Internet Technologies
4-http

F. Ricci
2010/2011
Content

- Hypertext Transfer Protocol
- Structure of a message
- Methods
- Headers
- Parameters and character encoding
- Proxy
- Caching
- HTTP 1.1: chunked transfer and persistent connection
- State management
- Cookies
Finally! We’re here.
Contact a Daemon Using Telnet

- First type in: `telnet www.google.it 80`
- Then the following: `GET index.html HTTP/1.0`
- Then two `CRLF`
- You will get the reply from the http daemon
HTTP

- HTTP stands for **Hypertext Transfer Protocol**
- HTTP (network protocol) delivers virtually all files and other data (**resources**) on the World Wide Web - HTML files, image files, query results, ...
- HTTP is used to transmit **resources**, i.e., some chunk of information that can be identified by a **URL**
- HTTP uses the client-server stateless model
  - An **HTTP client** opens a connection and sends a request message to an **HTTP server**;
  - the **HTTP server** then returns a response message (containing the requested resource).
Client and Server

- User uses HTTP client (Web Browser)
- It enters a URL (e.g. http://www.yahoo.com/)
- Makes a request to the server
- client initiates TCP connection (creates socket) to server, port 80
- server accepts TCP connection from client
- HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
- TCP connection closed
HTTP Server

- **Listens** on port 80 (usually)
- Handles HTTP requests
- Sends back responses
- Document root is a directory in the file system
- Server maps path to file system file
- URL Path “/” maps to Document Root
- Let’s say Document Root is C:\htdocs\`

  / => C:\htdocs\
  /images/ => C:\htdocs\images\
  /a/X.html => C:\htdocs\a\X.html
Example of Request

View HTTP Request and Response Header

HTTP(S)-URL: http://www.unibz.it

HTTP version: □ HTTP/1.1 □ HTTP/1.0 (with Host header) □ HTTP/1.0 (without Host header)

☐ Raw HTML view ☑ Accept-Encoding: gzip • Request type: ☑ GET □ POST □ HEAD □ TRACE

User agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.7) Gecko/20070914 Firefox/2.0.0.7 Web-Sniffer/1.0

HTTP Request Header

Connect to 193.206.186.140 on port 80 ... ok

GET / HTTP/1.1
Host: www.unibz.it
Connection: close
Accept-Encoding: gzip
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8
Accept-Language: en-us, en;q=0.5
Accept-Charset: ISO-8859-1, utf-8;q=0.7,*;q=0.7
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.7) Gecko/20070914 Firefox/2.0.0.7 Web-Sniffer/1.0
Referer: http://web-sniffer.net/
Example of Response

HTTP Response Header

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Status Code:</td>
<td>HTTP/1.1 200 OK</td>
</tr>
<tr>
<td>Date:</td>
<td>Wed, 26 Sep 2007 09:34 51 GMT</td>
</tr>
<tr>
<td>Server:</td>
<td>Apache/2.0.49 (Linux/SuSE)</td>
</tr>
<tr>
<td>Cache-Control:</td>
<td>no-cache</td>
</tr>
<tr>
<td>Set-Cookie:</td>
<td>SERVERCONNECTIONID=; expires=Wed, 02-Jan-1991 00:00:00 GMT; path=/;</td>
</tr>
<tr>
<td></td>
<td>domain=unibz.it</td>
</tr>
<tr>
<td>Set-Cookie:</td>
<td>SERVERCONNECTIONID=CONID2007082641691672735; path=/; domain=www.unitz.it</td>
</tr>
<tr>
<td>Set-Cookie:</td>
<td>CheckSum=daNadVniEtbXdks; path=/; domain=www.unitz.it</td>
</tr>
<tr>
<td>Set-Cookie:</td>
<td>wscharset=UTF-8; path=/; domain=www.unitz.it</td>
</tr>
<tr>
<td>Connection:</td>
<td>close</td>
</tr>
<tr>
<td>Transfer-Encoding:</td>
<td>chunked</td>
</tr>
<tr>
<td>Content-Type:</td>
<td>text/html; charset=UTF-8</td>
</tr>
</tbody>
</table>

Content (17.03 KiB)

