Сетевые средства, синхронное и асинхронное взаимодействие с сервером

Кулаков Кирилл Александрович

Организация взаимодействия клиента с сервером

- Синхронные запросы
  - Блокирование работы клиента до получения ответа
  - Передача больших объемов данных
  - Перезагрузка страниц
- Асинхронные запросы
  - Неблокирующие запросы
  - Передача малых объемов данных
  - Обработка данных в рамках страницы на клиенте

Ajax

- Одиночный запрос — одиночный ответ
- $.ajax(url, {settings})
  - accepts: Accept заголовка запроса
  - async: асинхронность
  - contentType: тип передаваемого содержимого
  - data: Данные, которые будут отправлены на сервер
  - error: callback обработки ошибки
  - method: HTTP метод
  - success: callback успешного запроса
  - ….

XmlHttpRequest

- XMLHttpRequest – это встроенный в браузер объект, который даёт возможность делать HTTP-запросы к серверу без перезагрузки страницы.
- let xhr = new XMLHttpRequest(); — создание

  xhr.open(method, URL, [async, user, password]) — конфигурация
- xhr.send([body]) — отправка
  - (on)load: получен какой-либо ответ, включая ответы с HTTP-ошибкой
  - (on)error: запрос не может быть выполнен
  - (on)progress: прогресс загрузки ответа

XmlHttpRequest

```
xhr.onload = function() {
  alert(`Загружено: ${xhr.status} ${xhr.response}`);
};

xhr.onerror = function() { // происходит, только когда запрос совсем не получилось выполнить
  alert(`Ошибка соединения`);
};

xhr.onprogress = function(event) { // запускается периодически
  // event.loaded - количество загруженных байт
  // event.lengthComputable = равно true, если сервер присылает заголовок Content-Length
  // event.total - количество байт всего (только если lengthComputable равно true)
  alert(`Загружено ${event.loaded} из ${event.total}`);
};
```

XmlHttpRequest: свойства

- xhr.responseType: указать ожидаемый тип ответа
  - "" (по умолчанию) – строка,
  - "text" – строка,
  - "arraybuffer" – ArrayBuffer (для бинарных данных, смотрите в ArrayBuffer, бинарные массивы),
  - "blob" – Blob (для бинарных данных, смотрите в Blob),
  - "document" – XML-документ (может использовать XPath и другие XML-методы),
  - "json" – JSON (парсится автоматически).
- xhr.response: получить ответ

XmlHttpRequest: свойства

- xhr.readyState: текущее состояние (событие (on)readystatechange)
  - UNSENT = 0; // исходное состояние
  - OPENED = 1; // вызван метод open
  - HEADERS_RECEIVED = 2; // получены заголовки ответа
  - LOADING = 3; // ответ в процессе передачи (данные частично получены)
  - DONE = 4; // запрос завершён
- xhr.abort(): завершить запрос
- setRequestHeader(name, value): заголовки

XmlHttpRequest: свойства

- getResponseHeader(name): Возвращает значение заголовка ответа name
- getAllResponseHeaders(): Возвращает все заголовки ответа
- upload: объект прогресса отправки запроса
  - loadstart – начало загрузки данных.
  - progress – генерируется периодически во время отправки на сервер.
  - abort – загрузка прервана.
  - error – ошибка, не связанная с HTTP.
  - load – загрузка успешно завершена.
  - timeout – вышло время, отведённое на загрузку (при установленном свойстве timeout).
  - loadend – загрузка завершена, вне зависимости от того, как – успешно или нет.
- xhr.withCredentials: отправка запроса на сторонний источник

Fetch

- Современный способ реализации AJAX запроса

  let promise = fetch(url, [options])

- Результат: объект Response
  - .status – код статуса HTTP-запроса, например 200.
  - .ok – логическое значение: будет true, если код HTTP-статуса в диапазоне 200-299.
  - .text() – читает ответ и возвращает как обычный текст
  - .json() – декодирует ответ в формате JSON
  - .formData() – возвращает ответ как объект FormData
  - .blob() – возвращает объект как Blob
  - .arrayBuffer() – возвращает ответ как ArrayBuffer
  - .body – объект ReadableStream

