hillel IT school

Lesson 25

План заняття

- Fetch;
- CORS;
- XMLHttpRequest;
- Web Socets

Запити між різними джерелами

Запити на інші сайти – надіслані до іншого домену (навіть субдомену), протоколу чи порту – потребують спеціальних заголовків від віддаленої сторони.

Ця політика називається “CORS”: Cross-Origin Resource Sharing (“спільне використання ресурсів між різними джерелами”). CORS існує для захисту інтернету від злих хакерів.

Протягом багатьох років скрипт з одного сайту не міг отримати доступ до вмісту іншого сайту.

Але з часом це змінилось

Безпечні запити

Запит є безпечним, якщо він задовольняє дві умови:

- Safe method: GET, POST або HEAD
- Safe headers – єдині дозволені спеціальні заголовки:

Accept,

Accept-Language,

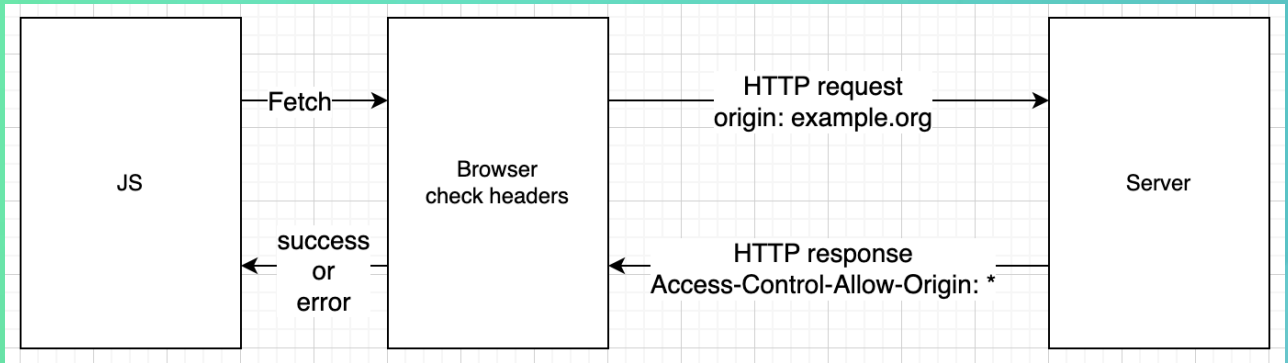
Content-Language,

Content-Type зі значенням application/x-www-form-urlencoded, multipart/form-data або text/plain.

Будь-який інший запит вважається “небезпечним”. Наприклад, запит із методом PUT або з HTTP-заголовком API-Key не відповідає обмеженням. Будь який сервервер, чи новий чи старий має змогу прийняти безпечний запит. Коли ми намагаємося зробити небезпечний запит, веб-переглядач надсилає спеціальний попередній запит “preflight”, який запитує сервер – чи погоджується він приймати такі запити між різними джерелами чи ні? І, якщо сервер явно не підтвердить це за допомогою заголовків, небезпечний запит не надсилається.

CORS для безпечних запитів

Якщо запит відбувається між різними джерелами, браузер завжди додає до нього заголовок Origin. Сервер може перевірити Origin і, якщо він погоджується прийняти такий запит, додати до відповіді спеціальний заголовок Access-Control-Allow-Origin. Цей заголовок має містити дозволене джерело або зірочку *. Тоді відповідь буде успішною, інакше – помилка.



Заголовки відповіді

Для запиту між різними джерелами JavaScript типово може мати доступ лише до так званих “безпечних” заголовків відповідей:

- Cache-Control
- Content-Language
- Content-Length
- Content-Type
- Expires
- Last-Modified
- …

Небезпечні запити

Запит перед друком використовує метод OPTIONS, без тіла запросу, але з трьома заголовками:

