UCL
Université catholique de Louvain

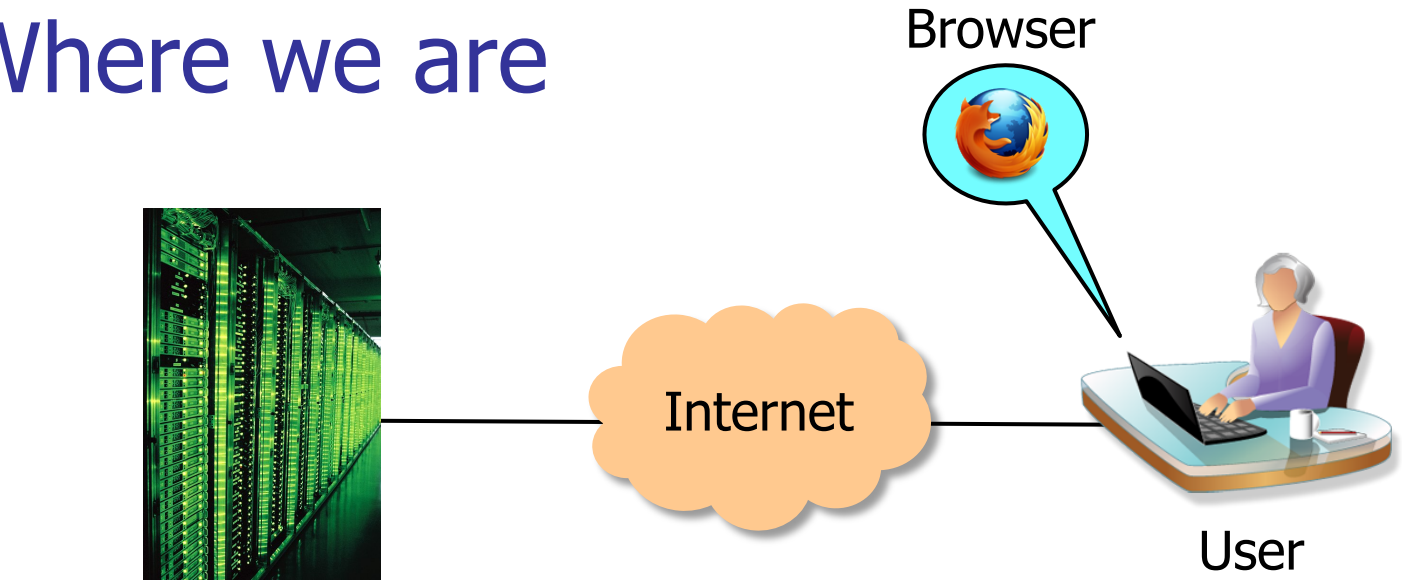
INGI2145: CLOUD COMPUTING (Fall 2014)

Web Programming

27 November 2014

Lecture slides adapted from Upenn NETS212 by A. Haeberlen, Z. Ives
Reproduced with permission

Where we are

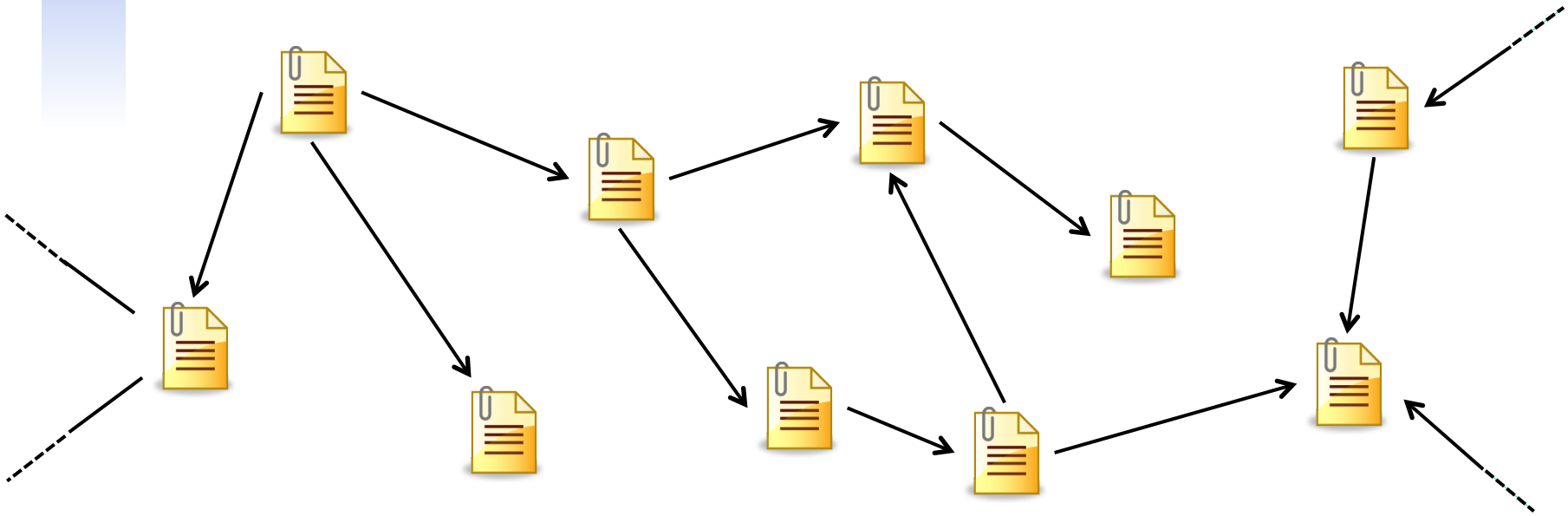


- So far: The 'backend'
  - Large-scale distributed system processes lots of data
  - Economic model, architecture, programming (MapReduce)
- Next: The 'frontend'
  - How to get the data to the users
  - How to build interactive web applications

Goals for the next two lectures

- Architecture of the Web
  - Key concepts: Hyperlink, URI/URL, ...
  - Building blocks: HTML, HTTP, DNS, ...
- Servers (esp., web servers)
  - Client/server model
  - State, and where to keep it
  - Architecture: Threads, thread pools, events
- Web applications
  - Dynamic content; maintaining state
  - Java servlets, Node.js

The World Wide Web (WWW)



- A service that runs on the Internet
  - Not identical to the Internet!
- A collection of interconnected documents and other resources
  - Not a hierarchy - any document can reference any other!

Where did the web come from?

- 1989: Tim Berners-Lee is at CERN
  - CERN is a large organization with high turnover
  - Useful information (e.g., about projects) is constantly being lost because it cannot be found
  - Need to organize information - but how?
  - Hierarchical structure is not flexible enough
- Writes "Information management: A proposal"
  - "The hope would be to allow a pool of information to develop which could grow and evolve with the organisation and the project it describes"
  - "CERN meets now some problems which the rest of the world will have to face soon"

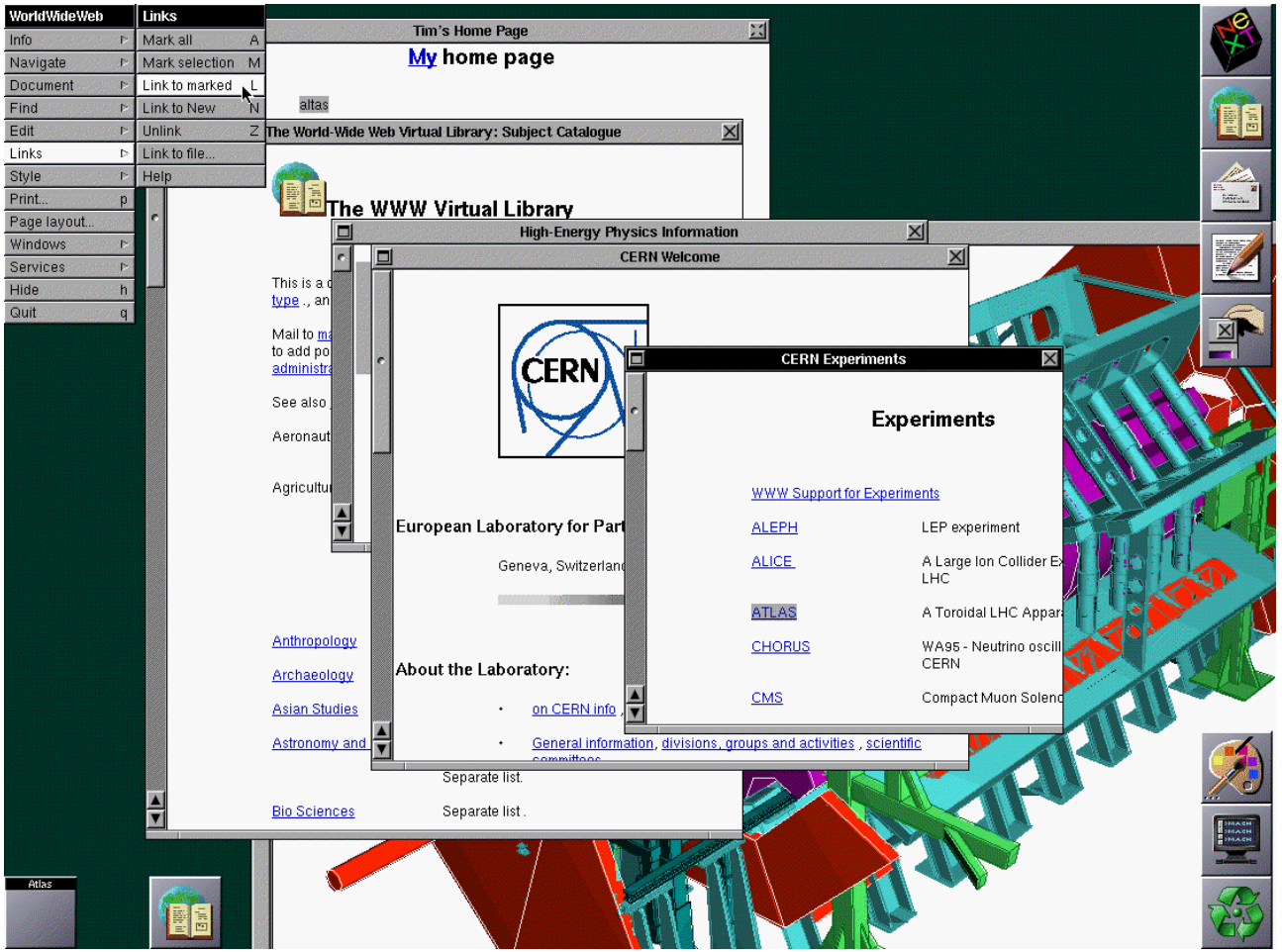
http://www.w3.org/History/1989/proposal.html

Was the Web invented from scratch?

- 1945: Vannevar Bush's article on the Memex
  - Could make+follow links between documents on microfiche
- 1960: Doug Engelbart's oNLine System (NLS)
  - Hypertext browsing+editing, email, etc.
  - Invents the mouse for this purpose
- 1989: Tim Berners-Lee writes "Information Management: A proposal" (at CERN)
  - Recirculated in 1990. In September, Berners-Lee's boss OKs purchase of a NeXT cube for the project
  - Berners-Lee writes the first web server and browser; demonstrable by Christmas 1990

http://www.w3.org/History.html

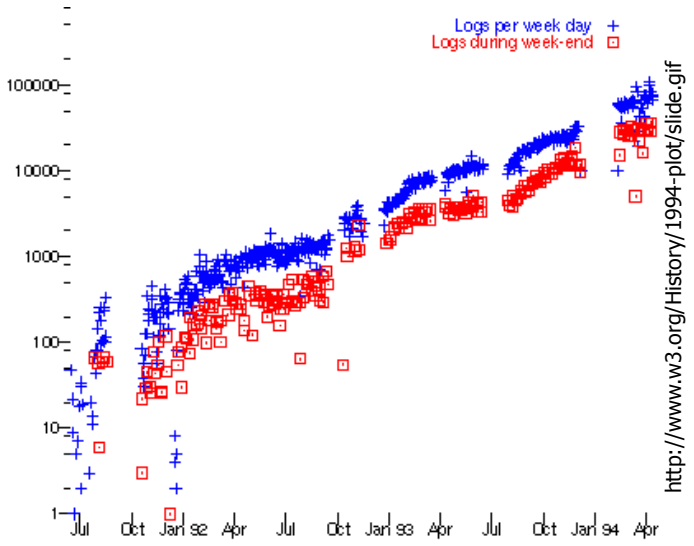
The first browser



- http://www.w3.org/History/1994/WWW/Journals/CACM/screensnap2_24c.gif

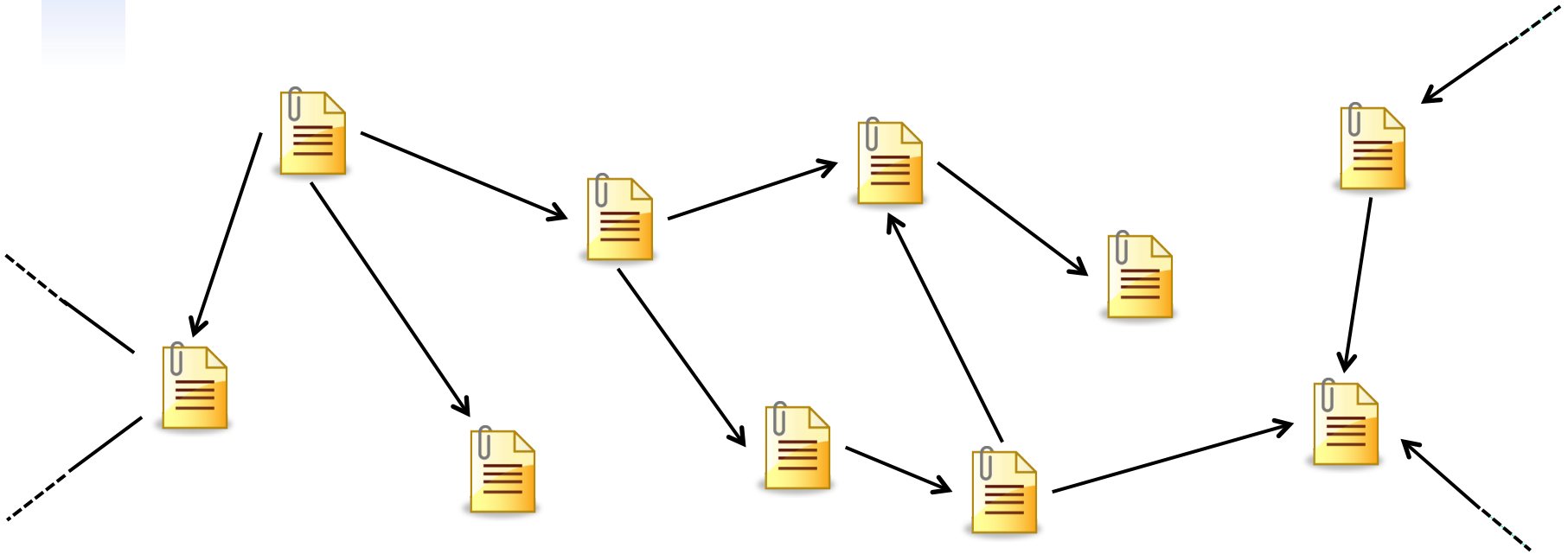
A brief history of the WWW

- Situation as of 1992:
  - Competing system (Gopher) is already in place
  - No graphical browsers for platforms other than NeXT
- 1993: Turning point
  - Feb: First alpha release of Marc Andreesen's "Mosaic for X"
    - Andreesen would later be a co-founder of Netscape
  - Feb: U. Minnesota announces it will charge license fees for its Gopher server
  - Apr: CERN says WWW will be freely usable by anyone



