Cross-Origin Resource Sharing (CORS)

- Cross-Origin Resource Sharing (CORS) is a security feature implemented by web browsers to control how web applications interact with resources hosted on different domains (origins).
- It ensures that web applications from one origin cannot arbitrarily make requests to another origin without explicit permission from the server being requested.
- CORS is a mechanism that uses HTTP headers to tell the browser whether a web page can access resources from a different origin. It allows servers to specify who can access their resources and how these requests should be made.
- CORS allows a server to explicitly permit some cross-origin requests, which is otherwise restricted by the Same-Origin Policy.
- CORS ensures that resources hosted on different origins can only be accessed by trusted origins by enforcing a policy where servers must explicitly allow cross-origin requests.
- For fetch requests, this affects how the browser handles requests and responses, requiring correct configuration of CORS headers on the server to enable smooth interaction between the client and server.

How CORS Works?

When a browser makes a cross-origin HTTP request, the following process takes place:

- **Preflight Request (For Certain Requests):** For requests that aren't considered "simple", the browser sends an OPTIONS request before the actual request. This is called a preflight request. Its purpose is to check whether the server allows the main request.
- **Simple Request:** If a request is considered simple (for instance, GET or POST without special headers), the browser sends it directly without a preflight check. However, the server still has to include the correct CORS headers in its response.
- **Server Response:** The server responds with CORS-related headers (like Access-Control-Allow-Origin) that instruct the browser whether or not to grant access to the resource.

CORS Request Flow

- Browser sends a request (or preflight request in some cases) to the server.
- Server responds with CORS headers, either permitting or denying the request.
- Browser checks the CORS headers to determine if the response should be accessible by the client-side script.

Understanding Origins

In the web, an origin is defined as a combination of:

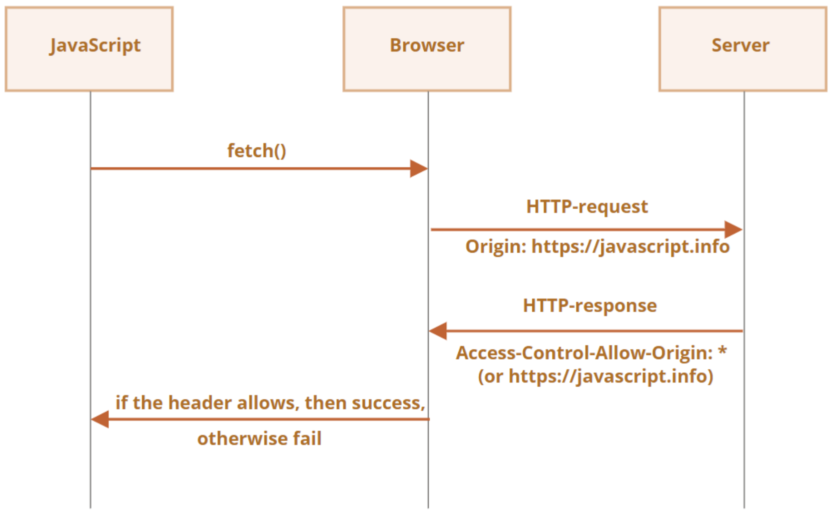
- Protocol: http:// or https://
- Domain: example.com
- Port (if applicable): :3000 or :80

So, two URLs like https://example.com:3000 and http://example.com have different origins due to the protocol and port. Same-origin policy (SOP) enforces that a web page can only make requests to the same origin unless the server explicitly allows it.

