department of computer science
faculty of sciences

Internet Programming

Web Programming

The World-Wide Web

General Philosophy

- Worldwide information sharing
- **Simplicity of use**: at last, an Internet tool for non-computer scientists!
- **Interoperability**: All machines must be able to communicate
  - Different machine architectures
  - Different operating systems
  - Different servers and browsers
  - There is strong need for standards!
- General model: **Client-Server architecture**

Main Actors

- **CERN**
  - Invented the Web: Tim Berners-Lee and Robert Caillaux, 1990
- **NCSA** (National Center for Supercomputing Applications)
  - Mosaic, 1993
- **W3C** (WWW Consortium)
  - 383 members (September 2013)
  - Issues protocol "recommendations": HTML, HTML5, XML, CSS, RDF, SMIL, PICS, etc.
  - Unifies development (e.g., by reference implementations, etc.)
- **IETF** (Internet Engineering Task Force)
  - Defines standards: HTTP/1.1, MIME
- Google, Apple, Microsoft, Sun, Netscape, IBM, etc.
  - …de-facto standards!
  - Interoperability problems

Documents vs. Pages

- A **document** is a data file
  - Text **or** image **or** video **or** postscript **or** PDF **or** …
  - Every document has a **type**
  - Some types have hypertext functionalities: HTML, PDF, Flash, …
- A **page** is what a user sees on the screen
  - Consists of one or more documents
  - Text **and** embedded images/videos **and** sound **and** …
- Attention: These terms are not standardized
  - Many people do not differentiate page and document
  - Some people swap their meaning
  - For this course we will stick to these definitions

Request Model

- Major functionality of the Web
  - "Give me the content of this document"
- The retrieval unit is the **document**
  - Retrieving an HTML page containing ***n*** images requires ***n+1*** requests to the web server
  - The documents are then assembled to form a page
- Each document is designated by a **URL** (Universal Resource Locator)

Dynamic Content

- Some documents are fundamentally **dynamic**
  - Search engines, page counters, etc.
  - Pages tailored to users' preferences (amazon.com)
  - Social Networks
- Some documents are fundamentally **static**, but it is simpler to generate them dynamically
  - Information web sites have 10,000+ pages, high update frequency
  - Wikis, Blogs, etc.
  - Editing web pages in pure HTML takes a lot of time and is error prone
- Solution: Separate the **content** from the **presentation**
  - The raw content is stored in a database and a program generates web pages on the fly
- Execution of code can take place at the **server** and/or at the **client**

Execution at the Server

- When the server receives certain requests, it does not deliver a file
  - Several mechanisms are used to dynamically generate pages

- **CGI** (Common Gateway Interface)
  - When the server receives a request, **it executes a program** provided by the information provider
  - Any executable program is OK (written in any language)
  - The output of the program is sent to the client as a normal document

- Scripting languages, e.g., PHP, Python, Perl, etc.
  - The web server contains a language interpreter
  - It **interprets** a document/program, the result is the response sent to the client

Execution at the Client

- Clients download a document which is a **program**, and execute it locally

- The program must be executable on any machine and any operating system
  - **Applets**: Programs compiled (in advance) in a bytecode language
    - Java Applets
    - An HTML document references the bytecode file
    - The browser downloads it and executes it locally
  - **Scripting languages** (usually Javascript)
    - Javascript instructions are written in the HTML document
    - **Executed locally by the browser**

- **Do not confuse server-side and client-side scripting languages!**

Web Standards

Which **Standard** for which **Function**?

- In which language must I write my documents? How do I represent data?
  - HTML (or XML, …)
- How does a user indicate which document he wants to browse?
  - URL (Uniform Resource Locator)
- How does the browser fetch the document?
  - HTTP (HyperText Transfer Protocol)
- How does the browser know how to display the document?
  - MIME (Multipurpose Internet Mail Extensions)

URL

- URL (Uniform Resource Locator)
  - A standard for designating documents of any type
- Defined in RFC 1738
- Format of a URL: <protocol_id>:<address>
  - mailto:spyros@cs.vu.nl
  - news:comp.os.research
- For many protocols, the <address> part is standardized as **<protocol_id>://<machine>**[**:<port>**]**/<path>**
  - HTTP: http://www.w3.org/Addressing
  - HTTPS: https://www.abnamro.nl/nl/dashboard/overview.html
  - FTP: ftp://ftp.cs.vu.nl/pub/VU_logo.png
  - RMI: rmi://flits.cs.vu.nl/Calculator
  - GOPHER: gopher://gopher.cs.vu.nl/DoesNotExist

MIME [1/2]

- MIME (Multipurpose Internet Mail Extensions)
  - Means to associate a **type** and an **encoding** with a document
  - Gives a meaning to a set of bytes
  - "How should I display this?"
- Defined in RFC 1521 and 1522
- Originally: a standard for email
  - Attached documents must have a type so that the mail client can display them accordingly
- The Web uses a (small) subset of MIME
  - Type and encoding

MIME [2/2]

- Each document is delivered together with its MIME encoding and type

- **Encoding**
  - Examples: **base64**, **x-gzip**
  - If necessary, the document is decoded (base64-decode, gunzip)

- **Type**
  - Examples: **text/plain**, **text/html**, **image/gif**, **application/pdf**, etc.
  - Three possible outcomes:
    - The browser knows it and has internal display functions (HTML, GIF, JPG, …)
      - The document is displayed internally
    - The browser knows which application can handle it (PDF, Word, …)
      - The document is displayed in an external viewer
    - The browser does not know it
      - The document is saved in a file

HTTP

- **HTTP** (HyperText Transfer Protocol)
  - The favorite communication protocol of the Web
  - There are others: **FTP**, **HTTPS**, etc.
- Versions
  - HTTP/1.1: complex, described in RFC 2616, current
  - HTTP/1.0: simple, described in RFC 1945, obsolete
- RPC-like protocol:
  - Connection
  - The client sends a request to the server
  - The server answers the request
  - Disconnection
  - Start again if there are more documents to fetch
- TCP connection initiated by client
  - By default on port 80 at the server
  - Otherwise, the one indicated in the URL

HTTP: Request Phase

- Format of a request

```
<Method> <Path> <HTTP_version> ←
<Optional_field>: <Value> ←
←
```

- Methods: GET, POST, HEAD, PUT, DELETE, TRACE, OPTIONS, …

- Optional fields:
  - **User-Agent**: identification of the browser
  - **Host**: indicate the server host name (user for virtual hosts)
  - **If-Modified-Since**: transfer the document only if it has been modified
  - **Authorization**: username + password (“basic authentication”)

- One empty line indicates the end of the request
  - The server keeps on reading until the end of the request

HTTP Response Phase [1/2]

- Format of a response

```
<HTTP_version> <Response_code> <Text> ←
Content-Type: <MIME_type> ←
<Optional_field>: <Value> ←
←
<Document>
```

- **Response_code**:
  - 2xx: Success (200="OK", 201="Created", etc.)
  - 3xx: Redirection (301="Permanent", 302="Temporary", etc.)
  - 4xx: Client Error (400="Bad request", 401="Unauthorized", 403="Forbidden", 404="Not found", etc.)
  - 5xx: Server Error (500="Internal server error", 501="Not implemented", etc.)

- **Text**: same information as **Response_code**, in plain text

HTTP Response Phase [2/2]

- **Optional fields**:
  - **Date**: date of the request
  - **Last-Modified**: date of last modification of the document
  - **Server**: identification of the server
  - **Content-Length**: size of the document (in bytes)

HTTP: Talking by hand with a server

- **You** can invoke a **web server** manually

```
$ telnet www.cs.vu.nl 80
Trying 130.37.24.11...
Connected to soling.cs.vu.nl (130.37.24.11).
Escape character is '^]'.
HEAD /images/dutch_flag.jpg HTTP/1.0
Host: www.cs.vu.nl

HTTP/1.1 200 OK
Date: Tue, 04 Dec 2001 16:57:35 GMT
Server: Apache/1.3.9 (Unix) mod_ssl/2.4.8 OpenSSL/0.9.4 PHP/4.0.4pl1
Last-Modified: Wed, 15 Aug 2001 11:18:29 GMT
Accept-Ranges: bytes
Content-Length: 2676
Connection: close
Content-Type: image/jpeg

Connection closed by foreign host.
$
```

HTTP/1.1 [1/2]

- If a page contains an HTML document and 10 images
  - Then 11 TCP connections/disconnections are necessary to fetch it
  - That's inefficient: TCP 3-way handshake, slow start
  - HTTP/1.1 allows to **reuse an open TCP connection**

- **Virtual hosts**
  - Can you run multiple web sites on a single machine?
    - e.g., **www.myprofessionalsite.com** and **www.mypersonalsite.org**?
    - Both names will resolve to the same IP address
  - Only one server can accept() on port 80
    - It must treat requests addressed at both virtual servers
  - When a request arrives, how does the server know which site is requested?
    - Requests do not specify hostname: **GET / HTTP/1.1**
    - Browsers add the server name in the request:

```
Host: www.mypersonalsite.org
```

HTTP/1.1 [2/2]

- Content negotiation:
  - Some documents can exist in different versions
    - Document type: GIF vs. JPG vs. PNG
    - Language: English vs. Dutch
    - Encoding: compressed vs. uncompressed
  - HTTP/1.1 allows a client to specify its preferences
    - I prefer PNG rather that GIF
    - I prefer Dutch rather than English
  - The server selects the “best” version

HTML

- **SGML** (Standard Generalized Markup Language)
  - a meta-standard for defining standard document formats
  - Each instantiation of SGML is a **DTD** (Document Type Definition)
    - A formal grammar specification
  - Each version of HTML is defined as one DTD
  - Advantage: a non-ambiguous specification, one can use syntax checking tools, etc.
  - But still, everybody writes wrong HTML pages!
    - Browsers have to be tolerant…
  - One online syntax checker: **http://validator.w3.org**

- HTML distinguishes the structure markup from the presentation markup
  - Structure markup in the HTML file
  - Presentation markup in the style file (or embedded in the HTML)

HTML Document Structure

```
<!DOCTYPE ...>
<html>
  <head>
  <title>...</title>
  ...
  </head>
  <body>
  ...
  </body>
</html>
```

