

universität freiburg

Databases and Information Systems WS 25/26

Lecture 10: Web applications, part 2

January 13, 2026

Prof. Dr. Hannah Bast
Department of Computer Science
University of Freiburg

Overview of this lecture

■ Organizational

- Your experiences with ES9 Web applications, part 1
- Exam on 17.03.2026 @ 9:00 We have 172 registrations

■ Contents

- Web applications, part 2:

JavaScript dynamically change content of web page

Vulnerabilities privacy, code injection, cross origin

Unicode ISO-8859-1, UTF-8, URL encoding

ES10: Make your web app dynamic ("search as you type") + secure + handle Unicode properly + add SPARQL results

■ Excerpts from your feedback [somewhat representative]

"Fun to make a webserver and eventually see it in my browser"

"Designing the website was very cool, but also kind of addictive"

"I was surprised how simple it is to build a basic web server"

"Lecture+sheet really nice, I really like the live coding sessions"

"It was quite interesting to try all those things by hand"

"Really interesting and easy to follow; styling with CSS was fun"

"Sheet a bit difficult to understand due to the amount of text"

"Generally a very enjoyable exercise to implement"

"I like web development and already do it a lot in my free-time"

Experiences with ES9

■ What do you want to do better [...] in 2026?

"Consistently do things for university and sports"

"More cool things with friends and of course stay healthy"

"More sport, stop smoking, getting my shit together"

"I wanted to reduce doomscrolling ... let's see"

"Improve my sleep (just sleep longer, like 8 hours regularly)"

"In 2026 I will do more (physical) exercises to prevent back/neck/shoulder injury from sitting at a PC too much"

■ What is JavaScript

- An programming language for code that runs inside a web browser, interacting with a web page

Nowadays, almost every web page contains JavaScript

Nowadays, also used coding outside of a browser (Node.js)

- In principle, JavaScript is as powerful as any other programming language (in particular, it's Turing-complete)

However, when running in the browser, it has only **limited access** to many resources, in particular, to your file system

- JavaScript can manipulate the various elements of a web page and communicate with other machines

We will see several examples of that in today's lecture

■ Language features

- An object-oriented script language, no particular similarity to Java

The name "JavaScript" was a marketing move to encourage adoption (Java was popular at the time and also used for web stuff)

Speed similar to Python, when interpreted line by line

Modern browsers perform just-in-time (JIT) compilation

- Variables are untyped ... but there is TypeScript

```
let x = 1;           // Scalar value.  
let s = "doof";      // String.  
let a1 = [1, "doof", bloed]; // Array (mixed types).  
let a2 = { "yes" : 5, "no" : 3 } // Associative array / map.
```

■ Variable declaration

- There are three kinds of declarations in JavaScript which differ with respect to their **scope** and their **changeability**

var: when declared outside of a function, the scope is the whole file; when declared inside of a function, the scope is the whole function ... **var is old, use rarely or never**

let: the scope is strictly the { ... } code block where the declaration is made

const: like let, but have to be initialized on declaration and then the value cannot be changed anymore

■ DOM = Document Object Model

- Well-defined scheme for how to address elements in a web page, in particular by JavaScript (JS) code
- Example 1: get the contents of an element with a particular id on the web page

HTML: `<div id="result">NO RESULT YET</div>`

JS: `document.querySelector("#result").innerHTML = "42";`

- Example 2: Do something in response to a user action (in this case a keypress in an input field)

HTML: `<input id="input" type="text" autocomplete="off">`

JS: `document.querySelector("#input").addEventListener("input", function() { ... });`

■ JavaScript as part of your HTML

- Option 1: **inline** code directly in the HTML

`<script>... arbitrary JavaScript code ...</script>`

The code will be parsed and executed as soon as it is encountered in the HTML

- Option 2: read a file with JavaScript code

`<head>`

`<script src="search.js" defer> </script>`

`</head>`

Without "defer": Will be executed when encountered

With "defer": Will be executed after the complete HTML has been parsed and the DOM of the page has been constructed

