JavaScript

Client and server communication

Bjarte Wang-Kileng

HVL

September 1, 2025

Western Norway University of Applied Sciences

Outline

1. Client and server communication
2. Ajax
3. Websockets
   - Not subject for the exam
4. Cross-origin resource sharing
   - Not subject for the exam
5. URLs to local resources

Client and server communication

- XMLHttpRequest
  - Client can fetch server data as e.g. objects, text, XML and HTML.
  - Both synchronous and asynchronous.
- Fetch API
  - Can fetch server data as text and JavaScript objects.
  - Similar to asynchronous mode of **XMLHttpRequest**.
  - Modern approach using Promises.
  - Only asynchronous fetching of data.
- Websockets
  - Bi-directional, full-duplex, communication between client and server.
- Server-Sent Events
  - Unidirectional messages from server to client.

Synchronous fetching of data

- Synchronous request, client will stop and wait for response.

```
let request = new XMLHttpRequest();
request.open("GET", URL, false);
request.send(null); // Stop and wait for response
if (request.status === 200) {
    console.log(`Got response ${request.responseText}`);
}
```

- Modern browsers only allow synchronous requests from WebWorkers.
- HTML documents can only be requested in asynchronous mode.
- For more info, see e.g.: Synchronous and asynchronous requests.

Asynchronous fetching of data

- The "A" of Ajax was *Asynchronous*
- Browser sends the request, but does not wait for the response. The browser immediately goes on to do other stuff.
- Browser does not block while data is fetched.
- A callback function can be run when data is received.

Outline

1. Client and server communication
2. Ajax
3. Websockets
   - Not subject for the exam
4. Cross-origin resource sharing
   - Not subject for the exam
5. URLs to local resources

Ajax

- Ajax was AJAX, from *Asynchronous JavaScript and XML*
- JavaScript in browser sends and fetches data from the web server.
- Data is transferred as a document from the web server.
- Usually formatted as JSON, but also as FORM data, XML, HTML, pure text and binary data.

XML, JSON or HTML

- JSON is good if the data should be processed by JavaScript.
- HTML is good if data is HTML to be displayed by browser.
- XML is good if data can be processed with XSLT or other DOM functionality.

Technologies for Ajax

- **Fetch** API.
  - We will only use Fetch for Ajax, with *async* and *await*.
  - Supported document types are text, JSON, FORM data and binary data.
- **XMLHttpRequest**.
  - More low-level API for Ajax.
  - Can receive XML- and HTML documents and all **Fetch** API types.

Fetch or XMLHttpRequest

- **XMLHttpRequest** has support for more document types.
- Text from **Fetch** can be converted to XML or HTML by JavaScript.
- **XMLHttpRequest** can be easier if XML or HTML documents.
  - Easy though to convert text received with **Fetch** to HTML or XML.

The Fetch API

- Returns a **Promise** that resolves when the HTTP headers are received from web server.
- Example of use:

```
async function requestData() {
    try {
        const response = await fetch("demo1.txt");
        console.log(`Got response from server: ${response}`);
    } catch(e) {
        console.log(`Got error ${e.message}.`);
    }
}
```

Fetch text document with GET

- Default HTTP method for fetch is GET.
- **Response** has methods to retrieve data.
  - Returns a **Promise**.
- Fetch a document as text:

```
async function requestData() {
    try {
        const response = await fetch("demo1.txt");
        const text = await response.text();
        console.log(`Response from server is '${text.trim()}'`);;
    } catch(e) {
        console.log(`Got error ${e.message}.`);
    }
}
```

POST and PUT data with Ajax

- JSON is a much used format, both for requests and responses.
  - Class FormData useful when sending FORM data as JSON.
- HTML5 FORM submission data types are also used with with Ajax.
  - Functions *encodeURI()* and *encodeURIComponent()* to encode data.
- We will only work with JSON and pure text.

