Introduction to HTTP and CGI

Ian Graham

Instructional and Research Computing

University of Toronto

igraham@utirc.utoronto.ca

(416) 978–4548

Purpose of This Course

To explain underlying technolgies of the Web, with the aim of understanding how to write CGI programs.

Requires: *Four key ingredients*

1. URLs *(**U**niform **R**esource **L**ocators)*

- The ***naming*** scheme for Internet resources;
- Includes method for transmitting data from the client to the server.

2. HTML *(**H**yper**T**ext **M**arkup **L**anguage)*

- Has elements (**ISINDEX**, **ISMAP**, **FORM**s) for collecting data from the user and sending it to the server.

3. HTTP *(**H**yper**T**ext **T**ransfer **P**rotocol)*

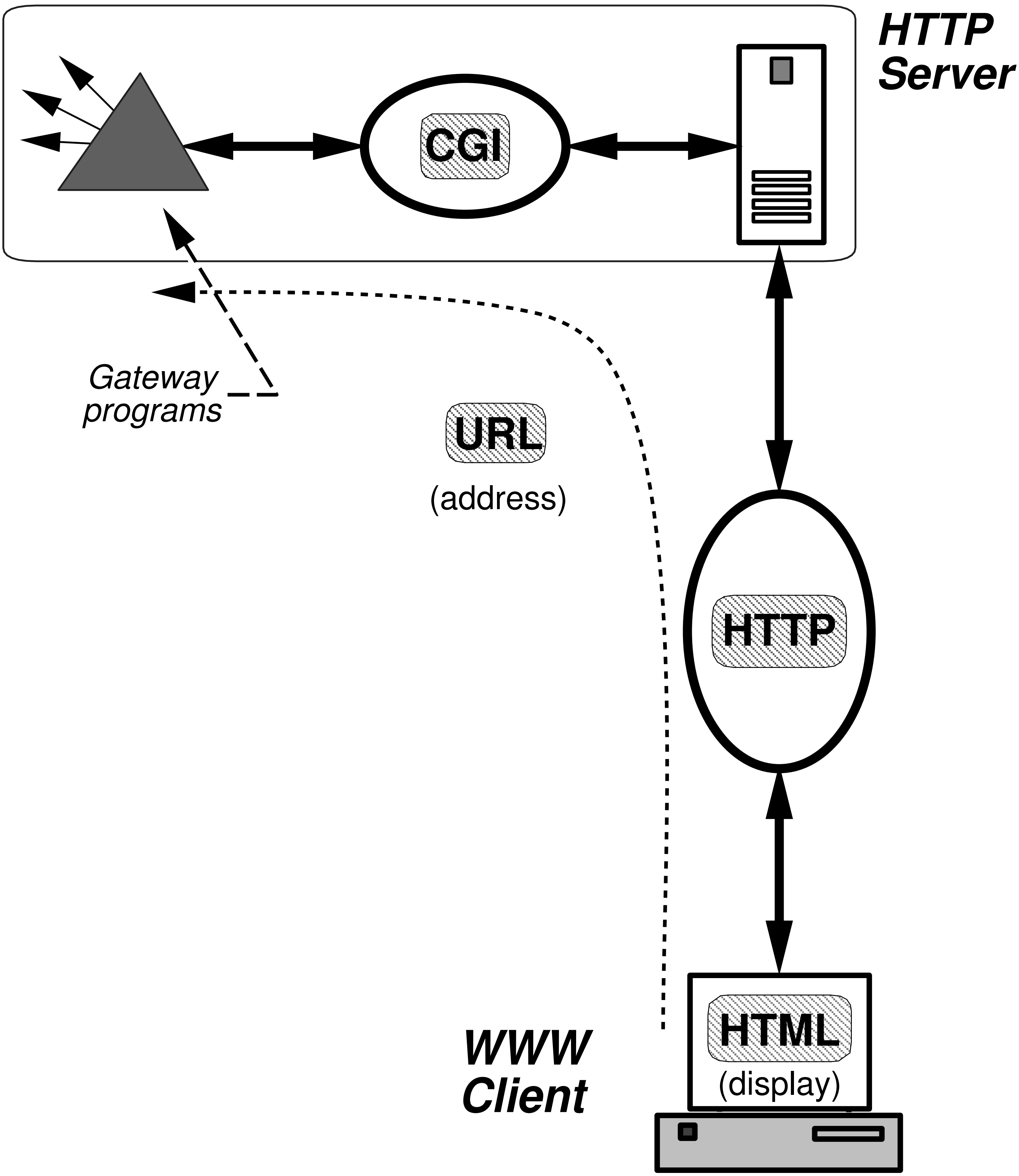
- A client–server protocol specifically designed for serving up hypertext documents.

