HTTP / HTML WWW

OVERVIEW OF HTTP, HTML, WWW AND WEB TECHNOLOGIES

Peter R. Egli
peteregli.net

Contents

1. Web Precursors
2. WWW Elements
3. HTML
4. Web Address
5. HTTP RFC2616 Protocol
6. The HTTP PATCH method - RFC5789
7. HTTP 1.0 versus HTTP 1.1
8. HTTP Authentication
9. Active / Stateful Web
10. Web Caching
11. Web Audio / Video Streaming

1. HTTP / Web precursors

➔ The world (=Internet) before the advent of HTTP/HTML:

1. WAIS: Wide Area Information Service was used for retrieval of documents from distributed and indexed databases.
2. gopher: precursor of WWW; lets browse servers through menues; gopher made use of WAIS, archie, FTP and telnet and provided a common interface; gopher was text only;
3. FTP: File Transfer Protocol was and still is used for tranferring files from host to host.
4. archie: allows accessing public lists of files.

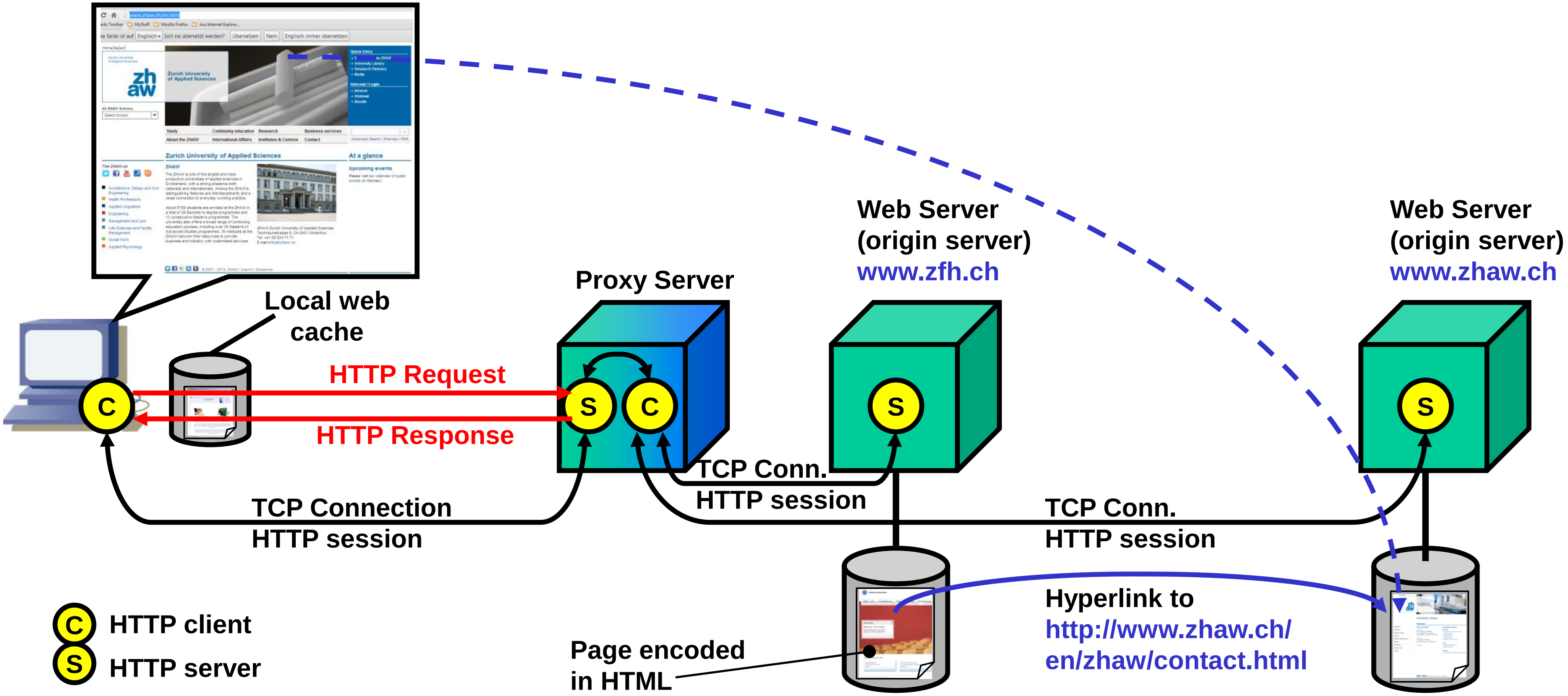
➔ HTTP / HTML is the „Lingua Franca“ of the Net; almost all information is today accessible through HTTP. HTTP has become the standard format for information representation (user graphical front end).

2. Web elements

➔ Web client (browser program or other) retrieves pages encoded in HTML from web server and displays the page (graphical representation).

➔ Page is taken from local cache (containing previous HTML responses) if it has not been modified (and has been retrieved previously).

➔ Proxy server has an HTTP client and server and performs some function, e.g. filtering (blocking certain pages, replacing content etc.) and possibly also caching.



3. HTML primer (1/4)

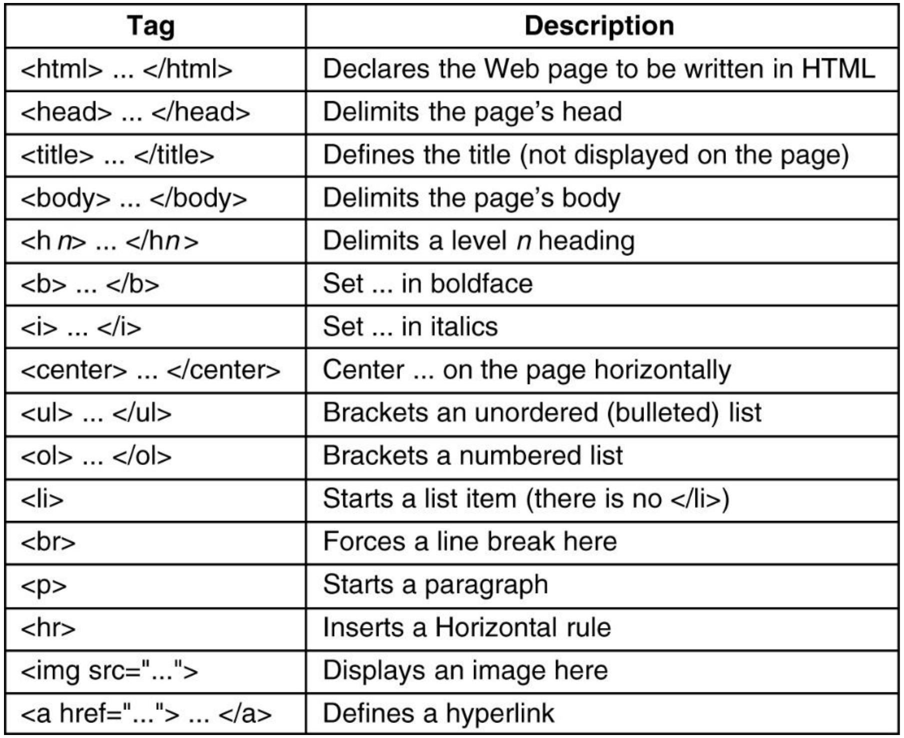
➔ Pages retrieved from web server are encoded in HTML (Hypertext Markup Language).

➔ HTML is one of many tag (<>) based languages (SGML Structured Generalized Markup Language).

➔ While very successful for web HTML has some problems:

- Mixture of content (information) and formatting (layout tags such as <br> and <b>).
- HTML is not well-formed (some tags do not have opening/closing tag pair such as <br>).
- Proper nesting of tags is sometimes violated.

➔ HTML tags:



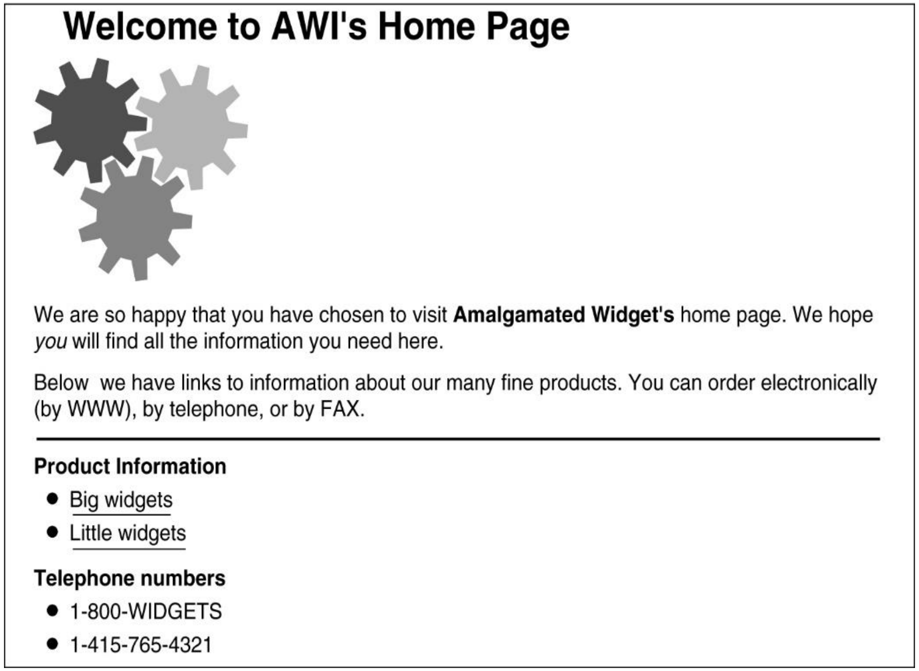
Tag	Description
<html> ... </html>	Declares the Web page to be written in HTML
<head> ... </head>	Delimits the page's head
<title> ... </title>	Defines the title (not displayed on the page)
<body> ... </body>	Delimits the page's body
<h *n*> ... </h*n*>	Delimits a level *n* heading
<b> ... </b>	Set ... in boldface
<i> ... </i>	Set ... in italics
<center> ... </center>	Center ... on the page horizontally
<ul> ... </ul>	Brackets an unordered (bulleted) list
<ol> ... </ol>	Brackets a numbered list
<li>	Starts a list item (there is no </li>)
<br>	Forces a line break here
<p>	Starts a paragraph
<hr>	Inserts a Horizontal rule
<img src="...">	Displays an image here
<a href="..."> ... </a>	Defines a hyperlink

3. HTML primer (2/4)

- ➔ HTML is a tree of nested tags that describe how the page should be displayed (more or less).
- ➔ Due to some freedom in displaying the pages will look differently on different browsers.
- ➔ Example HTML code and according display:

```
<html>
<head><title> AMALGAMATED  WIDGET, INC. </title> </head>
<body> <h1> Welcome to AWI's Home Page</h1>
<img src="http://www.widget.com/images/logo.gif" ALT="AWI Logo"> <br>
We are so happy that you have chosen to visit <b> Amalgamated Widget's </b>
home page. We hope <i> you </i> will find all the information you need here.
<p>Below we have links to information about our many fine products.
You can order electronically (by WWW), by telephone, or by fax. </p>
<hr>
<h2> Product information </h2>
<ul>
   <li> <a href="http://widget.com/products/big">  Big widgets </a>
   <li> <a href="http://widget.com/products/little">  Little widgets </a>
</ul>
<h2> Telephone numbers</h2>
<ul>
   <li> By telephone: 1-800-WIDGETS
   <li> By fax: 1-415-765-4321
</ul>
</body>
</html>
```

(a)



Welcome to AWI's Home Page

We are so happy that you have chosen to visit **Amalgamated Widget's** home page. We hope *you* will find all the information you need here.

Below we have links to information about our many fine products. You can order electronically (by WWW), by telephone, or by FAX.

Product Information

- Big widgets
- Little widgets

Telephone numbers

- 1-800-WIDGETS
- 1-415-765-4321

3. HTML primer (3/4)

➔ How to separate style/format and content (1):

Solution 1: CSS Cascading Style Sheets:

☺ Export some of the formatting information into separate files (*.css files).

☺ Reuse of defined templates to give pages the same look (background, colors etc.).

☺ Hierarchy of style sheets.

Example:

In HTML page header:

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

In HTML page body:

```
<p>Simple content text ready for control.</p>
```

In CSS file:

```
p {
  font-family: Verdana, Arial, Helvetica, sans-serif;
  font-size: 12px;
  color: #FF0000;
}
```

Example page:

http://www.csszengarden.com/

Solution 2: Usage of XHTML2 (eXtensible HTML):

XHTML2 is supposed to be the successor to HTML. Unlike HTML (=implementation of SGML) XHTML is an XML-language (XML is a more restrictive subset of SGML). XHTML has few presentational elements (CSS should be used instead for presentation=layout) but more structural elements. XHTML2 is not backward compatible with HTML.

See http://www.w3.org/TR/2005/WD-xhtml2-20050527/introduction.html.

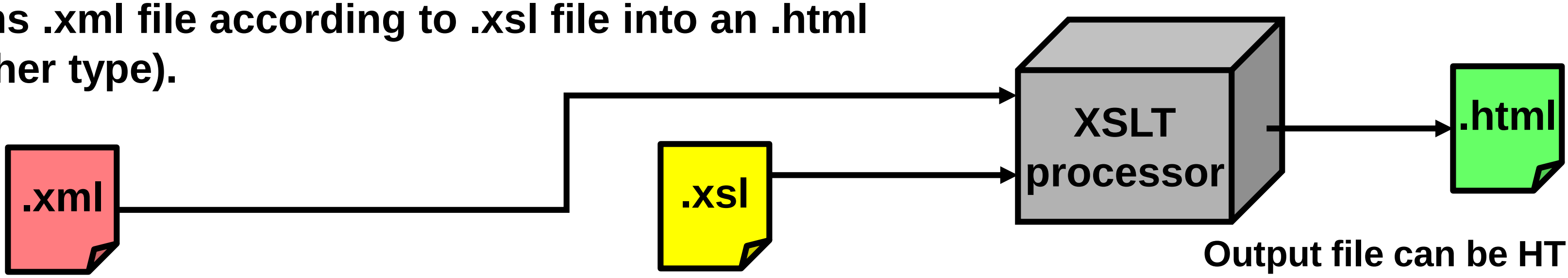
But: XHTML 2.0 discontinued by end of 2009 (to be replaced by HTML 5).

