Internet Cookies

by Bob Swart

CGI and ISAPI server-side applications communicate using HTTP, which is a stateless protocol. This means that in order to save state information in our web applications, we must do something special: yes, that's where we can use cookies. In fact, there are three common ways to save state information: fat URLs, hidden fields and cookies.

Let's assume we have a CGI application or an ISAPI DLL called WebServ, which starts by asking our name and which needs to maintain the value of our name for the remainder of the session. If no name can be maintained, then WebServ would need to ask for the name every time a user re-connects or even changes from one HTML page to another. This is one of the reasons for the need to maintain state.

Fat URLs

A common way to retain state information is by adding a variable with a value to the URL itself. For example, to maintain the fact that my name is Bob, I could add a variable `Name` with value `Bob` to the URL as follows:

```
http://www.drbob42.com/cgi-bin/
  WebServ.exe?Name=Bob
```

The above line contains the direct call (ie the fat URL itself). We can also embed the variable inside the `ACTION` part of the HTML `FORM` tag, as follows:

```
<FORM ACTION=
  "http://www.drbob42.com/
  cgi-bin/WebServ.exe?Name=Bob"
  METHOD=POST>
```

Note that the general `METHOD` to send `FORM` variables is still `POST`, although the state (`Name`) variable is passed using the `GET` protocol. This means we'll see the name and its value appear on the URL: something that can be experienced with some search engines on the web as well. It also means that our WebServ application must be able to handle both `GET` and `POST` at the same time (ie obtain the value of `Name` using the `GET` protocol and other input fields using the `POST` protocol). Fortunately, Delphi WebBroker components implement this using the `QueryFields` (`GET`) and `ContentFields` (`POST`) properties of the incoming request.

Personally, I believe that any information sent on the URL is error-prone, so I generally try to avoid it. However, using the `POST` method to send regular form fields and the `GET` method to send state fields is actually a nice way to separate the two kinds of fields; if your web server application is able to obtain them both, that is.

Hidden Fields

Using hidden fields is the second, and in my book the most flexible, way to maintain state information. A bit like the fat URL inside the `ACTION` part of the HTML form, which is also invisible, we enter hidden fields inside the CGI HTML form. The syntax for a hidden field is much like the syntax for a regular input field:

```
<INPUT TYPE=HIDDEN
  NAME="Name" VALUE="Bob">
```

This indicates that the hidden field called `Name` has a value of `Bob`. Hidden fields are invisible to the end-user, but the names and values are sent back to the web server and application as soon as the user hits the `Submit` button. All the hidden fields and values are passed using the `ACTION` method (ie `GET` or `POST`) of the entire CGI form, so the web server application only needs to perform the `GET` or `POST` protocol to get all the fields (hidden or otherwise).

Although hidden fields are quite flexible, and invisible yet always present when you need them, there's one thing that limits them: persistence over multiple sessions. Once you 'wander away' from the website and return later (five minutes, an hour or a month), you need to re-enter your name again, because hidden fields are only part of the content inside your browser. They 'live' as long as you have the CGI HTML form inside your browser window, and no longer.

For multi-session persistence, even years later, we need cookies!

Cookies

Cookies are sent by the server to the browser. When using cookies, the initiative is with the web server, but the client has the ability to deny or disable a cookie. Sometimes, servers even send cookies when you don't ask for them, which can be a reason why some people dislike cookies (like I did for years, for example).

There comes a time, however, when cookies are really useful, for example when maintaining state information *beyond a single session*. In these cases, when information must be retained for a period of time, cookies are just about the only possible solution.

