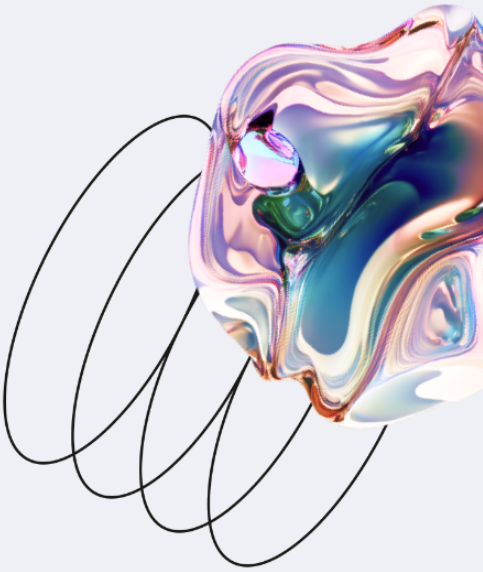
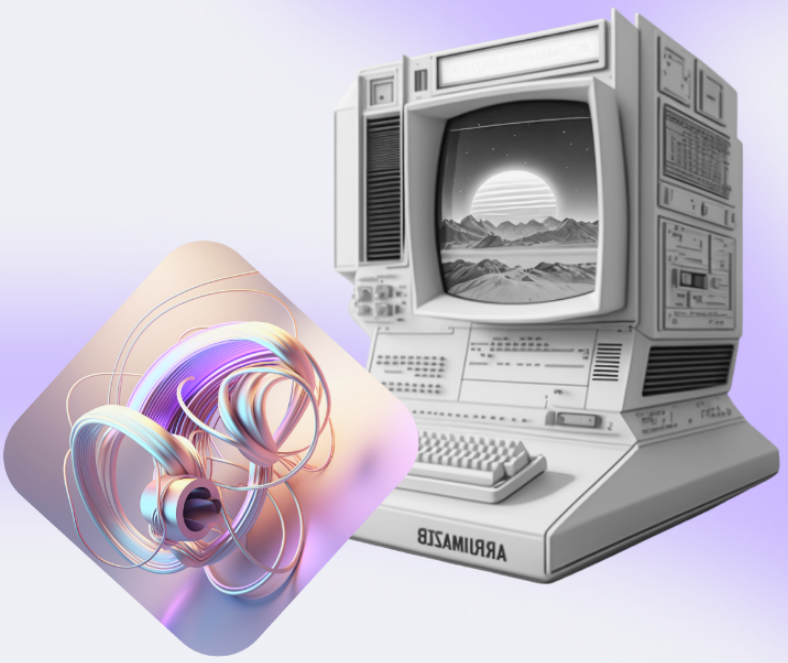
GeekBrains



Формы, сетевые запросы

JavaScript про API браузеров



Оглавление

Введение

Формы — это элементы управления, предназначенные для взаимодействия с пользователем, приёма от него данных и отправки их на сервер. В данном уроке мы изучим их подробнее для более эффективной работы с ними.

Сегодня также мы изучим работу с сетевыми запросами — это, в первую очередь, асинхронный запрос с помощью fetch() и работу с протоколом webSocket.

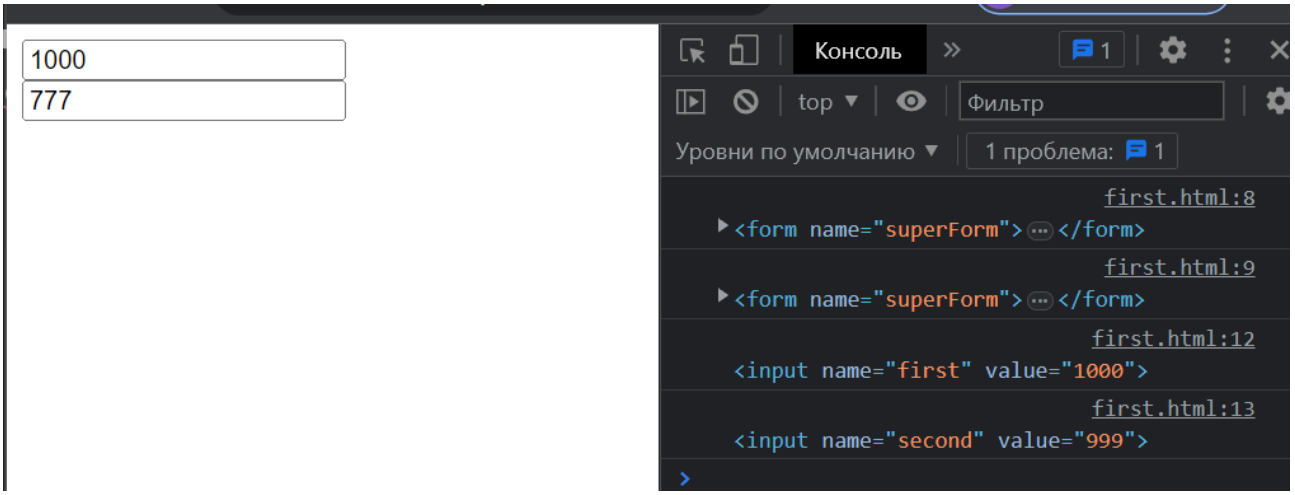
Навигация по формам

Самые часто встречающиеся в документе формы — это input, radio button и select. Чтобы они принадлежали к формам, их нужно обернуть в тег form.