Types of CORS Requests

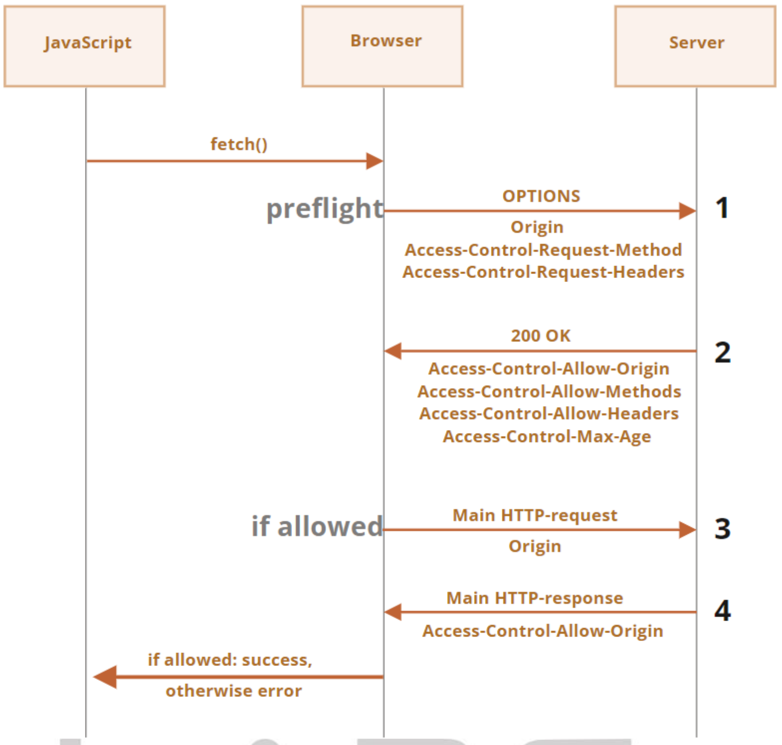
- **Simple Requests (Safe Request):** A request is considered "simple" if it satisfies the following conditions:
  - Uses one of these methods: GET, POST, or HEAD.
  - No custom headers are added (except for Content-Type, Accept, etc.)
  - The Content-Type is one of: application/x-www-form-urlencoded, multipart/form-data, or text/plain.

  For simple requests, the browser skips the preflight check and directly sends the request to the server.


JavaScript
Browser
Server
fetch()
HTTP-request
Origin: https://javascript.info
HTTP-response
Access-Control-Allow-Origin: *
(or https://javascript.info)
if the header allows, then success,
otherwise fail

- **Preflighted Requests (Unsafe Request):** A request triggers a preflight if:
  - It uses a method other than GET, POST, or HEAD (e.g., PUT, DELETE).
  - It includes custom headers.
  - It contains certain types of Content-Type like application/json.

  The browser sends an OPTIONS request to the server, and the server must respond with appropriate CORS headers for the main request to proceed.



Step 1 (preflight request): Prior to sending such a request, the browser, on its own, sends a preflight request that looks like this:

```
OPTIONS /service.json
Host: www.xyz.com
Origin: https://javascript.info
Access-Control-Request-Method: PATCH
Access-Control-Request-Headers: Content-Type,API-Key
```

- Method: OPTIONS.
- The path – exactly the same as the main request: /service.json.
- Cross-origin special headers:
  - Origin – the source origin.
  - Access-Control-Request-Method – requested method.
  - Access-Control-Request-Headers – a comma-separated list of "unsafe" headers.

Step 2 (preflight response): The server should respond with status 200 and the headers:

```
Access-Control-Allow-Origin: https://javascript.info
Access-Control-Allow-Methods: PATCH
Access-Control-Allow-Headers: Content-Type,API-Key.
```

That allows future communication, otherwise an error is triggered.

If the server expects other methods and headers in the future, it makes sense to allow them in advance by adding them to the list.

For example, this response also allows PUT, DELETE and additional headers:

```
200 OK
Access-Control-Allow-Origin: https://javascript.info
Access-Control-Allow-Methods: PUT,PATCH,DELETE
Access-Control-Allow-Headers: API-Key,Content-Type,If-Modified-Since,Cache-Control
Access-Control-Max-Age: 86400
```

Now the browser can see that PATCH is in Access-Control-Allow-Methods and Content-Type, API-Key are in the list Access-Control-Allow-Headers, so it sends out the main request.

Step 3 (actual request): When the preflight is successful, the browser now makes the main request. The process here is the same as for safe requests.
The main request has the Origin header (because it's cross-origin):

```
PATCH /service.json
Host: site.com
Content-Type: application/json
API-Key: secret
Origin: https://javascript.info
```

Step 4 (actual response): The server should not forget to add Access-Control-Allow-Origin to the main response. A successful preflight does not relieve from that:

```
Access-Control-Allow-Origin: https://javascript.info
```

Then JavaScript is able to read the main server response.

