Internet programming, notes for lecture 2

Table of Contents

Client - server interaction, Server side programming, 22.3.

Contents:

References:

World Wide Web: Beyond the Basics
Abrams, Marc. Prentice-Hall, 1998, ISBN 0-13-954785-1 (draft is available at http://ei.cs.vt.edu/~wwwbtb/book/)

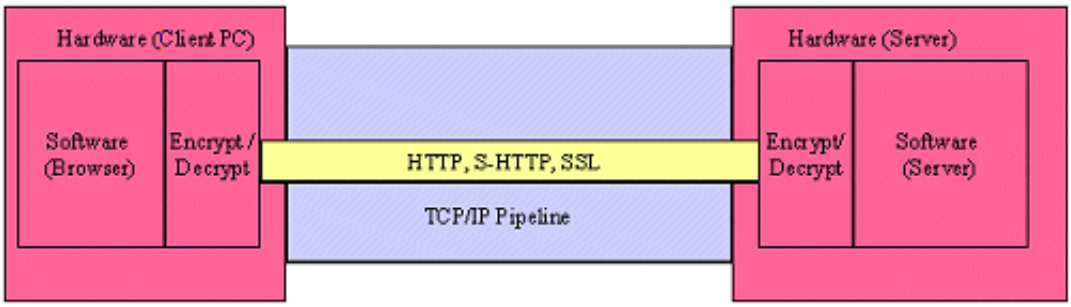
CGI Programming in C & Perl.
Boutell, Thomas. Addison-Wesley 1996 (in LUT library)

HTTP - Hypertext Transfer Protocol

How the Web Works

After the user has specified what he wishes to obtain from the web, that request has to be transmitted from the web browser to the web server. To do this, the information has to pass through the *Communication Layer*.

There are a host of technologies which all poke their way into the Communication Layer. These are not concern of our lecture, though. It is enought for us to know, that the main idea about the Communication Layer is that information needs to get from one place to another and that data needs to get there in one piece, in the right order, and with no parts of that data changed or corrupted. In addition, the information could/should be transferred securely.



HTTP - HyperText Transfer Protocol

HTTP is the network protocol used to deliver virtually all data - HTML files, image files, query results, etc. - on the World Wide Web. Usually, HTTP takes place through TCP/IP sockets.

Structure of HTTP Transactions

HTTP uses the client-server model: An HTTP client opens a connection and sends a request message to an HTTP server; the server then returns a response message, usually containing the resource that was requested.

After delivering the response, the server closes the connection or accepts a new request over the same connection; this depends on a version of HTTP protocol. HTTP is a stateless protocol, i.e. it does not maintain any connection information between transactions.

Versions of HTTP

HTTP/0.9
: The original HTTP developed in 1991 by Tim Barners-Lee.
Very simple minded protocol - its main purpose was to do raw data transfer.
Make connection, initiate request and get back response. All the responses were simply streams of ASCII characters.

HTTP/1.0
: Added MIME-like messages, header lines containing information about the data being transferred and modifiers for the request/response messages.
The first HTTP protocol to gain widespread use.

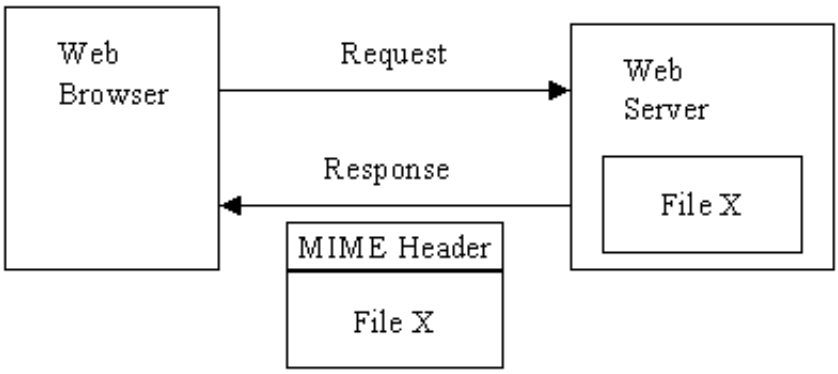
HTTP-NG (Next Generation)
: Persistent connections, ability to make multiplex multiple requests/responses over a single transport connection, and the ability to pass URLs to a protocol in a response message.