➤ *Listing 1: WebServ, set and get cookies.*

```
program WebServ;
uses
  DrBobCGI, SysUtils;
begin
  writeln('content-type: text/html');
  writeln('Set-Cookie: Name=Bob; path=/'); // non-persistent cookie
  writeln;
  writeln('<HTML>');
  writeln('<BODY>');
  writeln('<H1>Cookies</H1>');
  writeln('<HR>');
  writeln(CookieValue(''));
  writeln('</BODY>');
  writeln('</HTML>');
end.
```

```
unit DrBobCGI;
interface
type
  TRequestMethod = (Unknown,Get,Post);
var
  RequestMethod: TRequestMethod = Unknown;
  ContentLength: Integer = 0;
  function Value(const Field: ShortString): ShortString;
  function CookieValue(const Field: ShortString):
    ShortString;
implementation
uses
  Windows, SysUtils;
function _Value(const Field: ShortString; const Data:
  AnsiString; Sep: Char = '&'): ShortString;
  var
    i: Integer;
    Str: String[3];
    len: Byte absolute Result;
  begin
    len := 0; { Result := '' }
    i := Pos('&'+Field+'=',Data);
    if i = 0 then begin
      i := Pos(Field+'=',Data);
      if i > 1 then
        i := 0
    end else
      Inc(i); { skip '&' }
    if i > 0 then begin
      Inc(i,Length(Field)+1);
      while Data[i] <> Sep do begin
        Inc(len);
        if Data[i] = '%' then begin // special code
          Str := '$00';
          Str[2] := Data[i+1];
          Str[3] := Data[i+2];
          Inc(i,2);
          Result[len] := Chr(StrToInt(Str))
        end else
          Result[len] := Data[i];
        Inc(i)
      end
    end
  end {_Value};
const
  Data: AnsiString = '';
function Value(const Field: ShortString): ShortString;
  begin
    Result := _Value(Field, Data)
  end;
const
  Cookie: AnsiString = '';
function CookieValue(const Field: ShortString): ShortString;
  begin
    Result := _Value(Field, Cookie, ';')
  end;
var
  P: PChar;
  i: Integer;
  Str: ShortString;
initialization
  P := GetEnvironmentStrings;
  while P^ <> #0 do begin
    Str := StrPas(P);
    if Pos('REQUEST_METHOD=',Str) > 0 then begin
      Delete(Str,1,Pos('=',Str));
      if Str = 'POST' then
        RequestMethod := Post
      else if Str = 'GET' then
        RequestMethod := Get
    end;
    if Pos('CONTENT_LENGTH=',Str) = 1 then begin
      Delete(Str,1,Pos('=',Str));
      ContentLength := StrToInt(Str)
    end;
    if Pos('QUERY_STRING=',Str) > 0 then begin
      Delete(Str,1,Pos('=',Str));
      SetLength(Data,Length(Str)+1);
      Data := Str
    end;
    if Pos('HTTP_COOKIE=',Str) > 0 then begin
      Delete(Str,1,Pos('=',Str));
      SetLength(Cookie,Length(Str)+1);
      Cookie := Str
    end;
    Inc(P, StrLen(P)+1)
  end;
  if RequestMethod = Post then begin
    SetLength(Data,ContentLength+1);
    for i:=1 to ContentLength do read(Data[i]);
    Data[ContentLength+1] := '&';
  end;
  i := 0;
  while i < Length(Data) do begin
    Inc(i);
    if Data[i] = '+' then Data[i] := ' ';
  end;
  if i > 0 then
    Data[i+1] := '&'
  else
    Data := '&';
finalization
  Data := '';
  Cookie := ''
end.
```

➤ *Listing 2: DrBobCGI (revised).*

Set-Cookie

Assuming we use a regular 32-bit version of Delphi (not the WebBroker components, yet), then we need to use a low-level technique to set the value of a new cookie. Fortunately, cookies are set in the HTTP header that a CGI application (or ISAPI DLL) needs to return. The syntax is as follows (the uppercase fields are values that can be specified by the user):

```
Set-Cookie: NAME=VALUE;
  expires=DATE; path=PATH;
  domain=DOMAIN_NAME; secure
```

Both the NAME and the VALUE can be anything set by the user. So, we can have Name=Bob or Answer=42 or 1=2. Note that the NAME=VALUE pair is the only required attribute of a Set-Cookie command.

The DATE in expires=DATE defines the date after which the cookie is invalid (ie after which the cookie will no longer be available). DATE must be formatted as follows:

```
Day, DD-MMM-YYYY HH:MM:SS GMT
```

For example:

```
Mon, 01-Feb-1999 07:11:42 GMT
```

Note that GMT is the only legal time zone, enforcing consistency for international visitors and web servers. This means that we may need to convert our time to the GMT timezone, and format the date according to the above specifications, but that's just about the biggest problem we'll face when using cookies.

The DOMAIN in domain=DOMAIN specifies the internet domain name of the host from which the current URL is fetched. If the domain of the URL is the same as the DOMAIN for a specific cookie in the cookie-list (on disk), then the PATH (in path=PATH) of that cookie is checked as well to see if the cookie indeed should be sent along with the URL fetched from the domain and path as specified.