See http://www.w3.org/News/2009#item119

3. HTML primer (4/4)

➔ How to separate style/format and content (2):

<u>**Solution 3: XML and XSL (eXtensible Stylesheet Language, XSLT):**</u> **XSLT processor transforms .xml file according to .xsl file into an .html file (or other type).**



XML file contains content (data:)

```
<?xml version="1.0" ?>
<?xml-stylesheet type="text/xsl" href="b5.xsl"?>

<book_list>

<book>
   <title> Computer Networks, 4/e </title>
   <author> Andrew S. Tanenbaum </author>
   <year> 2003 </year>
</book>

<book>
   <title> Modern Operating Systems, 2/e </title>
   <author> Andrew S. Tanenbaum </author>
   <year> 2001 </year>
</book>

<book>
   <title> Structured Computer Organization, 4/e </title>
   <author> Andrew S. Tanenbaum </author>
   <year> 1999 </year>
</book>

</book_list>
```

XSL file contains transformation commands:

```
<?xml version=’1.0’?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/">

<html>
<body>

<table border="2">
  <tr>
     <th> Title</th>
     <th> Author</th>
     <th> Year </th>
  </tr>

  <xsl:for-each select="book_list/book">
  <tr>
     <td> <xsl:value-of select="title"/> </td>
     <td> <xsl:value-of select="author"/> </td>
     <td> <xsl:value-of select="year"/> </td>
 </tr>
 </xsl:for-each>
</table>

</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Output file can be HTML, RTF or any other text type:

```
<html>
<body >
<table border=“2”>
  <tr>
    <th>Title</th>
    <th>Author</th>
    <th>Year</th>
  </tr>
  <tr>
    <td>Computer Networks, 4/e</td>
    <td>Andrew S. Tanenbaum</td>
    <td>2003</td>
  </tr>
 <tr>
    <td>Modern Operating Systems, 2/e</td>
    <td>Andrew S. Tanenbaum</td>
    <td>2001</td>
  </tr>
 <tr>
    <td>Structured Computer Org., 4/e</td>
    <td>Andrew S. Tanenbaum</td>
    <td>1999</td>
  </tr>
</body>
</html>
```

4. Web address (1/4)

➔ URL Uniform Resource Locator RFC1738:

A web address (URL) is composed as follows:

```
scheme://[username[:password]@]host[:port]/path[?#query]
```

scheme	= Protocol (http, ftp, smtp, gopher, file, news, mailto, telnet, ldap).
username:password	= Optional credentials (no longer supported by M$ IE due to security concerns, Mozilla Firefox supports it though).
host	= FQDN of host, e.g. www.zhaw.ch; to be translated into IP address by DNS resolver.
port	= Listening port of server (optional); 80 (default) or 8080 (usually proxy).
path	= Path of requested resource on server, e.g. index.html (default).
query	= Optional context information (sequence of parameters in “key=value” notation separated by ‘&’).

Examples:

http://www.zhaw.ch:80/index.html	‘Standard’ HTTP URL.
http://www.google.ch/search?hl=de&q=url+query&meta=	HTTP URL with query string.
ftp://bart.isz.ch/	FTP protocol, initiates FTP transfer through web browser.
ftp://anonymous:mypassword@ftp.zhaw.ch/	FTP URL with credentials.
file:///C:/temp/page.html	File protocol, retrieval of local file (not via HTTP from server).
mailto:xegp@zhaw.ch	Mail protocol, starts mail client (user agent).

4. Web address (2/4)

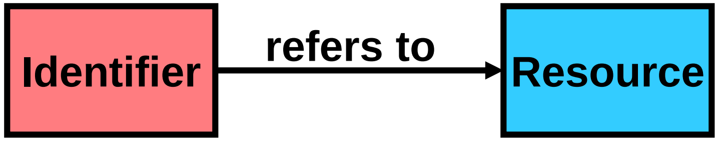
➔ URx RFC3986 (1):

➔ Resource:

A resource on the web is anything that can have an identity. The resource does not necessarily have to be accessible through a network. The term resource is rather conceptual, but includes also anything we consider a resource in the narrower sense.

Examples of resources:

- A physical noticeboard
- All people within the University of Zürich
- A book
- A sentence from a book
- A GIF image
- An HTML document
- A postscript document residing on an FTP server



➔ Identifier:

An identifier is an object that acts as a reference to something that has an identity (resources).
In the web an identifier is a string that conforms to the URI syntax. Classes of identifiers are URL, URN and URC.

Examples of identifiers:

- A forename and a surname
- A postcode
- An ISBN number
- An URL like http://www.zhaw.ch

4. Web address (3/4)

➔ URx RFC3986 (2):

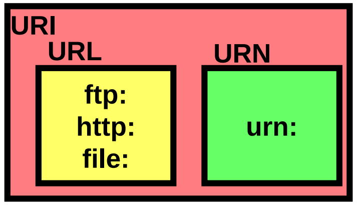
➔ URL Uniform Resource Locator:

URLs are standard way to address web documents. URLs give (preferred) location of resource (document) on web (IP address) and the access mechanism (scheme: ftp, http, mailto etc.).
Example: http://www.zhaw.ch

- ☺ Simple, widely used.
- ☹ Identify resource by location rather than name (real-world equivalent: identify person by location rather than name).
- ☹ If resource moves to another location/server links are broken („404 document not found“), i.e. URLs are transient.
- ☹ URLs can not carry describing meta-data (describing resource in more detail).
- ☹ Many tools confuse URL and URN and silently assume that URL is name and location of a resource.

➔ URI Uniform Resource Identifier:

Both URLs and URNs are URIs (super-class):



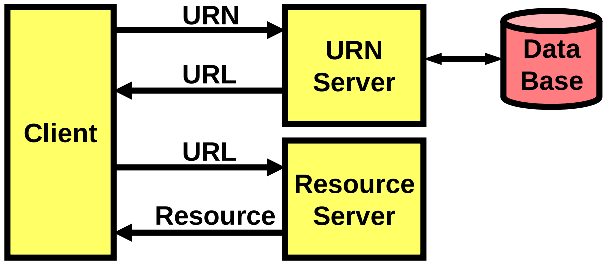
➔ URC Uniform Resource Citation:

URCs are descriptors of resources (URCs point to metadata of resources). They are unlikely to become a standard.

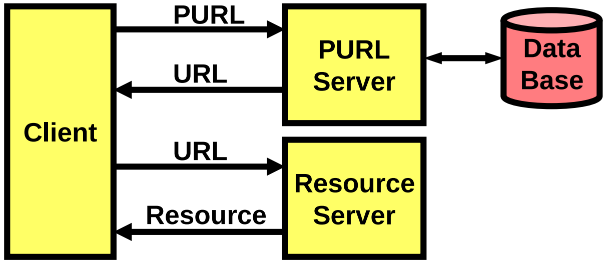
4. Web address (4/4)

➔ URx RFC3986 (3):

➔ URN Uniform Resource Name:
URNs identify a resource by a unique name, preceded by a namespace avoiding conflicts (delegation, hierarchy).
URNs are per definition globally unique and thus have global scope (conflict avoidance).
A URL is obtained by querying a URN server.
Open issues (not standardized yet): URN assignment, URN servers
Example: urn:/ISBN:7-678-12345-7.



PURL Peristent URLs:
PURLs (Persistent URLs) are a form of URNs. PURLs are meant to be an intermediate step in the development of URNs (and their usage).
PURLs are URLs that „point to“ another URL that points to the resource.
PURL is implemented as HTML redirection.
see https://purl.org/



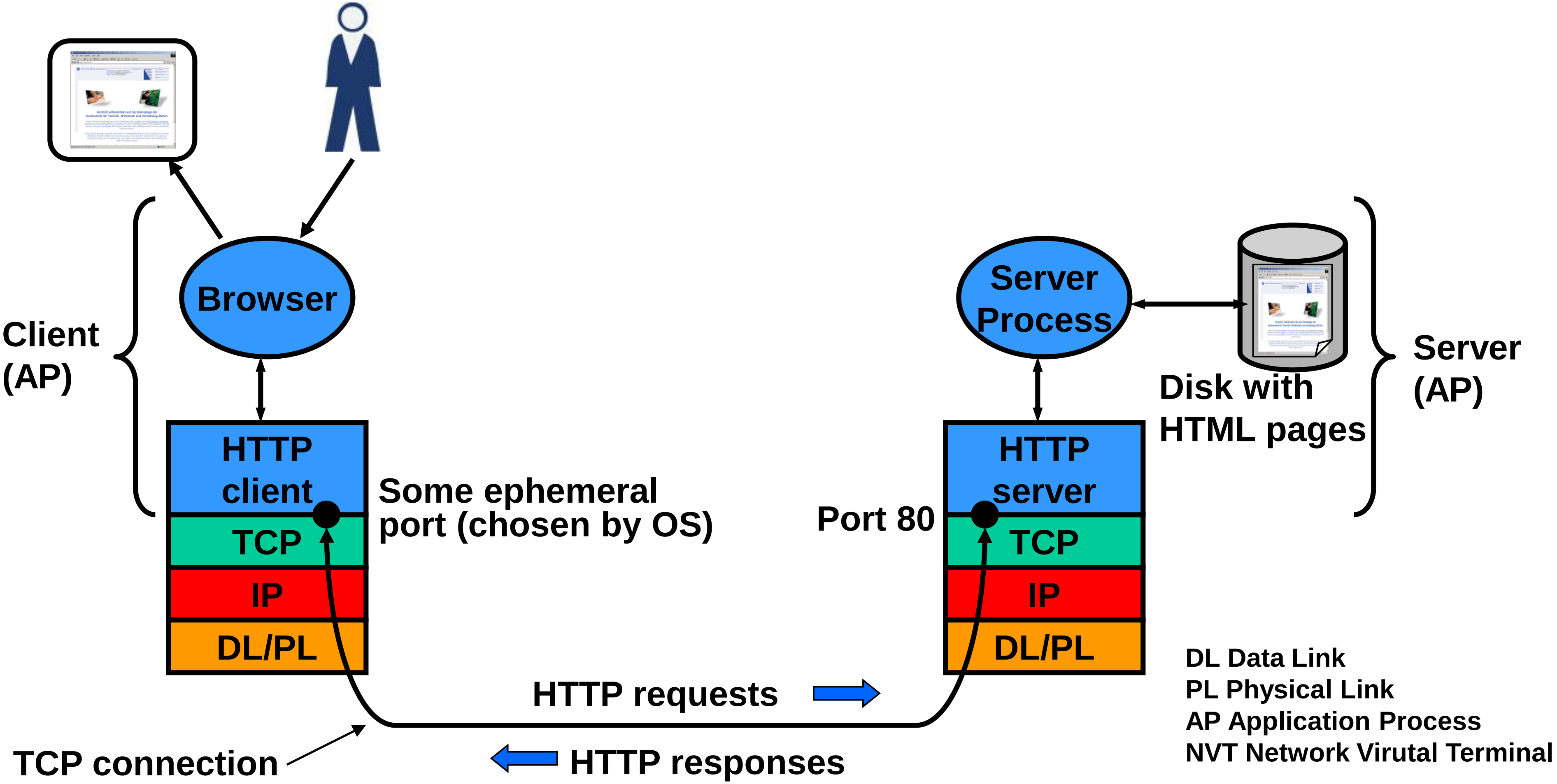
➔ Comparison of URN/URL with other concepts:

	URx:	DNS:	Books:
Location of copy of resource („where“)	URL	IP address	Location where to download E-book
Identification („what“)	URN	Domain name	ISBN-number

5. The HTTP RFC2616 protocol (1/7)

➔ HTTP is stateless: Client asks for info, gets it and then drops out (closes TCP connection). This statelessness is both HTTP‘s strength and weakness (simple but no session/state).

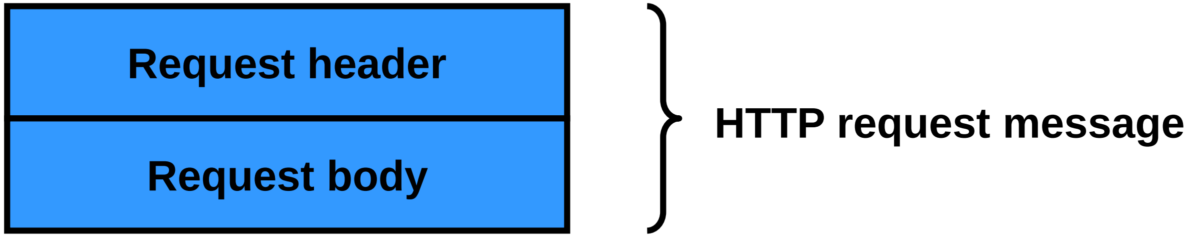
➔ Like SMTP HTTP commands/responses are based on NVT ASCII.



DL Data Link
PL Physical Link
AP Application Process
NVT Network Virutal Terminal

5. The HTTP RFC2616 protocol (2/7)

➔ **HTTP request consists of request header and an optional request body:**



➔ **Example:**

1. Request header:

```
GET /index.html HTTP/1.1
Connection: Keep-Alive

User-Agent: Mozilla/4.7 [en] (WinNT; I)
Accept: image/gif, image/x-xbitmap,
image/jpeg, image/pjpeg, image/png,*/*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8

```

Request type (GET), path, protocol & version.
Signal to server that connection should not be closed but reused for further requests.
Identifier of the client (Netscape).
Data types and encoding this client can handle (MIME types).

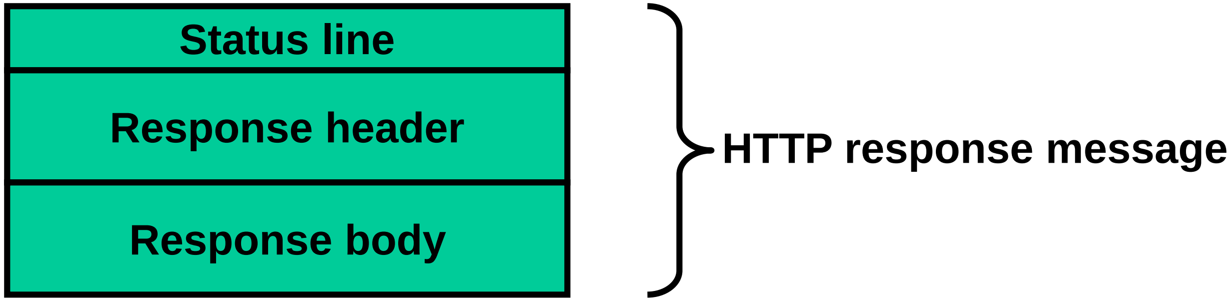
Allowed encoding of data in response.
Language that client can handle.
Character set that client can handle.
An empty line is inserted at the end of the header (like SMTP). This will cause the server to deliver the response (in the same TCP connection).

2. Request body:

The body is empty here. It is usually empty for normal GET requests; if non-empty the body contains data of a POST command.

5. The HTTP RFC2616 protocol (3/7)

➔ HTTP response consists of status line, response header and response body (HTML page):



➔ Example:

1. Status line:

```
HTTP/1.1 200 OK
```

Return code .

2. Response header:

```
Date: Thu, 09 Dec 1999 12:23:29 GMT
Server: Apache/1.3.9 (Linux Debian 2.2) ApacheJServ/1.0
Last-Modified: Mon, 04 Oct 1999 09:33:15 GMT
ETag: "0-374-37f8745b‟
Accept-Ranges: bytes