Доступ к формам в документе можно получить через специальную коллекцию document.forms. Доступ к конкретной форме можно получить как через её имя, так и через её порядковый номер. А доступ к элементам формы через свойство elements по имени.

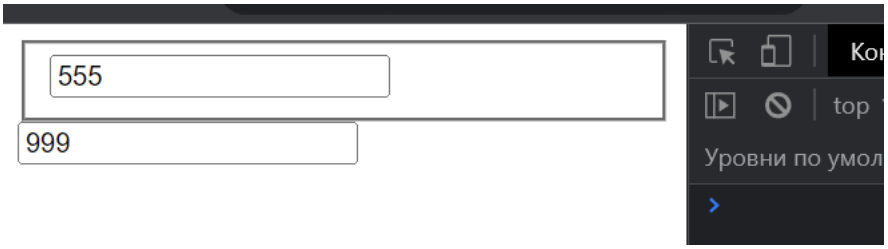
```
<html>
    <body>
        <form name="superForm">
            <input name="first" value="1000">
            <input name="second" value="999">
        </form>
        <script>
            console.log(document.forms.superForm);
            console.log(document.forms[0]);
            const form = document.forms.superForm;
            console.log(form.elements.first);
            console.log(form.elements.second);
            form.elements.second.value = 777;
        </script>
    </body>
</html>
```

В консоли будет:



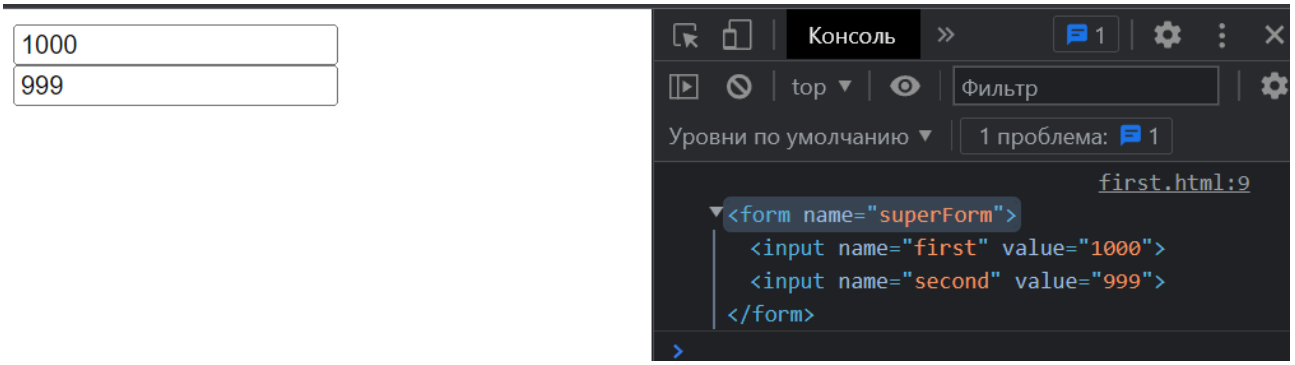
С помощью элемента fieldset можно разделить форму на несколько частей-подформы.

```html
<html>
    <body>
        <form name="superForm">
            <fieldset name="fieldSetSubForm">
                <input name="first" value="1000">
            </fieldset>
            <input name="second" value="999">
        </form>
        <script>
            const subForm = document.forms.superForm.elements.fieldSetSubForm;
            const inputFirst = subForm.elements.first;
            inputFirst.value = 555;
        </script>
    </body>
</html>
```



Ссылку на форму из элемента можно получить из свойства элемента form:

```html
<html>
    <body>
        <form name="superForm">
            <input name="first" value="1000">
            <input name="second" value="999">
        </form>
        <script>
            const inputFirst = document.forms.superForm.elements.first;
            console.log(inputFirst.form);
        </script>
    </body>
</html>
```



Описание форм можно посмотреть в официальной документации.

Элементы формы

Для большинства форм доступно строковое значение value, которое можно менять, как мы делали в примерах выше.

Для чекбоксов доступно булево значение checked.

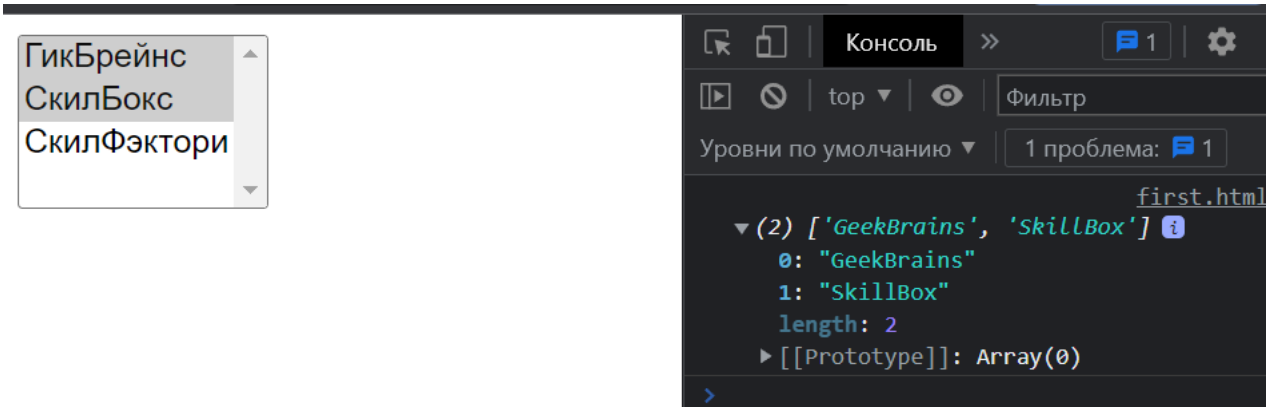
Для элемента select доступны три основные свойства. Это тот же value, который принимает значение выбранного элемента, options — это свойства, которые мы можем выбрать и selectedIndex — индекс выбранного элемента.

```html
<html>
    <body>
        <form name="superForm">
            <select name="selectForm">
                <option value="GeekBrains">ГикБрейнс</option>
                <option value="SkillBox">СкилБокс</option>
                <option value="SkillFactory">СкилФэктори</option>
            </select>
        </form>
        <script>
            const select = document.forms.superForm.selectForm;
            // Следующие три строки делают одно и то же
            select.options[2].selected = true;
            select.selectedIndex = 2;
            select.value = 'SkillFactory';
        </script>
    </body>
</html>
```

