



Формы, сетевые запросы

JavaScript про API браузеров



Оглавление

Введение	3
Навигация по формам	3
Элементы формы	5
Фокусировка	7
События изменения данных форм	9
Input	9
Change	9
Cut / copy / paste	9
Отправка формы	11
Взаимодействие с сервером	12
Fetch	13
FormData	18
Работа с fetch	20
Процесс получения данных	20
Прерывание запроса	22
Кроссдоменные запросы	23
WebSocket	24
Заключение	25

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`.

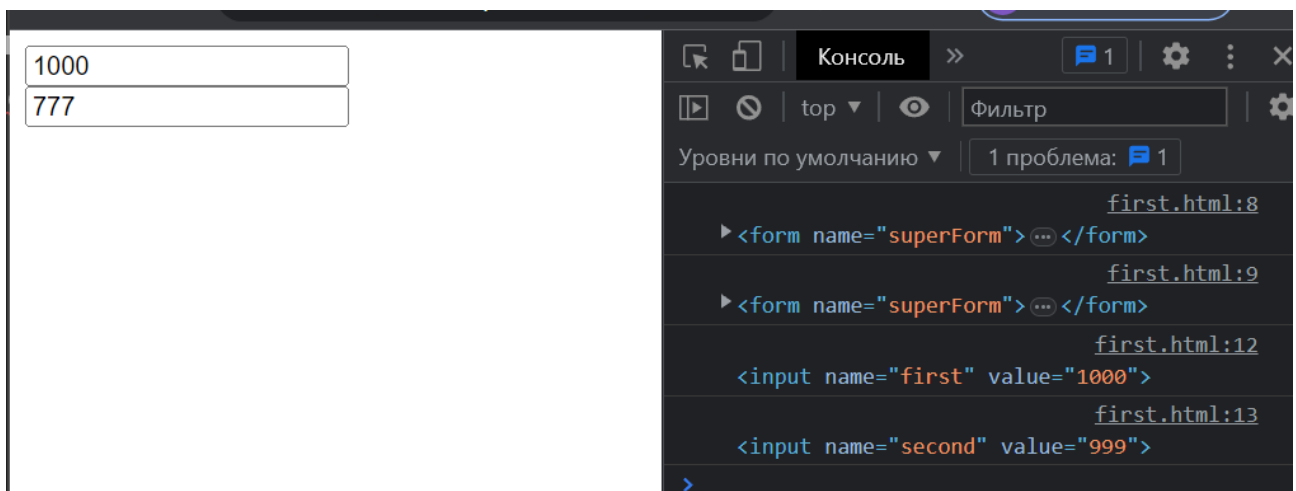
Навигация по формам

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.

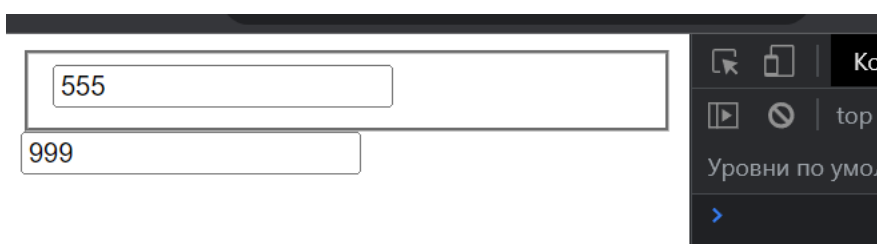
```
1 <html>
2   <body>
3     <form name="superForm">
4       <input name="first" value="1000">
5       <input name="second" value="999">
6     </form>
7     <script>
8       console.log(document.forms.superForm);
9       console.log(document.forms[0]);
10      const form = document.forms.superForm;
11      console.log(form.elements.first);
12      console.log(form.elements.second);
13      form.elements.second.value = 777;
14    </script>
15  </body>
16 </html>
```

В консоли будет:



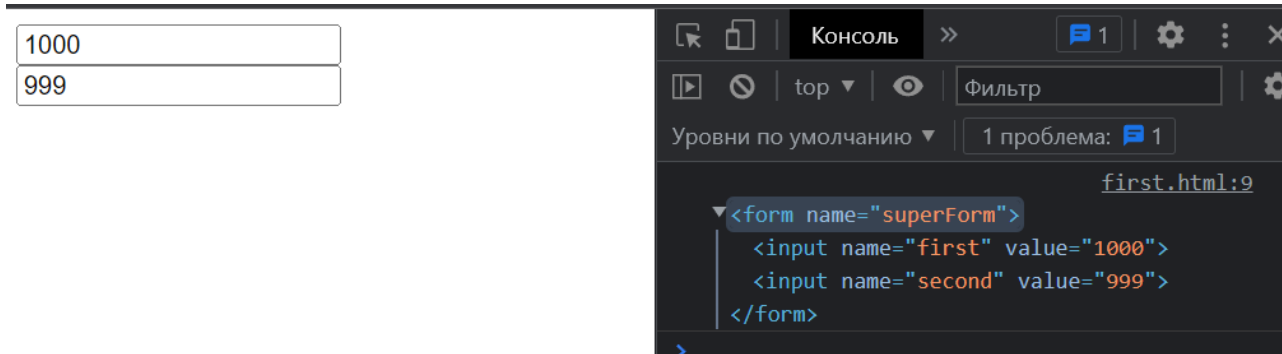
С помощью элемента `fieldset` можно разделить форму на несколько частей-подформы.

```
1 <html>
2   <body>
3     <form name="superForm">
4       <fieldset name="fieldSetSubForm">
5         <input name="first" value="1000">
6       </fieldset>
7       <input name="second" value="999">
8     </form>
9     <script>
10      const subForm = document.forms.superForm.elements.fieldSetSubForm;
11      const inputFirst = subForm.elements.first;
12      inputFirst.value = 555;
13    </script>
14  </body>
15 </html>
```



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

```
1 <html>
2   <body>
3     <form name="superForm">
4       <input name="first" value="1000">
5       <input name="second" value="999">
6     </form>
7     <script>
8       const inputFirst = document.forms.superForm.elements.first;
9       console.log(inputFirst.form);
10    </script>
11  </body>
12 </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.

```

1 <html>
2   <body>
3     <form name="superForm">
4       <select name="selectForm">
5         <option value="GeekBrains">ГикБрейнс</option>
6         <option value="SkillBox">СкилБокс</option>
7         <option value="SkillFactory">СкилФэктори</option>
8       </select>
9     </form>
10    <script>
11      const select = document.forms.superForm.selectForm;
12      // Следующие три строки делают одно и то же
13      select.options[2].selected = true;
14      select.selectedIndex = 2;
15      select.value = 'SkillFactory';
16    </script>
17  </body>
18 </html>

```

С помощью установки атрибута `selected` мы можем выбирать несколько вариантов:

```

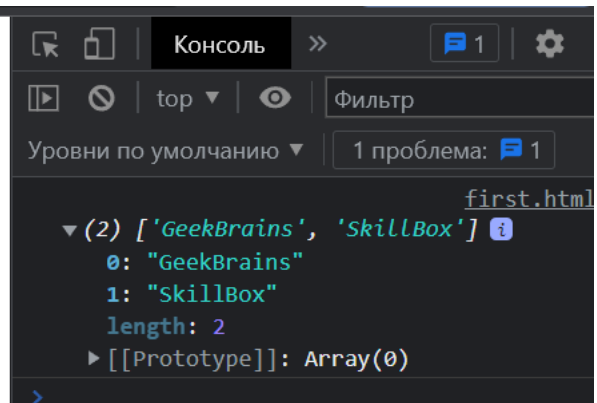
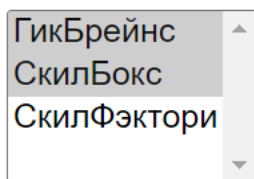
1 <html>
2   <body>
3     <form name="superForm">
4       <select name="selectForm" multiple>
5         <option value="GeekBrains" selected>ГикБрейнс</option>
6         <option value="SkillBox" selected>СкилБокс</option>
7         <option value="SkillFactory">СкилФэктори</option>
8       </select>
9     </form>
10    <script>
11      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 }

```