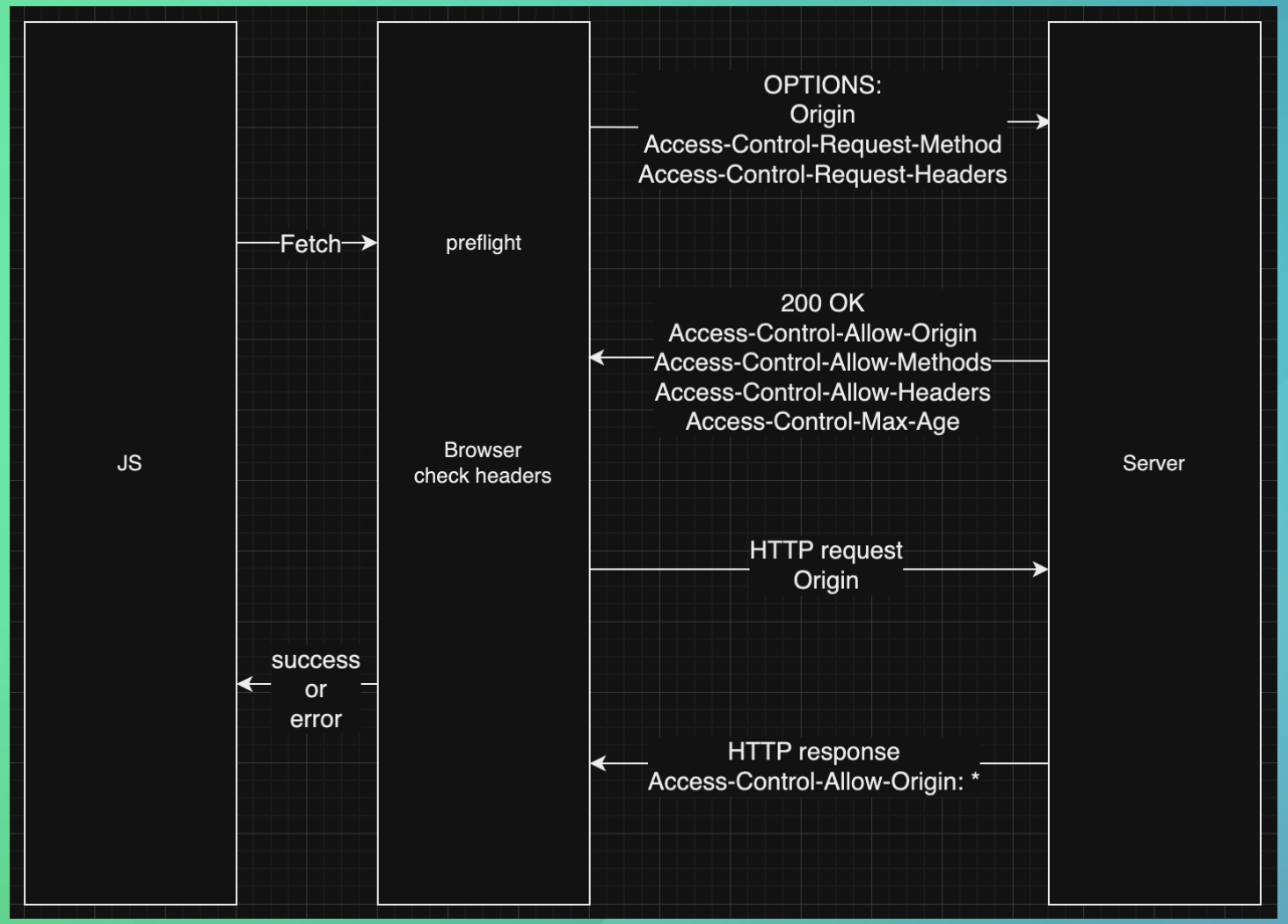
- Заголовок Access-Control-Request-Method містить метод небезпечного запиту.
- Заголовок Access-Control-Request-Headers надає список небезпечних HTTP-заголовків, що розділені комами.
- Заголовок Origin повідомляє, звідки надійшов запит (наприклад, https://example.org).

Якщо сервер погоджується обслуговувати запити, він повинен відповісти порожнім тілом, статусом 200 і заголовками:

- Access-Control-Allow-Origin має бути або *, або джерелом запиту, наприклад https://example.org, щоб дозволити це.
- Access-Control-Allow-Methods повинен мати дозволений метод.
- Access-Control-Allow-Headers повинен мати список дозволених заголовків.

Крім того, заголовок Access-Control-Max-Age може вказати кількість секунд для кешування дозволів. Тому веб-переглядачу не потрібно буде надсилати попередній запит для наступних подібних запитів в межах вже наданих дозволів.

Небезпечні запити



Fetch API

- referrer, referrerPolicy - встановлює HTTP-заголовок Referer. Зазвичай цей заголовок встановлюється автоматично і містить адресу сторінки, з якої був зроблений запит. Опція referrer дозволяє задати будь-який Referer (в межах поточного джерела) або видалити його. Опція **referrerPolicy** задає загальні правила для Referer. Referer policy doc
- mode - запобігає випадковим запитам між різними джерелами:

"cors" – запити між різними джерелами типово дозволені;

"same-origin" – запити між різними джерелами заборонені;

"no-cors" – дозволені лише безпечні запити між різними джерелами.

