COESIONE ITALIA 21-27 PIEMONTE

Cofinanziato dall'Unione europea

REGIONE PIEMONTE

FinTech Software Developer

Programmazione WEB - HTML | CSS | Javascript

Docente: Shadi Lahham

Intervento realizzato da ITS TECNOLOGIE DELL'INFORMAZIONE E DELLA COMUNICAZIONE

JSON and AJAX

Dynamic Content

Shadi Lahham - Web development

JSON

Javascript Object Notation

- Lightweight data-interchange format
  - Douglas Crockford in 2001 to replace XML
  - Easy to read and write for humans,
  - Easy to parse and generate for machines
- Has two structures
  - Collection of name:value pairs. In different languages, realized as an object, record, struct, dictionary, hash table, etc.
  - Ordered list of values. In most languages, realized as an array, vector, list, or sequence
- JSON vs Javascript objects
  - Keys must be stored with quotes
  - Values can be number, string, boolean, array, object, null
- Validate your JSON
  - Use JSONLint or jsonlint CLI.

Storing data in JSON

```
{
 "firstName": "Jane",
 "lastName": "Smith",
 "address": {
   "streetAddress": "425 2nd Street",
   "city": "San Francisco",
   "state": "CA",
   "postalCode": 94107
 },
 "phoneNumbers": [
   "212 732-1234",
   "646 123-4567" ]
}
```

JSON vs a Javascript object

```
{
 "firstName": "Jane",
 "lastName": "Smith",
 "address": {
   "streetAddress": "425 2nd Street",
   "city": "San Francisco",
   "state": "CA",
   "postalCode": 94107
 },
 "phoneNumbers": [
   "212 732-1234",
   "646 123-4567" ]
}
```

```
let person = {
  firstName: 'Jane',
  lastName: 'Smith',
  address: {
    streetAddress: '425 2nd Street',
    city: 'San Francisco',
    state: 'CA',
    postalCode: 94107
  },
  phoneNumbers: [ '212 732-1234', '646
123-4567' ]
};
```

Common JSON mistakes

```
{
  name: "John",                     // mistake: property name without quotes
  "surname": 'Smith',               // mistake: single quotes in value (must be double)
  'isAdmin': false,                 // mistake: single quotes in key (must be double)
  "birthday": new Date(2000, 2, 3), // mistake: no "new" is allowed, only bare values
  "friends": [0,1,2,3],             // here all fine
  "age": 25                         // mistake: missing comma
  "profession": "tester",           // mistake: trailing comma
}
```

Note

Comments are not allowed in a JSON file

The above example is just to illustrate possible errors in a JSON file

JSON array

```
[
 {
   "name": "John Smith",
   "email": "john.smith@example.com"
 },
 {
   "name": "David Jones",
   "email": "david.jones@example.com"
 }
]
```

Writing a JSON to the DOM

```
// assuming we get a profile from an external source
//
// myProfile
//
// {
//   "firstName": "Liz",
//   "lastName": "Howard",
//   "cats": [ "Tribbles", "Jean Claws" ]
// }

let p = document.createElement('p');
p.innerHTML = 'My name is ' + myProfile.firstName + ' ' + myProfile.lastName + '.';
p.innerHTML += 'My cats are ' + myProfile.cats.join(', ') + '.';
```

JSON methods

JavaScript provides these two methods:

- JSON.stringify to convert objects into JSON
- JSON.parse to convert JSON back into an object

JSON.stringify();

```
let student = {
  name: 'John',
  age: 30,
  isAdmin: false,
  courses: ['html', 'css', 'js'],
  wife: null
};

let json = JSON.stringify(student);
console.log(typeof json); // a string!
console.log(json);
```

Resultant JSON-encoded object

```
{
  "name": "John",
  "age": 30,
  "isAdmin": false,
  "courses": ["html", "css", "js"],
  "wife": null
}
```

JSON.parse();

```
let user = '{ "name": "John", "age": 35, "isAdmin": false, "friends": [0,1,2,3] }';

user = JSON.parse(user);

console.log(user.friends[1]); // 1
```

