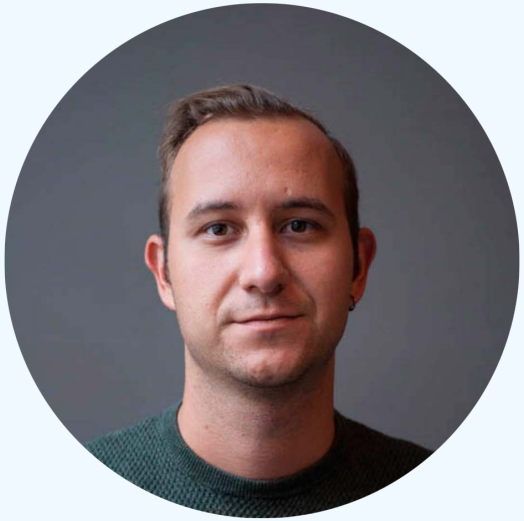
нетология
университет интернет-профессий

АСИНХРОННЫЕ HTTP-ЗАПРОСЫ

ИГОРЬ КУЗНЕЦОВ / SETKA



ИГОРЬ КУЗНЕЦОВ

CTO, Setka

@igkuz

fb.me/igkuznetsov

ПЛАН ЗАНЯТИЯ

1. XHR
2. Событие `readystatechange`
3. Событие `load`
4. Обработка ошибок
5. Событие `timeout`
6. Метод `abort()` и событие `abort`
7. События `loadstart` и `loadend`
8. Событие `progress`
9. Порядок вызова событий

XHR

«

*Объект `XMLHttpRequest` (или, как его кратко называют, **XHR**) дает возможность делать HTTP-запросы к серверу без перезагрузки страницы.*

Несмотря на слово XML в названии, `XMLHttpRequest` может работать с любыми данными, а не только с XML.

ДИНАМИЧЕСКАЯ ЗАГРУЗКА ДАННЫХ

Как правило, `XMLHttpRequest` используют для загрузки данных.

Например, мы хотим динамически загрузить курс валюты на сегодняшний день с определённого сервера, как это сделать с помощью `XMLHttpRequest`?

ПОДГРУЖАЕМ КУРС ВАЛЮТ

Подготовим простейшую страницу с пустым элементом `<span>`, внутрь которого мы хотим загрузить курс:

```
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <title>Курс валюты</title>
</head>
<body>
  <p>
    Курс USD/RUB на сегодня: <span id="result"></span> ₽.
  </p>
</body>
</html>
```

ПИШЕМ СКРИПТ ДЛЯ ПОЛУЧЕНИЯ КУРСА

Создаём экземпляр объекта `XMLHttpRequest`, сохраняем в переменную `xhr`:

```
var xhr = new XMLHttpRequest();
xhr.open(
 "GET",
 "https://netology-fbb-store-api.herokuapp.com/currency",
 false
);
xhr.send();
console.log(xhr.responseText);
```

МЕТОД `open` ДЛЯ НАСТРОЙКИ

Конфигурируем объект `xhr` с помощью метода `open`, который принимает три параметра:

1. `"GET"` – метод обращения к серверу по протоколу HTTP. "GET" переводится как «получить», т.е. этот метод предназначен для получения данных. Есть и другие методы, второй наиболее популярный метод – это **POST**. Упомяну также *PUT*, *PATCH*, *DELETE*, но на данном этапе не будем углубляться в их назначение и отличия друг от друга, они будут подробно рассмотрены в следующих лекциях;
2. `"https://netology-fbb-store-api.herokuapp.com/currency"` — адрес, на который мы хотим отправить запрос;
3. `false` – флаг, обозначающий тип нашего запроса. Значение `false` – синхронный, `true` – асинхронный.

ОТПРАВЛЯЕМ ЗАПРОС

При вызове метода `send` произойдёт реальное выполнение запроса к серверу. Обратите внимание, что в предыдущей строке мы определили, что хотим делать **синхронный** запрос (флагом `false`), а это значит, что после вызова `send` и до того момента, как от сервера будет получен ответ, главный поток будет «заморожен»: посетитель не сможет взаимодействовать со страницей – прокручивать, нажимать на кнопки и т.п.

