Тема 2. CORS

Теперь поговорим об основе основ безопасности в вебе. Как мы уже знаем, почти все, если не вообще все приложения в вебе работают по принципу клиент-серверной архитектуры. Любой более-менее серьёзный сайт будет делать запросы на бэкенд для получения и последующего отображения данных.

И очень может быть, что вы уже встречали такие ошибки в своей работе. Возникают они, как правило, в консоли вашего браузера:

```
No 'Access-Control-Allow-Origin' header is present on the requested
resource.

Cross-Origin Request Blocked: The Same Origin Policy disallows reading
the remote resource at https://example.com/
```

Если нет — не беда, вы их точно встретите и не раз. Разберёмся, что это, и для начала попробуем воспроизвести эти ошибки самостоятельно.

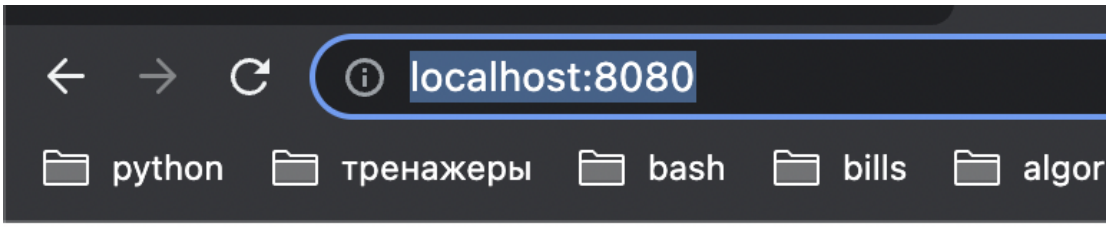
Для начала напишем простейший бэкенд:

```
from flask import Flask, jsonify

app = Flask(__name__)


@app.route('/')
def handler():
   return jsonify({"Hello": "User"})


if __name__ == '__main__':
   app.run(port=8080)
```

Запустим его и отправимся в консоль разработчика в браузере. Тут нам нужно будет зайти во вкладку консоли — по сути, это та же консоль, что и для питона в терминале, только с JavaScript. Тут мы попробуем написать простые скрипты, которые будут имитировать работу полноценного фронтенд-приложения.

Итак, как работает веб в клиент-серверной архитектуре? Где-то запущенно приложение, например на питоне, и ждёт HTTP-запросов. Пользователь открывает наш сайт, мы отдаём ему статику — это файлы с HTML-, JS- И CSS-кодом. Браузер начинает это дело рендерить и попутно выполнять JS-код. Этот код может запросить у бэкенда для отображения некоторые данные. И делает AJAX-запрос, то есть асинхронный JS-запрос. Бэкенд его обрабатывает, отдаёт какой-то json фронтенду, и тот отображает список с котиками.

Для начала откроем в браузере страничку с нашим API и получим json с бэкенда:

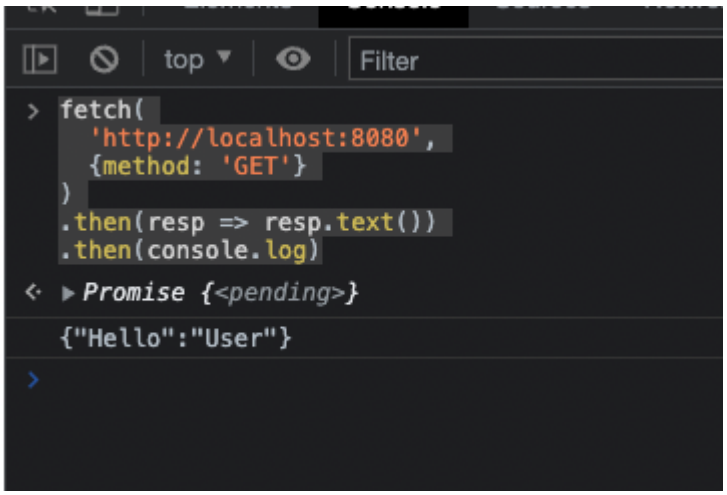



```
{"Hello":"User"}
```

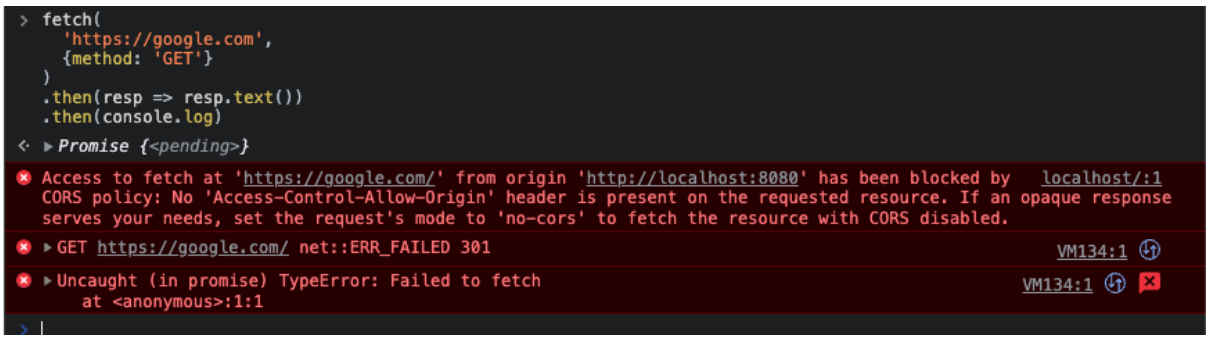
Открылось! А теперь откроем консоль разработчика и перейдём в консоль. Тут попробуем написать простой JS-запрос, который сделает HTTP-запрос на бэкенд. По сути это то же самое, что вызвать из Python библиотеку request, только мы это делаем из браузера и от имени открытой страницы. Давайте посмотрим.

```
fetch(
  'http://localhost:8080',
  {method: 'GET'}
)
.then(resp => resp.text())
.then(console.log)
```

Если мы выполним этот запрос, то увидим в консоли тот же json, что и в самом браузере.



Как вы понимаете, это ровно то, что делает любое фронтенд-приложение. А теперь попробуем из консоли открыть какой-нибудь другой сайт, например Google.



И увидим такую ошибку. Буквально тут написано следующее: попытка открыть ресурс Google с другого origin (это можно перевести как источник) localhost была заблокирована. Вот мы и увидели CORS в действии.

Разберёмся, что же произошло.

Когда-то давно весь сайт, то есть его HTML-код, был обычной статической страницей. Потом по чуть-чуть стали появляться теги, которые говорили браузеру, что нужно сделать уже после загрузки основного контента. Например, это тег image или iframe. Первый говорил, где браузер может загрузить картинку, а второй говорит браузеру, что ему нужно открыть ещё одну веб-страницу внутри текущей — так называемый фрейм (ссылку про то, что это такое, ищите под видео). Как вы понимаете, вполне может быть, что источник запроса и источник ответа могут различаться, как на примере кода у нас в консоли браузера: мы сделали запрос из локалхоста на другой источник — Google.

А что такое источник? Посмотрим на примере. Во-первых, под источником понимается тройка параметров: схема, хост и порт.

Например:

https://google.com:433 — это схема https, имя хоста google.com и стандартный порт для https-трафика. И получается, что если хотя бы один из этих параметров будет отличаться, то у нас включится cross-origin-взаимодействие.

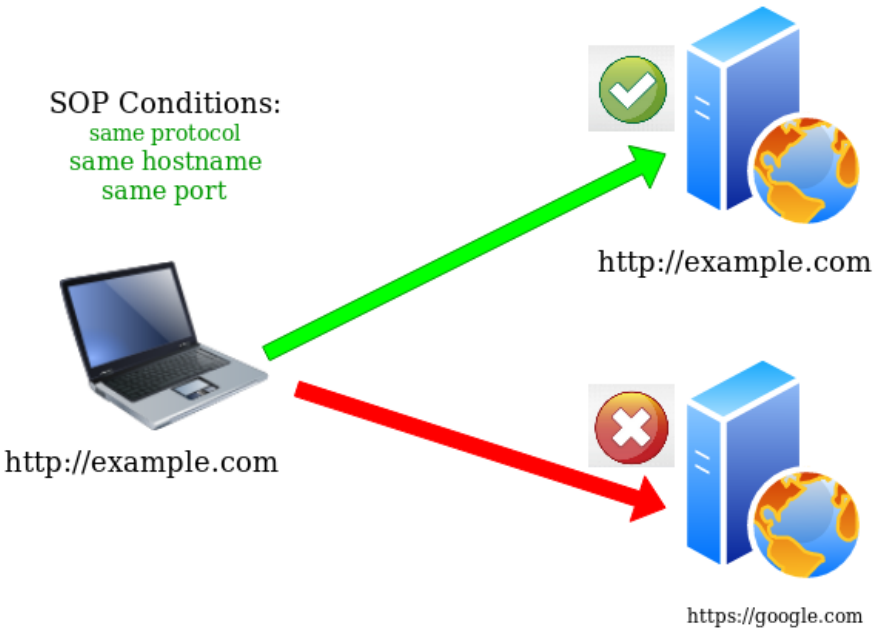
Попробуем поработать браузером и определим, где источник отличается, а где нет.

https://google.com/search — не отличается, отличен только путь.

http://google.ru — отличается схема и хост — вместо com тут ru.

https://google.com:80 — отличается, не тот порт.