HTTP/1.1
: Addresses new needs and overcome shortcomings of HTTP/1.0, as allowing multiple transaction over single connection, or negotiating a content type.

S-HTTP (Secure HTTP)
: Provides secure communication mechanisms between an HTTP client-server pair in order to enable spontaneous commercial transactions for a wide range of applications.

HTTP/1.0

The web client (browser) communicates with a Web server using one or more connections:



Main features of HTTP/1.0

Message types:

There are two kinds of messages: *requests*, and *responses*. Requests are made to get resource from the server and responses are the replies to requests. Both requests and responses consist of a header and an optional message body. Header and body are separated from each other by a blank line.

Header:

Header of the message consists of a start-line and one or more header fields. The content of the start-line in the header depends on whether the message is a request or response.

The start-line of the request header contains a method to be applied to the object requested, the URI of the object requested and a protocol version in use. The start-line of response header is called the status line. It begins with the HTTP version, followed by a 3-digit response code, followed by an English response phrase.

Body:

Body of the message consists of a raw data of the resource, like HTML code, binary data of the image, etc..

Header fields:

The header fields in both request and response header contain metainformation about the body. The metainformation is encoded using the MIME format.

A MIME header field consists of a field name, followed by a colon, a single space, and the field value. Field names are case sensitive. Header fields in HTTP messages can be classified into three categories: those that apply to requests, those that apply to responses, and those that describe the body. A full list of defined header fields can be found here. Unknown header fields should be ignored by a recipient.

Examples of HTTP header fields

in request

```
From: Juuso.Teekari@lut.fi
Accept: text/plain, text/html
Accept: audio/*
Accept-Encoding: x-compress; x-zip
UserAgent: LII-Cello/1.0  libwww/2.5
```

in response

```
Last-Modified: date
Content-Language: en
Content-Encoding: x-zip
```

in both request and header

```
Content-Length: int
```

A list of all header fiels defined by HTTP/1.0 specification.

HTTP requests

There are three kinds of request methods in HTTP/1.0. The method is specified in the start-line of the request header. The methods are:

GET
: Returns whatever resource is identified by the URI

HEAD
: The same as GET but returns only HTTP header and no document body
: Useful to check characteristics of a resource without actually downloading it, thus saving bandwidth.

POST
: Used for posting electronic mail, news, or sending forms that can be filled in by and interactive user. This is the only request that sends a body with the request.

GET method

Example of a start-line of GET request:

```
GET /path/to/file/index.html HTTP/1.0
```

The start-line may be followed by header fields, though HTTP 1.0 does not require any. For Net-politeness, consider including at least following two headers in your requests:

```
From: your.name@your.shrine
User-Agent: Program-name/x.xx
```

POST method

A POST request is used to send data to the server to be processed in some way, for example by a CGI script.

A POST request is different from a GET request in the following ways:

- There's a block of data sent with the request in the message body and there are usually extra headers to describe this message body, like **`Content-Type:`** and **`Content-Length:`**
- The *request URI* is not a resource to retrieve; it's usually a program to handle the data you're sending.
- The HTTP response is normally program output, not a static file.

HTTP responses

Most of HTTP response is very similar to HTTP request. The main difference is a start-line (or a status line), which gives the status code of the response. In addition, the response header contains different header fields, their examples were given earlier.

There are three categories of status codes:

Success 2xx
: These codes indicate success. The body section if present is the resource returned by the request.

Error 4xx, 5xx
: The 4xx codes are intended for cases in which the client seems to have erred, and the 5xx codes for the cases in which the server is aware that the server has erred.

Redirection 3xx
: The codes in this section indicate action to be taken (normally automatically) by the client in order to fulfill the request.

Examples of response codes:

```
OK 200
No Response 204
Bad request 400
Forbidden 403
Not found 404
Moved 301
```

Full list of status codes together with their description can be from W3C pages.