С помощью установки атрибута selected мы можем выбирать несколько вариантов:

```html
<html>
    <body>
        <form name="superForm">
            <select name="selectForm" multiple>
                <option value="GeekBrains" selected>ГикБрейнс</option>
                <option value="SkillBox" selected>СкилБокс</option>
                <option value="SkillFactory">СкилФэктори</option>
            </select>
        </form>
        <script>
            const select = document.forms.superForm.selectForm;
```

```
12              const selected = Array.from(select.options)
13                  .filter(option => option.selected)
14                  .map(option => option.value);
15              console.log(selected);
16          </script>
17      </body>
18  </html>
```



Фокусировка

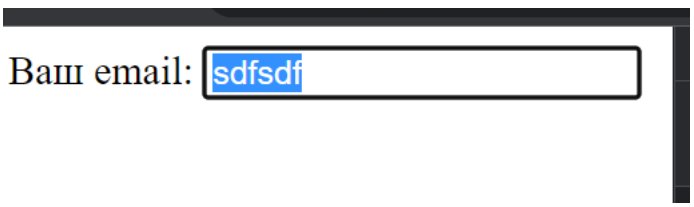
Фокус получает элемент, с которым пользователь взаимодействует в данный момент. Он устанавливается посредством клика мышью на элементе, нажатием клавиши Tab и некоторыми другими событиями. HTML-атрибут autofocus может автоматически фокусироваться на элементе при загрузке страницы.

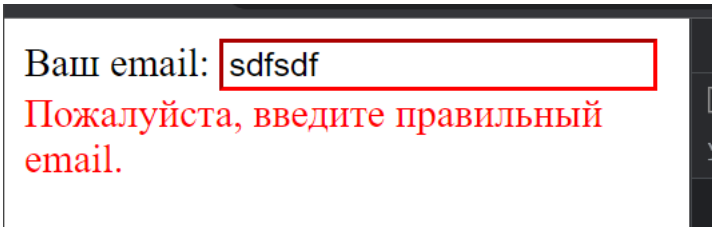
Для управления фокусировкой нам помогут события onfocus и onblur и методы focus() и blur(). Рассмотрим их подробнее.

События onfocus и onblur означают момент установки фокуса на элементе и его потерю. Эти события не всплывают. Но если требуется использовать всплытие, можно использовать аналогичные события focusin и focusout.

```
1  <html>
2      <body>
3          <style>
4              .invalid { border-color: red; }
5              #error { color: red }
```

```
        </style>
        <form name="superForm">
            Ваш email: <input type="email" name="input">
            <div id="error"></div>
        </form>
        <script>
            const input = document.forms.superForm.input;
            const error = document.getElementById('error');
            input.onblur = function() {
                if (!input.value.includes('@')) { // не email
                input.classList.add('invalid');
                error.innerHTML = 'Пожалуйста, введите правильный email.'
                }
            };
            input.onfocus = function() {
                if (this.classList.contains('invalid')) {
                this.classList.remove('invalid');
                error.innerHTML = "";
                }
            };
          </script>
    </body>
</html>
```





Здесь при установке фокуса на элемент мы очищаем элемент с описанием ошибки, так как пользователь что-то вводит. При потере фокуса считаем, что ввод окончен и мы можем проводить валидацию написанного.

Методами focus() и blur() можно принудительно вернуть фокус или снять его с формы. Например, при попытке отправки формы с ошибкой, можно вернуть фокус в первое поле с ошибкой, чтобы пользователю было проще её найти и исправить.

События изменения данных форм

Input

Событие input происходит при каждом изменении данных в форме. Происходит при любом изменении: вставке текста, голосовом вводе и так далее.

Change

Событие change происходит после ввода данных в элементе формы. В отличие от события input, change в элементе <input> происходит после окончания изменений, то есть, при потере фокуса. В остальных элементах происходит сразу.

