

# **CSCI 553: Networking III**

## **Unix Network Programming**

Spring 2007



---

## **Web Server Programming**



# Introduction

---

- Most of the web's power comes from the fact that browsers can interact with programs
  - More accurately, browsers can ask web servers to run programs on their behalf
- This lecture looks at what to do if you *receive* an HTTP request
  - Very important that you go through the lecture on **security** before putting your programs on the web



# The Pluggable Web

---

- Users want to make the web do different things
  - How to let them write programs that handle HTTP requests?
- Option #1: Require them to write socket-level code
  - Complicated and error-prone
  - Can only have one program listening to a socket at a time
- Option #2: have the web server accept the HTTP request, and then run the user's code
  - Recompiling the web server every time someone wants to add functionality would be a pain
  - So define a protocol that lets web servers run other programs



# The CGI Protocol

---

- The *Common Gateway Interface (CGI)* protocol specifies:
  - How a web server passes information to a program
  - How that program passes information back to the web server
- CGI does *not* specify:
  - A particular language
    - You can use Fortran, the shell, C, Java, Perl, Python...
  - How the web server figures out what program to run
    - Each web server has its own rules
    - We'll (briefly) talk about Apache's

# From Server to CGI

- Web server runs the CGI by creating a new process

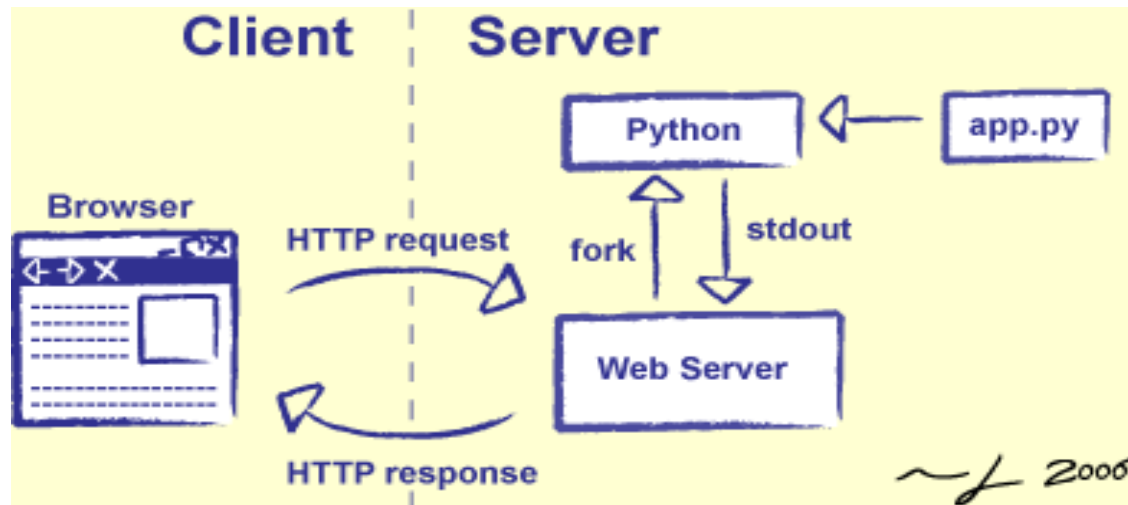


Figure 14.1: CGI Data Processing Cycle



# From Server to CGI

---

- Web server passes some information to the CGI process through environment variables

Name	Purpose	Example
REQUEST_METHOD	What kind of HTTP request is being handled	GET or POST
SCRIPT_NAME	The path to the script that's executing	/cgi-bin/post_photo.py
QUERY_STRING	The query parameters following "?" in the URL	name=mydog.jpg&expires=never
CONTENT_TYPE	The type of any extra data being sent with the request	img/jpeg
CONTENT_LENGTH	How much extra data is being sent with the request (in bytes)	17290

- The web server may also send **CONTENT\_LENGTH** bytes to the CGI on standard input
  - E.g., when a file is being uploaded



