The N-server problem

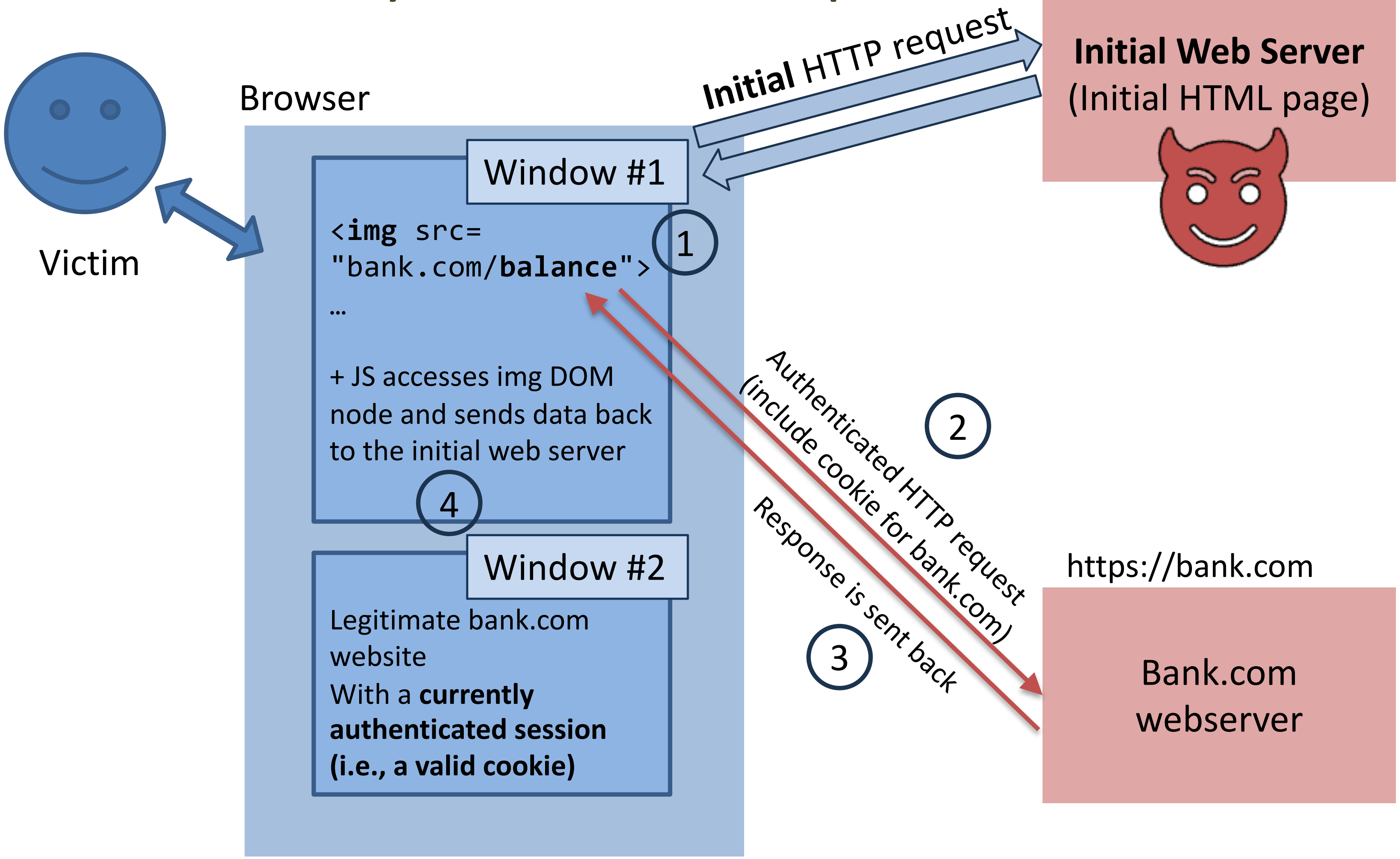
Cross-Origin Request Sharing

Enrico Masala
Fulvio Corno
Luigi De Russis

Loading Web Pages

- Loading a web page requires to load external resources (images, CSS, Javascript, …), potentially from different web servers
- Loading resources from different web servers without limitations was a "normal" behavior in the really old days of the Internet
- Now Internet has evolved into a platform able to run applications that can access/manipulate data, even personal/private data
- Browsers can also keep more than one web page open at the same time
- **How to ensure that private data is not shared across web pages coming from different servers?**
    - Especially when browsers have authenticated sessions active for websites