Note
The parse() method can potentially throw errors that need to be handled

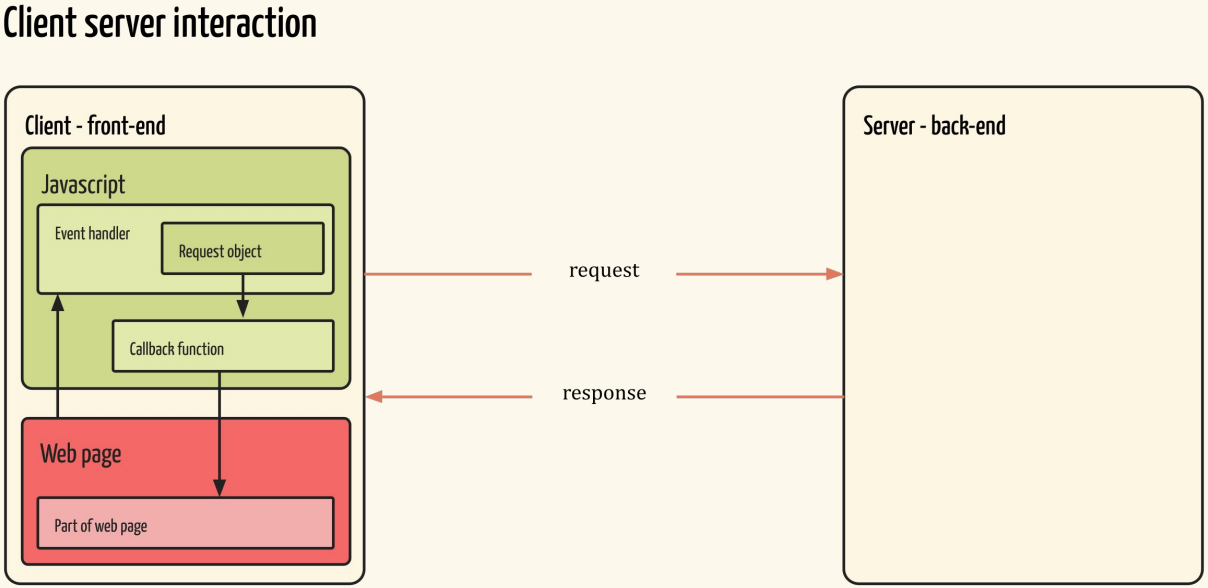
AJAX

Asynchronous JavaScript and 'XML'

Request & response

1. An event occurs in a web page, e.g. the page is loaded or a button is clicked
2. An XMLHttpRequest object is created by JavaScript
3. The XMLHttpRequest object sends a request to a web server
4. The server processes the request
5. The server sends a response back to the web page
6. The response is read by JavaScript
7. Proper action, such as a page update, is performed by JavaScript

Request & response



An XMLHttpRequest

```
// instantiate a new request
const request = new XMLHttpRequest();
const endpoint = 'https://fakeapi.example.com/data';

// add event listeners
request.addEventListener('load', function () {
  // transform a string into a usable object
  console.log(JSON.parse(request.responseText));
});

// prepare the request
request.open('GET', endpoint, true); // third parameter makes an asynchronous request (default)
request.setRequestHeader('Content-type', 'application/json'); // not needed for GET requests

// send the request
request.send();
```

HTTP verbs & CRUD operations

HTTP Verb	Action	CRUD
GET	Requests data from a specified resource	Read
POST	Submits data to be processed to create a new resource	Create
PUT	Uploads data to update an entire resource	Update
DELETE	Deletes the specified resource	Delete

Other HTTP verbs

HEAD, TRACE, OPTIONS, CONNECT, PATCH

HTTP verbs & CRUD operations

HTTP verbs

- used to specify the type of action to be performed on a resource when making requests over HTTP - Hypertext Transfer Protocol
- integral to the principles of REST - Representational State Transfer

CRUD operations

- correspond to basic actions that can be performed on data
    - Create, Read, Update, Delete