- Facility for sophisticated client–server interactions through....

4. CGI *(**C**ommon **G**ateway **I**nterface)*

- A specification for communication between the server and ***gateway*** programs resident on the server.

- This allows for Web interfaces to databases, and for the dynamic generation of HTML documents by ***gateway*** programs.


HTTP Server
CGI
Gateway programs
URL
(address)
HTTP
HTML
(display)
WWW Client

Course Outline

1. Introduction To URLs

- basic form of a URL
- query strings
- referencing programs, not files (server configuration issues)

2. Introduction to HTTP

3. HTML User Input Elements

- `<ISINDEX >`
- `<IMG SRC="..."  ISMAP>`
- **FORM** elements

4. CGI Specification and programming

- ***CGI***

1. Uniform Resource Locators

URLs are a way of encoding, in a single string of plain text, all the information required to access a particular Internet resource.

The the URL contains information about:

- The ***protocol*** to use,
- The ***domain nam***e and ***port number*** of the server,
- The ***path*** to the resource,
- The ***name*** of the resource,
- ***Query information*** to be passed to the resource (only applies to certain protocols)

URLs are designed for universal use in referencing Internet resources . Thus there are limitations on the characters (a ***subset*** of ASCII) that can be included in a URL.

Disallowed characters, including ISO Latin−1, can be included using an encoding scheme:

`%xx` (`xx` is the hex code for the character)

General Form of A full URL

`http://www.address.edu:8080/path/subdir/file.ext`

Directory/Resource Details

Usually the location of the resource under the Document directory. The scheme is different for `gopher:` URLs. This field is not needed for `telnet:` or `news:` URLs.

Port Number

If not the default port for the indicated protocol.

Domain name

Usually the Internet domain name (or IP number) of the server. For `news:` URLs this is the newsgroup name.

Protocol

This must be in lower case. Valid protocols are `http:`, `ftp:`, `gopher:`, `mailto:`, `wais:`, `telnet:`, `tn3270:`, `news:`, and so on. The meaning of the other fields depend on the protocol being used.

Query Data

Query data is sent within a URL by appending the query to the resource locator, separated by a question mark. For example:

`http://www.fo.ca/path/thing?`***`query_data_string`***

In general (but not always) the web client constructs this URL based on:

– the resource URL,
– the query data input by the user

The query data is *specially encoded* when appended to the URL –– for example, spaces are not allowed and must be encoded. We talk about this encoding later in the CGI section.

Extra Path Information

The path information (`/path/thing`) is treated specially when the reference is to a program. But first we will discuss how a server knows a URL points to a program, and not a data file.

URLs, Gateway Programs and Query Data

```
http://www.fo.ca/cgi-bin/prgm.pl?query_data
```

URL References to Programs

In general, you must configure the server to know that a particular directory contains programs, and then configure the server relate a script–aliased URL path to point to this directory. For example:

```
ScriptAlias /cgi-bin/ /path/dir/prgm-dir/
```

(This is the configuration line for the NCSA HTTP server), means that the URL:

http://server.fg.ca/cgi–bin/prgm.pl

Actually references the program:

/path/dir/prgm–dir/prgm.pl

The `query_data` is pased to this program using the CGI mechanisms (discussed later).

Example:

`http://www.fo.ca/cgi-bin/`**`prgm.pl`**`/extra/thing`

Extra Path Information

In the above example, it appears as if you are referencing a resource named `thing` in a subdirectory of `cgi-bin`. In a sense this is true – but the implementation is actually quite different.

In practice (this is part of CGI) this URL references the program ***prgm.pl.*** The path information `/extra/path` is passed to the gateway program through an environment variable. Typically you use the extra path information to give a gateway program the location of a configuration file.

We talk more about this extra path information when we discuss the CGI specification.

The HTTP Protocol

A Client–Server Protocol

Recall that all Internet tools use a client–server protocol. All actions are accomplished through messages that are passed between the client and server, using the defined protocol. Examples are FTP, gopher, WAIS, SMTP, and HTTP.

HTTP Protocol

The HTTP protocol was specifically designed for the World Wide Web for use in rapidly delivering data of any type, with a minimum of operational overhead. In this respect it is similar to but faster than FTP. Importantly, the HTTP protocol contains protocol messages that can be passed from client to server and vice versa to communicate important information about the transaction, the desired action, etc. In addition, the protocol can be used for access control (passwords).

Four Stages of an HTTP Transaction:

1. Open the connection
2. Client sends the request
3. Server responds to client
4. The connection is closed

Each transaction results in the exchange of ***at most*** one piece of data (file), or the processing of a single collection of passed data.