Sample HTTP Exchange

Retrieve slide from course lecture notes - file at URL

```
http://www.it.lut.fi/opetus/00-01/010577001/lecture2/slide1.html
```

Proceed as follows:

1. Open a socket to the host www.it.lut.fi, port 80
2. Send something like the following through the socket:

```
GET /opetus/99-00/010577000/lecture2/slide1.html HTTP/1.0
Accept: text/html
User-Agent: telnet
From: your-email-address
 * blank line *
```

3. The server should respond with something like the following, sent back through the same socket:

```
HTTP/1.0 200 OK
Date: Fri, 17 Mar 2000 10:45:39 GMT
Server: Apache/1.3.6 (Unix)
Connection: close
Content-Type: text/html

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head xmlns:date="http://www.jclark.com/xt/java/java.util.Date">
... more contents
</body>
</html>
```

4. After sending the response, the server closes the socket.

To familiarize yourself with requests and responses, manually experiment with HTTP using telnet.

HTTP/1.1

HTTP 1.1 addresses new needs and overcome shortcomings of HTTP 1.0. It is a superset of HTTP 1.0. Improvements include:

- Allowing multiple transactions to take place over a single *persistent connection.*
- Faster response for dynamically-generated pages, by supporting *chunked encoding*, which allows a response to be sent before its total length is known.
- Access Authentication by a simple challenge-response authentication mechanism.
- Allowing a selection the best representation for a given response when there are multiple representations available.by supporting *content negotiation.*
- Adding cache support to minimize the need to send requests in some cases.
- Adding four more request types allowing also to store or delete a resource from the server.

Both clients and servers have to do a few extra things in order to comply with HTTP 1.1.

HTTP/1.1 Clients

To comply with HTTP/1.1, clients must

- Include the **`Host:`** header field with each request.
- Accept responses with *chunked* data.
- Either support *persistent connections*, or include the "**`Connection: close`**" header field with each request.
- Handle the "**`100 Continue`**" response.

HTTP/1.1 request methods

In addition to HTTP/1.0 methods GET, HEAD and POST, HTTP/1.1 has added four more request types:

OPTIONS
: A request for information about the communication options available on the request/response chain identified by the Request-URI.

PUT
: Requests that the entity attached to the request be stored under the supplied URI. This means that the client can write to the server.

DELETE
: Remove the resource identified by the Request-URI. This allows the client to delete a file on a server.

TRACE
: Used to involve a remote, application-layer loop-back of the request message.

HTTP-NG

An enhanced replacement for HTTP/1.0 designed to correct the known performance problems in previous versions of HTTP, and to provide extra support for commercial transactions, including enhanced security and support for on-line payments.

Allows multiple requests to be sent over a single connection. These requests are asynchronous - there is no need for the client to wait for a response before sending out a different request. The server can also respond to requests in any order it sees fit.

Although HTTP-NG originated before HTTP/1.1, the latter seems to have overtaken it in terms of popularity. There have been some implementations of this protocol which yield better performance than HTTP/1.1.

For more info on HTTP-NG see

HTTP-NG Architectural Overview
: Spero, Simon. 1996
http://www.w3.org/pub/WWW/Protocols/HTTP-NG/http-ng-arch.html

Secure HTTP

Secure-HTTP or S-HTTP describes a syntax for securing messages sent using the HTTP protocol. Basically S-HTTP attempts to make the existing HTTP more secure my providing many features. It allows a variety of key management mechanisms, security policies and cryptographic algorithms by supporting option negotiation between parties for each transaction.

Main features

- Designed for use in conjunction with HTTP.
- S-HTTP aware agents can communicate with non S-HTTP agents, and vice versa, although such transactions do not utilize S-HTTP's security features.
- S-HTTP mimics the format and style of HTTP to ease integration. Certain headers are promoted to be Secure HTTP headers

  ```
  Secure * Secure-HTTP/1.2
  ```

  The response line will look like

  ```
  Secure-HTTP/1.2 200 OK
  ```

- Several cryptographic message format standards may be incorporated into S-HTTP clients and servers.

- Message protection can be done in three ways: signature, authentication, and encryption. Any message may be signed, authenticated, encrypted, or any combination of these.

Server-side programming

In 99.9% of web applications on the other end of the wire from the web browser is a web server. The web server is the entry point to the *Middleware Layer*.

The purpose of the Middleware Layer is to accept incoming requests and process them, using the resources provided by the web server, the machine that the web server runs on, or by the network of servers and resources that the web server is connected to.

Dynamic inclusion of information

Every URI that a server provides corresponds to a file on that server. Normally, that file is an image, HTML file, or other **static** document. When a server is asked to supply such a static document, the web server will find the given file on the local (or networked) file system and send it back to the browser. It is the most basic function of a web server.

