<WA1/>

2020

Fetch API

Enabling the link to the Server Side

Enrico Masala
Fulvio Corno
Luigi De Russis

Goal

- Sending asynchronous HTTP requests
- Loading data asynchronously
- Handling multiple requests
- Interrupting requests
- Using alternative libraries

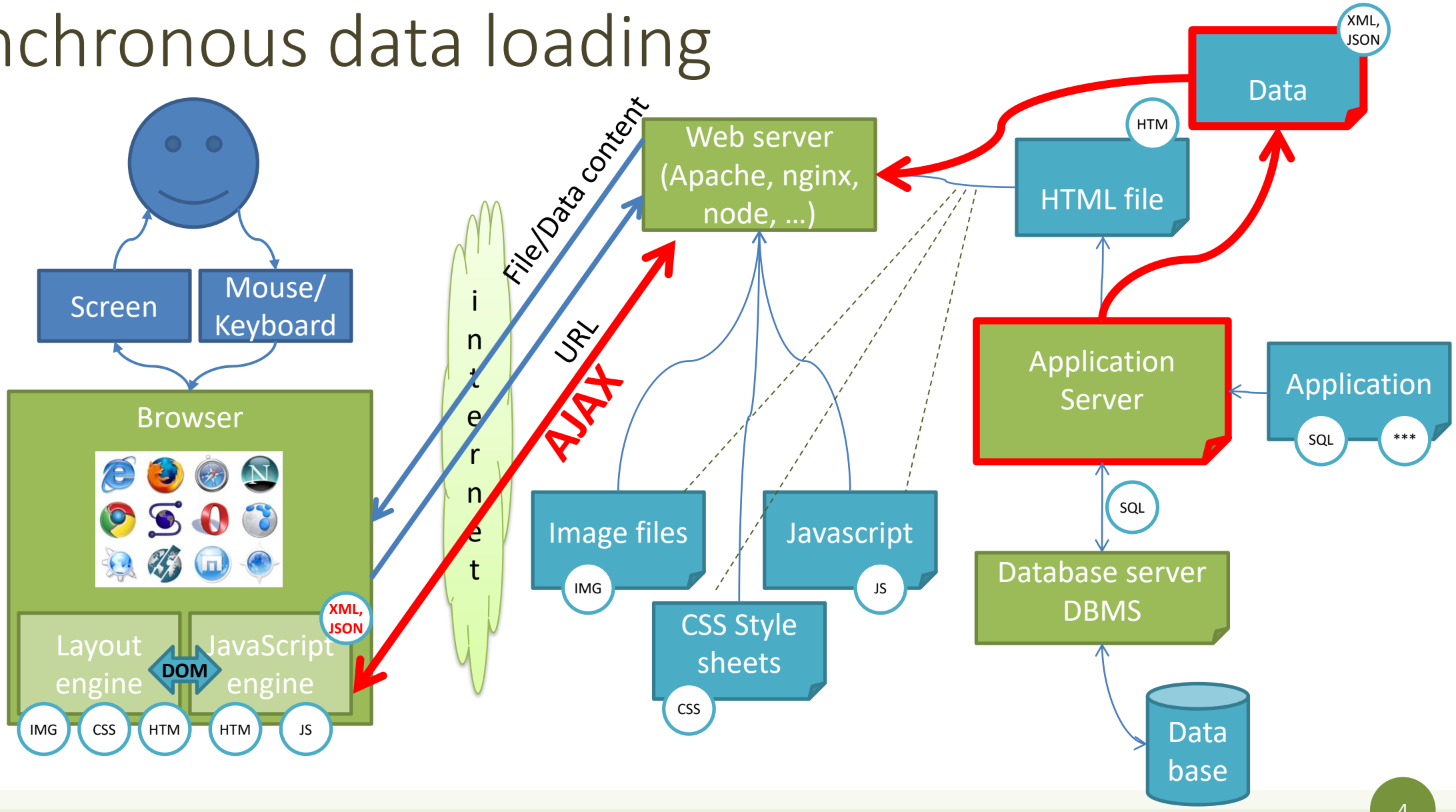
JavaScript: The Definitive Guide, 7th Edition
Chapter 11. Asynchronous JavaScript

Mozilla Developer Network:
Web technology for developers —
Web API — Fetch API
https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

Fetch API

ASYNCHRONOUS JS REQUESTS

Asynchronous data loading



Typical web applications

- Server provides the first HTML page, with JS and other resources
- User interacts with the application, generating events
  - Input events, mouse events, form events, …
- The JavaScript code decides new data is needed
  - Option #1: loads a new URL
    - **Not recommended** since 20+ years…
  - Option #2: requests new data from the server, receives and interprets the data, and modifies page content via DOM
    - Has to be done **asynchronously** not to **block the user interface**!