# From CGI to Server

---

- The CGI program sends data back to the web server by printing it to standard output
- The web server then forwards this directly to the client
  - Which means that the CGI program is responsible for creating headers
- Note: none of this works unless the web server has been configured to run the CGI
  - By default, modern servers won't do this unless they're told they can



# MIME Types

---

- Clients and servers need a way to specify data types to each other
  - Remember, bytes are just bytes: the browser doesn't magically know how to interpret them
- *Multipurpose Internet Mail Extensions* standard specifies how to do this
  - Organizes data types into families, and provides a two-part name for each type
  - Use the "Content-Type" header to specify the MIME type of the data being sent

<b>Family</b>	<b>Specific Type</b>	<b>Describes</b>
Text	text/html	Web pages
Image	image/jpeg	JPEG-format image
Audio	audio/x-mp3	MP3 audio file
Video	video/quicktime	Apple Quicktime video format
Application-specific data	application/pdf	Adobe PDF document



# Hello, CGI

---

- Simplest possible CGI pays no attention to query parameters or extra data
  - Just prints HTML to standard output, to be relayed to the client
  - Along with a Content-Type header to tell the client to expect HTML...
  - ...and a blank line to separate the headers from the data

```
#!/usr/bin/env python
```

```
# Headers and an extra blank line
```

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

```
print
```

```
# Body
```

```
print '<html><body><p>Hello, CGI!</p></body></html>'
```

# Invoking a CGI

- Invoke it by going to `http://www.yourserver.com/cgi-bin/hello_cgi.py`
  - By convention, CGI programs are put in a `cgi-bin` directory
- Browser displays the simple HTML page generated by the program

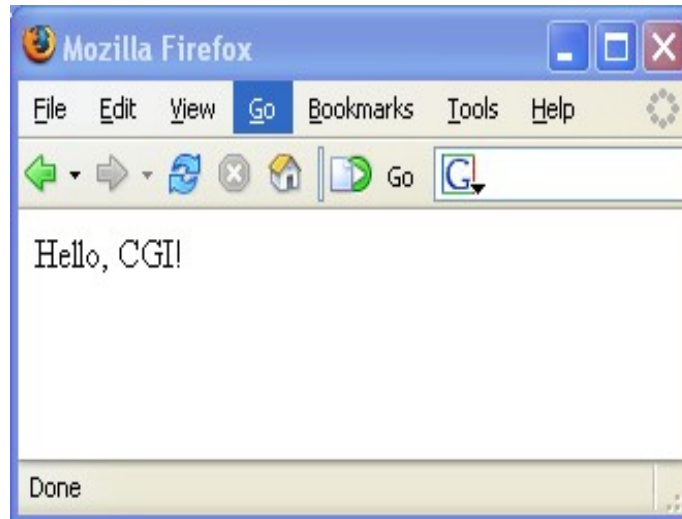


Figure 14.2: Basic CGI Output



# Generating Dynamic Content

---

- But the whole point of CGI is to generate content dynamically
  - E.g., show a list of environment variables and their values

```
#!/usr/bin/env python

import os, cgi

# Headers and an extra blank line
print 'Content-type: text/html'
print

# Body
print '<html><body>'
keys = os.environ.keys()
keys.sort()
for k in keys:
    print '<p>%s: %s</p>' % (cgi.escape(k),
        cgi.escape(os.environ[k]))
print '</body></html>'
```

# Generating Dynamic Content

- You'll use this frequently when debugging...