Cut / copy / paste

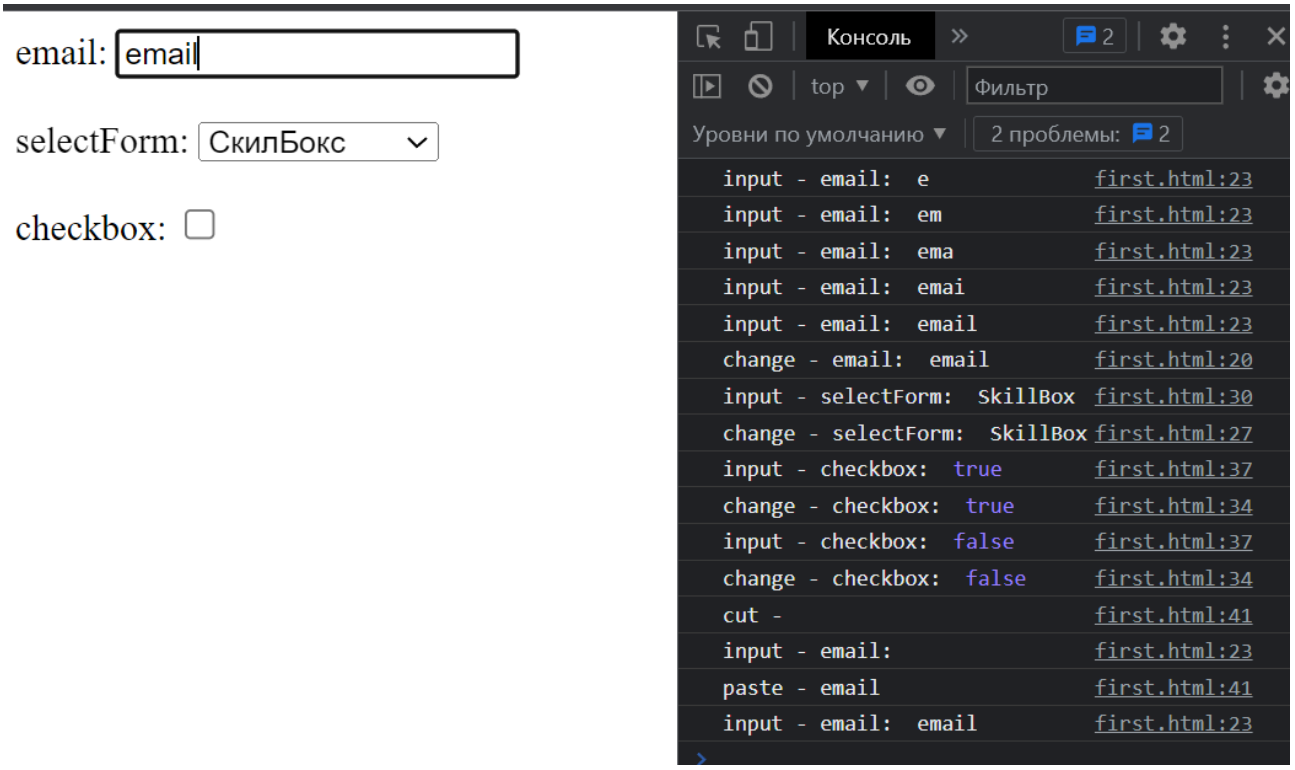
События вырезания содержимого, копирования и вставки. Для этих событий, в отличие от input и change, сработает метод preventDefault() — там он не сможет предотвратить изменения, так как сработает поздно.

Пример с работой рассмотренных событий:

```
<html>
    <body>
        <form name="superForm">
            email: <input type="email" name="email">
            <br />
            <br />
            selectForm: <select name="selectForm">
                <option value="GeekBrains">ГикБрейнс</option>
                <option value="SkillBox">СкилБокс</option>
```

```
10                <option value="SkillFactory">СкилФэктори</option>
11            </select>
12            <br />
13            <br />
14            checkbox: <input type="checkbox" name="checkbox">
15        </form>
16        <script>
17            const form = document.forms.superForm;
18            form.email.onchange = () => {
19                console.log('change -  email: ', form.email.value);
20            }
21            form.email.oninput = () => {
22                console.log('input -  email: ', form.email.value);
23            }
24            form.selectForm.onchange = () => {
25                console.log('change -  selectForm: ', form.selectForm.value);
26            }
27            form.selectForm.oninput = () => {
28                console.log('input -  selectForm: ', form.selectForm.value);
29            }
30            form.checkbox.onchange = () => {
31                console.log('change -  checkbox: ', form.checkbox.checked);
32            }
33            form.checkbox.oninput = () => {
34                console.log('input -  checkbox: ', form.checkbox.checked);
35            }
36            form.email.oncut = form.email.oncopy = form.email.onpaste = (event)
  => {
37                console.log(event.type + ' -  ' +
  event.clipboardData.getData('text/plain'));
38            }
```

```
39        </script>
40    </body>
41 </html>
```



Видим срабатывание события input при вводе каждого символа, после потери фокуса происходит change. Input срабатывает также после событий cut, copy и paste.

Для чекбокса выведи атрибут checked, так как в этом случае value не содержит ничего информативного.

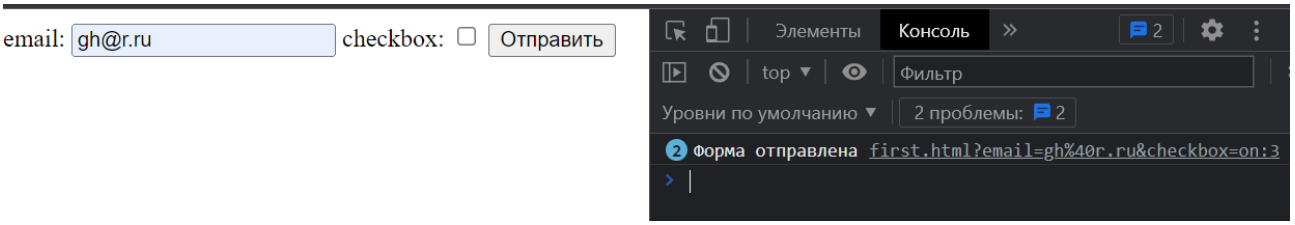
Отправка формы

Есть три основных способа отправить форму на сервер. Во-первых, можно в форме сделать элемент input с типом submit или image. Появится кнопка с надписью по умолчанию «Отправить» при нажатии на которую форма отправится. Во-вторых, можно кликнуть на любое поле формы и нажать Enter. В-третьих, можно отправить форму из скрипта, использовав метод form.submit().

