JavaScript в браузере finish

Сетевые запросы
и
Shadow-DOM

Сетевые запросы

- FormData
- Fetch: ход загрузки
- Fetch: запросы на другие сайты (CORS)
- Объекты URL
- XMLHttpRequest
- WebSocket

FormData

```
let formData = new FormData([form]);
```

Если передать в конструктор элемент HTML-формы form, то создаваемый объект автоматически прочитает из неё поля.

Его особенность заключается в том, что методы для работы с сетью, например fetch, позволяют указать объект FormData в свойстве тела запроса body.

Он будет соответствующим образом закодирован и отправлен с заголовком Content-Type: multipart/form-data.

То есть, для сервера это выглядит как обычная отправка формы.

Отправка простой формы

```
<form id="formElem">
  <input type="text" name="name" value="John">
  <input type="text" name="surname" value="Smith">
  <input type="submit">
</form>

<script>
  formElem.onsubmit = async (e) => {
    e.preventDefault();

    let response = await fetch('/article/formdata/post/user', {
      method: 'POST',
      body: new FormData(formElem)
    });

    let result = await response.json();

    alert(result.message);
  };
</script>
```

Методы объекта FormData

С помощью указанных ниже методов мы можем изменять поля в объекте FormData:

- formData.append(name, value) – добавляет к объекту поле с именем name и значением value,
- formData.append(name, blob, fileName) – добавляет поле, как будто в форме имеется элемент <input type="file">, третий аргумент fileName устанавливает имя файла (не имя поля формы), как будто это имя из файловой системы пользователя,
- formData.delete(name) – удаляет поле с заданным именем name,
- formData.get(name) – получает значение поля с именем name,
- formData.has(name) – если существует поле с именем name, то возвращает true, иначе false

```
let formData = new FormData();
formData.append('key1', 'value1');
formData.append('key2', 'value2');

// Список пар ключ/значение
for(let [name, value] of formData) {
  alert(`${name} = ${value}`); // key1=value1, потом key2=value2
}
```

Технически форма может иметь много полей с одним и тем же именем name, поэтому несколько вызовов append добавят несколько полей с одинаковыми именами.
Ещё существует метод set, его синтаксис такой же, как у append. Разница в том, что .set удаляет все уже имеющиеся поля с именем name и только затем добавляет новое. То есть этот метод гарантирует, что будет существовать только одно поле с именем name, в остальном он аналогичен .append:

- formData.set(name, value),
- formData.set(name, blob, fileName).

Отправка формы с файлом

Объекты FormData всегда отсылаются с заголовком Content-Type: multipart/form-data, этот способ кодировки позволяет отсылать файлы. Таким образом, поля <input type="file"> тоже отправляются, как это и происходит в случае обычной формы.

```
<form id="formElem">
  <input type="text" name="firstName" value="John">
  Картинка: <input type="file" name="picture" accept="image/*">
  <input type="submit">
</form>

<script>
  formElem.onsubmit = async (e) => {
    e.preventDefault();

    let response = await fetch('/article/formdata/post/user-avatar', {
      method: 'POST',
      body: new FormData(formElem)
    });

    let result = await response.json();

    alert(result.message);
  };
</script>
```

Отправка формы с Blob-данными

Но на практике бывает удобнее отправлять изображение не отдельно, а в составе формы, добавив дополнительные поля для имени и другие метаданные.

Кроме того, серверы часто настроены на приём именно форм, а не просто бинарных данных.

```
<body style="margin:0">
  <canvas id="canvasElem" width="100" height="80" style="border:1px solid"></canvas>

  <input type="button" value="Отправить" onclick="submit()">

  <script>
    canvasElem.onmousemove = function(e) {
      let ctx = canvasElem.getContext('2d');
      ctx.lineTo(e.clientX, e.clientY);
      ctx.stroke();
    };

    async function submit() {
      let imageBlob = await new Promise(resolve => canvasElem.toBlob(resolve, 'image/png'));

      let formData = new FormData();
      formData.append("firstName", "John");
      formData.append("image", imageBlob, "image.png");

      let response = await fetch('/article/formdata/post/image-form', {
        method: 'POST',
        body: formData
      });
      let result = await response.json();
      alert(result.message);
    }

  </script>
</body>
```

Fetch: ход загрузки

Метод fetch позволяет отслеживать процесс получения данных.

Заметим, на данный момент в fetch нет способа отслеживать процесс отправки. Для этого используйте XMLHttpRequest, позже мы его рассмотрим.

Чтобы отслеживать ход загрузки данных с сервера, можно использовать свойство response.body. Это ReadableStream («поток для чтения») – особый объект, который предоставляет тело ответа по частям, по мере поступления. Потоки для чтения описаны в спецификации Streams API.

