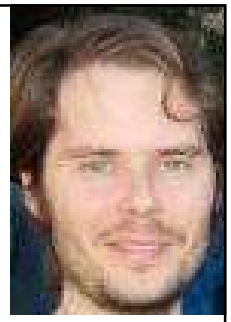
CSCC09F
Programming on the Web

HTTP Details for Developers

GET & POST methods, URL-encoding, session-state handling mechanisms

Dynamic Web Content


Rob McCool

- First generation Web servers provided primarily static, information-only HTML pages.
- Problematic for sites that wished to allow users to interact with their sites
- Needed:
  - dynamic server-side content generation
  - database interconnectivity
  - connection to existing legacy applications and new applications
- CGI invented at NCSA (home of Mosaic browser, precursor to Netscape) to solve this problem

Dynamic Server Technologies

- Common Gateway Interface (CGI) framework
  - typically used with dynamic scripting languages such as: Perl, Python; however, most languages will do
- Code Embedded within HTML page:
  - Server-Side Includes (SSI)
  - PHP (scripts embedded in HTML)
  - JSP (Java embedded in HTML)
  - Microsoft Active Server Pages (ASP) (JScript embedded in HTML, non-precompiled)
  - Server-Side JavaScript (JavaScript embedded in HTML, precompiled), Netscape LiveWire
- Java Servlets

CGI

- Why do we care about CGI, hasn't it has largely been replaced by more advanced technologies?
- True; however, CGI illustrates several important points in the context of a nice simple standard:
  - How values are passed from client environment to server (effectively parameters in Web application function calls)
  - How data is encoded for transmission from client to server and back
  - How forms hook into back-end requests
  - Also, CGI remains useful for quickly throwing together dynamic serving of content that doesn't justify use of an enterprise approach like Servlets
  - Some examples of how this is done are posted on the lectures Web page, but we won't be doing any CGI programming

CGI, An Example

- go to hello.cgi This causes the execution of the perl script hello.cgi (below)

```
#!/usr/local/bin/perl
print "Content-type:text/html\n\n";
print “Hello World!\n";
```

- Although posted examples use Perl and Python, CGI scripts can be written in most languages available on the Server, e.g. C, C++, VB, “shell” languages such as ksh, Ruby, Lisp, SmallTalk, etc...

HTTP "GET" Request Method

- Default method for communicating query information to server-side handler programs (e.g. Servlet, CGI)
- In general this does not result in a state change on the server – like a "getter" method
- URL identifies server-side handler program
- Don't need a form; why?
- Everything after the "?" in the URL is made available in the CGI **`QUERY_STRING`** environment variable, and through methods to Java Servlets

GET disadvantages

- Limited amount of information can be passed with GET. URL itself may have a length restriction imposed by client and/or server
- User must "URL-Encode" parameters for self when CGI not called through form. Any risks here?
- **`QUERY_STRING`** is visible in the URL (at the browser) and appears in server logs (which are sometimes public) – so not suitable for privacy-sensitive or secret parameter info

HTTP POST Request Method

- Use post for requests that change server state, e.g. modify database
- Cannot use POST using a URL (as with GET); instead initiated with HTML form submission, or JavaScript action such as Ajax request.
- To initiate from a form:
  - specify `method="post"`
  - form data automatically "URL-Encoded" (see below) by the browser before being sent to the server.
- But, can still populate **`QUERY_STRING`** by specifying it explicitly with the CGI URL
- Form data made available to CGI script as its stdin (standard input), and through methods for Servlets

POST advantages

- Arbitrarily long form data can be communicated in principle (however, some browsers impose limits).
- Form data is not visible in the URL, and usually does not appear in server log files, thus better than GET for privacy sensitive data.

- Any disadvantages?