При отправке формы генерируется событие формы submit, которое мы можем обработать.

Напишем пример, в котором при нажатии на чекбокс у нас тоже будет происходить отправка формы. Единственное, здесь не получится предотвратить действие по умолчанию.

```
<html>
    <body>
        <form name="superForm" onsubmit="console.log('Форма отправлена');
return false">
            email: <input type="email" name="email">
            checkbox: <input type="checkbox" name="checkbox">
            <input type="submit">
        </form>
        <script>
            const form = document.forms.superForm;
            form.checkbox.oninput = () => {
                form.submit();
            }
        </script>
    </body>
</html>
```



Обратите внимание, что в консоли теперь пишется, кроме названия вызвавшего скрипта, ещё и данные формы, представленные в виде GET-запроса. И об этом мы поговорим далее.

Взаимодействие с сервером

Теперь, когда мы изучили работу с формами, нам надо как-то взаимодействовать с API-сервером для предзаполнения форм и для отправки данных для обработки.

Информация должна храниться, обрабатываться и выдаваться по нашим запросам, генерироваться специально для нас.

Ранее мы уже изучали работу XMLHttpRequest. На его основе работали AJAX-запросы (Asynchronous JavaScript And XML). Как мы видим по аббревиатуре, стандарты разрабатывались, когда был популярен формат XML. Но те времена уже в прошлом и сам XML уже редко встретишь в проектах — гораздо чаще пользуются форматом JSON. И сам XMLHttpRequest уже практически не используется в чистом виде: есть более удобный и лаконичный fetch. Хотя, некоторые функции XMLHttpRequest ему недоступны, но они требуются нечасто.

Есть и другие возможности взаимодействия с сервером. Например, есть протокол WebSocket, позволяющий серверу инициировать отправку сообщений. Используется он реже, так как создаёт определённую нагрузку на сетевую инфраструктуру. Но в приложениях, требующих отступить от привычной системы запрос-ответ, таких как, например, чаты, он незаменим.

Fetch

Метод fetch является частью BOM (BrowserObjectModel) и не определяется стандартами ECMAScript. Он определяется стандартами WHATWG и W3C. Но так как он должен входить в состав языка, он тоже должен быть в ECMAScript. Впервые он появился в ECMAScript7 в 2016 году. Браузеры, выпущенные ранее, его не поддерживают. В JavaScript fetch является промисом, но возможностей у него больше, чем могут предоставить промисы.

В основе работы fetch используется архитектурный стиль взаимодействия компонентов распределённого приложения в сети, именуемый REST. Методы REST, являющиеся частью HTTP-протокола, используются в его параметрах, поэтому с ними нужно предварительно ознакомиться.

Итак, формат запроса:

```
1 const url = 'https://gb.ru';
2 const options = {
3   method: 'GET',
4   headers: {},
5   body: ''
6 }
7 let promise = fetch(url, options);
```

Fetch — это метод, принимающий один или два параметра: обязательный параметр url — строка с адресом сервера, на который идёт запрос и необязательный параметр options — объект, в котором задаются параметры запроса, такие как метод, заголовки и тело запроса.

Если объект options не задан, то это обычный GET-запрос, скачивающий данные с адреса, указанного в url. Данные нам поступают в два этапа. Сначала нам приходит асинхронный ответ в виде встроенного объекта Response, в котором мы можем посмотреть код статуса HTTP. В классе Response есть несколько методов, позволяющие получить асинхронное тело ответа. В итоге, чтобы получить тело ответа, нам нужно выполнить два асинхронных запроса.

Возьмём бесплатный API-сервер для проверки работы fetch. Пусть это будут факты о кошечках https://catfact.ninja. Для правильной работы схемы async/await используем самовызывающуюся функцию.

```
1 'use strict';
2 (async () => {
3   const url = 'https://catfact.ninja/fact';
4   const response = await fetch(url);
5   console.log(response.status);
6   console.log(response.ok);
7   const fact = await response.json();
8   console.log(fact);
9 })();
```

```
200                                                    first.js:5
true                                                   first.js:6
                                                       first.js:8
▼ {fact: 'Cats only use their meows to talk to humans, not e… when they are kittens to si
  gnal to their mother.', length: 170} i
    fact: "Cats only use their meows to talk to humans, not each other. The only time they
    length: 170
  ▶ [[Prototype]]: Object
>
```

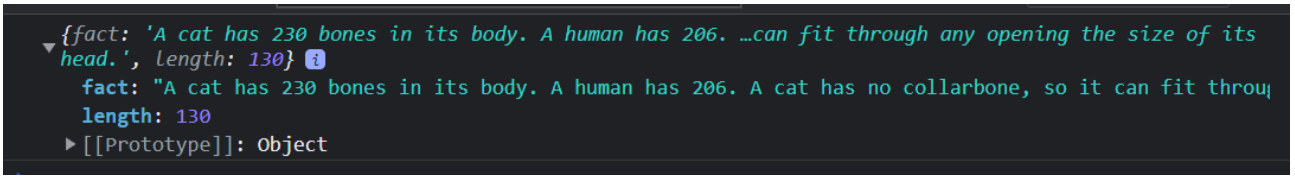
response.status возвращает нам HTTP-код ответа сервера. Просмотр кода в ответе помогает нам понять, что сайт доступен и нет ошибок сети. Наличие кода даже со статусами ошибок сервера (диапазон 500-599) означает, что сайт доступен. Свойство ok означает, что response.status находится в диапазоне 200-299.