- **DOCTYPE**: defines which DTD is used
  - <!DOCTYPE HTML PUBLIC “-//W3C//DTD XHTML 1.0 Strict//EN” “http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
  - <!DOCTYPE HTML PUBLIC “-//W3C//DTD HTML 4.01 Transitional//EN” “http://www.w3.org/TR/html4/loose.dtd”>

A few HTML tags

- Headers: <h1>…</h1>, … , <h6>…</h6>
- New paragraph: <p>
- Horizontal line: <hr>
- Embedded image: <img src="http://..." alt="…">
- Hypertext link: …
- List:

```
<ul>
  <li> … </li>
  <li> … </li>
</ul>
```

- You can find a comprehensive reference at:
  - http://www.htmlhelp.com

XML: the eXtended Markup Language

- XML is an extension of HTML
  - Describes any kind of structured document

- Anybody can define their own XML document structure by writing a DTD file
  - Example: a shopping list contains at least one item, and an item is made of parsed character data:

```
<!ELEMENT Shopping-List (Item)+>
<!ELEMENT Item (#PCDATA)>
```

  - Now you can write a shopping list with your newly defined tags:

```
<?xml version="1.0"?>
<!DOCTYPE Shopping-List SYSTEM "shoplist.dtd">
<Shopping-List>
  <Item>Chocolate</Item>
  <Item>Sugar</Item>
</Shopping-List>
```

CGI Programming

CGI Overview [1/2]

- CGI (Common Gateway Interface)
  - Dynamically generating web documents at request time
- Defined in RFC 3875
  - http://tools.ietf.org/html/rfc3875
- CGI examples
  - Search engines
  - Page counters
  - All form handling:
    - The client fills up the form
    - Sends the requests with the submitted information as parameters
    - The CGI is invoked with those parameters
    - Dynamically generates a Web page according to the parameters

CGI Overview [2/2]

- The server knows that certain URLs are related to CGIs instead of static documents
  - E.g., http://foo.cs.vu.nl/cgi-bin/mycgi
- When a request is received:
  - The web server replies with a standard HTTP header
  - It executes mycgi (i.e., fork() + exec())
  - All data that mycgi generates on stdout is sent to the client
  - The server does not know which MIME type is generated by the CGI!
    - mycgi must write the end of the HTTP header (e.g., Content-Type)
    - It must write an empty line, then the generated document
- CGIs can:
  - be programmed in any language
  - process input parameters as they want
    - e.g., send database requests, RPC to an application server, etc.

GET and POST requests [1/2]

- CGI requests can be sent via GET or POST HTTP methods
  - **GET** is normally used for **idempotent** requests
  - **POST** for **non-idempotent** ones
- A CGI can handle either one or both types
- GET:
  - Request parameters are embedded inside the URL:
  - **http://foo.cs.vu.nl/cgi-bin/mycgi?x=2&y=hello&city=Amstelveen**

```
GET /cgi-bin/mycgi?x=2&y=hello&city=Amstelveen HTTP/1.0 ←
←
```

  - One can invoke a GET CGI through an HTML form, or by an explicit hypertext link (with parameters)
  - The request string is available to the CGI as an environment variable called $QUERY_STRING

GET and POST requests [2/2]

- POST:
  - Request parameters are transmitted after the HTTP request header
  - The HTTP request must contain Content-Type and Content-Length fields:

```
POST /cgi-bin/mycgi HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 27

x=2&y=hello&city=Amstelveen
```

  - The request string is available to the CGI from the standard input (stdin)

- CGIs can check which HTTP method was used by checking the $REQUEST_METHOD environment variable

CGI Parameter Encoding

- Request parameters are encoded (x-www-form-urlencoded)
  - Space → "+"
  - Special characters → %<ASCII_code>
    - @ → %40
    - Ç → %C7
- All parameters are concatenated into a single string, with "&" separators
  - x=2 y=hello, world! anotherparam=xyz
  - x=2&y=hello%2C+world%21&anotherparam=xyz
- There are libraries to decode query strings for you…
  - In C: http://www.boutell.com/cgic/
  - In Perl: http://cgi-lib.berkeley.edu/
  - **You are NOT allowed to use them in your assignments!**

An example CGI program

- A finger interface: fing

```
#!/bin/sh
echo Content-Type: text/html
echo
echo "<html><head><title>$QUERY_STRING</title></head>"
echo "<body><h1>A finger interface</h1><pre>"
/usr/bin/finger "$QUERY_STRING"
echo "</pre></body></html>"
```

- Typical request:
  - **http://foo.cs.vu.nl/cgi-bin/fing?jsmith**

Handling forms

- Here is an example HTML form:

```
<html>
  <body>
    <h1>An addition form</h1>
    <form action="/cgi-bin/addition" method="post">
      <input type="text" name="x">
      <input type="text" name="y">
      <input type="submit" value="go!">
    </form>
  </body>
</html>
```

- Can you write the CGI that will handle the form and display the result of the addition?

- Is there any security issue with this CGI?

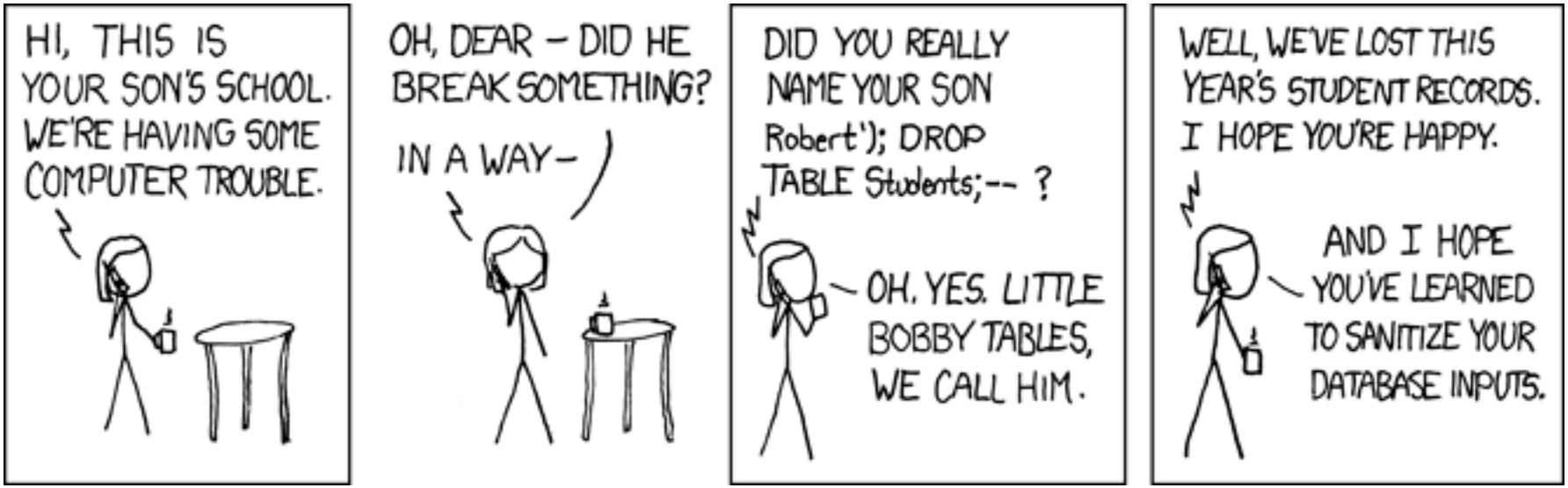
Security Issues with CGIs [1/2]

- It is extremely easy to unintentionally create security holes with CGIs

```
#!/usr/bin/perl
require "cgi-lib.pl";
print "Content-Type: text/plain\n\n";
if (&ReadParse) {
  system("finger $in{addr}");
}
```

- The programmer assumes parameters like: addr=spyros
- What if somebody submitted this request string?
  - `addr=jsmith;cat+/etc/passwd`
- The CGI would execute two commands:
  - `finger jsmith;cat /etc/passwd`
- This CGI is equivalent to a public Unix account!

…security issues!



Security Issues with CGIs [2/2]

- Secure version of this CGI:

```
#!/usr/bin/perl
require "cgi-lib.pl";
print "Content-Type: text/plain\n\n";
if ((&ReadParse) && ($in{addr}=~/^[a-zA-Z\.\@]*$/)) {
  system("finger $in{addr}");
} else {
  print "Error! Wrong parameter!\n";
}
```

- Conclusion: Each time you write a CGI, think that users may submit anything as request parameters!
  - CGIs **must always** check the **correctness of their input parameters**
  - Always think of the "worst case scenario"
  - Mostly concerned: CGIs requesting external resources
    - Shell commands
    - Database/application requests
    - File manipulation

Servlets

Servlets

- CGIs are fundamentally stateless
  - They do not keep state between two requests (except if they store it in a file/database)

- Issues with programming CGIs in Java
  - Java bytecode is not executable: one has to write a wrapper script which is the real CGI
    - `exec java FooBar.class $*`
  - Starting a Java Virtual Machine (JVM) takes a lot of time and resources
  - It is not very advisable to execute multiple Java virtual machines in parallel

- Servlets
  - The Web server contains **one** occurrence of the JVM
  - A given Servlet is **created once** at startup, then **requested multiple times**, then **destroyed**
  - Servlets can **maintain a state in memory** over several requests
  - Servlets can run in a **sandbox**, like applets

Server Programming

- A Servlet is a Java class that derives from **`HttpServlet`**
- It can overwrite inherited methods:

```
void init(ServletConfig config);
void destroy();
void doGet(HttpServletRequest req, HttpServletResponse res);
void doHead(HttpServletRequest req, HttpServletResponse res);
void doPost(HttpServletRequest req, HttpServletResponse res);
void doPut(HttpServletRequest req, HttpServletResponse res);
void doDelete(HttpServletRequest req, HttpServletResponse res);
```

- **`init()`** is called once at startup
- **`do***()`** methods are called multiple times, possibly concurrently
  - You must manage thread safeness yourselves!
- **`destroy()`** is called once when the server is stopped

