**Лабораторная работа 2.**

**Использование технологии AJAX для динамической загрузки контента страницы.**

**Цель работы:** Знакомство с технологией динамической загрузки контента страницы AJAX. Применение AJAX для автоматизации процесса загрузки содержимого страницы при скроллинге.

**Теоретический материал**

Большинство современных сайтов работают с динамическим контентом, то есть с контентом, который загружается автоматически без перезагрузки страницы с помощью асинхронных (то есть “с задержкой”) запросов, выполняющихся при определенных условиях.

Например, на странице каталога, изначально отображено 20 карточек товаров. Пользователь пролистывает страницу вниз, подгружаются еще 20 новых элементов. При дальнейшей прокрутке подгрузятся еще +20 товаров и так далее. Если бы на странице каталога сразу отображались сотни карточек товаров, то страница грузилась бы значительно дольше, чем страница с двадцатью элементами.

С появлением технологии AJAX стало возможным асинхронно генерировать запросы к серверу, чтобы изменять старые данные на сервере или получать новые.

AJAX — это аббревиатура, которая означает AsynchronousJavascriptand XML,которая дословно переводится как «асинхронный JavaScript и XML».

Асинхронный – действие выполняется в фоне (не в основном потоке), другими словами, таким образом, что оно не мешает пользователю взаимодействовать со страницей;

JavaScript – язык, на котором всё это делается (т.е. создание и настройка запроса, отправка его на сервер, получение ответа и его разбор, обновление страницы);

XML – формат для хранения и передачи данных, в настоящее время вместо него чаще используется JSON, но кроме них можно использовать и другие форматы.

При использовании AJAX нет необходимости обновлять каждый раз всю страницу, так как обновляется только ее конкретная часть. Это намного удобнее, так как не приходится долго ждать, и экономичнее. Правда в этом случае, разработчику необходимо следить, чтобы пользователь был в курсе того, что происходит на странице. Это можно реализовать с использованием индикаторов загрузки, текстовых сообщений о том, что идёт обмен данными с сервером. Необходимо также понимать, что не все браузеры поддерживают AJAX (старые версии браузеров и текстовые браузеры). Плюс Javascript может быть отключен пользователем.

В классической модели веб-приложения:

* Пользователь заходит на веб-страницу и нажимает на какой-нибудь её элемент;
* Браузер формирует и отправляет запрос серверу;
* В ответ сервер генерирует совершенно новую веб-страницу и отправляет её браузеру и т. д., после чего браузер полностью перезагружает всю страницу.

При использовании AJAX:

* Пользователь заходит на веб-страницу и нажимает на какой-нибудь её элемент;
* Скрипт (на языке JavaScript) определяет, какая информация необходима для обновления страницы;
* Браузер отправляет соответствующий запрос на сервер;
* Сервер возвращает только ту часть документа, на которую пришёл запрос;
* Скрипт вносит изменения с учётом полученной информации (без полной перезагрузки страницы).

**Создание асинхронных запросов с помощью XHR**

XHR – это аббревиатура от объекта XMLHttpRequest, который как раз и позволяет взаимодействовать с сервером через AJAX.

Объект XmlHttpRequest представляет собой компактную объектную модель для отправки сценарием обращений HTTP в обход браузера. Клиентский код сценария не может влиять на процесс размещения запроса и результат отправки запроса. XmlHttpRequest позволяет сценарию отправлять HTTP запросы и обрабатывать полученные ответы.

Написание запроса можно разбить на следующие этапы.

*Этап 1. Создание экземпляра объекта XMLHttpRequest:*

constxhr = new XMLHttpRequest();

*Этап 2. Инициализация запроса с помощью метода open():*

xhr.open(method, url[, async[, user[, password]]]);

Где:

* *method* – метод отправки запроса на сервер (GET, POST, PUT, PATCH, DELETE);
* *url* – URL для отправки запроса;
* *async* – определяет, как следует выполнять запрос: асинхронно (true – по умолчанию) или нет (false);
* *user* и *password* – имя пользователя и пароль, использующиеся для аутентификации (по умолчанию имеют значение null).

*Этап 3. Назначение обработчика на событие readystatechange объекта xhr:*

xhr.onreadystatechange = function() {

// ...

}

Это событие происходит при изменении статуса запроса readyState.

*readyState* – это свойство, содержащее числовой код, по которому можно определить в какой стадии сейчас находится запрос.

Статусы кодов readyState:

* 0 – создан объект XMLHttpRequest, но метод open() ещё не вызывался;
* 1 – открыто новое соединение с помощью open() (этот этап также включает установку значений HTTP заголовкам с помощью setRequestHeader());
* 2 – отправлен (вызван send() и получены заголовки ответа);
* 3 – получена часть ответа;
* 4 – завершён.

