

universität freiburg

Databases and Information Systems WS 25/26

Lecture 9: Web applications, part 1

December 16, 2025

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

Overview of this lecture

■ Organizational

- Your experiences with ES8 fuzzy prefix search

■ Contents

- How to build a search web application (with live coding)

Sockets create, accept, receive, send, close

Hypertext HTTP, Media Types, HTML, CSS, ...

ES8: build a web application that displays fuzzy prefix matches for the user query (using your code from ES8)

For ES9, you will build a simple **static** application, for ES10 you will make it **dynamic**; you will be amazed by the result

Experiences with ES8 1/2

■ Excerpts from your feedback [somewhat representative]

"Great assignment, I like that we revisit the inverted index"

"I truly appreciate that the sheets are built on [...] the lecture"

"Getting my hands dirty while implementing all from scratch is really fun in this class [...] encourages me to grasp every detail"

"Very rewarding to see the query working at the end"

"For me it was a nice sheet, a bit tricky but not too tricky"

"More challenging than the lectures before, but I liked it a lot"

"The hardest part was figuring out how to query the index"

"The text on the sheet was very long, but that is a good thing"

"Why is 'urn' missing at the bottom of the exercise sheet?"

Experiences with ES8 2/2

■ Why do only 20% of you submit exercise sheets?

"I have stopped two weeks ago because of lots of other stuff"

"Most humans prioritize short term enjoyment, and as such we need to get forced to do what's good for us"

"Some people just don't care that much and take the easy route"

"Only course with non-mandatory exercises in the 3rd semester"

"I was part of the 80% ... I am trying"

"I learn way more efficiently and thoroughly when I learn all of the stuff concentrated in one week, instead of a bit every week"

"Fungi are kinda awesome, and people like the numbers 7, 3, 4, 12 and 13 a bit too much"

■ Motivation

- Two programs / processes communicating with each other, possibly (and often) on two different machines
- For a typical web application:

Browser asking for **static** web page ... Lecture 9 (today)

Code in web page asking for **dynamic** contents ... Lecture 10

- Endpoint of such a communication channel is called **socket**
- Each socket belongs to a particular machine, called **host**, and has a unique id, called **port**, on that machine

The same machine can have many communication channels, hence the concept of (many) ports

- The basic request-response protocol

- **Server side:**

- Create a socket and bind it to a given port

- Listen on that port for incoming connections

- Wait, read request, compute result, send result

- **Client side:**

- Connect to socket on server (need machine name + port)

- OS automatically assigns unique port on client machine

- Send request, wait for result

■ Implementation, server side

- All programming languages have standard libraries for socket communication (for the server and for the client)

Python `socket`

Java `java.net.ServerSocket`

C++ `boost::asio` (asio = asynchronous IO)

- We provide basic code for the server socket communication on the Wiki (in Python)

During the lecture today, we will write more code, but that won't be provided this time; you only learn and remember the tricky details by developing this code **yourself**

Understand: one must **read in rounds**, since there is no universal "end of message" token

■ Implementation, server side, Python

- Create socket, wait for request, get request, send result

```
from socket import socket
```

```
server_socket = socket(AF_INET, SOCK_STREAM)
```

```
server_socket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
```

```
server_socket.bind(("0.0.0.0", port))
```

```
server_socket.listen()
```

```
connection, client_address = server_socket.accept()
```

```
connection.settimeout(5.0)
```

```
request_data = bytearray()
```

```
while (...): request_data.extend(connection.recv(num_bytes))
```

```
connection.sendall( ... answer data as byte array ...)
```

```
connection.close()
```

■ Implementation, client side

- For a web application, it suffices to implement the server
- The web browser plays the role of the client
- We can also test via simple communication programs, e.g.

`telnet <host> <port>    # Connect to host on given port`

- Alternatively, you can enter `http://<host>:<port>` in the address bar of your browser and see what happens

This will establish a connection using HTTP, see next slides

For ES8, you may also want to send HTTP request via the command line, for example using

`curl -v http://<host>:<port>    # -v prints headers etc.`

■ Bytes vs. Strings

- The data received from or sent to a client is **bytes**

In Python, you can use "bytes" objects (immutable, good enough for our purposes) or "bytearray" (can be modified)

- At other places in the code, it is often more convenient to work with **strings**
- To convert between bytes and strings one needs to specify how characters correspond to bytes

```
"knödel".encode("utf-8")          # result: b"kn\xc3\xb6del"
```

```
b"\xc3\xb6".decode("utf-8")      # result: "ö"
```

For this lecture and for ES9, we assume **UTF-8** encoding, which will be explained in detail in Lecture 10

■ HTTP = Hypertext Transfer Protocol

- Used by the browser to communicate with a (web) server
- The typical request looks as follows, where `/search.html` is the part of the URL after `http://<host>:<port>`

`GET /search.html HTTP/1.1 ...`

- An example of a typical result (`\r\n\r\n` before the content bytes)

`HTTP/1.1 200 OK`

`Content-Length: 137`

`Content-Type: text/html`

`... the 137 bytes of the content ...`

Note: HTTP 1 demands that newlines are encoded as `\r\n` but the recommendation is that a single `\n` is accepted, too