После получения ответа выполнение продолжится со следующей строки.

ЧИТАЕМ ОТВЕТ

В последней строке мы читаем полученный от сервера ответ из свойства `xhr.responseText` и выводим в консоль.

В результате в консоли увидим такой длинный текст:

```
[{"ID":"R01010","NumCode":"036",
  "CharCode":"AUD","Nominal":1,
  "Name":"Австралийский доллар",
  "Value":42.9217}, ...]
```

ЧИТАЕМ JSON

Это данные в формате JSON. Раскодируем их, чтобы получить стандартный JavaScript-массив:

```
var data = JSON.parse(xhr.responseText);
console.log(data);
```

ОДНА ВАЛЮТА – ОДИН ОБЪЕКТ

Теперь в консоли будет выведена не просто строка, а структура данных JavaScript-массив из объектов, где каждый объект описывает одну валюту.

Завершим наш пример, отобразив на экран пользователя курс USD/RUB на сегодняшний день в заранее подготовленный `<span>`.

НАЙДЕМ НУЖНУЮ ВАЛЮТУ

Найдём в массиве `data` элемент, у которого `currency.CharCode === "USD"` – это будет объект с подробной информацией о USD:

```
var usd = data.find(function (currency) {
  return currency.CharCode === "USD";
});
```

ВЫВОДИМ КУРС НА СТРАНИЦУ

В подготовленный `<span>` элемент на странице выведем значение курса, которое хранится в поле `Value`:

```
document.getElementById("result").innerHTML = usd.Value;
```

НЕДОСТАТКИ СИНХРОННОГО ЗАПРОСА

Помните, что при синхронном запросе страница блокируется, посетитель не сможет нажимать ссылки, кнопки и т.п.

Это будет особенно раздражать при обращении к медленному серверу или при медленном соединении с сетью, у пользователя будет ощущение, что сайт завис.

ДЕМОНСТРАЦИЯ «ЗАМОРОЗКИ»

Проверим как это выглядит, обратившись к специально подготовленному медленному серверу (обратите внимание на изменившийся адрес в методе `open`). В самом начале добавим обработчик кликов по странице, чтобы в процессе отправки запроса мы проверили отзывчивость страницы.

```
document.body.addEventListener("click", function() {
  console.log("click on body");
});
var xhr = new XMLHttpRequest();
xhr.open("GET",
 "https://netology-fbb-store-api.herokuapp.com/currency-s
 false);
xhr.send();
console.log(`Синхронный запрос завершен,
  ответ от сервера: ${xhr.responseText}`);
```

РЕАКЦИЯ НА КЛИКИ ВО ВРЕМЯ ЗАПРОСА

Сервер ответит нам только через 25 секунд после `xhr.send()` – всё это время клики по телу страницы не приведут ни к каким действиям, в частности, наш обработчик не будет выводить в консоль сообщения «click on body». Пока браузер ждёт ответа от сервера, страница выглядит «замороженной».

На самом деле, все события `click` будут накоплены в очереди, и как только синхронный запрос завершится, мы увидим в консоли сначала сообщение «Синхронный запрос завершен, ответ от сервера: ...», а затем множество сообщений «click on body».

РЕДКОЕ ИСПОЛЬЗОВАНИЕ

Синхронные вызовы используются чрезвычайно редко, так как блокируют взаимодействие со страницей до окончания загрузки. Посетитель не может даже прокручивать её. Никакой JavaScript не может быть выполнен, пока синхронный вызов не завершён.