Для представления тела ответа нам доступны несколько методов response. Они возвращают ответ каждый в своём формате:

- response.json() — JSON
- response.text() — текст
- response.blob() — Blob (бинарные данные с типом)
- response.arrayBuffer() — ArrayBuffer (низкоуровневое представление бинарных данных)
- response.formData() — объект FormData
- response.body — объект ReadableStream, с помощью которого можно считывать тело запроса по частям
- response.headers — коллекция заголовков ответов, похожая по использованию на Map

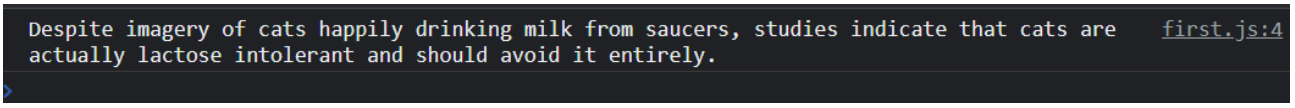
Перепишем тот же запрос в более читабельный вид с помощью промисов:

```
1 'use strict';
2 fetch('https://catfact.ninja/fact')
3   .then(response ⇒ response.json())
4   .then(console.log);
```



Или для вывода только факта:

```
1 'use strict';
2 fetch('https://catfact.ninja/fact')
3   .then(response ⇒ response.json())
4   .then(obj ⇒ console.log(obj.fact));
```



На самом деле всё оказалось проще, чем казалось, правда?

Разберём объект с параметрами запроса. В качестве заголовков можно устанавливать свои заголовки запроса, кроме:

- `Accept-Charset`
- `Accept-Encoding`
- `Access-Control-Request-Headers`
- `Access-Control-Request-Method`
- `Connection`
- `Content-Length`
- `Cookie`
- `Cookie2`
- `Date`
- `DNT`
- `Expect`
- `Host`
- `Keep-Alive`
- `Origin`
- `Referer`
- `Set-Cookie`
- `TE`
- `Trailer`
- `Transfer-Encoding`
- `Upgrade`
- `Via`
- `Proxy-`
- `Sec-`
- `X-HTTP-Method`

- `X-HTTP-Method-Override`
- `X-Method-Override`

Данный список взят из спецификации. Эти заголовки контролируются полностью браузером и служат для корректной работы протокола HTTP и обеспечения целостности данных.

Метод можем указывать любой, соответствующий стандарту. В тело для отправки данных мы можем добавить что-то одно из следующего списка:

- Строка
- Blob
- BufferSource
- FormData
- URLSearchParams

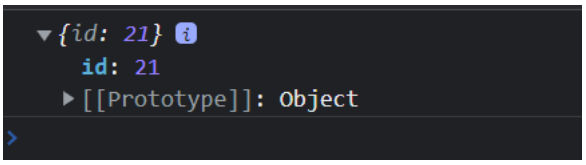
С помощью Blob или BufferSource мы можем отправлять двоичные файлы, например, картинки. Самое частое использование — строка, мы можем отправить любой текст, в том числе, и JSON. С помощью объекта FormData можно отправить данные формы как form/multipart. URLSearchParams используется редко, отправляет данные в кодировке x-www-form-urlencoded.

Используем API с сайта https://fakestoreapi.com для генерации POST-запроса:

```
'use strict';
fetch('https://fakestoreapi.com/products',{
            method: "POST",
            body: JSON.stringify(
                {
                    title: 'test product',
                    price: 13.5,
                    description: 'lorem ipsum set',
                    image: 'https://i.pravatar.cc',
                    category: 'electronic'
                }
            )
```

```
13        })
14            .then(res⇒res.json())
15            .then(json⇒console.log(json));
```

Этот API нам вернул id нового созданного объекта:



FormData

FormData — очень удобная вещь для автоматической генерации готового объекта из формы и отправкой её посредством сетевых запросов на сервер. Генерируется она с помощью конструктора:

```
const formData = new FormData(form);
```