```

6      </style>
7      <form name="superForm">
8          Ваш email: <input type="email" name="input">
9              <div id="error"></div>
10     </form>
11     <script>
12         const input = document.forms.superForm.input;
13         const error = document.getElementById('error');
14         input.onblur = function() {
15             if (!input.value.includes('@')) { // не email
16                 input.classList.add('invalid');
17                 error.innerHTML = 'Пожалуйста, введите правильный email.'
18             }
19         };
20         input.onfocus = function() {
21             if (this.classList.contains('invalid')) {
22                 this.classList.remove('invalid');
23                 error.innerHTML = "";
24             }
25         };
26     </script>
27 </body>
28 </html>

```

Ваш email:

Ваш email:

Пожалуйста, введите правильный email.

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:

```
1 <html>
2   <body>
3     <form name="superForm">
4       email: <input type="email" name="email">
5     <br />
6     <br />
7     selectForm: <select name="selectForm">
8       <option value="GeekBrains">ГикБрейнс</option>
9       <option value="SkillBox">СкилБокс</option>
```

```

10         <option value="SkillFactory">СкиллФэктори</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     }

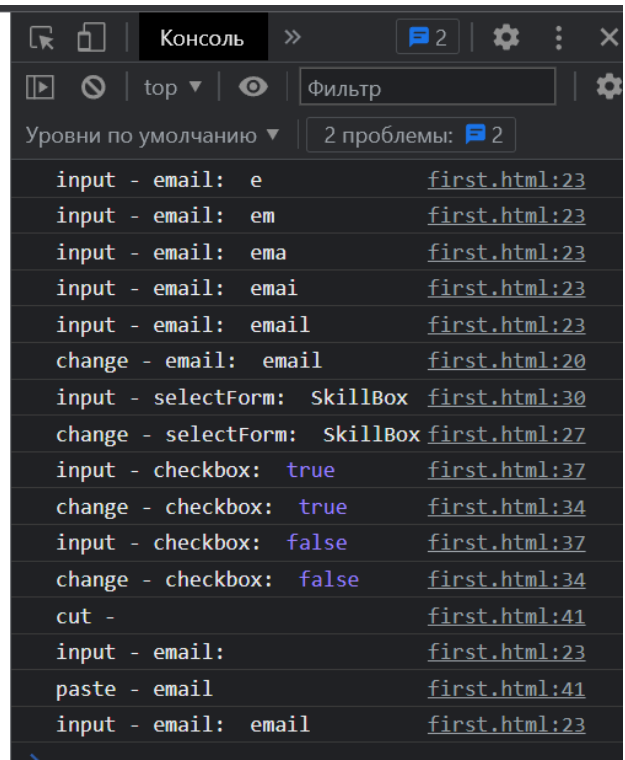
```

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

email:

selectForm:

checkbox: ☐



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

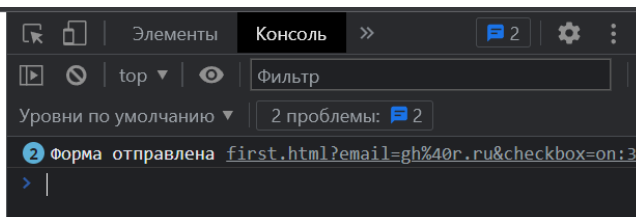
Есть три основных способа отправить форму на сервер. Во-первых, можно в форме сделать элемент input с типом submit или image. Появится кнопка с надписью по умолчанию «Отправить» при нажатии на которую форма отправится. Во-вторых, можно кликнуть на любое поле формы и нажать Enter. В-третьих, можно отправить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

```
1 <html>
2   <body>
3     <form name="superForm" onsubmit="console.log('Форма отправлена');
      return false">
4       email: <input type="email" name="email">
5       checkbox: <input type="checkbox" name="checkbox">
6       <input type="submit">
7     </form>
8     <script>
9       const form = document.forms.superForm;
10      form.checkbox.oninput = () => {
11        form.submit();
12      }
13    </script>
14  </body>
15 </html>
```

email: checkbox: ☐



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 самовывзывающуюся 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
{fact: 'Cats only use their meows to talk to humans, not e... when they are kittens to si
  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);
```

```
{fact: 'A cat has 230 bones in its body. A human has 206. ...can fit through any opening the size of its head.', length: 130}
  fact: "A cat has 230 bones in its body. A human has 206. A cat has no collarbone, so it can fit throug
  length: 130
  ► [[Prototype]]: Object
```

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));
```

```
Despite imagery of cats happily drinking milk from saucers, studies indicate that cats are actually lactose intolerant and should avoid it entirely. first.js:4
```

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

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

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

- o `X-HTTP-Method-Override`
- o `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:

```
1 'use strict';
2 fetch('https://fakestoreapi.com/products',{
3     method: "POST",
4     body: JSON.stringify(
5         {
6             title: 'test product',
7             price: 13.5,
8             description: 'lorem ipsum set',
9             image: 'https://i.pravatar.cc',
10            category: 'electronic'
11        }
12    )
13 })
```

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

Этот API нам вернул id нового созданного объекта:

```
▼ {id: 21} ⓘ  
  id: 21  
  ► [[Prototype]]: Object  
>
```

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:

```
1 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:

```
1 <html>  
2   <body>  
3     <form name="superForm">  
4       title: <input name="title" value="test product">  
5       <br />  
6       price: <input name="price" type="number" value="13.5">  
7       <br />  
8       description: <input name="description" value="lorem ipsum set">  
9       <br />
```

```

10         image: <input name="image" value="https://i.pravatar.cc">
11         <br />
12         category: <input name="category" value="electronic">
13         <input type="submit">
14     </form>
15     <script defer src="first.js"></script>
16 </body>
17 </html>

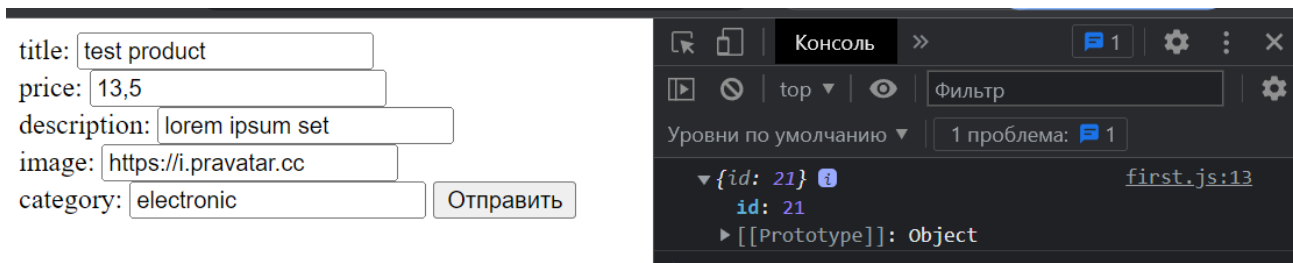
```

```

1 'use strict';
2 const form = document.forms.superForm;
3 form.onsubmit = async (e) => {
4     e.preventDefault();
5     let response = await fetch('https://fakestoreapi.com/products', {
6         method: 'POST',
7         body: new FormData(form)
8     });
9     let result = await response.json();
10    console.log(result);
11 };

```

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 полей 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 деструктурируем 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:

```
1 (async () => {
2   // Шаг 1: начинаем загрузку fetch, получаем поток для чтения
3   let response = await fetch('https://api.github.com/repos/javascript-
      tutorial/en.javascript.info/commits?per_page=100');
4   const reader = response.body.getReader();
5   // Шаг 2: считываем данные:
6   let receivedLength = 0; // количество байт, полученных на данный момент
7   let chunks = []; // массив полученных двоичных фрагментов (составляющих тело
      ответа)
8   while(true) {
9     const {done, value} = await reader.read();
10    if (done) break;
11    chunks.push(value);
12    receivedLength += value.length;
13    console.log(`Получено ${receivedLength} байт`)
14  }
15  // Шаг 3: соединим фрагменты в общий типизированный массив Uint8Array
16  let chunksAll = new Uint8Array(receivedLength);
17  let position = 0;
18  for(let chunk of chunks) {
19    chunksAll.set(chunk, position);
20    position += chunk.length;
21  }
22  // Шаг 4: декодируем Uint8Array обратно в строку
23  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`:

```
1 'use strict';
2 const controller = new AbortController();
3 const signal = controller.signal;
4 // срабатывает при вызове controller.abort()
5 signal.addEventListener('abort', () => console.log("отмена!"));
6 controller.abort(); // отмена
7 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:

```
✖ Access to fetch at 'https://mail.ru/' from origin 'null' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled. first.html:1
✖ ▶ POST https://mail.ru/ net::ERR_FAILED 200 first.js:6
✖ ▶ Uncaught (in promise) TypeError: Failed to fetch first.js:6
   at form.onsubmit (first.js:6:24)
```

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.