ДОБАВЛЯЕМ АСИНХРОННОСТЬ

Для того, чтобы запрос стал асинхронным, укажем третий параметр в методе `open` равным `true`.

```
1  document.body.addEventListener("click", function() {
2    console.log("click on body");
3  });
4  var xhr = new XMLHttpRequest();
5  xhr.open("GET",
6   "https://netology-fbb-store-api.herokuapp.com/currency-s
7   true);
8  xhr.send();
9  console.log(`Асинхронный запрос запущен,
10   ответ от сервера: ${xhr.responseText}`);
```

МЕДЛЕННЫЙ СЕРВЕР + АСИНХРОННЫЙ ЗАПРОС

В этом примере мы всё ещё обращаемся к специально подготовленному медленному серверу (который отвечает только через 25 секунд), но используем уже **асинхронный** запрос. Это значит, что после метода `send` код сразу продолжит выполняться и мы мгновенно увидим сообщение «Асинхронный запрос запущен, ответ от сервера: ». Далее, кликая по телу страницы, мы тут же будем получать сообщения в консоль «click on body». Ожидание ответа от сервера будет происходить в фоновом режиме, не блокируя пользовательский интерфейс.

НЕ ДОЖИДАЯСЬ ОТВЕТА

Но обратите внимание, что строка `console.log(`Асинхронный запрос запущен, ответ от сервера: ${xhr.responseText}`)` выводит сообщение «Асинхронный запрос запущен, ответ от сервера: », но переменная `xhr.responseText` пуста!

И действительно, сделав вывод в консоль сразу после асинхронного `xhr.send()`, браузер ещё не успел получить ответ от сервера. Приходит мысль, что перед чтением из `xhr.responseText` нужно подождать определённое время. Но сколько секунд ждать?

КАК УГАДАТЬ?

В данном примере сервер настроен так, что ответ приходит через 25 секунд. А что, если ответ придёт не через 25 секунд, а через 26? Или наоборот, сегодня сервер очень быстрый и отвечает через секунду – как угадать, сколько времени нужно «подождать» перед тем, как приступить к чтению из переменной `xhr.responseText`?

СОБЫТИЕ

`readystatechange`

ПОДПИСКА НА СОБЫТИЕ

Оказывается, объект `XMLHttpRequest` генерирует различные события, в частности, событие `readystatechange`, на которое можно подписаться с помощью уже известного вам метода `addEventListener`.

Дополним наш код следующими строками:

```
xhr.addEventListener(
  "readystatechange",
  onReadyStateChange
);
function onReadyStateChange() {
  if (xhr.readyState !== 4) return;
  console.log(xhr.responseText);
}
```

ГОТОВЫЙ ОТВЕТ

При помощи кода с предыдущего слайда мы выводим в консоль ответ от сервера из переменной `xhr.responseText`, как только этот ответ действительно получен, и нам не нужно угадывать, сколько времени «подождать».

ТЕКУЩЕЕ СОСТОЯНИЕ ЗАПРОСА

Событие `readystatechange` происходит несколько раз в процессе отсылки и получения ответа. При этом можно посмотреть «текущее состояние запроса» в свойстве `xhr.readyState`.

ВАРИАНТЫ СОСТОЯНИЙ

В примере выше нам интересно только состояние **4** – запрос завершён. Но есть и другие значения для `xhr.readyState`:

- **0** – начальное состояние;
- **1** – вызван `open`;
- **2** – получены заголовки ответа;
- **3** – загружается тело (получен очередной пакет данных);
- **4** – запрос завершён (именно это состояние нам интересно).

СОБЫТИЕ `load`

СОВРЕМЕННОЕ СВОЙСТВО `load`

В современной разработке по спецификации **XMLHttpRequest Level 2**, которая поддерживается всеми современными браузерами, вместо `readystatechange` мы можем использовать более удобное событие `load`.