Example: A Page Counter

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

public class HelloClientServlet extends HttpServlet
{
  private int counter;

  public void init(ServletConfig config) throws ServletException
  {
    super.init(config);
    counter = 0;
  }

  public String getServletInfo()  { return "MyNiceCounter v1.0"; }

  protected void doGet(HttpServletRequest req, HttpServletResponse res)
  throws ServletException, IOException
  {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    synchronized (this)
    {
      counter = counter + 1;
      out.println("<HTML><HEAD><TITLE>My counter page</TITLE>" +
                  "</HEAD><BODY>This page has been requested " +
                  counter + " times</BODY></HTML>");
    }
  out.close();
  }
}
```

PHP Programming

Server-Side Scripting Languages [1/2]

- Often, dynamic pages are mostly static
  - **A lot** of **static** information: page headers, layout, etc.
  - **A few** really **dynamic** parts inside
  - Using CGI or Servlets, one is obliged to write many static print statements
- Let's design pages by writing static HTML documents with **pieces of code** inside
  - The code is **executed by the server** at request time
  - Clients only see "regular" pages, **without code statements**
  - Pages can take parameters, like CGIs
  - This is supported by **template languages/frameworks**
- Scripting languages come with libraries providing functions especially designed for Web programming
  - E.g., URL/parameters manipulation, database gateways, etc.

Server-Side Scripting Languages [2/2]

- **SSI**: **S**erver-**S**ide **I**ncludes
  - A very old scripting language
  - Mostly provides: **`#include`**
- **PHP**: **P**HP **H**ypertext **P**reprocessor
  - A complete language
  - It is free!
  - It integrates nicely into Apache, IIS, and other major Web servers
- **ASP**: **A**ctive **S**erver **P**ages
  - The Microsoft equivalent to PHP
  - Language: essentially Basic
- **JSP**: **J**ava **S**erver **P**ages
  - Same, but looks like Java

PHP

- The most popular third-party module for Apache
  - Code and extensive docs available from http://www.php.net/
- Pieces of code are enclosed in <?php … ?>
- Example: this PHP page…

```
<html>
  <head><title>PHP Test</title></head>
  <body><?php print("Hello, World!"); ?></body>
</html>
```

…will appear to clients as:

```
<html>
  <head><title>PHP Test</title></head>
  <body>Hello, World!</body>
</html>
```

Parameter Decoding

- Each PHP script can be invoked as a CGI
- Parameters are decoded automatically and appear to the script as normal variables
- Example:
  - **http://foo.cs.vu.nl/~spyros/myphp.php?x=hello&y=2**

```
<html>
  <head><title>PHP Test</title></head>
  <body>
    <?php
      printf("x = %s<br>\n", $_GET['x']);
      printf("y = %s = %d<br>\n", $_GET['y'], $_GET['y']);
    ?>
  </body>
</html>
```

More Preset Variables

- PHP creates many other variables automatically
  - $_SERVER["HTTP_USER_AGENT"]
  - $_SERVER["REMOTE_ADDR"]
  - …
  - To get a complete list: `<? phpinfo(); ?>`
- Example:

```
<?php
   if(strstr($_REQUEST["HTTP_USER_AGENT"],"MSIE"))
   {
?>
     <center><b>You are using Internet Explorer</b></center>
<?php
   }
   else
   {
?>
     <center><b>You are not using Internet Explorer, good!</b></center>
<?php } ?>
```

- Did you notice that you can split PHP into pieces separated by HTML?

PHP Control Structures

- Like all languages, PHP has control structures

```
<?php
  while ($x!=0) {
     ...
  }

  if ($x==0) {
     ...
  }
  elseif ($x==1) {
     ...
  }
  else {
     ...
  }
?>
```

PHP Arrays [1/3]

- A very useful feature of PHP is the array variable type

- An array is a set of key-value associations

```
$type = array {
  "The Netherlands"  => "country",
  "Amsterdam"        => "city",
};
$type["Schiphol"] = "airport";
$test = $type["Amsterdam"];
```

PHP Arrays [2/3]

- Arrays are both **associative** and **indexed** at the same time

- When the keys are not specified, integers 0, 1, 2, … are used. E.g.:

```
$cities     =  array("Amsterdam", "Athens", "Zurich");
$myCity     =  $cities[1];
$cities[3]  =  "Washington";
```

  - …is equivalent to

```
$cities     =  array(0=>"Amsterdam", 1=>"Athens", 2=>"Zurich");
$myCity     =  $cities[1];
$cities[3]  =  "Washington";
```

- Integer and String keys can coexist
  - When a key is omitted, the largest integer index + 1 is used

```
$test   =  array(5=>"Amsterdam", "athina"=>"Athens", "Zurich");
// is equivalent to:
$test   =  array(5=>"Amsterdam", "athina"=>"Athens", 6=>"Zurich");
```

PHP Arrays [3/3]

- What about an **array of arrays**?

```
$students = array (
  "You"     => array ( "name" => "yourname", "student nb" => 1234),
  "Another" => array ( "name" => "hisname",  "student nb" => 5678),
  "Foobar"  => 42
);

$you      = $students["You"];
$yourname = $students["You"]["name"];

print("<pre>");
// This will print a human-readable version of $students
print_r($students);
print("</pre>");
```

PHP Functions [1/2]

- You can define functions:

```
function addition($x, $y)
{
  return $x + $y;
}
$foo = addition(10, 50);
```

- PHP contains a zillion predefined functions to help you programming
  - They are classified into categories:
    - Apache, Arrays, Aspell, BC, Bzip2, Calendar, CCVS, COM, Classes/Objects, ClibPDF, Crack, CURL, Cybercash, CyberMUT, Cyradm, ctype, dba, Date/Time, dBase, DBM, dbx, DB++, DIO, Directories, DOM XML, .NET, Errors and Logging, FrontBase, filePro, Filesystem, FDF, FriBiDi, FTP, Functions, gettext, GMP, HTTP, Hyperwave, etc.
  - For a full list, check the documentation:
    - **http://www.php.net/manual/en**

PHP Functions [2/2]

- QUESTION: What does the following code do?

```
<?php
  function lm($x) {
    printf("<font size=-1>[Last modified: %s]</font> ",
           date("D, d M Y H:i:s",  filemtime("$x")));
  }
?>

<html>
  <body>
    Download your new <a href="foo.ps.gz">assignment</a>
    <?php lm("foo.ps.gz"); ?>
  </body>
</html>
```

Manipulating HTTP Headers

- By default, PHP scripts are assumed to generate HTML
  - The **Content-Type** field of the response defaults to **text/html**

- But you can force it otherwise

```
<?php $len = filesize("foo.pdf");
      header("Content-type: application/pdf");
      header("Content-Length: $len");
      readfile("foo.pdf"); ?>
```

- Attention, you can only manipulate HTTP headers **before** displaying anything
  - This code will **not** work

```
<?php print("foo");
      header("Content-type: text/plain"); ?>
```

Access Control [1/3]

- In many cases you want to control who has access to certain pages
  - Administration pages
  - Your secret plans to rule the world
  - etc.

- You can do that by checking the client's IP address

```
if ($_REQUEST["REMOTE_ADDR"] != "130.37.193.13") {
  header("HTTP/1.0 403 forbidden access");
  print("You are not authorized to see my secret plans");
  exit;
}
```

  - This will work only if you know exactly who has access to 130.37.193.13
  - What if an authorized user is on vacation and wants to access the protected page from an internet café?

Access Control [2/3]

- You can easily setup access control using the **HTTP basic authentication**:
  - First HTTP request => return HTTP code 401 “please authenticate yourself”
  - The client types a username and password
  - Another HTTP request containing a username and a password => authorized to enter

- PHP has standard variables for username and password:
  - $PHP_AUTH_USER
  - $PHP_AUTH_PW

- Note that using this scheme **the documents are not encrypted**
  - Even the username and password are sent in clear text…
  - Anyone who intercepts the request with the password can gain access to your secret plans!

Access Control [3/3]

```
<?php
  // Check if the request carries an authentication
  if (!isset($_REQUEST["PHP_AUTH_USER"]))
  {
    // No authentication
    header('HTTP/1.0 401 Unauthorized');
    header('WWW-Authenticate: Basic realm="My secret zone"');
    print("Please authenticate yourself");
    exit;
  }

  elseif (($_REQUEST["PHP_AUTH_USER"]=="spyros") &&
          ($_REQUEST["PHP_AUTH_PW"]=="123"))
  {
    // Correct authentication
    printf("Hello, %s. You can enter my page.", $_REQUEST["PHP_AUTH_USER"]);
  }

  else
  {
    // Wrong authentication
    header('HTTP/1.0 403 Forbidden');
    print("Go away, you are not authorized here!");
    exit;
  }
?>
```

Session Tracking [1/4]

- By default, a Web server is stateless
  - Each request is treated independently
  - The Web server does not “remember” past requests
- In many cases, you will want to establish a session
  - Keep a memory of the past requests from each user
  - Example: in an e-commerce Web site
    - Browse to view products
    - Click to add items to your shopping basket
    - When you checkout, the server “remembers” which items you have selected

Session Tracking [2/4]

- Send cookies (i.e., pieces of data) to the client
  - The client will attach the same cookie with each further request
  - If the server sends a different cookie to each new client, then it can track which request has been made by which client
- A cookie is made of:
  - A name (“userid”)
  - A value (“2671050”)
  - An expiration date (“31 december 2015”)
  - A path (“/myshop/”)
  - A server domain (“.cs.vu.nl”)
  - A secure flag (allow transport only over encrypted connection)
- Until the cookie expires, the name and values will be sent together with every request addressed at a page whose server belongs to “.cs.vu.nl” and whose path starts with “/myshop/”.
- NOTE: Cookies can be altered by the user! Naïve use of cookies can infer security issues.

Session Tracking [3/4]

- Cookies are transmitted in the HTTP response header
  - You can only set cookies before printing anything (otherwise you may get weird errors)

```
<?php
  if (!isset($_COOKIE["userid"])) {
    srand((double)microtime()*1000000);
    $newid  = rand();
    $expire = time() + 3600; // in 1 hour
    setcookie("userid",$newid,$expire,"/myshop/",".cs.vu.nl",0);
    print("From now on, you will be known as user number $newid");
  }
  else {
    $id = $_COOKIE["userid"];
    print("You are identified as user number $id");
  }
?>
```

Session Tracking [4/4]

- Thanks to cookies, you can store whatever you need to remember about each user (e.g., one file per user)

- PHP has a great feature to automate sessions for you
  - It uses one standard cookie named “PHPSESSID”
  - It automatically sets it to new clients
  - It creates a temporary file for each client
  - You can choose which PHP variables to save into/reload from this file