The default value of DOMAIN is the host name of the server that generated the cookie, and the default value of PATH is a single / character (meaning everything matches). If you leave both of them empty, then a cookie set for any page in your website will be valid (ie sent along with) any other page of your website as well. This may or may not be what you intended, but at least the option is open to you. One warning: the PATH is case-sensitive (I guess the DOMAIN is as well), so be sure to remember that a cookie generated for HOME.HTM will not be sent back to home.htm (if you specified HOME.HTM as a specific path, that is).

Finally, the secure attribute specifies whether or not to use the cookie using a secure channel (ie

using HTTPS, which is HTTP over SSL). If secure is not specified, the cookie is considered harmless, and will be sent in the clear over unsecured channels.

Personally, when I use cookies, I want them to be available for every page of my website, so I usually don't bother with the DOMAIN or PATH attributes, nor do I use the secure attribute, which leaves the NAME=VALUE and expires=DATE attributes only.

If you don't specify a value for expires=DATE, then the cookie will be valid during the lifetime of the session only (ie once you close down the browser, the cookie is gone). So, in order to get a persistent cookie, you must set this expires DATE to a valid future value.

HTTP_COOKIE

Now that we know how to set the value and other attributes of a cookie, it's time to find out how to get (or read) the values from the cookies back from the cookie-file on our user's hard disk.

Fortunately, the hard work (getting the cookies from the cookie-file on disk) is done for us by the web browser, which passes the result in the header of the resulting HTML document (again) using the following syntax:

```
Cookie: NAME=VALUE; NAME=VALUE
```

In our specific case, we would get Cookie: Name=Bob only, but all matching cookie NAME=VALUES will be sent along with the URL we requested. Note that 'matching' here refers to matching DOMAIN and PATH, and only for cookies that are not expired yet.

Reading cookie values is nothing more than parsing the single Cookie: line and obtaining the NAME=VALUE pairs, just like regular CGI GET or POST variables. And, in fact, we'll see that we can use existing code (the DrBobCGI unit) that processes CGI GET and POST variables, and extend it with cookie support.

DrBobCGI

Now that we know how to set cookies (using a Set-Cookie statement) and get cookie values back (the values assigned to the HTTP_COOKIE environment string), it's time to pick up the old DrBobCGI unit and enhance it with cookie support.

Note that this new version of DrBobCGI is also used for the CGI protocol in my *Under Construction* article from Issue 44 (April 1999) about communication between Delphi and Java applications.

Cookies And WebBroker

Using Delphi's WebBroker technology (included in the Client/Server edition, or available to purchase as an add-on), cookies can be set as part of the Response, using the SetCookieField method. Like CGI values, a cookie is of the form NAME=VALUE, so we can put a Name=Bob in there as follows:

```
var
  Cookies: TStringList;
begin
  Cookies :=
    TStringList.Create;
  Cookies.Add('Name=Bob');
  Response.SetCookieField(
    Cookies,'','',Now+1,False);
  Cookies.Free
```

Note that we're using a TStringList to set up a list of cookie values. Each list of cookies can have a Domain (second parameter) and Path (third parameter) associated with it, to indicate which URL the cookie should be sent to. You can leave these blank if you wish. The fourth parameter specifies the expiration date of the cookie, which is set to the day Now+1 in my example, so when the user comes back tomorrow the cookie should have expired. The final parameter specifies if the cookie is used over a secure connection.

Now, assuming the user accepts the cookie, then having set the cookie is still only half the work. In a follow-up OnAction event we need to read the value of the cookie, to determine the value of Name. In this case, cookies are part of the Request class, just like the ContentFields, and they can be queried using the CookieFields property.

```
begin
  Name := StrToInt(
  Request.CookieFields.Values[
  'Name']);
```

Other than that, cookies work just like any CGI content field. Just remember that while a content field is part of your request (and is always up-to-date), a cookie may have been rejected, resulting in a possible older value (which was still on your disk).

Conclusions

So, we have looked at three ways of saving state and seen how easy it is to use one, cookies, in plain Delphi or with WebBroker. All the code is on this month's disk.

Bob Swart (aka Dr.Bob, visit www.drbob42.com) is a technical consultant and webmaster using Delphi, JBuilder and C++Builder for Bolesian and freelance technical author.