ДОЖИДАЕМСЯ ОКОНЧАТЕЛЬНОГО ОТВЕТА

Событие `load` вызывается всего один раз, когда от сервера полностью получены все данные, таким образом, мы избавляем себя от необходимости делать проверку текущего состояния запроса:

```
xhr.addEventListener("load", onLoad);

function onLoad() {
    console.log(xhr.responseText);
}
```

ОБНОВИМ СКРИПТ

Весь код

```
var xhr = new XMLHttpRequest();
xhr.addEventListener("load", onLoad);
xhr.open("GET",
 "https://netology-fbb-store-api.herokuapp.com/currency",
 true);
xhr.send();
function onLoad() {
  var data = JSON.parse(xhr.responseText);
  var usd = data.find(function (currency) {
    return currency.CharCode === "USD";
  });
  document
    .getElementById("result").innerHTML = usd.Value;
}
```

ОБРАБОТКА ОШИБОК

ВИДЫ ОШИБОК

Иногда что-то может пойти не так и возникнет ошибка.

Ошибки можно условно разделить на две категории: **ошибки сети** и **ошибки протокола HTTP**.

Ошибки сети *— это когда у пользователя пропал интернет или запрашиваемый сервер недоступен (пропал интернет в дата-центре, например). Или не удаётся распознать имя домена (ошибка DNS).*

Ошибки сети можно отловить с помощью события `error`.

ЛОВИМ ОШИБКУ СЕТИ

Продемонстрирую сетевую ошибку на примере обращения к несуществующему домену:

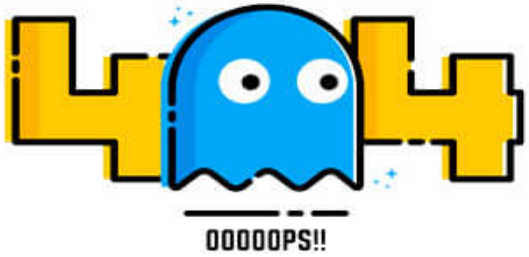
```
var xhr = new XMLHttpRequest();
xhr.addEventListener("load", onLoad);
xhr.addEventListener("error", onError);
xhr.open("GET", "http://not-exist-domain.com", true);
xhr.send();

function onLoad() {
  console.log("Сработало событие load");
}

function onError() {
  console.log("Сработало событие error");
}
```

«УЛОВ»

В этом примере мы увидим в консоли сообщение:

```
Сработало событие error
```

Обратите внимание, что события `load` в этом примере не произойдёт.



Ошибки протокола HTTP *— это такой тип ошибок, когда запрос успешно дошел до сервера, но сервер не смог обработать его.*

В этом случае сервер отвечает нам специальным HTTP-заголовком, содержащим код ошибки и краткое описание.

РАСШИФРУЕМ КОД ОТВЕТА

В стандарте HTTP определено множество кодов, описывающих различные типы HTTP-ответа, как успешных, так и нет. Код HTTP-ответа это трёхзначное число.

- **2xx** — обработка запроса завершилась успешно;
- **3xx** — в результате обработки запроса сервер перенаправляет нас по другому адресу;
- **4xx** — ошибка в самом запросе (например, указан неверный путь на сервере или нет доступа к запрашиваемому ресурсу);
- **5xx** — произошла ошибка на сервере в процессе обработки запроса (например, ошибка в приложении, обрабатывающем запрос, привела к его падению).

Подробнее о кодах HTTP-ответа

ОБРАБОТКА ОШИБОК HTTP

Ошибки протокола нужно проверять в обработчике события `load` с помощью свойства `xhr.status`.

«404 NOT FOUND»

Попробуем обратиться к несуществующему пути, чтобы получить от сервера ответ с ошибкой «404 Not Found»:

```
var xhr = new XMLHttpRequest();
xhr.addEventListener("load", onLoad);
xhr.addEventListener("error", onError);
xhr.open("GET", "http://netology.ru/404", true);
xhr.send();
function onLoad() {
 if (xhr.status !== 200) {
  console.log(`Ответ ${xhr.status}: ${xhr.statusText}`);
 } else {/* Успешный сценарий */}
}
function onError() {
 console.log("Сработало событие error");
}
```

ОТВЕТ СЕРВЕРА

В этом примере мы увидим в консоли сообщение `Ответ 404: Not Found`.

Обратите внимание, что событие `error` не произошло, так как на сетевом уровне мы успешно достучались до сервера и получили от него ответ.