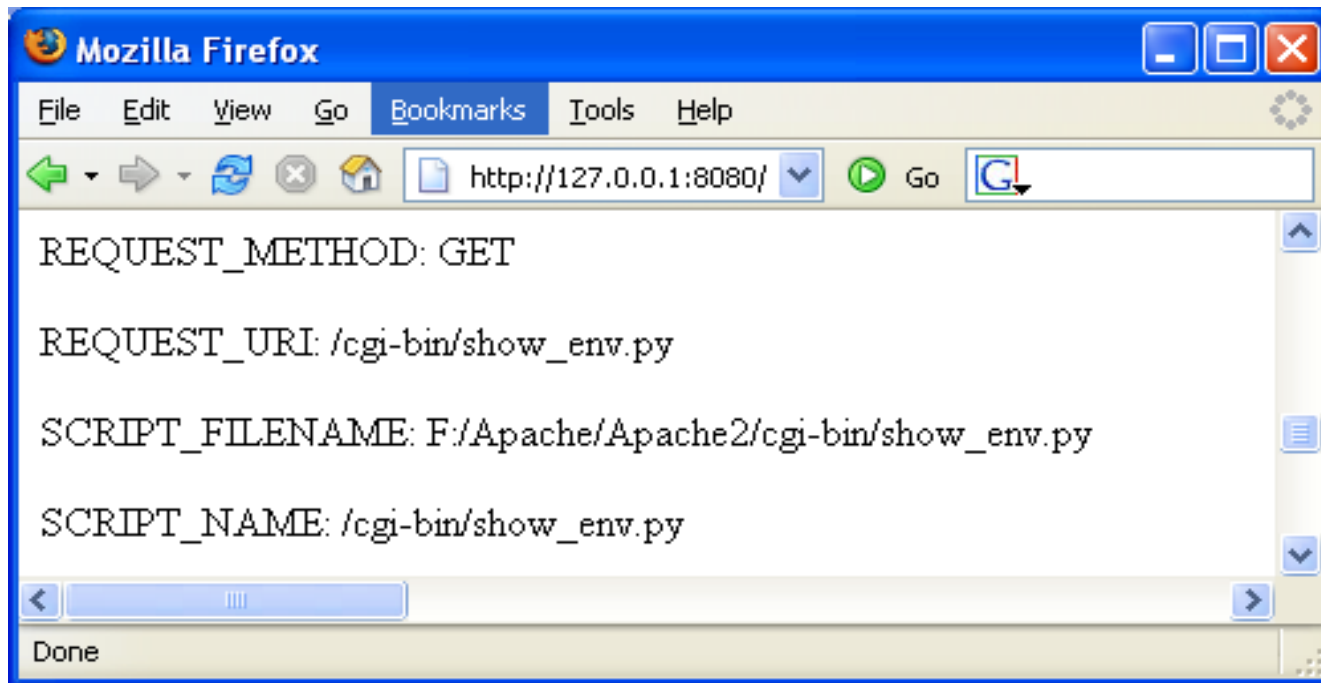


Figure 14.3: Environment Variable Output



# Forms

---

- Next step is to allow users to enter data
  - *Without* manually editing URLs to append parameters
- HTML *forms* allow users to enter text, choose items from lists, etc.
  - Not nearly as sophisticated as desktop interfaces
  - Although programmers are doing more every day (particularly using Javascript)