```
<?php
  session_start();    // Check cookie, or set it if this is a new user

  $_SESSION["count"]++; // Load variable 'count' from the file attached
                        // to this user. The variable will be saved
                        // again automatically when exiting.

  print("You have visited this page ".$_SESSION["count"]." times");
?>
```

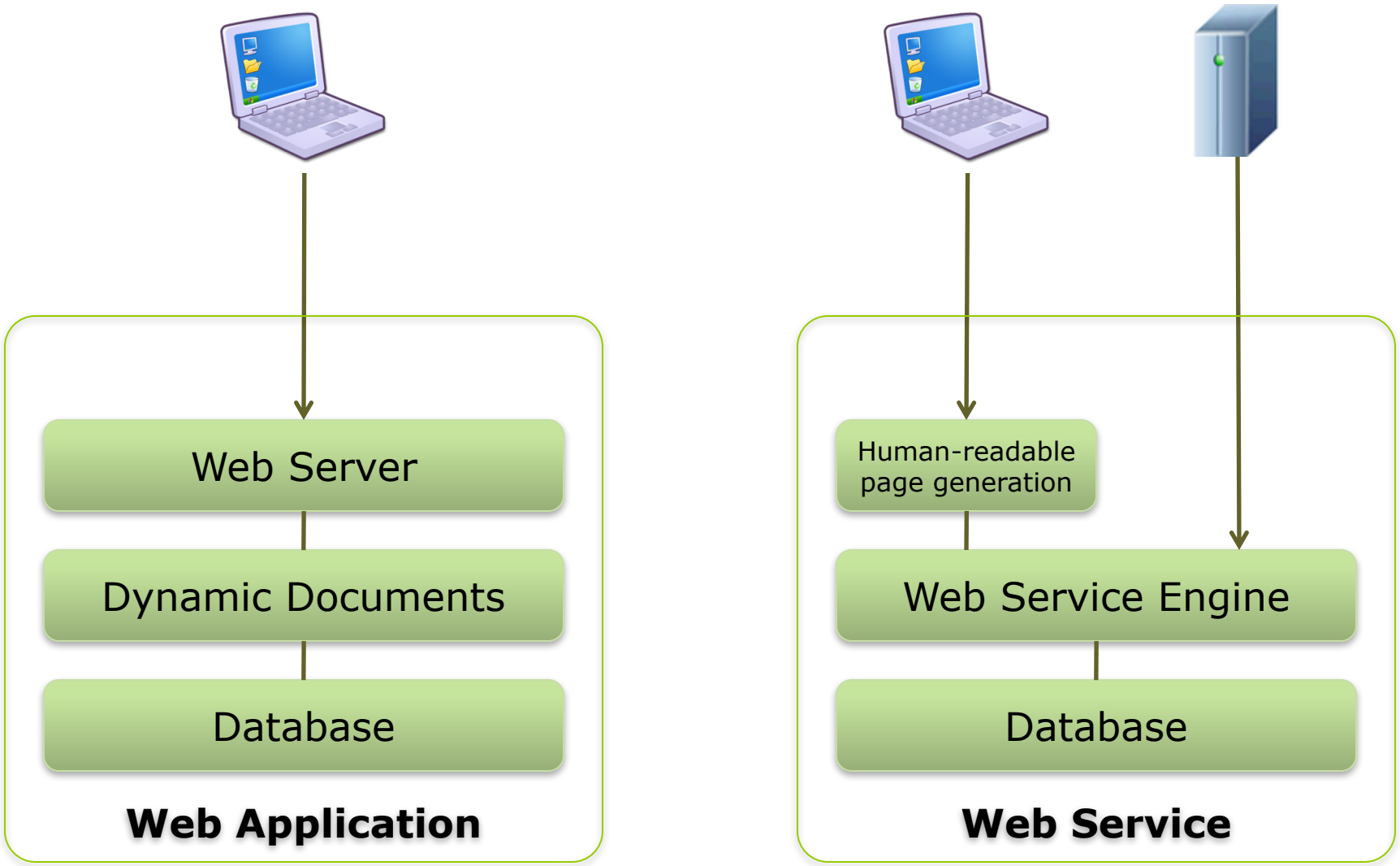
- You can destroy a session with **session_destroy()**

Web Services

From Web Apps to Web Services [1/2]

- The use of dynamic content allows the development of Web Applications (e.g., commerce sites)
  - Web Applications are fine as long as they are used by a human sitting behind the browser
  - But imagine that the client is an automated program
    - The program must pretend it's a human: Send Web requests, parse the HTML documents it receives, fill out the forms correctly, submit more requests
    - Quite difficult
    - If the site changes its visual layout, your automatic parsing programs will probably need to be updated
- Web Services provide a standardized interface for machine-to-machine interaction

From Web Apps to Web Services [2/2]



Web Services

- A Web Service is an **RPC-like interface to a Web Application**
  - Examples: Google, Amazon, etc.
  - You can write programs which automatically order books from Amazon
- To build an RPC system you need:
  - A communication protocol
  - A standard message format (marshalling parameters)
  - An IDL file defining the interface
  - Implementations of clients and servers that stick to the standards
- Web Services are built using Web technologies
  - Communication protocol: HTTP
  - Message format: something XML
  - A WSDL document defining the interface

WSDL

- **WSDL**: **W**eb **S**ervice **D**escription **L**anguage
  - One WSDL document contains the specification of the interface to one Web Service, plus the URL at which the service is running
  - WSDL documents are written in XML

- Once you have created a Web Service, you must distribute your WSDL specification to your clients
  - Clients will compile your WSDL specification to create stubs
  - They will use the stub to access your Web Service

- Problem: WSDL files are huge!
  - Example: 66 lines to describe

```
int GetStockPrice(char *symbol, float *Result);
```

  - Very often people generate a WSDL specification out of a Java or C interface, instead of generating the interface from the WSDL file

SOAP [1/3]

- **SOAP**: **S**imple **O**bject **A**ccess **P**rotocol
  - SOAP is the message format to write requests and responses to/from a Web Service
  - SOAP requests and responses are written in XML
- SOAP requests/responses can be transported with any protocol
  - But everybody uses HTTP
- SOAP is portable
  - HTTP is implemented on all kinds of platforms
  - XML is an open text-based document format
  - There are many existing implementations
    - Mostly in Java, but also in other languages

SOAP [2/3]

- Example request:

```
POST /InStock HTTP/1.1
Host: www.stock.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Body xmlns:m="http://www.stock.org/stock">
    <m:GetStockPrice>
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

SOAP [3/3]

- And the response:

```
HTTP/1.1 200 OK
Content-Type: application/soap; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

  <soap:Body xmlns:m="http://www.stock.org/stock">
    <m:GetStockPriceResponse>
      <m:Price>34.5</m:Price>
    </m:GetStockPriceResponse>
  </soap:Body>

</soap:Envelope>
```

UDDI [1/2]

- **UDDI**: **U**niversal **D**iscovery **D**escription and **I**ntegration
  - It is a business directory (e.g., Gouden Gids) in which you can register your Web Services
    - Send a description of your service (which business you provide, where you are located, etc.)
    - And the WSDL description of your service
  - Users can search for a service that fits their needs
    - Describe what they are looking for
    - Receive information about services that match the query
- Access to UDDI is realized via the SOAP protocol
- There are several UDDI servers around
  - Operated by IBM, Microsoft, SAP, etc.
  - They replicate each other's content
    - So, registering your service to any one of them is enough
    - And searching on any one of them is enough too

UDDI [2/2]

- Classifying Web Services
  - To describe a Web Service ("my service is about selling books") you need a business classification
  - There are several such classifications
- Unfortunately (almost) nobody uses UDDI as intended
  - Major service providers (Google, Amazon, etc.) register their services without classification
    - So the only way to find them is by searching them with their name, which greatly reduces the interest of the system
- It is hard to automate the **selection** of a service
  - E.g., write a client program which will use UDDI to automatically select a "good" book store where to buy the latest Harry Potter
    - Search for low prices – easy
    - Search for good delivery costs/delays – doable
    - Search for a company that you trust – hard

JavaScript

Reader

- There are thousands of books about JavaScript
  - Most of them of terrible quality ☹
- **A very good book about JavaScript**
  - **"JavaScript, The Definitive Guide"**

    David Flanagan, O'Reilly Media, 6th edition, 2011
  - The following slides are largely inspired by this book
  - You are not obliged to read it for Internet Programming, but it's just a suggestion for programming in JavaScript
- A good **tutorial** about the JavaScript language (not the libraries)

  `http://developer.mozilla.org/en/A_re-introduction_to_JavaScript`

JavaScript

- JavaScript is the defacto standard language for processing Web content at the client side
  - Check form contents, etc.
  - Build responsive user interfaces (hide the client-server latency)
  - AJAX: **A**synchronous **J**avascript **A**nd **X**ml

- **JavaScript != Java**

JavaScript	Java
Runs mostly inside a browser	Mostly standalone
Interpreted	Compiled into bytecode
Prototype-based	Class-based
Weakly typed	Strongly typed
Dynamically typed	Statically typed

JavaScript capabilities

JavaScript can	JavaScript cannot
Manipulate a page displayed in your browser	Read/Write files at the client machine
Open an HTTP connection	Open a UDP/TCP socket

JavaScript basics

- Basic types

```
var b = true;           // boolean
var i = 2;              // number
var s = "hello world"; // string
var c = 'character';    // string! JavaScript has no char type...
var u = undefined;      // variable u exists but its value is undefined
```

- Arrays

```
var empty  = [];                    // An empty array with no element
var fib    = [1, 1, 2, 3, 5, 8, 13]; // An array of 7 numeric elements
var misc   = [1.1, true, "hello"];   // An array of different types
var matrix = [[1,2,3], [4,5,6]];     // An array of arrays
```

Objects

- In JavaScript an **object** is **not** an instance of a class
  - JavaScript has no real concept of a class

- An object is an associative array
  - a set of key/value pairs, called **properties**

```
var empty  = {};   // An object with no properties
var point  = { x:2, y:0 };
var circle = { x:point.x, y:point.y, radius:2 };
circle.surface = 3.14159 * circle.radius * circle.radius;
```