How can a web server get to all other resources which are at his disposal?

There is multiple ways web server can exploit resources above and beyond file systems. To name some of them:

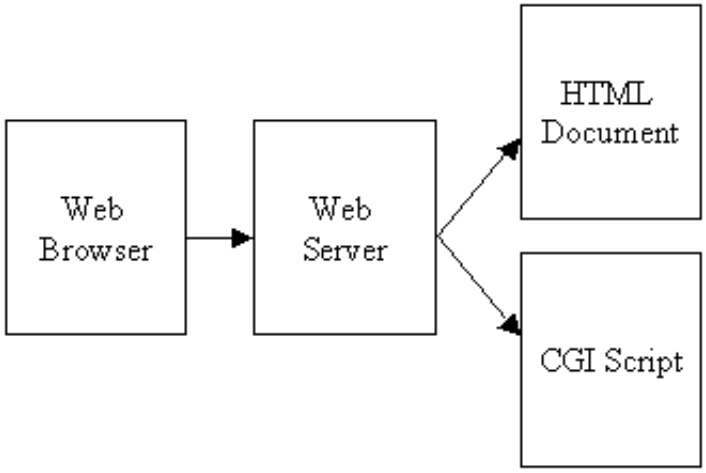
- CGI scripts
- SSI
- ASP
- PHP
- Servlets
- ...

CGI - Common Gateway Interface

CGI is a standard for interfacing external applications with information servers, such as HTTP or Web servers. The CGI is the most basic tool to access system resources on the server.

When a client asks for a URI that points to a CGI application, the following things happen:

1. **The WWW client sends a request to the HTTP server.**
   This request gives the name and location of the CGI application, and can provide parameters to the application.
2. **The HTTP server starts the application.**
   The server provides some extra information to the CGI application, including any parameters passed by the client.
3. **When the CGI application terminates, all output produced by the application is returned to the client.**
   Some additional information is added to the HTTP header produced by the CGI before the output is returned.



Creating CGI Applications

A CGI application (commonly called a CGI script) can be written in any language that can read STDIN, write to STDOUT, and read environment variables, i.e. virtually any programming language, including C/C++, Perl, TCL, or even shell scripting.

The output of CGI application goes more or less directly to a client. The CGI application therefore have to output both a header and a body just as in case of a static document.

Headers:

CGI application does not need to produce much header information. The only required header field is a Content type, written as

```
Content-type: type/subtype
```

Which other headers the CGI application outputs depends on what kind of document you are sending back, whether it be a **full document** or a **reference** to one.

Body:

The body contains all of the data which is to be displayed by the WWW client. The data can be documents of various types, as images, HTML documents, a plaintext documents, or perhaps even an audio clip. The interpretation of the data by a client depends on what type was given in header as **`Content-type`**

Return document types

The CGI application can return either a **full document** or a **reference** to one.

A full document with a corresponding MIME type:

Consist of a header and a body just as case of a static document.

```
Content-type: text/html

<html>
 <head>
  <title>Output of HTML from CGI script</title>
 </head>
 <body>
  <h1>Sample output</h1>
  <p>People have one thing in common: they are all different.</p>
 </body>
</html>
```

A reference to another document:

Instead of outputting the document, you can just tell the browser where to get the new one, or have the server automatically output the new one for you.

```
Content-type: text/html
Location: http://foo.foobar.fi/

<html>
 <head>
  <title>Sorry, document moved</title>
 </head>
 <body>
  <p>Now available at
  <a href="http//:foo.foobar.fi/">a new location</a>
  </p>
 </body>
</html>
```

Structure of a CGI Script

A typical sequence of steps:

1. Read the user's form input.
2. Do what you want with the data.
3. Write the response to STDOUT as HTML page.

Handling input to the CGI application

When a client asks a server to launch a CGI application, it can give the server some input to provide to the CGI application. There are three important ways to give input from a client to a CGI application:

1. Through a query string
2. Through the command line
3. Through standard input

A query string (GET method)

A WWW client can send a query to a CGI application by appending a '?' followed by a query string to the URL for the CGI application. This query string is composed of name-value pairs in the form of:

```
name1=value1&name2=value2&name3=value3
```

The names and values of variables are *URL-encoded*. There are some characters that have special purposes in a URL, such as the colon (':') and forward slash ('/'). Some characters are not allowed to appear in URLs at all, such as spaces. All spaces are replaced by a plus sign ('+'), and all special characters are replaced by %xx,

where xx is the hexadecimal representation of the ASCII value for a character.