СОБЫТИЕ `timeout`

МАКСИМАЛЬНОЕ ВРЕМЯ ОЖИДАНИЯ

Сервер может быть настолько медленным, что мы не хотим заставлять пользователя ждать бесконечно. Давайте определим максимальное время ожидания ответа от сервера. Если за указанное время ответа не будет, сообщим об этом пользователю.

Установим максимальное время с помощью свойства `xhr.timeout` в миллисекундах и обработчик события `timeout`.

Обратимся по специальному адресу на сервере, который отвечает только через 25 секунд, чтобы проверить, как сработает `timeout`, установленный в 10 секунд.

ПРОВЕРЯЕМ РАБОТУ ТАЙМЕРА

Весь код

```
xhr.timeout = 10000; // это 10 секунд в миллисекундах
xhr.addEventListener("load", onLoad);
xhr.addEventListener("timeout", onTimeout);
xhr.open("GET",
  "https://netology-fbb-store-api.herokuapp.com/currency-
    true);
xhr.send();
function onLoad() {
  console.log("Сработало событие load");
}
function onTimeout() {
  console.log("Сработало событие timeout");
}
```

ИТОГ ПРОВЕРКИ

Увидим в консоли сообщение `Сработало событие timeout`, но не увидим событий `load` и `error`.

МЕТОД `abort()` И СОБЫТИЕ `abort`

ВСЯ ВЛАСТЬ ПОЛЬЗОВАТЕЛЮ

Помимо жесткого ограничения по `timeout`, мы хотим дать пользователю возможность прервать запрос самостоятельно, если он устал ждать.

На этот случай у объекта `xhr` есть метод `abort()`, который позволяет прервать уже начатый запрос.

КНОПКА «ПРЕРВАТЬ ЗАПРОС»

Добавим на страницу простейшую кнопку `<button type="button" id="abort-btn">Прервать запрос</button>` и напишем обработчик клика, который будет вызывать метод `xhr.abort();`. Как только запрос прерван, мы получим событие `abort`, для которого также напишем обработчик.

ОБРАБОТЧИК ПРЕРЫВАНИЯ

```
var xhr = new XMLHttpRequest();
xhr.addEventListener("abort", onAbort);
xhr.open("GET",
 "https://netology-fbb-store-api.herokuapp.com/currency-s
  true);
xhr.send();
function onAbort() {
  console.log("Сработало событие abort");
}
var button = document.getElementById("abort-btn");
button.addEventListener("click", onButtonClick);
function onButtonClick() {
  xhr.abort();
}
```

ИТОГ РАБОТЫ

При открытии страницы у нас сразу посылается запрос к серверу. Это всё ещё тот самый медленный сервер, который ответит лишь через 25 секунд.

Если в течение этого времени нажать кнопку `«Прервать запрос»`, то запрос будет прерван и, благодаря нашему обработчику события `abort`, мы увидим в консоли сообщение `Сработало событие abort`.

Повторное нажатие на кнопку ни к чему не приведёт, так как запрос уже прерван. Если мы дождёмся ответа через 25 секунд или сработает `timeout` (если он задан), то кнопка `«Прервать запрос»` также потеряет актуальность.

Live Demo

События `loadstart` и `loadend`

РАСШИРЯЕМ ФУНКЦИОНАЛ

При отправке асинхронных запросов зачастую нам хотелось бы показать пользователю некий индикатор процесса, например, вращающийся спиннер или прогрессбар.

Этот индикатор нужно отобразить в начале запроса и скрыть по окончании. Кроме этого, мы могли бы отображать и скрывать кнопку `«Прервать запрос»` в нужный момент.

МЕХАНИКА РАБОТЫ

Событие `loadstart` срабатывает сразу после `xhr.send()` – это момент начала запроса.

Событие `loadend` срабатывает по окончании запроса, причём во всех возможных случаях: и при успехе, и при ошибке, и при `timeout` и при `abort`.