Content-Length: 884
Content-Type: text/html
```

Date/time when document was sent.
Timestamp and identifier of server.
Time stamp of the retrieved document.
Tag used for cache validation.
MIME formatted information on charset, length and type (html) of the result.
Length of data in body.
Type of content (body format).

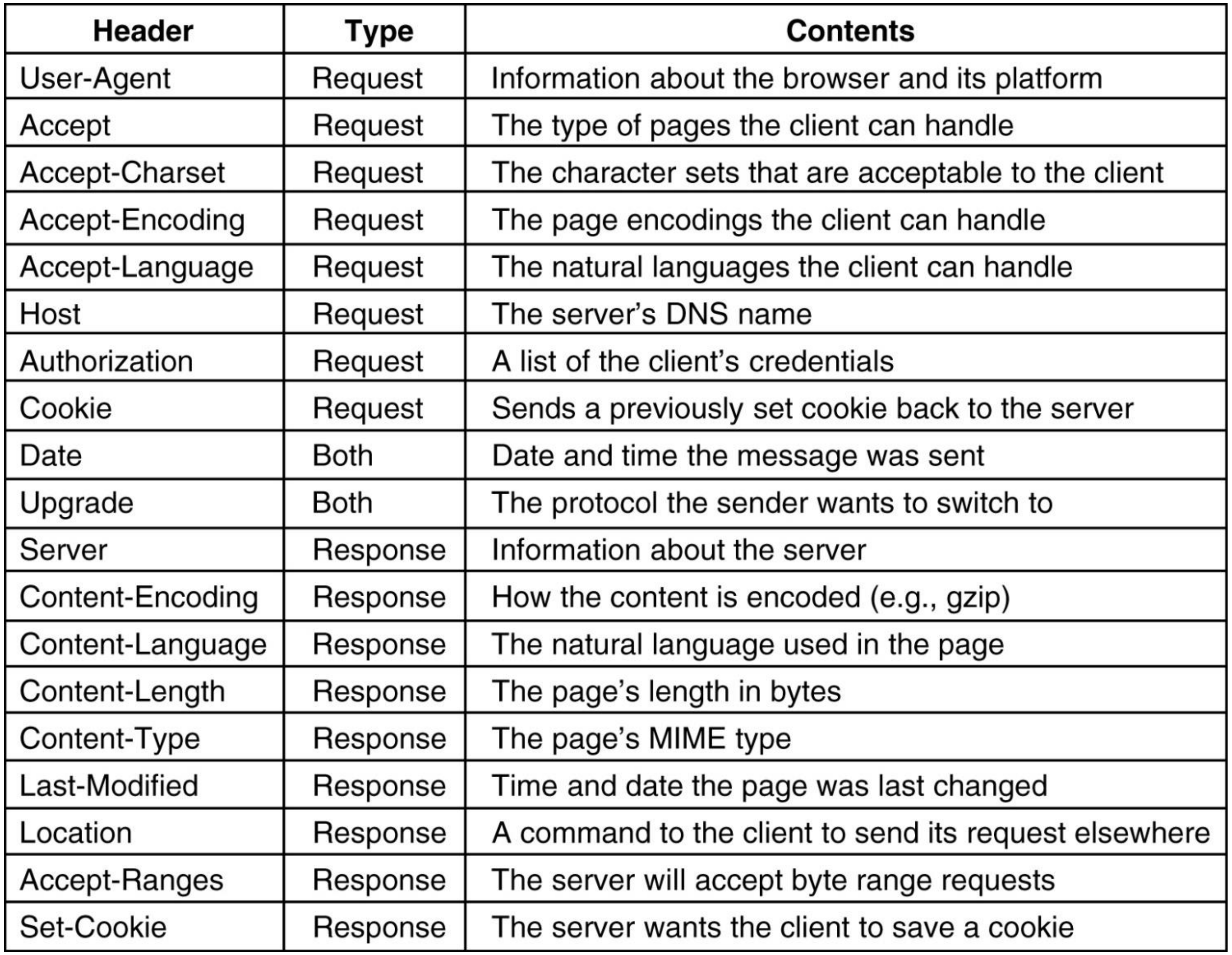
3. Response body:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2
 Final//EN"> [...]
```

The HTML document.

5. The HTTP RFC2616 protocol (4/7)

➔ **There are various HTTP request header fields to describe the session:**



Header	Type	Contents
User-Agent	Request	Information about the browser and its platform
Accept	Request	The type of pages the client can handle
Accept-Charset	Request	The character sets that are acceptable to the client
Accept-Encoding	Request	The page encodings the client can handle
Accept-Language	Request	The natural languages the client can handle
Host	Request	The server's DNS name
Authorization	Request	A list of the client's credentials
Cookie	Request	Sends a previously set cookie back to the server
Date	Both	Date and time the message was sent
Upgrade	Both	The protocol the sender wants to switch to
Server	Response	Information about the server
Content-Encoding	Response	How the content is encoded (e.g., gzip)
Content-Language	Response	The natural language used in the page
Content-Length	Response	The page's length in bytes
Content-Type	Response	The page's MIME type
Last-Modified	Response	Time and date the page was last changed
Location	Response	A command to the client to send its request elsewhere
Accept-Ranges	Response	The server will accept byte range requests
Set-Cookie	Response	The server wants the client to save a cookie

5. The HTTP RFC2616 protocol (5/7)

➔ HTTP methods (= ‚commands‘ client➔server):

Method	Description
GET	Request to read a resource
HEAD	Request to read a web page's header
PUT	Request to store a resource
POST	Request to append a resource
DELETE	Request to delete a resource
TRACE	Request to echo the incoming request
CONNECT	Reserved for tunneling through proxies
OPTIONS	Request to query the server's communication possibilities (methods etc.)
PATCH	Request for partial updates (see The HTTP PATCH method)

➔ HTTP uses return codes (server➔client) similar to SMTP and FTP. The codes are organized in classes (e.g. 2xx codes for success):

Code	Meaning	Example
1xx	Informational	100 = Server agrees to accept the client's request.
2xx	Success	200 = The request succeeded.
3xx	Redirection	301 = The requested resource moved parmentently.
4xx	Client error	404 = The requested resource was not found.
5xx	Server error	500 = An internal server error occurred.

5. The HTTP RFC2616 protocol (6/7)

➔ HTTP session with TELNET (TELNET client is „browser“):

HTTP uses NVT ASCII, thus connection to web server can also be established with TELNET client (even though TELNET client will not display pages graphically).

```
cmd> telnet www.zhaw.ch 80
GET / HTTP/1.1
Host: www.zhaw.ch
Connection: Keep-Alive

[here comes the page index.html]

GET /fileadmin/templates/img/zhaw_logo_de.gif HTTP/1.1
Host: www.zhaw.ch
Connection: Keep-Alive

[here comes the picture zhaw_logo_de.gif]

etc.
```

5. The HTTP RFC2616 protocol (7/7)

➔ HTML forms:

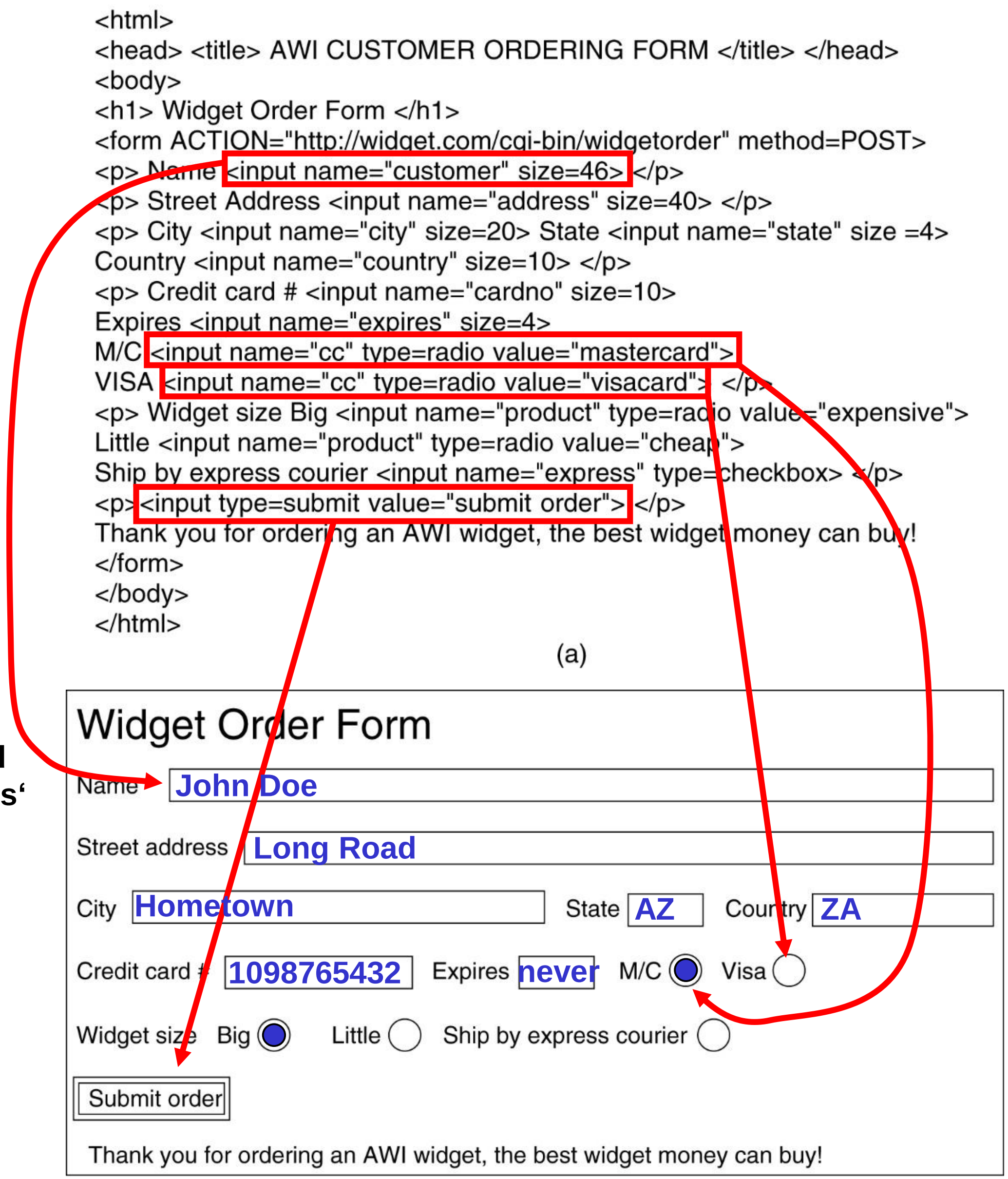
Q: How to send data from client to server?

A: Use forms (form tags):

```
<FORM action=„<URL>“ method=„post‘>
  <INPUT name=„“ type=text size=„6“>
  <INPUT name=„“ type=textarea rows=„9“ cols=„10“>
  <INPUT name=„“ type=password size=„8“>
  <INPUT name=„“ type=radio value=„“>
  <INPUT name=„“ type=radio value=„“>
  <INPUT name=„“ type=submit value=„“>
  <INPUT name=„“ type=checkbox value=„“>
  <INPUT name=„“ type=reset value=„“>
</FORM>
```

Once the user presses the submit button the browser sends the entered data concatenated into a string with a HTTP POST or GET request message. Spaces are replaced by ‚+‘ and parameters are separated by ‚&‘. Empty form fields‘ values are not sent.

```
POST /cgi-bin/widgetorder HTTP/1.1
Host: widget.com
\r\n
customer=Johnny+Sixpack&address=Long+Road
&city=Hometown&state=AZ&country=ZA
&cardno=1098765432&expires=never
&cc=visacard&product=expensive
```



```
<html>
<head> <title> AWI CUSTOMER ORDERING FORM </title> </head>
<body>
<h1> Widget Order Form </h1>
<form ACTION="http://widget.com/cgi-bin/widgetorder" method=POST>
<p> Name <input name="customer" size=46> </p>
<p> Street Address <input name="address" size=40> </p>
<p> City <input name="city" size=20> State <input name="state" size =4>
Country <input name="country" size=10> </p>
<p> Credit card # <input name="cardno" size=10>
Expires <input name="expires" size=4>
M/C <input name="cc" type=radio value="mastercard">
VISA <input name="cc" type=radio value="visacard"> </p>
<p> Widget size Big <input name="product" type=radio value="expensive">
Little <input name="product" type=radio value="cheap">
Ship by express courier <input name="express" type=checkbox> </p>
<p><input type=submit value="submit order"> </p>
Thank you for ordering an AWI widget, the best widget money can buy!
</form>
</body>
</html>
```

(a)

Widget Order Form

Name John Doe

Street address Long Road

City Hometown State AZ Country ZA

Credit card # 1098765432 Expires never M/C ● Visa ○

Widget size Big ● Little ○ Ship by express courier ○

Submit order

Thank you for ordering an AWI widget, the best widget money can buy!

6. The HTTP PATCH method - RFC5789 (1/2)

Problem with PUT method:

HTTP PUT does not allow partial updates of a resource.
With HTTP PUT, a resource update requires writing the entire resource, thus imposing network and server load.

Solution:

Partial resource update with HTTP PATCH method.
Partial updates with PATCH are always atomic, i.e. the server must perform the partial update in its entirety and never provide partially modified resources in a GET request.
The PATCH method is neither safe nor idempotent (see RFC2616).

Patch format:

The HTTP body contains a sequence of patch operations to be applied to the resource identified by the URI.
The format of the patch operations in the HTTP body is application specific.
Possible formats are:
a. JSON Patch (RFC6902, see example on next page)
b. XML Patch with XPath (RFC5261)
c. Unix diff format (see diff man pages)
d. Proprietary and application specific format

6. The HTTP PATCH method - RFC5789 (2/2)

PATCH Example (from RFC6902):

Request:

```
PATCH /my/data HTTP/1.1
Host: www.example.com
Content-Type: application/json-patch+json
Content-Length: 326
If-Match: "abc123"

[
 { "op": "test", "path": "/a/b/c", "value": "foo" },
 { "op": "remove", "path": "/a/b/c" },
 { "op": "add", "path": "/a/b/c", "value": [ "foo", "bar" ] },
 { "op": "replace", "path": "/a/b/c", "value": 42 },
 { "op": "move", "from": "/a/b/c", "path": "/a/b/d" },
 { "op": "copy", "from": "/a/b/d", "path": "/a/b/e" }
]
```

PATCH method on resource /my/data

Specification of patch format in HTTP body (JSON Patch in this example)

Conditional update avoiding conflicting updates from different clients.
If-Match field instructs the server to perform the partial update only if the resource did not change since the client last accessed the resource.

Sequence of patch operations in HTTP request body.
JSON PATCH format (RFC6902) in this example.

Response (successful PATCH):

```
HTTP/1.1 204 No Content
Content-Location: /file.txt
ETag: "e0023aa4f"
```

Return code 204 indicates success

New ETag value may be used in subsequent PATCH requests for conditional updates

7. HTTP 1.0 versus HTTP 1.1

→ HTTP 1.0:

In HTTP 1.0 a new TCP connection for each entity (each page, each picture, each sound file, Java applet etc.) is established. Up to 5 or even 10 connections are open during web server accesses.

