились только с первым пунктом — это то, что понадобится при выполнении практической работы. С остальными инструментами мы будем знакомиться по ходу, но если вам не терпится — добро пожаловать в дополнительные материалы темы.

# Дополнительно: стандартная отправка формы

Это дополнительный урок. То, что здесь изложено, на практике применяется редко. Поэтому вы можете пропустить этот урок и вернуться к нему после сдачи практической работы.

## Что, если не fetch?

В этой теме вы научились делать запросы из JavaScript методом fetch. Это современный способ обмена информацией с сервером. Когда мы загружаем данные через fetch, браузер не обновляет страницу, а перестраивает нужную часть DOM-дерева. Так сайт работает плавнее и пользователь доволен.

Но возможность обращаться к серверу через JavaScript появилась не сразу. До этого был другой, уже устаревший способ отправить запрос серверу. Возможно, вам не придётся пользоваться этим методом, но его всё равно полезно знать. Потому урок и дополнительный.

Как бы мы ни отправляли данные на сервер, нужна форма. Когда на ней происходит событие submit, браузер перезагружает страницу, а данные формы уходят на сервер. Чтобы перезагрузки не происходило, мы отменяли это стандартное поведение формы:

Скопировать кодJAVASCRIPT

form.addEventListener('submit', function (evt) {

evt.preventDefault();

});

В этом уроке мы разберём, что происходит, если событие submit не отменять.

## Поведение по умолчанию

Вёрстка в вебе изначально мало отличалась от бумажной. В начале 90-х сайт выглядел как журнал: можно было переходить от страницы к странице, просматривать изображения и текст. Но со временем этого стало мало — нужно было научить сайты получать и отправлять на сервер данные от пользователя. Для этого и были придуманы формы.

Работа с формами была организована так: пользователь вводит данные и отправляет форму на сервер. Для этого браузер формирует запрос. Чтобы сформировать адрес запроса и его тело, браузер должен как-то сообщить серверу, какие данные к какому полю относятся. Для этого формы были наделены атрибутами. Именно об этих атрибутах и пойдёт речь.

## Атрибуты форм

У тегов, относящихся к форме (form, input, select), есть специальные атрибуты. Они отвечают за то, как эта форма будет отправлена (если использовать стандартное поведение).

Рассмотрим простую форму регистрации:

Скопировать кодHTML

<form id="form">

<label for="username">

Имя пользователя

<input type="text" name="username" id="username">

</label>

<label for="email">

Email

<input type="email" name="email" id="email">

</label>

<label for="password">

Пароль

<input type="password" name="password" id="password">

</label>

<button type="submit">Отправить</button>

</form>

При нажатии «Отправить» страница перезагрузится, а в адресной строке появится:

Скопировать код

<путь к файлу с кодом>/index.html?username=&email=&password=

Что же произошло?

Когда мы нажали «Отправить», браузер сформировал запрос c определёнными параметрами: ?username=&email=&password=. Он взял их из атрибутов name полей ввода, а после знака равенства вставил значения этих параметров. Получилась строка, состоящая из пар «атрибут=значение», разделённых амперсандами. В нашем примере значения пустые, потому что мы не заполняли форму, а просто нажали «Отправить».

Дальше браузер отправил на сервер GET-запрос. По умолчанию запрос уйдёт к тому же ресурсу, от которого получена веб-страница, но это поведение можно изменить.

В нашем примере запрос будет отправлен на такой адрес:

Скопировать код

<путь к файлу с кодом>/index.html

У такого способа отправки данных есть ощутимые недостатки:

1. GET-запросы подходят для отправки нечувствительных данных. Пароль, email, имя пользователя, личные данные отправлять так нельзя — это небезопасно.
2. В GET-запрос можно уместить примерно 3000 знаков, отправить длинный текст так не выйдет.
3. GET-запрос лучше использовать для получения данных, а для отправки использовать POST-запрос. Так и вам, и другим разработчикам будет понятно, что каждый запрос делает: запрашивает данные или отправляет.