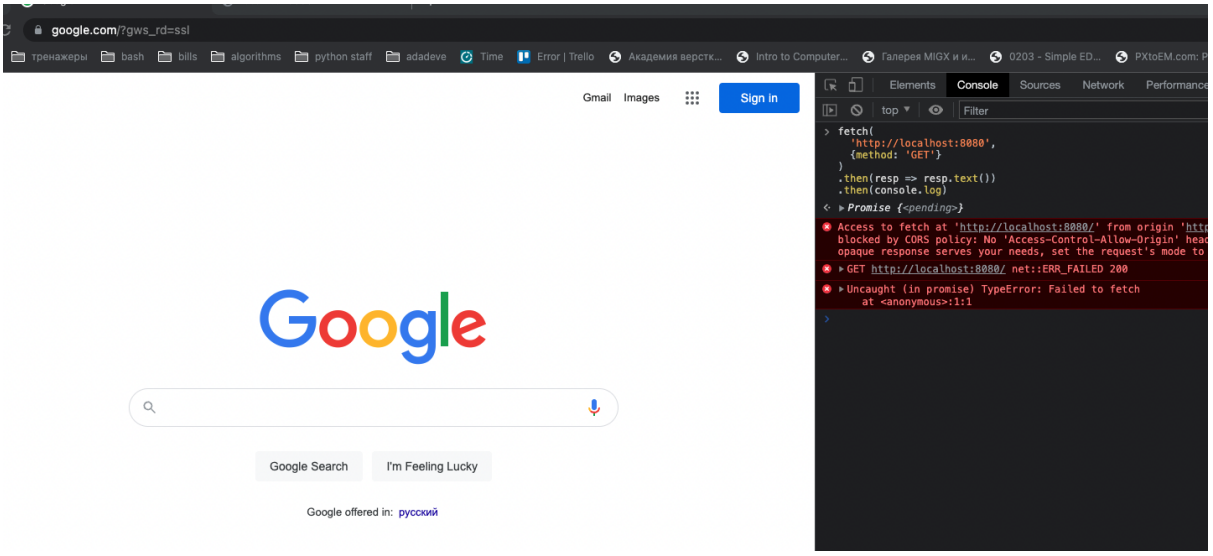
В чём проблема таких запросов? Предположим, ваша бабуля прислала ссылку на сайт с заголовком вроде «Стоит только выпить эту таблетку…» и вы, сама наивность, на неё нажали. Заходите на этот желтушный сайт и уже думаете закрыть вкладку, как внезапно приходит СМС о переводе средств с вашего счёта не пойми кому. А всё потому, что под капотом сайт сделал запрос на API банка на перевод денег — и вуаля, прощайте ваши денежки.



Как способ борьбы с такими атаками, в вебе есть так называемое правило одинакового источника, или по-английски same origin policy. По умолчанию браузер разрешит получить доступ только к ресурсам из совпадающего источника. Поэтому мы смогли выполнить скрипт-запрос на локалхост с локалхоста, но не смогли выполнить запрос на Google.

Окей, правило замечательное, но как тогда нам работать? Неужели придётся хранить всю статику у себя на сервере? А если наш бэкенд находится на другом хосте? А ведь так чаще всего и бывает. Чтобы запросы были безопасными и чтобы их вообще можно было делать, в браузер внедрён механизм CORS — Cross-Origin Resource Sharing, что можно перевести как технология совместного использования ресурсов из разных источников. То есть это некоторые правила, которые всё же позволят нам сделать запрос на другой источник и избежать ошибки, которую мы только что видели. Как это работает под капотом? Клиент, то есть браузер, автоматически подставляет хедер origin с полным названием хоста, который делает этот запрос.

То есть если мы сейчас сделаем обратную штуку, попробуем вызвать со страницы Google наш бэкенд, то ожидаемо увидим ошибку.