- An object always has some standard properties
  - **A constructor**: a function used to initialize the object
  - **toString()**: serializes the object into a string
  - **hasOwnProperty(p)**: returns true if the object has its own (non-inherited) property called **p**
  - …and a few others

Control structures

- JavaScript has the usual control structures

```
if (condition) { blabla; } else { bleble; }
while (condition) { blablabla; }
do { blabla; } while (condition);
for (x=0;x<10;x++) { blabla; }
try { blabla; } catch (e) { alert(e); }
```

- **For / in** is used to traverse all properties of an object

```
for (prop in o) {
  document.write("o." + prop + " = " + o[prop]);
  document.write("<br>");
}
```

Functions [1/2]

- The “usual” way to create a function

```
function fib(x) {
  if (x<2) return x;
  return fib(x-1) + fib(x-2);
}
```

- A function has a scope

```
function hypotenuse(a,b) {
  function square(x) { return x*x; }
  return Math.sqrt(square(a) + square(b));
}
```

Functions [2/2]

- A function is just another kind of variable

```
var fib2 = function(x) {
  if (x<2) return x;
  return fib2(x-1) + fib2(x-2);
}
var fib3 = fib2;
var x = fib3(10);
```

- You can also use unnamed functions

```
var myarray=[25, 8, 7, 41]

myarray.sort(function(a,b){return a - b})
//Array now becomes [7, 8, 25, 41]
```

Object constructors

- An object always has a constructor: a function used to initialize it
- Note that this code is the complete "definition of the class"!

```
function Rectangle(w, h) {
  this.width  = w;
  this.height = h;
  // no return statement
}

var rect = new Rectangle(2,4);
```

- Someone could add properties to an object

```
rect.area = function() { return this.height * this.width; }
```

Object methods

- You may want to define some standard methods for rectangles

```
function Rectangle(w, h) {
  this.width  = w;
  this.height = h;
  this.area   = function() { return this.height * this.width; }
}

var r = new Rectangle(2,4);
var a = r.area();  // will set a=8
```

- Another way to do it:

```
function Rectangle(w, h) {
  this.width  = w;
  this.height = h;
}

Rectangle.prototype.area = function() {
  return this.height * this.width;
}
```

Pattern matching

- JavaScript has built-in support for Perl-like regular expressions

```
var pattern = new RegExp("Java");
var pattern2 = /Java/;              // two equivalent notations

var text = "JavaScript is much more fun than Java";
while ((result = pattern.exec(text)) != null) {
  print("Matched '" + result[0] + "' at position " +
        result.index + "; next search begins at " +
        pattern.lastIndex);
}
```

- The output:

```
Matched 'Java' at position 0; next search begins at 4
Matched 'Java' at position 34; next search begins at 38
```

JavaScript as a dynamic language

- JavaScript can dynamically manipulate any class!

```
var r = new RegExp("foobar$");
r.prototype.myownmethod = function(x) { return x*x; }
```

- JavaScript can also dynamically execute any piece of code
  - For example after receiving it through the network

```
var string = "function f(x) {return x*x;}  print(f(32));"
eval(string);
```

Object scope

- An object is always created within the scope of another object
- If you use JavaScript inside a Web browser, then the root object is the current window
  - The following two statements are equivalent:

```
var answer    = 42;  // Create a new variable
window.answer = 42;  // Create a property of the Window object
```

- The window object contains a number of standard properties
  - **self**, **window**, **parent**, **top**: various other Window objects
  - **document**: document object
    - **forms[]**: an array of Form objects
    - **anchors[]**: an array of Anchor objects
    - …
  - …

Embedding JavaScript into HTML

- You can write JavaScript code into an HTML document thanks to the <script> tag
  - Similarly to PHP, the code can be broken into multiple sub-pieces

```
<html>
 <head>
  <title>Today's date</title>
  <script language="Javascript">
   function print_todays_date() { // Define a function for later use
    var d = new Date();          // Get the current date
    document.write(d.toString()); // Insert it into the document
   }
  </script>
 </head>
 <body>
  The current time is:
  <script language="Javascript">
   print_todays_date();          // Call the function defined above
  </script>
 </body>
</html>
```

Keeping JavaScript out of HTML files

- It is very easy to #include a piece of JavaScript into an HTML page
  - E.g., to include the same JavaScript functions in all related pages

```
<script src="../scripts/utils.js">
  // nothing here!
</script>
```

Event Handlers

- Normally your JavaScript code gets interpreted at the time the HTML page is being loaded
  - Like a normal (shell) script: read it, execute it, exit
  - But this does not allow to create an interactive document!

- JavaScript allows you to define event handlers
  - Similar to Unix signals:

    "execute this code when such an event happens"

```
<html>
  <head><title>Test document</title></head>
  <body>
    <p>Try to click <b onclick="alert('Hello world!');”>here</b>...
  </body>
</html>
```

  - Other event handlers: **onclick**, **onmousedown**, **onmouseup**, **onmouseover**, **onmouseout**, **onchange**, **onload**
  - Most HTML tags can support these handlers

The location object [1/2]

- Interesting standard object: **location**
  - Represents the current URL

- Change the current location (i.e., redirect)

```
location = 'http://www.google.com';
```

The location object [2/2]

- Read the current location:
  - The location object has 4 attributes: **protocol**, **host**, **pathname**, **search**
  - **location.search** contains the URL arguments

    http://www.foo.com/pathname?param1=12&param2=foobar

```
var args;                                       // Arguments including the
initial '?'
var query = location.search.substring(1);       // Arguments without the
                                                // initial '?'
var pairs = query.split("&");                   // Break at ampersand

for (var i=0; i<pairs.length; i++) {
  var pos = pairs[i].indexOf('=');              // Look for '='
  if (pos == -1) continue;                      // If not found, skip
  var argname = pairs[i].substring(0,pos);      // Extract the name
  var value   = pairs[i].substring(pos+1);      // Extract the value;
  value = decodeURIComponent(value);            // Decode it, if needed
  args[argname] = value;
}
```

AJAX

AJAX

- AJAX: **A**synchronous **J**avascript **A**nd **X**ML

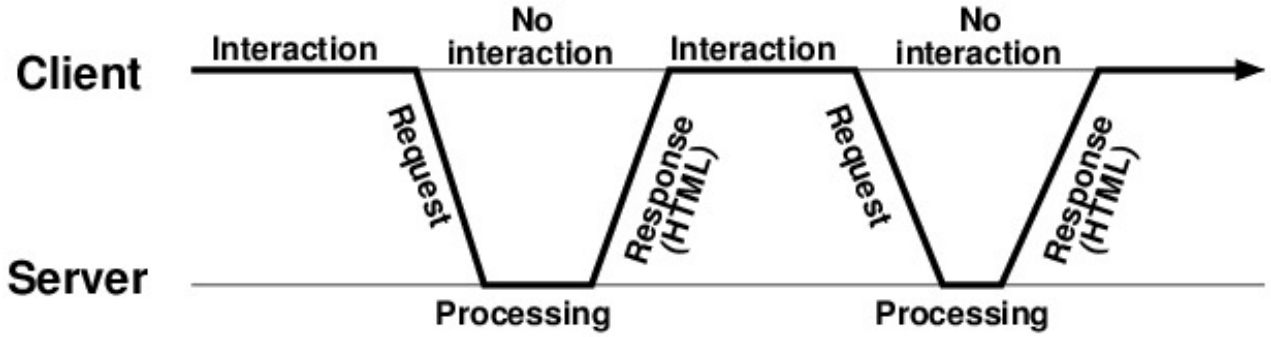
- Key feature: Scripted HTTP communication for dynamic browser/server interaction, without reloading pages
  - small updates
  - no style/script reloading
  - significant response acceleration

AJAX

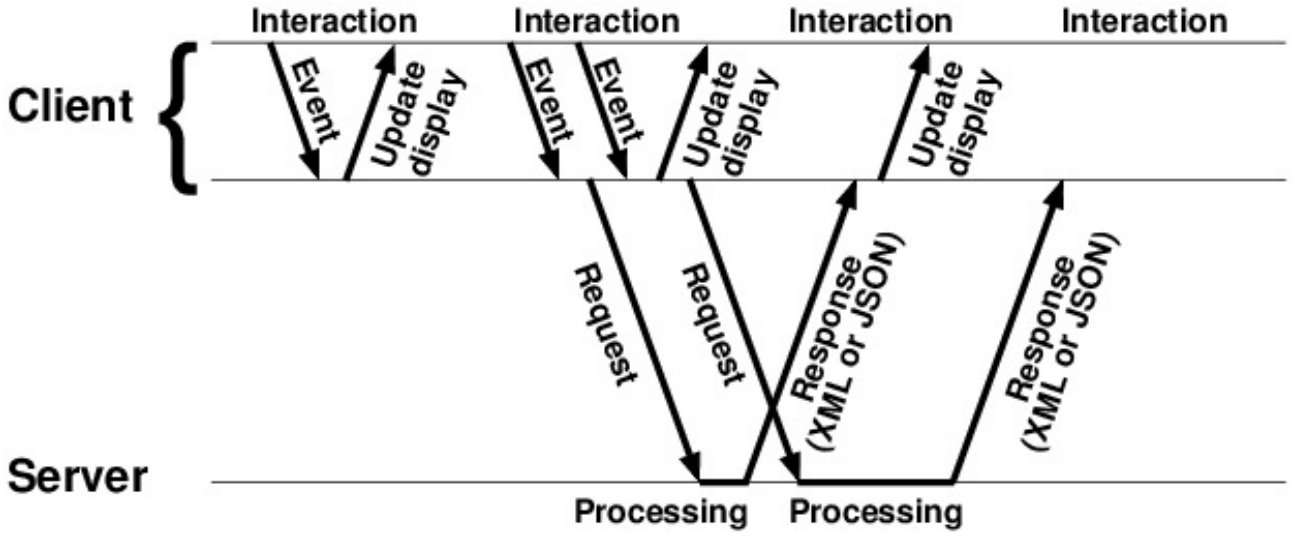
- AJAX is a collection of technologies
  - **Requests**: Issue external **HTTP requests**, mainly **asynchronous** ones
    - Don't block the browser, any reply will be processed when it arrives
  - **Serialization**: **Exchange structured data** with the web server
    - Usually the raw data that the browser will need to display
    - These data will usually be generated dynamically at the server (e.g., using PHP)
    - These data may be represented in XML format (or other formats)
  - **Updates**: **Dynamically update the HTML page** that is being displayed
    - The HTML file specifies the **initial state** of the page
    - Later on, you can add/remove/edit HTML elements in it via JavaScript