Load on the first web server (info.cern.ch)

Making the Web: Ingredients



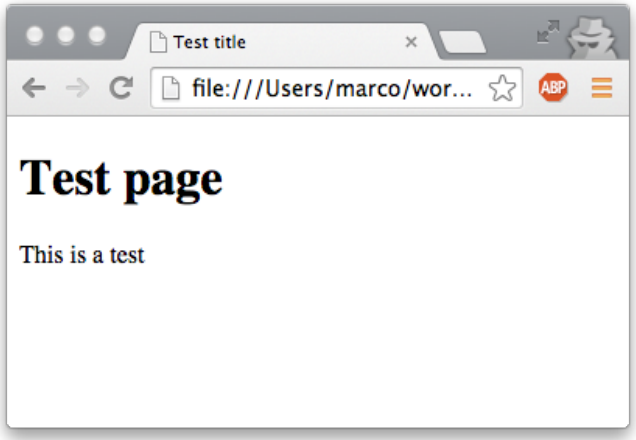
- What do we need to build the Web?

What do we need to make the Web work?

- Formats for writing the documents — HTML
- A program for displaying documents — Browser
- Unique names for the documents — URIs, URLs
- A way to find documents — DNS
- A system for delivering documents
  - Architecture — Client/server model
  - Efficient implementation — Threads; event-driven prog.
- A protocol for transferring documents — HTTP
- A way to make content dynamic
  - Programming model — Scripting; servlets
  - Keeping state — Cookies

HTML: Presentation and representation

```
<!doctype html>
<html>
  <head>
    <title>Test title</title>
  </head>
  <body>
    <h1>Test page</h1>
    <p>This is a test</p>
  </body>
</html>
```



Representation
(bits and bytes in the document)

Presentation
(document displayed by the browser)

- Original idea: Separate the two
  - Document representation would only describe the structure and the semantic content
  - Browser would take care of the visual layout
- Pros and cons of this approach?
  - Do you think people are following it today?

A simple web page

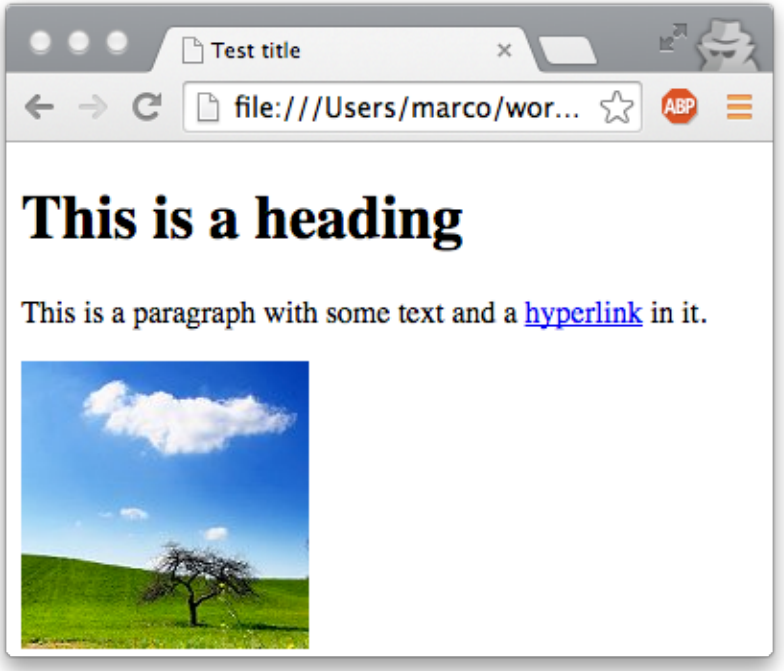
Begin tag

Content

End tag

```
<!doctype html>
<html>
  <head>
    <title>Test title</title>
  </head>
  <body>
    <h1>This is a heading</h1>
    <p>This is a paragraph with
    some text and a
    <a href="foo.html">hyperlink</a>
    in it.</p>
    <img src="cloud.jpg" alt="Cloud">
  </body>
</html>
```

Attributes



- General structure: Elements, tags, content
  - Most elements have a begin tag and an end tag (denoted by /)
  - Tags may have attributes, e.g., <img> has URL of the image
  - Hierarchy of elements

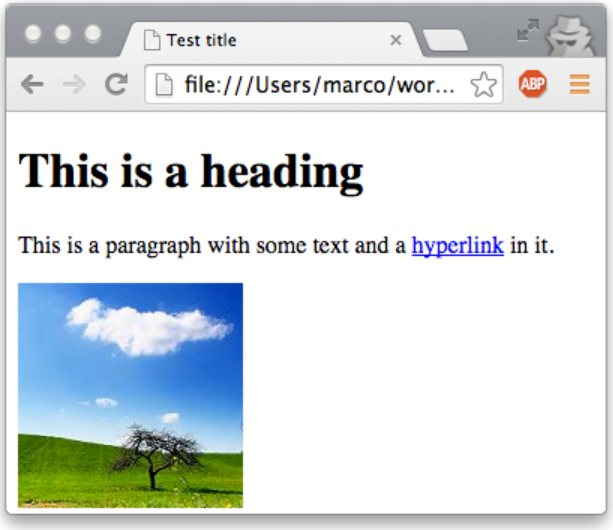
Some basic HTML elements

- Basic template for a HTML document:
  - Header: Contains the title; may contain metadata, e.g., author
  - Body: Contains the actual content

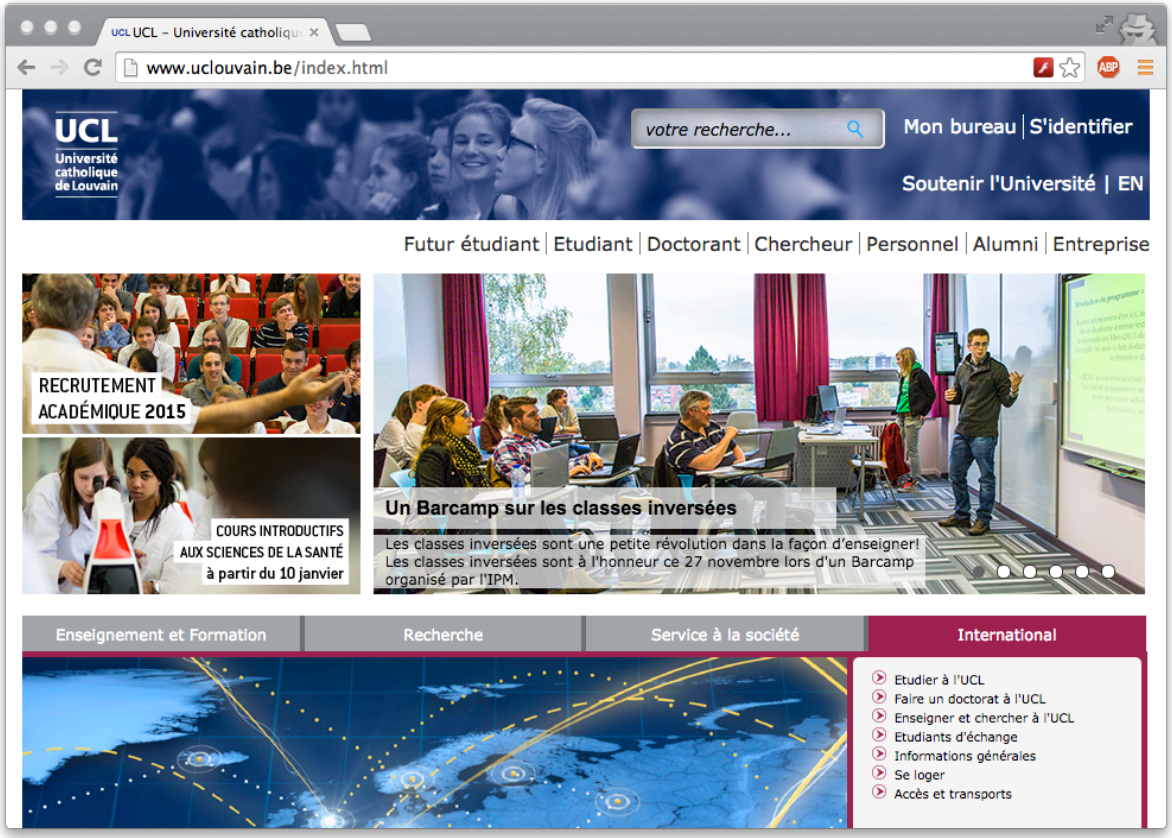
```
<!doctype html>
<html>
  <head>
    <title>My title</title>
  </head>
  <body>
    ... content ...
  </body>
</html>
```

- Typical HTML elements:
  - Headings (several levels): `<h1>...</h1>, <h2>...</h2>, ...`
  - Paragraphs: `<p>This is a paragraph</p>`
  - Forced linebreak: `This is a line.<br>This is the next line.`
  - Unordered lists: `<ul><li>Item 1<li>Item 2<li>Item 3</ul>`
  - Links: `This is a <a href="url">link</a>`
  - Images: `<img src="url" alt="Description">`

Is basic HTML rendering enough?



vs.

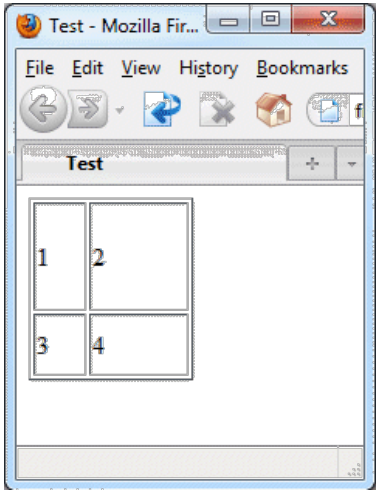


Taking control over the presentation

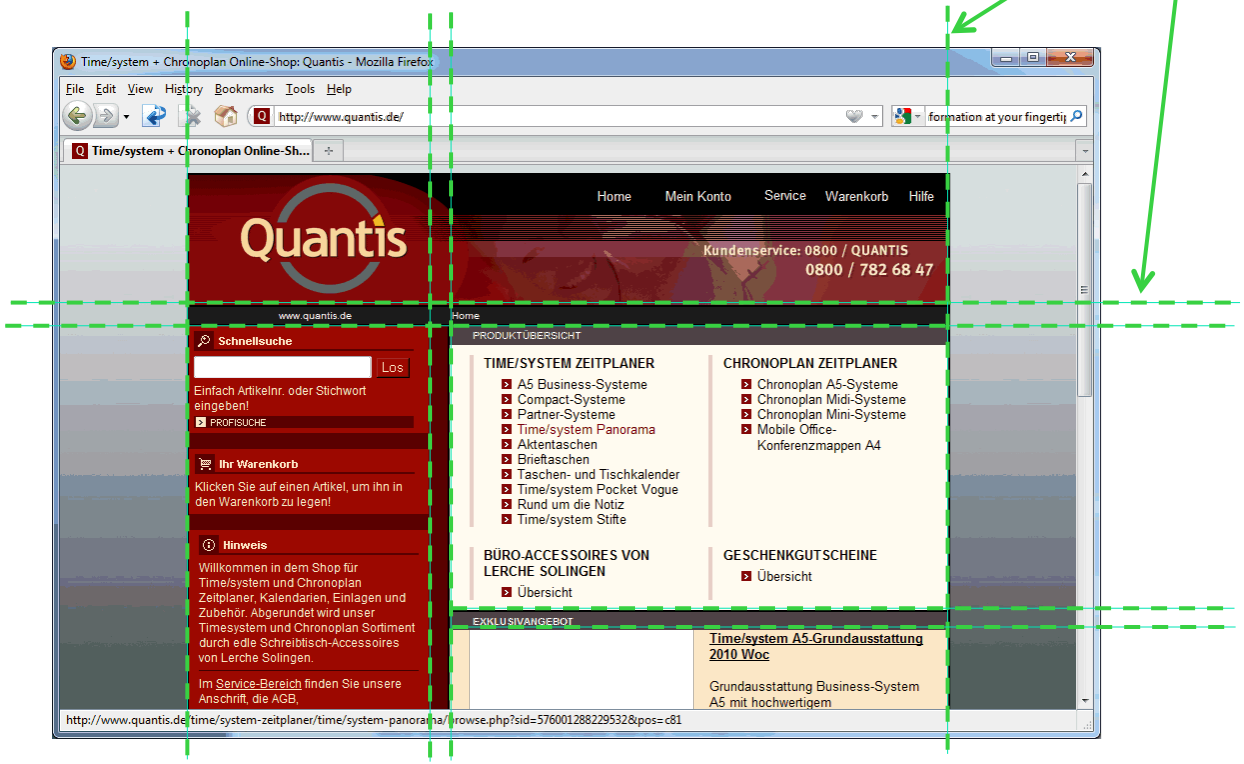
- You can explicitly tell the browser how to render certain elements:
  - Font: `<font size="17" color="red" face="Helvetica">`
  - Style: `<b>bold</b>`, `<i>italics</i>`, `<tt>teletype<tt>`, ...

