A Delphi CGI File Uploader

by Paul Warren

There have been quite a few articles in this magazine dealing with CGI programming. In all of them data was submitted from browser to server via URL encoding. An HTML `<FORM>` element is embedded in a document either with the attribute:

```
ENCTYPE=
  "application/x-www-urlencoded"
```

or with no attribute at all (where `ENCTYPE` defaults to the above). URL encoding will serve most of our needs, but not all. Is there any other way to submit data? As it turns out, there is. Form data can be submitted with the attribute:

```
ENCTYPE="multipart/form-data"
```

There are many new, interesting and potentially valuable things we can do when form data is submitted in this way.

In this first of a pair of articles we'll look at how we can upload files to a web server. In the next article we'll create a useful utility using what we learn here.

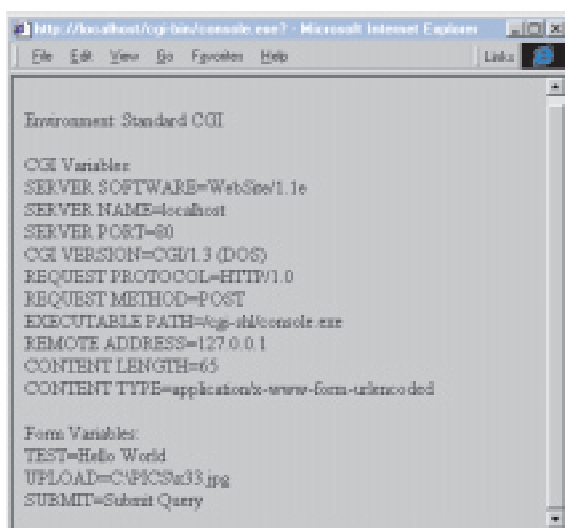
CGI Session Variables

Before we start building the CGI file uploader we'll quickly review CGI programming. In his article *Developing Dynamic Web Pages* in Issue 16, Steve Troxell told us how in standard CGI the 'web server communicates the parameters of the CGI session to the CGI program via environment variables.'

The CGI session parameters constitute a session 'header'. It's the `CONTENT TYPE` parameter of this header that carries the `ENCTYPE` variable we're interested in.

The form data, or user input if you prefer, is submitted through the `QUERY_STRING` environment variable when the form `ACTION` parameter is `GET` and through the standard input when `ACTION` is `POST`. For file uploading only the `POST` method is supported, since most files are clearly too large to pass by an environment variable.

Recognising that the session parameters of the CGI session are communicated to the CGI application the same way regardless of the `ENCTYPE` variable, it's clear that we can extend Steve's `TCGI` component to handle file uploads. All we have to do is examine the `CONTENT TYPE` variable and read the user input according to the result.



➤ Figure 1

RFC1867: Form-Based File Upload In HTML

Once we have determined that data is being submitted as `multipart/form-data` how do we read and decode it? Here we must turn to the relevant internet Request For Comments. A quick search revealed RFC 1867 which defines an experimental protocol for uploading files.

```
Content-type: multipart/form-data, boundary=AaB03x
--AaB03x
content-disposition: form-data; name="yourname"
John Doe
--AaB03x
content-disposition: form-data; name="yourfile"; filename="file1.txt"
Content-Type: text/plain
... contents of file1.txt ...
--AaB03x--
```

➤ Listing 1

At its simplest, RFC 1867 calls for each field of the form to be separated by a boundary not found elsewhere in the data. Between boundaries each part has a header and data separated by a blank line. The header has at least a

```
content-disposition: form-data
```

line with the `INPUT` element name. The data section has either the `VALUE` variable or the binary data. Listing 1 shows an example of a `multipart/form-data` encoded stream.

Notice the boundary is always preceded by '- -' and the final boundary is followed by '- -'. We'll be using this information later.

If you're interested in all the details, I have included RFC 1867 with the code on this month's disk. Note that not all the recommendations were adopted. For instance, RFC 1867 calls for multiple file uploads, which are not implemented on any browser I've tried.

With this brief description of `multipart/form-data` encoding let's get down to business and start creating our file uploader.