POST and PUT data, and mime type

- Data:

name	Åse Åsesen
address	Øsebukten 32

- Mime type “application/json; charset=utf-8”:

```
{
  "name": "Åse Åsesen",
  "address": "Øsebukten 32"
}
```

- Mime type “application/x-www-form-urlencoded”:

```
name=%C3%85se%20%C3%85sesen&address=%C3%98sebukten%2032
```

Posting and receiving JSON

```
async function requestData() {
    const data = {
        "givenname": "Ola",
        "familyname": "Olsen",
        "address": "Søndre Sotraveien 33"
    };
    const requestSettings = {
        "method": "POST",
        "headers": { "Content-Type": "application/json; charset=utf-8" },
        "body": JSON.stringify(data),
        "cache": "no-cache",
        "redirect": "error"
    };

    try {
        const response = await fetch(URL, requestSettings);
        const object = await response.json();

        console.log(`Server response: '${JSON.stringify(object)}'`);
    } catch(e) {
        console.log(`Got error ${e.message}.`);
    }
}
```

Response

- Property *status* of Response is the server response HTTP status.
- Property *header* is the Headers of the response.
- Property *ok* is a boolan that indicates if response was successful.
- Property *url* is the URL seen by the response for the request.

Response status

- Property *status* of **Response** is the server response HTTP status of the request.
- Value 200 means success.
- Value 404 tells that the requested document does not exist.
- Check the HTTP standard for other values.

Working with the response

```
async function requestDocument(url) {
    try {
        const response = await fetch(url)

        if (response.ok) {
            console.log(`Got headers for '${response.url}'`)
        } else {
            console.log(`Could not get '${response.url}'`)
        }

        console.log(`Status code: ${response.status}`)

        console.log("The headers of the response:")
        for (let pair of response.headers) {
            console.log(`* '${pair[0]}': ${pair[1]}'`)
        }
    } catch(e) {
        console.log(`Got error ${e.message}.`)
    }
}
```

Repeated Ajax requests

- Can use **Window** methods *setInterval()* and *setTimeout()*.
- Method *setTimeout()* executes a method after a delay.
- Method *setInterval()* repeats a method with a delay.
- *setTimeout()* gives the best approach for repeated Ajax requests.

setTimeout() for repeated Ajax requests

- Only if request was successful, repeat with a delay.
- Avoids parallel fetching of data with slow connections
- Avoids repeating forever request that fails.

Sharing Ajax connection between modules

- Ajax connection details can change.
  - Moved to a new server.
  - Web server can be put behind a gateway server.
- Ajax can be replaced with another solution.
  - Store locally, e.g. IndexedDB or localForage, then Ajax when Internet.
  - Websokets.

Common module for Ajax connections

Let a common module or component manage the Ajax connections.

- Only one place to maintain connection details.

Precautions with Ajax

- Client can have turned off JavaScript.
  - Inform about alternative sources to the information.
- Inform client that data is fetched from web server with Ajax.
  - Can e.g. change color on icons.
- If larger data transmissions, inform about transmission progress.
- If transmission errors, client must be informed.
- To abort a slow data transfer, use an AbortController.

Outline

1. Client and server communication
2. Ajax
3. Websockets
   - Not subject for the exam
4. Cross-origin resource sharing
   - Not subject for the exam
5. URLs to local resources

Websockets

Read-only, not subject for the exam

- Bi-directional, full-duplex, communication between client and server.
  - The server can push data to the client at any time.
  - With Ajax, all messages from server is a response to a client request.
- Low latency, real-time client/server communication.
- Client can keep connection open for new messages.
  - Reduced overhead of each message.
  - With Ajax, each message must initiate a new HTTP server request.