Eight−bit data Channel

All information is passed as octets. Conseqently you can transmit any form of data (charcter, binary, etc.) during an HTTP transaction.

Protocol Specification

All HTTP messages and data are sent as eight−bit character data down a single communications channel. When a client contacts a server it sends to the server an ***HTTP request header***, followed by any data (if any) the client wants to send. In response the server sends and ***HTTP response header***, followed by the data (if any) it is sending to the client. An example follows.

Client−to−Server Request

Here is a typical request from a client to a server, in this case a request for a file from the server.

1. Client contacts server

For example, following a hypertext link in a document that looks like:

<A HREF="***http://domain.edu/Tests/file.html***">text </A>.

The following Information is sent to the server by the client (here by the Mosaic browser):

```
GET /Tests/file.html HTTP/1.0
Accept: text/plain
Accept: application/html
Accept: text/x-html
Accept: text/html
Accept: audio/*
.
.
.
Accept: */*
User-Agent:  NCSA Mosaic for the X Window
System/2.4  libwww/2.12 modified
[a blank line, containing only CRLF ]
```

Request Header

The request header contains several fields, that:

- define what the request is for, and how it should be processed,
- describe the capabilities of the browser,

The end of the request header is indicated by a single line containing only a CRLF (carriage–return line–feed) pair (like SMTP).

The first line: The Method Field

The first line contains the ***method field.*** This indicates the desired HTTP method , the resource being requested, the protocol version, etc. I.e:

```
GET /Tests/file.html HTTP/1.0
```

Other common methods are ***POST*** (for sending data from the client to the server (not as part of the URL) and ***HEAD*** (retrieve only header information about the resource).

The Other Lines: Client Information

The other fields contain information about the client.

The Accept: fields tell the server the different types of data that the client can accept. These are sent as ***MIME Content–type*** messages. Thus

```
Accept: text/plain
```

means that the client can accept plain text files, and so on.

This MIME type is important, as this is how HTTP servers tell clients the type of data being sent.

The other headers are informational:

`User-Agent:`	program making the request,
`From:`	who is making the request,
`Referer:`	gives the URL of the document making this request.

and so on.

2. Server Responds

The server responds by returning the desired data. It first sends server response headers, which communicate information about

- the state of the transaction,
- the type of data being sent,
- any additional information,

This is followed by the actual data being sent to the client.

The next page shows the data returned after a simple request for a file.

Server Response

Status Line

MIME Content-type

```
HTTP/1.0 200 OK
Date: Friday, 23-Sep-94 16:04:09 GMT
Server: NCSA/1.3
MIME-version: 1.0
Content-type: text/html
Last-modified: Friday, 10-Feb-95 16:03:27 GMT
Content-length: 145
[blank line, containing only CRLF]
<html>
<head>
<title> Test HTML file </title>
</head>

<body>

<h1>  This is a test file</h1>
<p>  So what did you expect, art?

</body>
</html>
```

The data

But:

How does the server know the type of the document it is sending?

Answer:

Each server has a ***database*** that maps ***filename suffixes*** (extensions) to ***MIME types***. For example:

```
.html            text/html
.txt             text/plain
.ps              application/postscript
.gif             image/gif
.jpg .jpeg       image/jpeg
```

Summary:

1. client sends a ***request header*** message to the server, requesting the resource and passing along information about the capabilities of the client.

2. The server returns a ***response header*** describing the state of the transaction (success, error) and containing descriptive information about the data being returned. This header is followed by the data.

   In particular the server sends a MIME `Content-type` header that gives the ***MIME–type*** of the data being sent.

   The client determines the type of the data ***exclusively*** from the MIME–type.

Relevance To CGI Programs

Servers determine the MIME type of data files through a convention relating filename suffixes (e.g. .gif, .html, ...) to MIME types –– a server has a database relating extensions to types.

This does not work for gateway programs, as the filename suffix is not related to the returned data type.

Therefore a gateway program must itself return header messages to the server, stating the data type being returned by the gateway program.

This is discussed later on in the CGI section.

HTML User–Input Elements

HTML has several elements that accept user input and send this input to an HTTP server.

These are:

1. ISINDEX

The tag `<ISINDEX>` should appear in the **HEAD** of an HTML document. When present, **ISINDEX** tells the browser that it should query the user for a text string. When the user submits this query the browser ***encodes*** the string, appends it to the URL of the document being displayed, and re–accesses the URL.

This means that the document containing an I**SINDEX** must come from a gateway program that understands what to do with query data.