URL-Encoding

- Standard way to encode name/value pairs as a single string (**`QUERY_STRING`** or **`stdin`**)
- Specified in RFC 3986:
  'Uniform Resource Identifiers (URI): Generic Syntax'
- Why encode?
  - Conform to URL syntax rules which deem certain characters to be “unsafe”, e.g. spaces (which may be changed when transcribed or printed), tag delimiters (< and >) to avoid confusion between CGI URL and HTML, the # character (since it is a URL fragment marker), etc.
  - Forms often encode multiple values, want a standardized way to pack and unpack them into a single string value (**`QUERY_STRING`** or **`stdin`**)

URL-Encoding Rules

- All submitted form data is concatenated into a single string of ampersand (&) separated name=value pairs, one pair for each form tag. Like this:

```
form_field_1=value_1
&form_field_2=value_2
&...
```

URL-Encoding Rules

- Spaces in a name or value are replaced by a plus (+) sign. This is because URL's cannot have spaces in them, and for a form with attribute **`method="GET"`**, the form data is inserted into the URL
- Other special characters (eg =, &, +) are replaced by a percent sign (%) followed by the two-digit hexadecimal equivalent of that punctuation character in the ASCII character set, e.g. %26 for &

Form Example

See form.html

Notes:

- Observe the URL-encoded form variables in the GET form
- Observe stdin in the POST form
- Observe the hidden variables in both forms

Statelessness

- Recall from earlier coverage of HTTP that this client-server protocol is stateless.
  - good for keeping the server simple
  - but is a bit of a problem for implementing transaction-oriented services over the Web, where a user performs a sequence of interactions that constitute a single "session"
    - e.g. online shopping, banking, airline flight reservation, concert ticket order, etc.
  - how can server-side code tell which requests are from a user who already has an active session in progress – e.g. a partially filled shopping cart?

Statelessness

- We're going to look at each of the options in common use for tracking state, roughly in order of increasing popularity and utility (i.e. save the best for last)
- Pay particular attention to the limitations and drawbacks associated with the different approaches
  - some approaches not suitable for certain kinds of applications (e.g. privacy, security concerns)
  - it may be necessary to use a combination of approaches for some applications

Statefulness

- What can we do with session state info? All kinds of magical (and useful) things:
  - "shopping cart" – keep track of selections made across multiple client requests – applicable to many areas of e-commerce
  - identification of users and sessions – e.g. for authenticated access to secure sites such as online banking
  - track user movement through Web site (e.g. what has user looked at) – think about how Amazon keeps track of your browsing/shopping behavior/history
  - customize views of Web site using user preferences or previous behavior ("I only browse items that are on sale").

Tracking State: Hidden Variables

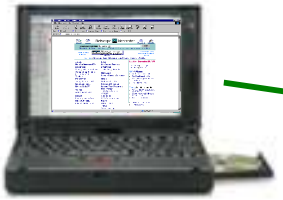
- Store state within server-generated Web pages using hidden variables in forms
  - **<input type="hidden" …>**
  - applicable for sessions that consist of a sequence of form-based interactions between browser and server.
  - current form represents server's view of current state; thus requires all state-representation hidden-variables must appear in all forms (otherwise, the state is lost)

Hidden Variables

Example:

- initial form asks for your name.
- all subsequent forms use your name to address you personally, “dear Sammy” …
- how does the computer know my name!?
- each and every form must be server generated. why?
- each form-generating server program reads the previous form's fields, extracts your name, re-embeds it in a new hidden field.

Hidden Variables



Client page request,
usually based on form

Server response with
server-generated form
including hidden form-field

time

Client submission of form with
hidden field + other info

Server response with
(new) server-generated form
including hidden form-field

Hidden Variables

- remember, current submitted form page represents server's (complete) view of current state; thus if you omit something, it disappears from the session state!

Example: (see AddForm.html (view source))

```
<input type=hidden name=secret
       value="Don't tell anyone!!">
```

Hidden Variables pro & con

- Advantages:
  - supported by all browsers
  - user cannot disable this feature
- Limitations
  - hidden fields only work across a series of back-to-back form submissions (lose state if visit a non-hidden-state-form server reference).
  - easy to view hidden fields with “view source” – thus not suitable for private or secret information.
  - relatively easy for users to modify hidden fields, thus setting their own state – risks?

Tracking State: URL Rewriting

- Store state in page URL's: rewrite embedded URLs for your site so that they include state variables
  - ❑ Each URL, that refers to pages that are part of your app, transformed into a GET request; why?
  - ❑ Requires all URL's contain all state information (leading to possibly long URL's)
  - ❑ Current submitted GET request represents server's view of current session state

URL Rewriting

Client page request

every server-referencing URL on returned page has …?var1=val1&var2=val2&…, appended, for all state var's

time

Client page request based on embedded URL reference

every server-referencing URL on returned page has …?var1=val1&var2=val2&…, appended, for all state var's

URL Rewriting Examples

- Try out example: URLRewrite.cgi
  - Follow the links a few times
  - Play with the reload and back buttons
  - Visit the site by URL alone:

    `URLRewrite.cgi`
- Trick the application into thinking you have visited it 1000 times.