Websocket client

Read-only, not subject for the exam

```
// Open a websocket
const URL = document.location.host;
const webSocket = new WebSocket(`ws://${URL}/Websocket/demo`);

// Callback too run when websocket is opened
webSocket.addEventListener("open",openCallback);

// Callback to run when data is received from the server
webSocket.addEventListener("message",messageCallback);

// Callback to run when socket is closed
webSocket.addEventListener("close",closeCallback);

function openCallback () { ... }

function messageCallback(event) { ... }

function closeCallback(event) { ... }
```

Spring Boot WebSocketConfigurer

Read-only, not subject for the exam

```
@Configuration
@EnableWebSocket
public class WebsocketConfig implements WebSocketConfigurer {
    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(demoHandler(), "/socket");
    }

    @Bean
    public WebsocketDemo demoHandler() {
        return new WebsocketDemo();
    }
}
```

Spring Boot WebsocketHandler

Read-only, not subject for the exam

```
public class WebsocketDemo extends TextWebSocketHandler {
    @Override
    public void afterConnectionEstablished(WebSocketSession session) { ... }

    @Override
    public void handleMessage(WebSocketSession session, WebSocketMessage<?> message) { ... }


    @Override
    public void afterConnectionClosed(WebSocketSession session, CloseStatus status) { .... }
}
```

Outline

1. Client and server communication
2. Ajax
3. Websockets
   - Not subject for the exam
4. Cross-origin resource sharing
   - Not subject for the exam
5. URLs to local resources

Web content origin

Read-only, not subject for the exam

- An *origin* is defined by protocol (e.g. *http*), host and port.
- Examples of same origins:
  - `http://example.com/app1`
  - `http://example.com/app2`
  - `http://example.com:80/app2` (port 80 is default for http)
- Different origins due to different protocols:
  - `http://example.com:8080/app1`
  - `https://example.com:8080/app1`
- Different origins due to different hosts:
  - `http://host.no/app1`
  - `http://example.com/app1`
- Different origins due to different ports:
  - `http://example.com/app1`
  - `http://example.com:8080/app1`

Same-origin policy (SOP)

Read-only, not subject for the exam

- Applies to JavaScript, not HTML tags.
  - Tags like *SCRIPT* and *IMG* can load documents across origins.
  - HTML attribute crossorigin can specify how to act if not same origin.
- JavaScript can access data from URL only if same origin.
- Restricts what network messages one origin can send to another.
  - Ajax and Websockets only possible to origin of HTML document.
- Simple requests allowed across origins, but not response document.
  - Requests that are allowed through HTML tags are still possible.
  - Browser can store, but not read data across origins.

Why Same-origin policy?

Read-only, not subject for the exam

- Protection against Cross Site Request Forgery (CSRF) and other Cross-Domain attacks.
- Assume a user that has logged in to site `https://mybank.no`.
- User then visits `https://attacker.no`.
- If not SOP, `attacker.no` has access to all open browser sessions.
  - `attacker.no` can access `mybank.no` using privileges of user.
- SOP does not prevent CSRF through HTML tags or simple requests.

Cross-origin resource sharing (CORS)

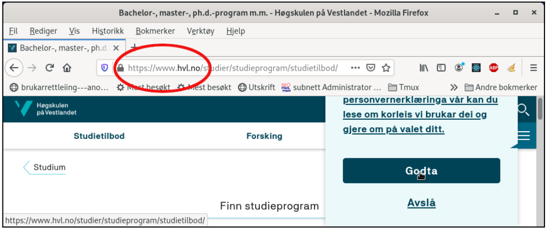
Read-only, not subject for the exam

- CORS possible through specific HTTP headers.
- CORS starts with a preflight between browser and web server.
  - Browser sends a HTTP **OPTIONS** containing an *Origin* header.
  - Web server must respond with an *Access-Control-Allow-Origin* header.
- Only if preflight is successful will browser send the actual request.

CORS headers

Read-only, not subject for the exam

- *Origin* specifies the domain of the web document of browser.
  - The web server shown in URL field of browser.