Loading data asynchronously

- Make asynchronous HTTP requests using browser-provided Web API
- Possible since IE5 (1998) via the `XMLHttpRequest` (XHR) object
- Popularized a few years later by Google and others
  - Dynamic suggestions while typing in search box, without reloading the page

- NB: For security reasons, loading data is, by default, possible only from the same server
  - Possible to allow loading from other servers via **CORS** (Cross Origin Resource Sharing)

XMLHttpRequest (XHR)

- Standardized by W3C in 2006 but already available in many browsers
- Quite complex to use for the developer
  - Requires managing the XHR object states, callbacks, etc
  - Inconsistencies between browsers
  - Some libraries (notably JQuery) provided some easier interface
- Still supported, but *not recommended*

Fetch API

- Modern way of asynchronous data loading in JS
- Uses Promises instead of callbacks
- Provides a generic definition of Request and Response objects, as well as other support for network requests (Headers)
- Well supported in the browser context: included in the HTML5 living standard
  - Supported by all browsers since 2016/2017 (except IE)

How to use fetch

- Use the **`fetch()`** method
  - Parameter: URL of the resource
- Available in almost any context (e.g., from `window` object)
- Returns a **`Promise`** that will resolve once the load operation finishes
  - Resolves to the **`Response`** object, that allows to access the details of the HTTP transaction and the content
  - The promise is rejected only in case of network errors
- Some small differences with `JQuery.ajax()`
  - E.g., cookie handling

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

Example

- Just handle the promise (`.then` or `await`)

```
fetch('http://example.com/tasks.json')
  .then((response) => {
    return response.json();
  })
  .then((data) => {
    console.log(data);
  })
```

```
async function loadData() {
  let response = await
     fetch('http://example.com/tasks.json');
  let data = await response.json();
  console.log(data);
}

loadData();
```

`Response` object

- The fulfilled promise returns a `Response` object
- Main properties
  - `Response.ok` (boolean): HTTP successful (code 200-299)
  - `Response.status`, `Response.statusText`
  - `Response.type` : basic or cors (CORS explained later in the course)
  - `Response.url` : final URL (potentially after HTTP redirects)
  - `Response.body`: a Readable stream of the body content

https://developer.mozilla.org/en-US/docs/Web/API/Response

Accessing response headers

```
fetch('http://localhost/data.json')
  .then(response => {
    console.log(response.headers.get('Content-Type'));
    console.log(response.headers.get('Date'));

    console.log(response.status);
    console.log(response.statusText);
    console.log(response.type);
    console.log(response.url);
  }
```

```
application/html; charset=utf-8
Sat, 11 Apr 2020 13:41:04 GMT

404
Not Found
undefined
http://localhost/data.json
```

https://developers.google.com/web/updates/2015/03/introduction-to-fetch

Error handling

- Promise is only rejected for non-HTTP errors (e.g., network connection error)
  - Any HTTP status value (200 OK, 404: Not found, 500: Internal server error, ...) returns a **fulfilled** Promise
- Suggested error handling approach:
  - Check `response.ok` : boolean value (true for HTTP status 200-299)
  - Check content type header (depends on the application needs)
  - Provide a `catch()` for other types of errors

Example: error handling

```
fetch(url)
  .then(response => {
    if (!response.ok) { throw Error(response.statusText) }
    let type = response.headers.get('content-type');
    if (type !== 'application/json') {
        //then() returns a rejected promise if something is thrown
        throw new TypeError(`Expected JSON, got ${type}`)
    }
    return response;
  })
  .then(response => {
  //...
  })
  .catch(err => console.log(err))  // either the throw value or other errors
```

Fetch options

- `const fetchResponsePromise = fetch(resource [, init])`
- Main properties of `init` object
  - `method`
  - `headers` (an object with a property per each header)
  - `body`
  - `mode` (cors, no-cors, same-origin)
  - `credentials` (omit, same-origin, include), to send cookies with the request
  - `signal`: an AbortSignal object instance to communicate with the fetch request

https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/fetch

Example: POST method

```
fetch(url, {
    method: 'post',
    headers: {
      "Content-type": "application/x-www-form-urlencoded; charset=UTF-8"
    },
    body: 'foo=bar&lorem=ipsum'
  })
  .then(responseData => console.log(responseData))
  .catch(function (error) {
    console.log('Request failed', error);
  });
```