- **Credentialed Requests:** Requests that include credentials (like cookies, HTTP authentication) need the server to explicitly allow them. The server must include the header Access-Control-Allow-Credentials: true, and the client must make the request with credentials: 'include' in the fetch request.

  A request with credentials is much more powerful than without them. If allowed, it grants JavaScript the full power to act on behalf of the user and access sensitive information using their credentials.

  Does the server really trust the script that much?
  Then it must explicitly allow requests with credentials with an additional header.

  To send credentials in fetch, we need to add the option credentials: "include", like this:

```
fetch('http://xyz.com', {
  credentials: "include"
});
```

  Now fetch sends cookies originating from xyz.com with request to that site.

  If the server agrees to accept the request with credentials, it should add a header Access-Control-Allow-Credentials: true to the response, in addition to Access-Control-Allow-Origin.

  For example:

```
200 OK
Access-Control-Allow-Origin: https://javascript.info
Access-Control-Allow-Credentials: true
```

  Please note: Access-Control-Allow-Origin is prohibited from using a star * for requests with credentials. Like shown above, it must provide the exact origin there. That's an additional safety measure, to ensure that the server really knows who it trusts to make such requests.

Impact of CORS on Fetch

When using the Fetch API (or similar client-side HTTP libraries like Axios), CORS can impact how requests are handled:

- **Same-Origin Requests:** No CORS headers are needed if the request is made to the same origin.
- **Cross-Origin Requests (with CORS):** If a request is made to a different origin, the browser will expect the correct CORS headers in the response. If these headers are missing or incorrect, the browser blocks access to the response, and the fetch call fails with a CORS error.
- **Preflight Requests in Fetch:** If your fetch request involves custom headers or uses methods like PUT, the browser will automatically issue a preflight request. The response from this request must include the correct CORS headers; otherwise, the browser will block the actual request.
- **Handling Cookies with Fetch:** By default, fetch requests don't include credentials like cookies. If you want to send cookies with a cross-origin request, you must set the credentials option in fetch to 'include', and the server must allow credentials with the Access-Control-Allow-Credentials header.

CORS Errors in Fetch

Common CORS errors and their causes:

- **No 'Access-Control-Allow-Origin' header:** This occurs when the server doesn't return the CORS headers, meaning the browser blocks access.
- **Preflight response doesn't succeed:** This error can happen if the server doesn't handle the OPTIONS request properly or doesn't allow the requested method or headers.
- **Credentials flag and Access-Control-Allow-Origin mismatch:** If you request with credentials (credentials: 'include'), but the server returns Access-Control-Allow-Origin: *, the browser will block the request.

Security Implications: While CORS is essential for enabling secure cross-origin communication, it can expose vulnerabilities if not configured properly. For example:

- Allowing * (wildcard) in Access-Control-Allow-Origin opens up resources to any website, which may be unsafe.
- Incorrect handling of credentialed requests can lead to Cross-Site Request Forgery (CSRF) attacks.

Example

Server-Side (Express.js)   //api.example.com/data

```
app.use((req, res, next) => {
  res.header('Access-Control-Allow-Origin', 'https://your-frontend.com');
  res.header('Access-Control-Allow-Methods', 'GET,POST,PUT,DELETE');
  res.header('Access-Control-Allow-Headers', 'Content-Type, Authorization');
  res.header('Access-Control-Allow-Credentials', 'true');
  next();
});
```

Client-Side (Fetch with CORS)

```
fetch('https://jsonplaceholder.typicode.com/photos', {
            method: 'GET',
            credentials: 'include', // include cookies
            headers: {
                        'Content-Type': 'application/json'
            }

})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error('CORS error:', error));
```

Example: Simple GET Request (Same-Origin)

When the request is made to the same origin (same domain, protocol, and port), so CORS is not required.

```
fetch('/api/data')  // Same-origin request
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

Example: GET Request to a Different Origin (Cross-Origin Request)

When we are requesting data from a different origin (CORS is required). In this case, the server must allow cross-origin requests by setting Access-Control-Allow-Origin in its response headers.

Client-Side