■ Communicating with other machines via **fetch**

- In JavaScript, you can send a request to another machine and receive a result in an asynchronous function as follows:

```
async function myFct() {  
  ...  
  const response = await fetch("api?query=42")  
    .then(response => response.text())  
}
```

- This code uses many different concepts, briefly explained on the following slides (see the references for details)

asynchronous functions and the keywords **async**, **await** and **then** ... see slide 13

the => operator ... see slide 15

■ Asynchronous functions

- **JavaScript is single-threaded**, not multi-threaded

In particular, at every given point in time, at most one line of JavaScript code is interpreted and executed

- Still, JavaScript supports asynchronous functions

Some built-in functions are asynchronous (like `fetch` or `setTimeout`) or a function can be declared **async** (next slide)

- Asynchronous functions can spend part of their time **outside** of the normal JavaScript execution

For example, `fetch` waits for the other machine to send sth

In the meantime, other JavaScript can (and will) be executed

■ `async`, `await`, `then`

- All three of these pertain to asynchronous execution

`async`: When a function calls any asynchronous function, it must be declared `async` (it's a compiler error otherwise)

`await`: Wait for the result of an asynchronous function and execute other JavaScript code in the meantime

`then`: When the result of the asynchronous function is ready, process it with the given function

- All three of these are used in `search.js` and needed for ES10

In particular, see slide 10: `"fetch"` first returns the header, `"then"` returns the body as an object of the specified type (text, JSON, etc.), `"await"` blocks until this is done

■ Promises

- Asynchronous functions like `fetch` return a object called **promise**. A promise has two components:

1. The current status of the asynchronous execution ("pending", "fulfilled", "rejected")
2. The result of the function (if "fulfilled")

- One can also work explicitly with Promise objects

```
const promise = new Promise(function(resolve, reject) {  
  // Code that calls the function "resolve" on success  
  // and the function "reject" on failure  
})
```

- For the built-in functions you can just use **async**, **await**, and **then** and the handling of Promises happens implicitly

■ The => operator

- In JavaScript many functions have an anonymous function as an argument, for example:

```
response = await fetch("...").then(  
    function(response) { return response.json() } );
```

- The => is simply a shortcut for an anonymous function that consist solely of a return statement ... so the following code is exactly equivalent to the code above

```
response = await fetch("...").then(  
    response => response.json() );
```

- It also works for multiple arguments

```
new Promise( (resolve, reject) => reject("I always fail") );
```

■ JSON = JavaScript Object Notation

- The result from a computation is often a complex object, e.g., an array or a map ... if sent as a mere string, we would need code to parse that string on the JavaScript side
- **JSON** is content already in the form of JavaScript code

```
[ 2, 3, 5, 7, 11, 13 ]           // an array
```

```
{ "query": "6*7", "result": "42" } // a map
```

- You can assign the JSON object directly to a variable (on the previous slide it is assigned to response) and use it:

```
response[2]           // third element, if it is an array
```

```
response.result       // value for key, if it is a map
```

■ Multi-threaded server

- Your JavaScript from ES10 will generate concurrent requests
When you type fast, new requests will be send before the previous requests have been completed
- A typical server would therefore process each request in an own thread, so that the processing can be done in parallel and the server is not blocked when a request is expensive to compute
- However, Python is notoriously terrible at multi-threading

Python has a so-called **global interpreter lock**, that is, only one piece of Python code can be interpreted at a time

There is a module called **multiprocessing**, but that forks the whole process with its complete data (for ES9: the q-gram index)

However, there is now **python3.14-nogil** ... see forum post

■ Motivation

- Web Apps are particularly vulnerable to privacy breaches

Because data + code is sent back and forth between multiple computers (foreign to each other), with so many different layers of software and hardware in-between

- We will briefly look at three kinds of vulnerabilities today:

Access to private data

Execution of code injected by an attacker

Communication of trusted information to an untrusted site

- Top-10 web app vulnerabilities ... google: [OWASP Top Ten](#)