## Настройка запроса

Есть способ исправить все эти недостатки — использовать атрибуты action и method.

Первый — action — позволяет указать путь к обработчику — серверной программе, которой предстоит обрабатывать форму. Как мы уже говорили, по умолчанию запрос уйдёт к тому же ресурсу, с которого загрузилась страница. Как раз атрибут action позволяет это изменить.

Второй атрибут — method — даёт возможность задать метод, которым данные формы нужно отправлять. Может принимать два значения: GET и POST.

В полях ввода, мы указали атрибут name. С помощью этого атрибута данные, введённые в определённое поле, будут присвоены значению, которое указано в атрибуте name. Допустим, мы указали в поле «Имя пользователя» значение супер фронтенд, в поле «email» — vasyafrontend@yandex.ru, а в поле пароля — 12345678. Тогда, при отправке методом GET будет сформирована строка:

Скопировать код

\*<путь к файлу с кодом>/register?username=супер фронтенд&email=vasyafrontend@yandex.ru&password=12345678\*

При отправке методом POST — это будет преобразовано в объект FormData, из которого на сервере эти данные можно будет получить по значению ключа — по аналогии с обычным JavaScript объектом.

Скопировать кодHTML

<form id="form" action="/register" method="POST">

<label for="username">

Имя пользователя

<input type="text" name="username" id="username">

</label>

<label for="email">

Email

<input type="email" name="email" id="email">

</label>

<label for="password">

Пароль

<input type="password" name="password" id="password">

</label>

<button type="submit">Отправить</button>

</form>

Методами PUT, PATCH и DELETE стандартно отправить форму никак не выйдет.

### Кодировка данных

Как мы уже сказали, все данные, которые ввёл пользователь, попадают либо в адресную строку, либо в тело запроса. Но как именно составить такую строку или такое тело? Есть несколько способов. Какой именно из них использовать, указывают в атрибуте enctype. Он может принимать одно из трёх значений:

1. application/x-www-form-urlencoded

Значение по умолчанию. При таком способе кодирования все латинские символы и цифры останутся, пробелы будут заменены плюсами (+), а русские буквы и другие спецсимволы — на шестнадцатиричные коды.

Вот как будут выглядеть данные формы:

Скопировать код

%3C%D0%BF%D1%83%D1%82%D1%8C%20%D0%BA%20%D1%84%D0%B0%D0%B9%D0%BB%D1%83%20%D1%81%20%D0%BA%D0%BE%D0%B4%D0%BE%D0%BC%3E%2Fregister%3Fusername%3D%D1%81%D1%83%D0%BF%D0%B5%D1%80%20%D1%84%D1%80%D0%BE%D0%BD%D1%82%D0%B5%D0%BD%D0%B4%26email%3Dvasyafrontend%[40yandex.ru](http://40yandex.ru/)%26password%3D12345678

1. multipart/form-data

Этот тип кодирования можно использовать только с запросами, у которых есть тело. То есть, если вы отправляете форму стандартным образом, — с POST-запросами. Работает он так: все данные разбиваются на пары «атрибут=значение», разделённые какой-то строкой. Эту строку-разделитель можно генерировать случайно или указать в заголовке Content-Type. Дальше данные без каких-либо преобразований отправляются в теле запроса, а не в URL. Этот способ кодирования обычно используют для отправки файлов.

1. text/plain заменяет пробелы на плюсы, а сами данные — не трогает. При этом данные формы будут выглядеть так:

Скопировать код

<путь к файлу с кодом>/register?username=супер+фронтенд&email=vasyafrontend@yandex.ru&password=12345678

А код так:

Скопировать кодHTML

<form id="form" action="/register" method="POST" enctype="text/plain">

<label for="username">

Имя пользователя

<input type="text" name="username" id="username">

</label>

<label for="email">

Email

<input type="email" name="email" id="email">

</label>

<label for="password">

Пароль

<input type="password" name="password" id="password">

</label>

<button type="submit">Отправить</button>

</form>

В современном коде обработкой форм всегда занимается JavaScript. Способ отправки запросов, описанный в этом уроке, может пригодиться, если вы разбираете старый код. Либо если хотите заставить сайт работать, даже если у пользователя JavaScript отключён.