URL Rewriting pro & con

- Advantages:
  - supported by all browsers
  - user cannot disable this feature
- Limitations
  - URL's highly visible
    - e.g. to user via location window, in status window on link mouseover, via view-page source, recorded in server logs
    - not suitable for private or secret information.
  - relatively easy for users to modify URL's, thus setting their own state – risks? Believe it or not, it happens!
  - browser/server restrictions on URL length

Tracking State: Cookies

- Store state in the browser: cookies
  - server-side programs can create cookies and send them to the browser in the response HTTP header, using the **`Set-Cookie`** response header.
  - cookies are name=value pairs, e.g.

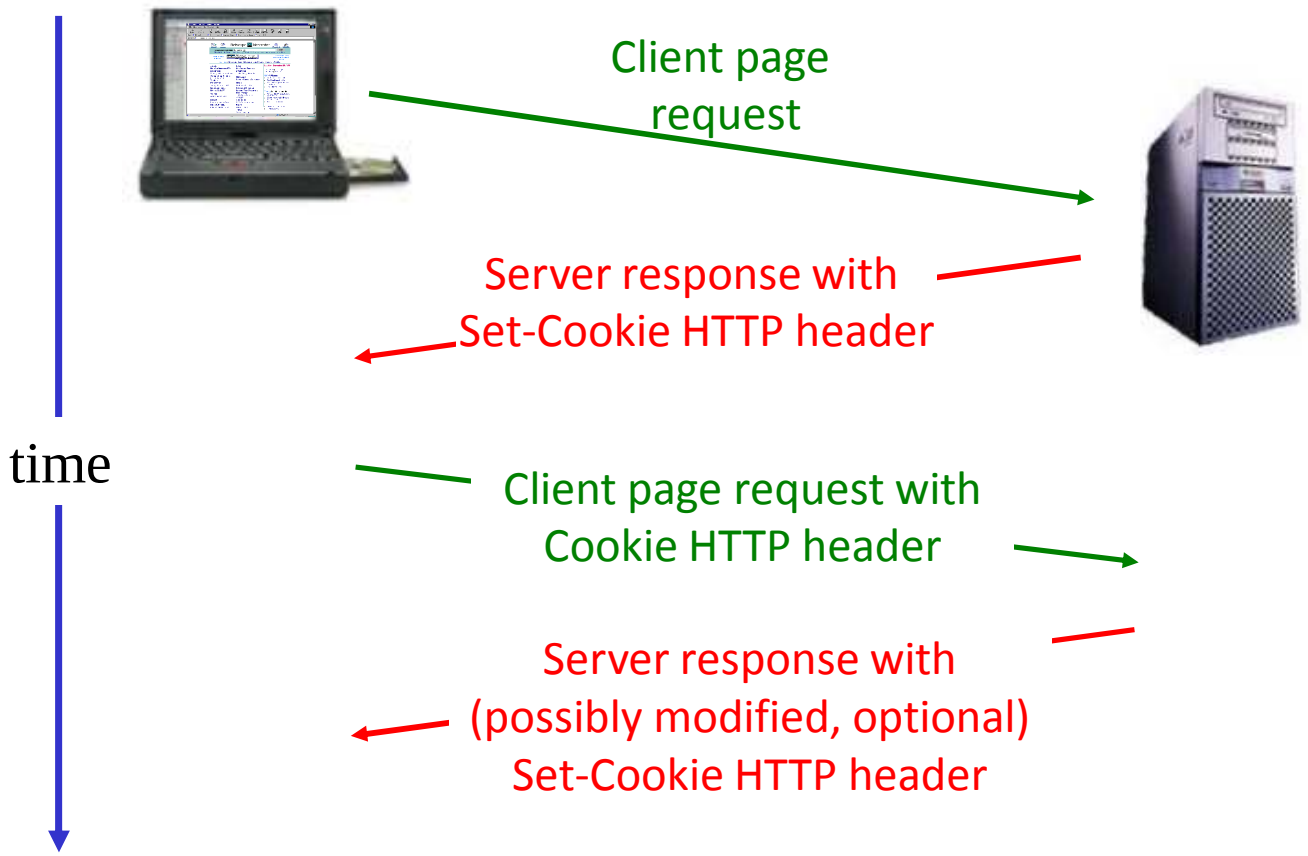
```
Set-Cookie: name=user; value=utorid;
expires=Wed, 24-Oct-2014 15:32:11 GMT;
domain=mathlab.utsc.utoronto.ca; login=true
```