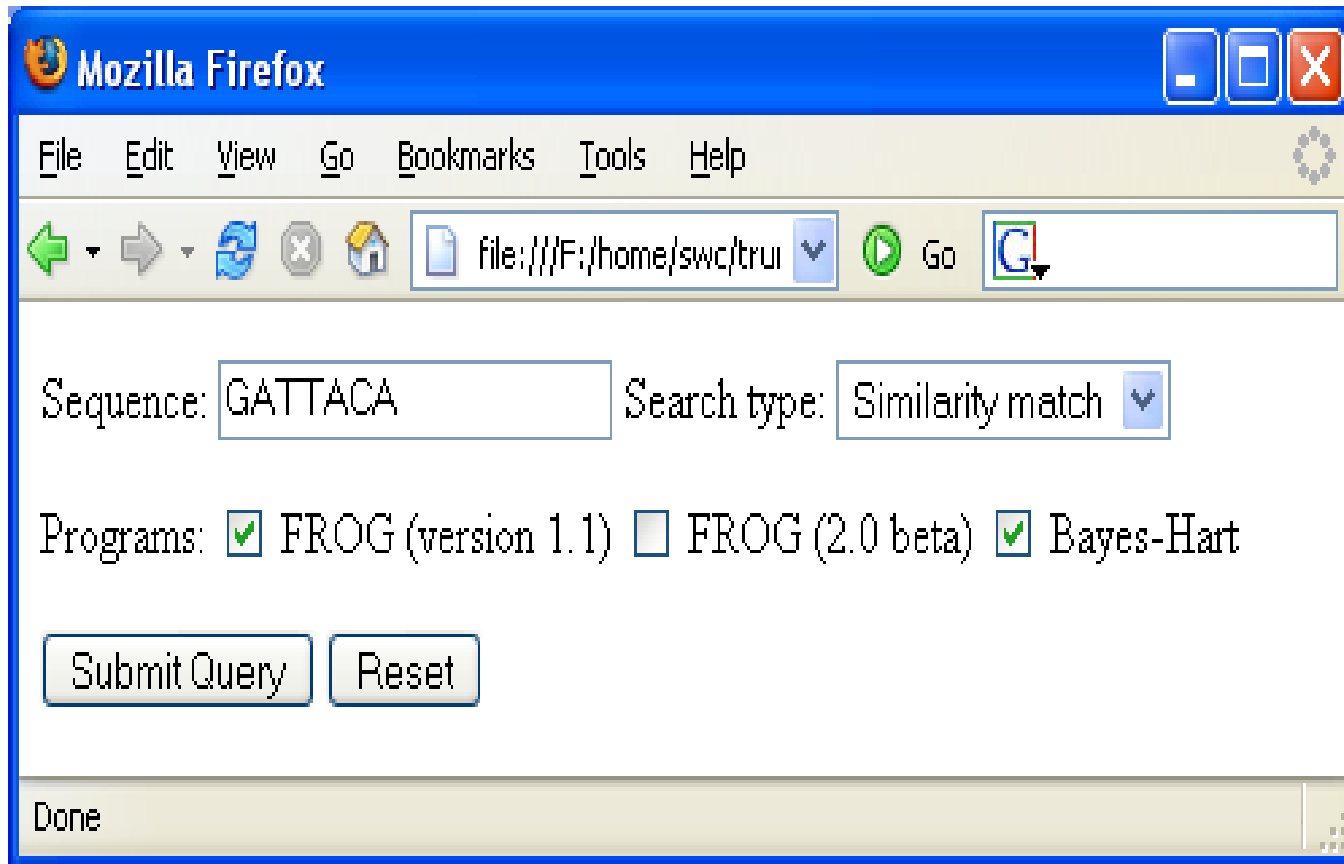


# Creating Forms

---

- Create a form using a `<form>...</form>` element
  - action attribute specifies the URL to send data to
  - method attribute specifies the type of HTTP request to send
    - Usually "POST" for HTML forms
- Inside the form, can have:
  - `<select/>` elements to let users choose values from a list
    - List items specified using `<option/>` elements
  - `<input/>` elements for other kind of data
    - If type is "text", get a one-line text entry box
    - If type is "checkbox", get an on/off checkbox
    - "submit" and "reset" create buttons to submit the form, or re-set the data to initial values

# A Simple Form



The image shows a screenshot of a Mozilla Firefox browser window. The window title is "Mozilla Firefox". The menu bar includes "File", "Edit", "View", "Go", "Bookmarks", "Tools", and "Help". The address bar shows the file path "file:///F:/home/swc/trui" and a "Go" button. The main content area contains a form with the following elements:

- Sequence:
- Search type:
- Programs:  FROG (version 1.1)  FROG (2.0 beta)  Bayes-Hart
- Buttons:

The status bar at the bottom of the window displays "Done".