- credentials - визначає, чи має fetch надсилати файли cookie та заголовки HTTP-Authorization разом із запитом:

"same-origin" – типове значення, не надсилати запити між різними джерелами;

"include" – завжди надсилати, вимагає Access-Control-Allow-Credentials від сервера іншого джерела, щоб JavaScript міг отримати доступ до відповіді.

"omit" – ніколи не надсилати, навіть для запитів з однаковим джерелом.

Fetch API

- cache - типово запити fetch використовують стандартне HTTP-кешування. Опції cache дозволяють ігнорувати HTTP-кеш або тонко налаштувати його використання:

"default" – fetch використовує стандартні правила HTTP-кешу та заголовки,

"no-store" – повністю ігнорувати HTTP-кеш, цей режим стане типовим, якщо ми встановимо заголовок If-Modified-Since, If-None-Match, If-Unmodified -Since, If-Match або If-Range,

"reload" – не брати результат із HTTP-кешу (якщо є), а заповнити кеш відповіддю (якщо заголовки відповіді дозволяють цю дію),

"no-cache" – створити умовний запит, якщо є кешована відповідь, і звичайний запит в іншому випадку. Заповнити HTTP-кеш відповіддю,

"force-cache" – використовувати відповідь з HTTP-кешу, навіть якщо він застарілий. Якщо в HTTP-кеші немає відповіді, зробити звичайний HTTP-запит, поводитися як зазвичай,

"only-if-cached" – використовувати відповідь з HTTP-кешу, навіть якщо він застарілий. Якщо в HTTP-кеші немає відповіді, то видати помилку. Працює лише тоді, коли mode має значення "same-origin".

Fetch API

- redirect - дозволяє змінити типову поведінку:

"follow" – типове значення, слідувати HTTP-переадресаціям;

"error" – помилка при HTTP-переадресації;

"manual" – дозволяє обробляти HTTP-переадресації вручну. У разі переадресації ми отримаємо спеціальний об’єкт відповіді з response.type="opaqueredirect" та нульовим/порожнім статусом і більшістю інших властивостей.

- integrity - дозволяє перевірити, чи відповідає відповідь контрольній сумі відомій наперед.
- keepalive - вказує на те, що запит має “пережити” веб-сторінку, яка його ініціювала.

URL

Синтаксис для створення URL об’єктів:

const url = new URL(url, [base]);

- url – повний URL чи, якщо задано другий параметр, тільки шлях (дивись далі),
- base – необов’язковий параметр з “основою” відносно якої буде побудовано URL, якщо в першому параметрі передано тільки шлях.

URL

Url властивості на прикладі = https://example.com:8080/users/1?userName=example&country=Ukraine#hash

https – protocol

example.com - hostname

8080 – port

example.com:8080 – host

https://example.com:8080 – origin

/users/1 - pathname

?userName=example&country=Ukraine - search

#hash - hash

https://example.com:8080/users/1?userName=example&country=Ukraine#hash - href

Search params

URL має властивість: url.searchParams, об'єкт типу URLSearchParams.

Він надає зручні методи для роботи з параметрами пошуку:

- append(name, value) – додати параметр з іменем name,
- delete(name) – видалити параметр з іменем name,
- get(name) – отримати значення параметру з іменем name,
- getAll(name) – отримати всі параметри, що мають ім'я name (наприклад, ?user=John&user=Pete),
- has(name) – перевірити чи існує параметр з іменем name,
- set(name, value) – встановити/замінити параметр з іменем name,
- sort() – відсортувати параметри за іменем, рідко стає в нагоді,
- …і це об'єкт також можна перебрати, подібно до Map.

Кодування

Набір символів, що можуть дозволено до використання в URL-адресах, визначено в стандарті RFC3986. Усі інші символи, що не дозволені стандартом, повинні бути закодовані. Наприклад, не латинські букви та пробіли мають бути замінені на їх UTF-8 коди, що починаються з %. Пробіл буде закодовано у вигляді %20.

URL об’єкт виконає всі перетворення автоматично.

До появи URL об’єктів, розробники використовували рядки для URL-адрес. Для цього є вбудовані функції:

- encodeURI – закодувати URL-адресу повністю. Кодує тільки символи, що заборонені до використання в URL.
- decodeURI – розкодувати її.
- encodeURIComponent – закодувати частину URL-адреси, наприклад, параметри пошуку, шлях чи хеш. закодує деякі символи та символи: # $ & + , / : ; = ? @.
- decodeURIComponent – розкодувати відповідну частину.

