8. HTTP and Server-Side Processing

8.1. HTTP/1.0 Overview

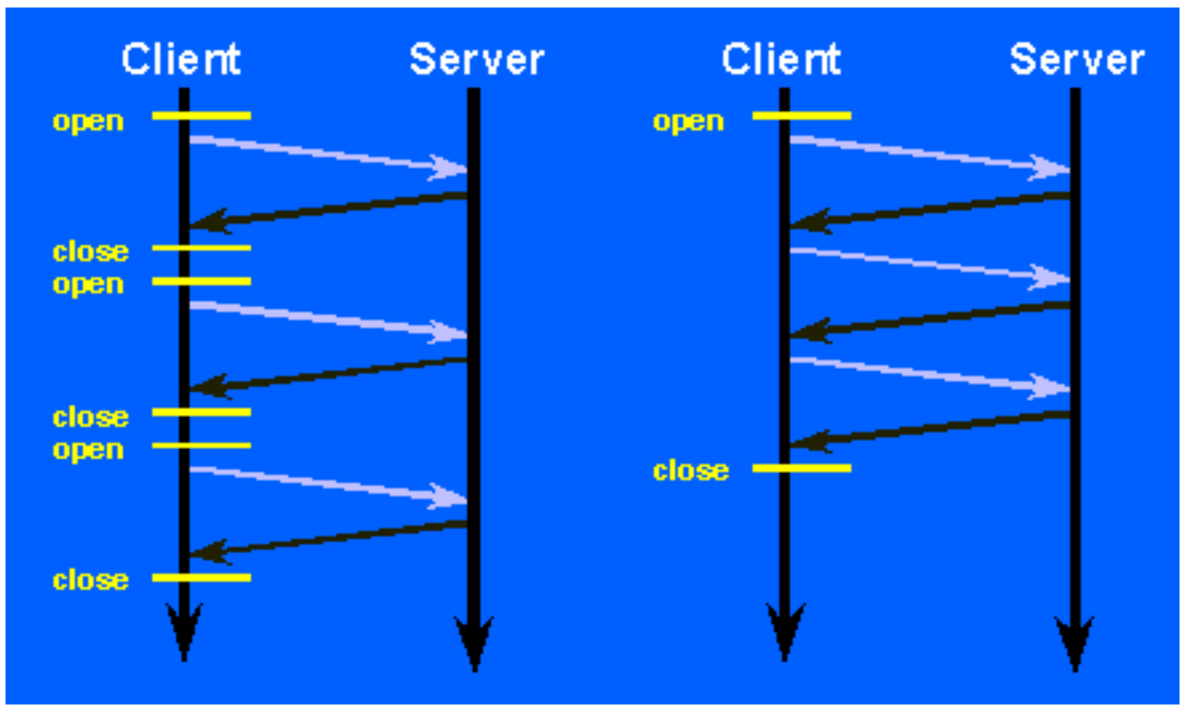
- application-layer transfer protocol used by browsers to interact with web servers
- normally runs over TCP
- defined in RFC 2616
- based on client/server architecture
- HTTP is *stateless*: servers retain no information about past requests
- interaction between client and server has 4 phases:
    - client connects to server
    - client sends request to server
    - server sends response to client
    - server closes connection (although *persistent* connections are possible)