Request events

loadstart
fires when the process of loading data has begun. This event always fires first

progress
fires multiple times as data is being loaded, giving access to intermediate data

error
fires when loading has failed

abort
fires when data loading has been canceled by calling abort()

load
fires only when all data has been successfully read

Request events

loadend
fires when the object has finished transferring data
always fires and will always fire after error, abort, or load

timeout
fires when progression is terminated due to preset time expiring

readystatechange
fires when the readyState attribute of a document has changed

A request using onLoad

```
// instantiate a new request
const request = new XMLHttpRequest();
const endpoint = 'https://fake.service.com/username?id=some-unique-id';

// prepare the request
request.open('GET', endpoint);

// shortcut for addEventListener with 'load' event
request.onload = function () {
  if (request.status === 200) {
    console.log("User's name is " + request.responseText);
  } else {
    console.log('Request failed.  Returned status of ' + request.status);
  }
};

// send the request
request.send();
```

A request using readyState

```
const request = new XMLHttpRequest();
const method = 'GET';
const endpoint = 'https://developer.mozilla.org/';

request.open(method, endpoint);

request.onreadystatechange = function () {
  if (request.readyState === XMLHttpRequest.DONE && request.status === 200) {
    console.log(request.responseText);
  }
};

request.send();
```

XMLHttpRequest.readyState values

0	UNSENT	Client has been created. open() not called yet.
1	OPENED	open() has been called.
2	HEADERS_RECEIVED	send() has been called, and headers and status are available.
3	LOADING	Downloading; responseText holds partial data.
4	DONE	The operation is complete.

You can use them as constants e.g. `XMLHttpRequest.DONE`

HTTP Status Messages

To check the status use `XMLHttpRequest.status` **and** `XMLHttpRequest.statusText`

Categories
1xx: Information
2xx: Successful
3xx: Redirection
4xx: Client Error
5xx: Server Error

Most common
200 OK
403 Forbidden
404 Not Found
500 Internal Server Error

Complete list
HTTP Messages , HTTP Status Codes

A PUT request example

```
const request = new XMLHttpRequest();
const endpoint = 'https://fake.service.com/user/1234';
const payload = { name: 'John Smith', age: 34 };

// prepare the request - specifying the content type and encoding
request.open('PUT', endpoint);
request.setRequestHeader('Content-Type', 'application/json;charset=UTF-8');

request.onload = function () {
  if (request.status === 200) {
    let userInfo = JSON.parse(request.responseText);
  }
};

request.send(JSON.stringify(payload));
```

Question

What would we need to change if this were a POST request?

Content type

Media type (formerly MIME type - Multipurpose Internet Mail Extensions)

- a standard way of describing a data type in the body of an HTTP message or email
- The MIME type is passed in the Content-Type header

```
request.setRequestHeader('Content-Type', 'application/json;charset=UTF-8');
request.setRequestHeader('Content-type', 'application/json');
```

text/plain for plain text
text/html for HTML content
application/javascript for JavaScript files
application/xml for XML data
image/jpeg for JPEG images
image/png for PNG images
audio/webm for WebM audio files
video/mp4 for MP4 video files