Figure 14.4: A Simple Form



# Parameter Names

---

- Each `<input/>` element has a name attribute
  - These become the names of the parameters that the client sends to the server
  - The input elements' values are the parameters' values
- Submitting the form shown above with default values produces:
  - `os.environ['REQUEST_METHOD']`: "POST"
  - `os.environ['SCRIPT_NAME']`: `"/cgi-bin/simple_form.py"`
  - `os.environ['CONTENT_TYPE']`: `"application/x-www-form-urlencoded"`
  - `os.environ['REQUEST_LENGTH']`: "80"
  - Standard input: `sequence=GATTACA&search_type=Similarity+match&program=FROG-11&program=Bayes-Hart`



# Handling Forms

---

- *Could* handle form data directly
  - Read and parse environment variables
  - Read extra data from standard input
- But the mechanics are the same each time, so use Python's `cgi` module instead
  - Defines a dictionary-like object called `FieldStorage`
    - Keys are parameter names
    - Values are either strings (if there's a single value associated with the parameter) or lists (if there are many)
- When a `FieldStorage` object is created, it reads and stores information contained in the URL and environment
  - Which means that a CGI program should only ever create one
- Program can read extra data from `sys.stdin`



# Form Handling Example

---

- Example: show the parameters send to a script

```
#!/usr/bin/env python
import cgi

print 'Content-type: text/html'
print
print '<html><body>'
form = cgi.FieldStorage()
for key in form.keys():
    value = form.getvalue(key)
    if isinstance(value, list):
        value = '[' + ', '.join(value) + ']'
    print '<p>%s: %s</p>' % (cgi.escape(key), cgi.escape(value))
print '</body></html>'
```

## URL

[http://www.third-bit.com/swc/show\\_params.py?a=0](http://www.third-bit.com/swc/show_params.py?a=0)

[http://www.third-bit.com/swc/show\\_params.py?  
a=0&b=hello](http://www.third-bit.com/swc/show_params.py?a=0&b=hello)

[http://www.third-bit.com/swc/show\\_params.py?  
a=0&b=hello&a=22](http://www.third-bit.com/swc/show_params.py?a=0&b=hello&a=22)

## Value of

**a**

"0"

"0"

[0, 22]

## Value of

**b**

None

"hello"

"hello"



# Development Tips

---

- During development, add `import cgitb; cgitb.enable()` to the top of the program
  - `cgitb` is the CGI traceback module
  - When enabled, it will create a web page showing a stack trace when something goes wrong in your script
- Testing whether a `FieldStorage` value is a string or a list is tedious
  - In almost all cases, you'll know whether to expect one value or many
  - Use `FieldStorage.getfirst(name)` to get the unique value
    - Returns the first, if there are many
  - `FieldStorage.getlist(name)` always returns a list of values
    - Empty list if there's no data associated with name
    - If there's only one value, get a single-item list

# Maintaining State

- Often want to change the data a server is managing, as well as read it
  - Update a description of an experiment, change your preferred email address, etc.
- The industrial-strength solution is to use a *three-tier architecture*

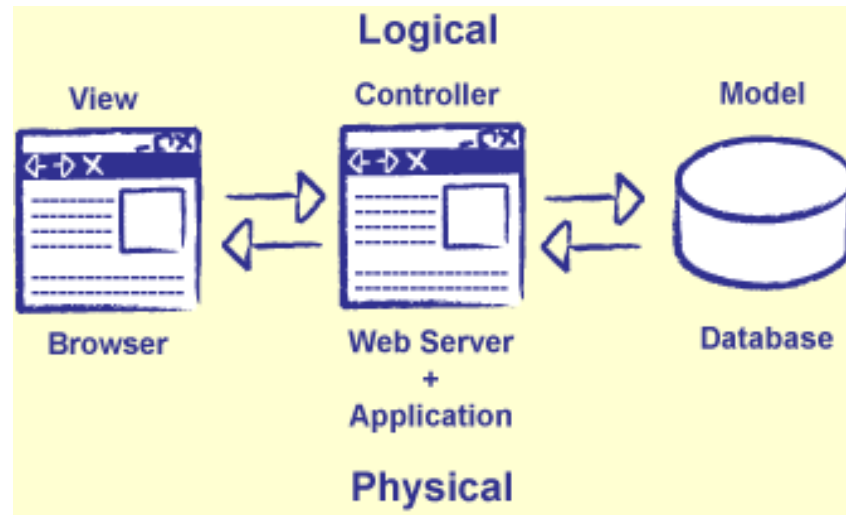


Figure 14.5: Three Tier Architecture

- CGI program stuffs parameters from HTTP requests into SQL queries
- Runs the queries
- Translates results into HTML to send back to the client



# Maintaining State in Files

---

- Simple programs can often get away with using files
  - The CGI program re-reads the file each time it processes a request
  - And re-writes it if there have been any updates
- Example: append messages to a web page
  - Old messages are saved in a file, one per line

Hi, is anyone reading this site?