В отличие от response.text(), response.json() и других методов, response.body даёт полный контроль над процессом чтения, и мы можем подсчитать, сколько данных получено на каждый момент.

Результат вызова await reader.read() – это объект с двумя свойствами:

- done – true, когда чтение закончено, иначе false.
- value – типизированный массив данных ответа Uint8Array.

```
// вместо response.json() и других методов
const reader = response.body.getReader();

// бесконечный цикл, пока идёт загрузка
while(true) {
  // done становится true в последнем фрагменте
  // value - Uint8Array из байтов каждого фрагмента
  const {done, value} = await reader.read();

  if (done) {
    break;
  }

  console.log(`Получено ${value.length} байт`)
}
```

Fetch: прерывание запроса

Как мы знаем, метод fetch возвращает
понятия «отмены» промиса. Как же пре

Для таких целей существует специальн
AbortController, который можно использ
других асинхронных задач.

```
// прервать через 1 секунду
let controller = new AbortController();
setTimeout(() => controller.abort(), 1000);

try {
  let response = await fetch('/article/fetch-abort/demo/hang', {
    signal: controller.signal
  });
} catch(err) {
  if (err.name == 'AbortError') { // обработать ошибку от вызова abort()
    alert("Прервано!");
  } else {
    throw err;
  }
}
```

Fetch: запросы на другие сайты (CORS)

CORS (Cross-Origin Resource Sharing) — это механизм безопасности, который позволяет ограничить, какие веб-ресурсы могут обращаться к ресурсам на другом домене. Он предназначен для защиты от атак, связанных с выполнением нежелательных запросов на других доменах с помощью JavaScript в браузере.

Веб-браузеры обычно ограничивают запросы между разными доменами из соображений безопасности. Например, скрипт, загруженный с одного домена, не может просто так отправить запрос на другой домен и получить ответ. Это называется политикой одного источника (Same-Origin Policy).

XMLHttpRequest (до fetch)

XMLHttpRequest — это встроенный объект в JavaScript, который позволяет веб-странице взаимодействовать с сервером, не перезагружая страницу. С его помощью можно выполнять асинхронные запросы к серверу, получать данные и обновлять страницу динамически.

Основные возможности XMLHttpRequest:

- Асинхронные запросы: Вы можете отправлять запросы к серверу и получать данные без перезагрузки страницы, что позволяет создавать более интерактивные и отзывчивые веб-приложения.
- Поддержка различных форматов данных: Вы можете отправлять и получать данные в различных форматах, таких как текст, JSON, XML, и даже бинарные данные.
- Управление запросами: Вы можете настроить заголовки HTTP-запросов, обрабатывать ошибки, отслеживать состояние запроса и многое другое.

XMLHttpRequest => readyState

Состояния объекта XMLHttpRequest:

XMLHttpRequest имеет пять состояний, которые можно отслеживать с помощью свойства readyState:

- (UNSENT): Объект был создан. Метод open() еще не был вызван.
- (OPENED): Метод open() был вызван.
- (HEADERS_RECEIVED): Метод send() был вызван, и заголовки ответа и статус доступны.
- (LOADING): Загрузка данных; если данные поступают в пакетах, responseText содержит часть данных.
- (DONE): Операция завершена. Данные полностью получены.

Обработка ошибок и другие функции:

Установка заголовков запроса: Вы можете использовать метод setRequestHeader() для установки необходимых заголовков.

Обработка ошибок: Вы можете добавить обработчики для различных событий, таких как onerror, ontimeout, чтобы обрабатывать ошибки запроса.

```
xhr.onerror = function() {
    console.error("Request failed");
};

xhr.ontimeout = function() {
    console.error("Request timed out");
};

// Устанавливаем тайм-аут для запроса (например, 5000 мс)
xhr.timeout = 5000;
```

Объекты URL

В JavaScript объекты URL представляют собой средства для работы с URL-адресами. Они предоставляют удобный интерфейс для анализа, создания, манипулирования и проверки URL-адресов.

Создание объекта URL

Для создания объекта URL можно использовать конструктор URL(), который принимает два аргумента

```
// Пример с полным URL
const url = new URL('https://example.com:8000/path/name?query=string#hash');

// Пример с базовым URL
const relativeUrl = new URL('/path/name', 'https://example.com');
```

Свойства объекта URL

Объект URL имеет различные свойства, которые представляют части URL:

- href: Полный URL-адрес.
- protocol: Протокол (например, https:).
- hostname: Имя хоста (например, example.com).
- port: Порт (например, 8000).
- pathname: Путь (например, /path/name).
- search: Строка запроса (например, ?query=string).
- hash: Хеш-часть (например, #hash).
- host: Хост и порт вместе (например, example.com:8000).
- origin: Источник (например, https://example.com:8000).