ПОКАЗЫВАЕМ ИНДИКАТОР ПРОЦЕССА

Весь код

```
var button = document.getElementById("abort-btn");
function onLoadStart() {
  console.log("Сработало событие loadstart");
  statusBlock.innerHTML = 'Сработало событие loadstart';
  spinner.innerHTML =
    '<img src="loader.gif"> Идёт загрузка...';
  button.style.display = "inline";
}
function onLoadEnd() {
  console.log("Сработало событие loadend");
  spinner.innerHTML = '';
  button.style.display = "none";
}
```

СОБЫТИЕ `progress`

«СКОЛЬКО ВЕШАТЬ В ГРАММАХ?»

Крутящийся индикатор это хорошо, но ещё лучше это конкретный индикатор в процентах, чтобы пользователь мог оценить время, оставшееся до окончания запроса.

Для этого можно подписать на событие `progress`, а внутри обработчика будет доступна информации о количестве уже скаченных байтов и общем объёме данных, планируемых к загрузке (если, конечно, сервер заранее сообщил нам общий размер ответа в заголовке `Content-Length`).

ТЕХНИЧЕСКИЙ МОМЕНТ

Обратите внимание, что в предыдущих примерах мы обращались к специально подготовленному «медленному» серверу, который отдавал данные через 25 секунд. Сами данные при этом достаточно малы (пара килобайт) и загружаются мгновенно. То есть, в примерах выше нам не удалось бы сделать индикатор процесса загрузки, так как он бы сначала 25 секунд показывал состояние 0%, а затем мгновенно переключался бы на 100% – в этом нет толка.

ГРУЗИМ БОЛЬШОЙ ОБЪЕМ

Индикатор процесса загрузки в процентах актуален при скачивании большого объёма данных. В следующем примере для полноты картины добавим кнопки `Загрузить` и `Отменить загрузку`:

```
<div id="result"></div>
<button id="start-btn">Загрузить</button>
<button id="abort-btn">Отменить загрузку</button>
```

ОБРАБАТЫВАЕМ СОБЫТИЕ `progress`

Вот как будет выглядеть JavaScript-код, обрабатывающий событие `progress` (обратите особое внимание на функцию `onProgress`):

```
xhr.addEventListener("progress", onProgress);
xhr.addEventListener("abort", onAbort);
xhr.open("GET",
 "https://netology-fbb-store-api.herokuapp.com/largedata"
  true);
```

ОТСЛЕЖИВАЕМ НАЖАТИЕ КНОПОК

```
var startButton = document.getElementById("start-btn");
startButton.addEventListener("click", onStartClick);
function onStartClick() {
  this.style.display = 'none';
  xhr.send();
}
var button = document.getElementById("abort-btn");
button.addEventListener("click", onButtonClick);
function onButtonClick() {
  xhr.abort();
}
function onAbort() {
  document.getElementById("result").innerHTML =
    "Сработало событие abort";
}
```

ОТСЛЕЖИВАЕМ ПРОГРЕСС

Весь код

```
function onProgress(event) {
  console.log("В очередной раз сработало событие
    progress");
  var percent = Math
    .round(event.loaded / event.total * 100);
  document.getElementById("result").innerHTML =
    `Загружено ${event.loaded} из
      ${event.total} байт: ${percent}%`;
}
```

ОСОБЕННОЕСТИ `onprogress`

Ещё особенности, которые необходимо учитывать при использовании `onprogress`:

- сервер может не сообщить браузеру общий объём данных заранее в заголовке `Content-Length` и тогда `event.total` будет равно `0`. Если мы хотим выводить значение в процентах, то нужно это учесть, чтобы избежать ошибки деления на 0;
- событие происходит при каждом полученном байте, но не чаще, чем раз в 50 мс.

ПОРЯДОК ВЫЗОВА СОБЫТИЙ

РАСПОЛОЖИМ ПО ПОРЯДКУ

Мы рассмотрели все события, которые генерируются объектом `XMLHttpRequest`.

Подведём итог, расположив все события в порядке их появления.