I was wondering the same thing.

I wasn't sure if we were supposed to post here.

Good point. Is there way to delete messages?



# Maintaining State in Files

---

- Script checks the incoming parameters to decide what to do
  - If newmessage is there, append it, and display results
  - If newmessage *isn't* there, someone's visiting the page, rather than submitting the form

```
# Get existing messages.  
infile = open('messages.txt', 'r')  
lines = [x.rstrip() for x in infile.readlines()]  
infile.close()
```

```
# Add more data?  
form = cgi.FieldStorage()  
if form.has_key('newmessage'):  
    lines.append(form.getfirst('newmessage'))  
    outfile = open('messages.txt', 'w')  
    for line in lines:  
        print >> outfile, line  
    outfile.close()
```



# HTML Generation

---

- Note that there is no static HTML page in this example
  - What the user sees is always generated by a program

```
# Display.
print 'Content-Type: text/html'
print
print '<html><body>'
for line in lines:
    print '<p>' + line + '</p>'
print '''
<form action="http://www.third-bit.com/swc/message_form.py" method="post">
  <p>Your message:
    <input name="newmessage" type="text"/>
  </p>
  <p>
    <input type="submit"/>
    <input type="reset"/>
  </p>
</form>
'''
print '</body></html>'
```



# HTML Templating

---

- A lot of this program is devoted to copying values into an HTML *template*
  - There are lots of good systems out there, in many languages, for doing this
    - *Kid*, [WebWare](#) in Python
    - *Java Server Pages (JSPs)* in Java
    - Please do *not* write one of your own



# What About Concurrency?

---

- What happens if two users try to save messages at the same time?
  - I/O is typically slower than processing
  - So most web servers try to overlap operations
- Race condition:
  - First instance of message\_form.py opens messages.txt, reads lines, closes file
  - Second instance opens messages.txt, reads *the same lines*, closes file
  - First instance re-opens file, writes out original data plus one new line
  - Second instance re-opens file, writes out original plus a *different* new line
  - First instance's message has been lost!



# File Locking

---

- Solution is to *lock* the file
  - As the name implies, gives one process exclusive rights to the file
  - After the first process *acquires* the lock, any other process that tries to read or write the file is *suspended* until the first *releases* it
- Mechanics are different on different operating systems
  - But the *Python Cookbook* includes a generic file locking function that works on both Unix and Windows



# Implementing Locking

---

```
# Get existing messages.
msgfile = open('messages.txt', 'r+')
fcntl.flock(msgfile.fileno(), fcntl.LOCK_EX)
lines = [x.rstrip() for x in msgfile.readlines()]

# Add more data?
form = cgi.FieldStorage()
if form.has_key('newmessage'):
    lines.append(form.getfirst('newmessage'))
    msgfile.seek(0)
    for line in lines:
        print >> msgfile, line

# Unlock and close.
fcntl.flock(msgfile.fileno(), fcntl.LOCK_UN)
msgfile.close()
```



# Who Are You?

---

- How to maintain state on the client?
  - Need to know which shopping cart to display for a particular user
- HTTP is a stateless protocol
  - If a client makes a second (or third, or fourth...) request, server has no reliable way of connecting it to the first one
- Can guess based on client address, elapsed time, etc.
  - But it's just a guess

# Cookies

- Solution is for the server to create a *cookie*
  - A string that is sent to the client in an HTTP response header
- Client saves it (either in memory or on disk)