Манипуляция URL

```
const url = new URL('https://example.com/path/name');

url.pathname = '/new/path';
url.search = '?new=query';
url.hash = '#newHash';

console.log(url.href); // https://example.com/new/path?new=query#newHash
```

Работа с параметрами строки запроса

```
const url = new URL('https://example.com/path?name=value');

// Получение значения параметра
console.log(url.searchParams.get('name')); // value

// Добавление нового параметра
url.searchParams.append('newParam', 'newValue');
console.log(url.href); // https://example.com/path?name=value&newParam=newValue

// Удаление параметра
url.searchParams.delete('name');
console.log(url.href); // https://example.com/path?newParam=newValue

// Установка нового значения параметра
url.searchParams.set('newParam', 'updatedValue');
console.log(url.href); // https://example.com/path?newParam=updatedValue
```

web-sockets

WebSocket — это протокол связи, предоставляющий полный дуплексный канал связи поверх одного TCP-соединения. Он предназначен для реализации веб-приложений, которые требуют постоянного обновления данных от сервера без необходимости многократного открытия и закрытия соединений (как это происходит при использовании обычных HTTP-запросов).

Основные особенности WebSocket:

- Полный дуплекс: Клиент и сервер могут отправлять данные одновременно.
- Меньшие накладные расходы: После установления соединения WebSocket заголовки HTTP используются только один раз, что уменьшает накладные расходы по сравнению с обычными HTTP-запросами.
- Поддержка в реальном времени: Подходит для приложений, требующих мгновенного обмена данными (например, чаты, уведомления, игры).

web sockets server

```
const WebSocket = require("ws");

const server = new WebSocket.Server({ port: 8080 });

let clients = [];

server.on("connection", (socket) => {
  console.log("Client connected");
  clients.push(socket);

  // Отправка приветственного сообщения клиенту в формате JSON
  socket.send(JSON.stringify({ message: "Welcome to the WebSocket server!" }));

  // Обработка полученных сообщений от клиента
  socket.on("message", (message) => {
    console.log("Received:", message);
    try {
      const data = JSON.parse(message);
      console.log("Parsed message:", data);

      // Рассылаем сообщение всем подключенным клиентам
      clients.forEach((client) => {
        if (client !== socket && client.readyState === WebSocket.OPEN) {
          client.send(JSON.stringify({ message: data.message }));
        }
      });
    } catch (error) {
      console.error("Error parsing JSON:", error);
    }
  });

  // Обработка закрытия соединения
  socket.on("close", () => {
    console.log("Client disconnected");
    clients = clients.filter((client) => client !== socket);
  });
});

console.log(`WebSocket server is running ${server.path}`);
```

```
const WebSocket = require("ws");

const server = new WebSocket.Server({ port: 8080 });

let clients = [];

server.on("connection", (socket) => {
  console.log("Client connected");
  clients.push(socket);

  // Отправка приветственного сообщения клиенту в формате JSON
  socket.send(JSON.stringify({ message: "Welcome to the WebSocket server!" }));

  // Обработка полученных сообщений от клиента
  socket.on("message", (message) => {
    console.log("Received:", message);
    try {
      const data = JSON.parse(message);
      console.log("Parsed message:", data);

      // Рассылаем сообщение всем подключенным клиентам
      clients.forEach((client) => {
        if (client !== socket && client.readyState === WebSocket.OPEN) {
          client.send(JSON.stringify({ message: data.message }));
        }
      });
    } catch (error) {
      console.error("Error parsing JSON:", error);
    }
  });

  // Обработка закрытия соединения
  socket.on("close", () => {
    console.log("Client disconnected");
    clients = clients.filter((client) => client !== socket);
  });
});

console.log("WebSocket server is running on ws://localhost:8080");
```

Основные методы HTTP-запросов:

Сетевые запросы являются основой взаимодействия между клиентом и сервером в интернете. Вот основные методы HTTP-запросов, которые широко используются в веб-разработке:

- GET
- POST
- PUT
- DELETE
- HEAD
- OPTION
- PATCH

GET

Назначение: Запрос данных с сервера.

Характеристики:

- Идэмпотентный: Множественные одинаковые запросы не изменяют состояние сервера.
- Безопасный: Предназначен только для получения данных, не изменяя их.

Примеры использования: Получение веб-страниц, изображений, данных через API.

POST

Назначение: Отправка данных на сервер для создания или обновления ресурса.

Характеристики:

- Не идэмпотентный: Повторные запросы могут приводить к разным результатам (например, к дублированию записей).
- Используется для создания новых ресурсов.

Примеры использования: Отправка форм данных, создание новых записей в базе данных.