OWASP = Open Web Application Security Project

■ Access to private data

- When writing or configuring a web server, take care to serve only those files / data you want to serve
- Here is a very simple and easy-to-overlook exploit:

<http://tura:8888//etc/passwd>

- This is easily fixed by carefully restricting access

For example, only serve files in a certain directory subtree

Even safer: only server files from a "whitelist" ... for all other files, return a 404 (Not Found) or a 403 (Forbidden)

■ Code Injection

- Exploit: make a web site execute malicious code

Example 1: enter JavaScript into search box

`<a href="javascript:alert('Hacked!')">Click me!</a>`

Example 2: send someone a mail with a link

`...index.html?user=guest<script>alert("Ha!")</script>`

- Note: the `<script>...</script>` part can be made more unsuspecting by URL-decoding (see slide 31):

`...index.html?user=guest%3C%73%63%72%69%70...`

■ Code Injection

- Exploit: make a web site execute malicious code

Example 3: post to forum with some script in it

I have a question<script>... JavaScript code that sends user info by mail to evil person ...</script>

Note: The <script>...</script> will not show on the website, but code will be executed by **any client** viewing the post

JS code could also open Gmail Tab and inspect private mail

- This can be fixed by carefully checking the content that is dynamically added to a webpage

ES9: if you don't pay attention, **strange** things might happen

■ The Same-Origin-Policy (SOP)

- Domain + port of client and server URL must be **identical**

<http://tura:8888/search.html>

http://tura:8888/?q=6*7

- To understand why, consider the following scenario:

You somehow get redirected to an evil site that looks just like your banking website, e.g. <http://www.comdirekt.de>

Without the same-origin-policy, the evil site could now communicate with the bank server like the real site

Worse: with stolen session cookie, evil person could do anything in your name without you even participating

■ CORS = Cross-Origin Resource Sharing

- When JavaScript requests a resource from a **different** host+port (than the website on which the script is executed), the following header is added to the request:

Origin: http://<host name>:<port>

- The result (think: JSON) from that other host+port will be accepted only if it contains a header like the following:

Access-Control-Allow-Origin: http://<host name>:<port>

Otherwise: "Cross-Origin Request Blocked: ..."

- To allow any requesting site to use the result, use header:

Access-Control-Allow-Origin: *

■ Exceptions to the Same-Origin-Policy (SOP)

- JavaScript can be loaded from **anywhere**; this is important for loading JavaScript libraries:

```
<script src="http://code.jquery.com/jquery1.10.2.js">
```

- Seemed reasonable at the time, because in HTML, objects like images could also be loaded from anywhere
- However, this allows security hacks like **JSONP**

JSONP used to be a standard to send JSON cross-origin

The idea was to dynamically add a piece of HTML like
<script>callback("...")</script> to the DOM, which then executes callback which then does what one wants

This is no longer allowed, but there are still exploits like this

■ Motivation

- To represent text in binary, we need a standard for how to represent the characters of the alphabet, numbers, etc.
- For a very long time, this standard was **ASCII** :
 - 1 Byte per symbol = can represent 256 different symbols
- Obviously there are more than 256 symbols in the world
 - Chinese alone has (tens of) thousands of different symbols

■ Solution before Unicode

- Use the ASCII codes 32 – 126 for common symbols, which (almost) everybody working with computers needs

a-z A-Z 0-9 () [] { } , . : ; " ' ...

(Codes 0 – 31 and 127 used for control characters)

- For the ASCII codes 128 – 255, have (many) different **variants**, depending on the context

One such variant: **ISO-8859-1** = codes for the most frequently used funny characters from most of the European languages

à á â ã ä å ç è é ë ì í î ï ð ñ ò ó ô õ ö ø ...