- You can define tables with a specific layout

```
<table border="1">
  <tr height="70">
    <td width="30">1</td>
    <td width="60">2</td>
  </tr>
  <tr height="40">
     <td width="30">3</td>
     <td width="60">4</td>
  </tr>
</table>
```



The world before CSS



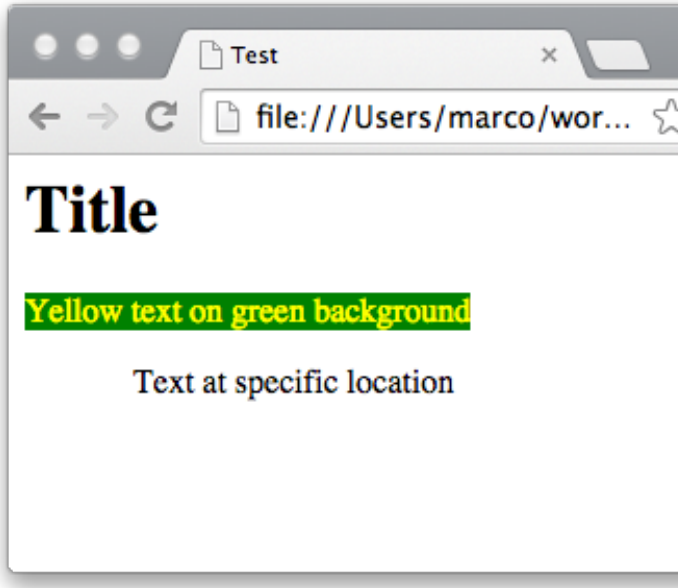
- Web designers used tables and formatting to create very complex layouts
  - Result: Documents were huge (lots of redundancy) and basically unreadable

Today: Cascading Style Sheets

```
<html>
  <head><title>Test</title></head>
  <link rel="stylesheet" href="test.css">
  <body>
    <h1>Title</h1>
    <span class="ytext">Yellow text on
    green background</span>
    <span class="loc">Text at
    specific location</span>
  </body>
</html>
```

Reference

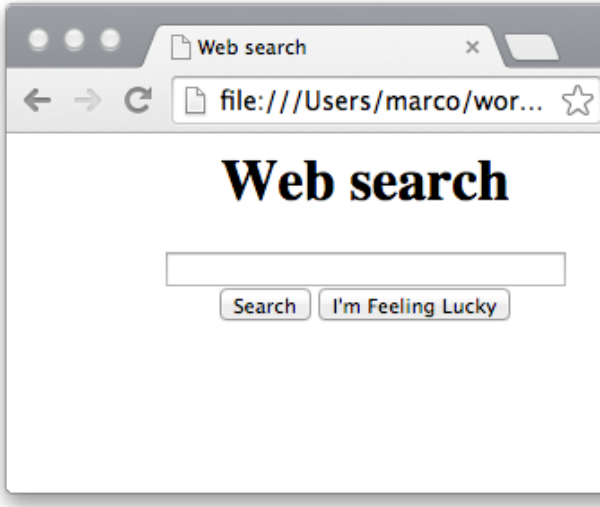
```
.ytext { color:yellow; background:green }
.loc { position:absolute; top:100px; left:60px }
```



- Idea: Separate content and formatting again
  - Formatting instructions are kept in a separate file that is linked from the document
  - Document is annotated with references to the formatting instructions, via class="xxx" attribute or special elements (`<span>...</span>`, `<div>...</div>`)

Forms

```
<html>
  <head><title>Web search</title></head>
  <body>
    <center><h1>Web search</h1>
    <form action="search.html" method="post">
      <input type="text" size="40" name="term"><br>
      <input type="submit" value="Search">
      <input type="button" value="I'm Feeling Lucky">
    </form></center>
  </body>
</html>
```



- What if we want the user to input some data?
  - `<form>` element creates and input form in the document
  - Several input types available: Single line of text, multiline text, radio buttons, checkboxes, buttons, dropdown boxes...
  - Data is sent over the network once the form is submitted
  - One way to create interactive 'web applications'; more about this later

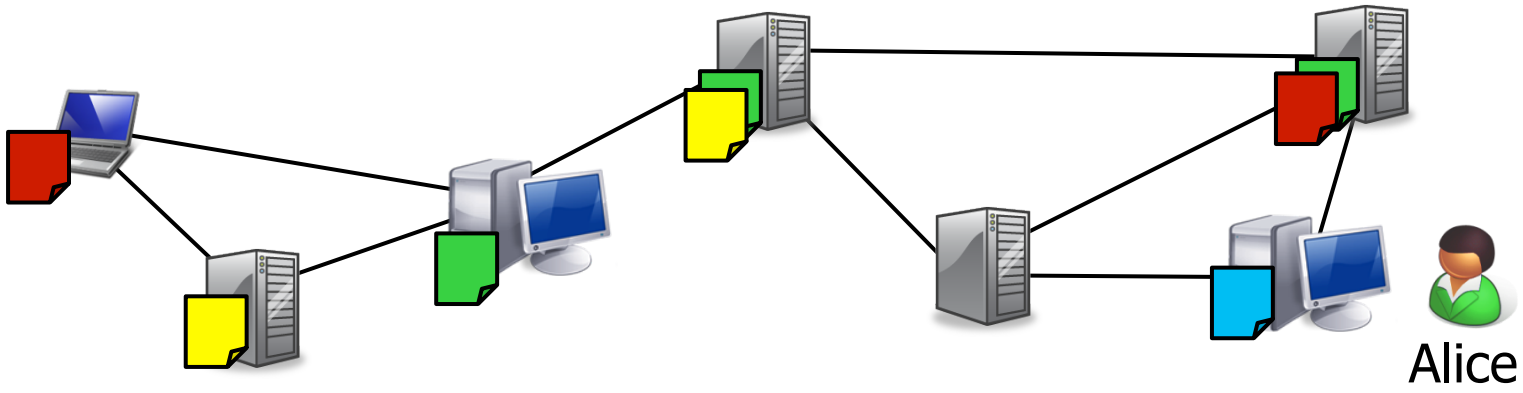
Recap: Hypertext Markup Language

- Separation of content and visual layout
- Elements, tags, attributes
  - Hierarchical structure; elements contain other elements
- Some common elements
  - Standard elements: <html>, <head>, <title>, <body>
  - Structure: <h1>,
, <p>, <a>, <ul>, <img>
  - Tables: <table>, <tr>, <td>
  - Formatting: <font>, <b>, <i>, <pre>
  - Forms: <form>, <input>
- Cascading Style Sheets
  - Usually in a separate file + referenced from the main file

What do we need to make the Web work?

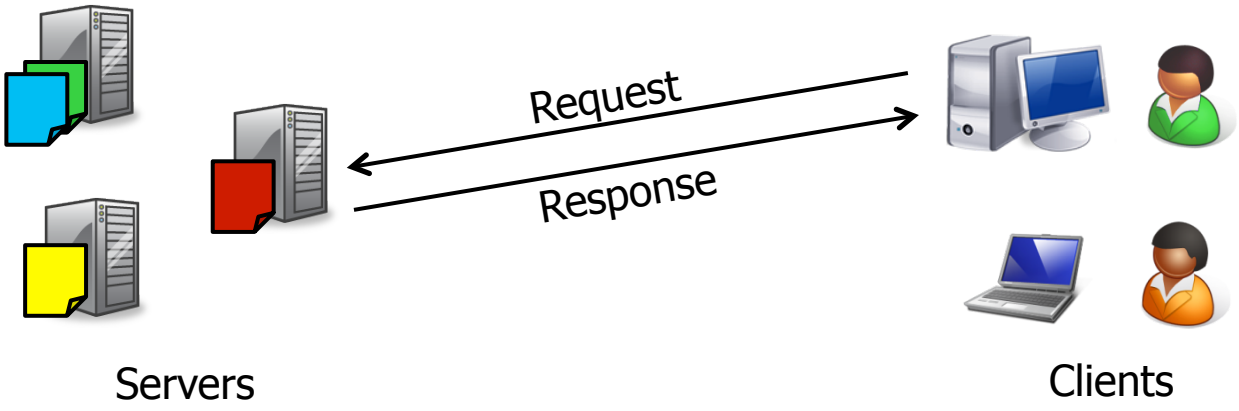
- Formats for writing the documents — HTML
- A program for displaying documents — Browser
- Unique names for the documents
- A way to find documents
- A system for delivering documents
  - Architecture
  - Efficient implementation
- A protocol for transferring documents
- A way to make content dynamic
  - Programming model
  - Keeping state

The peer-to-peer model



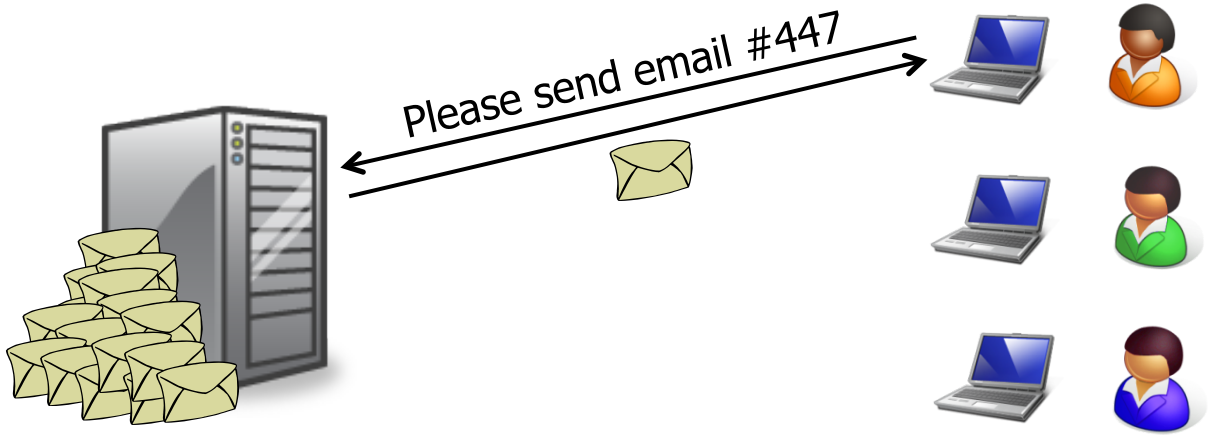
- How the Web could work (but doesn't):
  - Each machine locally stores some documents
  - If a machine needs a new document, it asks some other machines until it finds one that already has the document
  - No machine is special - they are all 'peers'
- Pros and cons of this approach?

The client-server model



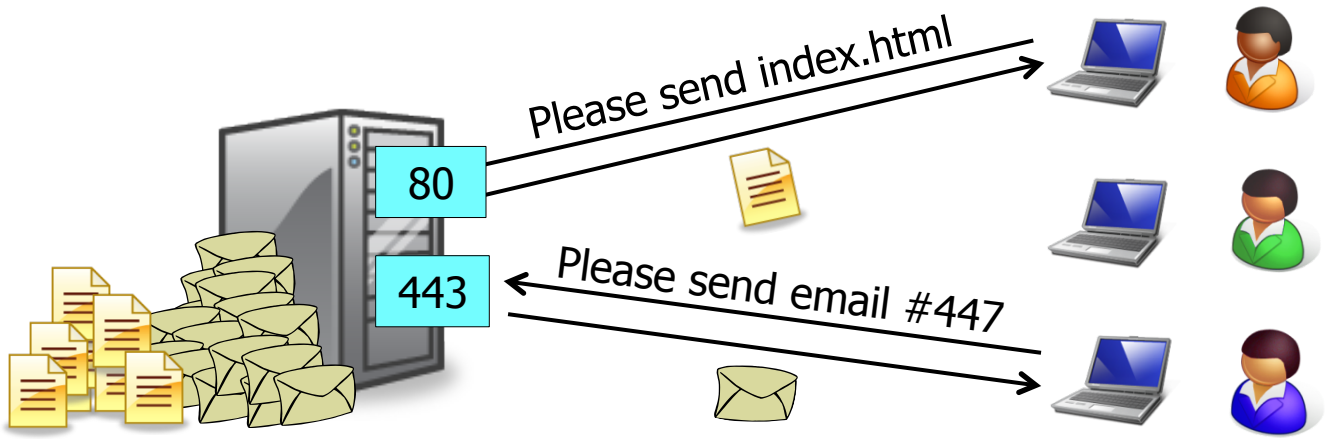
- How the Web actually works today:
  - Some machines (servers) offer documents
  - Other machines (clients) use documents
  - Clients can request documents from servers
  - Model is used for many other services, not just for the web
- Pros and cons of this approach?