Bachelor-, master-, ph.d.-program m.m. - Høgskulen på Vestlandet - Mozilla Firefox
Fil Rediger Vis Historikk Bokmerker Verktøy Hjelp
Bachelor-, master-, ph.d.
https://www.hvl.no/studier/studieprogram/studietilbod/
brukarrettleiing---ano… Mest besøkt Mest besøkt Utskrift subnett Administrator … Tmux Andre bokmerker
Høgskulen på Vestlandet
Studietilbod
Forsking
Studium
Finn studieprogram
Godta
Avslå
https://www.hvl.no/studier/studieprogram/studietilbod/

personvernerklæringa vår kan du lese om korleis vi brukar dei og gjere om på valet ditt.

- *Access-Control-Allow-Origin* is the origin that can access the document from JavaScript.
  - Only a single value is allowed.
  - Server can respond with the received origin to signal access.
  - Value “*” specifies all origins.

CORS headers and security

Read-only, not subject for the exam

- CORS-header mismatch tells browser not to allow JavaScript.
    - Nothing but the browser itself prevents JavaScript to send the request.
- CORS-header mismatch tells browser not to continue after preflight.
    - Nothing but the browser itself prevents the request.
- *Origin* is set by client.
    - Browser does not allow modifying value, but other client tools can.
    - Web server do not care, but can use value to produce a corresponding *Access-Control-Allow-Origin*.
- Web server should restrict use of *Access-Control-Allow-Origin* for protected data.

SOP and CORS headers, and security

Securing a site require other mechanism than SOP or CORS headers, e.g. authorization of requests.

URLs and application resources

- URL paths can change on deployment, or if using a gateway server.
- Assume link in application:

```
<a href="/path/to/application/a.html">
```

- Deployment can change path to local resources.
  - In Eclipse: “https://localhost:8080/path/to/application/a.html”
  - Deployed: “http://server:80/some/prefix/path/to/application/a.html”
- User clicks link;
  - Browser to server: “https://server:80/path/to/application/a.html”
  - Missing “/some/prefix”, and server responds with HTTP status 404.

URLs to local resources

Only use relative paths to local resources.

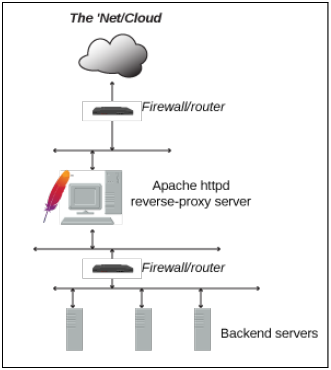
Absolute paths can cause application to fail when deployed to webserver.

Relative URLs

- Never use absolute paths for local application resources.
  - Application should still work if you prepend local URLs with "./".
- Never start URL to local application resource with:
  - Never use "http://".
  - Never use "https://".
  - Never start path with "/".
- Always use relative paths, e.g.
  - "relative/path/to/resource"
  - "./relative/path/to/resource"
  - "../relative/path/to/resource"
- Script path properties will work, e.g. *import.meta.url* and *document.currentScript.src*.
  - These are set by browser and should have the correct values.

Gateway server

- Also named reverse proxy server.
- Redirect requests to other web servers.
- Common to use Apache as gateway to JavaEE servers.
  - Apache handles client connections and SSL certificates.
  - Port 80 or 443 between client and Apache.
  - Port 8080 between Apache and TomEE.
- Allow web servers behind firewalls.



Redirection by reverse proxy

- Can redirect based on IP-number, host name and URL path.
  - If IP-number, reverse proxy must have multiple network cards.
  - If host name, multiple host names must point to IP of reverse proxy.
- Example with redirect on URL path:
  - Client to gateway: “https://gateway.server/servers/A/application”
  - Gateway to webserver: “http://serverA:8080/application”
- Absolute paths can differ!
  - Browser see “/servers/A/application”.
  - Application uses “/application”.
- Application will fail on absolute paths to local resources!