AJAX

- Non-AJAX web application:



- AJAX web application:



Example: Google Maps

- Google Maps extensively relies on AJAX for smooth interaction
  - Type in a location, your browser sends a search request to the server
  - The server returns a page with the actual coordinates, and links to the tiles (square map segments) to be displayed
  - Your browser fetches the tiles asynchronously…
  - …while you can still use your mouse to scroll/zoom in the document
  - …and new tiles are downloaded as needed

- For more information on how Google Maps uses AJAX:
  - Link **Mapping Google** on the course's web page

AJAX Components

- AJAX combines:
  - **Requests** to keep the web app responsive
  - **Serialization** to represent structured data
  - **Updates** to dynamically update a document

- Let's see them one by one…

Requests: Synchronous

- **XMLHttpRequest** is a standard JavaScript object to send synchronous or asynchronous HTTP requests
  - Note: XMLHttpRequest has nothing to do with XML ☹

- To issue a synchronous request:

```
try {
  var req = new XMLHttpRequest();  // Create a new request
  req.open("GET", "http://www.cs.vu.nl/", false);// False = synchronous
  req.setRequestHeader("User-Agent","Spyros/1.0"); // Overrides default
  req.send(null);                                  // Sends the request

  if (req.status == 200) {
    alert(req.responseText); // We received a response
  }
  else {alert("Error " + req.status + ": " + req.statusText);}
}
catch (e) {   // What could throw an exception?
  alert("Your browser is not happy: " + e);
}
```

Requests: Asynchronous

- Sending an asynchronous HTTP request is easy:

```
var req = new XMLHttpRequest();                    // Create a new request
req.open("GET", "http://www.cs.vu.nl/", true); // True = asynchronous
req.setRequestHeader("User-Agent","Spyros/1.0"); // Overrides default
req.send(null);                                    // Sends the request
```

- req.send(null) will return **immediately**

- **req.readyState** lets you check the current status of the request

readyState	meaning
0	**open()** has not been called yet
1	**open()** has been called, but **send()** has not been called yet
2	**send()** has been called, but we don't have any answer yet
3	We are currently receiving data
4	The request is finished

Requests: Asynchronous Response

- An HTTP request may take any amount of time
  - Busy waiting??? Forbidden!!
  - So, when should you check the response status?

- How do you know when you should check for the request status?
  - Create an event handler to be called when the request status changes

```
var req = new XMLHttpRequest();

// Register an event handler
req.onreadystatechange = function() {
  if (req.readyState == 4) {      // If the request is finished
    if (req.status == 200) {      // If the request was successful
      alert(req.responseText); // Display the received response
    }
  }
}
req.open("GET", "http://www.cs.vu.nl/"); // No third parameter => async
req.send(null);
```

Requests: Sending POST queries

- You must encode each parameter's name and value
  - Function **encodeURIComponent()** does most of the work for you
  - Except for one thing: it encodes spaces as %20 instead of +!

```
function encodeForPost(string) {
  var regexp = /%20/g; // Reg. expr. to match %20 any number of times
  return encodeURIComponent(string).replace(regexp,"+");
}

var param = encodeForPost("name1") + "=" + encodeForPost("IntProg")
    + "&" + encodeForPost("name2") + "=" + encodeForPost("42.5");
```

Requests: Sending POST queries

- To send the POST query:

```
var req = new XMLHttpRequest();
req.open("POST", "http://www.cs.vu.nl/~spyros/test.php");
req.setRequestHeader("Content-Type","application/x-www-form-urlencoded");
req.onreadystatechange = function() {
  if ((req.readyState == 4) && (req.status == 200)) {
    alert(req.responseText); // Display the received response
  }
}
req.send(param);
```

Requests: XMLHttpRequest caveats

- Internet Explorer 5 and 6 had different ways to create the same XMLHttpRequest object ☹
  - Don't use these browsers for the Internet Programming class!
  - IE7 and IE8 should be OK (try at your own risk!)
  - (After object creation, the rest of the interface is the same)
- Your browser will not allow sending queries to any server
  - You can only send requests to the same server that sent you the JavaScript code
  - That's for security reasons
- Different browsers interpret **readyState==3** differently
  - Better to wait until **readyState=4** before you start using the response

Serialization: JSON

- JSON: **J**avascript **S**tandard **O**bject **N**otation
  - Yet another object serialization standard…
    - Represent structured information in a standardized format
    - Make it easy to communicate between different platforms/languages
  - JSON has a couple of nice features
    - Very lightweight compared to XML (data size, processing speed)
    - There are libraries for it in every language

JavaScript	JSON	XML

`var x = new Object();`
`x.name = "John Doe";`
`x.books = ["First",`
`"Second"];`
	`{`
	`"name": "John Doe",`
	`"books": [`
	`"First",`
	`"Second"`
	`]`
	`}`
		`<author>`
		`<name>John Doe</name>`
		`<books>`
		`<book>First</book>`
		`<book>Second</book>`
		`</books>`
		`</author>`

Serialization: Parsing JSON

- Parsing JSON strings *looks* trivial:

```
var string = '{"name": "John Doe", "books": ["First", "Second"]}';
var struct = eval('(' + string + ')');
```

  - But this can open a security hole
  - You are making browsers execute any code received from the server supposed to send you data only
  - Better not to use this method at all…

Serialization: Parsing JSON

- Use the **www.json.org** library's **parse()** method:

```
<script src="http://www.json.org/json2.js"></script>
<script language="Javascript">
var string = '{"name": "John Doe", "books": ["First", "Second"]}';
try {var struct = JSON.parse(string);}
catch (e) {alert("Error!" + e);}
</script>
```

- and **stringify()** method:

```
<script src="http://www.json.org/json2.js"></script>
<script language="Javascript">
var object = new Object();
obj.name = "John Doe";
obj.books = ["First", "Second"];
var json = JSON.stringify(object);
alert("JSON representation:" + json);
</script>
```

Serialization: Parsing JSON in PHP

- You may want to parse/generate JSON representations in PHP
  - PHP runs at the server side, generates structured data in JSON format
  - JavaScript runs at the client side, receives the JSON data, uses it to display something useful

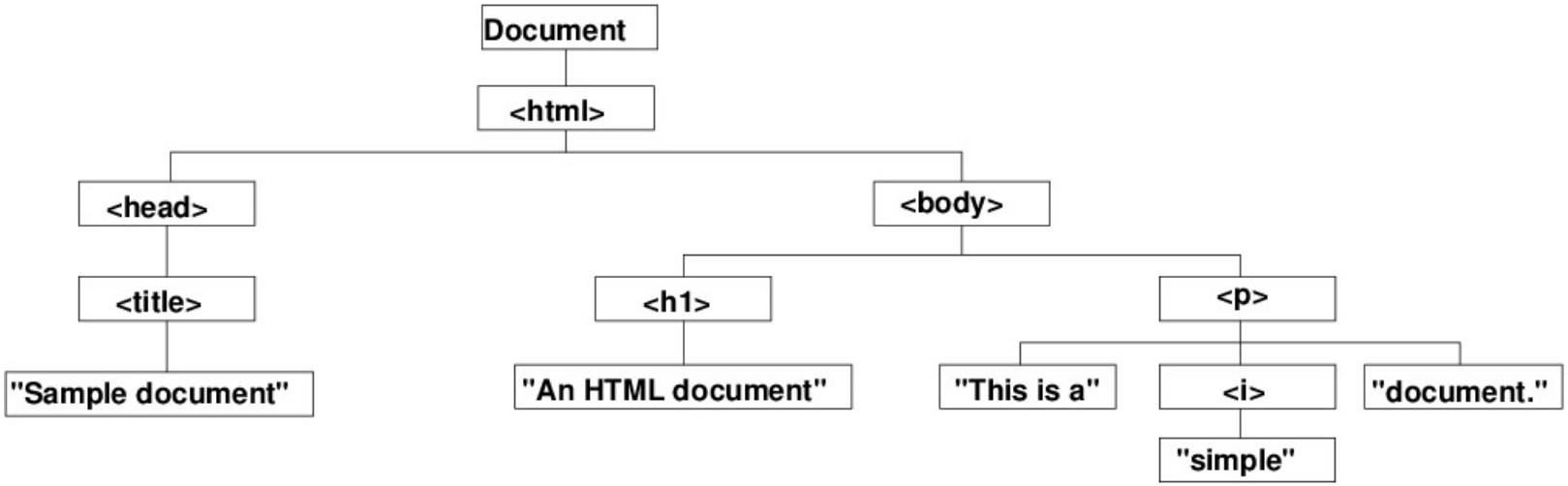
```
<html>
<head><title>PHP/JSON Example</title></head>
<body>
<?php
 $x1["name"] = "John Doe";
 $x1["books"] = array("First one", "Second one");
 echo "The JSON representation of string x1 is: " . json_encode($x1);
 // json_decode() is available too
?>
</body>
</html>
```

Updates: The DOM model

- DOM: **D**ocument **O**bject **M**odel
  - You write this:

```
<html>
  <head><title>Sample Document</title></head>
  <body>
    <h1>An HTML Document</h1>
    <p>This is a <i>simple</i> document.
  </body>
</html>
```

  - Your browser represents it internally as this:



Updates: Navigating the DOM tree

- **document**: standard object of type node; represents the current document
  - **document.childNodes**: array of sub-documents (explore it recursively to navigate the whole tree)
  - **document.nodeType**: represents the type of node:
    - **DOCUMENT_NODE**: the entire document
    - **ELEMENT_NODE**: element node (i.e., an HTML tag)
    - **TEXT_NODE**: text node (i.e., some text)
    - and a few others…
- For “element” nodes only:
  - **document.nodeName**: type of HTML tag (“HEAD”, “P”, etc.)
  - **document.attributes**: array of all attributes of the tag
- For “text” nodes only:
  - **document.data**: the actual text of that node