Security Risk Example



Victim
Browser
Window #1
<img src=
"bank.com/balance">
…
+ JS accesses img DOM node and sends data back to the initial web server
1
4
Window #2
Legitimate bank.com website
With a currently authenticated session (i.e., a valid cookie)
Initial HTTP request
https://tricky.com
Initial Web Server
(Initial HTML page)
Authenticated HTTP request
(include cookie for bank.com)
2
Response is sent back
3
https://bank.com
Bank.com webserver

1. The user visits a malicious website that creates a DOM <img> node that references https://bank.com and not https://tricky.com
2. The request is sent by the browser, **with cookies**, to https://bank.com
3. Private data is retrieved (the image is not shown since the data is not an image)

4. **Without additional restrictions, the private data could be accessed** via the <img> DOM object by the JS running in Window #1, which then can send such data to the initial malicious website

- **Note**: The browser must load the image src URL in any case for backward compatibility

Website separation

- To limit the previous security risks, now browsers **do not allow** DOM and JS access data and cookies coming from servers which are not the initial one
- However, to avoid “breaking the old web”, requests for `<img>`, `<link>`, `<script>` are still sent even if URLs point to different servers
  - And the received content is also rendered and incorporated in the page if it can be parsed correctly (images, CSS, Javascript)
- But <u>DOM and JS cannot access</u> such loaded content, to <u>avoid data stealing</u>
- This is because the response of such requests could include private data if sent to a server for which an authenticated session is already active (valid cookies)
  - In fact, browsers always send cookies <u>automatically</u> to the server that initially set them

Same Origin Policy (SOP)

- Website separation in practice: browsers implement the **Same Origin Policy (SOP)**, i.e., content and data coming from **origins** different from the initial server can be loaded but not freely accessed by the page

- What is an "origin"?
- An **origin** consists of a **URI scheme**, **domain** and **port number**:

```
http://example.com:3456/site/
```