  - each **`Set-Cookie`** header creates one distinct cookie on the client

Tracking State: Cookies

- Store state at the browser: cookies
  - browser maintains a table associating cookies with Web server that sent them
  - during subsequent interactions with that same Web server, uses **`Cookie`** HTTP request header to include cookies with HTTP request
  - what does the server do with cookies – isn't it server side programs that actually need to get their hands on the cookies?

Tracking State: Cookies



Tracking State: Cookies

Example:

LoginForm.html try visiting the Count link before and after logging in

Cookie optional attributes

- Expiration time - cookie stored and returned until expiration time (at which point browser automatically deletes it). If not specified, cookie deleted on next browser shutdown.
- Domain - (eg utsc.utoronto.ca) return cookie to any server in this domain. If not specified, return cookie only to server host sending cookie.
- Path - cookies sent only to scripts within the path. If not specified, return cookies to any script
- Secure flag - return cookies only on a secure (SSL/HTTPS) channel – when might this be important?

Cookie pro & con

- Advantages:
  - supported by all current browsers
  - state data is relatively private and somewhat protected against user tampering (but not truly secure)
  - much more powerful model of state: user not restricted to consecutive-request interaction with server, can be persistent for long periods of time, control of cookie distribution (which servers get them).
- Limitations
  - user can disable or block cookie handling. Yikes!
  - browsers limit the size of cookies (at least 4K each, up to 20 cookies per server)

Tracking State so far

- state of client-server interaction represented as extra information passed back and forth between client and server
- Hidden fields – size not restricted like URL rewriting, but requires continuous (uninterrupted) submission of form data to maintain state
- URL rewriting – works on all clients, but limits volume of state info, and highly visible (URL location field and server logs)
- Cookies – stored on client and passed to server with each request; very flexible.

Tracking State - Caching

- Beware the "back-button problem": page user is viewing may not reflect state stored at the server, may expose private info
- e.g, hotel reservation, user backs up to earlier state to change date of arrival; what should server do?
  - causes problems for client-side state handled with hidden fields or URL-Rewriting
- User Sam logs onto banking site from airport kiosk, logs out; next user uses back button to view Sam's banking info?
- can prevent browser from caching state using HTTP header options:
  - **`pragma: no-cache`** (for HTTP 1.0 and supporting older browsers)
  - **`Cache-Control: no-cache`** and **`no-store`** (for HTTP 1.1)

Examples of Cache Blocking

Go to notcached.cgi, then follow the link to the cached version, finally, use the back and forward buttons a few times and observe whether the times remain static or advance forward (indicating a non-cached page) as you flip back and forth between the non-cached and cached versions.

Sources:

notcached.cgi (can not be cached)

cached.cgi (can be cached)

Storing State at Server

- state is stored at the server (e.g. in a database)
- each client request includes a token identifying the browser session
  - tokens passed via cookies, hidden var's, or URL rewriting
  - token is a reference to actual state data values (not values themselves). e.g.: a random id number; privacy implications?
- key feature of this design: only the token is passed, actual state values are stored only on the server
  - recall size limitations, privacy concerns described above
- at each request, the executing CGI script uses the token to fetch associated session state

State-Tracking Summary

- Combinations of the above techniques (server-side state, cookies, URL-rewriting, hidden fields) are used in practice to track session state information
- Server-side state avoids the "back button" problem, even for URL-rewriting and hidden-fields, provided session token is not changed for duration of session.
- Disabling page caching avoids the security/privacy issues for pages cached by browsers

State-Tracking Summary

- Server-side state can survive client crash if cookies are used, and can survive server crash provided state is written to persistent storage prior to crash.
- For authentication-based services, store the uid and password only on the server, and have server delete the session record after logout, or a certain elapsed time (abandoned session).