INFO I211 Information Infrastructure II

Spring 2014

Lecture 15: Introduction to CGI scripts

To understand how CGI scripts interact with web browsers and web servers we begin by reviewing a simpler interaction: how static HTML files are *requested* by and *displayed* by users. Let's say that I have the following simple, basic HTML file in my `DocumentRoot` *(*)* directory, and its name is `lecture15.html`.

```
<html>
<head>
<title> Hello I211 world! </title>
</head>
<body>
<h1> Hello I211 world! </h1>
<p> How is Lecture 15? </p>
</body>
</html>
```

(*) `DocumentRoot` is a name referring to the directory which contains all the files and subdirectories that are reachable by the web/CGI server, in my case it's `/u/mitja/cgi-pub/`)

The path to the file is

```
/u/mitja/cgi-pub/lecture15.html
```

and it has to be made readable by the world for it to be accesible over the web:

```
silo.soic.indiana.edu$ pwd
/nfs/nfs4/home/mitja/cgi-pub
silo.soic.indiana.edu$ emacs -nw lecture15.html
silo.soic.indiana.edu$ cat lecture15.html
<head>
<title> Hello I211 world! </title>
</head>
<body>
<h1> Hello I211 world! </h1>
<p> How is Lecture 15? </p>
</body>
</html>
silo.soic.indiana.edu$ ls -l lecture15.html
-rw------- 1 faculty     127 10 mar 22:09 lecture15.html
silo.soic.indiana.edu$ chmod ugo+r lecture15.html
silo.soic.indiana.edu$ ls -l lecture15.html
-rw-r--r--  1 faculty    127 10 mar 22:09 lecture15.html
silo.soic.indiana.edu$
```

Once we've created the HTML text, it may seem that the process of delivering it to a web browser should be a trivial task. But serving even a simple page like this one requires that a lot of coordination occur between the browser and the web server on which the page is stored.

By web server we mean a program running on a host machine that uses the Hypertext Transport Protocol (HTTP) to communicate with the browser. The program that exists on burrow cluster systems as

```
/usr/sbin/httpd
```

is such a web server, and all web sites are based on similar servers. The web is based on a client-server model. This means that there is a server (that provides resources) and a client (which requests them).

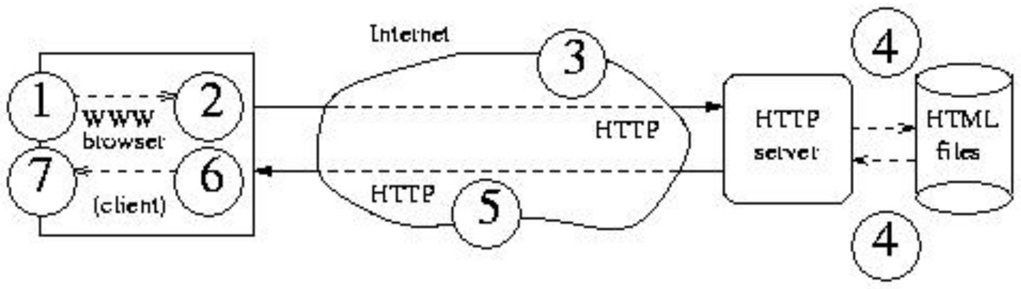
Here's what we need to keep in mind about these:

- both the server and the client are programs
- they may reside on the same machine or on different ones
- both the client and the server have agreed in advance on some mechanism that allows them to communicate with each other (such a mechanism is called a protocol).

For the World Wide Web the role of the client is played by a web browser (such as ...SeaMonkey) and the web server is the software that delivers resources such as: text files, images, movies, sound files, to one or more web browsers.

There are millions, billions of web servers throughout the world (wide web) but they are all acessible from any browser because they have all agreed to use a common protocol - the Hypertext Transfer Protocol (HTTP). HTTP is based on an exchange of requests and responses.

Each request can be thought of as a command, or action, which is sent by the browser to the server to be carried out. The server performs the requested service and returns its answer in the form of a response.



The components of a simple WWW interaction are the user, the client, and the server. The client acts as an intermediary between the user and the server.

Steps 1-7 detail the basic information flow in a simple HTTP transaction. Essentially the client requests a file and the server delivers it. The entire HTTP process takes place as a result of simple transactions of requests and responses.