Servers



- Server: A machine that offers services to other machines
  - Examples: Mail server, file server, chat server, print server, terminal server, web server, name server, game server, ...
- Protocol often uses request/response pattern

Port numbers and well-known ports

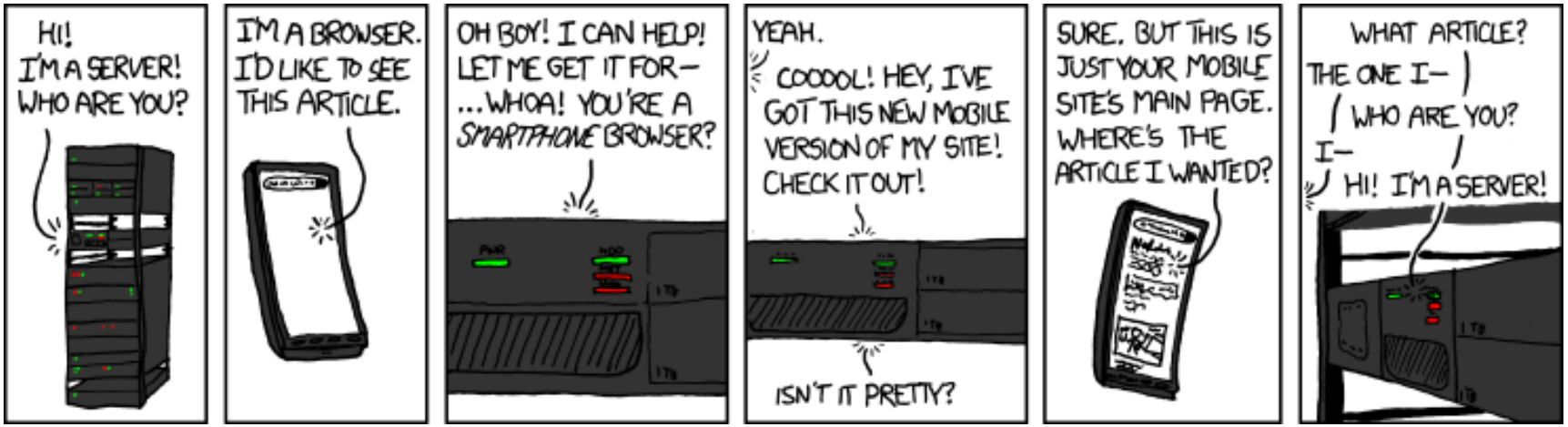


- A single machine can host multiple services
  - Typically distinguished by TCP port number
  - Many services have well-known port numbers, e.g., web servers use port 80, SSH servers use port 22, ...

State, and where to keep it

- What if clients make multiple related requests?
  - Example: Open file, read data, read more data, close file
  - Need to remember some state between requests, e.g., which file was opened, or how much data has already been read
  - Who should keep this state: Client, server, or both?
  - If it is kept on the client, how does the server access it?
  - If it is on the server, how does the client reference it?
- If there is no state, or the client keeps all of it, we can build a stateless server
  - Server can forget everything about completed requests
  - Pros and cons of such a design?

Server attention span



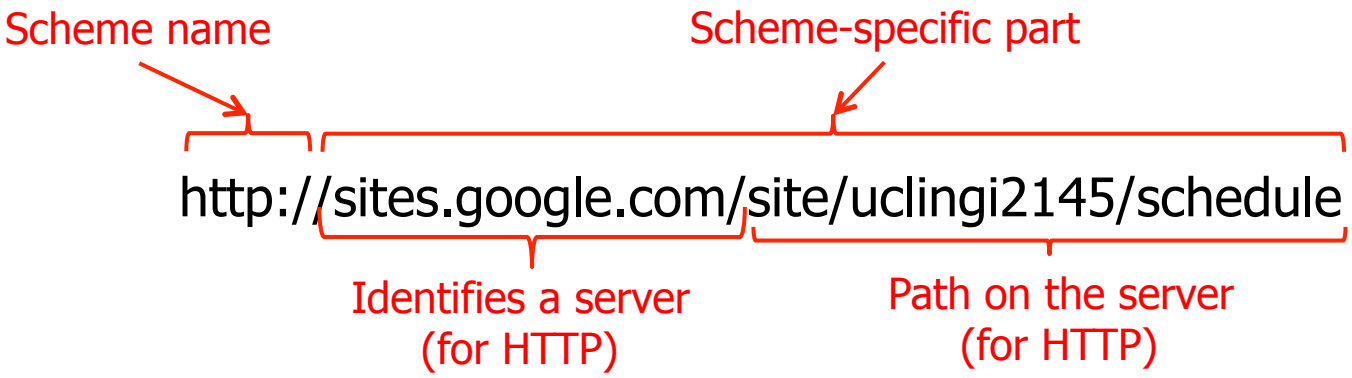
Recap: Client-server model

- Many possible system architectures
  - Examples: Client/server and peer-to-peer
  - Each has its own set of tradeoffs
  - Web uses client/server, but it could have been otherwise
- Client-server model
  - Functionality implemented by special machines
  - Request/response pattern
- State, and where to keep it
  - Could be on the client, on the server, or on both; pros/cons
  - Stateless servers

What do we need to make the Web work?

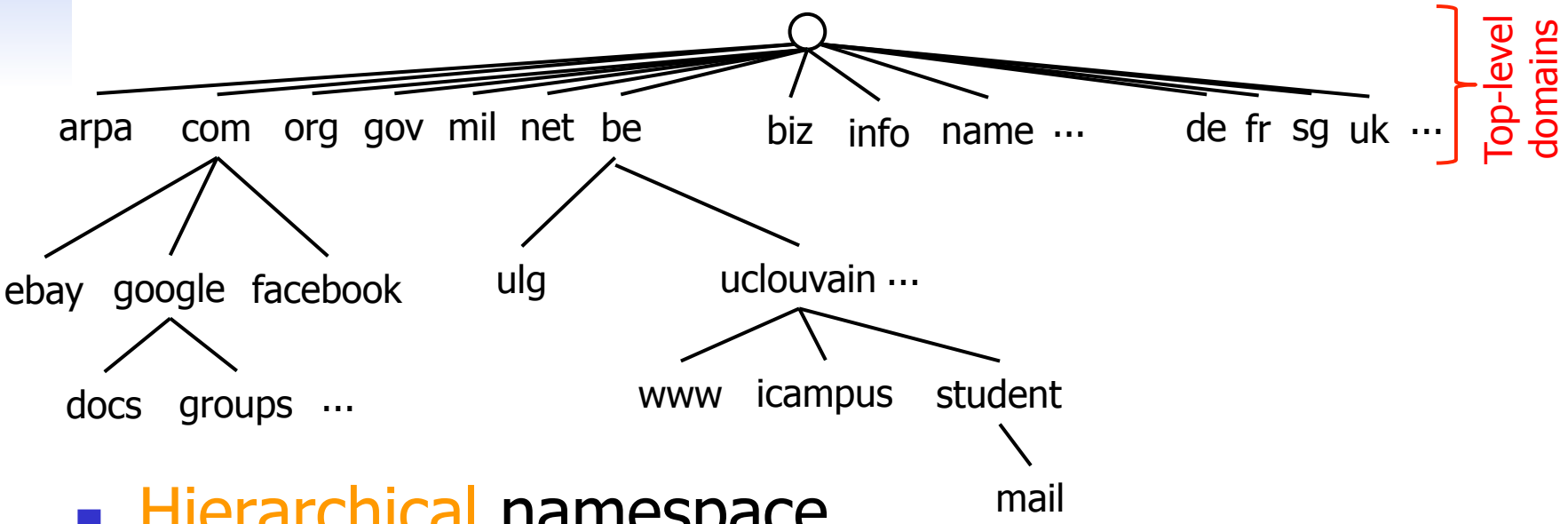
- Formats for writing the documents — HTML
- A program for displaying documents — Browser
- Unique names for the documents — URIs, URLs
- A way to find documents — DNS
- A system for delivering documents
  - Architecture — Client/server model
  - Efficient implementation
- A protocol for transferring documents
- A way to make content dynamic
  - Programming model
  - Keeping state

URIs, URNs, and URLs



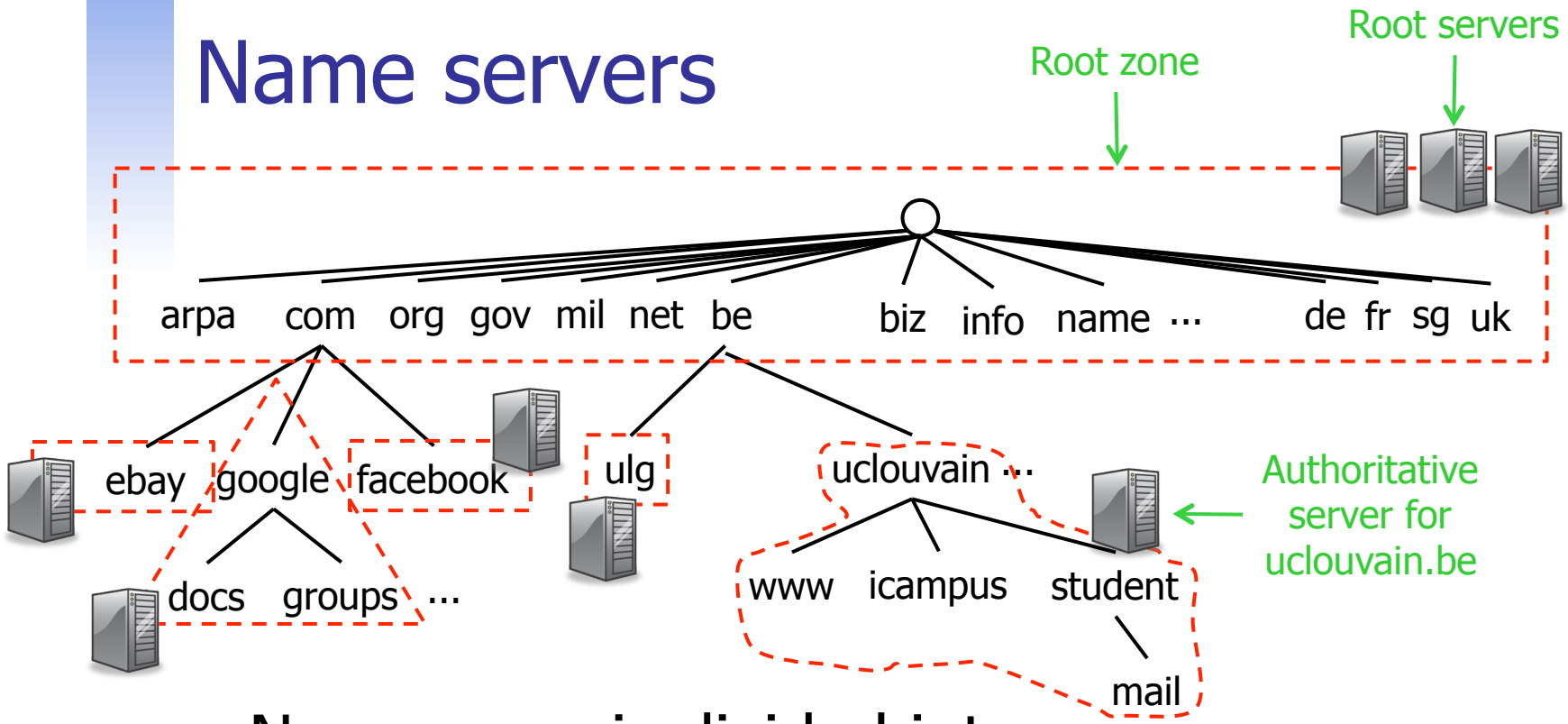
- Uniform Resource Identifier (URI)
  - Comes in two forms: URN and URL
- Uniform Resource Name (URN)
  - Specifies *what* to find, independent of its location
  - Example: urn:isbn:1449311520 (for the course textbook)
- Uniform Resource Locator (URL)
  - Specifies *where* to find something
  - <scheme>://[user[:password]@]<server>[:port]/[path][/resource][?param1=value1&param2=value2&...]

DNS namespace



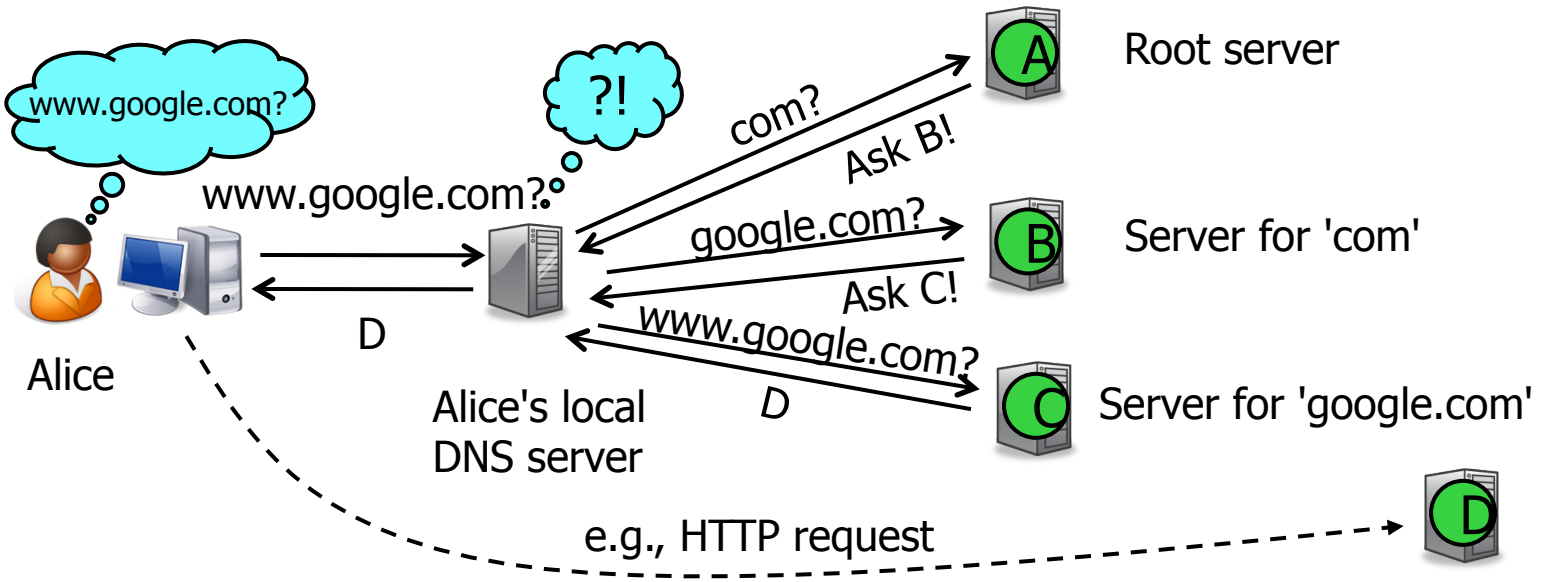
- Hierarchical namespace
  - First level managed by the Internet Corporation for Assigned Names and Numbers (ICANN)
  - Authority over other levels is delegated
    - Second level generally managed by registrars
    - Further levels managed by organizations or individuals
  - Given a DNS name, how can we find the corresponding host?