Command line parameters

They are rarely used for CGI applications, because this works only if the query string contains no equals sign. If an equals sign is present in the query string after the question mark ('?'), then the entire query string is provided through the **`QUERY_STRING`** environment variable.

For example, if the following URI is accessed:

```
http://www.nowhere.com/cgi-bin/foo.cgi?chicken
```

then the string **`chicken`** would be the first parameter passed to foo.cgi when the server started its execution.

Standard input (POST and PUT method)

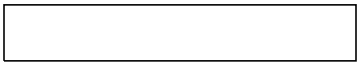
Information is only given to a CGI application through standard input when the CGI is accessed with either the POST or PUT methods. The body of the POST or PUT request is used as the standard input to the CGI application. The format of the information will be the same as if it were provided in the **`QUERY_STRING`** environment variable via the GET method, and it is URL encoded.

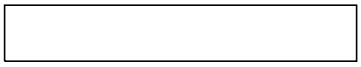
Creating a query string

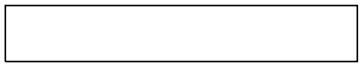
A query string is most often produced as a result of submitting a content of an HTML form.The names in name-value pairs are what you defined in the INPUT tags (or SELECT or TEXTAREA tags), and the values are whatever the user typed in or selected.

Example:

The fields from this form:

First entry field: 

Second entry field: 

Third entry field:  --- Select Option:

- ◇ Frogs
- ◇ Peaches
- ◇ Cream
- ◇ Newts

Will be coded as:

```
entry1=black+light&entry2=blurb&entry3=left+handed+screwdriver
```

HTML code of the form

The HTML code, which generated previous form is as follows:

```
<form action='http://www.lut.fi/cgi-bin/some-prgm' method='GET'>
 First entry field: <input name='entry1' value='black light'/><br/>
 Second entry field: <input name='entry2' value='blurb'/><br/>
 Third entry field: <input name='entry3' value='left handed screwdriver'/>  ---
 Select Option: <select name='entry4'>
    <option value='no1'/>Frogs
    <option value='no2'/>Peaches
    <option value='no3'/>Cream
    <option value='no4'/>Newts
 </select>
</form>
```