2. `<IMG SRC=".." ISMAP>`

The **ISMAP** attribute turns the **IMG** element into an active image. This works only when the image is surrounded by an anchor specifying where the imagemapped info is to be sent for processing. That is:

```
<A HREF="http://www.se.ca/cgi-bin/imagemap/ex/path">
    <IMG SRC="image.gif" ISMAP>
</A>
```

When you click on the image the browser measures the ***pixel*** coordinates of the mouse pointer, relative to the ***upper left hand*** corner of the image, and then accesses the anchored URL, apending the coordinates to the URL using the form ***URL?ix,iy***, where ***ix*** and ***iy*** are the integer x and y pixel coordinates. For example:

```
http://www.se.ca/cgi-bin/imagemap/ex/path?123,17
```

Here the `/ex/path` extra path indicates a database for this particular *image mapped* file.

3. FORMs

HTML FORMs can be used to solicit user input – when submitted, the browser is responsible for encoding the form data and transmitting that data to a server (the URL to which the data are submitted is indicated in the **FORM** element).

Each input element must be given a ***name*** using the **NAME** attribute. This assigns a variable name to the input element.

The ***value*** of the input element can be set either through a **VALUE** attribute (for menus, checkboxes or buttons) or through text input by the user. The data are then sent to the server as a collection of strings of the form

```
name=value
```

The strings name and value are specially encoded, as is the string formed by combining all the **`name=value`** pairs for the different input elements. (discussed later).

Example FORM

Here is an example HTML document that uses the HTML **FORM** tags to collect and send data back to a server. This document is found at:

http://www.utirc.utoronto/ca/CGI_Course/form.html

```
<HTML> <HEAD>
<TITLE> Simple Form Example </TITLE>
</HEAD> <BODY>

<H3> FORM  Example:</H2>
```

URL

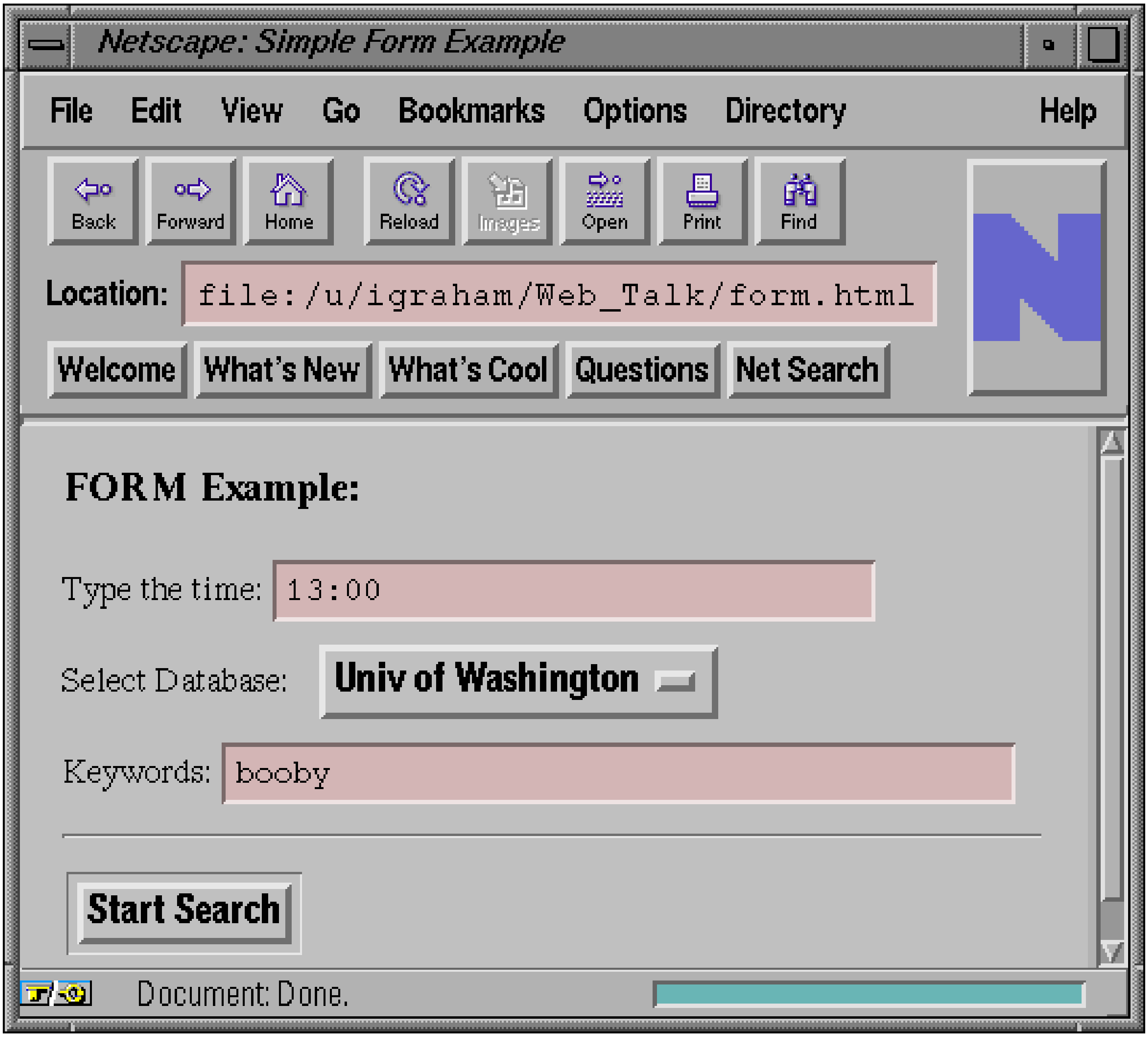
```
<FORM ACTION="http://name.ca/cgi-bin/prgm.pl"
METHOD="GET">
```