Из этой цепочки специально исключены все события `readystatechange`, так как при использовании современного стандарта **XMLHttpRequest Level 2** они просто не нужны, не удобны в использовании.

ПОРЯДОК ПОЯВЛЕНИЯ СОБЫТИЙ

– `loadstart` обозначает, что был вызван метод `xhr.send()` и началась передача данных;

– `progress` может происходить несколько раз при каждом полученном байте, но не чаще, чем раз в 50 мс;

– Далее происходит одно из четырёх возможных событий:

1. `load` при успешном получении данных;
2. `error` при сетевой ошибке;
3. `abort` при явной отмене соединения через `xhr.abort()`;
4. `timeout` при превышении времени ожидания, заданном в свойстве `xhr.timeout`;

– В самом конце, вне зависимости от исхода (успешного или нет), – `loadend`.

Объект `XMLHttpRequest` и его типы

- Объект `XMLHttpRequest` позволяет делать HTTP-запросы к серверу без перезагрузки страницы.
- В методе `open` передаются параметры запроса: метод обращения, адрес и его тип (синхронный или асинхронный).
- Синхронный тип запроса практически не используется. При его отправке главный поток будет «заморожен»: посетитель не сможет взаимодействовать со страницей до получения ответа.
- При асинхронном запросе ожидание ответа сервера будет происходить в фоновом режиме, не блокируя пользовательский интерфейс.

СОБЫТИЯ ОБЪЕКТА `XMLHttpRequest`

— Для того, чтобы своевременно узнать о выполнении асинхронного запроса, нам нужно подписаться на его событие `readystatechange`.

— Событие `readystatechange` отслеживает изменение состояния нашего запроса.

— Само состояние хранится в свойстве `xhr.readyState` и может принимать значение от 0 до 4, где 0 означает начальное состояние запроса, а 4 его завершение.

— Состояние `xhr.readyState` меняется несколько раз в процессе отсылки и, соответственно, столько же раз у нас сработает событие `readystatechange`.

— Вместо отслеживания состояний через `readystatechange` можно использовать более удобное событие `load`. Однако оно не поддерживается старыми браузерами.

ОШИБКИ ПРИ ОТПРАВКЕ ЗАПРОСА

- При отправке запроса может возникнуть ошибка.
- Ошибки можно условно разделить на две категории: **ошибки сети** и **ошибки протокола HTTP**.
- **Ошибки сети** можно отлавливать чере событие `error`.
- **Ошибки протокола HTTP** нужно проверять в обработчике события `load` с помощью свойства `xhr.status`.
- Данные типы ошибок являются взаимоисключающими: если случилась ошибка сети, значит ответа на наш запрос мы не получим и ошибки протокола не будет. И наоборот.

ПРЕРЫВАНИЕ ОТПРАВЛЕННОГО ЗАПРОСА

- При помощи свойства `xhr.timeout` мы можем указать максимальное время ожидания запроса (в миллисекундах).
- Обработчик события `timeout` позволит нам установить таймер, который будет срабатывать, когда указанное в свойстве `xhr.timeout` время ожидания истекло.
- Если на нашем запросе сработал таймер, то события `load` и `error` уже не сработают.
- Отправка запроса может быть отменена через вызов метода `xhr.abort()`.
- Навесив на элемент интерфейса вызов события `xhr.abort()`, мы можем предоставить пользователю возможность отменить отправку запроса.

ИНЫЕ СОБЫТИЯ

– Событие `loadstart` срабатывает сразу после xhr.send() – это момент начала запроса.

– Событие `loadend` причём во всех возможных случаях: и при успехе, и при ошибке, и при timeout и при abort.

– Подписка на событие `progress` позволяет получить информацию о количестве уже скаченных байтов и общем объёме данных, планируемых к загрузке.

– Событие `progress` срабатывает при каждом полученном байте, но не чаще, чем раз в 50 мс.

нетология
университет интернет-профессий

Задавайте вопросы и напишите отзыв о лекции!

ИГОРЬ КУЗНЕЦОВ

@igkuz

fb.me/igkuznetsov