Если мы в это же время будем смотреть на логи сервера, то увидим, что он успешно отработал запрос, который вернулся браузеру, но тот отказался его рендерить. И мы можем легко убедиться в том, что хедер присутствует в запросе:

```python
from flask import Flask, jsonify, request

app = Flask(__name__)


@app.route('/', methods=['GET'])
def handler():
    print(request.headers)
    return jsonify({"Hello": "User"})


if __name__ == '__main__':
    app.run(port=8080, debug=True)
```

Посмотрим, какие хедеры приходят нам в запросе:

```
127.0.0.1 - - [08/Feb/2022 18:13:25] "GET / HTTP/1.1" 200 -
Host: localhost:8080
Connection: keep-alive
Sec-Ch-Ua: " Not A;Brand";v="99", "Chromium";v="98", "Google Chrome";v="98"
Sec-Ch-Ua-Mobile: ?0
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/98.0.4758.80 Safari/537.36
Sec-Ch-Ua-Platform: "macOS"
Accept: */*
Origin: https://www.google.com
Sec-Fetch-Site: cross-site
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: https://www.google.com/
Accept-Encoding: gzip, deflate, br
Accept-Language: en-GB,en;q=0.9
```

Видим, что браузер сам подставил нужный хедер в запрос. В то же время если мы выполним тот же запрос с открытой страницы локалхоста, то этого хедера в запросе не будет.

Из такого поведения понятно, что теперь нужно что-то сделать бэкенду, чтобы браузер согласился отрендерить запрос. Наверное, вы догадываетесь, что тут речь идёт о каких-то хедерах.

Есть целая группа хедеров, которые начинаются с Access-Control- и далее некая настройка, которые управляют поведением cors. Самый важный из них — Access-Control-Allow-Origin.

Попробуем добавить его в наш сервер. Кстати, очевидно, что все ответы нашего сервера должны иметь такой хедер, и будет не слишком удобно каждый раз руками добавлять его внутрь каждого из эндпоинтов.

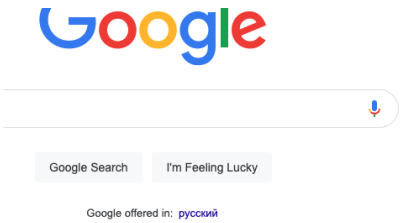
Сделаем это красиво. Для этого воспользуемся декоратором after_request:

```python
from flask import Flask, jsonify, request, Response

app = Flask(__name__)


@app.route('/', methods=['GET'])
def handler():
    print(request.headers)
    return jsonify({"Hello": "User"})


@app.after_request
def set_cors(response: Response):
    response.headers['Access-Control-Allow-Origin'] = 'https://google.com/'
    return response

if __name__ == '__main__':
    app.run(port=8080, debug=True)
```

Попробуем вызвать наш сервер из окна браузера:

```
> fetch(
    'http://localhost:8080',
    {method: 'GET'}
  )
  .then(resp => resp.text())
  .then(console.log)
◟ ▸ Promise {<pending>}
✖ Access to fetch at 'http://localhost:8080/' from origin 'https://www.google.com' has been    www.google.com/:1
  blocked by CORS policy: The 'Access-Control-Allow-Origin' header has a value 'https://google.com/' that is not
  equal to the supplied origin. Have the server send the header with a valid value, or, if an opaque response
  serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.
✖ ▸ GET http://localhost:8080/ net::ERR_FAILED 200                                          VM69:1
✖ ▸ Uncaught (in promise) TypeError: Failed to fetch                                        VM69:1
      at <anonymous>:1:1
>
```

Ошибка всё ещё есть, но сообщение уже другое. Видим, что мы немного ошиблись в названии хоста — забыли указать www в начале. Исправим это.



Работает — мы победили CORS. Заметим также, что указать единовременно мы можем только один источник. Либо вместо конкретного origin поставить звёздочку, так мы разрешим вообще всё. Но пользуйтесь такой возможностью с осторожностью.

Помимо allow-origin есть ещё несколько хедеров:

- Access-Control-Allow-Methods — указывает список методов, которые проходят CORS-политику.
- Access-Control-Allow-Headers — указывает список хедеров, которые разрешены CORS.
- Access-Control-Max-Age — указывает, сколько времени браузер может кешировать информацию, полученную из двух предыдущих хедеров.

Окей, на этом про CORS всё. Теперь нас точно не испугает страшное сообщение об ошибке cors в консоли браузера, и мы знаем, что это всего лишь мера безопасности, которая делает интернет чуточку менее опасным местом. И самое важное: мы знаем, как этим управлять. Впереди немного практики, и мы продолжим изучать потенциальные проблемы в безопасности веба.