- **Problem:** if you need more than one variant, you need to switch the encoding in the middle of the document

■ The Unicode solution

- Simply assign a **unique** number, called **code point**, to (almost) every character / symbol in the world, e.g.

a	:	97	(hex = 61)	
A	:	65	(hex = 41)	
ä	:	228	(hex = E4)	
α	:	945	(hex = 03B1)	requires 2 bytes
€	:	8364	(hex = 20AC)	requires 2 bytes
	:	128512	(hex = 1F600)	requires 3 bytes

- Unicode knows 1,114,112 code points (hex: 0 .. 10FFFF)

Note: **one byte** is not enough for most characters, and **two bytes** are also not enough for all characters

■ UTF = Unicode Transformation Format

- There are different schemes for how to actually represent these code points in binary
- **UTF-32:** always use **4 bytes** per code point
Enough for all 1,114,112 known code points
- **UTF-16:** use **2 bytes** for the common code points, and 4 bytes for the others ... *used for Java strings, see slide 36*
- **UTF-8:** use **1 byte** for the very common code points, and 2 or 3 or 4 bytes for the others ... *details on next 3 slides*

UTF-32 is a **fixed-size** encoding

UTF-16 and UTF-8 are **variable-byte** encodings

■ Details of UTF-8

- **1 Byte:** Code point in $[0, 127]$ = xxxxxxx

UTF-8 code: 0xxxxxxx 7 Bits

- **2 Bytes:** Code point in $[128, 2047]$ = yyyxxxxxxxx

UTF-8 code: 110yyyxx 10xxxxxx 11 Bits

- **3 Bytes:** Unicode in $[2048, 65535]$ = yyyyyyyyyxxxxxxxx

UTF-8 code: 1110yyyy 10yyyyxx 10xxxxxx 16 Bits

- **4 Bytes:** Unicode in $[65536, 2^{21} - 1]$ = zzzzyyyyyyyyyxxxxxxxx

UTF-8 code: 11110zzz 10zzyyyy 10yyyyxx 10xxxxxx 21 Bits

In principle, this could continue with 5 bytes and 6 bytes,
but $2^{21} \approx 2\text{M}$ is enough for the 1.1M Unicode code points

■ UTF-8 has the following nice properties

- **ASCII** compatible: a string of characters with ASCII codes < 128 is the same in ASCII as in UTF-8

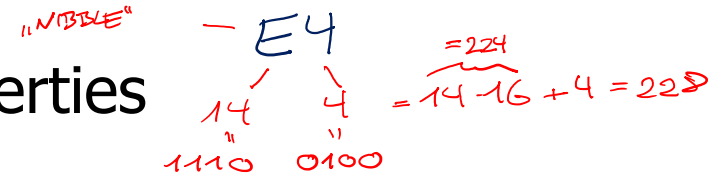
So old C / C++ code only fails on the special characters

- **ISO-8859-1** compatible: characters with code $1xyyyyyy$ have the 2-byte UTF-8 encoding $1100001x\ 10yyyyyy$

For ex: ä = 228 = 0xE4 = 0b11100100 in ISO-8859-1
and 0b1100000011 0b10100100 = 0xC3 0xA4 in UTF-8

- Only rarely used characters need more than 2 bytes
- Easy to decode: codes start and end at byte boundaries, and one can decode starting from anywhere within a string

Just move to the next byte not starting with 10



$= 12 = C$ 3 $= 10 = A$ 4

■ Some more properties of UTF-8

- In a multi-byte UTF-8 character all bytes are ≥ 128 , and vice versa such bytes occur only for multi-byte characters
- The number of leading 1s in the first byte of a multi-byte character is equal to the number of bytes of its code
- For every Unicode in $[0, 2^{21} - 1]$ there is **exactly one** valid UTF-8 multi-byte sequence
- But not all multi-byte sequences are valid UTF-8, for example:

1100000x 10xxxxxx is not valid

This should be encoded with 1 byte, as 0xxxxxxx

- Invalid bytes in a UTF-8 sequence are shown as 

■ URL decoding and encoding, motivation

- In a URL, only a restricted character set is allowed:
a-z A-Z 0-9 \$ % / - _ . + ! * ... and a few more
- In particular, the following characters are **not** allowed:
space, ä, ã, â, ...
- The arguments of a GET request are part of the URL