HTTP ***method* (*GET*** **or** ***POST*)**

```
Type the time: <INPUT TYPE="textbox" NAME="time"
SIZE=30>  <BR> Select Database:

<SELECT NAME="data_base">
   <OPTION VALUE="harvard"> Harvard On-line
   <OPTION VALUE="toronto"> Univ. of Toronto
   <OPTION VALUE="seattle" selected> Univ of
Washington
</SELECT>
<BR>Keywords:
<INPUT TYPE="text" NAME="srch" SIZE=40>
 <HR>
<INPUT TYPE="submit" NAME="submit" VALUE="Start
Search">
</BODY> </HTML>
```

This document looks like:



Note the various input elements. Pressing *"Start Search"* sends the data to the server using the HTTP ***GET method***

With the HTTP ***GET*** method the data is sent to the server *appended as query data* to the URL used to reference the server–side program designed to understand the form's data.

Thus the URL accessed from the preceeding form is:

```
http://name.ca/cgi-bin/prgm.pl?time=13%3A00&dat
a_base=seattle&srch=booby&submit=Start+Search
```

```
http://name.ca/cgi-bin/prgm.pl?time=13%3A00&data_base=seattle&srch=booby&submit=Star
t+Search)
```

The data is sent as a series of *`name=value`* pairs, separated by the `&` character. Note how Spaces are encoded as +'s, and ***disallowed characters*** are encoded by the `%xx` coding (e.g. the semicolon "**:**" becomes `%3A`).

This is called ***URL encoding*** of the data.

FORM Input Elements

```
<INPUT NAME="name"
       VALUE="value"
       TYPE="checkbox","radio",
            "text", "password",
            "reset", "submit"
            "hidden" ...>

<SELECT NAME="name"
        MULTIPLE SIZE=n>
   <OPTION VALUE="value"...>
</SELECT>


<TEXTAREA NAME="name"...>
   text (becomes the "value")
</TEXTAREA>
```

URL Encoding of ISINDEX and FORM Data

The Encoding scheme has three stages. First the ISINDEX query string (or, the name and value strings from forms) are encoded as follows:

1. Non−ASCII characters (codes >128) are encoded via their URL octal encodings: **`%xx`**

2. Disallowed or special ASCII characters are encoded via their URL encodings. The disallowed or special ASCII characters are:

`` `~!#$%^&()+={}|[]\:";'<>?,/ `` *TAB*

That is: ***every*** punctuation character ***except*** the five characters**:**

`@ * _ -` and **.**

3. Space characters are encoded as plus signs (+).

If an **ISINDEX** query, the browser now appends the query string to the URL. Two more steps are required if the data is from a **FORM**:

4. The encoded `name` and `value` strings are combined into strings, with the equals sign (`=`) as separator. That is:

```
name=value
```

5. The `name=value` strings are combined into a single string, with the ampersand (`&`) as separator. For example:

```
name1=value1&name2=value2&....
```

You need to know all this because:

A) A gateway program must ***decode*** these strings to access the data.

B) You must do the ***encoding*** yourself if you want to hard–code a URL containing a query.

The Common Gateway Interface

The CGI specifies how data are communicated between an HTTP server and server−side gateway programs.

1. Server to Gateway Program

There are ***THREE*** ways data sent from the client to the server is passed to a gateway program:

1. through ***environment variables***, (like the query string and extra path information),
2. as ***command−line arguments*** (uncommon)
3. as ***standard input*** (used when the client sends data to the HTTP server using the HTTP ***POST*** method).

The server also uses environment variable to communicate information about the server and client that are not directly part of the sent query.