8.2. HTTP Interaction

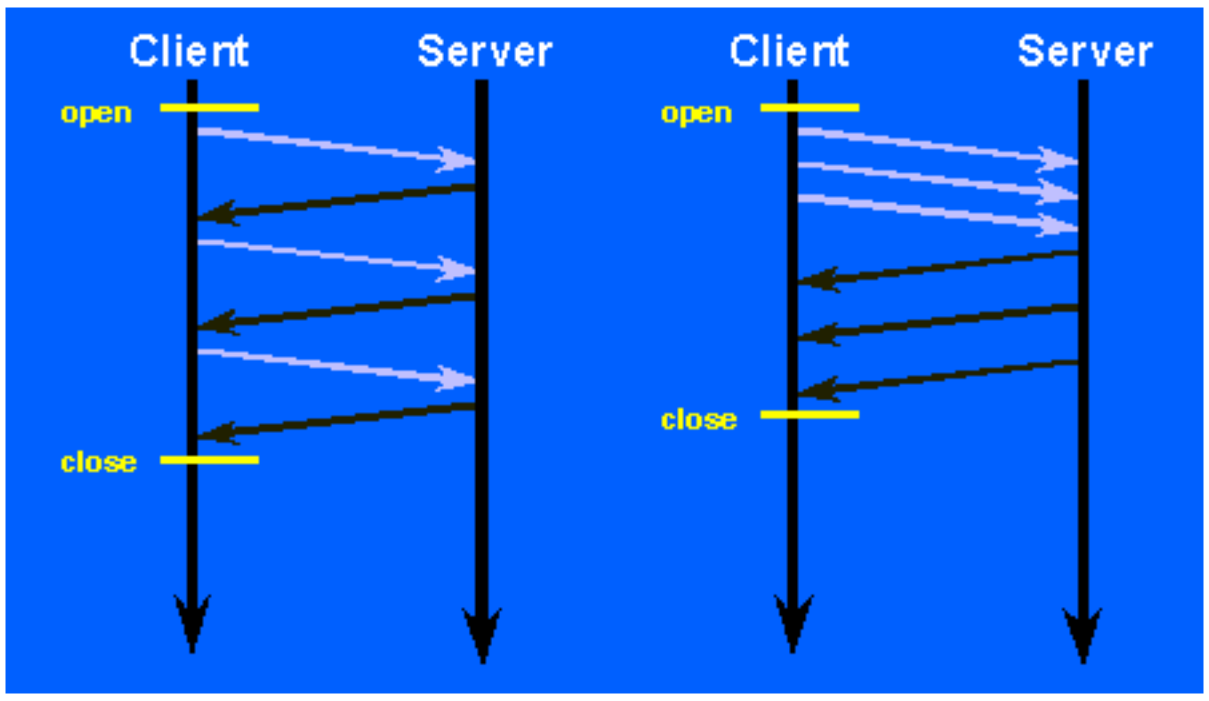
- assume we enter the URL `http://www.dcs.bbk.ac.uk/news/` into the address bar of a browser
- the browser (HTTP client) initiates a TCP connection to the server `www.dcs.bbk.ac.uk` on port 80
- the browser sends an HTTP request message to the server asking for resource `/news/`
- the HTTP server retrieves the resource, encapsulates it in an HTTP response message, and sends this to the browser
- the HTTP server tells TCP to close the connection
- the browser receives the response
- if the HTML in the response includes images, the process has to repeat

8.3. HTTP/1.1 Persistent Connections

- poor performance of HTTP/1.0 is due to a separate connection for each request
- *persistent* connections send multiple request and response interactions over a single TCP connection
- this results in improved performance
- persistent connections are the default



8.4. Pipelining



- *pipelining* can be used over an HTTP persistent connection
- allows a client to make multiple requests without waiting for each response
- this results in better utilisation of connection
- the server processes requests concurrently
- in principle, server could send responses in the order requests complete which would minimise waiting times
- but HTTP has no way of identifying a response with a request (it is *stateless*)
- so the specification states that the server must send responses in the *same* order as requests

8.5. HTTP Client Requests

- each *client request* message has the format:

```
method URL-path HTTP-version (request-line)
headers (0 or more lines)
<blank line> (CRLF)
message-body (only if a POST or PUT method)
```

- some *request methods*:
  - `GET`: request document named by `URL-path`
  - `HEAD`: return only header information of `URL-path` (e.g., test for validity, recent modification)
  - `POST`: submit information to entity on server given by `URL-path`
  - `PUT`: server will replace entity given by `URL-path`
- example of a request-line:

```
GET /index.html HTTP/1.0
```

`URL-path` includes optional query string or fragment identifier (anchor)

8.6. HTTP Server Responses

- each *server response* message has format:

```
HTTP-version status-code reason-phrase (status-line)
headers (0 or more lines)
<blank line> (CRLF)
message-body
```

- example of status-line:

```
HTTP/1.0 200 OK
```

- response includes data as `message-body` if request successful; otherwise `reason-phrase` states why unsuccessful
- examples of status codes and reason phrases:

Status Code	Reason Phrase
200	OK
401	Unauthorized
404	Not Found
500	Internal Server Error