Origin Comparison

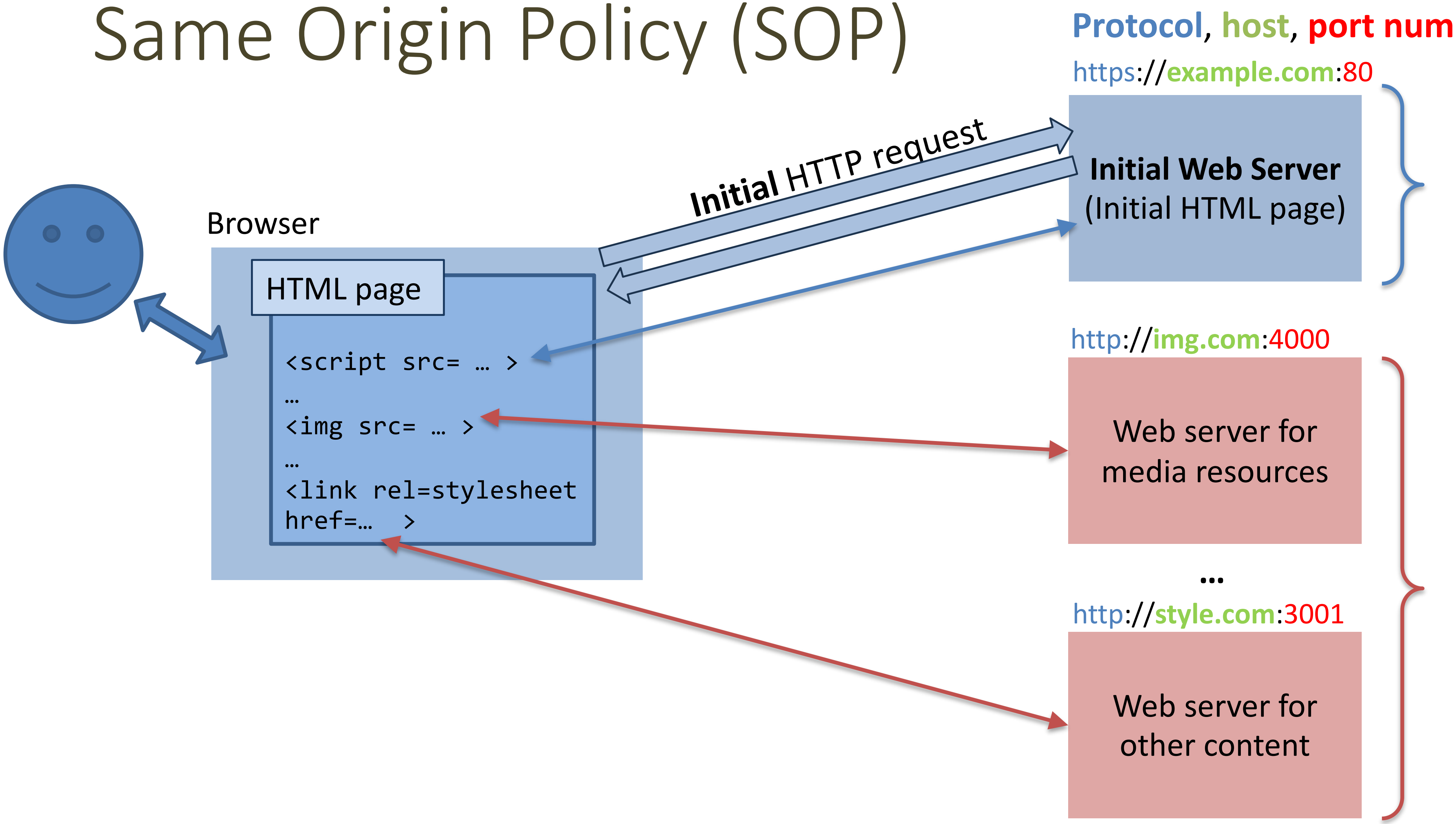
- An origin is the same only if presents the **same URI scheme**, **domain** and **port** number
- Consider this starting URL: `http://website.com/site/page.html`

New URL	Same Origin?
`http://website.com/example`	Yes: same scheme, domain, port
`http://website.com/site-two`	Yes: same scheme, domain, port
`**https**://website.com/example`	No: different scheme **and (implicit) port**
`http://**en**.website.com/example`	No: different domain
`http://**www**.website.com/example`	No: different domain
`http://website.com:**8080**/example`	No: different port

Note: implicit ports for http (80) and https (443)

https://portswigger.net/web-security/cors/same-origin-policy

Same Origin Policy (SOP)



```
<script src= … >
…
<img src= … >
…
<link rel=stylesheet
href=…  >
```

Everything is allowed: **Same Origin Policy (SOP)**

Resources are rendered and incorporated in the page but cannot be accessed by DOM and JS

What if they also need to be accessed?