```
fetch('https://jsonplaceholder.typicode.com/photos')  // Cross-origin request
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('CORS error:', error));
```

Server-Side

For this request to work, the external server must respond with something like:

```
Access-Control-Allow-Origin: *
```

or

```
Access-Control-Allow-Origin: https://xyz.com
```

Example: POST Request with Simple Headers (Cross-Origin)

In this example, you send data using a POST request to a different origin. The request uses simple headers (Content-Type: application/x-www-form-urlencoded), so no preflight request is triggered.

```
fetch('https://jsonplaceholder.typicode.com/photos/upload', {
          method: 'POST',
          headers: {
            'Content-Type': 'application/x-www-form-urlencoded'  // Simple header
  },
  body: new URLSearchParams({ name: 'Udit Chauhan', age: '26' })
})
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('CORS error:', error));
```

In this case, the server must allow cross-origin POST requests and set the appropriate CORS headers in the response.

Example: Request with Custom Headers (Preflight Request)

When you add custom headers (like Authorization), the browser will first send a preflight OPTIONS request to check if the server allows this.

Client-Side

```
        fetch('https://jsonplaceholder.typicode.com/photos/protected ', {
                method: 'GET',
                headers: {
                  'Authorization': 'Bearer some-token',  // Custom header triggers preflight
                  'Content-Type': 'application/json'       }
        })
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('CORS error:', error));
```

Server-Side: In this case, the server must allow custom headers and respond to the preflight request with:

```
        Access-Control-Allow-Origin: https://your-website.com
        Access-Control-Allow-Methods: GET
        Access-Control-Allow-Headers: Authorization
```

Example: Request with Credentials (Cookies or Authentication)

If you need to include cookies or other credentials, you need to set the credentials option in the Fetch API, and the server must respond with Access-Control-Allow-Credentials.

Client Side

```
        fetch('https://jsonplaceholder.typicode.com/photos/user', {
                        method: 'GET',
                        credentials: 'include',  // Include cookies or credentials
                        headers: {
                                    'Content-Type': 'application/json'  }
        })
          .then(response => response.json())
          .then(data => console.log(data))
          .catch(error => console.error('CORS error:', error));
```

Server-Side: In this case, the server must allow credentials:

```
        Access-Control-Allow-Origin: https://your-website.com
        Access-Control-Allow-Credentials: true
```

AbortController

- In JavaScript, you can abort a Fetch request using the AbortController API.
- This is useful when you want to cancel a fetch operation, either due to user interaction (like canceling a form submission) or time-sensitive operations (like aborting a slow network request).
- The AbortController is a JavaScript API that provides a mechanism to abort web requests like fetch, as well as other asynchronous operations that can be canceled.
- It was introduced to give developers more control over long-running or unnecessary operations and has since become a key part of modern web development, particularly in cases of handling user interactions and network requests.
- AbortController was introduced to:
  - **Prevent wasteful operations:** By truly aborting a request, you save processing power and bandwidth on both the client and server side.
  - **Better control over asynchronous flows:** Developers can now handle complex scenarios such as aborting long-running requests, canceling form submissions when the user navigates away, or stopping repeated requests in real-time applications.
  - **Resource optimization:** By canceling unnecessary requests, AbortController allows for more efficient use of system resources, reducing memory and CPU usage.

AbortController API

- **AbortController:** A constructor that creates an instance of AbortController.

```
const controller = new AbortController();
```

- **abort():** The method you call to cancel the request. When abort() is called, any associated fetch or other signal-sensitive operation is immediately aborted, and an AbortError is thrown.

```
controller.abort(); // Aborts the associated fetch request
```

- **signal:** The property that is used to pass the abort signal to fetch or any other process that supports AbortSignal. The signal listens for the abort() method and cancels the request.