Extending TCGI

Steve Troxell's `TCGI` component read all the CGI session variables into a `TStringList` called `CGIItems`. If the form data is submitted as

```
application/x-www-urlencoded
```

we want `TCGI` to behave as Steve originally designed it (Figure 1). Only when data is submitted as `multipart/form-data` should the

```
case EnvironmentType of
  etStdCGI :
    begin
      for I := 0 to NumCGIVars - 1 do
        FCGIItems.Values[CGIVarNames[I]] := GetEnv(CGIVars[I, etStdCGI]);
        // added by Paul Warren 03/99
        if Pos('multipart/form-data', FCGIItems.Values['CONTENT TYPE']) <> 0 then
          begin
          FCGIItems.Values['CONTENT BOUNDARY'] := Copy(FCGIItems.Values[
            'CONTENT TYPE'], Pos('boundary=', FCGIItems.Values[
            'CONTENT TYPE'])+9, Length(FCGIItems.Values['CONTENT TYPE']));
          LoadMultiCGIUserData;
        end else
         // end of addition
         LoadStdCGIUserData;
    end;
  etWinCGI :
    begin
      for I := 0 to NumCGIVars - 1 do
        FCGIItems.Values[CGIVarNames[I]] :=
          WinCGIProfile.ReadString('CGI', CGIVars[I, etWinCGI], '');
      LoadWinCGIUserData;
    end;
end;
```

➤ *Above: Listing 2*

➤ *Below: Listing 3*

```
open standard input
while not Eof do begin
  readln(line)
  if line <> '' then begin
    while true do begin
      if line contains a header set labelstr = name and break
      if line contains a boundary then break
      append valuestr to line
      readln(line)
    end
  end
  if labelstr and valuestr <> ''
    add labelstr=valuestr to FormItems
end
```

```
open standard input
while not Eof do begin
  readln(line)
  if hascontent then begin
    read data until boundary encountered
    set hascontent = false
  end
  if line <> '' then begin
    while true do begin
      if line contains a header set labelstr = name and break
      if line contains content-type set hascontent = true and break
      if line contains a boundary then break
      append line to valuestr
      readln(line)
    end
  end
  if labelstr and valuestr <> ''
    add labelstr=valuestr to FormItems
end
```

➤ *Listing 4*

component decode the multipart stream. Listing 2 shows part of the `TCGI.Create` constructor with the commented modification which checks the encoding type.

I have added a new CGI session variable `CONTENT BOUNDARY` for convenience. We will need to use the boundary variable later in parsing the multipart stream. My routine `LoadMultiCGIUserData` is called only when `CONTENT TYPE` is multipart form-data.

Parsing the multipart stream looks rather simple. Unfortunately it doesn't follow all the conventions of the MIME multipart/message type that it is modelled on. Specifically, the multipart/message type that is used for email with attachments calls for binary data to be UU or Base64 encoded before submission. This way the receiving application can use a `readln` inside a `while not Eof` loop to read in the data. Parsing a stream this way is very easy.

Multipart/form-data can include binary data which obviously excludes using `readln`. But since the rest of the multipart stream is CRLF delimited either `readln` or `Pos` is needed to parse the fields. How do we escape this paradox? The best way to solve the problem is to ignore it, at least for now.

Parsing The Multipart Stream

Steve Troxell parsed the user input into a `TStringList` called `FormItems` in his implementation of `TCGI`. We'll follow the same convention in our extension of the component.

Let's just assume the file being sent is plain ASCII text. In this case the pseudocode in Listing 3 would work fine.

First we read a line from standard input. If the line is blank we simply read another. If it is not blank we enter an infinite loop. Then we test the line to see if it is a header. If it is we set `labelstr` and break. Next we test to see if the line is a boundary and if it is we break. Finally, if neither condition is met we append the line to `valuestr` and read another line. By appending to `valuestr` this way we can handle all the `INPUT` types as well as the `TEXTAREA` element.

Sooner or later we will encounter a header or boundary and break the loop, at which time we add the `labelstr=valuestr` pair to `FormItems`. In the case of a text file the value part of the `label=value` pair will be the file. I suspect there are limitations to the size of file you could upload this way but it does work. It still doesn't help us for binary files though.