8.7. Some request and response headers

- form of each HTTP header is `field: value`
- (some) client *request* headers
  - `Host:` the domain name (and port) of the server; required in every request; allows server to differentiate requests for multiple hosts with same IP address
  - `User-Agent:` information about the client program (type, version)
  - `Accept:` formats acceptable to the client, given using MIME types
- (some) server *response* headers
  - `Server`: name and version of server
  - `Content-Type`: the (MIME) media type of the resource being returned
  - `Content-Length`: size of message body in bytes
  - `Last-Modified`: date and time when entity was last modified

8.8. Example of request and response

Machine responses are `this colour` below:

```
Peter-Woods-MacBook-Pro:~ ptw$ telnet www.dcs.bbk.ac.uk 80
Trying 193.61.29.21...
Connected to www.dcs.bbk.ac.uk.
Escape character is '^]'.
GET / HTTP/1.0

HTTP/1.1 200 OK
Date: Fri, 18 Nov 2011 17:44:06 GMT
Server: Apache/2.2.16 (Unix) mod_ssl/2.2.16 OpenSSL/0.9.8o DAV/2 SVN/1.6.5 mod_fcgid/2.3.6 mod_perl/2.0.4 Perl/v5.
Connection: close
Content-Type: text/html

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitiona
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<title>Computer Science and Information Systems Birkbeck University of London</title>

...

</body>
</html>

Connection closed by foreign host.
Peter-Woods-MacBook-Pro:~ ptw$
```

8.9. Example using HTML form

- servers don't only deliver (static) documents
- they can also run programs to generate content
- usually based on user input via an HTML form, e.g.:
- Who invented the WWW (surname only)? Submit answer Erase answer

8.10. HTML for form

```
<form action="http://titan.dcs.bbk.ac.uk/~ptw/teaching/IWT/server/www-inventor.php"
      method="GET">
  <label for="www-inventor">Who invented the WWW (surname only)?</label>
  <input type="text" id="www-inventor" name="inventor" />
  <input type="submit" value="Submit answer" />
  <input type="reset" value="Erase answer" />
</form>
```

- `action` gives URI of PHP script (see later) to run
- `method` used is `GET`
- only one input control with `name` attribute
- so `?inventor=Berners-Lee`, e.g., is added to URL-path

8.11. HTML form processing

- HTML `form` element has
  - `action` attribute whose value is the URI for processing agent
  - `method` attribute whose value specifies the HTTP method to use for sending data: `GET` or `POST`
- form input data is sent to the server as `name=value` pairs
  - `name` is the value of the `name` attribute of an input control
  - `value` is the value of the corresponding `value` attribute or is entered by the user
- `name=value` pairs are separated by `&` characters
- for `GET`, `name=value` pairs form the URI query string
- for `POST`, `name=value` pairs form the message body

8.12. Sessions and Cookies

- a *session* is a sequence of related interactions between a client and a server
- HTTP is a *stateless* protocol
  - no way of storing current state of a session, e.g., contents of shopping cart
- one solution is to use *cookies*

8.13. Cookies

- small data items stored on clients (web browser) but managed by servers
- can track users independently of sessions
- cookies are exchanged using HTTP headers
- server can set a cookie as part of an HTTP response, e.g.,

  ```
  Set-Cookie: sessionid=724A3B616C0D89B3B521C70612
  ```

- client (web browser) will store this cookie
- in subsequent requests to the same server, client will include HTTP request header, e.g.,

  ```
  Cookie: sessionid=724A3B616C0D89B3B521C70612
  ```

8.14. Server-side processing technologies

- web servers, such as Apache, usually include support for programming languages such as
  - *Perl*
  - *PHP*
- as well as very old techniques such as
  - *server-side include* (SSI) pages
  - *common gateway interface* (CGI) programs
- Apache Tomcat, e.g., also supports
  - Java *servlets*
  - *Java server pages* (JSP)
- frameworks for building web applications include
  - *ASP.NET* (C#, Microsoft)
  - *Django* (Python)
  - *Express* (uses *Node.js*, Javascript)

8.15. Node.js

- Node.js is a Javascript runtime environment based on Google Chrome's V8 Javascript engine
- uses an event-driven non-blocking model of programming
- includes modules to implement a web server

8.16. Node.js example

- let web.js contain the following Javascript code:

```
var http = require('http');
var fs = require('fs');
var port = 8080;

function process_request(request, response) {
   response.writeHead(200);
   response.end(fs.readFileSync(__dirname + request.url));
}

var server = http.createServer(process_request);
server.listen(port);

console.log('Listening on port ' + port);
```