In particular, the ?query=... part of your web app for ES8

For ES9, this part can contain funny characters like
in Smørrebrød, Gemüsesoufflé, or Crème brûlée

■ URL decoding and encoding, realization

- The encoding of the HTML is specified in the head section

`<head> ... <meta charset="utf-8"> ... </head>`

For example, if we specify `charset="iso-8859-1"` and the text file contains an `ä` in UTF-8, the web page will show `Ã¸`

`ö` is `C3A4` in UTF8, but in ISO-8859-1, `C3` is `Ã` and `A4` is `¸`

- In fetch, special characters are encoded by a `%` followed by the code in hex-decimal ... the default encoding is UTF-8

`ä` : UTF-8 code `C3A4` → URL-encoded as `%C3%A4`

■ Encoding in files

- Inside a text editor, what you see is depends on the encoding used by the editor to interpret the file

This can usually be changed in the menu of the editor

- When you type or print something on the terminal, what you see depends on the encoding used by the terminal

This can usually be changed in the menu of the terminal

- To view the **byte-wise** contents of a file, independent of it's encoding use the Linux tool **xxd**, for example:

```
xxd <file>          // hex codes + chars, 16 bytes per line
xxd -g 1 <file>     // groups of 1 byte (default is: 2 bytes)
xxd -b <file>       // binary codes + chars, 6 bytes per line
```

■ Encoding in C++11

(outputs on UTF-8 terminal)

- In C++, there is `std::string` and `std::wstring`

```
// std::string = array of char (char = 1 byte)
```

```
std::string s = "\xc3\xa4"; std::cout << s;
```

outputs ä

```
// std::wstring = array of wchar_t (Unicode)
```

```
std::wcout.sync_with_stdio(false);
```

```
std::wcout.imbue(std::locale(""));
```

```
std::wstring w = L"ä"; std::wcout << w;
```

outputs ä

```
// Convert between std::string and std::wstring
```

```
std::wstring_convert<std::codecvt_utf8<wchar_t>> conv;
```

```
std::string utf8string = conv.to_bytes(L"ä");
```

{ c3, a4 }

```
std::wstring wstring = conv.from_bytes("ä");
```

{ U+00e4 }

■ Encoding in Java

- In Java, there is String and byte[]

// String = array of char (and a char has 2 bytes)

"ä".length(); 1 (U+00E4)

// Strings are UTF-16 encoded (and charAt runs in constant time)

"😄".length(); 2

"😄".charAt(0); 💎 (U+D83D)

"😄".charAt(1); 💎 (U+DE00)

"😄".codePointAt(0); 128512 (U+1F600)

// Convert between String and byte array

byte[] b = "ä".getBytes("UTF-8"); { 0xc3, 0xa4 }

new String(b, "UTF-8").charAt(0); ä (U+00E4)

Unicode 13/13

■ Encoding in Python3

outputs on UTF-8 terminal

- Python has both "byte array" strings and Unicode strings

```
// Byte array strings = b"..."
```

```
print(b"\xc3\xa4")
```

b'\xc3\xa4'

```
print(b"\xc3\xa4".decode("UTF-8"))
```

ä

```
print(b"\xc3\xa4".decode("ISO-8859-1"))
```

Ãä

```
// Unicode strings = u"..."
```

```
print(len(u"ä"))
```

1

```
print(u"ä".encode("UTF-8"))
```

b'\xc3\xa4'

```
// Specify source code encoding
```

```
# -*- encoding: iso-8859-1
```

```
print(u"ä")
```

Ãä

```
print(u"ä".encode("utf-8"))
```

b'\xc3\x83\xc2\xa4'

References

■ CORS

- http://en.wikipedia.org/wiki/Cross-origin_resource_sharing
- http://en.wikipedia.org/wiki/Cross-site_scripting

■ UTF-8, URL encoding and decoding

- <http://en.wikipedia.org/wiki/UTF-8>
- <http://www.utf8-chartable.de>
- http://www.w3schools.com/tags/ref_urlencode.asp