Fetch: примеры

- Через await

```
let url = 'http://localhost/data';
let response = await fetch(url);
let commits = await response.json(); // читаем ответ в формате JSON
alert(commits[0].author.login);
```

- Через промисы

```
fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits')
  .then(response => response.json())
  .then(commits => alert(commits[0].author.login));
```

Fetch: настройки

- response.headers: заголовки ответа
- headers: заголовки запроса

```
let response = fetch(protectedUrl, {headers: {Authentication: 'secret'}});
```

- method: HTTP метод
- body: тело запроса

```
let response = await fetch('/user', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json;charset=utf-8'
  },
  body: JSON.stringify(user)
});
```

WebSocket

- Спецификация RFC 6455
- Обмен данными между браузером и сервером через постоянное соединение
- Данные передаются по нему в обоих направлениях в виде «пакетов», без разрыва соединения и дополнительных HTTP-запросов.
- Префикс протокола ws:// или wss://

WebSocket

- API:
  - Событие (on)open – соединение установлено
  - Событие (on)message – получены данные
  - Событие (on)error – ошибка
  - Событие (on)close – соединение закрыто
  - Метод send — отправка данных

WebSocket: пример

```
let socket = new WebSocket("ws://servername");
socket.onopen = function(e) {
  alert("[open] Соединение установлено");
  alert("Отправляем данные на сервер");
  socket.send("Меня зовут Джон");
};
socket.onmessage = function(event) {
  alert(`[message] Данные получены с сервера: ${event.data}`);
};
socket.onclose = function(event) {
  if (event.wasClean) {
    alert(`[close] Соединение закрыто чисто, код=${event.code} причина=${event.reason}`);
  } else {
    // например, сервер убил процесс или сеть недоступна
    // обычно в этом случае event.code 1006
    alert('[close] Соединение прервано');
  }
};
socket.onerror = function(error) {
  alert(`[error] ${error.message}`);
};
```

WebSocket: пример сервера

```
const http = require('http');
const ws = require('ws');
const wss = new ws.Server({noServer: true});
function accept(req, res) {
  // все входящие запросы должны использовать websockets
  if (!req.headers.upgrade || req.headers.upgrade.toLowerCase() != 'websocket') {
    res.end();
    return;
  }
  // может быть заголовок Connection: keep-alive, Upgrade
  if (!req.headers.connection.match(/\bupgrade\b/i)) {
    res.end();
    return;
  }
  wss.handleUpgrade(req, req.socket, Buffer.alloc(0), onConnect);
}
function onConnect(ws) {
  ws.on('message', function (message) {
    let name = message.match(/([\p{Alpha}\p{M}\p{Nd}\p{Pc}\p{Join_C}]+)$/gu) || "Гость";
    ws.send(`Привет с сервера, ${name}!`);

    setTimeout(() => ws.close(1000, "Пока!"), 5000);
  });
}
if (!module.parent) {
  http.createServer(accept).listen(8080);
} else {
  exports.accept = accept;
}
```

WebSocket: открытие

- Браузер проверяет поддержку вебсокета при открытии соединения
- **Origin** – источник текущей страницы (например https://javascript.info). Объект WebSocket по своей природе не завязан на текущий источник. Нет никаких специальных заголовков или других ограничений. Старые сервера все равно не могут работать с WebSocket, поэтому проблем с совместимостью нет. Но заголовок Origin важен, так как он позволяет серверу решать, использовать ли WebSocket с этим сайтом.
- **Connection**: Upgrade – сигнализирует, что клиент хотел бы изменить протокол.
- **Upgrade**: websocket – запрошен протокол «websocket».
- **Sec-WebSocket-Key** – случайный ключ, созданный браузером для обеспечения безопасности.
- **Sec-WebSocket-Version** – версия протокола WebSocket, текущая версия 13.