<!-- web4biz Content Management System, a product of ic4b AG Germany -- http://www.ic4b.com
<!-- created 25.09.2007 - 11:34 , Version 03.05.03.00 -->
<!--DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<!-- block language management part I -->

<!-- end block language management part I -->
Request and Responses

- **Request** and **responses** have two parts: **headers** and **content** (is called *message-body* in the specs)

- If you type a **URL into your browser**, the browser creates an **HTTP request** and sends it to a server

- The server finds the **requested file** and sends it back in an **HTTP response**

- The response **headers** are describing things like the **type of web server**, the **file type of the response**, the **length of the response** and other info

- The response **content** is the file data.
Request and Response Format

- The format of the request and response messages are similar and both kinds of messages consist of:
  - an initial line,
  - zero or more header lines,
  - a blank line (i.e. a CRLF by itself), and
  - an optional message body (e.g. a file, or query data, or query output).

- Put another way, the format of an HTTP message is:
  - <initial line, different for request vs. response>
  - Header1: value1
  - Header2: value2
  - Header3: value3
  - <optional message body goes here, like file contents or query data; it can be many lines long, or even binary data $&*%@!^%@>
HTTP Specifications

HTTP-message = Request | Response

generic-message = start-line
    *(message-header CRLF)
    CRLF
    [ message-body ]

start-line = Request-Line | Status-Line
message-header = field-name ":" [ field-value ]

http://www.w3.org/Protocols/rfc2616/rfc2616.html
HTTP Message
Initial Request Line

- A request line has three parts, separated by spaces:
  - a **method** name,
  - the **local path** of the requested resource,
  - and the **version of HTTP** being used.

- A typical request line is:
  - `GET /path/to/file/index.html HTTP/1.0`

- **GET** is the most common HTTP method; it says "give me this resource".

- The path is the part of the URL after the host name, also called the **request URI** (a URI is like a URL, but more general, see lecture 2)

- The HTTP version always takes the form "**HTTP/x.x**", uppercase – current version is 1.1.
Initial Response Line (status)

- The initial response line, called the status line, also has three parts separated by spaces:
  - the **HTTP version**,  
  - a **response status code** that gives the result of the request,  
  - and an English **reason phrase** describing the status code

- **Examples:**
  - HTTP/1.1 200 OK  
  - HTTP/1.1 404 Not Found

- The HTTP version is in the same format as in the request line, "**HTTP/x.x**".
HTTP Status Codes

The status code response groups

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>1xx</td>
<td>Information</td>
<td>100 = server agrees to handle client’s request</td>
</tr>
<tr>
<td>2xx</td>
<td>Success</td>
<td>200 = request succeeded; 204 = no content present</td>
</tr>
<tr>
<td>3xx</td>
<td>Redirection</td>
<td>301 = page moved; 304 = cached page still valid</td>
</tr>
<tr>
<td>4xx</td>
<td>Client error</td>
<td>403 = forbidden page; 404 = page not found</td>
</tr>
<tr>
<td>5xx</td>
<td>Server error</td>
<td>500 = internal server error; 503 = try again later</td>
</tr>
</tbody>
</table>

Details:
http://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html#sec6.1.1
Header Lines

- Header lines provide information about the request or response, or about the object sent in the message body.
- Header lines should end in CRLF.
- The header name is **not case-sensitive** (though the value may be).
- Any number of spaces or tabs may be between the ":" and the value.
- Header lines beginning with space or tab are actually part of the previous header line, folded into multiple lines for easy reading.
- Examples: following two headers are equivalent:
  
  Header1: some-long-value-1a, some-long-value-1b
  HEADER1: some-long-value-1a,
  some-long-value-1b
HTTP Message Headers

- HTTP 1.0 defines 16 headers, none are required. HTTP 1.1 defines 46 headers (1 required host: )
  [http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14](http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14)

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

- An HTTP message may have a body of data sent after the header lines.
- In a **response**: the requested **resource** returned to the client (or perhaps explanatory text if there's an error).
- In a **request**: user-entered data or uploaded files are sent to the server.
- If an HTTP message includes a body, there are usually header lines in the message that describe the body:
  - The **Content-Type:** header gives the MIME-type of the data in the body, such as **text/html** or **image/gif**.
  - The **Content-Length:** header gives the number of bytes in the body.
MIME types

- An **Internet media type** (originally **MIME**) and sometimes a **Content-type** (because of HTTP) is a two-part identifier for file formats on the Internet.