☺ Faster display (browser could start to display many entities (objects) at the same time).

☹ Eats up precious server resources and ports (10 open connections per client).

☹ HTTP sessions usually short so TCP seldom gets past slow start phase, thus data transfer is not optimal.

→ HTTP 1.1:

Client can request (and usually does) server to leave TCP connection open (save resources).
Requests (GET) are still self-contained and repeatable.
This allows the client to do pipelining: send multiple requests (for multiple resources) in a row without waiting for the first resource to arrive completely. This makes best use of TCPs flow control (send window opens up) and thus improves performance.

→ Examples 1.0 versus 1.1:

M$ IE Internet Options→Advanced→Use HTTP 1.1
Load http://www.film.com/ with HTTP 1.0 and HTTP 1.1 (clear cache before loading page).

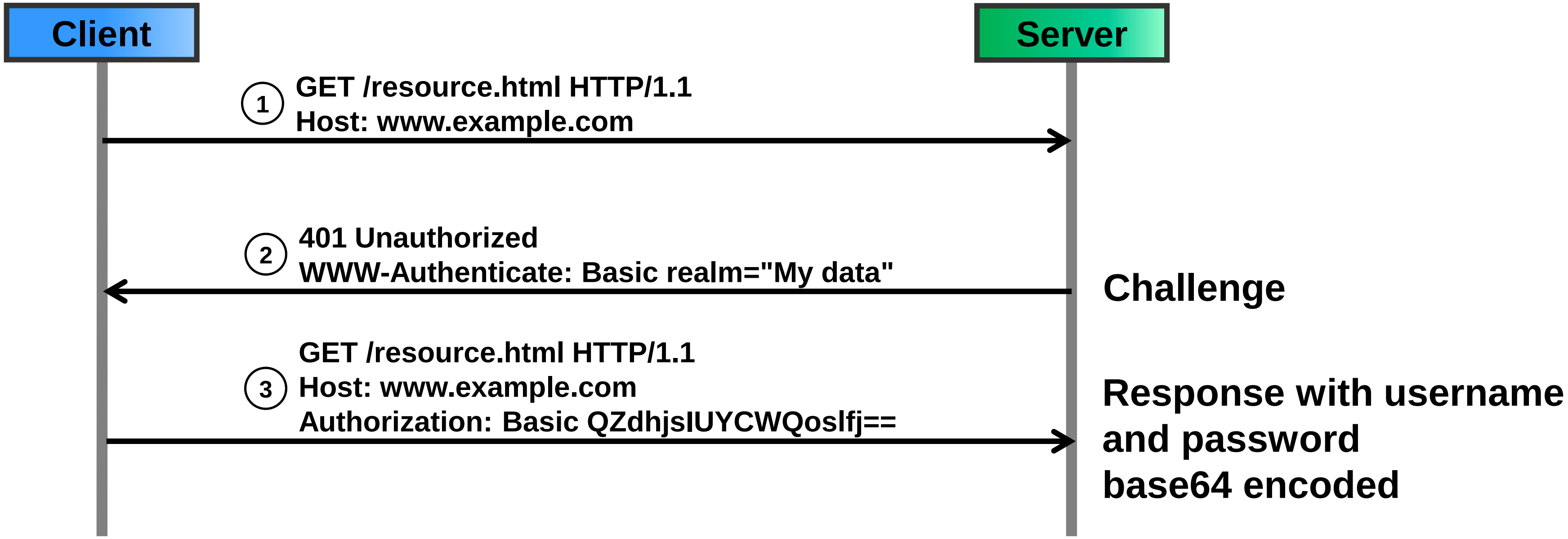
8. HTTP Authentication (1/2)

RFC2617 defines 2 authentication mechanisms (Basic and Digest) based on the Authorization header defined in RFC2616.

The HTTP Authorization header is extensible so any authentication mechanism is possible including proprietary schemes.

a. Basic authentication (RFC2617)

Basic authentication is a challenge-response mechanism that transfers credentials (username and password) in clear text.
For hiding the password HTTPs (TLS) can be used.



```
GET /resource.html HTTP/1.1
Host: www.example.com
```

```
401 Unauthorized
WWW-Authenticate: Basic realm="My data"
```

Challenge

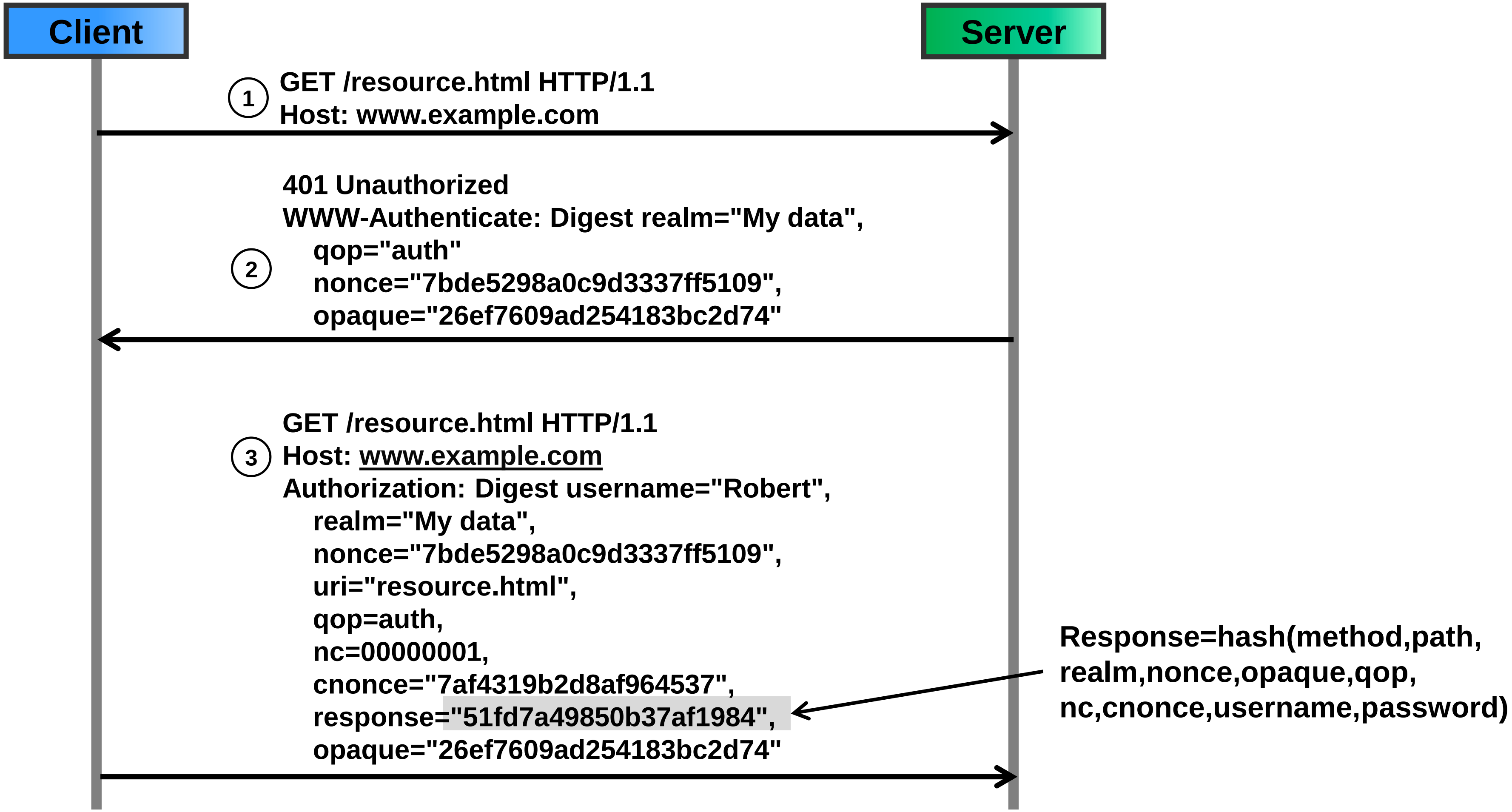
```
GET /resource.html HTTP/1.1
Host: www.example.com
Authorization: Basic QZdhjsIUYCWQoslfj==
```

Response with username and password base64 encoded

8. HTTP Authentication (2/2)

b. Digest access authentication (RFC2617)

Digest access authentication does not require encryption (via HTTPs) to protect the password. The credentials are hashed to prevent eavesdropping.



1

```
GET /resource.html HTTP/1.1
Host: www.example.com
```

2

```
401 Unauthorized
WWW-Authenticate: Digest realm="My data",
   qop="auth"
   nonce="7bde5298a0c9d3337ff5109",
   opaque="26ef7609ad254183bc2d74"
```

3

```
GET /resource.html HTTP/1.1
Host: www.example.com
Authorization: Digest username="Robert",
   realm="My data",
   nonce="7bde5298a0c9d3337ff5109",
   uri="resource.html",
   qop=auth,
   nc=00000001,
   cnonce="7af4319b2d8af964537",
   response="51fd7a49850b37af1984",
   opaque="26ef7609ad254183bc2d74"
```

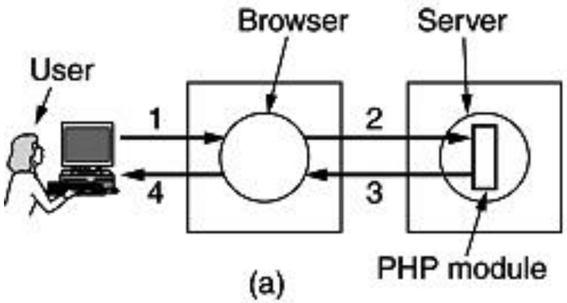
Response=hash(method,path, realm,nonce,opaque,qop, nc,cnonce,username,password)

9. Active web, stateful web (1/20)

➔ Pure HTML pages are static and do not provide ways to generate content dynamically and depending on user input (interactivity). Browser (client side) and server vendors (server side) added features that overcome this lack of interactivity.

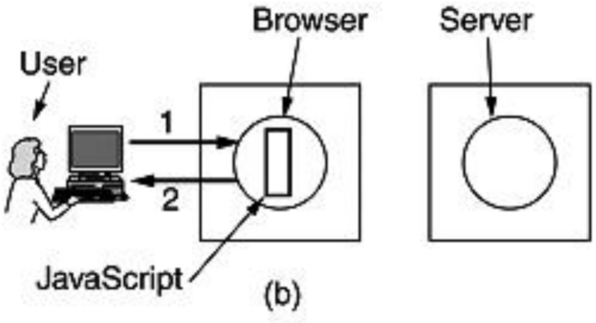
a. Server side technologies:

CGI Common Gateway Interface (scripting)
JSP Java Server Pages (Java code embedded in HTML pages)
Java Servlets (Java code)
ASP Active Server Pages (scripting)
PHP Hypertext Preprocessor (scripting)
SSI Server Side Includes (scripting)
ESI Edge Side Includes (on caching proxy)



b. Client side technologies:

Javascript (scripting)
Java applets (executables)
Plugins (executables)
Helper applications (executables)
Cookies
Hidden form fields



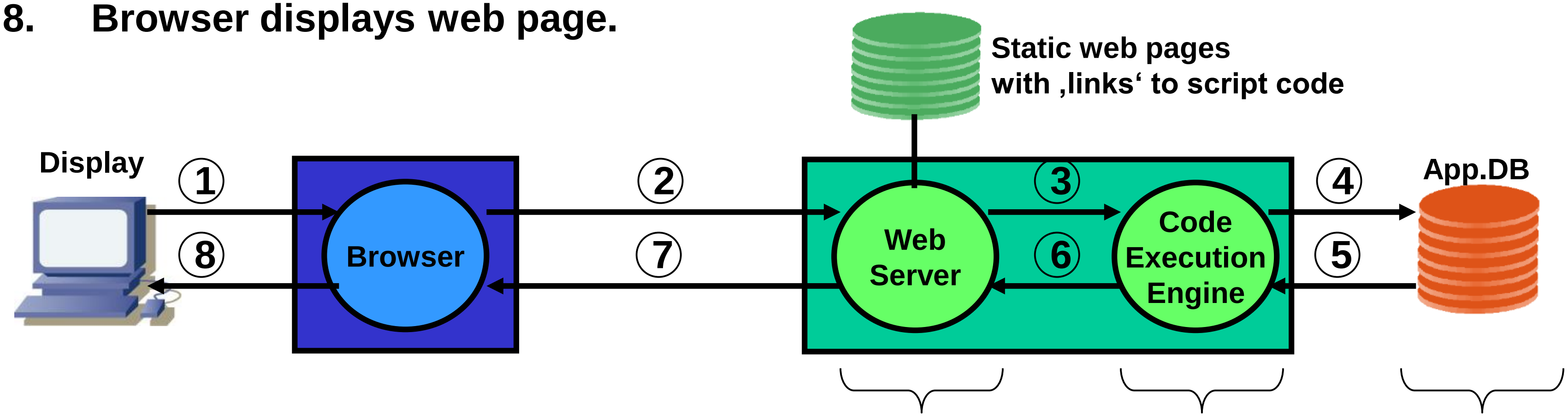
c. Combined client & server side technologies:

AJAX

9. Active web, stateful web (2/20)

➔ General model for server side dynamic page generation:

1. User fills in form (HTML <form>).
2. Browser sends form data as POST or GET request to web server.
3. Web server hands user data and script code that is embedded in HTML to script execution engine.
4. Script executes (database access, call other programs etc.).
5. Data is received from database (or from other application).
6. Script composes web page on-the-fly containing retrieved data.
7. Web server sends back composed page to browser.
8. Browser displays web page.

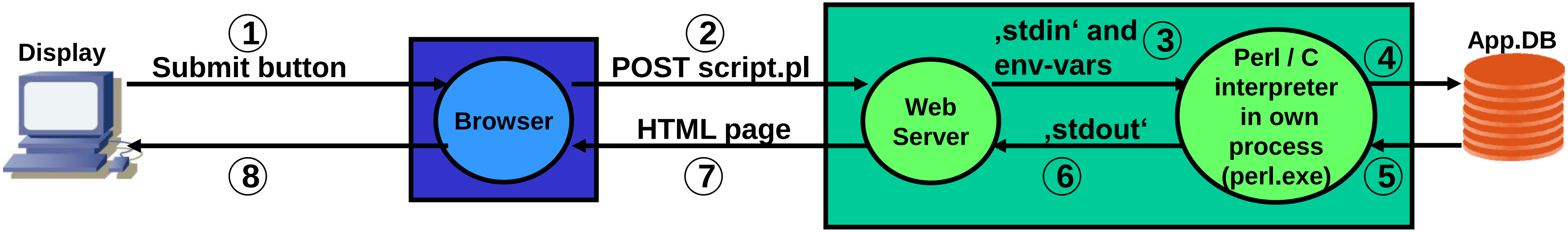


JSP/servlets:	Apache	Tomcat	mySQL/JDBC
ASP:	IIS	IIS script engine	mySQL
CGI:	Apache/IIS	Perl/C interpreter	div.
PHP:	Apache	PHP interpreter	mySQL

9. Active web, stateful web (3/20)

➔ CGI Common Gateway Interface (server side scripting) (1):

CGI is an early and still widely used technology for dynamic web pages.
The Server communicates with Perl or C scripts via environment variables and stdin/stdout.