PUT

Назначение: Обновление ресурса на сервере или создание ресурса, если он еще не существует.

Характеристики:

- Идэмпотентный: Множественные одинаковые запросы приводят к одному и тому же результату.
- Используется для полного обновления ресурса.

Примеры использования: Обновление информации о пользователе, загрузка файлов на сервер.

DELETE

Назначение: Удаление ресурса с сервера.

Характеристики:

- Идэмпотентный: Повторные запросы приводят к одному и тому же результату (удаление ресурса).

Примеры использования: Удаление записей из базы данных, удаление файлов с сервера.

HEAD

Назначение: Получение заголовков ответа без тела.

Характеристики:

- Идэмпотентный.
- Безопасный.

Примеры использования: Проверка доступности ресурса, получение метаданных.

OPTIONS

Назначение: Получение информации о методах, поддерживаемых сервером для конкретного ресурса.

Характеристики:

- Идэмпотентный.
- Безопасный.

Примеры использования: Определение возможностей сервера для определенного URL.

PATCH

Назначение: Частичное обновление ресурса на сервере.

Характеристики:

- Не идэмпотентный (может быть идэмпотентным в зависимости от реализации).

Примеры использования: Обновление отдельных полей записи в базе данных.

Shadow-DOM

Shadow DOM — это технология, которая позволяет создавать и инкапсулировать отдельные компоненты на веб-странице. Она помогает изолировать стили и разметку отдельных компонентов от остального содержимого страницы. Это делает компоненты более надежными и повторно используемыми.

Основные концепции Shadow DOM:

- Shadow Tree: Это отдельное дерево DOM, которое находится внутри элемента. Оно отличается от основного DOM дерева и изолировано от него.
- Shadow Host: Элемент, внутри которого создается Shadow DOM. Например, <my-element> может быть хостом для Shadow DOM.
- Shadow Root: Корневой элемент Shadow DOM. Это начало Shadow Tree.
- Shadow Boundary: Граница, которая разделяет основной DOM и Shadow DOM.

Преимущества Shadow DOM:

- Инкапсуляция стилей и разметки: Стили и разметка внутри Shadow DOM не влияют на основной DOM и наоборот.
- Повторное использование компонентов: Можно создать компоненты, которые можно использовать на разных страницах без изменений в их внутренней структуре и стилях.
- Улучшение читаемости и поддержки кода: Легче управлять небольшими, изолированными частями кода, чем большим монолитом.

Пример

- Создание кастомного элемента: Мы создаем новый класс MyElement, который наследуется от HTMLElement.
- Создание shadow root: В конструкторе элемента вызывается метод attachShadow({ mode: 'open' }), который создает и прикрепляет shadow root к элементу. Параметр mode: 'open' указывает, что мы можем получить доступ к shadow DOM через элемент, например, element.shadowRoot.
- Добавление стилей и разметки: Внутри shadow root создаются стили и элементы. Эти стили и элементы не влияют на внешний DOM.
- Регистрация элемента: Используя customElements.define, мы регистрируем наш кастомный элемент, чтобы его можно было использовать в HTML как <my-element>.

```
<body>
  <!-- Shadow host -->
  <my-element></my-element>

  <script>
    // Определение класса для кастомного элемента
    class MyElement extends HTMLElement {
      constructor() {
        super();

        // Создание shadow root
        const shadow = this.attachShadow({ mode: 'open' });

        // Создание элементов внутри shadow DOM
        const wrapper = document.createElement('div');
        wrapper.setAttribute('class', 'wrapper');

        const style = document.createElement('style');
        style.textContent = `
        .wrapper {
          background-color: lightgrey;
          padding: 10px;
          border-radius: 5px;
        }
      `;

        // Добавление элементов в shadow root
        shadow.appendChild(style);
        shadow.appendChild(wrapper);
        wrapper.textContent = 'Hello from Shadow DOM!';
      }
    }

    // Регистрация кастомного элемента
    customElements.define('my-element', MyElement);
  </script>
</body>
```

Shadow-dom Итого

Важно знать:

- Shadow DOM может быть открыт (mode: 'open') или закрыт (mode: 'closed'). В закрытом режиме доступ к shadow root из JavaScript невозможен.
- Инкапсуляция стилей работает благодаря специфичности CSS внутри shadow DOM. Внешние стили не применяются к элементам внутри shadow DOM.

Заключение

Shadow DOM — мощный инструмент для создания изолированных и повторно используемых компонентов. Понимание этой технологии поможет вам создавать более чистый, организованный и поддерживаемый код.

Ресурсы

Сетевые запросы - ТЫК

Web компоненты - ТЫК

Пример web-socket-ов - ТЫК

Shadow-dom пример - ТЫК