- A media type is composed of at least two parts: a *type*, a *subtype*, and one or more optional parameters.

- **Example**
  - `Content-Type:text/html; charset=UTF-8`

- **Type** *application*: **Multipurpose files**
  - `application/javascript`, `application/octet-stream`, `application/octet-stream`, ...

- **Type** *audio*: `audio/mpeg`, `audio/x-wav`

- **Type** *image*: `image/gif`, `image/png`, `image/jpeg`

- **Type** *text*: `text/css`, `text/html`, `text/xml`

- **Type** *video*: `video/mpeg`, `video/mp4`, `video/quicktime`

[http://en.wikipedia.org/wiki/MIME_type]
Parameters

- Clients can pass **parameters** to the server (e.g. rawhtml=yes)
- Parameters are simple **name and value pairs**
- Parameters are generally collected from **HTML forms**
- Form values are sent as parameters to the server when you click the **Submit** or **Next** button on the form
- The client **encodes** parameters before they are sent to the server (**why?**)
- Multiple parameters are separated by **ampersands** (&)

**example**
Unsafe Characters

- Some Characters need to be encoded
  - ~  [ASCII: 126 (0x7E)]
  - SPACE  [ASCII: 32 (0x20)]
  - %  [ASCII: 37 (0x25)]

- Examples
  - http://www.bob.com/%7Ekelly/
  - http://www.bob.com/my%20home
  - %20page.html
  - http://www.bob.com/100%25Crankiness.html

URL Encoding reference
http://www.w3schools.com/TAGS/ref_urlencode.asp
http://www.asciitable.com/
Parameters

- **Encoding rules:**
  - **Space** characters are converted to a plus (+) sign (or %20)
  - The following characters remain unchanged (*but can also be encoded*): lowercase letters a-z, uppercase letters A-Z, the numbers 0-9, the period (.), the hyphen (-) the asterisk (*), and the underscore (_)
  - All other characters are converted into “%xy”, where “xy” is a hexadecimal number that represents the low 8 bits of the character
    - ! * ' ( ) ; : @ & = + $ , / ? % # [ ]
HTTP Methods

The built-in HTTP request methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Request to read a Web page</td>
</tr>
<tr>
<td>HEAD</td>
<td>Request to read a Web page’s header</td>
</tr>
<tr>
<td>PUT</td>
<td>Request to store a Web page</td>
</tr>
<tr>
<td>POST</td>
<td>Append to a named resource (e.g., a Web page)</td>
</tr>
<tr>
<td>DELETE</td>
<td>Remove the Web page</td>
</tr>
<tr>
<td>TRACE</td>
<td>Echo the incoming request</td>
</tr>
<tr>
<td>CONNECT</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>OPTIONS</td>
<td>Query certain options</td>
</tr>
</tbody>
</table>

Details:
http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9
Parameters with GET and POST

- With a GET request, parameters are added at the end of the URL in encoded form.
- If your URL is the following:
  
  http://localhost/myServlet

- The parameter “user” added to the URL:
  
  http://localhost/myServlet?user=myName

- Additional parameters can be added, separated by &:
  
  http://localhost/myServlet?user=myName&zip=080808

- **POST** is basically the same as GET, but parameters are handled differently:
  
  - parameters are passed as the body of request with the same type of encoding.
  - If you have lots of parameters or binary data, you may use the POST request.

Try **example** with POST.
HTTP Proxy Servers

- "Middlemen" between clients and servers
- May "rewrite" HTTP requests and responses
Proxy

- An **HTTP proxy** is a program that acts as an intermediary between a client and a server.
- It receives requests from clients, and forwards those requests to the intended servers.
- Thus, a proxy has functions of both a **client** and a **server**.
- Requests to a proxy differ from normal requests: in the first line, they use the complete URL of the resource being requested, instead of just the path.
- For example,
  - `GET http://www.somehost.com/path/file.html` HTTP/1.0
- That way, the proxy knows which server to forward the request to (though the proxy itself may use another proxy).
Why Proxy Servers?