If you look again at Listing 1 you'll notice that the message part containing the file has a `Content-Type:` line in the header. If we test for `content-type` in the `while true do` loop we could set a boolean variable to `True` indicating the next content to be read is potentially binary and break the loop. After we read one more line which will be blank we could then process the data differently until we reach the next boundary. Listing 4 shows how this could be done.

We are nearly there. All we need to do is find a replacement for the `while not Eof do` loop and devise a way to read the file contents up to the next boundary.

End Of Message Detection

We saw earlier that the final boundary has two hyphens added to it. This is an excellent way to decide when we have reached the end of the message. When we test

for the presence of a boundary we can also test for the remaining hyphens. In this case we set `Eom` to `true` and break the loop. Now instead of testing for `Eof` we test for `Eom` in the main loop. When `Eom` is `true` we close the standard input and exit.

Unfortunately binary data can contain an end of file marker and even though we are no longer testing for `Eof`, `readln` fails because it thinks it is reading beyond the end of the file. We will need a replacement for `readln` before we can finish.

A Readln Analog

I tried looking at the source for the `readln` function for ideas, but it is written in assembler, which may as well be Greek for me. I do, however, have an old modified Boyer-Moore search algorithm I used for searching binary files looking for various byte sequences. Since the Boyer-Moore algorithm is reputed to be fast and efficient I thought I would try using that in a `readln` replacement.

Listing 5 shows the function `readl1n`. I haven't reproduced the Boyer-Moore search algorithm `SearchBuf` here since it's on the disk and it's not really important to understand at this point.

`SearchBuf` returns the location of the first CRLF pair in `Buffer`. Next, we need to set the length of the variable `Value` to `Result` (plus two for the CRLF pair). Then we move `Result` bytes of `Buffer` to `Value[1]`. Finally we delete `Result` bytes from `Buffer` by moving `Buffer[Result]` to `Buffer[0]`.

You may be wondering where `Buffer` came from. Well, we know we can't use `readln` on the standard input and `Move` doesn't work on file types so we need to

➤ *Listing 5*

```
function readl1n(var Value: string): integer;
begin
  Result := SearchBuf(#13#10, Buffer[0], ContentLength)+2;
  SetLength(Value, Result);
  Move(Buffer[0], Value[1], Result);
  Move(Buffer[Result], Buffer[0], Length(Buffer)-Result);
end;
```