Name servers



- Namespace is divided into zones
  - TLDs belong to the root zone
- Each zone has an authoritative name server
  - Authoritative server knows, for each name in its zone, which machine corresponds to a given name, or which other name server is responsible

Name resolution in DNS



- Name lookup is recursive
  - Domain name is resolved step by step, starting with the TLD
- Name servers cache results of lookups
  - Why?

Recap: Naming

- Web documents are referenced with URLs
  - URL: Scheme name, domain name, path to document
- DNS maps domain names to machines
  - Hierarchical namespace; root managed by ICANN
  - Distributed system of domain name servers
  - Recursive name lookup
  - Caching

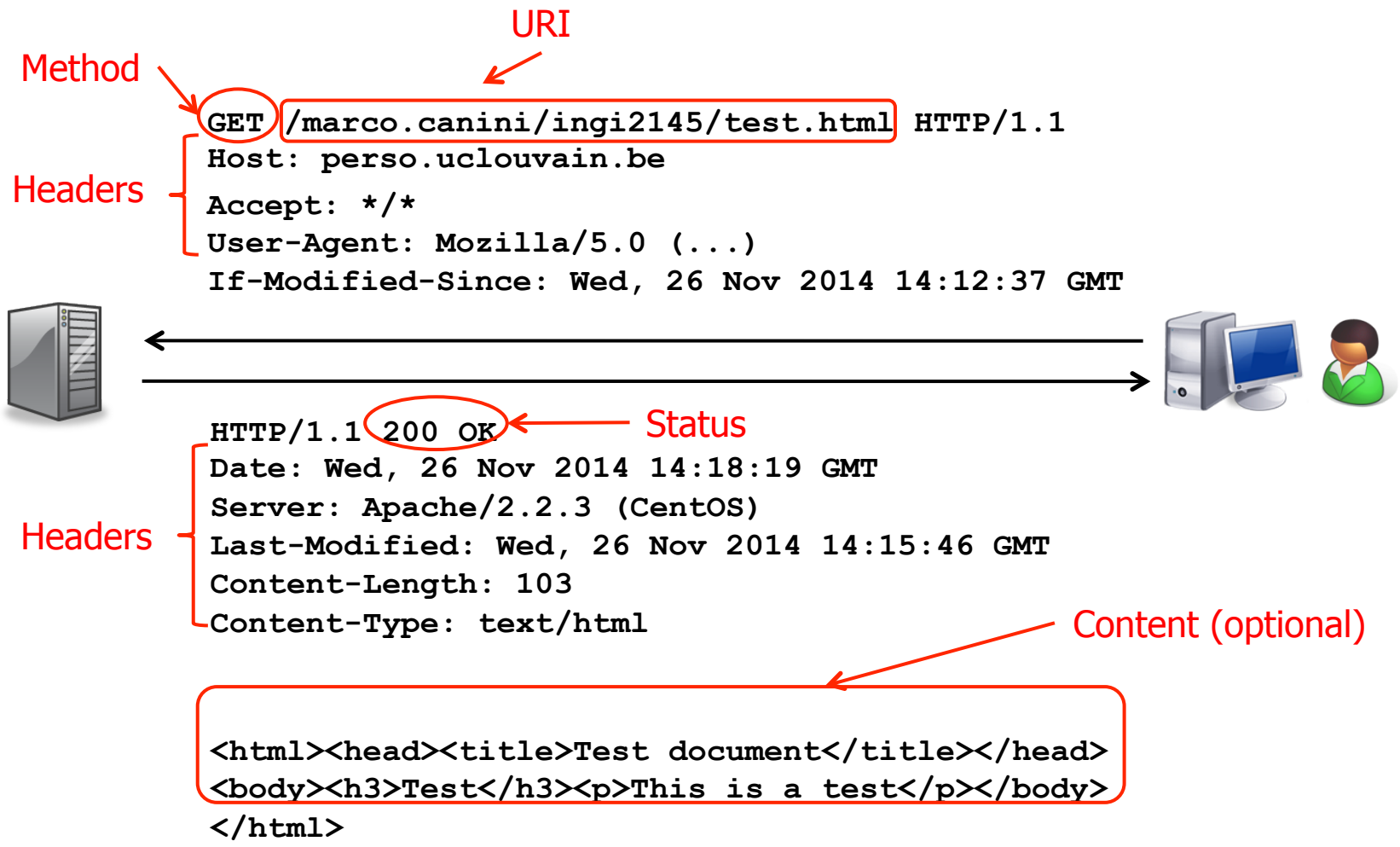
What do we need to make the Web work?

- Formats for writing the documents — HTML
- A program for displaying documents — Browser
- Unique names for the documents — URIs, URLs
- A way to find documents — DNS
- A system for delivering documents
  - Architecture — Client/server model
  - Efficient implementation
- A protocol for transferring documents — HTTP
- A way to make content dynamic
  - Programming model
  - Keeping state

The HTTP protocol

- How to communicate with a web server?
  - Use the HyperText Transfer Protocol (HTTP)
- A very simple protocol
  - First specified in 1990
  - Runs on top of TCP/IP
  - Default port 80 (unsecure), or 443 (secure, with SSL)
  - Originally stateless (HTTP/1.0), but current version (HTTP/1.1) added support for persistent connections
    - Why?
- Development continues (e.g., SPDY)
  - The current proposal for HTTP 2.0 is based on this!

Example: A simple HTTP request

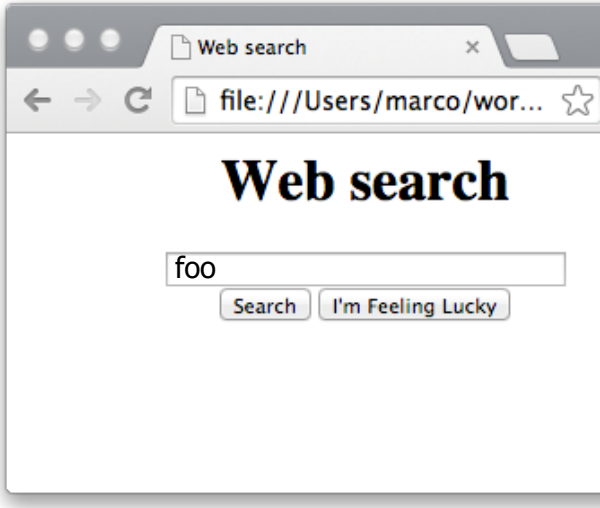


Common HTTP methods

- GET
  - Retrieve whatever information is identified by the URI
- HEAD
  - Like GET, but retrieves only metadata, not the actual object
- PUT
  - Store information under the specified URI
- DELETE
  - Delete the information specified by the URI
- POST
  - Adds new information to whatever is identified by the URI
  - Intended, e.g., for newsgroup posts; today, used mostly to implement dynamic content via forms

Forms and GET/POST

```
<html>
  <head><title>Web search</title></head>
  <body>
    <center><h1>Web search</h1>
    <form action="search.html" method="">
      <input type="text" size="40" name="term"><br>
      <input type="submit" value="Search">
      <input type="button" value="I'm Feeling Lucky">
    </form></center>
  </body>
</html>
```



- What happens when we hit 'Search'?

With method="get":

```
GET /search.html?term=foo HTTP/1.1
Accept: text/html
```

With method="post":

```
POST /search.html HTTP/1.1
Accept: text/html

term=foo
```

GET or POST?

- GET should be used for idempotent requests
  - Requests that are safe to re-execute without side-effects, such as making a purchase or committing an edit
  - Browser warns user when resending a POST, but not a GET
- GET has length restrictions
  - Data is put into the URL, whose length is restricted
- GET should only be used with text
  - POST works with arbitrary data (including binaries)

Headers

```
GET /index.html HTTP/1.1
Host: www.uclouvain.be
Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,
  image/webp,*/*;q=0.8
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_5)
  AppleWebKit/537.36 (KHTML, like Gecko) Chrome/38.0.2125.122 Safari/537.36
Referer: https://www.google.com/
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8

HTTP/1.1 200 OK
Date: Wed, 26 Nov 2014 14:28:13 GMT
Server: Apache/2.2.3 (CentOS)
Set-Cookie: JSESSIONID=62CC31...; Path=/cps
Set-Cookie: CPSSESSIONID_ucl=SID-E...; Path=/
Expires: Wed, 26 Nov 2014 14:28:13 GMT
Connection: Keep-Alive
Content-Type: text/html;charset=UTF-8

<html>...
```

- Both the request and the response can contain headers with additional information

Status codes

- Server sends back a status code to report how the request was processed

- Common status codes:
  - 200 OK
  - 301 Moved Permanently
  - 304 Not Modified
  - 401 Unauthorized
  - 403 Forbidden
  - 404 Not Found
  - 500 Internal Server Error

Recap: HTTP

- HTTP - a simple, stateless protocol
  - Server does not (need to) remember past requests
- Request and response contain headers
  - Used to exchange additional information, e.g., to request content in specific formats, or to exchange metadata
- Common HTTP methods:
  - GET - Retrieve a specific document
  - HEAD - Get metadata for a specific document
  - POST - 'Add' information to a document (used for forms)

What do we need to make the Web work?

- Formats for writing the documents — HTML
- A program for displaying documents — Browser
- Unique names for the documents — URIs, URLs
- A way to find documents — DNS
- A system for delivering documents
  - Architecture — Client/server model
  - Efficient implementation — Threads; event-driven prog.
- A protocol for transferring documents — HTTP
- A way to make content dynamic
  - Programming model
  - Keeping state

A simple web server

```
socket = listen(port 80);
while (true) {
  connection = accept(socket);
  request = connection.read();
  if (request == "GET <document>") {
    data = filesystem.read(document);
    connection.write("HTTP/1.1 200 OK");
    connection.write(data);
  } else {
    connection.write("HTTP/1.1 400 Bad request");
   }
  close(connection);
}
```

- Is this a good (web) server? If not, why not?

The need for concurrency

- What if the server receives lots of requests?
  - Idea #1: Process them serially
  - Problem: Slow client can block everyone else
  - Idea #2: One server for each request
  - Problem: Wasteful
- Server needs to handle requests concurrently
  - Available resources are multiplexed between requests
- How do we do this?
  - Threads, thread pools
  - Events

Refresher: Threads and processes

- Physical machine has some fixed number of processor cores - say, c
  - What if we need more than c threads of execution?
- Idea: Time-share cores
  - A single core works on one thread for a while, then context-switch to another one
  - Switching can be done cooperatively: Each thread yields the core to another thread when it has nothing to do
  - Switching can also be preemptive: Each thread gets to run for a fixed amount of time (quantum, typ. 10-100ms)
  - Pros and cons of preemption?
  - Difference between a thread and a process?

A simple thread-based web server

```
worker(connection) {
  request = connection.read();
  if (request == "GET <document>") {
    data = filesystem.read(document);
    connection.write("HTTP/1.1 200 OK");
    connection.write(data);
  } else {
    connection.write("HTTP/1.1 400 Bad
req");
  }
  close(connection);
}
main() {
  socket = listen(port 80);
  while (true) {
    connection = accept(socket);
    (new Thread).run(worker, connection);
  }
}
```

Worker thread (one per connection)

Main loop (accepts new connections and launches new threads)

Thread-based servers

- Relevant Java constructs:
  - Worker can be a subclass of Thread + implement run()
  - Worker w = new Worker(); w.start();
  - Alternative: Worker implements Runnable
  - Thread t = new Thread(w); t.start();
- What if the threads share some resources?
  - Potential race conditions + consistency issues
  - Can use synchronization and locking
  - Need to be careful about deadlock, livelock, starvation

Thread pools

- Problem: Threads operations are expensive
  - Creating and destroying a thread takes time
  - Context switching between threads takes time
  - Making too many threads can exhaust available resources

- Idea: Keep a thread pool with a fixed number of worker threads
  - When a worker is done handling a request, it is assigned another one
  - When there are more concurrenct requests than workers, requests must wait in a queue until a worker is available
  - When there are few requests, some workers may be idle
  - How many threads should we put into the pool?

Event-driven programming

- What benefits did the threads really give us?
  - If we have one core and run 1,000 threads on it, does the work really get done faster than with one thread?
- The server's work is driven by events, e.g.:
  - Incoming connection: Need to set up data structures
  - Request arrives: Need to parse + start reading document
  - Document read: Need to send back to the client
  - Entire document sent: Close connection, clean up structures
- Why not base server architecture on events?
  - All we need is a single thread!

An event-based web server

```
handleNewConnection(e) { startReading(e.connection); }
handleRequestRead(e) {
  if (e.request == "GET <document>") {
    issueFilesystemRead(document);
  } else {
    issueWrite(e.connection, "HTTP/1.1 400 Bad req");
  }
}
/* other handlers go here */

main() {
  EventQueue q;
  while (true) {
    e = q.getNextEvent();
    case e of {
      NewConnection: handleNewConnection(e);
      RequestArrived: handleRequestRead(e);
      FileReadCompleted: handleFileRead(e);
      AllDataWritten: handleDataWritten(e);
    }
  }
}
```

Event handlers (must not block)

Who puts events into the queue?