```
fetch('https://jsonplaceholder.typicode.com/users', { signal: controller.signal
})
```

How AbortController Works

1. **Creating an AbortController:** The AbortController is an object that you can instantiate. Each instance has two important properties:
   - controller: An instance of AbortController.
   - signal: A signal property from the controller that you pass to the operation (e.g., fetch) you want to control.

   ```
   const controller = new AbortController();
   ```

2. **Using the signal Property:** The signal is passed as an option to an asynchronous operation, like fetch. The signal essentially links the request to the controller, allowing you to control the request.

   ```
   fetch('https://jsonplaceholder.typicode.com/users', { signal: controller.signal
   })
   ```

3. **Calling abort():** When you want to stop the operation, you call the abort() method on the AbortController. This will abort the associated asynchronous operation and trigger an AbortError in the request, which can be caught in a catch block.

   ```
   controller.abort(); // Aborts the fetch request
   ```

4. **Handling Abort in Promises:** When a fetch request is aborted, it rejects the promise with an AbortError. You can catch this specific error and handle it as needed.

   ```
   fetch('https://jsonplaceholder.typicode.com/users', { signal: controller.signal
   })
     .then(response => response.json())
     .then(data => console.log(data))
     .catch(err => {
       if (err.name === 'AbortError') {
         console.log('Fetch request aborted');
       } else {
         console.error('Fetch error:', err);
       }
     });
   ```

Advantages of Using AbortController

- **User-Driven Abortion:** In situations where a user wants to cancel an ongoing action (e.g., canceling a form submission or halting a search request), you can use AbortController to stop the request immediately and prevent any unnecessary work from being done.

  Example: Cancel a form submission.

  ```
  const controller = new AbortController();
  document.getElementById('cancel-btn').addEventListener('click', () => {
    controller.abort(); // Cancel the form submission request
  });
  fetch('https://jsonplaceholder.typicode.com/users', { signal: controller.signal
  })
    .then(response => response.json())
    .then(data => console.log('Data:', data))
    .catch(error => {
      if (error.name === 'AbortError') {
        console.error('Request was aborted due to timeout');    }
    });
  ```

- **Performance Optimization:** Instead of letting old requests finish when they are no longer needed, aborting them prevents unnecessary network traffic and processing on the server.

- **Timeout Handling:** You can abort a slow network request if it exceeds a given time limit, ensuring that your app doesn’t hang indefinitely waiting for responses from unreliable endpoints.

  Example: Set a timeout to abort a request.

  ```
  const controller = new AbortController();
  const timeoutId = setTimeout(() => controller.abort(), 5000);
  fetch('https://jsonplaceholder.typicode.com/photos', { signal:
  controller.signal })
    .then(response => response.json())
    .then(data => console.log('Data:', data))
    .catch(error => {
      if (error.name === 'AbortError') {
        console.error('Request was aborted due to timeout');
      }  });
  ```

- **Handling Multiple Requests:** When a user makes repeated requests (e.g., in a search field with suggestions), you can cancel the previous request whenever a new one is initiated, ensuring that you only process the latest request.
  Example: Cancel old search requests as new ones are triggered.

```
let lastController;
function search(id) {
  if (lastController) lastController.abort(); // Abort previous request
  const controller = new AbortController();
  lastController = controller;
  fetch(`https://jsonplaceholder.typicode.com/photos/?id=${id}`, { signal: controller.signal })
    .then(response => response.json())
    .then(data => console.log('Search results:', data))
    .catch(err => {
      if (err.name === 'AbortError') {
        console.log('Previous request aborted');
      } else {
        console.error('Fetch error:', err);
      }
    });
}
search(11);
search(1001); // This will abort the search request for id=11.
```

Limitations and Considerations

- **Not Supported in All APIs:** While AbortController is useful in canceling fetch requests, it isn't natively supported by all asynchronous operations. However, many browser APIs are gradually adopting it (e.g., WebSocket, EventSource).
- **Error Handling:** Aborted requests throw an AbortError. This requires careful handling in your catch blocks to ensure that the application gracefully recovers from these canceled requests and doesn't treat them like traditional errors.
- **Can Only Be Aborted Once:** An AbortController signal can only be used to abort a request once. After calling abort(), that controller is no longer usable for other requests. You need to create a new instance for additional abort operations.