Этот урок не принёс вам практических навыков. Зато теперь вы можете достичь в своих проектах лучшего взаимопонимания между клиентом и сервером. Даже безо всякой поддержки со стороны JavaScript.

# Работа с API. Заключение

Не расстраивайтесь, если концепция асинхронности не даётся сразу: практика и упорство расставят всё по местам.

Подключение сайта к серверу — отдельный вид магии. Наконец-то проект станет полноценным: добавленные карточки и отредактированное имя будут сохраняться после перезагрузки страниц. Кроме этого, на странице будут появляться карточки, загруженные другими студентами вашей когорты — можно будет их лайкать, смотреть сколько лайков у каждой карточки и кто их поставил.

Пока все эти возможности есть только в наших мечтах. Но совсем скоро вы запрограммируете эту функциональность!

 Небольшой совет, как реализовать удаление карточки с подтверждением через popup. Если бы вам надо было расписать логики удаления без присутствия промежуточного попапа (то есть удаление происходило сразу при нажатии на кнопку с помойным ведерком), то это могло бы выглядеть примерно так:

const createCardFunction = (data) => {  
 const card = new Card({  
 ...  
 handleCardClick: () => {  
 ...  
 },  
 handleLikeClick: (id) => {  
 ...  
 },  
 handleDislikeLikeClick: (id) => {  
 ...  
 },  
 handleDeleteClick: (id) => {  
 api.removeCard(id).then(res => {  
 card.removeCard()  
 }).catch(err => console.error(err))  
 }  
 }, ...); ...  
 }

Суть тут в том, что мы изнутри класса Card вызывали бы функцию handleDeleteClick, передаваемую в конструктор в качестве аргумента при создании экземпляра.  
Что делала бы эта функция? Она отправляла api запрос на удаление, дожидалась ответа и в случае если все ок - вызывала публичный метод removeCard (для того чтобы убрать карточку из разметки).  
Теперь как добавить промежуточную логику с попапом, который ожидает подтверждения пользователя? все просто! Предположим что вы создали класс для этого попапа с подтверждением, в котором есть публичный метод, который динамически позволяет менять функцию, которая вызывается при нажатии на кнопку сабмита.  
Следовательно вам надо this.\_handleSubmitCallback использовать внутри коллбека submit, который вы будете накладывать на форму текущего попапа. выглядеть это может примерно вот так

class PopupWithFormSubmit extends Popup {  
 ...  
 setSubmitAction(submitAction) {  
 this.\_handleSubmitCallback = submitAction;  
 } setEventListeners () {  
 // тут на форму текущего попапа накладываете обработчик и внутри него вызываете метод который заключен в this.\_handleSubmitCallback  
 }  
 ...  
}

то есть приватный метод \_handleSubmitCallback будет перезаписываться через публичный метод setSubmitAction. Теперь мы можем переписать логику карточки следующим образом:

const submitPopup = ...  
const createCardFunction = (data) => {  
 const card = new Card({  
 ...  
 handleDeleteClick: (id) => {  
 submitPopup.setSubmitAction(\_ => {  
 // сюда прописать действия которые необходимо выполнить после нажатия на кнопку внутри попапа подтверждения  
 })  
 // а тут открыть попап уже с установленным действием, то есть удалением текущей карточки  
 }  
 }, ...);  
 ...  
}

сначала пройдем по алгоритму для постановки \ снятия лайка

1. Нажали на кнопку лайк
2. Проверили есть ли уже лайк. Если есть отправляем delete запрос, если нет, то PUT
3. Как только получили результаты с сервера, обновляем количество лайков и состояние кнопки (окрашена или нет)