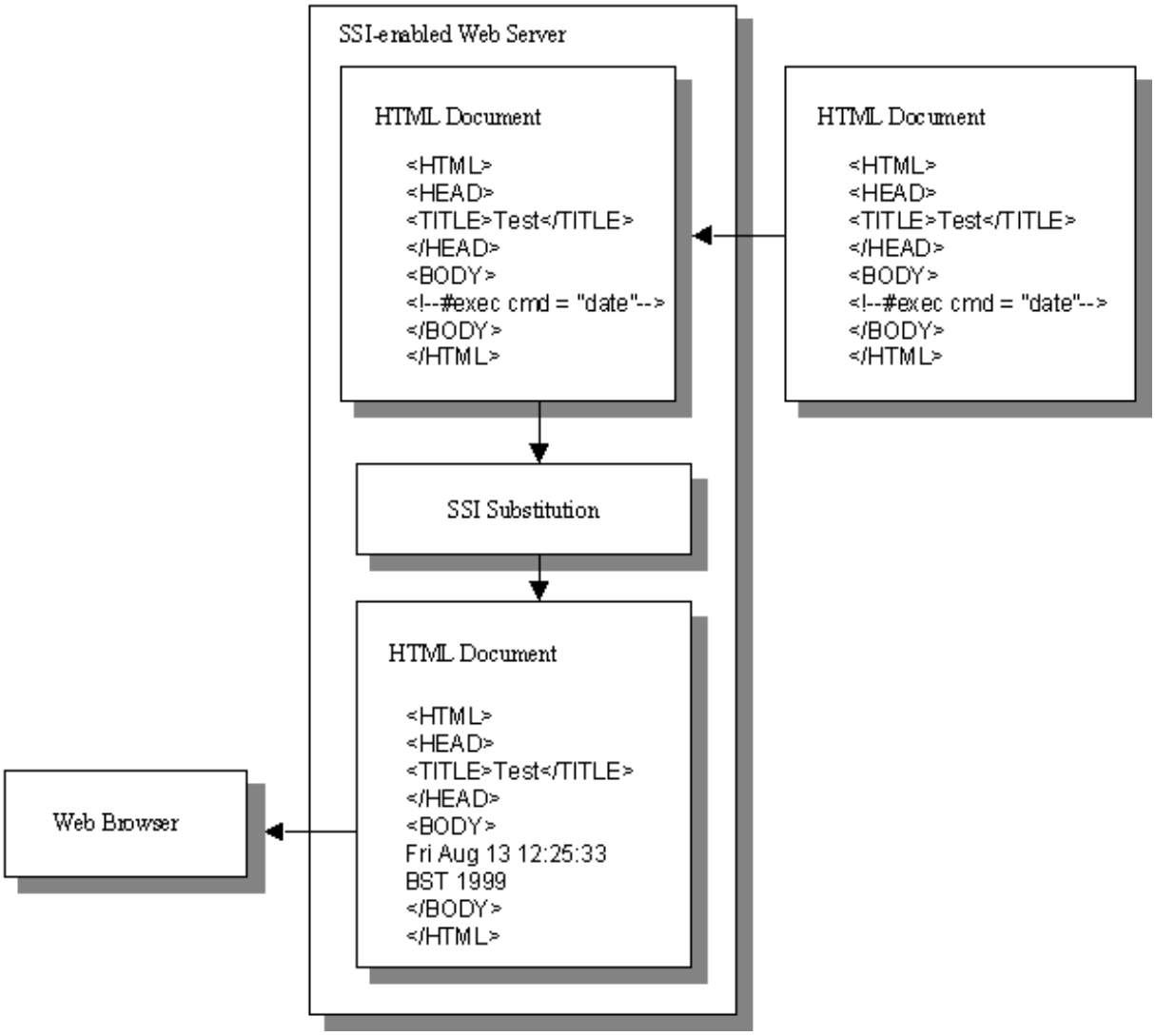
SSI - server-side includes

There are some problems with CGI. Perhaps the most serious problem is speed. Every time the web server gets a CGI request, it needs to execute the CGI application.

One way to get around this problem is to embed the processing into the web server itself. Rather than rely on another layer, most web servers provide several ways to extend the web server itself; to add logic and processing power.

The earliest technology to take advantage of this idea was SSI. The concept of SSI is simple. An application developer codes special tags into her HTML document. Those special tags are understood by the web server and can be translated on the fly by the web server as the HTML document passes through on its way to the browser.

The server-side include feature allows you to dynamically include information in your HTML documents -- each time the document is accessed, the document accesses gateway (or other) programs and includes the program output inline with the document.



All include directives are formatted as SGML comments within your document. Each directive has the following format:

```
<!--#command tag1="value1" tag2="value2" -->
```

Each command takes different arguments, most only accept one tag at a time.

Here are some examples of commands:

```
<!--#include virtual="/dir/file.ext"-->
<!--#include file="subdir/file.ext"-->
<!--#exec cmp="ls" -->
<!--#exec cgi="environment.cgi" -->
<!--#echo var="DOCUMENT_NAME"-->
<!--#fsize file="sample.txt"-->
<!--#config sizefmt="bytes"-->
```

For full list of commands and their description see Server Side Includes by Linda Cole.

Examples

```
<!--#config timefmt="%A, %B %d, %Y"-->
<!--#echo var="LAST_MODIFIED"-->
```

Appears as:
Thursday, March 22, 2001

```
<!--#config timefmt="%A, the  %d of  %B, in the year  %Y" -->
<!--#flastmod file="notes.html" -->
```

Appears as:

ASP (Active Server Pages)

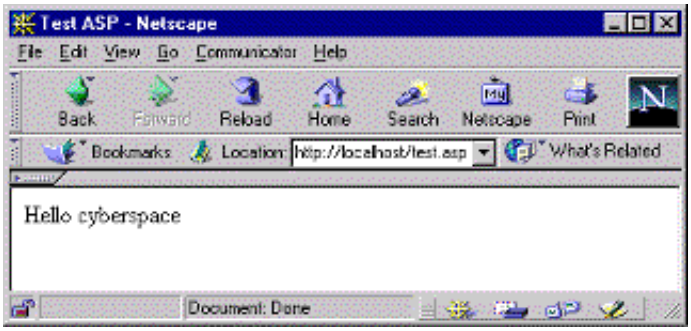
SSI based technology is limited to the range of commands/tags offered by the SSI-enabled web server provider. Web developers needed a way to embed dynamically interpreted code into HTML that can be processed by the web server on demand. What was needed was a hybrid of CGI and SSI. *ASP* was born.