- `http` is the web server module
- `fs` is the file system module
- `createServer` launches a web server, and is passed a function to call when requests arrive
- `process_request` takes the request and produces a response
- the response is simply the contents of the file (`request.url`)

8.17. Running and using Node

- to run the web server specified by `web.js`, we enter the following at the command line

```
node web.js
```

- we get the following message displayed

```
Listening on port 8080
```

- in another shell (command) window, we enter

```
curl -i http://localhost:8080/test.html
```

(`curl` is a Unix command for retrieving URLs from the command line; `-i` includes headers)
- the response is

```
HTTP/1.1 200 OK
Date: Mon, 02 Mar 2020 11:35:04 GMT
Connection: keep-alive
Transfer-Encoding: chunked

<html>
 ... contents of test.html ...
</html>
```

8.18. PHP

- *PHP: Hypertext Preprocessor* (PHP) is an open-source, server-side scripting language
- variables don't have to be declared and are not strongly typed
- uses `<?php` and `?>` delimiters for PHP code
- delimiters differentiate PHP code from static HTML
- server needs to have PHP installed
- comes as part of Apache web server

8.19. PHP code for www-inventor.php

```
<html>
<head><title>WWW Inventor</title></head>
<body>
<h1>
<?php
if ($_GET['inventor'] == "Berners-Lee")
   echo "Congratulations, you answered correctly.";
else
   echo "Sorry, the correct answer is Berners-Lee.";
?>
</h1>
</body>
</html>
```

- variable names start with `$`
- built-in variable `$_GET` is an associative array providing access to the values of `name=value` pairs in the query string
- so `$_GET['inventor']` gets value of `inventor` parameter in query string
- output passed to server via standard output - using `echo` or `print`

8.20. Processing XML with PHP (1)

- very simple application: an XML file containing acronyms:

```
<dictionary>
  <acronym>AJAX</acronym>
    ...
  <acronym>XSLT</acronym>
</dictionary>
```

- want to transform the XML on the server using a stylesheet and send the resulting HTML output to the browser
- stylesheet produces an unordered list with each list item containing an acronym:

```
<xsl:template match="/dictionary">
  ...
  <ul>
    <xsl:for-each select="acronym">
      <li>
        <xsl:value-of select="."/>
      </li>
    </xsl:for-each
  </ul>
  ...
</xsl:template>
```

8.21. Processing XML with PHP (2)

- PHP code in `acronyms.php` is:

```
<?php
$xmlDoc = new DomDocument();
$xmlDoc->load("acronyms.xml");

$xslDoc = new DomDocument();
$xslDoc->load("acronyms.xsl");

$processor = new XSLTProcessor();
$xslDoc = $processor->importStylesheet($xslDoc);

$htmlDoc = $processor->transformToDoc($xmlDoc);
print $htmlDoc->saveXML();
?>
```

- `DomDocument` and `XSLTProcessor` are PHP objects
- `load`, `importStylesheet` and `transformToDoc` are PHP methods
- `transformToDoc` uses the associated stylesheet to return an HTML/XML document
- the `saveXML` method converts a DOM tree to a string

8.22. Retrieving JSON

- information about Nobel prizes is available at `https://www.nobelprize.org`
- they also provide an API for retrieving information about prizes in JSON
- e.g., to retrieve the winners from 1991, use the URL `http://api.nobelprize.org/v1/prize.json?year=1991`
- query parameters include
  - `year`: year in which prizes were awarded
  - `yearTo`: ending year for a range of years (`year` required)
  - `category`: one of the 6 categories
  - `numberOfLaureates`: filter prizes by number of winners sharing the prize

8.23. REST

- the Nobel prize API is what is called a *REST* API
- *REST* (REpresentational State Transfer), or RESTful, web services provide
  - access to and manipulation of web resources
  - using a uniform and predefined set of stateless operations
  - when using HTTP, these operations include `GET`, `POST`, `PUT`, `DELETE`
  - all information is encoded in the URLs and resources returned
- typically
  - a URI identifies a resource
  - `GET` will retrieve a resource
  - `POST` will create a new resource
  - `PUT` will overwrite an existing resource
  - `DELETE` will delete a resource