Dispatch loop (distributes events to handlers)

Continuations

- What if, in response to some event, we must perform a blocking system call?
  - Example: Request arrives; now we need to read the file from disk (blocking read() call) and send it back to the client
  - What would happen if we called read() in the event handler?
  - Solution: Write two event handlers:
    - Handler A parses the request and issues a non-blocking read (using a special system call)
    - Handler B is called when the read completes and sends data to client
- What if handler A has some state that handler B needs to know?
  - Must be saved explicitly in a continuation

Event-driven programming in Node

```
var getStudents = function(coursename, callback) {
  simpleDB.getAttributes({DomainName:'students', ItemName: coursename},
    function (err, data) {
      if (err) {
        console.log("Error occurred!");
        callback(null);
      } else if (data.Attributes == undefined) {
        console.log("No results!");
        callback(null);
      } else {
        console.log("Got data for course: " + coursename);
        console.log("The data is: "+data);
        callback(data);
      }
    }
  );
};
```

Must always call callback, even if the operation has failed!

Inner callback has access to the state of the outer function!

- Node programs are event-driven
  - If a function needs to block (e.g., network I/O), it must take a callback function and call that function once the blocking operation completes
  - Where is the continuation?

Pros and cons

- Thread-based server or event-driven one?
  - Event-driven servers typically have better performance
  - Event-driven servers do not need as much synchronization
    - Just a single thread - no concurrency!!! (on a single core)
    - However, may need some flags if events can share resources
  - Thread-based servers are typically easier to write+maintain

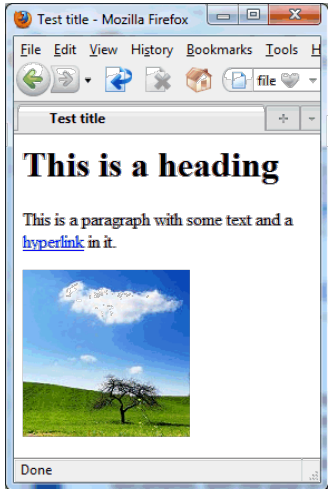
Recap: Web servers

- Need to process requests concurrently
  - Otherwise, extremely difficult to achieve high throughput
- Common server architectures:
  - Single-threaded
  - Multithreaded
  - Thread pools
  - Event-driven
- Event-driven architecture:
  - Harder to program, but very efficient
- Most of this also applies to other kinds of servers, not just to web servers

What do we need to make the Web work?

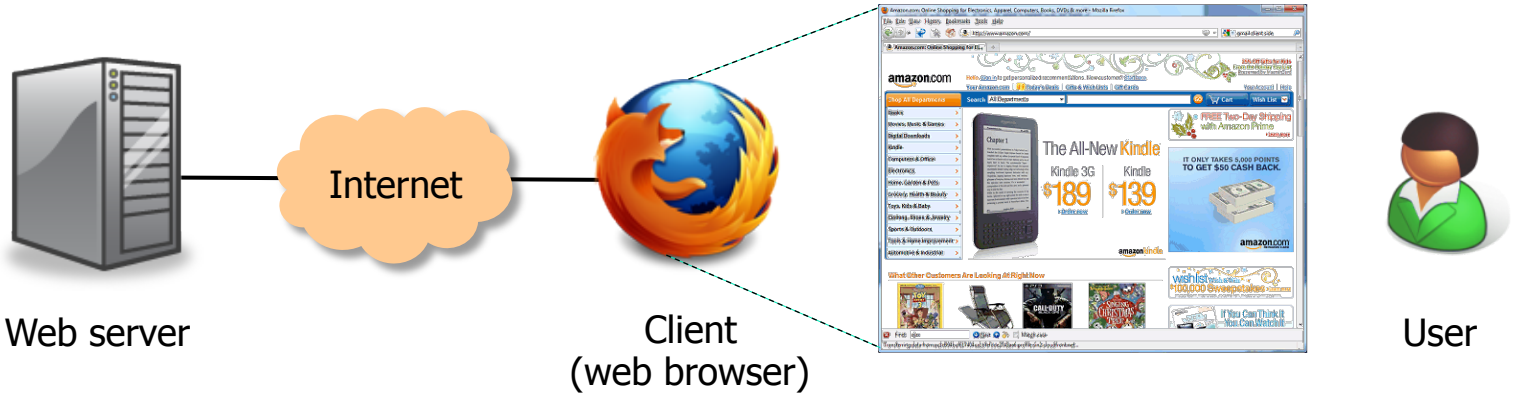
- Formats for writing the documents — HTML
- A program for displaying documents — Browser
- Unique names for the documents — URIs, URLs
- A way to find documents — DNS
- A system for delivering documents
  - Architecture — Client/server model
  - Efficient implementation — Threads; event-driven prog.
- A protocol for transferring documents — HTTP
- A way to make content dynamic
  - Programming model
  - Keeping state

Web applications



- So far: Writing and delivering static content
- But many web pages today are dynamic
  - State (shopping carts), computation (recommendations), rich I/O (videoconferencing), interactivity, ...

Client-side and server-side



- Where does the web application run?
  - Can run on the server, on the client, or have parts on both
    - Modern browsers are highly programmable and can run complex applications (example: client-side part of Google's Gmail)
    - Some believe the browser will be 'the new operating system'
  - Client-side technologies: JavaScript, Java applets, Flash, ...
  - Server-side technologies: CGI, PHP, Java servlets, Node.js, …

Dynamic content

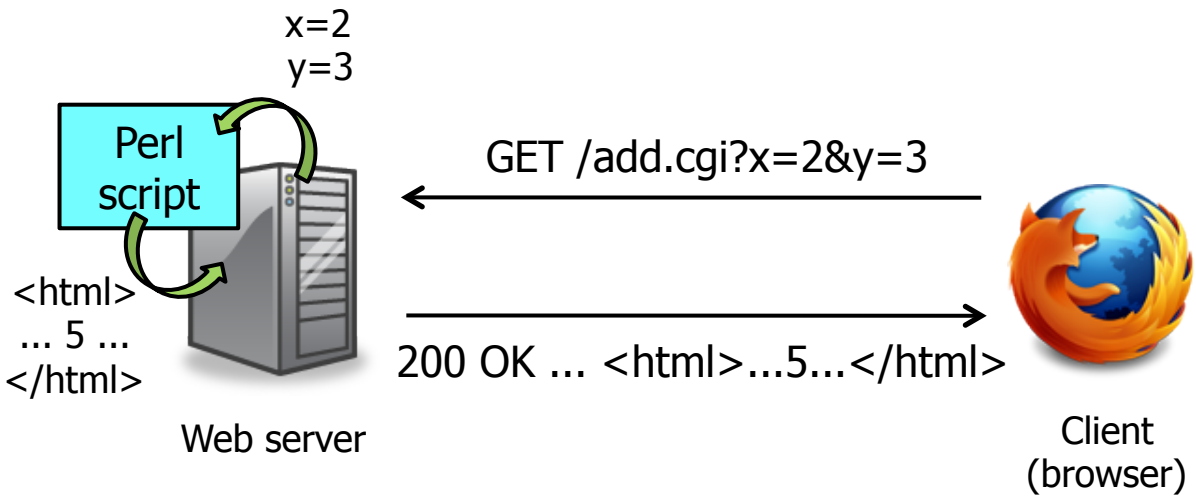
- Web application technologies 

  - Background: CGI
  - Java Servlets
- Node.js / Express / EJS
  - Express framework
  - SimpleDB bindings
  - Example application: Dictionary
- Session management and cookies

Dynamic content

- How can we make content dynamic?
  - Web server needs to return different web pages, depending on how the user interacts with the web application
- Idea #1: Build web app into the web server
  - Why is this not a good idea?
- Idea #2: Loadable modules
  - Is this a good idea?
  - Pros and cons?

CGI



- Common Gateway Interface (CGI)
  - Idea: When dynamic content is requested, the web server runs an external program that produces the web page
  - Program is often written in a scripting language ('CGI script')
  - Perl is among the most popular choices

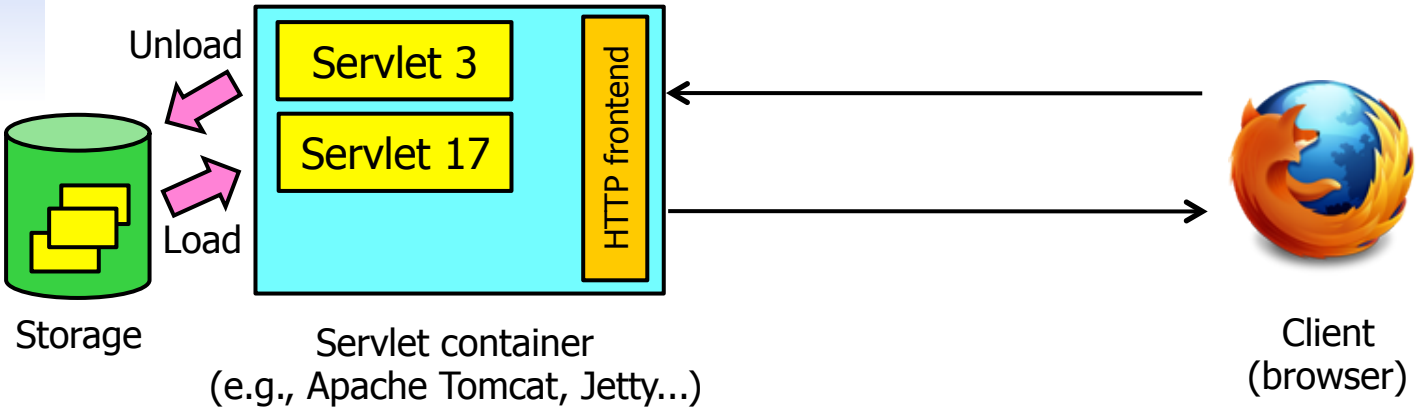
CGI

- A little more detail:
  1. Server receives HTTP request
     - Example: GET /cgi-bin/shoppingCart.pl?user=ahae&product=iPad
  2. Server decides, based on URL, which program to run
  3. Server prepares information for the program
     - Metadata goes into environment variables, e.g., QUERY_STRING, REMOTE_HOST, REMOTE_USER, SCRIPT_NAME, ...
     - User-submitted data (e.g., in a PUT or POST) goes into stdin
  4. Server launches the program as a separate process
  5. Program produces the web page and writes it to stdout
  6. Server reads the web page and returns it to the client

Drawbacks of CGI

- Each invocation creates a new process
  - Time-consuming: Process creation can take much longer than the actual work
  - Inefficient: Many copies of the same code in memory
  - Cumbersome: Must store session state in the file system
- CGIs are native programs
  - Security risk: CGIs can do almost anything; difficult to run third-party CGIs; bugs (shell escapes! buffer overflows!)
  - Low portability: A CGI that runs on one web server may not necessarily run on another
  - However, this can also be an advantage (high speed)

What is a servlet?

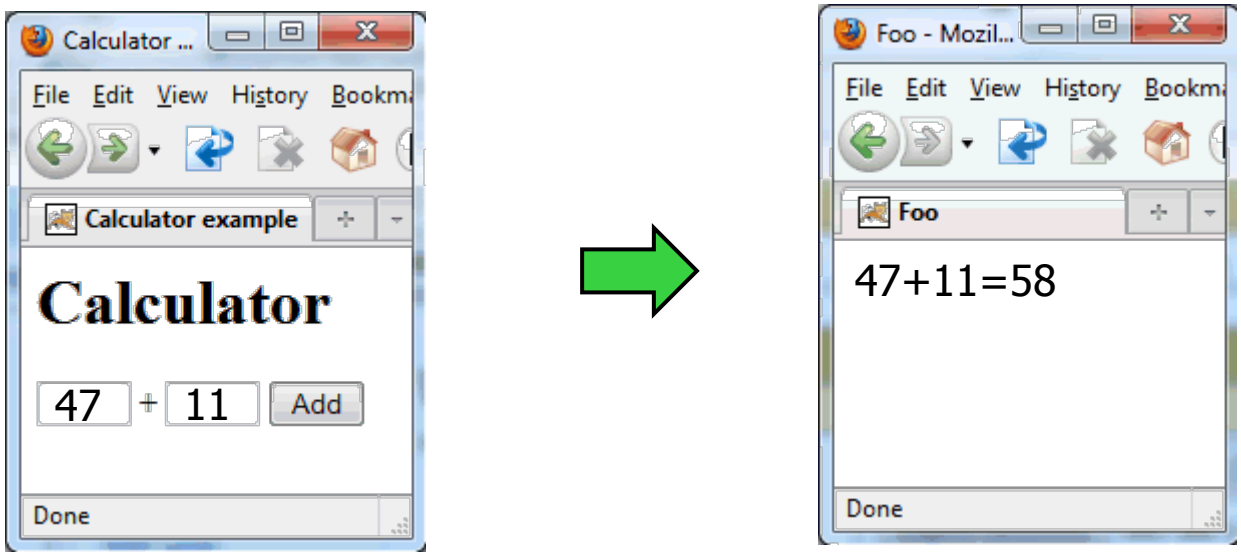


- Servlet: A Java class that can respond to HTTP requests
  - Implements a specific method that is given the request from the client, and that is expected to produce a response
  - Servlets run in a special web server, the servlet container
    - Only one instance per servlet; each request is its own thread
  - Servlet container loads/unloads servlets, routes requests to servlets, handles interaction with client (HTTP protocol), ...