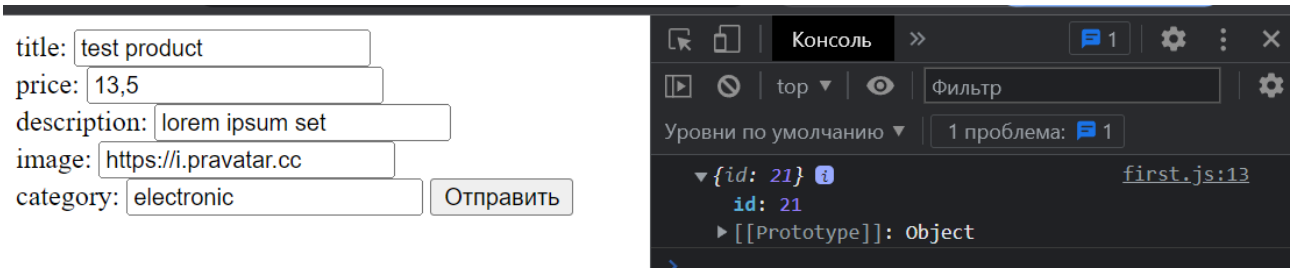
Вообще, параметр в конструкторе необязателен. Но если мы передадим туда элемент формы, то форма прикрепится к телу запроса и будет отправлена в нужном виде на сервер. При этом установится заголовок «Content-Type: multipart/form-data». По предыдущему примеру создадим форму и отправим её на сервер:

```
<html>
    <body>
        <form name="superForm">
            title: <input name="title" value="test product">
            <br />
            price: <input name="price" type="number" value="13.5">
            <br />
            description: <input name="description" value="lorem ipsum set">
            <br />
```

```
            image: <input name="image" value="https://i.pravatar.cc">
            <br />
            category: <input name="category" value="electronic">
            <input type="submit">
        </form>
        <script defer src="first.js"></script>
    </body>
</html>
```

```
'use strict';
const form = document.forms.superForm;
form.onsubmit = async (e) => {
  e.preventDefault();
  let response = await fetch('https://fakestoreapi.com/products', {
    method: 'POST',
    body: new FormData(form)
  });
  let result = await response.json();
  console.log(result);
};
```

В итоге у нас отобразится:



Данные мы сразу подставили в значения value, но их можно поменять по своему усмотрению. Сервер принял запрос как валидный и выдал id нового элемента.

Сам класс FormData живой, и мы можем управлять из скрипта его содержимым. Методы класса FormData:

- formData.delete(name) — удаляет поле с именем name
- formData.get(name) — возвращает значение поля с именем name
- formData.has(name) — если существует поле с именем name, то возвращает true, иначе false

Для вставки содержимого существуют методы set и append, они полностью одинаковы в использовании, за исключением одного нюанса: append может добавить несколько полей с одним и тем же именем, а set сначала удалит предыдущие вхождения послей с этим именем, а потом добавит новое. Использовать их можно двумя способами — с двумя аргументами при обычном поле, с тремя аргументами, если тип данных — файл.

- formData.append(name, value) — добавляет к объекту поле с именем name и значением value
- formData.append(name, blob, fileName) — добавляет поле с файлом, который записывается во второй аргумент, третий аргумент задаёт имя файла

Работа с fetch

Процесс получения данных

Fetch позволяет отобразить процесс получения данных. На данный момент показ процесса отправки fetch не поддерживает, но для этого можно обратиться к XMLHttpRequest.

Для получения процесса отправки существует специальный «поток для чтения», который описан в соответствующей спецификации. Он содержится в свойстве response.body.

```
1 const reader = response.body.getReader();
2 while(true) {
3   const {done, value} = await reader.read();
4   if (done) break;
5   console.log(`Получено ${value.length} байт`)
6 }
```

Мы делаем читателя с помощью метода response.body.getReader. Потом запускаем бесконечный цикл и каждый раз читаем его состояние с помощью метода read. Его состояние деструктуризируем в переменные done и value, куда записывается, окончен ли цикл чтения и массив Uint8Array с данными ответа соответственно. Если значение done становится равно true, то цикл чтения закончен и наш цикл запросов завершается.
Рабочий пример с отображением длины загружаемых данных:

```
(async () => {
  // Шаг 1: начинаем загрузку fetch, получаем поток для чтения
  let response = await fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits?per_page=100');
  const reader = response.body.getReader();
  // Шаг 2: считываем данные:
  let receivedLength = 0; // количество байт, полученных на данный момент
  let chunks = []; // массив полученных двоичных фрагментов (составляющих тело ответа)
  while(true) {
    const {done, value} = await reader.read();
    if (done) break;
    chunks.push(value);
    receivedLength += value.length;
    console.log(`Получено ${receivedLength} байт`)
  }
  // Шаг 3: соединим фрагменты в общий типизированный массив Uint8Array
  let chunksAll = new Uint8Array(receivedLength);
  let position = 0;
  for(let chunk of chunks) {
    chunksAll.set(chunk, position);
    position += chunk.length;
  }
  // Шаг 4: декодируем Uint8Array обратно в строку
  let result = new TextDecoder("utf-8").decode(chunksAll);
```

```
24   // Готово!
25   let commits = JSON.parse(result);
26   console.log(commits);
27 })();
```

```
Получено 86107 байт                                                    first.js:19
Получено 151643 байт                                                   first.js:19
Получено 300719 байт                                                   first.js:19
Получено 366255 байт                                                   first.js:19
Получено 431791 байт                                                   first.js:19
Получено 471927 байт                                                   first.js:19
                                                                       first.js:35
  (100) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  ▶ {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}]
>
```

Прерывание запроса

Для отмены fetch и других асинхронных запросов существует специальный класс AbortController с одним свойством signal и тоже одним методом abort(). Для fetch в параметрах указывается свойство signal, в который мы записываем контроллер, созданный с помощью конструктора класса AbortController. Далее мы вызываем в нужном месте метод abort(). И всё, работа fetch прервана.