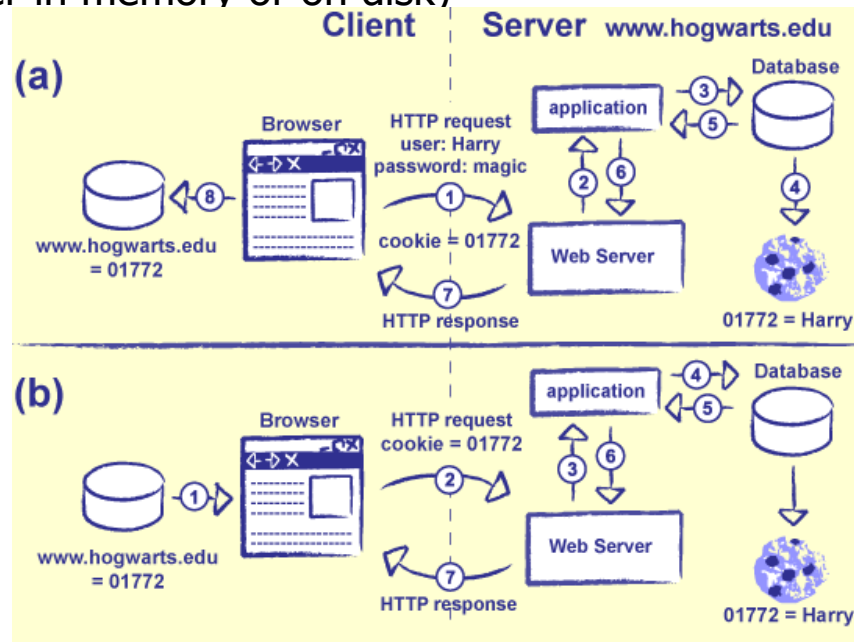


Figure 14.6: Cookies

- The next time the client sends a request to the site, it sends the cookie back to the server
  - Like giving someone a claim check for their luggage



# Creating Cookies

---

- Represent cookies in Python using `Cookie.SimpleCookie`
  - Do not use `SmartCookie`: it is potentially insecure
- When creating, add values to a cookie as if it were a dictionary
  - Convert it to a string (e.g., by printing it) to create the required HTTP header
- When the cookie comes back:
  - Get the value associated of the environment variable `"HTTP_COOKIE"`
  - Create a `SimpleCookie`
  - Pass the `"HTTP_COOKIE"` value to the cookie's `load` method



# Cookie Example

---

- Example: count the number of times a user has visited a web site
  - If there's no cookie, create one with a count of 1
  - Otherwise, increment the count
  - Create a new cookie to send back to the user
  - Display the count



# Cookie Example

---

```
# Get old count.
count = 0
if os.environ.has_key('HTTP_COOKIE'):
    cookie = Cookie.SimpleCookie()
    cookie.load(os.environ['HTTP_COOKIE'])
    if cookie.has_key('count'):
        count = int(cookie['count'].value)

# Create new count.
count += 1
cookie = Cookie.SimpleCookie()
cookie['count'] = count

# Display.
print 'Content-Type: text/html'
print cookie
print
print '<html><body>'
print '<p>Visits: %d</p>' % count
print '</body></html>'
```



# Cookie Tip

---

- Can control how long a cookie is valid by setting an expiry value
  - Either the number of milliseconds
  - Or the time it should expire (in *UTC*)
    - Use `time.asctime(time.gmtime())` to create the value
- Do *not* put sensitive information in cookies
  - Browsers store them in files on disk
  - Villains can watch network traffic, and steal data
- Cookies should instead be random values that act as keys into server-side information
  - Talk about this more in the *next lecture*



# Summary

---

- CGI is example of *event-driven programming*
  - The framework invokes your code at specific times, and passes it specific information
  - What happens the rest of the time isn't your concern
    - At least, until something goes wrong, and you have to debug it
- Simple CGI programs can accomplish a lot
  - The entire first generation of web applications were built this way
- But they can easily become very complicated
  - Discuss alternatives in the *final lecture*