*Более современный способ* – это вместо readystatechange слушать события load и error*:*

xhr.onload = () => {

…

}

*Этап 4. Отправка запроса. Выполняется это с помощью метода send().*

xhr.send();

Если запрос асинхронный, то выполнение send() не останавливает дальнейшее выполнение программы. В противном случае (если запрос синхронный), программа приостанавливается и возобновляет своё выполнение только после получения ответа от сервера.

В send() при необходимости можно передать аргумент (данные серверу в теле запроса). Если это не сделать, то по умолчанию будет использоваться значение null.

**Создание асинхронных запросов с помощью fetch()**

Метод fetch() — современный и очень мощный, поэтому начнём с него. Он не поддерживается старыми, но поддерживается всеми современными браузерами.

Базовый синтаксис:

let promise =fetch(url,[options])

*url* – URL для отправки запроса.

*options* – дополнительные параметры: метод, заголовки и так далее.

Без options это простой GET-запрос, скачивающий содержимое по адресу url.

Браузер сразу же начинает запрос и возвращает промис, который внешний код использует для получения результата.

Процесс получения ответа обычно происходит в два этапа.

*Во-первых*, promise выполняется с объектом встроенного класса Response в качестве результата, как только сервер пришлёт заголовки ответа.

На этом этапе мы можем проверить статус HTTP-запроса и определить, выполнился ли он успешно, а также посмотреть заголовки, но пока без тела ответа.

Промис завершается с ошибкой, если fetch не смог выполнить HTTP-запрос, например, при ошибке сети или если нет такого сайта. HTTP-статусы 404 и 500 не являются ошибкой.

*Во-вторых*, для получения тела ответа нам нужно использовать дополнительный вызов метода.

Response предоставляет несколько методов, основанных на промисах, для доступа к телу ответа в различных форматах:

* response.text() – читает ответ и возвращает как обычный текст,
* response.json() – декодирует ответ в формате JSON,
* response.formData() – возвращает ответ как объект FormData (разберём его в следующей главе),
* response.blob() – возвращает объект как Blob (бинарные данные с типом),
* response.arrayBuffer() – возвращает ответ как ArrayBuffer (низкоуровневое представление бинарных данных),
* помимо этого, response.body – это объект ReadableStream, с помощью которого можно считывать тело запроса по частям. Мы рассмотрим и такой пример несколько позже.

Пример получения JSON-объекта:

let response = await fetch(url);

let commits = await response.json();

То же самое без await, с использованием промисов:

fetch(url).then(response =>response.json()).then(commits => alert(commits));

**Заголовки ответа**

Заголовки ответа хранятся в объекте response.headers.

response.headers.get('Content-Type');

**Заголовки запроса**

Для установки заголовка запроса в *fetch* мы можем использовать опцию *headers*. Она содержит объект с исходящими заголовками, например:

letresponse = fetch(protectedUrl, {

headers: {

Authentication: 'secret'

}

});

Есть список [запрещённых HTTP-заголовков](https://fetch.spec.whatwg.org/#forbidden-header-name), которые нельзя установить.

**POST-запросы**

Для отправки POST-запроса или запроса с другим методом, нам необходимо использовать fetch параметры:

* method – HTTP метод, например POST,
* body – тело запроса, одно из списка:
* строка (например, в формате JSON),
* объект FormData для отправки данных как form/multipart,
* Blob/BufferSource для отправки бинарных данных,
* URLSearchParams для отправки данных в кодировке x-www-form-urlencoded, используется редко.

Чаще всего используется JSON.

Пример:

let response = await fetch('/article/fetch/post/user', {

method: 'POST',

headers: {

'Content-Type': 'application/json;charset=utf-8'

},

body: JSON.stringify(user)

});

let result = await response.json();

alert(result.message);

**Контрольноезадание**

Взяв за основу готовую ранее реализацию сайта с пагинацией страницы каталога, изменить загрузку элементов каталога.

Реализовать несколько видов пагинации:

1. С помощью Javascript

Из БД получить все записи (например с новостями). Далее разбиваем все записи на страницы, например по 3 записи на одной странице. Соответственно сформировать навигацию по этим страницам (Первая, Предыдущая, Следующая, Последняя)

2 С помощью технологии AJAX

Модифицировать пагинацию страницы с использованием технологии AJAX: Вместо загрузки определенного количества элементов каталога по нажатию на кнопку «следующая страница» или её аналога, загружать список элементов при скроллинге страницы.

Реализовать данную задачу с использованием объекта *XMLHttpRequest,* а также с помощью метода *fetch.*

Форму вывода результатов работы придумайте самостоятельно. Например, на разных страницах: на одной – с использованием Javascript, на второй – с использованием XMLHttpRequest, а на третьей – с помощью метода fetch.