

# Information Retrieval

WS 2017 / 2018

Lecture 7, Tuesday December 5<sup>th</sup>, 2017  
(Web applications, part 2)

Prof. Dr. Hannah Bast  
Chair of Algorithms and Data Structures  
Department of Computer Science  
University of Freiburg

# Overview of this lecture

---

## ■ Organizational

- Your experiences with ES6      web application
- New exam date      **Mo Feb 19, 14 – 17 h**

## ■ Contents

- Web applications, part 2:

JavaScript	dynamically change content of web page
Vulnerabilities	privacy, code injection, cross origin
Cookies	store information across web sessions
Unicode	ISO-8859-1, UTF-8, URL encoding

**ES7:** make your web app dynamic ("search as you type")  
+ secure + use cookies + deal with Unicode properly

## ■ Experiences + Results

- Many of you found the topic very interesting and the exercise sheet a lot of fun to do
- The basic stuff was done fairly quickly for most
- Some problems with timeouts in C++ with boost::asio
- Some of you spent quite a bit of time on a nice design

We will show a selection next week, when you have added dynamic content and fixed encoding issues

- No errors in the TIP file this time
- Favorite Wikidata entites: [Q42](#), [Q149](#), [Q404](#), [Q2013](#)

## ■ Motivation

- A language that runs as part of a web page

Can do (almost) arbitrary computation

Can do (almost) arbitrary communication

Can dynamically change the contents of the web page in response to user actions

Nowadays, there is hardly a web page anymore without JavaScript in it

## ■ Language features

- An object-oriented script language, with a syntax similar to Java, hence the name

Speed similar to Python, when interpreted line by line

Modern browsers perform just-in-time (JIT) compilation,  
in order to achieve speeds similar to Java

- Variables are untyped

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

## ■ DOM = Document Object Model

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

In the HTML:

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

In the JavaScript:

```
document.getElementById("result").innerHTML = "42";
```

## ■ JavaScript as part of your HTML

- Option 1: inline code directly in the HTML

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

The code will then be executed when that part of the HTML is processed by the web browser

- Option 2: include file(s) with JavaScript code in the head section of the HTML file

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

Again: this will be executed when that part of the HTML is processed (in particular, before the body of the HTML)

- AJAX = Asynchronous JavaScript and XML
  - Old name for communication between JavaScript in browser and some server elsewhere ... typical code:

```
xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4 && xhr.status == 200) {
    response = xhr.responseText;
    ... process the response ... }
}
xhr.open("GET", "<url>", true);
xhr.send();
```

Much simpler with libraries like jQuery ... next slides



## ■ jQuery

- jQuery is a JavaScript library with convenient functions for all the common stuff ... include via

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

- Some example code snippets: (more on the next slides)

`$(document).ready(function() { ... })`    run the ... code as soon as the page has fully loaded

`$("#heading").html("Different text")`    change contents of HTML element with id "heading"

`$("#input").val()`    contents of HTML element with id "input"

Understand: \$ is a valid variable name in JavaScript and in jQuery used as a (convenient) alias for the central object of the library

## ■ jQuery

- Offers a much cleaner separation between static elements (HTML) and dynamic code (JavaScript)
- For example: do something after each keypress

### **Raw JavaScript:**

HTML: `<input id="query" onkeypress="myFct()"/>`

JavaScript: `myFct() { /* ... code here ... */ }`

### **With jQuery:**

HTML: `<input id="query">`

JavaScript: `$("#query").keyup(function() { ... })`

## ■ jQuery, communication with server

- For example: launch GET request and do something with the result:

```
var host = window.location.hostname;  
var port = window.location.port;  
var url = "http://" + host + ":" + port + "/?q=" + query;  
$.get(url, function(response) {  
    console.log(response);  
    $("#result").html(response.result);  
})
```

Note: writing to the "console" is quite useful for debugging;  
you can view it in the browser via F12 → Console

## ■ 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
```

```
{ "result": "42", "status": "OK" } // 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 maps
```

## ■ jQueryUI

- Extension of jQuery for more complex UI elements

```
<script src="https://code.jquery.com/ui/1.12.1/  
                                jquery-ui.js"></script>
```

```
<link rel="stylesheet" src="https://code.jquery.  
                                com/ui/1.12.1/themes/base/jquery-ui.css">
```

- For example, autocomplete from fixed set of strings

- HTML: `<input id="input">`

- JavaScript: 