Servlets vs CGI

	CGI	Servlets
Requests handled by	Processes (heavyweight)	Threads (lightweight)
Copies of the code in memory	Potentially many	One
Session state stored in	File system	Servlet container (HttpSession)
Security	Problematic	Handled by Java sandbox
Portability	Varies (many CGIs platform-specific)	Java

A simple example



- Running example: A calculator web-app
  - User enters two integers into a HTML form and submits
    - Result: GET request to calculate?num1=47&num2=11
  - Web app adds them and displays the sum

The Calculator servlet

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class CalculatorServlet extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse
response)
       throws java.io.IOException {
    int v1 = Integer.valueOf(request.getParameter("num1")).intValue();
    int v2 = Integer.valueOf(request.getParameter("num2")).intValue();
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html><head><title>Hello</title></head>");
    out.println("<body>"+v1+"+"+v2+"="+(v1+v2)+"</body></html>");
} }
```

Numbers from the GET request become parameters

- Two easy steps to make a servlet:
  - Create a subclass of HttpServlet
  - Overload the doGet() method
    - Read input from HttpServletRequest, write output to HttpServletResponse
    - Do not use instance variables to store session state! (why?)

Dynamic content

- Web application technologies
  - Background: CGI 
  - Java Servlets 
- Node.js / Express / EJS 

  - Express framework
  - SimpleDB bindings
  - Example application: Dictionary
- Session management and cookies

What is Node.js?

- A platform for JavaScript-based network apps
  - Based on Google's JavaScript engine from Chrome
  - Comes with a built-in HTTP server library
  - Lots of libraries and tools available; even has its own package manager (npm)

- Event-driven programming model
  - There is a single "thread", which must never block
  - If your program needs to wait for something (e.g., a response from some server you contacted), it must provide a callback function

What is JavaScript?

- A widely-used programming language
  - Started out at Netscape in 1995
  - Widely used on the web; supported by every major browser
  - Also used in many other places: PDFs, certain games, ...
  - ... and now even on the server side (Node.js)!

- What is it like?
  - Dynamic typing, duck typing
  - Object-based, but associative arrays instead of 'classes'
  - Prototypes instead of inheritance
  - Supports run-time evaluation via eval()
  - First-class functions

What is Express?

- Express is a minimal and flexible framework for writing web applications in Node.js
  - Built-in handling of HTTP requests
  - You can tell it to 'route' requests for certain URLs to a function you specify
    - Example: When /login is requested, call function handleLogin()
  - These functions are given objects that represent the request and the response, not unlike Servlets
  - Supports parameter handling, sessions, cookies, JSON parsing, and many other features
  - API reference: http://expressjs.com/api.html

```
var express = require('express');
var app = express();

app.get('/', function(req, res) {
  res.send('hello world');
});

app.listen(3000);
```

The Request object

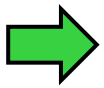
req.param(name)	Parameter 'name', if present
req.query	Parsed query string (from URL)
req.body	Parsed request body
req.files	Uploaded files
req.cookies.foo	Value of cookie 'foo', if present
req.get(field)	Value of request header 'field'
req.ip	Remote IP address
req.path	URL path name
req.secure	Is HTTPS being used?
...	

The Response object

- req.status(code) Sets status 'code' (e.g., 200)
- req.set(n,v) Sets header 'n' to value 'v'
- res.cookie(n,v) Sets cookie 'n' to value 'v'
- res.clearCookie(n) Clears cookie 'n'
- res.redirect(url) Redirects browser to new URL
- res.send(body) Sends response (HTML, JSON...)
- res.type(t) Sets Content-type to t
- res.sendfile(path) Sends a file
- ...

What is Embedded JS (EJS)?

```
app.get('/', function(req, res) {
  res.send('<html><head><title>'+
    'Lookup result</title></head>'+
    '<body><h1>Search result</h1>'+
    req.param('word')+' means '+
    +lookupWord(req.param('word')));
  );
});
```

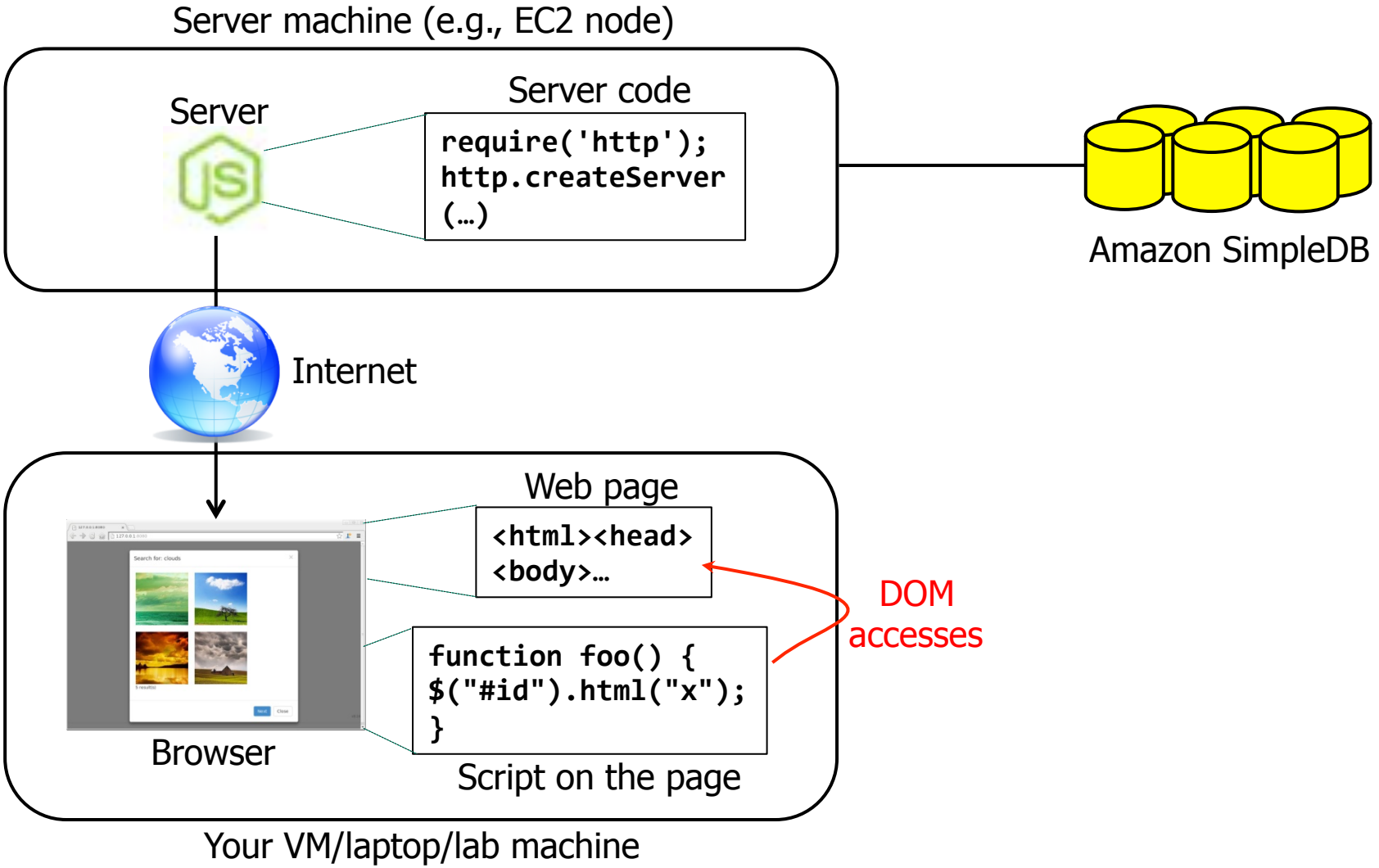


```
...
 w = req.param('word');
 res.render('results.ejs',
  {blank1:w, blank2:lookupWord(w)});
```

```
<html><head><title>Lookup result</title>
</head><body><h1>Search result</h1>
<% =blank1 %> means <% =blank2 %>
```

- We don't want HTML in our JavaScript code!
- EJS allows you to write 'page templates'
  - You can have 'blanks' in certain places that can be filled in by your program at runtime
  - <% =value %> is replaced by variable 'value' from the array given to render()
  - <% someJavaScriptCode() %> is executed
    - Can do conditionals, loops, etc.

How do the pieces fit together?



How to structure the app

- Your web app will have several pieces:
  - Main application logic
  - 'Routes' for displaying specific pages (/login, /main, ...)
  - Database model (get/set functions, queries, ...)
  - Views (HTML or EJS files)
- Suggestion: Keep them in different directories
  - routes/ for the route functions
  - model/ for the database functions
  - views/ for the HTML pages and EJS templates
  - Keep only app.js/package.json/config... in main directory

"Hello world" with Node/Express

```
var express = require('express');
var bodyParser = require('body-parser');
var morgan = require('morgan'); // logger
var routes = require('./routes/routes.js');
var app = express();
app.use(bodyParser.urlencoded({ extended: true }));
app.use(morgan('combined'))

app.get('/', routes.get_main);
app.post('/results', routes.post_results);
app.listen(8080);
console.log('Server running on port 8080');
```

app.js

```
<html><body>
  <h1>Dictionary lookup</h1>
  <form action="/results" method="post">
    <input type="text" name="myInputField">
    <input type="submit" value="Search">
  </form>
</body></html>
```

views/main.ejs

```
<html><body>
  <h1>Lookup results</h1>
  You searched for: <%= theInput %><p>
  <a href="/">Back to search</a>
</body></html>
```

views/results.ejs

```
var getMain = function(req, res) {
  res.render('main.ejs', {});
};

var postResults = function(req, res) {
  var x = req.body.myInputField;
  res.render('results.ejs', {theInput: x});
};

var routes = {
  get_main: getMain,
  post_results: postResults
};

module.exports = routes;
```

routes/routes.js

```
{
  "name": "HelloWorld",
  "description": "Demo",
  "version": "0.0.1",
  "dependencies": {
     "express": "*",
 "body-parser": "*",
     "morgan": "*",
     "ejs": "*"
   }
}
```

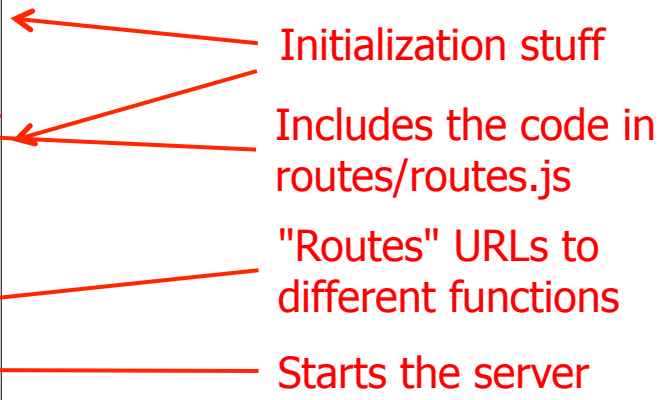
package.json

The main application file

```
var express = require('express');
var bodyParser = require('body-parser');
var morgan = require('morgan'); // logger
var routes = require('./routes/routes.js');
var app = express();
app.use(bodyParser.urlencoded({ extended: true }));
app.use(morgan('combined'))

app.get('/', routes.get_main);
app.post('/results', routes.post_results);
app.listen(8080);
console.log('Server running on port 8080');
```

app.js



- What is going on here?
  - app.js is the "main" file (you run "nodejs app.js" to start)
  - Does some initialization stuff and starts the server
- Key element: URL routing
  - "If you receive a POST http://localhost/results request, call the function routes.post_results to handle it"
  - Need one such line for each 'page' our web application has

The request handlers (routes)

```
var getMain = function(req, res) {
  res.render('main.ejs', {});
};

var postResults = function(req, res) {
  var x = req.body.myInputField;
  res.render('results.ejs', {theInput: x});
};

var routes = {
  get_main: getMain,
  post_results: postResults
};

module.exports = routes;
```

Simply displays a page

Extract POSTed form data from request (req)

Display a page with the 'theInput' blank filled in

Makes a 'class' that contains all the request handlers we've defined here

Exports the 'class'

routes/routes.js

- Defines a 'request handler' for each page
  - Has access to the HTTP request (req), e.g., for extracting posted data, and to the response (res) for writing output
  - The .ejs pages are normal HTML pages but can have 'blanks' in them that we can fill with data at runtime
  - Need a new page? Just add a new handler!

The page templates

```
<html><body>
  <h1>Dictionary lookup</h1>
  <form action="/results" method="post">
    <input type="text" name="myInputField">
    <input type="submit" value="Search">
  </form>
</body></html>
```

views/main.ejs

```
<html><body>
  <h1>Lookup results</h1>
  You searched for: <%= theInput %><p>
  <a href="/">Back to search</a>
</body></html>
```

views/results.ejs

- The .ejs files are 'templates' for HTML pages
  - Don't want to 'println()' the entire page (messy!)
  - Instead, you can write normal HTML with some 'blanks' that can be filled in by the program at runtime
  - Syntax for the blanks: <%= someUniqueName %>
  - Values are given as the second argument of render(), which is basically a mapping from unique names to values
  - See also http://embeddedjs.com/getting_started.html and http://code.google.com/p/embeddedjavascript/w/list

The package manifest

```
{
  "name": "HelloWorld",
  "description": "Demo",
  "version": "0.0.1",
  "dependencies": {
     "express": "*",
 "body-parser": "*",
     "morgan": "*",
     "ejs": "*"
   }
}
```

package.json

- Contains some metadata about your web app
  - Name, description, version number, etc.
- ... including its dependencies
  - Names of the Node modules you are using, and the required versions (or '*' to designate 'any version')
  - Once you have such a file, you can simply use 'npm install' to download all the required modules!
    - No need to ship node_modules with your app

Let's add some real data!