1. User presses submit button in HTML form page.
2. Browser packs form data into POST (or GET) request and sends it to server.
3. Server sets environment variables, writes form data to server‘s stdout and starts script.
4. Script reads request by reading environment variable $ENV{'QUERY_STRING'} (GET request) or by reading from stdin via read(STDIN,$in,$ENV{'CONTENT_LENGTH'}) (POST/PUT request). The script does what it is supposed to do (database access etc.).
5. Script receives data from database etc.
6. The script writes the result (full HTML page) to stdout.
7. Web server reads from stdin (coupled to stdout of script process) and sends HTML page to the browser.
8. Browser displays newly created HTML page.

Remark: stdin/stdout are standard file handles (always present) for applications to read from keyboard and write to display (JavaSystem.out.println(„hello“)). stdin and stdout are often used to connect processes.

9. Active web, stateful web (4/20)

➔ CGI Common Gateway Interface (2):

Environment variables are used for passing data/parameters between HTTP server and CGI script. Examples of environment variables used by CGI:

QUERY_STRING The information which follows the ? in the URL which referenced this script. This is the query information. It should not be decoded in any fashion. This variable should always be set when there is query information, regardless of command line decoding.

CONTENT_LENGTH The length of the said content as given by the client.

REQUEST_METHOD The method with which the request was made. For HTTP, this is "GET", "HEAD", "POST", etc.

SERVER_PORT The port number to which the request was sent.

PATH_INFO The extra path information, as given by the client. In other words, scripts can be accessed by their virtual pathname, followed by extra information at the end of this path. The extra information is sent as PATH_INFO. This information should be decoded by the server if it comes from a URL before it is passed to the CGI script.

REMOTE_HOST The hostname making the request. If the server does not have this information, it should set REMOTE_ADDR and leave this unset.

REMOTE_ADDR The IP address of the remote host making the request.

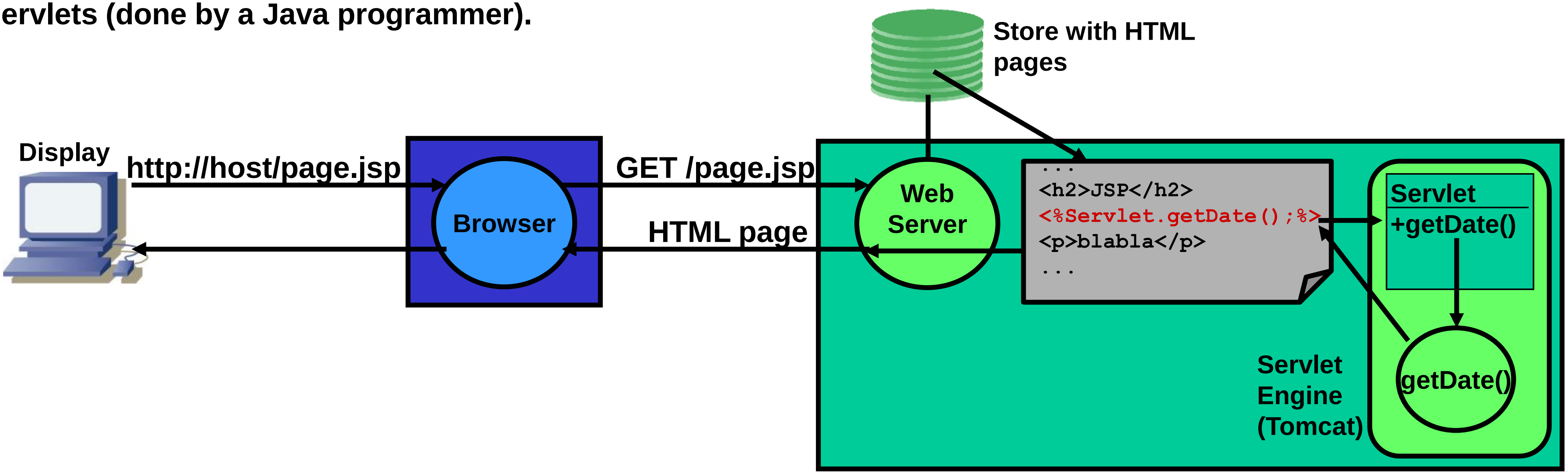
CGI pros/cons:

- ☺ **CGI is very simple.**
- ☹ **CGI is stateless (no support for multistep transactions).**
- ☹ **CGI has high overhead: program/executable invocation for each request (separate process for each HTTP request, even if the scripts to be executed are the same!).**
- ☹ **CGI scripts are executed in standard OS shell (security problems).**

9. Active web, stateful web (5/20)

➔ JSP Java Server Pages (1):

The JSP page functions as servlet ‚front-end' (static HTML code with ‚entry points' into Java servlet code). In principle all HTML formating can be packed into Java code (pure servlet). It is at the discretion of the designer to split the functionality into static HTML pages (JSP) and dynamic servlet code. Naturally the static part will be implemented as HTML pages (done by web designer) and the dynamic part implemented as servlets (done by a Java programmer).



JSPs are an alternative to CGI. JSP is similar to CGI (see above) but the capabilities of JSP are much more powerful (basically all funtionality that Java offers). Akin to CGI JSP allows to mix static HTML code with dynamically generated HTML code (generated by Java code). But JSP allows to separate the HTML code (format) from the content (data) in a cleaner way than CGI does.

JSP is also more efficient than CGI since the code is executed in a thread instead of a separate process (less overhead).

9. Active web, stateful web (6/20)

➔ JSP Java Server Pages (2):

JSP code in HTML pages is enclosed in <% ...%> tags.

Simple JSP example (JSP code marked *red*):

```
<html>
  <head>
  <meta name="author" content="pegli">
  <title>
  </title>
  </head>
  <body bgcolor=#00ffff>
  <% out.println("<h2>Client IP:port is:</h2>"); %>
  <h1><%= request.getRemoteHost() + ":" +
          request.getRemotePort() %>
  </h1>
  </body>
</html>
```

HTML page as it is sent to browser:

```
<html>
  <head>
  <meta name="author" content="pegli">
  <title>
  </title>
  </head>
  <body bgcolor=#00ffff>
  <h2>Client IP:port is:</h2>
  <h1>127.0.0.1:4499</h1>


  </body>
</html>
```

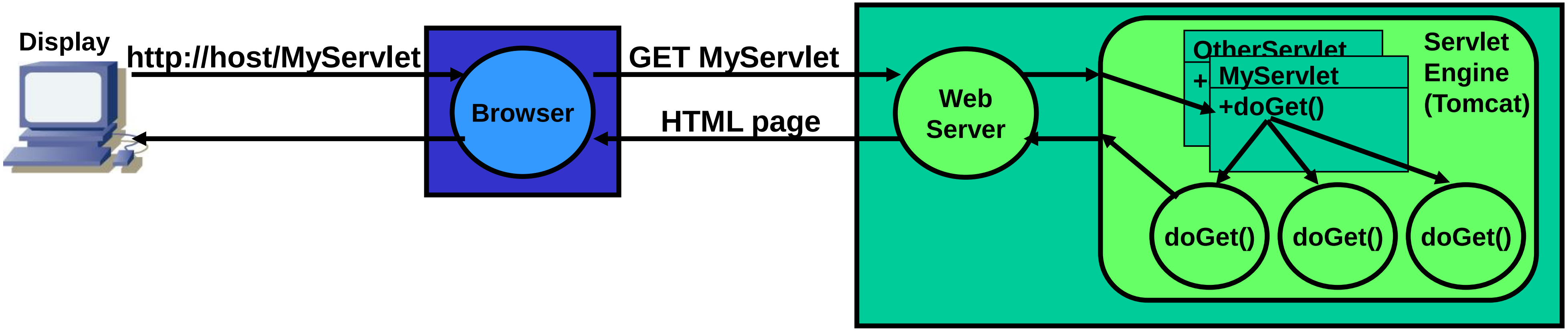
9. Active web, stateful web (7/20)

➔ Java Servlets (1):

The Java servlet engine (servlet runtime environment) provides the following facilities:

1. Parse and decode HTML form data.
2. Reading and setting HTML headers.
3. Handling cookies.
4. Tracking sessions.

N.B.: Java Servlets are not server side scripts but must be compiled (javac) before they can be called.



The servlet engine (e.g. Tomcat) instantiates each servlet class once but creates a new temporary thread for each new HTTP GET request; the thread executes the doGet() method that each servlet class must implement (the servlet extends HttpServlet class and has to implement the doGet() method). Once the GET request has been serviced the doGet() method completes and the thread is terminated (‚run to completion‘). The HTTP request (input argument) and HTTP response (output argument) are passed to the servlet as objects: `doGet(HttpServletRequest request, HttpServletResponse response).`

9. Active web, stateful web (8/20)

➔ Java Servlets (2):

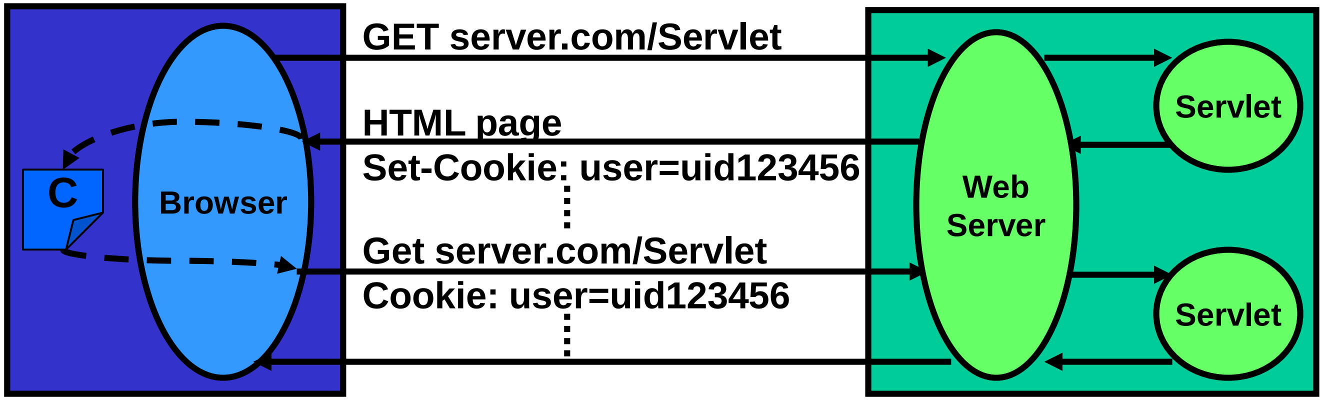
HTTP is stateless, i.e. there is no session (there is only an underlying TCP connection that lives for some seconds and then goes away). A session represents the current status of the relationship between client and web server including the current state of user data (username etc.). How can a session be controlled for pages like shopping sites (‚shopping cart‘) where the web server needs to remember what the user placed in the shopping cart?

Session tracking solution 1: Use cookies:

Servlet can place a cookie on the client (browser) via Cookie class.

```
Cookie userCookie = new Cookie(„user“,“uid123456“); //add parameter user=uid123456
response.addCookie(userCookie);                     //add cookie to HTTP response header
```

The next time the user accesses the same page the browser sends the cookie back to the server (HTTP header field `Cookie:`).



```
Cookie c = new Cookie(„user“,
                      “uid1234“);
response.setCookie(c);
```

```
Cookie c[] = request.getCookies();
String s =
ServletUtilities.getCookieValue(
    c,“user“,NO_VALUE);
```

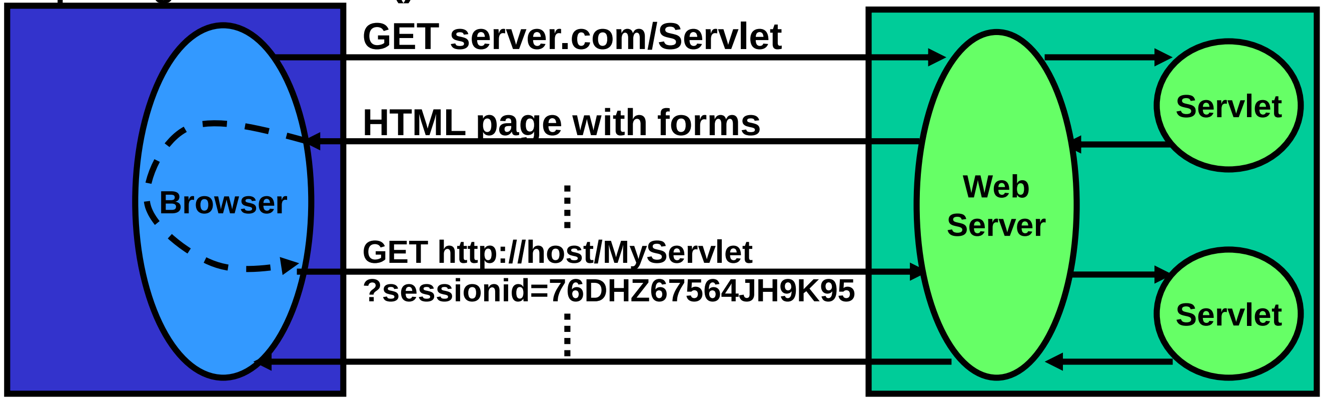
9. Active web, stateful web (9/20)

➔ Java Servlets (3):

Session tracking solution 2: URL rewriting:
URL rewriting appends the session tracking information to the URL using GET-style encoding or extra path.

a. GET-style encoding:

Servlet creates pages with links (URLs) and appends parameter/value pairs to the URLs as follows:
http://host/MyServlet?sessionid=76DHZ67564JH9K95
The servlet engine extracts the parameters and makes them available to the servlet via the request.getParameter() method.



```
String val =
request.getParameter(
„sessionid“);
```

b. Extra path information:

Session IDs (and other parameters) can also be appended to the URL in so-called extra path notation:
`http://host/MyServlet/sessionid/76DHZ67564JH9K95`
The servlet engine still invokes MyServlet but makes the extra path (`sessionid/76DHZ67564JH9K95`) available through the request.getExtraPath() method.

9. Active web, stateful web (10/20)

➔ ASP Active Server Pages:

ASP is Microsofts technology for dynamic web pages. The ASP model is very similar to the JSP model.

ASP uses Visual Basic or JavaScript (Microsofts JScript = ECMA262 standard) as script language. Script code is embedded into HTML pages with <%...%> tags (like JSP).

ASP runs on IIS (Internet Information Server, Microsofts web server).