ASP is a server extension of the IIS web server released by Microsoft. ASP allows developers to code custom tags in JavaScript (JScript) or VBScript. These tags can be interpreted by IIS before the pages are sent out.

An ASP page at its core is simply a text file that has been named using the extension .asp and which contains HTML and scripting. Scripting, usually in VBScript provides a means to embed programmatic logic into HTML files that will be dynamically interpreted as the HTML page goes through the web server and also provides access to any server side object.

Example 1: Simple ASP page

```
<HTML>
  <HEAD>
    <TITLE>Test ASP</TITLE>
  </HEAD>
  <BODY>
    <% @Language = "VBScript" %>
    <% Response.Write("Hello cyberspace") %>
  </BODY>
</HTML>
```



PHP

Despite popularity of ASP, developers wanted more stable and less proprietary solution for inclusion of dynamic information.

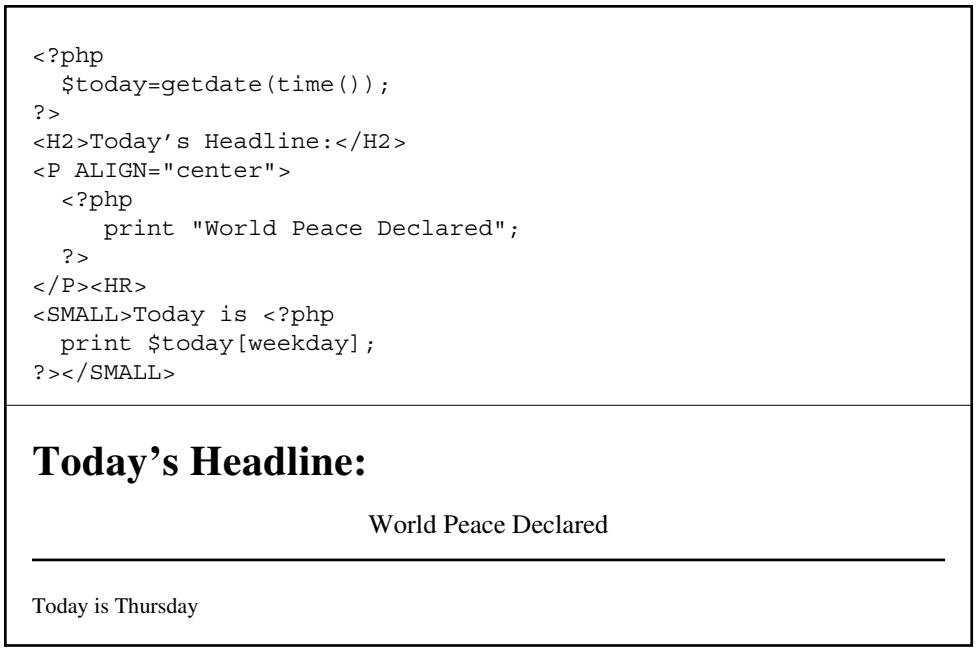
Much of PHP syntax is borrowed from C, Java and Perl with a couple of unique PHP-specific features thrown in. The goal of the language is alike the goal of ASP. To allow web developers to write dynamically generated pages quickly.

Like with ASP, PHP pages require a web server with a special support build in. At this time of writing, there are only two real methods of using PHP with a web server: either executing the PHP interpreter from a CGI wrapper, so that PHP runs as any CGI script would, or integrating PHP with the Apache web server.

PHP code is contained within a special tag, separating it from the other HTML on the page:

Example 1: Simple PHP page

```
<?php
  $today=getdate(time());
?>
<H2>Today's Headline:</H2>
<P ALIGN="center">
  <?php
     print "World Peace Declared";
  ?>
</P><HR>
<SMALL>Today is <?php
  print $today[weekday];
?></SMALL>
```



Last update: Tue Mar 27 12:58:03 EEST 2001