Updates: Navigating the DOM tree

- QUESTION: Count the number of HTML elements

```
<html>
<head><script>
  function countTags(n) {  // n must be a DOM node
    var numTags = 0;
    if (n.nodeType == Node.ELEMENT_NODE)    // Is this an HTML tag?
      numTags++;
    var children = n.childNodes;            // Get the list of children nodes
    for (var i=0; i<children.length; i++) { // Loop through the children
      numTags += countTags(children[i]);    // Recurse on each one
    }
    return numTags;
  }
</script></head>
<body onload="alert('This document has ' + countTags(document) + ' tags')">
  This is a <i>sample</i> document.
</body>
</html>
```

  - This document has 5 HTML elements (not 4!!)
    - **HTML**, **HEAD**, **SCRIPT**, **BODY**, **I**

Updates: Searching tags in a DOM tree

- Traversing the whole DOM tree can be quite painful if you are looking only for one specific tag
  - JavaScript allows you to search for a specific tag
  - Find tags by name:

```
var tables = document.getElementsByTagName("table");
// returns an array containing all "table" tags
alert("This document contains " + tables.length + " tables");
```

  - Find tags by identifier:
    - In the HTML code:

```
<table width="80%" border="0" id="myNiceTag”>
...
</table>
```

    - In the JavaScript section:

```
var myTable = document.getElementById(“myNiceTag”);
```

Updates: Changing text in a document

- Just edit the data field of the text node in the DOM tree

```
<html><head>
  <script language="JavaScript">
    function show() {
      var p = document.getElementById("editMe"); // Gives you the <p> tag
      var t = p.childNodes[0];                   // Gives you the text itself
      t.data = "3+4=7";
    }
  </script>
</head>
<body>
  <h1>A simple puzzle</h1>
  <p id="editMe">3+4=....? <button onclick="show();">Show the
answer!</button>
</body>
</html>
```

- Note: You cannot create new HTML tags like this!

```
t.data = "3+4 = <b>7</b>";
```

  - this will display "3+4 = 7", instead of "3+4 = **7**"

Updates: Changing attributes of tag

- If you want to change the attribute of an existing HTML tag
  - e.g., change <p id="x"> into <p id="x" align="center">

```
var p = document.getElementById("x");
p.setAttribute("align","center");
```

- Check the value of an attribute:

```
var p = document.getElementById("x");
var a = p.getAttribute("align");
```

- Remove an attribute:

```
var p = document.getElementById("x");
p.removeAttribute("align");
```

Updates: Changing DOM structure

- Create a new element (not connected to the DOM tree yet):

```
var b = document.createElement("b");        // Creates new <b> element
var t1 = document.createTextNode("hello"); // Creates new Text element
var t2 = document.createTextNode("HELLO"); // Creates new Text element
```

- Add a child element to a tag:

```
b.appendChild(t1);     // Add element t1 as the last child of b
b.insertBefore(t2,t1); // Inserts element t1 before t2 in b's children
var tag = document.getElementById("inserthere"); // Find right location
tag.appendChild(b);    // Insert the new tag in the document's DOM tree
```

- Replace an element with another:

```
b.replaceChild(t1,t2); // Replaces child node t1 with t2
```

- Remove an element:

```
b.removeChild(t2);
```

jQuery

JavaScript libraries

- Manipulating the DOM in pure JavaScript
  - Cumbersome
  - Hard to maintain
  - Error-prone
- Web Programming should be
  - Creative!
  - Fun!

JavaScript libraries

- A multitude of functionalities can be provided as add-ons:
  - DOM traversal
  - Data manipulation
  - Theming / Graphics
  - Animations / Effects
  - etc.

- A plethora of JavaScript libraries have emerged
  - jQuery: DOM & Data manipulation, AJAX, effects & animation
  - React: User interface
  - Ionic: Complete HTML5 framework
  - AngularJS: Enterprise framework for large web apps, easy binding of UI with JavaScript objects, sync, communication, etc.
  - D3.js: Data Visualization
  - Three.js, Meteor,

What is jQuery?

- A library that offers:
  - Data Manipulation
  - DOM Traversal and Manipulation
  - Events
  - AJAX
  - Effects and Animation

- **$**
  - The global jQuery function!
  - Also called “jQuery”

Why jQuery?

- In pure Javascript:

```
var elems = document.getElementsByTagName("img");
for (var i = 0; i< elems.length; i++) {
        elems[i].style.display = "none";
}
```

- Using jQuery:

```
$('img').hide();
```

Why jQuery?

- In pure Javascript:

```
var p = document.createElement('p');
p.appendChild(document.createTextNode('Hello World!'));
p.style.cssFloat = 'center';
p.style.backgroundColor = 'blue';
p.className = 'special';
document.getElementById("inserthere").appendChild(p);
```

- Using jQuery:

```
var p = $('<p>Hello World!</p>');
p.css({'float': 'center', 'background-color': 'blue'});
p.addClass('special');
$('#inserthere').append(p);
```

Why jQuery?

- This HTML code…

```
<p>Hello World!</p>
<p>This is a <b>jQuery</b> tutorial</p>
```

- …after this command…

```
$('p').addClass('myText');
```

- …turns into this:

```
<p class="myText">Hello World!</p>
<p class="myText">This is a <b>jQuery</b> tutorial</p>
```

jQuery Selectors

Selector	Return value
$("*")	all elements
$(this)	the current HTML element
$("p")	all <p> elements
$("#main")	the element with id="main"
$(".intro")	all elements with class="intro"
$("p.intro")	all <p> elements with class="intro"
$("p:first")	the first <p> element
$("ul li:first")	the first of the first
$("ul li:first-child")	the first or every
$("[href]")	all elements with a href attribute
$(":button")	all <button> elements

Object types

- jQuery selectors return **arrays** of DOM element objects

```
var paragraphs = $('p');     // an array
```

- We can use any of them as a regular DOM object

```
var para = paragraphs[0];    // a regular DOM object
```

- Or we can get hold of its jQuery encapsulation

```
var para2 = $(paragraphs[0]);    // a jQuery object
para2.html('this is the <b>new text</b> in my paragraph');
```

Animations, Effects, Widgets

- jQuery also supports:
  - animation of DOM objects
  - slide objects
  - fade objects in/out
  - drag
  - resize
  - progress bars
  - many many more!

Node.js

JavaScript: Why not on servers?!

- JavaScript is a very thoroughly designed language
  - Contrary to PHP, that was designed in an ad-hoc fashion!
- More productive to focus on one language for both sides
- Google' s V8 JavaScript Engine
  - One of the fastest for dynamic weakly-typed languages
- By now, huge ecosystem of modules

Node.js

- Event-driven architecture
  - Asynchronous I/O
  - Non-blocking I/O calls
  - Use of callbacks
- Single-threaded Event Loop
  - Single-threaded user experience
  - Easier, safer, faster development
- **Isn’t a single-threaded event loop a bottleneck?!**

Dissecting a Node.js App

A typical Node.js application has the following components:

- Server
  - Receives requests from browsers
- Router
  - Decides which function should handle each request
- Request Handlers
  - The workers doing the actual work
- View Logic
  - Deciding what to display on users' browsers after each request

Hello World!

- A very simple Hello World application

```
var http = require("http");

http.createServer(
  function(request, response)
  {
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World!");
    response.end();
  }
).listen(8888);
```

Hello World!

- A very simple Hello World application

```
var http = require("http");

function onRequest(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World!!!");
  response.end();
}

server = http.createServer(onRequest);

server.listen(8888);
```

Separating code into files

- server.js

```
var http = require("http");

function start() {
  function onRequest(request, response) {
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World!!!");
    response.end();
  }

  server = http.createServer(onRequest);
  server.listen(8888);
}

exports.start = start
```

- index.js

```
var server = require("./server");

server.start();
```

Placing logic in handlers

- Say we want to handle two URLs:
  - http://localhost:8888/one
  - http://localhost:8888/two

- It is better structured if we put the handlers' logic in a separate file

Placing logic in handlers

- handlers.js **(Watch out! Naive approach!)**

```
function one() {
  return "This is start handler ONE";
}

function two() {
  return "This is start handler TWO";
}

var handle = {}
handle['/'] = one;
handle['/one'] = one;
handle['/two'] = two;

function route(path) {
  if (path in handle)
    return handle[path]()
  else
    return "Path "+path+" not handled";
}

exports.route = route
```

Placing logic in handlers

- server.js **(Watch out! Naive approach!)**

```
var http = require("http");
var url = require("url");
var handlers = require("./handlers");

function start()
{
  function onRequest(request, response)
  {
    path = url.parse(request.url).pathname;
    message = handlers.route(path);

    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write(message);
    response.end();
  }

  server = http.createServer(onRequest);
  server.listen(8888);
}

exports.start = start
```

Placing logic in handlers

- Now imagine if a handler invokes a heavy call...

```
function one() {
  result = db("SELECT * FROM products WHERE ...");
  return "The result is " + result;
}

function two() {
  return "This is start handler TWO";
}

...
```

- **HINT: The event loop is single threaded! :-)**

Asynchronous Calls with Callbacks

- The synchronous, blocking call:

```
result = db("SELECT * FROM products WHERE ...");
```

- would be replaced by an asynchronous, non-blocking version, with a callback:

```
db("SELECT * FROM products WHERE ...",
    function(r)
    {
      result = r;
    }
  );
```

Asynchronous Calls with Callbacks

- So, our handler:

```
function one() {
  result = db("SELECT * FROM products WHERE ...");
  return "The result is " + result;
}
```

- would be replaced by the following one:

```
function one(request, response) {
  result = db("SELECT * FROM products WHERE ...",
    function(result) {
      response.writeHead(200, {"Content-Type": "text/plain"});
      response.write("The result is " + result);
      response.end();
    }
  );
}
```

Event-Based Model

- Traditional model:
  - Pass the data from deepest function (worker) all the way back to the caller
- Event-based Model:
  - Pass the function used for responding all the way down to the worker, and let him call it!