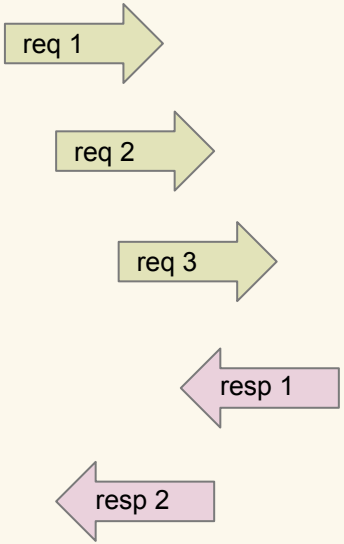
Important MIME types for Web developers

Request & response order

Client

Sends
request 1
request 2
request 3

response 3 never arrives

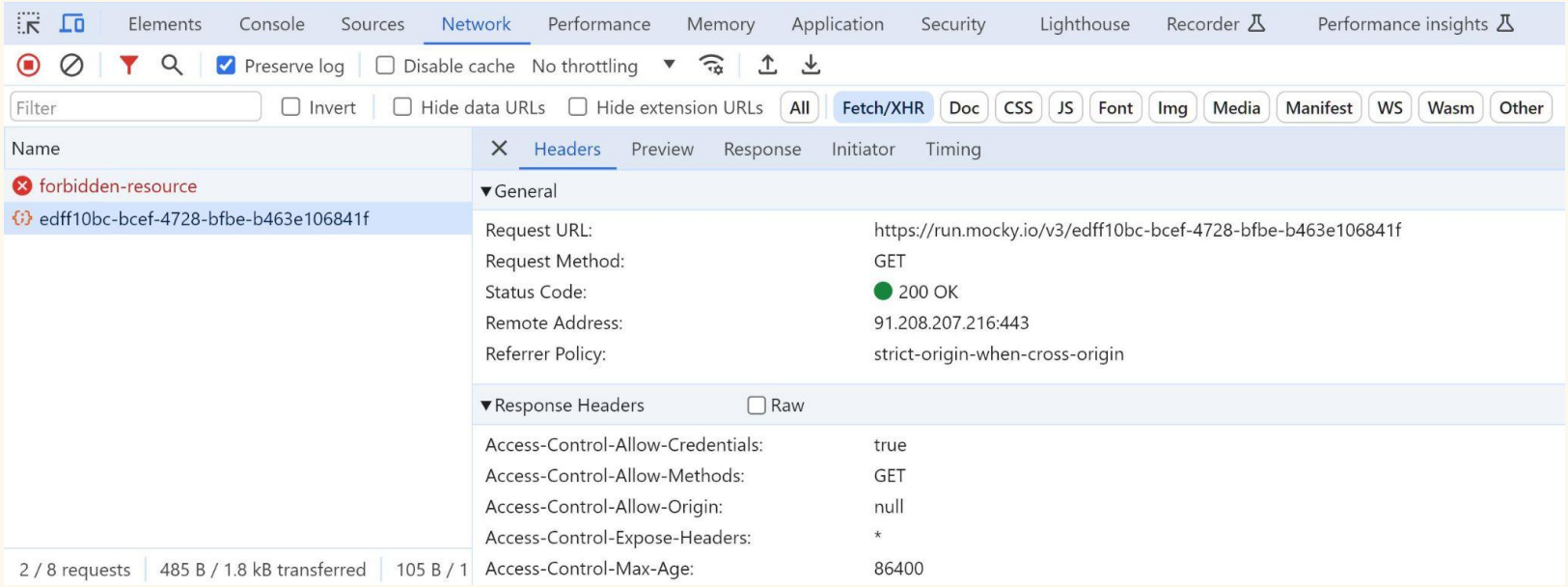


Server

Receives
request 1
request 2
request 3

Sends
response 2
response 1
.
breaks and doesn't send other responses

Debugging requests



DevTools

Use to debug all network activity. Filter by Fetch/XHR requests

Status and error handling

```
const request = new XMLHttpRequest();
request.open('GET', endpoint);
request.onload = function () {
  if (request.status === 200) {
    // do something useful with request
  } else {
    console.error("Request didn't load successfully. Error code:", request.statusText);
  }
};
request.onerror = function () {
  console.error('Network error');
};
request.send();
```

Notes

Remember to use onprogress to show a loader give users a feedback message

Always handle errors since there is no guarantee that HTTP requests will succeed

Fetch API

Fetch API

The Browser APIs offer a way to send HTTP requests from the front-end of a web app, enabling live updates of dynamic content without the need to refresh the page

This functionality enables communication with a web server and allows for responses in JSON, plain text, or XML format

The Fetch API, a contemporary substitute for XHR, was introduced in modern browsers to simplify asynchronous HTTP requests

Fetch API

```
const endpoint = 'https://run.mocky.io/v3/fake';

fetch(endpoint)
  .then(response => {
    if (response.ok) {
      return response.json();
    } else {
      throw new Error('Network response was not ok.');
    }
  })
  .then(data => {
    // do something with data
    console.log(data);
  })
  .catch(error => {
    console.error('Error fetching data:', error);
  });
```

Put using Fetch

```
const endpoint = 'https://fake.pipedream.net';

fetch(endpoint, {
  method: 'PUT',
  body: JSON.stringify({ name: 'John', age: 30 }),
  headers: {
    'Content-Type': 'application/json'
  }
})
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json();
  })
  .then(data => console.log(data))
  .catch(error => console.error('Error making PUT request:', error));
```

Simulating the backend

JSON blob

JSON blob allows you to create JSON objects online and access them them as endpoint via HTTPS requests
https://jsonblob.com/api#endpoints