```
<!DOCTYPE html>
<html>
<body>
  <h1>Lookup results</h1>
  You searched for: <%= theInput %><p>
  <%if (result != null) { %>
  Translation: <%= result %><p>
  <% } %>
  <%if (message != null) { %>
  <font color="red"><%= message %><p>
  <% } %>
  <a href="/">Back to search</a>
</body>
</html>
```

Our extra 'blank' for the translation

Conditional (works because of EJS)

views/results.ejs

- Let's show translations of the words
  - Simple add a new 'blank' to the results.ejs page template
  - But what if no result was found, or an error occurred?
  - Add conditionals to only show the result and error elements when there is actually something to be shown

Database schema and model

- We need a database to store the translations
  - We'll use SimpleDB for this
  - Let's store English->German and English->French
- What would be a good way to keep this data?
  - How many tables are needed?
  - What data will they contain?
  - Which columns will they have?
  - This is called a 'schema'

ItemName	German	French
apple	Apfel	pomme
pear	Birne	poire

- How will your program access the data?
  - BAD: Hard-code SimpleDB calls everywhere
  - GOOD: Write a 'model' with wrapper functions, like lookup(term,language), addWord(term,translation,lang), ...

Accessing the database

```
var AWS = require('aws-sdk');
AWS.config.loadFromPath('config.json');
var simpledb = new AWS.SimpleDB();

var myDB_lookup = function(term, language, callback){
  simpledb.getAttributes({DomainName:'words', ItemName: term}, function (err, data) {
    if (err) {
      callback(null, "Lookup error: "+err);
    } else if (data.Attributes == undefined) {
      callback(null, null);
    } else {
      var results = {};
      for (i = 0; i<data.Attributes.length; i++) {
        if (data.Attributes[i].Name === language)
          results.translation = data.Attributes[i].Value;
      }
      callback(results, null);
    }
  });
};

var database = {
  lookup: myDB_lookup
};

module.exports = database;
```

models/simpleDB.js

```
{
  "name": "HelloWorld",
  "description": "Demo",
  "version": "0.0.1",
  "dependencies": {
     "express": "~3.3.5",
     "ejs": "*",
     "aws-sdk": "*"
   }
}
```

package.json

```
{
  "accessKeyId": "yourAccessKeyIDhere",
  "secretAccessKey": "yourSecretKeyhere",
  "region": "us-east-1"
}
```

config.json

SimpleDB API

createDomain	Creates a new domain
deleteDomain	Deletes a domain
listDomains	Lists all of current user's domains
domainMetadata	Returns information about domain
putAttributes	Creates or replaces attr. of item
getAttributes	Returns attributes of item
deleteAttributes	Deletes attributes from item
select	Returns attributes matching expr.
batchDeleteAttributes	Multiple DeleteAttributes
batchPutAttributes	Multiple PutAttributes

- See also: http://docs.aws.amazon.com/AWSJavaScriptSDK/latest/frames.html

Doing the actual lookups

```
var db = require('../models/simpleDB.js');

var getMain = function(req, res) {
  res.render('main.ejs', {});
};

var postResults = function(req, res) {
  var userInput = req.body.myInputField;
  db.lookup(userInput, "german", function(data, err) {
    if (err) {
      res.render('results.ejs',
       {theInput: userInput, message: err, result: null});
    } else if (data) {
      res.render('results.ejs',
       {theInput: userInput, message: null, result: data.translation});
    } else {
      res.render('results.ejs',
       {theInput: userInput, result: null, message: 'We did not find anything'});
    }
  });
};

var routes = {
  get_main: getMain,
  post_results: postResults
};

module.exports = routes;
```

Include the database code

Database lookup, needs a callback that will receive results (or error)

Fill in multiple 'blanks'

routes/routes.js

Loading the data

```
var AWS = require('aws-sdk');
AWS.config.loadFromPath('./config.json');
var simpledb = new AWS.SimpleDB();
var async = require('async');
var words = [{English:'apple', German:'Apfel', French:'pomme'},
             {English:'pear',  German:'Birne', French:'poire'}];

simpledb.deleteDomain({DomainName:'words'},
  function(err, data) {
    if (err) {
      console.log("Cannot delete: "+err);
    } else {
      simpledb.createDomain({DomainName:'words'}, function(err, data) {
        if (err) {
          console.log("Cannot create: "+err);
        } else {
          async.forEach(words, function(w, callback) {
            simpledb.putAttributes({DomainName:'words', ItemName:w.English,
              Attributes: [{Name:'german', Value:w.German},
                           {Name:'french', Value:w.French}]},
              function(err, data) {
                if (err)
                  console.log("Cannot put: "+err);
                callback();
              });
          });
        }
      });
    }
  });
```

loader.js

Parameters in Express

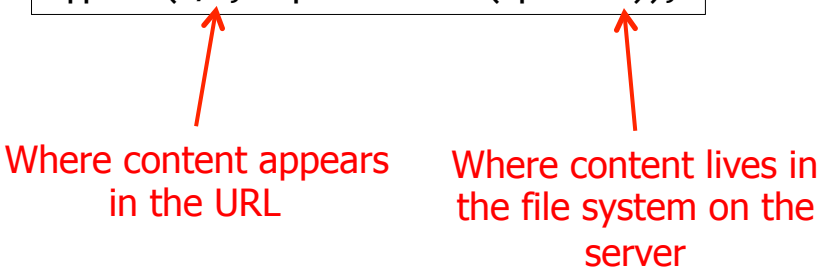
```
app.param('id', /^\d+$/);

app.get('/user/:id', function(req, res) {
  res.send('user ' + req.params.id);
});
```

- Express can automatically parse parameters from a given URL
  - Syntax: /your/url/here/:paramName
  - Available to your function as req.params.paramName
  - Can have more than one, e.g., /user/:uid/photos/:file
- Parameters can also be validated
  - app.param('name', regEx)

Serving static content

```
app.use('/', express.static("public"));
```



- Your web app will probably have static files
  - Examples: Images, client-side JavaScript, ...
- Writing an app.get(...) route every time would be too cumbersome
- Solution: express.static

Dynamic content

- Web application technologies
  - Background: CGI 
  - Java Servlets 
- Node.js / Express / EJS
  - Express framework 
  - SimpleDB bindings 
  - Example application: Dictionary 
- Session management and cookies 

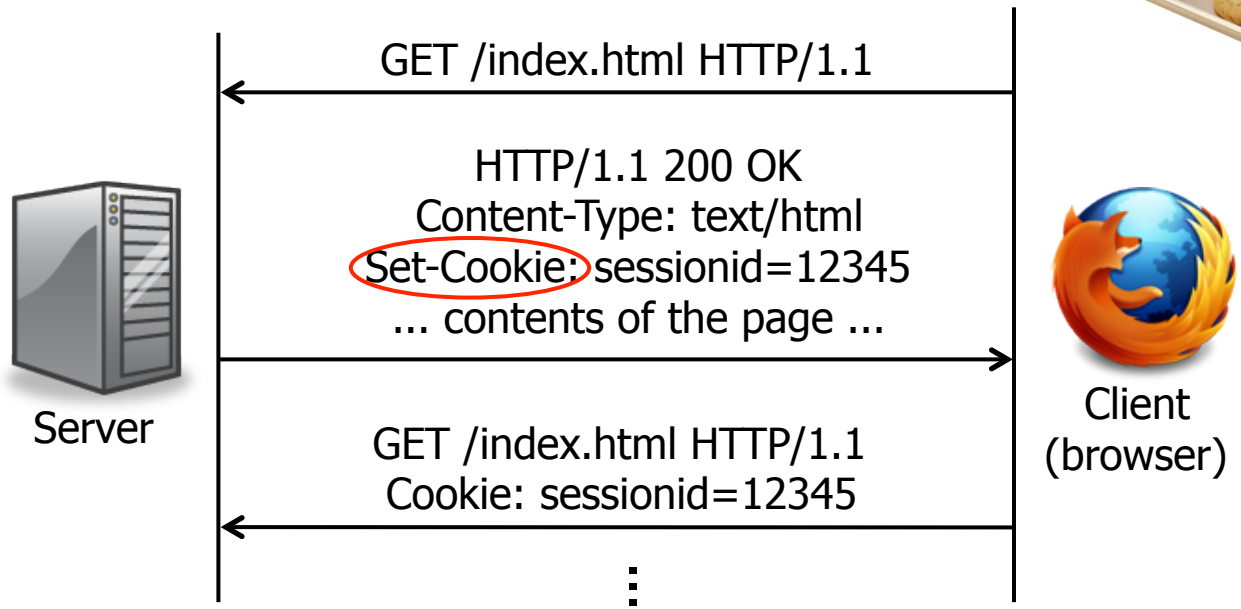
Client-side vs server-side (last time)

- What if web app needs to remember information between requests in a session?
  - Example: Contents of shopping cart, login name of user, ...
- Recap from last time: Client-side/server-side
  - Even if the actual information is kept on the server side, client still needs some kind of identifier (session ID)
- Now: Discuss four common approaches
  - URL rewriting and hidden variables
  - Cookies
  - Session object

URL rewriting and hidden variables

- Idea: Session ID is part of every URL
  - Example 1: http://my.server.com/shoppingCart?sid=012345
  - Example 2: http://my.server.com/012345/shoppingCart
  - Why is the first one better?
- Technique #1: Rewrite all the URLs
  - Before returning the page to the client, look for hyperlinks and append the session ID
    - Example: <a href="foo.html"> → <a href="foo.html?sid=012345">
    - In which cases will this approach not work?
- Technique #2: Hidden variables
  - <input type="hidden" name="sid" value="012345">
  - Hidden fields are not shown by the browser

HTTP cookies



- What is a cookie?
  - A set of key-value pairs that a web site can store in your browser (example: 'sessionid=12345')
  - Created with a Set-Cookie header in the HTTP response
  - Browser sends the cookie in all subsequent requests to the same web site until it expires

Node solution: express.session

```
var cookieParser = require('cookie-parser')
var session = require('express-session')
app.use(cookieParser());
app.use(session({secret: 'thisIsMySecret'});
...
app.get('/test', function(req, res) {
  if (req.session.lastPage)
    req.write('Last page was: '+req.session.lastPage);
  req.session.lastPage = '/test';
  req.send('This is a test.');
}
```

- Abstracts away details of session management
  - Developer only sees a key-value store
  - Behind the scenes, cookies are used to implement it
  - State is stored and retrieved via the 'req.session' object

A few more words on cookies

```
...
Set-Cookie: sessionid=12345;
            expires=Tue, 02-Nov-2010 23:59:59 GMT;
            path=/;
            domain=.mkse.net
...
```

- Each cookie can have several attributes:
  - An expiration date
    - If not specified, defaults to end of current session
  - A domain and a path
- Browser only sends the cookies whose path and domain match the requested page
  - Why this restriction?

What are cookies being used for?

- Many useful things:
  - Convenient session management (compare: URL rewriting)
  - Remembering user preferences on web sites
  - Storing contents of shopping carts etc.
- Some problematic things:
  - Storing sensitive information (e.g., passwords)
  - Tracking users across sessions & across different web sites to gather information about them

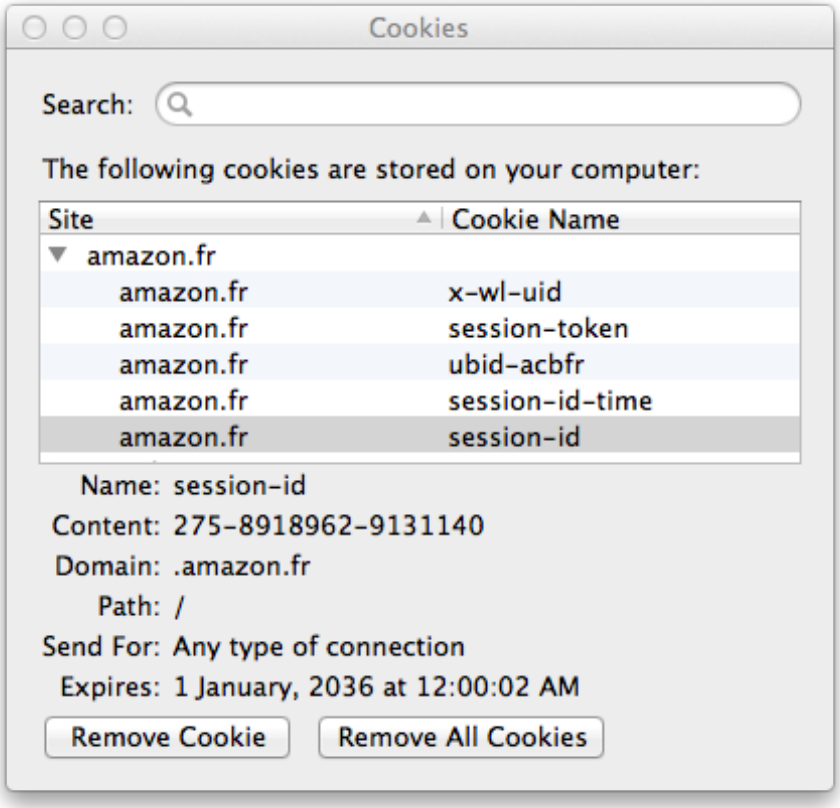
The DoubleClick cookie

For the Google Display Network, we serve ads based on the content of the site you view. For example, if you visit a gardening site, ads on that site may be related to gardening. In addition, we may serve ads based on your interests. As you browse websites that have partnered with us or Google sites using the DoubleClick cookie, such as YouTube, Google may place the DoubleClick cookie in your browser to understand the types of pages visited or content that you viewed. Based on this information, Google associates your browser with relevant interest categories and uses these categories to show interest-based ads. For example, if you frequently visit travel websites, Google may show more ads related to travel. Google can also use the types of pages that you have visited or content that you have viewed to infer your gender and the age category you belong to. For example, If the sites that you visit have a majority of female visitors (based on aggregated survey data on site visitation), we may associate your cookie with the female demographic category.

(Source: http://www.google.com/privacy_ads.html)

- Used by the Google Display Network
  - DoubleClick used to be its own company, but was acquired by Google in 2008 (for $3.1 billion)
- Tracks users across different visited sites
  - Associates browser with 'relevant interest categories'

Cookie management in the browser



- Firefox: Tools/Options/Privacy/Show Cookies
- Chrome: Settings/Privacy/Content settings/All cookies and site data

The Evercookie

- Arms race:
  - Advertisers want to track users
  - Privacy-conscious users do not want to be tracked
- What if users simply delete cookies?
  - Most browsers offer convenient dialogs and/or plugins
  - But: Cookies are not the only way to store data in browsers
- Recent development: The 'evercookie'
  - Stores cookie in eight separate ways: HTTP cookies, Flash cookies, force-cached PNGs, web history (!), HTML5 session storage, HTML5 local storage, HTML5 global storage, HTML5 database storage
  - If any of the eight survives, it recreates the others

http://www.schneier.com/blog/archives/2010/09/evercookies.html

Recap: Session management, cookies

- Several ways to manage sessions
  - URL rewriting, hidden variables, cookies...
- HttpSession
  - Abstract key-value store for session state
  - Implemented by the servlet container, e.g., with URL rewriting or with cookies
- Cookies
  - Small pieces of data that web sites can store in browsers
  - Cookies can persist even after the browser is closed
  - Useful for many things, but also for tracking users