■ HTTP = Hypertext Transfer Protocol

- There are many more request types ... for example:

POST (instead of GET)

POST can send arbitrarily large data along with the URL

- And many more headers ... for example

HTTP/1.1 404 Not found

To indicate that the requested resource does not exist

HTTP/1.1 403 Forbidden

To indicate that this resource is a no no for you

For ES8, you should implement 404 and 403

■ Media types aka Content Types aka MIME types

- Standard names for the different types of content sent across the internet, formerly known as MIME types

MIME = Multipurpose Internet Mail Extensions

- Examples

text/plain

plain text

text/html

HTML ... see slides 15 + 16

text/css

CSS ... see slide 17

image/x-icon

ICO image (notably: favicon.ico)

image/jpeg

JPG image

application/javascript

JavaScript ... Lecture 10

application/json

JSON ... Lecture 10

■ Browser Development Console

- Extremely useful for debugging web applications, or in general to understand better what is going on

Chrome **F12** / Ctrl+Shift+I

Firefox **F12** / Ctrl+Shift+I

Internet Explorer **F12**

- Important sections for us:

Network: requests sent and results received

Elements: elements of the HTML page ... see Lecture 10

Console: output from the JavaScript ... see Lecture 10

■ HTML = Hypertext Markup Language

- Language for specifying the content of a web page
- XML-like language, general structure:

```
<html>  
  <head>  
    ... meta information + includes ...  
  </head>  
  <body>  
    ... contents of the page ...  
  </body>  
</html>
```

■ HTML

- Example tags for the `<head>...</head>` section:

```
<link rel="stylesheet" type="text/css" href="..." />  
<script src="..."></script>
```

Include **style** information and **code** ... see coming slides

- Example tags for the `<body>...</body>` section

<code><h1>...</h1></code>	Level-1 heading
<code><p> ... <p></code>	A paragraph of text
<code><input> ... </input></code>	Input field
<code><div> ... </div></code>	Arbitrary "logical" section

■ CSS = Cascading Style Sheets

- Specify style information (layout, font, color, etc) independent from the contents of the page
- Has its own (simple) syntax ... for example, all level-1 headings in blue and boldface

```
h1 { color : blue; font-weight: bold }
```

- When several rules apply to the same element, the most "specific" rule wins ... hence the name "cascading"

CSS is used by practically every website nowadays

For ES8, make a web page with a reasonably "**nice**" look and feel, using CSS ... it's worth spending time on this!

■ Forms

- Purpose: upon user action (typically: click on a button), load a new HTML page with new or updated information
- Typical HTML code:

```
<form>
```

```
  <input type="text" value="xyz" name="abc"/>
```

```
  <input type="submit" value="Search"/>
```

```
</form>
```

- The second `<input .../>` is shown as a button labeled "Search"
- When clicking on that button, the browser will append `?abc=xyz` to the URL and issue the corresp. GET request

With `<form action="prefix">` you can modify the URL

■ Template variables

- For a **SERP** (Search Engine Result Page), one wants the original page with the input augmented by the results
- This can be realized via template variables as follows:

```
<form>
```

```
  <input type="text" value="%QUERY%" name="query"/>
```

```
  <input type="submit" value="Search"/>
```

```
</form>
```

```
%RESULT%
```

- The server code can then, after having computed its result, substitute the `%...%` accordingly

For ES8, replace `%QUERY%` by the query and `%RESULT%` by the result (computed via your code from ES7)

■ URL Encoding and Decoding

- In a URL, only a restricted character set is allowed:
a-z A-Z 0-9 \$ % / - _ . + ! * ... and a few more
- In particular, the following are forbidden: **space**, ä, ã, â, ...
- Using forms, the query becomes part of the URL and "forbidden" characters must be "escaped"

This is called URL encoding (and decoding)

- In particular, a **space** becomes a **+** and an **ä** becomes **%C3%A4** or **%E4** (this will be explained in Lecture 9)

For ES8, you must explicitly replace all **+** in the URL by spaces again, but you can ignore URL encoding of special chars like ä ... we will deal with that in Lecture 9 and ES9

■ Reusing connections

- Most browsers want to keep the connection to the server open for future requests ... the corresponding request header and value is

Connection: keep-alive

- Your server for ES9 will only serve one request at a time

This can be a problem when serving the request takes some time and the browser sends a new request via the same connection while the server is still busy processing

- If you experience any problems with this, you can send the browser the following HTTP header together with your result (this will ask the browser to use a new connection for its next request)

Connection: close

References

- Relevant Wikipedia articles (in order of appearance)

http://en.wikipedia.org/wiki/Network_socket

http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

http://en.wikipedia.org/wiki/Internet_media_type

<http://en.wikipedia.org/wiki/HTML>

http://en.wikipedia.org/wiki/Cascading_Style_Sheets

[https://en.wikipedia.org/wiki/Form_\(HTML\)](https://en.wikipedia.org/wiki/Form_(HTML))

https://en.wikipedia.org/wiki/Web_template_system

<https://en.wikipedia.org/wiki/Percent-encoding>

https://en.wikipedia.org/wiki/List_of_HTTP_header_fields