Simple ASP example (ASP code marked *red*):

```
<%@ Language=VBScript %>
<html>
  <head>
    <title>Example ASP page</title>
  </head>
  <body>
    <%FirstVar = "Hello world!"%>
    The time is: <%=time%> <BR>
    <%FOR i=1 TO 10%>
    <%=FirstVar%><BR>
    <%NEXT%>
  </body>
</html>
```

HTML page as it is sent to browser:

```
<html>
  <head>
    <title>Example ASP page</title>
  </head>
  <body>

    The time is: 13:30:07 <BR>
    Hello World
    ....
    Hello World
  </body>
</html>
```

9. Active web, stateful web (11/20)

➔ PHP (PHP: Hypertext Preprocessor) (server side scripting):

PHP is a script language executed on server by script module. Script code is embedded into HTML pages and executed on-the-fly before the page is delivered to the client. The client (browser) does not ‚see' PHP code since the code is executed on the server and its output embedded into the HTML page.
PHP code is enclosed in <? ...?> tags (or alternatively in <?php...> tags).

Simple PHP example (PHP code marked *red*):

```
<?php
  $title="My first PHP script."
?>
<html>
  <head>
  <meta name="author" content="pegli">
  <?php include("lang/en.php");?>
  <title>
    <?=$TEXT['global-xampp'];?>
    <?php include('.version');?>
  </title>
  </head>
  <body bgcolor=#ffffff>
  <h1><?php echo($title)?></h1>
  <h2>Your browser is:</h2>
  <?php echo $HTTP_USER_AGENT;?>
  </body>
</html>
```

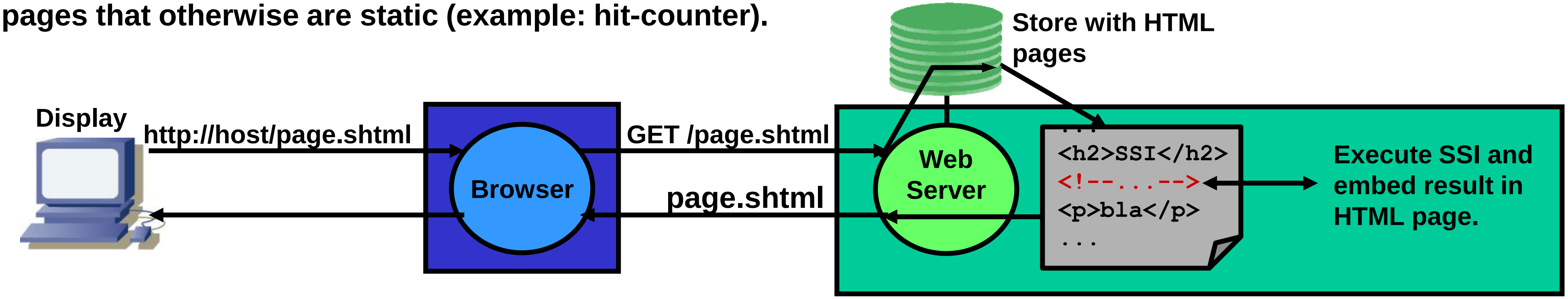
HTML page as it is sent to browser:

```
<html>
  <head>
  <meta name="author" content="pegli">

  <title>
    XAMPP for Windows
    Version 1.4.2
  </title>
  </head>
  <body bgcolor=#ffffff>
  <h1>My first PHP script.</h1>
  <h2>Your browser is:</h2>
  Mozilla/4.0 (compatible; MSIE 6.0;...
</body>
</html>
```

9. Active web, stateful web (12/20)

➔ SSI Server Side Includes (1):

SSI are directives (commands) placed inside HTML. These directives are executed on the server when the page(s) are served. SSI is better suited for adding small pieces of dynamically generated information to pages that otherwise are static (example: hit-counter).



```
...
<h2>SSI</h2>
<!--...-->
<p>bla</p>
...
```

SSI HTML pages usually have a suffix .shtml. This tells the web server (e.g. Apache) to inspect the page and execute the SSI directives.

SSI directives syntax (uses SGML comment syntax):
<!--#element attribute=value attribute=value ... -->

SSI directives (examples):

<!--#echo var="DATE_LOCAL" -->	Today‘s date.
<!--#flastmod file="index.html" -->	Include date of last modification of page.
<!--#include virtual="/cgi-bin/counter.pl" -->	Include hit-counter.
<!--#include virtual="/footer.html" -->	Include standard footer.
<!--#exec cmd="ls" -->	Execute a command such as ls (directory listing; use with utmost care!).
<!--#set var="name" value="Rich" -->	Setting a variable for later use.

9. Active web, stateful web (13/20)

➔ SSI Server Side Includes (2):

SSI directives HTML pages are enclosed in <!-- ...--> tags.

Simple SSI example (SSI code marked *red*):

```
<html>
  <head>
  <meta name="author" content="pegli">
  <title>
  </title>
  </head>
  <body bgcolor=#00f080>
  <h1>SSI SSI Example Page</h1>
  <h2>
    <!--#config timefmt="%A %B %d, %Y" -->
    Today is <!--#echo var="DATE_LOCAL" -->
  </h2>
  </body>
</html>
```

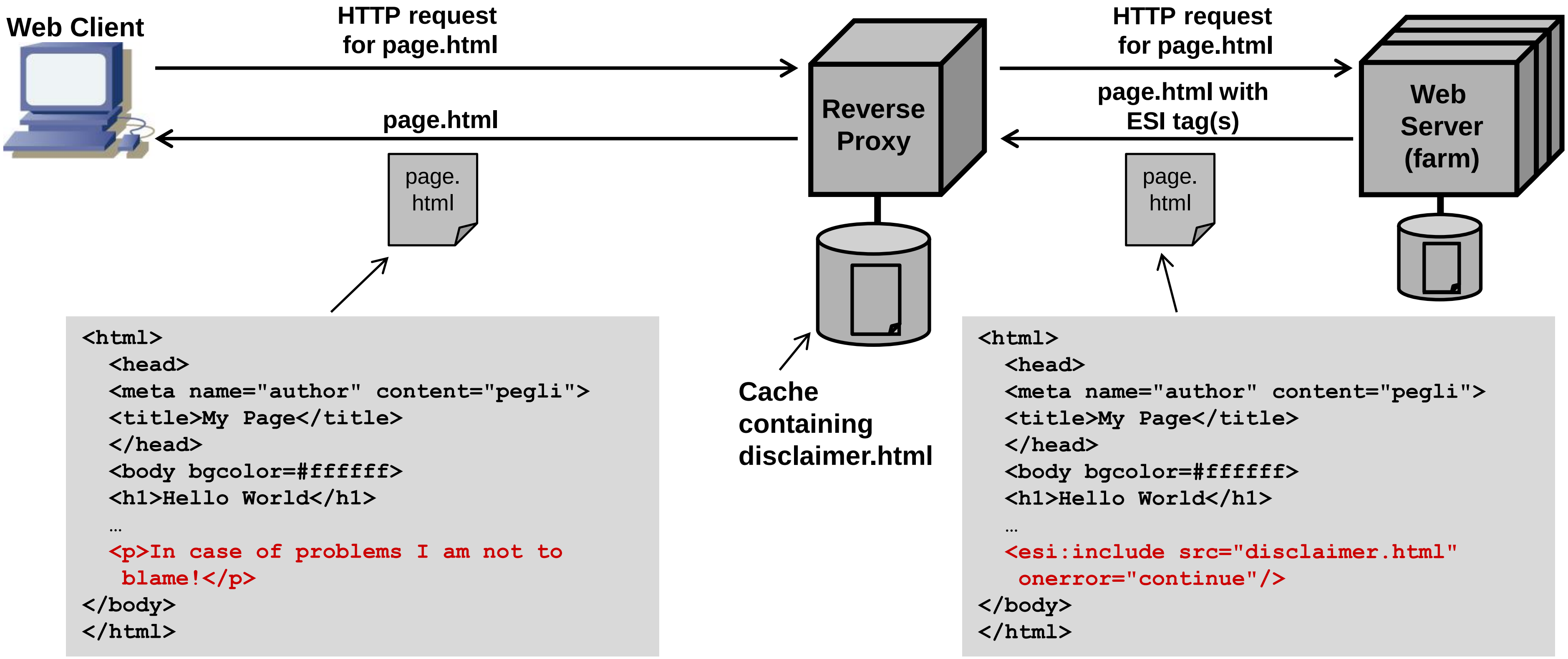
HTML page as it is sent to browser:

```
<html>
  <head>
  <meta name="author" content="pegli">
  <title>
  </title>
  </head>
  <body bgcolor=#00f080>
  <h1>SSI SSI Example Page</h1>
  <h2>
    Today is Tuesday November 16, 2004

  </h2>
  </body>
</html>
```

9. Active web, stateful web (14/20)

➔ ESI Edge Side Includes:

Web pages often contain static content (cacheable) and dynamic content (non-cacheable). In ESI, a proxy (e.g. reverse proxy, see below) performs the processing of ESI tags in HTML pages. The web server creates the web page including dynamic content but leaves ESI tags untouched. These are processed by an ESI processor on the reverse proxy (the content addressed by ESI tag is retrieved from a cache).


Web Client
HTTP request for page.html
page.html
page. html
Reverse Proxy
HTTP request for page.html
page.html with ESI tag(s)
page. html
Web Server (farm)
Cache containing disclaimer.html

```
<html>
  <head>
  <meta name="author" content="pegli">
  <title>My Page</title>
  </head>
  <body bgcolor=#ffffff>
  <h1>Hello World</h1>
  …
  <p>In case of problems I am not to
   blame!</p>
</body>
</html>
```

```
<html>
  <head>
  <meta name="author" content="pegli">
  <title>My Page</title>
  </head>
  <body bgcolor=#ffffff>
  <h1>Hello World</h1>
  …
  <esi:include src="disclaimer.html"
   onerror="continue"/>
</body>
</html>
```

9. Active web, stateful web (15/20)

➔ JavaScript (Client side scripting):

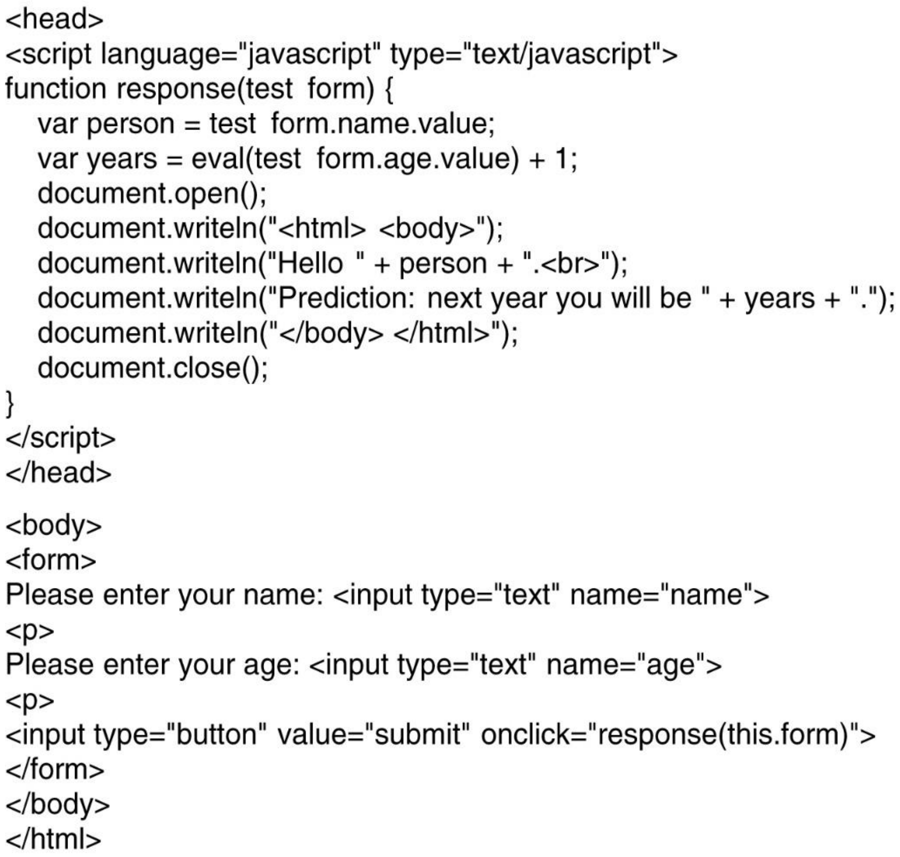
JavaScript is a script language executed on the client by web browser.
JavaScript is embedded into HTML page with <SCRIPT></SCRIPT> tags.
JavaScript has many high-level programming features like:

- **variables (types boolean, numeric, string)**
- **arithmetic, logical and bitwise operators**
- **for() and while() control loops**
- **functions**

JavaScript has limited access to the machine it is running on (security restrictions).
Usage of static HTML pages with embedded Javascript along with style sheets (CSS) is called DHTML (Dynamic HTML).

N.B.: JavaScript has nothing to do with Java, i.e. it‘s not a stripped down Java version!

Page with JavaScript example:



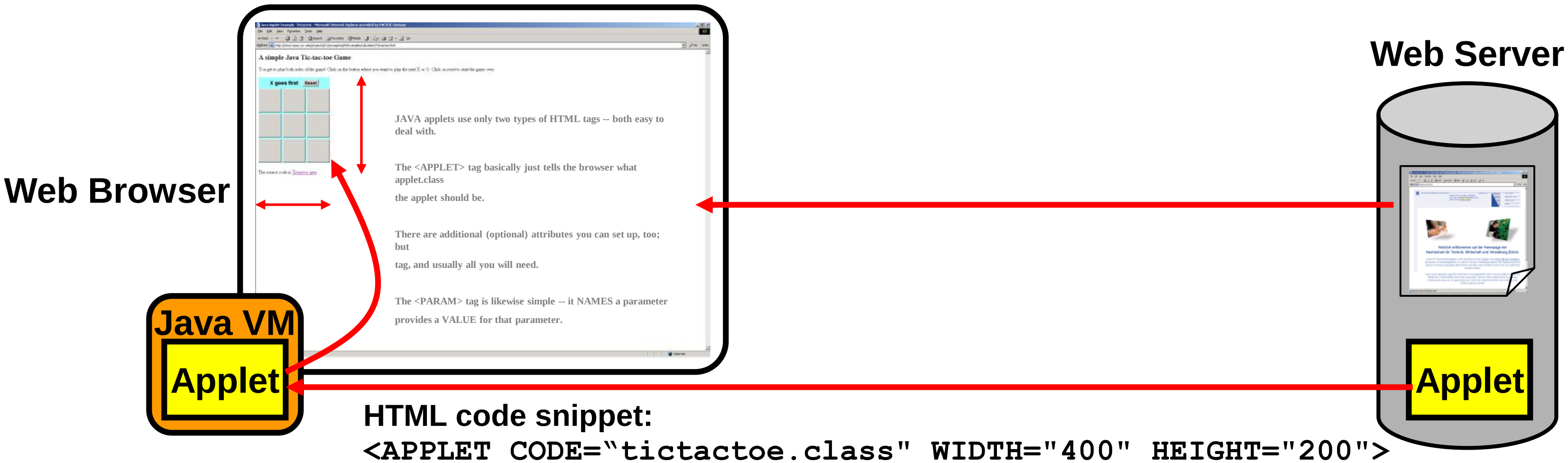
```
<head>
<script language="javascript" type="text/javascript">
function response(test  form) {
    var person = test  form.name.value;
    var years = eval(test  form.age.value) + 1;
    document.open();
    document.writeln("<html>  <body>");
    document.writeln("Hello  " + person + ".<br>");
    document.writeln("Prediction:  next year you will be " + years + ".");
    document.writeln("</body> </html>");
    document.close();
}
</script>
</head>

<body>
<form>
Please enter your name: <input type="text" name="name">
<p>
Please enter your age: <input type="text" name="age">
<p>
<input type="button" value="submit" onclick="response(this.form)">
</form>
</body>
</html>
```

9. Active web, stateful web (16/20)

➔ Java applets (client side executable code):

Java applets are executed in client web browser context (Java Runtime Environment as browser plugin). The browser hands over control of a specific area in web page to applet (applet produces graphical output).