There are **TWO** ways information can be passed from the gateway program to the server:

1. data written to ***standard output:***
   - ***Server directives***, followed by
   - ***Data*** to be returned to the client

2. the ***name*** of the gateway program: names beginning with **nph–** (***N***on ***P***arsed ***H***eaders) are special. In this case, the data sent to standard output contains:
   - Complete ***HTTP response header***
   - ***Data*** to be returned to client

CGI Example

(http://www.utirc.utoronto.ca/cgi–bin/test–cgi)

Server directive

```
#!/bin/sh
echo Content-type: text/html
echo
echo "<ISINDEX>"
echo "<HTML><HEAD><TITLE>CGI Test Program
echo " </TITLE></HEAD><BODY><PRE>"
echo CGI/1.0 test script report:
echo
echo argc is $#. argv is "$*".
echo
echo SERVER_SOFTWARE = $SERVER_SOFTWARE
echo SERVER_NAME = $SERVER_NAME
echo GATEWAY_INTERFACE = $GATEWAY_INTERFACE
echo SERVER_PROTOCOL = $SERVER_PROTOCOL
echo SERVER_PORT = $SERVER_PORT
echo REQUEST_METHOD = $REQUEST_METHOD
echo HTTP_ACCEPT = "$HTTP_ACCEPT"
echo PATH_INFO = "$PATH_INFO"
echo PATH_TRANSLATED = "$PATH_TRANSLATED"
echo SCRIPT_NAME = "$SCRIPT_NAME"
echo QUERY_STRING = "$QUERY_STRING"
echo REMOTE_HOST = $REMOTE_HOST
echo REMOTE_ADDR = $REMOTE_ADDR
echo REMOTE_USER = $REMOTE_USER
echo AUTH_TYPE = $AUTH_TYPE
echo CONTENT_TYPE = $CONTENT_TYPE
echo CONTENT_LENGTH = $CONTENT_LENGTH
echo "</PRE></BODY></HTML>"
```

Environment Variables Sent to Client

1. Information about the server:

SERVER_SOFTWARE	PATH_INFO
SERVER_NAME	SCRIPT_NAME
SERVER_PROTOCOL	...
GATEWAY_INTERFACE	

2. Information about client–server link

REMOTE_HOST	REMOTE_ADDR
REMOTE_USER	AUTH_TYPE
REQUEST_METHOD	....

3. Information passed by client to server

QUERY STRING	(*GET* method only, empty if *POST*)
CONTENT_LENGTH	(if *POST* method, otherwise empty)
CONTENT_TYPE	(" " " " " " " ")
HTTP_***NAME***	**contents of all HTTP headers sent from client to server, e.g:**

HTTP_ACCEPT
HTTP_USER_AGENT
HTTP_REFERER
...

Command–Line Arguments

If this is an **ISINDEX** query, the query string is also passed to the gateway program as a series of command line arguments, each space–separated word in the query becoming a single argument.

The server determines if this is an ISINDEX query by looking for unencoded equals signs (=) –– if there are any, then the query string is coming from a **FORM**, and not **ISINDEX**.

Note:

Multiple spaces will give an empty command line argument.

ISINDEX Example
A Phonebook Search

(**http://www.utirc.utoronto.ca/cgi-bin/srch-example**)

Returned data type

```
#!/bin/sh
echo Content-Type:  text/html
echo
if [ $# = 0 ]          # is the number of arguments == 0 ?
then                   # do this part if there are NO arguments
    echo "<HEAD> <TITLE>Local Phonebook Search</TITLE>"
    echo "<ISINDEX> </HEAD> <BODY>"
    echo "<H1>Local Phonebook Search</H1>"
    echo "Enter your search in the search field.<P>"
    echo "</BODY>"
else                   # this part if there ARE arguments
    echo "<HEAD> <TITLE>Result of search for \"$*\".</TITLE>"
    echo "</HEAD> <BODY>"
    echo "<H1>Result of search for \"$*\".</H1>"
    echo "<PRE>"
    for i in $*
    do
       grep -i $i /svc/www/Personnel.list
    done
    echo "</PRE></BODY>"
fi
```

Imagemap Example

(http://www.physics.utoronto.ca/map/stgeorge.html)

Here is an imagemapped image, from the physics department:

```
<title>St. George Campus</title>
<h2>St. George Campus</h2>

<A HREF="/scripts/imagemap/uoftmap">
     <IMG SRC="/map/campusmap.gif" ISMAP>
</a>

<P>Try clicking on building 78 (the physics
building), which is near the lower left
hand corner of the map.<P> Try clicking
on some other buildings and see what you
get. Not all buildings are referenced yet!
If you get any wrong answers, please send
mail to wyllie@physics.utoronto.ca
<P>thanks,
<P>andrew
<HR>
<P> Any questions, comments or bug reports
regarding this server can be sent to
<CODE>www@physics.utoronto.ca</CODE>
```