8.24. PHP and JSON

- let's use a simple form to pass a year value to the Nobel prize API
- it is sent via a PHP script (next slide) which also turns the JSON response into HTML
- the `name` of the textbox in the form is `year`
- the `action` value is `http://titan.dcs.bbk.ac.uk/~ptw/teaching/IWT/server/nobel-year.php`
- Enter a year: [ ] Submit answer Erase answer

8.25. PHP code for Nobel Prizes (1)

```
<html>
<body>
<h1>Nobel Prize Winners</h1>
<?php
  $year = $_GET['year'];
  if ($year >= 1901 && $year <= 2017) {
    $url = 'http://api.nobelprize.org/v1/prize.json?year=' . $year;
    $string = file_get_contents($url);

    # Read the JSON output into an associative array
    $result  = json_decode($string, true);

    print "<p>In $year, the prizes were awarded as follows:</p><ul>\n";
     ...  # see next slide
    print "</ul>";
  }
  else {
    print "<p>Year value out of range; years range from 1901 to 2017</p>";
  }
?>
</body>
</html>
```

- we previously used `$_GET` (on slide 15)
- string concatenation in PHP uses `.`
- `file_get_contents` returns file contents as a string
- `json_decode` with second parameter set to `true` returns JSON string as an associative array
- variable references, such as `$year`, inside strings delimited by *double quotes* are dereferenced (but not for single quotes)

8.26. PHP code for Nobel Prizes (2)

returned JSON data is as follows (for `year` 1991):

```
{"prizes":
 [
  {"year":"1991",
   "category":"physics",
   "laureates":[
    {"id":"141",
     "firstname":"Pierre-Gilles",
     "surname":"de Gennes",
     "motivation":"...",
     "share":"1"}]},
  {"year":"1991",
   "category":"chemistry",
   "laureates":[
    {"id":"276",
     "firstname":"Richard R.",
     "surname":"Ernst",
     "motivation":"...",
     "share":"1"}]},
  {"year":"1991",
   "category":"medicine",
   "laureates":[
    {"id":"444",
     "firstname":"Erwin",
     "surname":"Neher",
     "motivation":"...",
     "share":"2"},
    {"id":"445",
     "firstname":"Bert",
     "surname":"Sakmann",
     "motivation":"...",
     "share":"2"}]},
  {"year":"1991",
   "category":"literature",
   "laureates":[
    {"id":"668",
     "firstname":"Nadine",
     "surname":"Gordimer",
     "motivation":"...",
    "share":"1"}]},
  {"year":"1991",
   "category":"peace",
   "laureates":[
    {"id":"553",
     "firstname":"Aung San Suu Kyi",
     "motivation":"...",
     "share":"1",
     "surname":""}]},
  {"year":"1991",
   "category":"economics",
   "laureates":[
    {"id":"707",
     "firstname":"Ronald H.",
     "surname":"Coase",
     "motivation":"...",
     "share":"1"}]}
 ]
}
```

8.27. PHP code for Nobel Prizes (3)

- recall `$result` contains the JSON data as an associative array
- the rest of the PHP code is as follows:

```
# Find out how many prizes are listed
$num_prizes = count($result['prizes']);

for ($i = 0; $i < $num_prizes; $i++) {

  # Print out the category
  $cat = $result['prizes'][$i]['category'];
  print "<li>in $cat to <ul>\n";

  # Find out how many winners in this category
  $num_winners = count($result['prizes'][$i]['laureates']);

  for ($j = 0; $j < $num_winners; $j++) {

    # Print out the names
    $firstname = $result['prizes'][$i]['laureates'][$j]['firstname'];
    $surname = $result['prizes'][$i]['laureates'][$j]['surname'];
    print "<li>$firstname $surname </li>\n";

  }
  print "</ul></li>\n";
}
```

- `count` counts number of elements in an array

8.28. Ajax

- *Ajax* = Asynchronous Javascript and XML
- Ajax is used by many websites to implement responsive and dynamic web applications
- examples include Google's Gmail, Suggest and Maps services
- based on the `XMLHttpRequest` object (built in to browsers)
- no longer restricted to retrieving XML
- used by the jQuery `get` method:
  `$.get( url [, data ] [, success ] [, dataType ] )` is shorthand for:

  ```
  $.ajax({
    url: url,
    data: data,
    success: success,
    dataType: dataType
  });
  ```

  both return a `jqXHR` object (a superset of `XMLHttpRequest`)