1. The user sees a URL

   ```
   http://cgi.soic.indiana.edu:80/~mitja/lecture15.html
   ```

   and clicks the hyperlink or types the URL into the browser.

2. The browser interprets this command: It is different from printing, creating the bookmark, saving a file, changing any preferences, etc. This command (there is often a menu entry named `"Open Location"` in web browsers) says that the server at the address corresponding to the DNS name

   ```
   cgi.soic.indiana.edu
   ```

   needs to be contacted on port `80` and that the file `/lecture15.html` file is being requested.

   For this, the browser sends the HTTP `GET` command to the server (not shown here - we'll look at how this works when we simulate this request process using `telnet`). The path to the requested file is relative to the server's document root).

3. The browser sends the `GET` request to the server, indicating what file it needs. This request travels over the Internet, going through routers and other internet devices, until it reaches the web server's host: `cgi` in the SoIC domain.

(*there's a network security aspect here that we will need to address later*).

4. The server receives and parses the request. It uses the file extension (`.html`) to determine the type of information in the file. The `.html` means that it will send back to the browser the file but it will first say: the file's `Content-type: text/html`.

   You do not have to include "text/html" in the file, it is inferred by the server from the file's extension. But the server does send this information to the browser as part of the header, followed by the data (the actual file) as explained below.

5. An HTTP response goes from server to the client. The headers that are part of the message indicate that the request was OK and that the data returned is of

   ```
   Content-type: text/html
   ```

   The headers are then followed by (a blank line and then by) the HTML data itself.

6. The `Content-type:` part of the header tells the browser that the data is text formatted in HTML, so the browser *renders* the text appropriately, highlighting hyperlinks, etc.

7. The user views the HTML page as rendered by the browser, and has the opportunity to select another hyperlink, starting the cycle over again

A text-based diversion: impersonating a web browser with `telnet`

Well before the Hyper Text Transfer Protocoll (HTTP) and the Hyper Text Markup Language (HTML) were invented, `telnet` was around to connect from text-based client computers connected via the internet to server computers. The `telnet` client program is still around, but most servers don't allow direct text-based (and unencrypted!) `telnet` connections nowadays. However, we can use `telnet` to simulate a connection —as if we were a web client— to a remote server, by explicitly requesting a connection on *port 80* (the default port number for web servers) and requesting one single page using an HTTP command named `get`:

```
silo.soic.indiana.edu$ telnet cgi.soic.indiana.edu 80
Trying 129.79.247.127...
Connected to cgi.soic.indiana.edu.
Escape character is '^]'.
GET  /~mitja/lecture15.html
<html>
<head>
<title> Hello I211 world! </title>
</head>
<body>
<h1> Hello I211 world! </h1>
<p> How is Lecture 15? </p>
</body>
</html>
Connection closed by foreign host.
```

Of course you want to achieve more than the limited functionality from a web server — serving static web pages is only the beginning. You may for example want to query a database and return the result to the user. The HTTP server can't do this *directly*, and instead it delegates the reques to an external program to which it has access, which acts as a *gateway* (or intermediary) between the server and the data repository.

When the server receives a request to access the database it passes the request to a *gateway* program which does whatever is necessary to get the data and return the results to the web server.

The server then repackages the information from the script, and forwards the information back to the client. (In a sense the server acts as a sort of translator, taking data from either a file or script and providing it to the browsers in a consistent and uniform manner).

We make two observations now:

- Passing data in the *correct format* to the script is crucial.
- Typically clients (web browsers) do not know (or care) whether the web server handled the request internally or offloaded the work to other programs; they just interpret the returned result and display it to the user.

Clearly, in order to make all this data processing and transmission work properly, the gateway programs and the server must communicate with each other. The details of this interaction are specified by the Common Gateway Interface (CGI).

The CGI protocol

- defines the input that a script can expect to receive from the server,
- as well as the output it must return to the server in order to be understood.

What the script does in between the input and the output, is entirely up to the script.

So the process of servicing the

```
http://cgi.soic.indiana.edu/~mitja/lecture15.cgi
```

request is different, because by the file extension of the file specified in the URL the web server realizes that it needs to execute the script specified by that address (or path) rather than simply retrieving the file.