- Content filtering
  - Packet filter (firewall)
  - Prevent children from accessing sites
- Used to contact server in special way
  - Firewall (shadow the client IP number)
  - Security
- Rewrite content
  - Swedish chef (translation)
- Redirect requests
  - Content router
  - To improve performance
- Logging activity.
Why Proxy Servers?

Free University of Bozen - Bolzano
TRILINGUAL AND INTERNATIONAL

Public | Prospective students | Students | Organisation & Staff

UNIVERSITY

- Numbers and Facts
- Events

PRESS & COMMUNICATION

RESEARCH

BUSINESS

RECOGNITION OF AUSTRIAN QUALIFICATIONS

SCHOOLS

ALUMNI

UNIVERSITY PRESS

DER BODEN UNTER DEN FÜSSEN

WENN RESSENDISKRIMINIERUNG, TRADITION UND KONVENTION EINER JUNGEN FRAU DEN BODEN UNTER DEN FÜSSEN WEGSCHREIßEN, BIETET DIESE JÄGERIN MS STEFFI FÜR EINEN MITREISSENDEN ROMAN ANGESIEDELT HAT.

RENNPAPPE_09

NACH DEM GROSSER ERFolg IM JAHr 2007 FUNDET SICH DAS ZWEITE Mal DER INTERNAtionALE WORKSHOP Rennpappe_09 an der Fakultät für Design und Künste statt. Die Idee zur "rennpappe" stammt...

UNIVERSITÀ A MISURA DI FAMIGLIA

Assegnare lavoro e studio ai tanti liberi della famiglia è particolarmente importante per le Università, che per alcuni anni sono
HTTP Versions

- **HTTP/1.0 (May 1996)**
  - This is the first protocol revision to specify its version in communications and is still in wide use, especially by proxy servers

- **HTTP/1.1 (June 1999)**
  - Faster response, by allowing multiple transactions to take place over a single **persistent connection**
  - Faster response and great bandwidth savings, by adding **cache support** (If-Modified-Since, If-Unmodified-Since)
  - Faster response for dynamically-generated pages, by supporting **chunked encoding**, which allows a response to be sent before its total length is known
  - Efficient use of **IP addresses**, by allowing multiple domains to be served from a single IP address.
HTTP 1.1 Client

- To **comply** with HTTP 1.1, clients must
  - include the **Host:** header with each request
  - accept responses with **chunked** data
  - either support **persistent connections**, or include the "**Connection: close**" header with each request
  - handle the "**100 Continue**" response.
In HTTP 1.1, one server at one IP address can be *multi-homed*, i.e. the home of several Web domains: "www.unibz.it" and "www.visitbozen.com" can live on the same server.

Actually this is the standard for web hosting (IP sharing – virtual hosting) [link](#).

Thus, every HTTP request must specify which host name the request is intended for, with the **Host:** header.

A complete HTTP 1.1 request might be:

```
GET /path/file.html HTTP/1.1
Host: www.host1.com:80
```

**Host:** is the only required header in an HTTP 1.1 request. *It's also the most urgently needed new feature in HTTP 1.1*

The majority of sites require the Host header (otherwise bad request).
Chunked Transfer-Encoding

- To start sending a response before knowing its total length, the server must use the simple *chunked transfer-encoding*.
- The complete response is broken into smaller chunks and sent in series.

HTTP/1.1 200 OK
Date: Fri, 31 Dec 1999 23:59:59 GMT
Content-Type: text/plain
Transfer-Encoding: chunked

1a; ignore-stuff-here
abcdefghijklmnopqrstuvwxyz
10
1234567890abcdef
0
some-footer: some-value
another-footer: another-value
[blank line here]

1a = 16+10 characters in the first chunk
10 = 16 characters in the second chunk
end

data in the first chunk
data in the second chunk
Non-Persistent HTTP: Response time

Definition of **RTT**: time for a small packet to travel from client to server and back (round trip time)

Response time:
- one RTT to initiate TCP connection
- one RTT for HTTP request and first few bytes of HTTP response to return
- file transmission time

**total = 2RTT + transmit time**
Persistent Connections

- In HTTP 1.0 TCP connections are closed after each request and response, so each resource to be retrieved requires its own connection.

- Opening and closing TCP connections takes a substantial amount of CPU time, bandwidth, and memory.