8.29. XMLHttpRequest object

- `XMLHttpRequest` object (XHR for short) is a WHATWG specification
- WHATWG = Web Hypertext Application Technology Working Group
- using standard DOM, we can send a request as follows:

```
var xhr = new XMLHttpRequest();
xhr.open("GET", url, false);
xhr.send(null);
var xmldoc = xhr.responseXML;
```

- first create an `XMLHttpRequest` object

- the `open` method sets up an HTTP request:
    - the first argument is an HTTP method (`GET` in this case)
    - the second argument is the URL of the XML document
    - the third argument specifies whether the script executes asynchronously with the request or not
    - `false` means synchronous: interpreter waits for response before continuing (not recommended)
- the `send` method sends the HTTP request to the server; content is null for `GET`
- `responseXML` property makes response available as XML (`responseText` can also be used)

8.30. Asynchronous requests

- when calls to `open` are *asynchronous*, we need to specify an event handler
- `XHR` has a `readyState` property and an `onreadystatechange` event
- values for `readyState` are:
    - 0 = unsent - open() has not yet been called
    - 1 = opened - send() has not yet been called
    - 2 = headers received - send() has been called, headers in response are available
    - 3 = loading - the response's body is being received
    - 4 = loaded - finished
- we must specify the function to call when `readyState` changes:

```
var xhr = new XMLHttpRequest();
xhr.open("GET", url, true);
xhr.onreadystatechange = function;
...
```

8.31. Cross-Origin Resource Sharing (CORS)

- when issuing requests, the request must be to the same *origin* (domain, protocol, port) as the loaded page
- this is to try to prevent *cross-site scripting* which can result in security vulnerabilities
- *Cross-Origin Resource Sharing* (CORS) is a mechanism to grant an application permission to access selected resources from a server at a different origin
- uses additional HTTP headers, including:
    - `Origin` in the request
    - `Access-Control-Allow-Origin` in the response
- `Origin` specifies the domain name origin of the request
- `Access-Control-Allow-Origin` value can be `*` for any origin, or the single origin in the request

8.32. CORS in PHP

- server-side programming environments (e.g., PHP) provide
    - access to HTTP request headers
    - ability to set HTTP response headers
- consider the following fragment of PHP:

```
if($_SERVER['HTTP_ORIGIN'] == "http://www.dcs.bbk.ac.uk") {
    header('Access-Control-Allow-Origin: http://www.dcs.bbk.ac.uk');
    ...
}
```

    - `$_SERVER` is an associative array of values available from the server
    - `HTTP_ORIGIN` gives the value of the `Origin:` header
    - `header()` function sets an HTTP header to be sent with the response

8.33. Google Suggest lookalike

- example taken from Dynamic AJAX Suggest Tutorial
-
- code for form above is:

  ```
  <form>
  <input type="text" id="txtSearch" name="txtSearch"
    onkeyup="searchSuggest();" autocomplete="off" />
  <div id="search_suggest">
  </div>
  </form>
  ```

- text box has `id="txtSearch"`
- `searchSuggest()` (next slide) is called in response to an `onkeyup` event
- `autocomplete="off"` turns off browser default to suggest recently entered data
- `div` with `id="search_suggest"` is where suggestions will appear

8.34. searchSuggest function

```
var searchReq = getXmlHttpRequestObject();

function searchSuggest() {
 if (searchReq.readyState == 4 || searchReq.readyState == 0) {
  var str = escape(document.getElementById('txtSearch').value);
  searchReq.open("GET", 'searchSuggest.php?search=' + str, true);
  searchReq.onreadystatechange = handleSearchSuggest;
  searchReq.send(null);
 }
}
```

- `getXmlHttpRequestObject` just returns an XMLHttpRequest object
- text box in form has `id="txtSearch"`
- note that calls to `open` are asynchronous
- so we must specify the function to call (`handleSearchSuggest`) when `readyState` changes

8.35. searchSuggest.php (1)

```
<?php
header("Content-Type: application/json; charset=utf-8");
...
$suggestions = array("Ajax","ASP","Javascript","JSP","XML","XPath","XSD","XSLT");
$result = array();

// Make sure that a value was sent.
if (isset($_GET['search']) && $_GET['search'] != '') {
   $search = $_GET['search'];
   ... // see next slide
}
?>
```