```
$("#input").autocomplete({  
    source: [ ... array of strings from  
                which to autocomplete ... ]  
});
```

## ■ Motivation

- Web Apps are particularly vulnerable to privacy breaches

Because data + code is sent back 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
- We saw a simple problem + exploit in the last lecture

<http://etna.informatik.privat:8888//etc/passwd>

- This is easily fixed by carefully restricting access

For example, only serve files in a certain directory subtree

Even safer: a "whitelist" of files are served ... 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 27):

`...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

ES7: 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://etna.informatik.privat:8888/search.html>

<http://etna.informatik.privat:8888/?q=zurich>

- 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.postbank.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) then must be augmented by the following header

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

- The website can access the result if and only if both host name and port match **exactly**
- For a public service, the result can also be returned with

Access-Control-Allow-Origin: `*`

- Exceptions to the Same-Origin-Policy (SOP)
    - JavaScript can be loaded from **anywhere**
    - That way we could use jQuery without downloading it:  
`<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**, which dynamically adds `<script>myFct("...")</script>` to the HTML tree, which lets `myFct` do arbitrary things with "..."
- A hack to circumvent SOP ... which became a standard

## ■ Basic mechanism

- A cookie is simply a string associated with a web page that is stored on the client's computer
- Each client has its own cookie, which is typically used for user data and preferences
- A cookie can contain any string as contents, but the convention is that it contains a sequence of key-value pairs, separated by semicolons, for example:

`user=cookie-monster; prefers=kekse`

- Implementation in raw JavaScript is **very** simple, just read and write this string via the variable `document.cookie`
- See the next slide for examples

## ■ Adding key-value pairs to a Cookie

- To add a key-value pair, just write

```
document.cookie = "user=cookie-monster";
```

- Multiple assignments **add** to the string ... weird but true

```
document.cookie = "user=cookie-monster";  
document.cookie = "prefers=kekse";
```

- To overwrite the value for a key, just write again

```
document.cookie = "prefers=kekse";  
document.cookie = "prefers=kruemel";
```

- View all cookies stored for the application in browser via  
F12 → Application → Storage → Cookies

## ■ Getting the value for a particular key

- In raw JavaScript, need some string processing:

```
var cookies = document.cookie.split(";");
for (var i = 0; i < cookies.length; i++) {
    var args = cookies[i].replace(/\s/g, "").split("=");
    if (args[0] == "user") alert("Hi " + args[1] + " !!!");
}
```

- Different kinds of cookies

- **Chocolate chip cookie**

- Invented by Ruth Wakefield around 1938

- **Session cookie** ... lasts as long as your browser is open

- user=cookie-monster

- **Persistent cookie** ... lasts until the specified date

- user=cookie-monster; expires=Wed 05 Dec 2017 15:45

- **Third-party cookies** ... from JavaScript from other domains

- These typically contain information about what you did on a website (e.g. your preferences) or simply to track you



## ■ Using **js-cookie** ... <https://github.com/js-cookie/js-cookie>

- Setting a cookie

```
Cookies.set("user", "cookie-monster");
```

- Value of a cookie

```
var user = Cookies.get("user");
```

- Removing a cookie

```
Cookies.remove("user");
```

- Cookie with expiry date (10 days from now)

```
Cookies.set("user", "cookie-monster", { expires: 10});
```

## ■ 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 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 all the 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: 1 Byte not enough for most characters, and 2 Bytes are also not enough for all characters

## ■ UTF = Unicode Transformation Standard

- 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 37*
- **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-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  $[128, 2047]$  = yyxxxxxxxx

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

# Unicode 6/13

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

For ex:  $\ddot{a} = 228 = 0xE4 = 0b\underline{11100100}$  in ISO-8859-1  
and  $0b\underline{11000001}\underline{1}\ 0b\underline{10100100} = 0xC3\ 0xA4$  in UTF-8

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

Just move to the next byte not starting with **10**

$C3 = \tilde{A}$

$A4 = \ddot{A}$


That is the reason, why you often see  $\tilde{A}\ddot{A}$  in output with messed up encoding

## ■ Some more properties of UTF-8

- In a multi-byte UTF-8 character all bytes are  $\geq 128$ , 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 chars are **not** allowed:

space, ä, ã, â, ...

- Arguments of GET request are part of the URL

In particular, the ?q=... part of your web app for ES6

For ES7 (Wikidata search), this part can contain funny characters like in Smørrebrød or Gemüsesoufflé

## ■ URL decoding and encoding, realization

- Special characters are encoded by a % followed by the code in hex-decimal ... for example:

If encoding of web page is UTF-8

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

If encoding of web page is ISO-8859-1:

ä : ISO-8859-1 code E4 → URL-encoded as %E4

## ■ 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)

## ■ 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"))
```

Ã

```
// 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")
```

Ã

```
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-origin_resource_sharing)
- [http://en.wikipedia.org/wiki/Cross-site scripting](http://en.wikipedia.org/wiki/Cross-site_scripting)

## ■ Cookies

- [http://en.wikipedia.org/wiki/HTTP cookie](http://en.wikipedia.org/wiki/HTTP_cookie)
- [http://www.w3schools.com/js/js cookies.asp](http://www.w3schools.com/js/js_cookies.asp)

## ■ 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](http://www.w3schools.com/tags/ref_urlencode.asp)