- Most Web pages consist of several files on the same server, so much can be saved by allowing several requests and responses to be sent through a single persistent connection.

- Persistent connections are the default in HTTP 1.1.
Persistent Connections

- Persistent connection is the default in HTTP1.1 unless:

- If a client includes the "Connection: close" header in the request, then the connection will be closed after the corresponding response.

- If a response contains "Connection: close", then the server will close the connection following that response - don’t send any more requests through that connection.

- A server might close the connection **before all responses are sent** - a client must keep track of requests and resend them as needed.
The "100 Continue" Response

- During the course of an HTTP 1.1 client sending a request to a server, the server might respond with an interim "100 Continue" response.
- This means the server has received the first part of the request.

HTTP/1.1 100 Continue

HTTP/1.1 200 OK
Date: Fri, 31 Dec 1999 23:59:59 GMT
Content-Type: text/plain
Content-Length: 42
some-footer: some-value
another-footer: another-value
abcdefghijklmnopqrstuvwxyz1234567890abcdef

http://www.jmarshall.com/easy/http/
Caching

Hierarchical caching with three proxies.
Caching

- **Who should do the caching?**
  - Client: browser stores last visited pages
  - Proxy: stores most commonly accessed pages
  - Server: sometimes does not recompute a dynamic page

- **How long should pages be cached?**
  - Some pages should never be cached – stocks rates (*no-cache*)
  - If the server used *Last-Modified* response header – then client should not cache pages that are recently modified
  - Use the *If-Modified-Since* request header – see next slides.
The **Date** and Last-Modified Headers

- Caching is an important improvement in HTTP 1.1 - it can't work without *timestamped* responses
- Servers must timestamp every response with a **Date:** header containing the current time, in the form
  
  Date: Fri, 31 Dec 2010 23:59:59 GMT

- All responses except those with 100-level status (but including error responses) **must** include the **Date:** header
- All time values in HTTP use Greenwich Mean (meridian) Time
- **Last-Modified** header indicates really when the page has been modified (it is not required).
If-Modified-Since: or If-Unmodified-Since:

- To avoid sending resources that don't need to be sent, thus saving bandwidth
  - Request headers: **If-Modified-Since:** and **If-Unmodified-Since:**
  - *only send to me the resource if it has changed since this date* (the other says the opposite)
- If the requested resource **has been modified** since the given date, the server ignores the header and return the resource
- Otherwise it returns a "**304 Not Modified**" response, including the **Date:** header and no message body:
  
  `HTTP/1.1 304 Not Modified
  Date: Fri, 31 Dec 2010 23:59:59 GMT`
Statelessness and Cookies

Some examples of cookies.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Path</th>
<th>Content</th>
<th>Expires</th>
<th>Secure</th>
</tr>
</thead>
<tbody>
<tr>
<td>toms-casino.com</td>
<td>/</td>
<td>CustomerID=497793521</td>
<td>15-10-02 17:00</td>
<td>Yes</td>
</tr>
<tr>
<td>joes-store.com</td>
<td>/</td>
<td>Cart=1-00501;1-07031;2-13721</td>
<td>11-10-02 14:22</td>
<td>No</td>
</tr>
<tr>
<td>aportal.com</td>
<td>/</td>
<td>Prefs=Stk:SUNW+ORCL;Spt:Jets</td>
<td>31-12-10 23:59</td>
<td>No</td>
</tr>
<tr>
<td>sneaky.com</td>
<td>/</td>
<td>UserID=3627239101</td>
<td>31-12-12 23:59</td>
<td>No</td>
</tr>
</tbody>
</table>
Continuity Problem (User’s Point of View)
The Illusion of Continuity

- User thinks that choices made on page 1 are remembered on page 3

- However
  - HTTP is **Stateless**
    - Requests from same user do not necessarily come in adjacent requests
    - There may be several other requests in the middle.
Continuity Problem: Server’s Point of View

Request 1

Request 2
Solution: Store State Somewhere

- HTTP is stateless

- **Server Side:**
  - Makes Server Really Complicated
  - State per client!
  - *We will see this solution in the next lectures*