The Java applet can access the outside world (file system, network) only when user grants according access rights (security). Network access is restricted to the server from which the applet was loaded.



A simple Java Tic-tac-toe Game

You get to play both sides of the game! Click on the button where you want to play the next X or O. Click on reset to start the game over.

The source code is Tictactoe.java

JAVA applets use only two types of HTML tags -- both easy to deal with.

The <APPLET> tag basically just tells the browser what applet.class

the applet should be.

There are additional (optional) attributes you can set up, too; but

tag, and usually all you will need.

The <PARAM> tag is likewise simple -- it NAMES a parameter

provides a VALUE for that parameter.

HTML code snippet:

```
<APPLET CODE=“tictactoe.class" WIDTH="400" HEIGHT="200">
<PARAM NAME="SPEED" VALUE="100">
<PARAM NAME="IMAGE1" VALUE="thisimage.gif">
<PARAM NAME="IMAGE2" VALUE="thatimage.jpg">
</APPLET>
```

Due to security issues with Java in the browser, applets should not be used anymore.

9. Active web, stateful web (17/20)

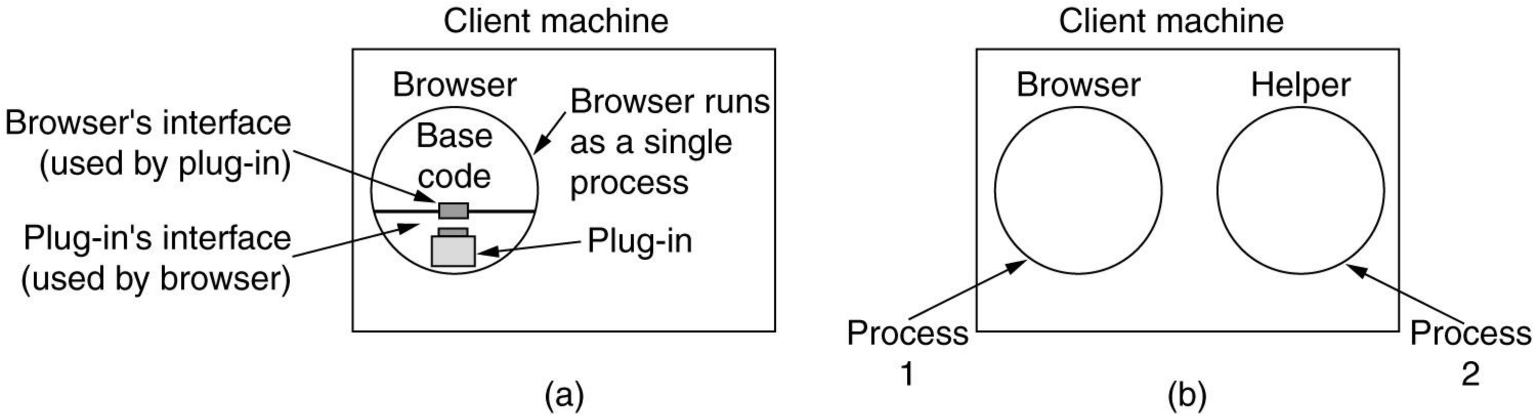
➔ Client side plugins and helper applications:

a. Plugins:

Plugins are small applications linked into browser at run-time (e.g. Flash). They run in the browser‘s process.

b. Helper applications:

Helper applications are standalone applications that are started by browser on request, e.g. audio / video streamer. Helper applications run in their own process and are able to run without browser.


Client machine
Browser
Browser's interface (used by plug-in)
Base code
Browser runs as a single process
Plug-in's interface (used by browser)
Plug-in
(a)
Client machine
Browser
Helper
Process 1
Process 2
(b)

9. Active web, stateful web (18/20)

➔ Cookies (RFC2109):

HTML header field „Set-cookie:“ instructs browser to store information on source and content.
When browser visits the web page the next time it sends an HTML header with a „Cookie:“ field with the value = stored cookie.

Cookies are not programs but only static data that is stored/sent along with web pages.

Session cookie: Exists as long as surfing session (e.g. web shopping session). Stores info about items in the shopping cart. Session cookies go away as soon as the web session is closed (browser closed).

Persistent cookie: Used to recognize a user, e.g. for remembering the username and password (cookies with expiration date are persistent).

☺ Supports any type (non-text too).

☺ Scales better since the job of storing information is offloaded to the client.

➔ Hidden fields:

Hidden fields are HTML form fields tagged with the HIDDEN attribute.
They allow to shuffle back and forth (invisibly) embedded state information (without cookies).

☺ Simple, no cookies required.

☹ Applicable only to text/html type.

Example: http://www.htmlcodetutorial.com/forms/_INPUT_TYPE_HIDDEN.html

9. Active web, stateful web (19/20)

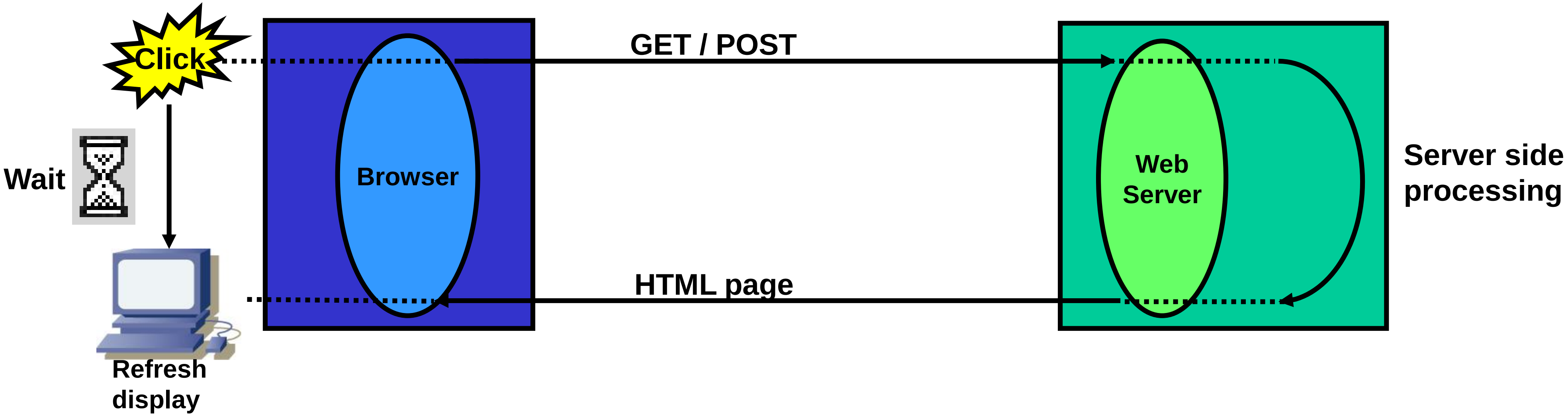
➔ AJAX – Asynchronous Javascript + XML (1):

Traditional web application model:

a. „Click, wait and refresh“: Web client (browser) requests a page (GET, POST), waits for the response (HTML page) and eventually displays the new page.

b. Synchronous requests: It‘s always the client (browser) initiating the request while the server merely responds to such requests (one-way requests and responses).

This makes the protocol simple but difficult to use for interactive applications.



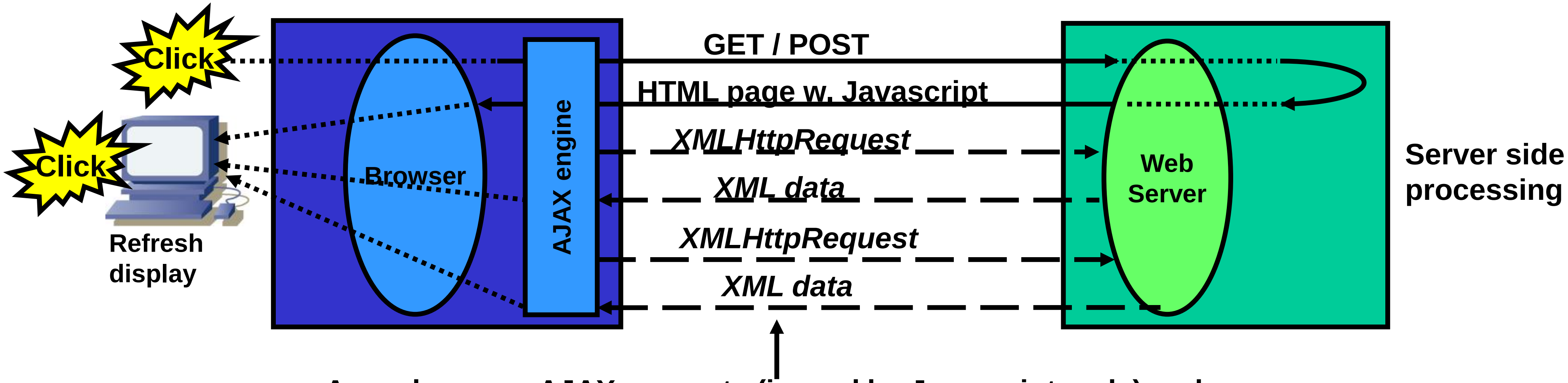
9. Active web, stateful web (20/20)

➔ AJAX – Asynchronous Javascript + XML (2):

AJAX model:

a. Partial UI updates: The requested HTML page contains Javascript code that can load data (XML encoded) from the server asynchronously (independent of user actions). The Javascript code uses an additional component (AJAX engine) in order to access the server (AJAX engine basically adds XMLHttpRequest object to Javascript).

b. Asynchronous updates: The Javascript may access the server asynchronously thus decoupling user interaction from server interaction. The user can continue to use the GUI/application while the Javascript accesses the server in the background. Upon reception of new information only the affected part of the GUI is updated.



Asynchronous AJAX requests (issued by Javascript code) and responses with XML encoded data.

Demo: http://www.pushlets.com/pushlet/examples/ajax/ajax-1.html

10. Web caching (1/6)

➔ Web caching purpose and web caching hierarchy:

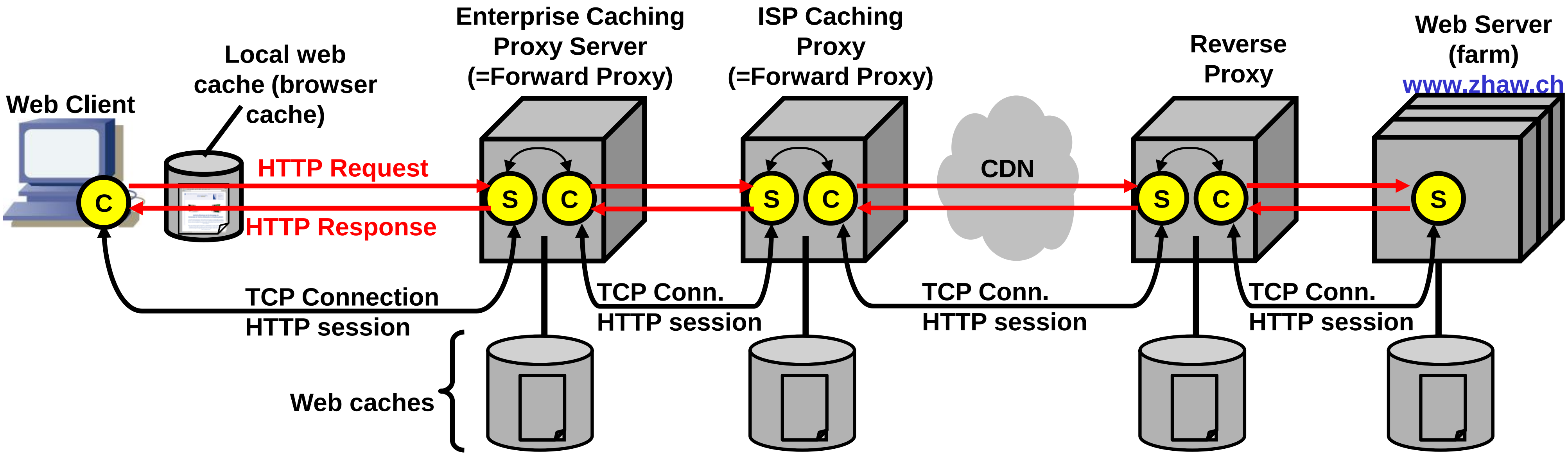
Web caching means that the client or caching server (proxy) retrieves the requested page from a local storage (if present) in order to:
1. Reduce latency (increase responsiveness)
2. Reduce traffic (traffic source is closer to traffic sink = client)

➔Companies (enterprises) set up proxies for controlling access to the network (ban / allow sites) and may combine it with a cache.

➔ ISPs (Internet Service Providers) set up proxy caches to reduce traffic (main objective). Hit rate (page retrieval from cache instead source server) can reach 50%!

➔ Web site hosters make use of CDNs (Content Delivery Networks) to deliver content (web pages) faster from locations closer to the requesting client (using anycast routing, DNS-based request routing, HTML rewriting etc.).

➔ At the final web server location reverse proxies distribute the load over a battery of web servers (load balancing).



10. Web caching (2/6)

➔ How it works:

Cache validators in the HTML header specifiy if and how a page is to be cached (header fields `Cache-Control, Expires, Last-Modified, ETag`).

Every web object (or element) such as pages, images etc. is cached or not cached according to the following rules:

1. If HTTP header tell cache not to keep the object, the cache won‘t (‚no-cache‘).
2. If no validator is present the cache will mark the object as uncacheable.
3. A cached object is considered fresh (able to be sent to the client without checking with the origin server) if:

- **It has an expiry time or other age-controlling directive set, and is still within the fresh-period.**
- **If a browser cache has already seen the object, and has been set to check once a session.**
- **If a proxy cache has seen the object recently, and it was modified relatively long ago.**

4. If an object is stale, the origin server will be asked to validate the object, or tell the cache whether the copy that it has is still good (validation).