Program = **`/scripts/imagemap`**
Extra Path Info = **`uoftmap`**

FORM Example
(GET Method)

(http://www.physics.utoronto.ca/Examples/form.html)

```
<HTML>
<HEAD>
 <TITLE> Simple Form Example </TITLE>
</HEAD>
<BODY>
<H3> FORM  Example:</H2>

<FORM
ACTION="http://www.utirc.utoronto.ca/cgi-bin/test-
cgi/extra" METHOD="GET">

Type the time:
    <INPUT TYPE="textbox" NAME="time" SIZE=30><BR>
Select Database:
<SELECT NAME="data_base">
      <OPTION VALUE="harvard">Harvard On-line
      <OPTION VALUE="toronto">Univ. of Toronto
      <OPTION VALUE="seattle">Univ of Washington
</SELECT> <BR>

Keywords: <INPUT TYPE="text"
                 NAME="srch"
                 SIZE=40>
<HR>
<INPUT TYPE="submit"
       NAME="submit"
       VALUE="Start Search">

</BODY>
</HTML>
```

FORM Example

(POST Method)

(http://www.physics.utoronto.ca/Examples/form2.html)

```
<HTML>
<HEAD>
 <TITLE> Simple Form Example </TITLE>
</HEAD>
<BODY>
<H3> FORM  Example:</H2>

<FORM
ACTION="http://www.utirc.utoronto.ca/cgi-bin/test-
cgi/extra" METHOD="POST">

Type the time:
    <INPUT TYPE="textbox" NAME="time" SIZE=30><BR>
Select Database:
<SELECT NAME="data_base">
      <OPTION VALUE="harvard">Harvard On-line
      <OPTION VALUE="toronto">Univ. of Toronto
      <OPTION VALUE="seattle">Univ of Washington
</SELECT> <BR>

Keywords: <INPUT TYPE="text"
                 NAME="srch"
                 SIZE=40>
<HR>
<INPUT TYPE="submit"
       NAME="submit"
       VALUE="Start Search">

</BODY>
</HTML>
```

Decoding a FORM Query String

(hacky perl example)

```
$input=$ENV{"QUERY_STRING"};# input comes from query string;


@tmp=split("&",$input);      # split into separate fields.
foreach (@tmp) {             # iterate over the fields
   ($name,$value) = split("=", $_);
                             # decode URL encodings
   $name  =~  s/%(..)/pack("c",hex($1))/ge;
   $value =~  s/%(..)/pack("c",hex($1))/ge;

     .... not do something useful with it....

}
```

Decoding FORM Standard Input String

(hacky perl example)

```
$input=<STDIN>;               # input comes from query string;
chop($input); chop($input); # truncate trailing CRLF pair

@tmp=split("&",$input);      # split into separate fields.
foreach (@tmp) {             # iterate over the fields
   ($name,$value) = split("=", $_);
                             # decode URL encodings
   $name  =~  s/%(..)/pack("c",hex($1))/ge;
   $value =~  s/%(..)/pack("c",hex($1))/ge;

     .... not do something useful with it....

}
```

GET vs POST

Advantages and Disadvantages:

GET

Advantages: A User can store URL data as a bookmark, and can hence bookmark a location in your collection of FORM documents.

Disadvantages: Query string can get very long with some forms, and can exceed the allowed string length.

POST

Advantages: Any length of data possible (streams to stdin)

Disadvantages: A aser cannot store a document accessed using a POST.

Inter–Form Communication/ State Preservation

With Gateway programming you often need to preserve state between accesses to a gateway program.

Recall that the server has no memory of past connections –– you must, as a gateway programmer, built into your gateway programs mechanisms for recording the state of a transaction.

There are a few ways to do this:

- ***TYPE="hidden"*** form **INPUT** elements
- Temporary local file on the server

TYPE="hidden" INPUT elements:

1. Data is accumulated by FORM in first HTML document.

2. User preses "submit" –– data are sent to a server–side gateway program.

3. Gateway program processes data, dynamically generates new HTML document. The program places all the data from the first FORM inside "hidden" FORM elements.

4. User reads second FORM, adds new info, and pressed "submit". New data, plus the data in the hidden element are sent to the server.

5. And so on.

Temporary Local File:

1. Data is accumulated by FORM in first HTML document.

2. User preses "submit" –– data are sent to a server–side gateway program.

3. Gateway program processes data, dynamically generates new HTML document. The program places all the data from the first FORM inside a temporary file (database), and returns the name of the file in a ***TYPE="hidden"*** FORM **INPUT** element..