- **Client Side:**
  - Server puts little notes on the client side
  - When client submits the next form, it also (unknowingly) submits these little notes
  - Server reads the notes, remembers who the client is – or obtain the info to generate a page that logically follows.
Technique: Hidden Fields

- In a Form a field may be "hidden" – it is sent to the server but not displayed to the user

```html
<form name="myform" action="http://www.mydomain.com/dothat" method="POST">
  <input type="text" size="25" name="Product" value="Enter the product name here!">
  <input type="hidden" name="uid" value="007">
</form>
```

- Simple way to store state on client side
- But what if the client (user) closes browser, returns to your site 30 seconds later?
- The uid information is written in the requested url, so it can be easily seen by a malicious user (better use POST!).
Technique: HTTP Cookies

- **Idea**
  - **Server** (e.g., a servlet) sends a simple (**cookie**) name and value to client - telling which URLs this cookie is valid for
  - **Client returns** the same name and value when it connects to the same site (or same domain, depending on cookie settings)

- Typical Uses of Cookies – storing information client side
  - Identifying a user during an e-commerce session
  - Avoiding username and password
  - Customizing a site
  - Focusing advertising

- Positive side: Site will remember who you are
- Negative side: Privacy?
Cookie Syntax

- On HTTP **response**, the **server** writes a header:
  
  ```
  Set-Cookie: NAME=VALUE; expires=DATE; path=PATH; domain=DOMAIN_NAME; secure
  ```

- On HTTP **requests**, the **client** (browser):
  
  - looks through cookie database,
  - finds all cookies that match the current URL (domain + path), and
  - **Writes a header:**
    
    ```
    Cookie: NAME1=OPAQUE_STRING1; NAME2=OPAQUE_STRING2; ...
    ```

  - "opaque" means that the cookie is not explicitly containing any information but enable the server to understand who is the client.
Cookie Example

Client requests a document, and receives in the response:
Set-Cookie: CUSTOMER=WILE_E_COYOTE; path=/; expires=Wednesday, 09-Nov-2011 23:12:40 GMT

When client requests a URL in path "/" on this server, it sends:
Cookie: CUSTOMER=WILE_E_COYOTE

Client requests a document, and receives in the response:
Set-Cookie: PART_NUMBER=ROCKET_LAUNCHER_0001; path=/

When client requests a URL in path "/" on this server, it sends:
Cookie: CUSTOMER=WILE_E_COYOTE;
PART_NUMBER=ROCKET_LAUNCHER_0001

Client receives:
Set-Cookie: SHIPPING=FEDEX; path=/foo
Cookie Example

When client requests a URL in path "/" on this server, it sends:

Cookie: CUSTOMER=WILE_E_COYOTE;
       PART_NUMBER=ROCKET_LAUNCHER_0001

When client requests a URL in path "/foo" on this server, it sends:

Cookie: CUSTOMER=WILE_E_COYOTE;
       PART_NUMBER=ROCKET_LAUNCHER_0001;
       SHIPPING=FEDEX
Cookies and Focused Advertising
Some Problems with Cookies

- The problem is privacy, not security
  - Servers can remember your previous actions
  - If you give out personal information, servers can link that information to your previous actions
  - Servers can share cookie information through use of a cooperating third party like doubleclick.net
  - Poorly designed sites store sensitive information like credit card numbers directly in cookie
  - JavaScript bugs let hostile sites steal cookies (old browsers)

- Moral for servlet authors
  - If cookies are not critical to your task, avoid servlets that totally fail when cookies are disabled
  - Don't put sensitive info in cookies.
Scenario

- Two sites can share data on a user by each loading small images off the same third-party site and the third party site uses cookies to identify the users
- search-site.com and random-site.com want to display direct ads from ad-site.com, based on the user search history
- If the user searched for "Java Servlets" the search site can return a page with a (small) image link
  - `<IMG SRC="http://ad-site.com/banner?data=Java+Servlets" ...>`
- Therefore, the browser connect to ad-site.com that returns the small (invisible) image and a cookie
- After, the user visits random-site.com that can return a banner generated by ad-site.com
  - `<IMG SRC="http://ad-site.com/banner" ...>`
- The ad-site receives the request and its cookie
- Looking at his database can understand that this user searched on search-site for "Java Servlets" and can send a "personalized" banner.
Manually Deleting Cookies