XMLHttpRequest

XMLHttpRequest – це вбудований об’єкт браузера, який дозволяє виконувати HTTP-запити за допомогою JavaScript.

У сучасній веб-розробці XMLHttpRequest використовується з трьох причин:

- З історичних причин: нам потрібно підтримувати наявні скрипти які використовують XMLHttpRequest.
- Нам потрібно підтримувати старі браузери і ми не хочемо використовувати поліфіли (наприклад, щоб зменшити кількість коду).
- Нам потрібен функціонал, який fetch ще не підтримує, наприклад відстежування ходу вивантажання.

XMLHttpRequest має два режими роботи: синхронний і асинхронний.

Асинхроний

- Створити XMLHttpRequest:

```
let xhr = new XMLHttpRequest();
```

- Іціалізувати його:

```
xhr.open(method, URL, [async, user, password])
```

Цей метод приймає основні параметри запиту:

- method – HTTP-метод. Зазвичай "GET" або "POST".
- URL – URL для запиту, зазвичай це рядок, але може бути і об’єктом URL.
- async – якщо явно встановлено значення false, тоді запит буде синхронним, ми розглянемо це трохи пізніше.
- user, password – логін та пароль для базової HTTP аутентифікації (якщо потрібно).

Асинхроний

- Надіслати запит:

```
xhr.send([body]);
```

- Прослуховувати події xhr.

Найчастіше використовуються ці три події:

- load – спрацьовує коли запит завершено (навіть якщо статус HTTP дорівнює 400 або 500) і відповідь сервера повністю завантажено.
- error – спрацьовує коли запит не може бути виконано, наприклад мережа не працює або задана не коректна URL-адреса.
- progress – періодично спрацьовує під час завантаження відповіді, та повідомляє, скільки байтів було завантажено.

Асинхроний

Після відповіді сервера, ми можемо отримати результат у таких властивостях xhr:

- Status - Код HTTP статусу (число): 200, 404, 403 і так далі, може бути 0 у разі помилки не пов’язаної з HTTP.
- StatusText - Статусне повідомлення HTTP (рядок): зазвичай OK для 200, Not Found для 404, Forbidden для 403 тощо.
- response (старі скрипти можуть використовувати responseText) - Тіло відповіді сервера.

Ми також можемо вказати час очікування відповіді за допомогою відповідної властивості:

```
xhr.timeout = 10000;
```

Тип відповіді

xhr.responseType:

- "" (за замовчуванням) – отримати як рядок,
- "text" – отримати як рядок,
- "arraybuffer" – отримати як ArrayBuffer (для двійкових даних),
- "blob" – отримати як Blob (для двійкових даних),
- "document" – отримати як XML-документ (може використовувати XPath та інші методи XML) або HTML-документ (на основі типу MIME для отриманих даних),
- "json" – отримати як JSON (автоматичний аналіз).

Стани запиту

XMLHttpRequest змінює свій стан в процесі виконання запиту. Поточний стан доступний у властивості xhr.readyState.

Специфікація

Стани:

- UNSENT = 0; // початковий стан
- OPENED = 1; // викликано метод open
- HEADERS_RECEIVED = 2; // отримано заголовки відповіді
- LOADING = 3; // завантажується відповідь (отримано пакет даних)
- DONE = 4; // запит завершено

Скасування запиту

Ми можемо скасувати запит у будь-який час. Це можна зробити за допомогою метода xhr.abort():

```
xhr.abort();  // скасовуємо запит
```

Синхронні запити

Якщо в методі open для третього параметра async встановлено значення false, то запит виконується синхронно. Іншими словами, виконання JavaScript призупиняється в момент виклику метода send() і відновлюється після отримання відповіді.

HTTP-заголовки

Існує 3 методи для роботи з HTTP-заголовками:

1. setRequestHeader(name, value) - Встановлює заголовок запиту із заданими ім’ям name та значенням value. Декількома заголовками керує виключно браузер, наприклад Referer і Host. Видалити заголовок неможливо
2. getResponseHeader(name) - Отримує заголовок відповіді за заданим ім’ям name (крім Set-Cookie і Set-Cookie2).
3. getAllResponseHeaders() - Повертає всі заголовки відповіді, крім Set-Cookie і Set-Cookie2.

POST, FormData

Щоб зробити запит POST, ми можемо використати вбудований об’єкт FormData.

Ми створюємо об’єкт, за бажанням додаємо дані з форми, та якщо потрібно, додаємо поля за допомогою метода append, а потім:

1. xhr.open('POST', ...) – встановлюємо метод POST.
2. xhr.send(formData) – та надсилаємо форму на сервер.