4. User reads second FORM, adds new info, and pressed "submit". New data, plus the the name of the state–preservation file, are sent to the server gateway program.

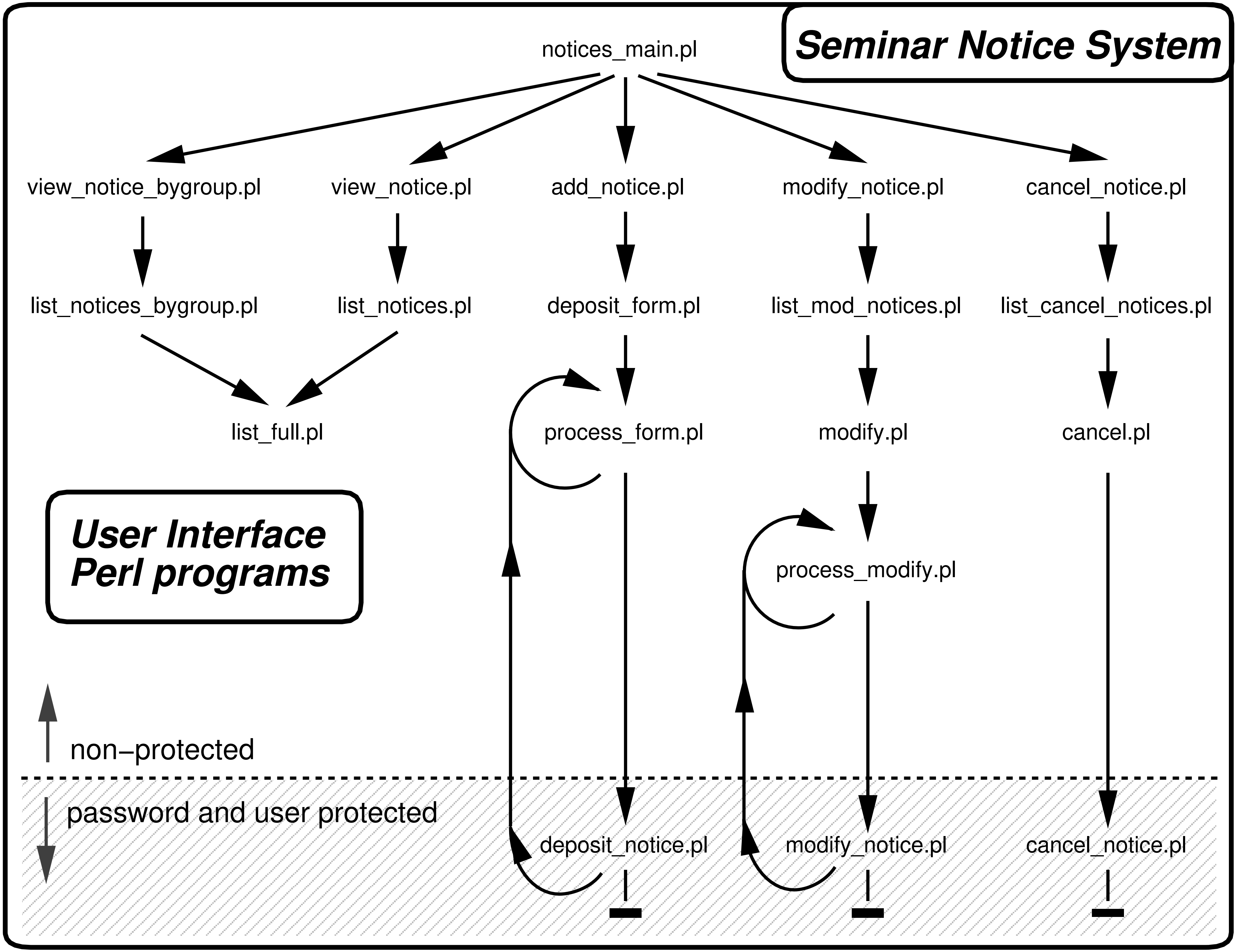
5. And so on.

TYPE="hidden" INPUT elements:

Advantages:	no fiddling with local files
Disadvantages:	slow if lots of data

Temporary Local File:

Advantages:	faster if lots of FORM data
Disadvantages:	nasty programming


Seminar Notice System
notices_main.pl
view_notice_bygroup.pl
view_notice.pl
add_notice.pl
modify_notice.pl
cancel_notice.pl
list_notices_bygroup.pl
list_notices.pl
deposit_form.pl
list_mod_notices.pl
list_cancel_notices.pl
list_full.pl
process_form.pl
modify.pl
cancel.pl
User Interface
Perl programs
process_modify.pl
non–protected
password and user protected
deposit_notice.pl
modify_notice.pl
cancel_notice.pl