Example: sending JSON content

```
let objectToSend = {'title': 'Do homework' , 'urgent': true, 'private': false,
'sharedWithIds': [3, 24, 58] };

fetch(url, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(objectToSend),  // Conversion in JSON format
  })
  .catch(function (error) {
    console.log('Failed to store data on server: ', error);
  });
```

Example: asynchronous file upload

```
<input type="file" id="fileUpload" />
```

```
const handleImageUpload = event => {
  const files = event.target.files  // all files selected by the user
  const formData = new FormData()   // need to correctly encode body
  formData.append('myFileName', files[0])

  fetch('/saveImage', {
    method: 'POST',
    body: formData
  })
  .then(response => response.json())
  .then(data => { console.log(data.path) })
  .catch(error => { console.error(error) })
}
document.querySelector('#fileUpload').addEventListener('change', event => {
  handleImageUpload(event)
})
```

https://flaviocopes.com/file-upload-using-ajax/

Reading the Response body

- Can use (**only once**) one of the following methods
  - …then body is "**consumed**"
- These methods also return a Promise, that returns the response body…
  - `response.text()` : as plain text (string)
  - `response.json()` : as a JS object, by parsing the body as JSON
  - `response.formData()` : as a FormData object
  - `response.blob()` : as Blob (binary data with type)
  - `response.arrayBuffer()` : as ArrayBuffer (low-level representation of binary data)
- `response.body` is a ReadableStreaming object to read it chunk-by-chunk

https://javascript.info/fetch

Sequential fetches

- Easy with `async`: no need to nest another fetch in `.then()` method

```
const getFirstUserData = async () => {
  const response = await fetch('/users.json'); // get users list
  const users = await response.json(); // parse JSON
  const user = users[0]; // pick first user
  const userResponse = await fetch(`/users/${user.name}`); // get user data
  const userData = await user.json(); // parse JSON
  return userData;
}

getFirstUserData()
```

Parallel fetches

- Multiple fetches in parallel: use `Promise.all()`

```
// array of URLs
const urls = [url1, url2];

// Convert to an array of Promises
const promises = urls.map(url => fetch(url).then(r => r.text()));  // Return promises
// .then(...): Wait on the Promise that is settled when the whole body is arrived

// Run all promises in parallel, wait for all
Promise.all(promises)
  .then(bodies => { for (const body of bodies) console.log(body); })
  .catch(e => console.error(e))
```

Parallel fetches

- Processing content as soon as all fetches receives the start of a (potentially long) response

```
// array of URLs
const urls = [url1, url2];

// Convert to an array of Promises
const promises = urls.map(url => fetch(url) );
// Wait only for the fetch Promise

// Run all promises in parallel, wait for all
Promise.all(promises)
  .then(results => { // process according to the order needed by the app
       for (const res of results) res.text().then( t => console.log(t) );
  })
  .catch(e => console.error(e))
```

Interrupt/cancel a request

- Reasonably well supported in browser: pass `signal` in fetch options

```
const controller = new AbortController();
const cancelButton = document.querySelector('#cancel');

cancelButton.addEventListener('click', function() {
  controller.abort();  // Download canceled
});

function fetchVideo() {
  //...
  fetch(url, {signal: controller.signal}).then(response => {
    //...
  }).catch(err => console.log(err.message); )
}
```

https://developer.mozilla.org/en-US/docs/Web/API/AbortController

Basic fetch vs other libraries

- Most common alternative library: **Axios**
  - Does polyfill for older browsers
  - Has an easier way to cancel a request
  - Has a way to set a response **timeout** (not supported by fetch, which needs a `setTimeout()` to call the `AbortController.abort()` method)
  - Easier support for progress bar via Axios Progress Bar module (fetch requires quite some code around a `ReadableStream` object)
  - Performs automatic JSON conversion
  - Provides an easier way to separate responses of parallel requests
  - Works well also in Node.js (fetch is not included by default)

https://flaviocopes.com/axios/

Axios example

```
axios({
  method: 'post',
  url: '/login',
  timeout: 4000,    // 4 seconds timeout
  data: {    // Directly an object, automatically converted into bytes
    firstName: 'David',
    lastName: 'Pollock'
  }
})
.then(response => {/* handle the response */})
.catch(error => console.error('timeout exceeded'))
```

https://blog.logrocket.com/axios-or-fetch-api/

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/