POST requests

- In POST requests, the data posted may be huge (think in MBs)
  - Is not delivered in one go (because then it would be blocking)
  - We need to specify callbacks for the chunks

- It makes sense to “collect” all post data in server.js, so we implement it once for all posts to all paths:

```
var postData = “”;

request.addListener("data", function(postDataChunk) {
  postData += postDataChunk;
});

request.addListener("end", function() {
  handlers.route(request, response, postData);
});
```

WebSockets

Pull Model

- The Web has been built around the request/response paradigm
  - Requests initiated by the browser
  - Responses sent by the web server
  - Its origins due to the Web's initially static content
  - Navigating from page to page
- AJAX (introduced in 2005) gave for the first time the option to dynamically fetch content from the server, without leaving the current page
  - The Request/Response paradigm is still there
  - Always initiated by the client
  - Based on the pull model: The client pulls information from the server
- What if the server needs to push data to the client?
  - Server-initiated notifications (e.g., Facebook, Gmail, ...)
  - Data streaming

Server-initiated Push

- Many names
  - Comet, Ajax Push, Reverse Ajax, HTTP Server Push, and more
- Usually based on the long poll technique
  - The client initiates a normal HTTP request (e.g., Ajax)
  - The server replies only when it has data to “push”
  - This is a persistent, long-lasting request
  - After the server replies, the browser initiates a new request
  - Gives the illusion of a push

WebSockets

- WebSockets are part of the HTML5 specification
- A **WebSocket** provides
  - A **full-duplex** channel between a browser and a web server

Handshaking

- To make a WebSocket connection, a client starts by making an HTTP request to the server

```
GET ws://websocket.example.com/MyWS HTTP/1.1
Origin: http://example.com
Connection: Upgrade
Host: websocket.example.com
Upgrade: websocket
```

- If the server supports the WebSocket protocol, it “upgrades” the connection

```
HTTP/1.1 101 WebSocket Protocol Handshake
Date: Wed, 16 Oct 2013 10:07:34 GMT
Connection: Upgrade
Upgrade: WebSocket
```

- Now, browser and web server may start sending data over the same TCP connection

WebSockets on Node.js

- Node.js does not include a “default” WebSocket implementation
  - You should pick one of the many libraries
  - E.g., ws, socket.io, sockjs, faye, etc.

- **ws** is one of the simplest and fastest libraries

```
var WebSocketServer = require('ws').Server;
var wss = new WebSocketServer({port: 8080, path: "/app/myWS"});

wss.on('connection', function(ws) {
  ws.on('message', function(message) {
    console.log('received: %s', message);
  });

  ws.send('something');
});
```

WebSockets on the browser

- Modern browsers support WebSockets by default

```
<script>
  var ws = new WebSocket('ws://www.host.com/app/myWS');

  ws.onmessage = function(event) {
    // event.data will contain the data received.
  });

  ws.send("Hello World!");
<script>
```

WebRTC

Browser: The Universal Client

- The Universal Client
  - GUI for virtually any application
  - Asynchronous events
  - Asynchronous communication
  - Data streams (WebSockets)
  - Local storage (caching)

- But something is still missing...
  - Skype / Viber / etc.
  - VoIP / Video calls
  - Browser-to-Browser communication!

- WebRTC comes to fill in the gap!
  - Adds **Real Time Communication** between browsers

WebRTC

- Developed by Google
  - Released as open source in May 2011
- Enables
  - Real-Time Communication
  - Audio, Video, Data
  - P2P Communication between browsers
  - NAT Traversal
- “WebRTC and HTML5 could enable the same transformation for real-time communications that the original browser did for information”
- JavaScript, Java, and Objective-C bindings

WebRTC APIs

- Acquiring local audio and video
  - Management of **local multimedia hardware**
  - Cameras, microphones, etc.

```
getUserMedia
```

- Communicating audio and video
  - Management of **multimedia streams**

```
RTCPeerConnection
```

- Communicating data
  - Send/Receive arbitrary **data structures**

```
RTCDataChannel
```

- Peer Connection
  - Setup connections (UDP/RTP streams) between browsers
  - NAT traversal

WebRTC APIs

WebRTC provides a JavaScript API, with three main categories:

- User Media API
  - Management of **local multimedia hardware**
  - Cameras, microphones, etc.
- Stream API
  - Management of **multimedia streams**
- Peer Connection API
  - **UDP/RTP streams** between browsers
  - NAT traversal

MediaStream (getUserMedia)

- Obtain a MediaStream with navigator.getUserMedia()
  - Place a video element in your DOM tree

```
<video id="localVideo" autoplay style="width:40%;"></video>
```

  - Then attach local camera's stream to it

```
var constraints = {video: true};

function successCallback(stream) {
  var video = document.querySelector("#localVideo");
  video.src = window.URL.createObjectURL(stream);
}

function errorCallback(error) {
  console.log("navigator.getUserMedia error: ", error);
}

navigator.getUserMedia(constraints, successCallback, errorCallback);
```

MediaStream (getUserMedia)

- Examples
  - http://simpl.info/getusermedia/
  - http://idevelop.github.com/ascii-camera/
  - http://webcamtoy.com

Constraints Object

- Controls the contents of the MediaStream
  - Media type, resolution, aspect ratio, frame rate

- Example
  - In all cases keep a 4:3 aspect ratio
  - If possible 60 fps, and in that case limit the max resolution

```
var constraints = {
  video: {
    mandatory: { minAspectRatio: 1.333, maxAspectRatio: 1.334 },
    optional [
      { minFrameRate: 60 },
      { maxWidth: 640 },
      { maxHeigth: 480 }
    ]
  }
}
```

MediaStream: Audio

- Acquire just the local audio input stream

```
var constraints = {audio: true};

function successCallback(stream) {
  var audioContext = new webkitAudioContext();

  // Create an AudioNode from the stream
  var source = audioContext.createMediaStreamSource(stream);

  // Connect it to the destination or any other node for processing!
  source.connect(audioContext.destination);
}

navigator.getUserMedia(constraints, successCallback);
```

RTCPeerConnection

- Lots of functionality!
  - Signal processing
  - Codec handling
  - P2P communication
  - Security
  - Bandwidth management

RTCPeerConnection Example

```
var pc = new RTCPeerConnection(null);
pc.onaddstream = gotRemoteStream;
pc.addStream(localStream);
pc.createOffer(gotOffer);

function gotOffer(desc) {
  pc.setLocalDescription(desc);
  sendOffer(desc);
}

function gotAnswer(desc) {
  pc.setRemoteDescription(desc);
}

function gotRemoteStream(e) {
  attachMediaStream(remoteVideo, e.stream);
}
```

RTCDataChannel

- Same API as WebSockets
  - Arbitrary data structures
- Stream Control Transmission protocol (SCTP)
  - Configurable

	TCP	UDP	SCTP
Reliability	YES	NO	CONFIG
Ordered Delivery	YES	NO	CONFIG
Transmission	Byte	Message	Message
Flow control	YES	NO	YES
Congestion Control	YES	NO	YES

RTCDataChannel

- Same API as WebSockets

```
var pc = new RTCPeerConnection();
var dc = pc.createDataChannel("my send data channel");

dc.onmessage = function(event) {
  console.log("received : " + event.data );
};

dc.onopen = function() {console.log("send datachannel open");};

dc.onclose = function() {console.log("send datachannel close");};

pc.ondatachannel = function(event) {
  console.log("new receive datachannel") ;
  receiveChannel = event.channel;
  receiveChannel.onmessage = function(event) {
    console.log("received message" + event.data);
  };
};
```

RTCDataChannel Example

```
var pc = new RTCPeerConnection(servers,
  {optional: [{RtpDataChannels: true}]});

pc.ondatachannel = function(event) {
  receiveChannel = event.channel;
  receiveChannel.onmessage = function(event){
    document.querySelector("div#receive").innerHTML = event.data;
  };
};

sendChannel = pc.createDataChannel("sendDataChannel", {reliable:
false});

document.querySelector("button#send").onclick = function (){
  var data = document.querySelector("textarea#send").value;
  sendChannel.send(data);
};
```

Signaling

- Communication is P2P...
  - ...BUT, it needs a non-P2P way to bootstrap!
  - i.e., it needs a server
- What type of server?!
  - Web server?
  - RPC / RMI
  - Custom socket server?
  - snail mail? ☺
  - ANYTHING!
- WebRTC Signaling is abstract
  - Use any messaging mechanism / protocol
  - Typically we use a web server's WebSocket

NAT Traversal

- Ideal case
  - browsers have publicly accessible IP
- Typical case
  - browsers are behind NAT/firewall
  - Use STUN server
- Worst case
  - browsers are behind very strict firewall
  - Use TURN server

ICE

- Interactive Connectivity Establishment (ICE)
  - Tries all possible ways
  - Finds best way to connect

```
{
  ' iceServers ': [
    {
      'url ': ' stun : stun .l. google . com :19302 '
    },
    {
      'url ': ' turn :192.158.29.39:3478? transport =udp ',
      ' credential ': ' JZEOEt2V3Qb0y27GRntt2u2PAYA =',
      ' username ': '28224511:1379330808 '
    },
    {
      'url ': ' turn :192.158.29.39:3478? transport =tcp ',
      ' credential ': ' JZEOEt2V3Qb0y27GRntt2u2PAYA =',
      ' username ': '28224511:1379330808 '
    }
  ]
}
```

Creating Offers

```
function start(isCaller) {
    peerConnection = new RTCPeerConnection(peerConnectionConfig);
    peerConnection.onicecandidate = gotIceCandidate;
    peerConnection.onaddstream = gotRemoteStream;
    peerConnection.addStream(localStream);
    if(isCaller) {peerConnection.createOffer(gotDescr, createOfferErr);}
}

function gotDescr(description) {
    peerConnection.setLocalDescription(description, function () {
        serverConnection.send(JSON.stringify({'sdp': description}));
    }, function() {console.log('set description error')});
}

function gotIceCandidate(event) {
    if(event.candidate != null) {
        serverConnection.send(JSON.stringify({'ice': event.candidate}));
    }
}

function gotRemoteStream(event) {
    remoteVideo.src = window.URL.createObjectURL(event.stream);
}

function createOfferErr(error) { console.log(error); }
```

Lots of material online!

- https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API
- https://shanetully.com/2014/09/a-dead-simple-webrtc-example/
- https://www.html5rocks.com/en/tutorials/webrtc/basics/
- http://io13webrtc.appspot.com/

That's all folks!