- **Запрос**:

GET /chat

Host: javascript.info

Origin: https://javascript.info

Connection: Upgrade

Upgrade: websocket

Sec-WebSocket-Key: Iv8io/9s+lYFgZWcXczP8Q==

Sec-WebSocket-Version: 13

- **Ответ**:

101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: hsBlbuDTkk24srzEOTBUlZAlC2g=

WebSocket: дополнения

- **Sec-WebSocket-Extensions**: deflate-frame означает, что браузер поддерживает сжатие данных.
  - Расширение – это что-то, связанное с передачей данных, расширяющее сам протокол WebSocket.
  - Заголовок Sec-WebSocket-Extensions отправляется браузером автоматически со списком всевозможных расширений, которые он поддерживает.
- **Sec-WebSocket-Protocol**: soap, wamp означает, что мы будем передавать не только произвольные данные, но и данные в протоколах SOAP или WAMP (The WebSocket Application Messaging Protocol" – «протокол обмена сообщениями WebSocket приложений»).
  - заголовок описывает не передачу, а формат данных, который мы собираемся использовать.
  - Официальные подпротоколы WebSocket регистрируются в каталоге IANA. http://www.iana.org/assignments/websocket/websocket.xml

WebSocket: дополнения

- let socket = new WebSocket("wss://javascript.info/chat", ["soap", "wamp"]);
- Запрос:

  GET /chat

  ...

  Sec-WebSocket-Version: 13

  Sec-WebSocket-Extensions: deflate-frame

  Sec-WebSocket-Protocol: soap, wamp
- Ответ:

  101 Switching Protocols

  Upgrade: websocket

  Connection: Upgrade

  Sec-WebSocket-Accept: hsBlbuDTkk24srzEOTBUlZAlC2g=

  Sec-WebSocket-Extensions: deflate-frame

  Sec-WebSocket-Protocol: soap

WebSocket: передача данных

- Поток данных в WebSocket состоит из «фреймов», фрагментов данных, которые могут быть отправлены любой стороной:
  - «текстовые фреймы» – содержат текстовые данные, которые стороны отправляют друг другу.
  - «бинарные фреймы» – содержат бинарные данные, которые стороны отправляют друг другу.
  - «пинг-понг фреймы» используется для проверки соединения; отправляется с сервера, браузер реагирует на них автоматически.
  - также есть «фрейм закрытия соединения» и некоторые другие служебные фреймы.
- В браузере напрямую работаем только с текстовыми и бинарными фреймами.

WebSocket: передача данных

- Метод WebSocket .send() может отправлять и текстовые и бинарные данные
- При получении данных, текст всегда поступает в виде строки, для бинарных данных - один из двух форматов: Blob или ArrayBuffer.
  - задаётся свойством socket.bufferType, по умолчанию оно равно "blob", так что бинарные данные поступают в виде Blob-объектов.

```
socket.bufferType = "arraybuffer";
socket.onmessage = (event) => {
  // event.data является строкой (если текст) или arraybuffer (если двоичные данные)
}
```

WebSocket: передача данных

Буферизация передачи данных для медленных соединений

Свойство socket.bufferedAmount хранит количество байт буферизованных данных

```
setInterval(() => {
  if (socket.bufferedAmount == 0) {
    socket.send(moreData());
  }
}, 100);
```

WebSocket: состояние

- существует дополнительное свойство socket.readyState со значениями:
  - 0 – «CONNECTING»: соединение ещё не установлено,
  - 1 – «OPEN»: обмен данными,
  - 2 – «CLOSING»: соединение закрывается,
  - 3 – «CLOSED»: соединение закрыто.