Important
Read the documentation and experiment with creating a JSON to make sure you understand how it works

Alternatives

Mocky
My JSON Server
JSONBin.io
Beeceptor
Reqres
JSON Server - requires nodejs

CORS

Cross-origin resource sharing

CORS

Same-Origin Policy

Browsers restrict web pages from making requests to different origins as a security measure to prevent malicious attacks

Cross-Origin Resource Sharing

Allows controlled access to resources from different origins

CORS on MDN

CORS on Wikipedia

Same-origin policy

Requests originating from
`https://store.company.com/dir/page.html`

URL	Outcome	Reason
https://store.company.com/**dir2**/other.html	success	
https://store.company.com/**dir/inner**/other.html	success	
http://store.company.com/not-secure.html	failure	Different protocol
https://store.company.com**:81**/dir/other.html	failure	Different port
https://**news**.company.com/dir/other.html	failure	Different host

Preflight

Preflight Requests

Before making an request, the browser makes an OPTIONS request to check if the server allows the actual request

Both XMLHttpRequest and the Fetch API may trigger a preflight request for certain cross-origin requests

Preflight

Requests that trigger a preflight:

- Requests that use methods other than GET, HEAD, or POST
- Requests that include headers other than the simple headers
  - e.g. Content-Type
- Requests that use certain types of content types
  - e.g. application/json with a custom header

HTML crossorigin attribute

Specifies CORS usage for resources that are fetched from a different domain than the HTML page

- e.g. images, scripts, fonts

Prevents CORS errors that arise when fetching resources without proper permissions

- browser defaults to same-origin which might lead to CORS issues
- often used when embedding resources from CDNs

Applies to HTML elements like <audio>, <img>, <link>, <script>, and <video>

```
<img src="image.jpg" alt="image description" crossorigin="anonymous">
<link rel="stylesheet" href="styles.css" crossorigin="anonymous">
<script src="script.js" crossorigin="anonymous"></script>
```

crossorigin | MDN

Your turn

1.Factory

- Write car.json, a JSON that represents a car object
  - Make your object complete, having at least one property of the following types
  - Number, String, Boolean, Array, Object, Null
- Write a factory.json that represents a car factory
  - Follow the same rules above
- Transform car.json into cars.json with 5 cars
- Cars should belong to a factory
  - Write two variants of factory.json
  - One that has cars directly embedded in the factory JSON structure
  - Another that uses cars referring to their IDs

2.DOM Factory

- Write your cars and factory objects as JSON strings in a variable
- Parse them with JSON.parse();
- Write each of them to the DOM in a list
    - You should use a styled CSS list with no bullets
    - Don’t use <table>

Bonus

3.Remote factory

- Use jsonblob to store JSON data about cars and a car factory
- You can use as many blobs as you need. Decide the structure in a way to reduce the amount of data you modify with HTTP requests
- Write an application that displays a factory with a list of cars
- Clicking on each car should display a collapsible panel with more information about the car
- It should be possible to edit the car details
- Save the modified data to jsonblob with an HTTP request
- Whenever data is modified you should reload the new data from jsonblob once the writing has finished

Continues on next page >>>

3.Remote factory

- You should handle all error cases in your application. If an HTTP request fails, you should display a message to the user
- Your project should include a folder called ‘json’ with all the initial json files that you upload to jsonblob (the initial state of your DB)
- Your readme (markdown) should include links to all the jsonblobs that you are using as well as a list of their IDs

4.Parallel factory

- Create another version of the factory that uses the same jsonblobs that you created for the previous exercise
- Make sure that each car information is stored in a different jsonblob
- The page should display the list of cars with detailed information about each car directly visible without a collapsible panel
- Make sure that you request all jsonblobs in parallel (at the same time) not in sequence (one after another)
- Show a loader or a loading message while loading and show the list only when data has returned from all jsonblobs and all requests finished
- Make sure that your code handles all **errors**

References

JSON

JSON.org

JSON Syntax

JSON.stringify()

JSON.parse()

References

XMLHttpRequest
XMLHttpRequest.open()
XMLHttpRequest.send()
Using XMLHttpRequest

References

HTTP Methods GET vs POST

An introduction to HTTP verbs

HTTP request methods