В свойство controller.signal.aborted записывается булево значение, прерван ли запрос.

```
1 'use strict';
2 const controller = new AbortController();
3 fetch(url, {
4   signal: controller.signal
5 });
6 controller.abort();
```

```
▶ Uncaught (in promise) DOMException: The user aborted a request.  first.js:8
>
```

Для использования в других асинхронных задачах можно применить addEventListener, который вешается на signal:

```
'use strict';
const controller = new AbortController();
const signal = controller.signal;
// срабатывает при вызове controller.abort()
signal.addEventListener('abort', () ⇒ console.log("отмена!"));
controller.abort(); // отмена
console.log(signal.aborted); // true
```

Кроссдоменные запросы

Поговорим подробнее о политике совместного использования ресурсов между разными источниками (CORS). CORS, расшифровывается как Cross-Origin Resource Sharing, определяет эту политику с помощью специальных заголовков. Чаще всего эта политика запрещает загружать какие-либо ресурсы с других доменов в целях безопасности: злоумышленники могут просто обратиться к нашему серверу и запросто получить несанкционированный доступ к данным.

Раньше, на заре развития JavaScript, такие вызовы были вообще запрещены. Но тогда бы мы не смогли работать с примерами выше, с бесплатными API-серверами. Поэтому появились настройки для сервера, позволяющие регулировать данное ограничение.

Простыми запросами называются запросы методами HEAD, GET или POST и которые могут иметь только такие заголовки:

- Accept
- Accept-language
- Content-language
- Content-type со значениями «application/x-www-form-urlencoded», «multipart/form-data» или «text/plain»

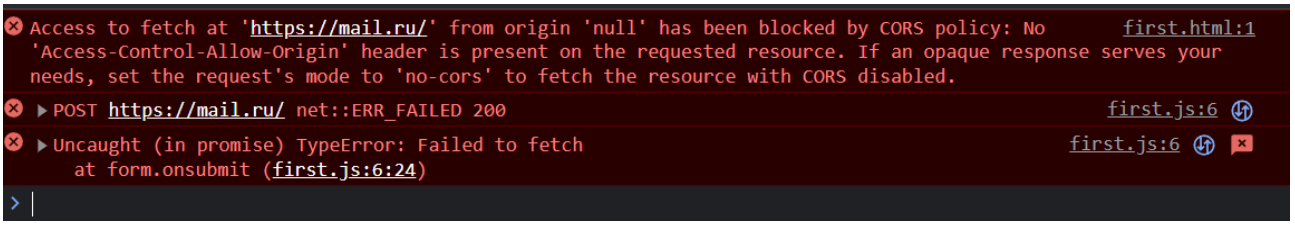
Простой запрос может быть сделан из скрипта или формы без каких-то специальных методов. Для остальных запросов отсылается предварительный запрос на сервер (по-английски называется «preflight») для того, чтобы спросить у

сервера, может ли он принять этот запрос. И если сервер явно не указывает, что может, то запрос блокируется.

Браузер для запросов на сервер играет роль доверенного посредника. Он устанавливает заголовки Origin с указанием источника, внутри которого может быть указан домен, поддомен и порт. Сервер проверяет этот заголовок. И если заголовок подходит под разрешённые, то сервер ставит свой заголовок Access-Control-Allow-Origin в ответ. В нём указываются только разрешённые Origin, либо знак «*», если разрешаются любые. Если заголовок не подходит, происходит блокировка запроса.

Поменяем в примере с отправкой формы адрес запроса на mail.ru.

В консоли будет:



То же самое было, когда мы пытались подключить модули. И в том случае тоже проверяются заголовки ответа.

WebSocket

Протокол WebSocket — это особый протокол передачи данных в сети. Он создаёт постоянное соединение между сервером и клиентом и позволяет инициировать передачу данных сервером. С его помощью легко реализуются чаты и онлайн-игры, для которых требуется постоянное соединение.

Для работы с WebSocket существует специальный класс WebSocket, у которого есть два метода и четыре события для работы, ничего сложного.

Вначале нужно создать экземпляр класса через конструктор, в параметре которого нужно указать url с указанием специального протокола ws или wss — это ws с использованием SSL-сертификата:

```
1 const webSocket = new WebSocket('wss://gb.ru');
```

После создания экземпляра класса можно слушать события, которых всего 4:

- open — происходит при открытии соединения
- close — происходит при закрытии соединения
- error — происходит при ошибке
- message — пришло сообщение с данными

Сразу, как экземпляр создан, он начинает устанавливать соединение. Для этого браузер сначала отправляет на сервер специальный http-запрос на сервер на предмет поддержки сервером данного соединения. Если сервер отвечает положительно, то далее идёт общение по протоколу WebSocket.

Мы можем отправить сообщения на сервер с помощью метода send(). Сообщения разбиваются на фрагменты, которые могут быть одним из следующих типов:

- Текстовые
- Бинарные
- Пинг
- Фрейм закрытия
- Служебные

Текстовые и бинарные содержат соответствующие данные, отправляемые на сервер. Пинги нужны для проверки соединения, на них сервер отвечает автоматически.

Бинарные данные могут быть отправлены в формате Blob или ArrayBuffer.

С помощью метода close мы можем закрыть соединение.

Заключение

В заключительном уроке нашего курса мы рассмотрели, как работать с формами и сетевыми запросами в JavaScript. Теперь вы полностью готовы к тому, чтобы полноценно написать свою первую веб-страницу. Формы позволяют удобно передать данные от клиента, а сетевые запросы помогут связаться с сервером для получения и отправки данных.