Handling the Cross-Origin situation

- Option #1: do NOT load data except from the initial server (which is trusted by definition)
  - This approach is **too limiting** for modern web application where data can reside on (several) different web servers
- Option #2: Find a mechanism to allow Cross-Origin data access in a controlled fashion
  - Allow to separate the load onto different servers
  - Allow to use any other server, in particular API servers (even 3rd party ones, even more than one)

Mozilla Developer Network:
Web technology for developers —
HTTP — Cross-Origin Resource Sharing (CORS)
https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

Accessing multiple websites

CROSS-ORIGIN REQUEST SHARING (CORS)

Solving the Cross-Origin Issue: CORS

- Cross-Origin Resource Sharing (**CORS**): a **standard** mechanism to **handle cross-domain requests**
- CORS defines a **set of HTTP headers** that allow the browser and server to communicate about which requests are safe to be accessed by DOM and JS
- The **server** defines which origins are accepted for any request

https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

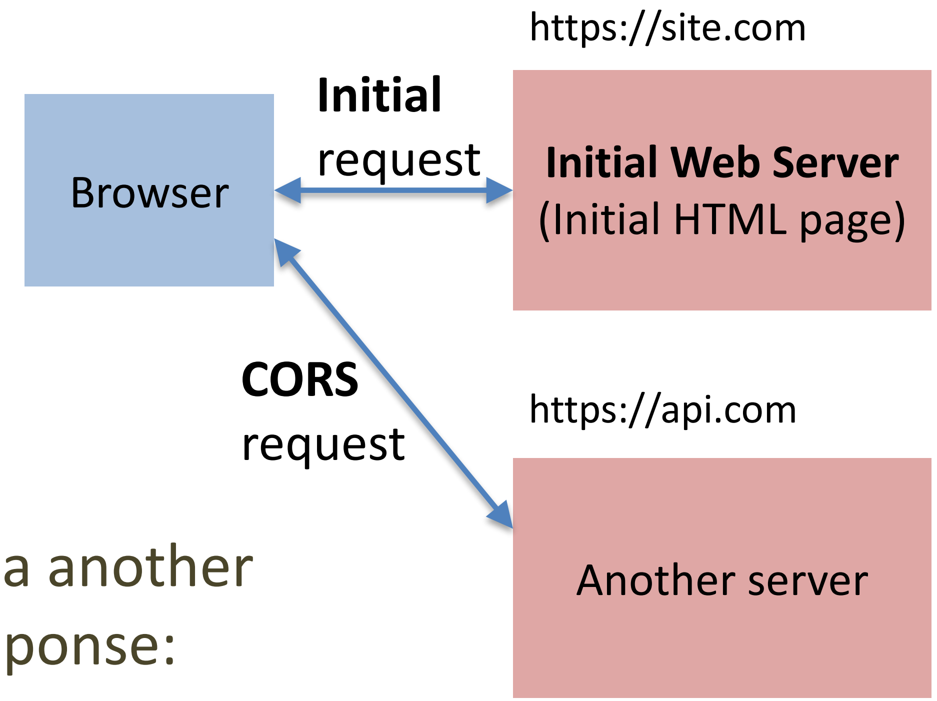
https://fetch.spec.whatwg.org/#http-cors-protocol

CORS in Practice

- The browser knows that the request is addressed to a different origin, i.e., it is a CORS request
- The browser sends the request including the origin by means of a specific HTTP header:

  `Origin:` `https://site.com`
- The *Response* from the receiving server includes, via another header, which origins can access the data in the response:

  `Access-Control-Allow-Origin:` `https://site.com`
- If the two match, the **browser** allows the DOM and JS to access the response
  – Otherwise, the content cannot be accessed, it will appear as "failed to load"



No CORS outside browsers

- CORS DOES NOT APPLY when making requests outside browsers
  - curl, wget, REST Client, etc., do not care about the extra headers
  - origin = null for the server receiving the request if the `Origin:` header is missing
- The `Origin:` header can be set to any value by external programs
  - It CANNOT be used as a security mechanism
  - The web server must rely on proper security mechanisms to authorize data access (cookies etc.)