Форма надсилається з кодуванням multipart/form-data. Якщо нам більше подобається формат JSON, тоді використовуємо JSON.stringify і надсилаємо дані як рядок. Також встановлюємо заголовок Content-Type: application/json.

Хід завантаження

Подія progress спрацьовує лише на етапі завантаження даних з сервера.

Тобто якщо ми відправляємо POST запит, то XMLHttpRequest спочатку завантажує наші дані (тіло запиту) на сервер, а потім завантажує відповідь з сервера.

Існує ще один об’єкт без методів, призначений виключно для відстеження подій завантаження на сервер: xhr.upload.

Він генерує події, подібні до xhr, але xhr.upload ініціює їх виключно під час завантаження на сервер:

- loadstart – розпочато завантаження на сервер.
- progress – спрацьовує періодично під час завантаження.
- abort – завантаження перервано.
- error – помилка не пов’язана з HTTP.
- load – завантаження успішно завершено.
- timeout – час очікування завантаження минув (якщо встановлено властивість timeout).
- loadend – завантаження завершено (успішно або з помилкою).

Запити на інші джерела

XMLHttpRequest може робити запити інші джерела (сайти) використовуючи ту саму політику CORS, що й fetch.

Так само як і fetch, він за замовчуванням не надсилає іншим джерелам заголовки HTTP-авторизації та cookie. Щоб увімкнути їх, встановіть для xhr.withCredentials значення true

Тривале опитування

Тривале опитування (Long polling) – це найпростіший спосіб реалізувати постійне з'єднання із сервером, без використання спеціального протоколу, такого як WebSocket або Server Side Events.

Як це працює:

1. На сервер надсилається запит.
2. Сервер не закриває з'єднання, поки не з'явиться повідомлення для відправки.
3. Коли з'являється повідомлення – сервер відповідає ним на запит.
4. Браузер негайно робить новий запит.

WebSocket

WebSocket – це протокол, описаний у специфікації RFC 6455, і забезпечує спосіб обміну даними між браузером і сервером через постійне з'єднання. Дані можна передавати в обох напрямках у вигляді "пакетів", без розриву з'єднання та додаткових HTTP-запитів.

Синтаксис:

```
const socket = new WebSocket("ws://exapmle.com");
```

Важливо!

Завжди віддавайте перевагу wss://

Відкриття веб-сокета

Коли створюється new WebSocket(url), він починає підключатися негайно.

приклад заголовків браузера для запиту:

- Origin – джерело клієнтської сторінки, наприклад https://example.com. Об'єкти WebSocket є перехресними за своєю природою. Немає спеціальних заголовків чи інших обмежень.
- Connection: Upgrade – сигналізує про те, що клієнт хоче змінити протокол.
- Upgrade: websocket – запитуваний протокол – "websocket".
- Sec-WebSocket-Key – випадковий ключ, згенерований браузером, який використовується для забезпечення підтримки сервером протоколу WebSocket. Він є випадковим, щоб проксі-сервери не кешували будь-які наступні повідомлення.
- Sec-WebSocket-Version – версія протоколу WebSocket, 13 є поточною.

Передача даних

Зв’язок WebSocket складається з “фреймів” – фрагментів даних, які можуть бути відправлені будь-якою стороною і можуть бути кількох видів:

- “текстові фрейми” – містять текстові дані, які сторони надсилають одна одній.
- “фрейми двійкових даних” – містять бінарні дані, які сторони надсилають одна одній.
- “ping/pong фрейми” використовуються для перевірки з’єднання, надсилаються сервером, браузер реагує на них автоматично.
- також існує “фрейм закриття з’єднання” та кілька інших сервісних фреймів.

У браузері ми безпосередньо працюємо тільки з текстовими або двійковими фреймами.

Метод WebSocket .send() може надсилати текстові або двійкові дані.

Коли ми отримуємо дані, текст завжди надходить у вигляді рядка. А для двійкових даних ми можемо вибирати між форматами Blob і ArrayBuffer.

Закриття з’єднання

Зазвичай, коли одна зі сторін хоче закрити з’єднання (і браузер, і сервер мають рівні права), вони надсилають “фрейм закриття з’єднання” з цифровим кодом і текстовою причиною.

Метод для цього:

socket.close([code], [reason]);

- code це спеціальний код закриття WebSocket (необов’язковий параметр)
- reason це рядок, який описує причину закриття (необов’язковий параметр)

Дякую за увагу