WebSocket: завершение

- socket.close([code], [reason]);
  - code – специальный WebSocket-код закрытия (не обязателен). https://tools.ietf.org/html/rfc6455#section-7.4.1
  - reason – строка с описанием причины закрытия (не обязательна).

```
// закрывающая сторона:
socket.close(1000, "работа закончена");

// другая сторона:
socket.onclose = event => {
  // event.code === 1000
  // event.reason === "работа закончена"
  // event.wasClean === true (закрыто чисто)
};
```

Длинные опросы

- Длинные опросы – это самый простой способ поддерживать постоянное соединение с сервером, не используя при этом никаких специфических протоколов
  - Запрос отправляется на сервер.
  - Сервер не закрывает соединение, пока у него не возникнет сообщение для отсылки.
  - Когда появляется сообщение – сервер отвечает на запрос, посылая его.
  - Браузер немедленно делает новый запрос.

Длинные опросы

```
async function subscribe() {
  let response = await fetch("/subscribe");

  if (response.status == 502) {
    // Статус 502 - это таймаут соединения;
    // возможен, когда соединение ожидало слишком долго
    // и сервер (или промежуточный прокси) закрыл его
    // давайте восстановим связь
    await subscribe();
  } else if (response.status != 200) {
    // Какая-то ошибка, покажем её
    showMessage(response.statusText);
    // Подключимся снова через секунду.
    await new Promise(resolve => setTimeout(resolve, 1000));
    await subscribe();
  } else {
    // Получим и покажем сообщение
    let message = await response.text();
    showMessage(message);
    // И снова вызовем subscribe() для получения следующего сообщения
    await subscribe();
  }
}

subscribe();
```

Server Sent Events

- Спецификация Server-Sent Events описывает встроенный класс EventSource, который позволяет поддерживать соединение с сервером и получать от него события.
- Как и в случае с WebSocket, соединение постоянно.
- Но есть несколько важных различий:
  - Однонаправленность: данные посылает только сервер
  - Только текст
  - Обычный протокол HTTP

Server Sent Events

- Чтобы начать получать данные, нам нужно просто создать new EventSource(url)
- Сервер должен ответить со статусом 200 и заголовком Content-Type: text/event-stream, затем он должен поддерживать соединение открытым и отправлять сообщения в особом формате:
  - Текст сообщения указывается после data:, пробел после двоеточия необязателен.
  - Сообщения разделяются двойным переносом строки \n\n.
  - Чтобы разделить сообщение на несколько строк, мы можем отправить несколько data: подряд (третье сообщение).
- Пример:

```
data: Сообщение 1

data: Сообщение 2

data: Сообщение 3
data: в две строки
```

Server Sent Events

- Для каждого сообщения генерируется событие (on)message

```
let eventSource = new EventSource("/events/subscribe");


eventSource.onmessage = function(event) {
  console.log("Новое сообщение", event.data);
  // этот код выведет в консоль 3 сообщения, из потока данных выше
}
```

- Кроссдоменные запросы:

```
let source = new EventSource("https://another-site.com/events", {
  withCredentials: true
});
```

Server Sent Events

- Идентификатор сообщения
  - Поле id
- Пример:

  data: Сообщение 1

  id: 1

  data: Сообщение 2

  id: 2
- Результат: eventSource.lastEventId
- При переподключении: заголовок Last-Event-ID с последним полученным сообщением

Server Sent Events

- Типы событий:
  - Базовые:
    - message – получено сообщение, доступно как event.data.
    - open – соединение открыто.
    - error – не удалось установить соединение, например, сервер вернул статус 500.
  - Собственные:
    - eventSource.addEventListener('join', event => { alert(`${event.data} зашёл`); });

      event: join

      data: Боб

      data: Привет

socket.io

- Библиотека реализации сокетов с возможностью обратной поддержки
  - сервер сокета Web Node.js с унифицированным, абстрактным API
  - автоматический откат к сокетам Flash
  - автоматический откат к длинному опросу AJAX
  - автоматический откат к многокомпонентному потоковому AJAX
  - автоматический откат к потоковому <iframe>
  - автоматический откат к JSONP для более слабых клиентов