CORS Preflight Requests

- The `Access-Control-Allow-Origin` value is not known before the browser sends an actual request
- What if a browser sends private information to servers that do not allow its `Origin`?
- Before sending requests with *private/sensitive information*, the browser always checks if such a request is safe to send by means of the HTTP “OPTION” method
  - Example: when doing POST, PUT, requests with special (e.g. cookie) headers, …
- Such cross-site requests are said "*preflighted*"
- This is **performed automatically** by the browser for every request
  - Need to know because it might impact application performance (2 HTTP exchanges vs 1)

https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

CORS Preflight example

```
// fetch https://api.com/the/resource/you/request with method POST

OPTIONS /the/resource/you/request
Access-Control-Request-Method: POST
Access-Control-Request-Headers: origin, x-requested-with, accept
Origin: https://an-origin.com
```

```
// Response from https://api.com/the/resource/you/request

HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://an-origin.com
Access-Control-Allow-Methods: POST, GET, OPTIONS, DELETE
```

https://flaviocopes.com/express-cors/

CORS with Authentication

- By default, fetch requests do not send credentials (e.g., cookies)
- If needed, `fetch` has an option in the `init` object to include them
- Values: `'omit'` (default), `'same origin'` (send only in requests to the same origin), `'include'`

```
fetch('https://example.com', {
  …
  credentials: 'include'
  …
});
```

CORS with <script> tags

- Scripts loaded via `<script>` tag from <u>other origins</u> run with the same privileges of the other scripts in the web application
- Only load scripts you trust, and always include the integrity attribute to prevent malicious code injection
  - Note: to make the browser check the integrity attribute, the `crossorigin` attribute MUST be set (to “anonymous”), otherwise the integrity will not be checked
  - Note that in this case the server must send

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
```

```
<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js"
integrity="sha384-YvpcrYf0tY3lHB60NNkmXc5s9fDVZLESaAA55NDzOxhy9GkcIdslK1eN7N6jIeHz"
crossorigin="anonymous"></script>
```

https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity

Loading modules via script tag

- Modules loaded via `<script>` are subject to CORS rules
- Unfortunately, modules loaded from the file system have **origin `null`** so browsers prevent module loading **even from local file system** (file:// URI)
- Solution: serve content from a (local) web server as static files

```
<body>
 ...
 <script type="module" src="main.js"></script>
 <script type="module" src="index.js"></script>
</body>
```

```
// In index.js:
import * as jsdom from 'main.js';  // import requires the .js be loaded as a module
```

https://github.com/expressjs/cors

https://flaviocopes.com/express-cors/

Controlling Allowed Origins in your Server

CORS ON THE SERVER SIDE

Enabling CORS in Express application

- Use the middleware `cors`
  - http://expressjs.com/en/resources/middleware/cors.html
  - `npm install cors`

```
const express = require('express');
const cors = require('cors');
const app = express();

app.use(cors());  // Careful: enables all origins
```

Simple Usage

- Enable **CORS for all** requests (for the app server)

  ```
  app.use(cors())
  ```

  By default, **all origins** will be enabled for all HTTP methods

  Also, enabling it as application-level middleware (i.e., `app.use(…)`) automatically handles preflight requests for all routes

- Enable CORS for a **Single Route**