➔A fresh object will be sent immediately to the browser.
➔A validated object will avoid sending the object again.

10. Web caching (3/6)

➔ How to control web caches:

1. Use HTML meta tags:

```
<META http-equiv="Pragma" content="no-cache">
```

Problem: such meta tags are often only honored by browser caches, but not by proxy caches.

2. Use ‚Expires' HTTP header field:

```
HTTP/1.1 200 OK
Date: Fri, 30 Oct 1998 13:19:41 GMT
Server: Apache/1.3.3 (Unix)
Cache-Control: max-age=3600, must-revalidate
Expires: Fri, 30 Oct 1998 14:19:41 GMT
Content-Length: 1040
Content-Type: text/html
```

E.g. Expires field can be set to the time when usually updates are made to web pages.

Problem: ‚Expires' field requires client and server to have same absolute time base (date, time). Sometimes clients do not have correct absolute time (GMT Greenwich Mean Time), e.g. small network appliances. Additionally time zone corrections and summertime make usage of this field more difficult.

10. Web caching (4/6)

➔ How to control web caches (cont‘d):

3. Use ‚Cache-Control‘ HTTP header field with ‚max-age‘:

```
HTTP/1.1 200 OK
Date: Fri, 30 Oct 1998 13:19:41 GMT
Server: Apache/1.3.3 (Unix)
Cache-Control: max-age=3600, must-revalidate
Content-Length: 1040
Content-Type: text/html
```

‚max-age‘=[seconds] specifies the maximum amount of time that an object will be considered fresh. Similar to Expires, this directive allows more flexibility. [seconds] is the number of seconds from the time of the request you wish the object to be fresh for.

‚s-maxage‘=[seconds] is similar to max-age, except that it only applies to proxy (shared) caches.

‚public‘ marks the response as cacheable, even if it would normally be uncacheable. For instance, if the pages are authenticated, the public directive makes them cacheable.

‚no-cache‘ forces caches (both proxy and browser) to submit the request to the origin server for validation before releasing a cached copy, every time. This is useful to assure that authentication is respected (in combination with public), or to maintain rigid object freshness, without sacrificing all of the benefits of caching.

‚must-revalidate‘ tells caches that they must obey any freshness information you give them about an object. The HTTP allows caches to take liberties with the freshness of objects; by specifying this header, you're telling the cache that you want it to strictly follow your rules.

‚proxy-revalidate‘ similar to ‚must-revalidate‘, except that it only applies to proxy caches.

10. Web caching (5/6)

➔ How to control web caches (cont'd):

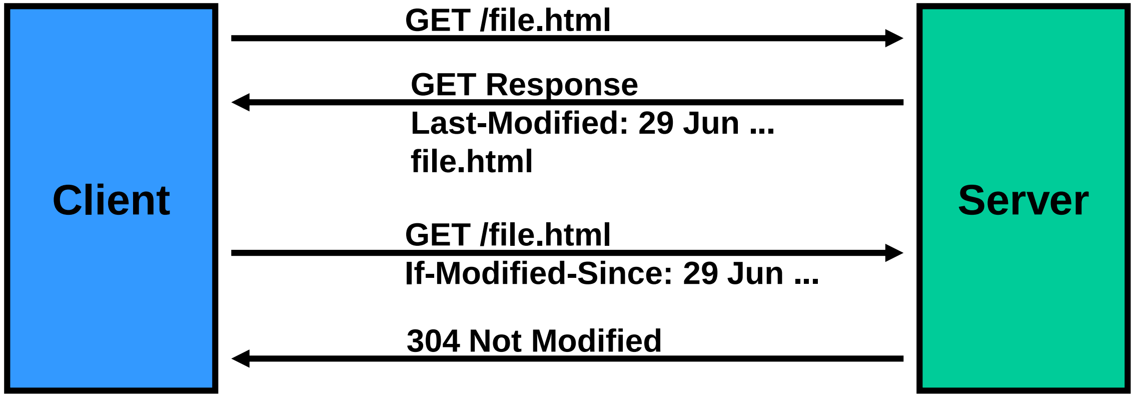
4. Use validators/validation:

In response header:

`Last-Modified: Mon, 29 Jun 1998 02:28:12 GMT`

In new request header:

`If-Modified-Since: Mon, 29 Jun 1998 02:28:12 GMT`



If present the „Last-Modified" header field tells when the object (file, page) was last modified. This header field is stored in the cache along with the page / object it belongs to.

When the cache has an object with Last-Modified header it can use it to ask the server if the object has changed since the last time it was seen with a If-Modified-Since header field.

If the object has not changed the server will not return the object and the client will retrieve the object from the cache.

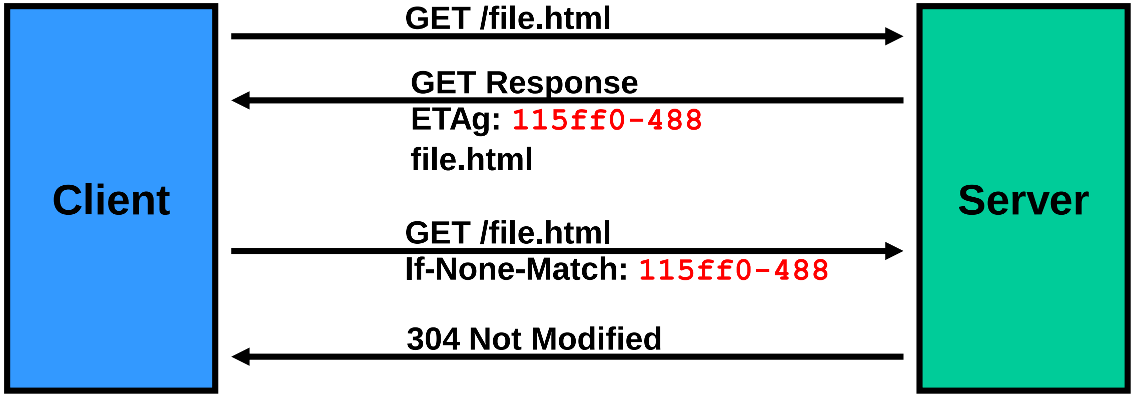
10. Web caching (6/6)

➔ How to control web caches (cont‘d):

5. Use ETag validators/validation:

ETag = simple hash value of requested page; if a page changes its hash value changes too.

```
In response header:
ETag: 115ff0-488

In new request header:
If-None-Match: 115ff0-488
```



ETag is a unique tag generated each time an object is changed (or created). The server performs only a full match on ETags.

Cache directive conflict: in case of conflict of multiple cache control directives the most restrictive is taken, i.e. the one that is most likely to preserve semantic transparency.

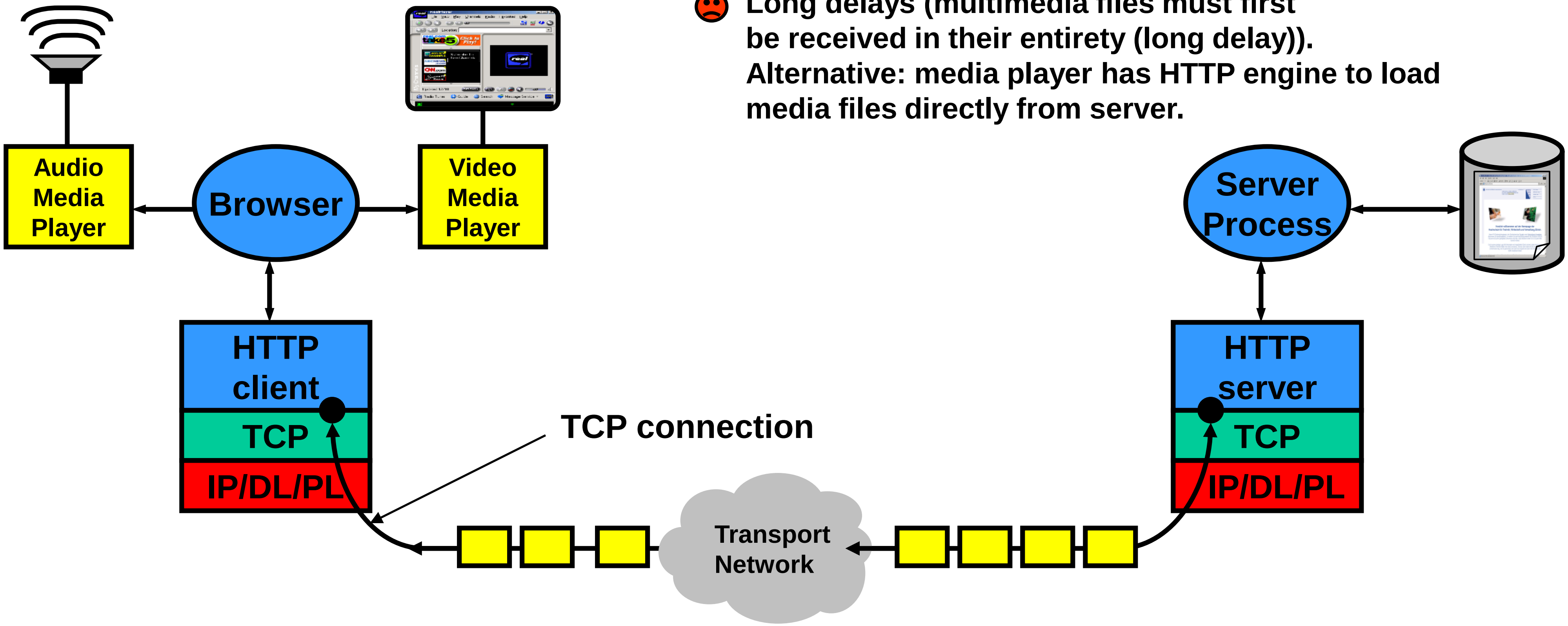
11. Web audio/video streaming (1/3)

A. Conventional design:

1. Web browser requests HTML page with MP3 content (Content-Type: audio/mp3).
2. Web browser receives entire audio/video file.
3. Web browser starts helper app. (MP3 player) and starts passing video packets/frames to it.
4. Audio player decompresses the frames and plays them back.

☺ Simple design.

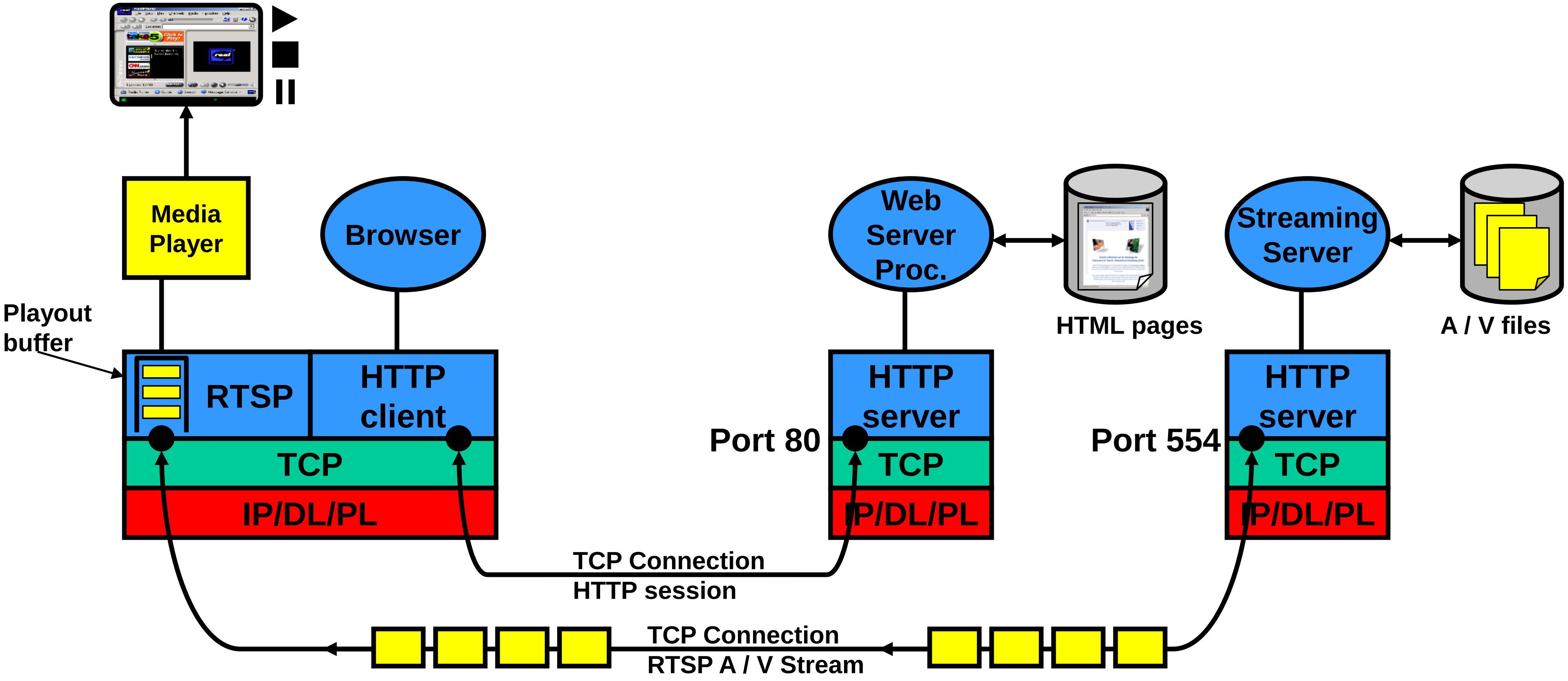
☹ Long delays (multimedia files must first be received in their entirety (long delay)).
Alternative: media player has HTTP engine to load media files directly from server.



11. Web audio/video streaming (2/3)

B. Using Streaming Server with RTSP RFC2326 (1):

The browser starts the media player and passes it a URL from where to stream media. The media player then opens an RTSP session to the media server, starts streaming (download) the media file and plays the content back (playback while streaming).



11. Web audio/video streaming (3/3)

B. Using Streaming Server with RTSP RFC2326 (2):

1. User clicks on video/movie/audio link.
2. Browser sends HTTP GET request for video/movie/audio file to web server.
3. Web server responds with a meta file containing information on requested file (media type).
4. The browser determines from Content-type: field in the meta file the media player (helper application) to invoke and passes the contents of the meta file to it (e.g. RealPlayer).
5. The media player reads the URL from the meta file (streaming server) and sends an RTSP SETUP message to the streaming server (start a new streaming session).
6. After some negotiation the streaming server responds with an RTSP 200 OK message.
7. The server starts sending audio / video frames encapsulated in RTSP.
8. The media player first fills its playout buffer until it is sufficiently full to prevent playout underruns.
9. Media player starts playing out audio / video from playout buffer.
10. Audio / Video stream is controlled by media player with RTSP COMMANDS:

RTSP PLAY	instruct server to start sending audio / video (or resume sending)
RTSP PAUSE	instruct server to (temporarily) stop sending audio / video stream
RTSP TEARDOWN	stop RTSP session