Теперь по изменениям, которые я думаю, есть смысл внести

1. Необходимо создать в index.js метод createCard(data), который будет принимать данные, создавать карточку и возвращать ее. Сейчас у тебя дублируется код по созданию карточки
2. Действия по обновлению состояния кнопки лайк и количества лайков нужно описать в then блоке, после  того как запрос успешно выполнится. Сейчас у тебя вот так

api.setLike(item.\_id)  
 .then((res) => {  
 return res  
 })

В данном случае нет смысла в return res , он вернется в следующий then, которого у нас нет. Нужно сделать, как то так

api.setLike(item.\_id)  
 .then((res) => {  
 item.handleLikes(res);  
 })

3. Рассмотрим метод handleLikes в классе Card. Он берет информацию с сервера (res), обновляет массив с лайками в твоей карточке и вызывает функцию для обновления интерфейса (закрасить кнопку и отобразить количество). Ты сейчас не обновляешь массив с лайками в своей карточке, поэтому у тебя сначала делаются только PUT запросы, а после перезагрузки только DELETE

handleLikes(data) {  
 this.\_myLike = data.likes;  
 this.\_updateLikesView();  
 }

4. Метод updateLikesView довольно простой и у тебя уже реализован (называется getView, лучше переименовать, т.к. он ничего не возвращает и сейчас из названия не понятно, что он именно за лайки ответственный). Этот метод проверяет наличие твоего id внутри массива лайков (this.\_myLike.find(el => el.\_id === this.\_userId)  -> тоже лучше вынести в отдельную функцию) и соответственно обновляет кнопку и количество. У тебя уже это все есть, просто нужно убрать вызовы функций this.\_removeLike(); и this.\_toggleLike()  
5. Осталось последнее - немного поменять слушатель при нажатии на кнопку лайка. а). Нам нужно проверить, есть ли твой id в массиве с лайками (хорошо, что в пункте 4 ты сделаешь отдельную функцию для этой проверки :slightly_smiling_face:). б) если есть, то вызываем метод  this.\_removeLike(); , если нет, то this.\_toggleLike()Попробуй сделать по пунктам из этого алгоритма, понимаю, что информации много, поэтому смело задавай вопросы.  
Главное пойми, что если мы хотим внести какие то изменения на нашем сайте, после того как получим ответ от сервера, то нужно делать это в then блоке. В этом блоке мы на 100% уверены, что запрос выполнился успешно. Сейчас ты отправляешь запрос, после изменений в интерфейсе, а запрос же может выполнится с ошибкой, в таком случае получается, что ты пользователю показываешь ложную инфорамацию

https://ca.slack-edge.com/TPV9DP0N4-U01F0MC84KF-dbeaf4d2b0a4-48

[**Елена Свиридова**](https://app.slack.com/team/U01F0MC84KF)  [1 day ago](https://yandex-students.slack.com/archives/G01ENBXA9BR/p1618402714459200?thread_ts=1618387025.444700&cid=G01ENBXA9BR)

Спасибо, буду думать!

https://ca.slack-edge.com/TPV9DP0N4-U01F0MC84KF-dbeaf4d2b0a4-48

[**Елена Свиридова**](https://app.slack.com/team/U01F0MC84KF)  [19 hours ago](https://yandex-students.slack.com/archives/G01ENBXA9BR/p1618429203488900?thread_ts=1618387025.444700&cid=G01ENBXA9BR)

[@Михаил Зятьков](https://yandex-students.slack.com/team/U01EVLJ5XJM)  все получилось в итоге)у меня было две проблемы , с которыми я ещё поборолась, я не делала проверку на лайк при загрузке карточек, и я брала длину массива лайков из данных карточки а не из запроса на лайк, сдала на проверку, жду ревью!спасибо большое, без вас бы не справилась!