  ```
  app.get('/products/:id', cors(), function (req, res, next) {
    …
    res.json({msg: 'This is CORS-enabled for a Single Route'})
  })
  ```

http://expressjs.com/en/resources/middleware/cors.html#enabling-cors-pre-flight

Configuration options

- The `cors(options)` call accepts a JS configuration object
- Always specify the allowed origins (as a string, function, regexp, array)
  - E.g., `"origin": "https://appwebsite.com"`
- May specify the allowed methods
- May fine-tune allowed headers and credentials *(more on this later)*

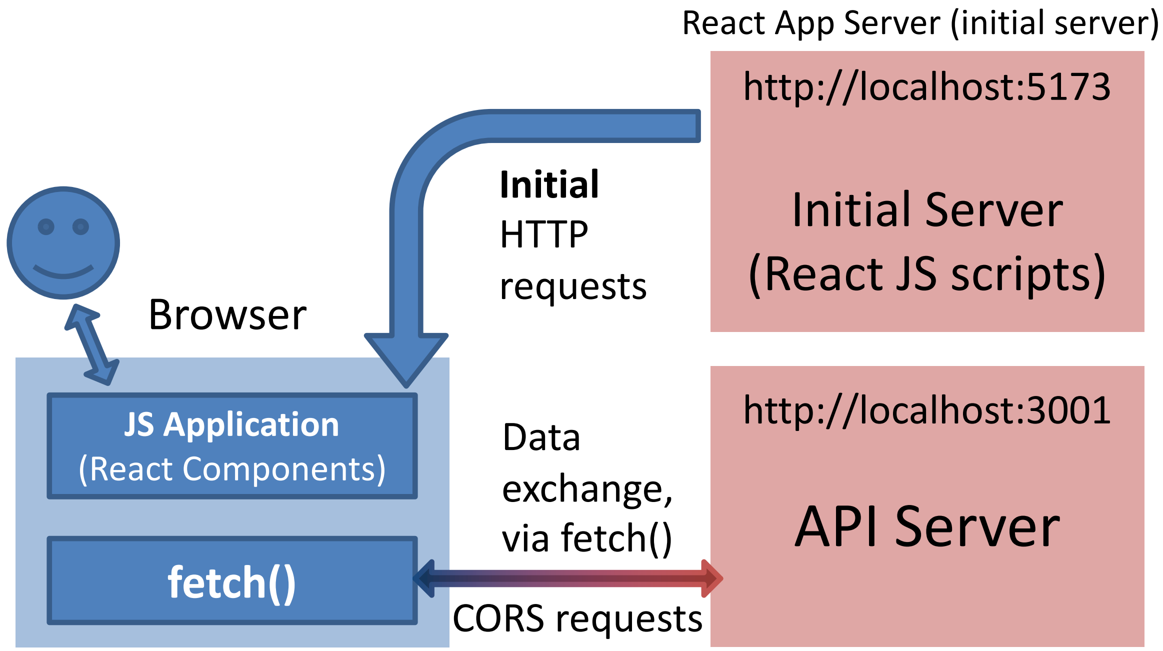
Default configuration options

```
{
  "origin": "*",
  "methods": "GET,HEAD,PUT,PATCH,POST,DELETE",
  "preflightContinue": false,
  "optionsSuccessStatus": 204
}
```

https://expressjs.com/en/resources/middleware/cors.html

CORS with React App

- There is a CORS issue: 2 servers (React dev server and API server)
- Remember that:
  - The server must be configured to accept CORS requests
  - The client will also need to use “credentials: include” if fetching from authenticated endpoints *(reminded later in the course)*



React App Server (initial server)
http://localhost:5173
Initial Server (React JS scripts)
Initial HTTP requests
Browser
JS Application (React Components)
fetch()
Data exchange, via fetch()
CORS requests
http://localhost:3001
API Server

```
const express = require('express');
const cors = require('cors');
const app = express();
const corsOptions = {
  origin: 'http://localhost:5173',
  credentials: true,
};
app.use(cors(corsOptions));
```

https://expressjs.com/en/resources/middleware/cors.html

References

- A tutorial on CORS
  - https://auth0.com/blog/cors-tutorial-a-guide-to-cross-origin-resource-sharing/
- https://portswigger.net/web-security/cors
- https://github.com/expressjs/cors
- https://flaviocopes.com/express-cors/
- https://owasp.org/www-community/attacks/CORS_OriginHeaderScrutiny
- https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/11-Client-side_Testing/07-Testing_Cross_Origin_Resource_Sharing
- https://en.wikipedia.org/wiki/Cross-origin_resource_sharing

License

- These slides are distributed under a Creative Commons license “**Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)**”
- **You are free to:**
  - **Share** — copy and redistribute the material in any medium or format
  - **Adapt** — remix, transform, and build upon the material
  - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
  - **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - **NonCommercial** — You may not use the material for commercial purposes.
  - **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
- https://creativecommons.org/licenses/by-nc-sa/4.0/