- set header to specify contents in response is JSON
- `isset` function checks that a value is not NULL
- `$result` is an array

8.36. searchSuggest.php (2)

```
...
   foreach ($suggestions as $suggestion) {
      // Add each suggestion to the $result array
      if (strpos($suggestion,$search)===0)
         $result[] = $suggestion;
   }
   echo json_encode($result);
...
```

- `foreach` allows iteration through array
- `strpos` returns first position where second argument is found in first argument, or `FALSE` if not found
- since `FALSE==0` is true in PHP, need to use strict comparison `===` to check types are equal too
- if value for `$search` matches `$suggestion` at 0'th position
  - add `$suggestion` to `$result` array
- output the `$result` array as JSON using `json_encode`

8.37. handleSearchSuggest function

```
//Called when the AJAX response is returned.
function handleSearchSuggest() {
 if (searchReq.readyState == 4) {
  var ss = document.getElementById('search_suggest');
  ss.innerHTML = '';
  var str = JSON.parse(searchReq.responseText); // PTW modified for JSON
  // For loop to build list of suggestions goes here (next slide)
 }
}
```

- `div` element under textbox in form has `id="search_suggest"`
- note the use of `responseText` since response is text (in JSON format)
- `JSON.parse()` parses JSON text, returning a JSON type
- `for` loop is on the next slide

8.38. handleSearchSuggest function (for loop)

```
...
  // Returned suggestions are in array str
  for(i=0; i < str.length; i++) {
   //Build our element string.  This is cleaner using the DOM,
   //but IE doesn't support dynamically added attributes.
   var suggest = '<div onmouseover="javascript:suggestOver(this);" ';
   suggest += 'onmouseout="javascript:suggestOut(this);" ';
   suggest += 'onclick="javascript:setSearch(this.innerHTML);" ';
   suggest += 'class="suggest_link">' + str[i] + '</div>';
   ss.innerHTML += suggest;
  }
...
```

- each suggestion is output as a `div` element with `mouseover`, `mouseout` and `onclick` events
- `suggestOver`, `suggestOut` and `setSearch` are defined on next slide

8.39. Mouse over, mouse out and click functions

```
//Mouse over function
function suggestOver(div_value) {
  div_value.className = 'suggest_link_over';
}
//Mouse out function
function suggestOut(div_value) {
  div_value.className = 'suggest_link';
}
//Click function
function setSearch(value) {
  document.getElementById('txtSearch').value = value;
  document.getElementById('search_suggest').innerHTML = '';
}
```

- `suggest_link_over` is a CSS class value which highlights the suggestion
- `suggest_link` unhighlights the suggestion
- `setSearch()` enters the value in the search box and clears the suggestions
- all code is in [`ajax_search.js`]() and the CSS file [`suggest.css`]()

8.40. Exercises

- Combine the functionality of the [Google Suggest lookalike]() with that for [retrieving acronyms in XML]() in order to achieve the following. When a user types into the textbox, the suggestions are retrieved from an XML file on the server, rather than being hardcoded into the program. Note that PHP has a `getElementsByTagName` method which returns a `DOMNodeList`. A `DOMNodeList` has a `length` property and an `item` method which takes an index value and returns a `DOMNode`. A `DOMNode` has `firstChild` and `nodeValue` properties (see the [PHP DOMDocument Class]() and links from there).
- Rewrite the Javascript code in [`ajax_search.js`]() for the [Google Suggest lookalike]() so that it uses jQuery.

8.41. Links to more information

- [RFC2616]() which specifies HTTP/1.1
- [Wikipedia article on HTTP]()
- [Wikipedia article on REST]()
- [PHP home page]()
- [jQuery `get` method]()
- [Cross-Origin Resource Sharing]()
- [Server-side web frameworks]()

HTTP is covered briefly in Section 4.6 of [Comer]. Section 7.3 of [Tanenbaum] covers HTML forms and HTTP, and mentions CGI and PHP. Servlets are covered in Chapter 9 and JSP in Chapter 10 of [Moller and Schwartzbach]. Server-side processing of XML using PHP (and VB.NET) is covered in Chapters 11 (12) and 13 in [Jacobs]. AJAX is covered in Chapter 9 in [Jacobs]. It is also mentioned in Section 7.3.3 of [Tanenbaum].