Upon starting the script, the server provides such scripts with a variety of potentially useful information (such as the name of the machine from which the request originated, the type of browser used, etc.) and then starts the script. What follows is of no concern to the server, other than the output of the script, which the server will send back to the requesting browser.

You take a lot of responsibility this way if you're writing the script. While the server doesn't care how the script generates its output, it does need to know the format of the output - the script's output is, after all, the server's input (on the path back to the user).

Recall that when the web server delivers a static file to the browser (such as the simple HTML page we wrote above), it uses a filename extension (e.g. `.html`) to determine what to return in the `Content-type` header. This technique doesn't work to determine the output of the file received from CGI scripts, because a script's filename is unrelated to the type of information it returns. A script named `getpic`, for example, may return an image as its data (`Content-type: image/gif`) while the similarly named `getinfo` might return HTML text (`Content-type: text/html`). It is even possible for a single script to output different sorts of data depending upon the context in which it is called. Therefore it is absolutely essential that the script notify the server of the type of data it is generating, so that the server can pass this information on to the client.

The `lecture15` script is going to be:

```
#!/usr/local/bin/python3
print ("Content-type: text/html\n\n<html> \
<head> \
<title> Hello I211 world! </title> \
</head> \
<body> \
<h1> Hello I211 world! </h1> \
<p> How is Lecture 15? </p> \
</body> \
</html>  \
")
```

The above Python program has one statement only, which prints a few lines.

The first line starts by specifying the media type that identifies the data in the body. The `\n\n` that follow the header information are translated by Python into *newlines*. The first ends the line with the content type, while the second inserts a (mandatory) blank line that separates the header from the rest of the message.

The remainder of the script simply outputs HTML text that looks suspiciously similar to the contents of the static `lecture15.html` shown earlier, beginning with the familiar `<html>` tag and ending with `</html>`. All of this text output by the `print` statement is sent to the server which executed the script.

The server captures the output, constructs a set of HTTP message headers (including the `Content type` returned from the script), and sends these headers and the rest of the script's output to the browser. Upon receiving and interpreting the data, the browser is left with the HTML shown in `blue` in the code fragment above.

This is rendered by the receiving browser in the exact same way as the `lecture15.html` file that we started with. So if we were to look *only* at the output we couldn't make any difference between the two approaches.

CGI scripts are quite flexible precisely because the server itself is not really involved in the process. The server's primary responsibilities are to

- start the CGI script
- send the proper information to the CGI script, and
- pass the CGI script's output back to the browser

It is up to the CGI script to do most of the work. This division of labor leads to a protocol that is simple yet powerful.

To summarize, the information flow is as follows:

- The user selects a URL that references a script:

  ```
  http://cgi.soic.indiana.edu:80/~mitja/lecture15.cgi
  ```

- The client interprets the URL in the usual way: the

  ```
  GET
  ```

  message requesting

  ```
  /~mitja/lecture15.cgi
  ```

  is sent to

  ```
  cgi.soic.indiana.edu
  ```

  on port

  ```
  80
  ```

- The web server receives request. Using the file extension (the path ends with `lecture15.cgi`) it determines it should *run* the script instead of simply retriving the file.

- The web server starts the script and sends it information in the form of environment variables and standard input
- The gateway program (the CGI script we just wrote in Python) undertakes any necessary processing and then produces output. The program suplies a `Content-type` header to indicate the format of the data to the server, for example:

  ```
  Content-type: text/html
  ```

  The headers are then followed by the HTML text generated by the script. From here on it's the same story as before:
- The headers (including `Content-type`) and data go directly from the CGI script (a.k.a the gateway program we wrote in Python) to the web server.
- Having successfully received data from the gateway program, the web server simply relays the information to the web client.
- An HTTP response goes from server to browser. Included in the response are a status indicating the request was OK, and the `Content-type` header from the script. Following the headers is the actual HTML script.
- The `Content-type` header tells the browser that the data is HTML, so the browser formats and renders the text appropriately, including highlighting links.
- The user views the HTML output and has the opportunity to select another hyperlink, starting the cycle again.

Last updated: 2014-03-12
Mitja Hmeljak