➤ *Listing 6*

```
procedure TCGI.LoadMultiCGIUserData;
{ Reads, parses, and decodes values for the standard CGI
  form variables in a multipart form. }
const
  Eom: boolean = false;
  HasContent: boolean = false;
var
  ContentLength: LongInt;
  InputFCB: File;
  RequestMethod: string;
  S: string;
  LabelStr: String;
  ValueStr: String;
  Buffer: array of char;
  AttachStream: TMemoryStream;
  function readl1n(var Value: string): integer;
  begin
    Result := SearchBuf(#13#10, Buffer[0], ContentLength)+2;
    SetLength(Value, Result);
    Move(Buffer[0], Value[1], Result);
    Move(Buffer[Result], Buffer[0], Length(Buffer)-Result);
  end;
  function readAttachment: integer;
  begin
    Result := SearchBuf(#13#10'--'+CGIItems.Values['CONTENT
      BOUNDARY'], Buffer[0], ContentLength);
    AttachStream.Write(Buffer[0], Result);
    Move(Buffer[Result], Buffer[0], Length(Buffer)-Result);
  end;
begin
  RequestMethod :=
    Uppercase(FCGIItems.Values['REQUEST METHOD']);
  if RequestMethod = 'POST' then begin
    if FCGIItems.Values['CONTENT TYPE'] <> '' then begin
      ContentLength :=
        StrToInt(FCGIItems.Values['CONTENT LENGTH']);
      AssignFile(InputFCB, '');  { standard input }
      Reset(InputFCB, 1);
      try
        SetLength(Buffer, ContentLength);
        BlockRead(InputFCB, Buffer[0], ContentLength);
        while not Eom do begin
          readl1n(S); // read a line
          if HasContent then begin
            // if there is content...
            AttachStream := TMemoryStream.Create;
            try
              // copy to memory stream
              readAttachment;
              // write file to disk
              AttachStream.SaveToFile('c:\temp\'+
                ChangeFileExt(ExtractFileName(
                FFormItems.Values['FILENAME']),'')+
                FloatToStr(TimeStampToMSecs(
                DateTimeToTimeStamp(Time)))+ExtractFileExt(
                FFormItems.Values['FILENAME']));
              // save temp file name as form variable
              FFormItems.Values['TEMPFILE'] :=
                'c:\temp\'+ChangeFileExt(ExtractFileName(
                FFormItems.Values['FILENAME']),'')+
                FloatToStr(TimeStampToMSecs(
                DateTimeToTimeStamp(Time)))+ExtractFileExt(
                FFormItems.Values['FILENAME']);
            finally
              AttachStream.Free;
            end;
            HasContent := false;
          end;
          if S <> #13#10 then begin
            while true do begin
              if Pos('Content-Disposition', S) <> 0 then
                begin
                // delete to first "
                System.Delete(S, 1, Pos('"', S));
                // copy name
                LabelStr := System.Copy(S,1,Pos('"',S)-1);
                // delete name
                System.Delete(S, 1, Pos('"', S));
                if Pos('FILENAME', uppercase(S)) <> 0 then
                begin
                  LabelStr := 'FILENAME';
                  // delete to filename
                  System.Delete(S, 1, Pos('"', S));
                  // copy value
                  ValueStr := System.Copy(S,1,Pos('"',S)-1);
                end;
                Break;
              end;
              if Pos('Content-Type', S) <> 0 then begin
                LabelStr := 'CONTENT-TYPE';
                // delete to :
                System.Delete(S, 1, Pos(':', S)+1);
                // copy name
                ValueStr := System.Copy(S, 1, Length(S));
                HasContent := true;
                Break;
              end;
              if Pos(CGIItems.Values['CONTENT BOUNDARY'], S)
                <> 0 then begin
                // remove first 2 chars
                System.Delete(S, 1, 2);
                // check for Eom
                System.Delete(S, 1, Length(CGIItems.Values[
                  'CONTENT BOUNDARY']));
                if S = '--'#13#10 then Eom := true;
                // lower has content flag if got here
                HasContent := false;
                Break;
              end;
              ValueStr := ValueStr + Copy(S, 1, Pos(#13#10,
                S)-1); // append to valuestr
              readl1n(S); // read another line
            end;
          end;
          if ValueStr <> '' then begin
            FFormItems.Values[LabelStr] := ValueStr;
            LabelStr := '';
            ValueStr := '';
          end;
        end;
      finally
        CloseFile(InputFCB);
      end;
    end;
  end;
end;
```

`BlockRead` the entire stream to a `Buffer`. At last we have reached the final step.

Reading The File

If and when we encounter a file in the multipart stream we need to read it into some kind of buffer and save it to disk. Listing 6 is the full source developed from our pseudocode.

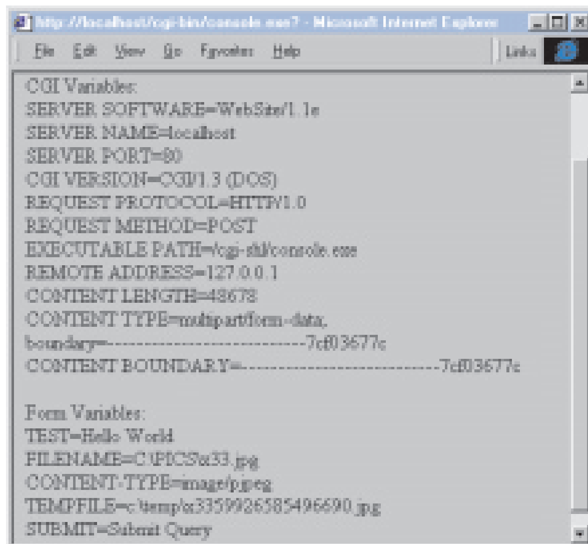
When the variable `HasContent` is `true` we create a `TMemoryStream` and call `ReadAttachment`, which works like `readln` except that it searches for the next boundary instead of the next CRLF pair.

In order to make this extension of `TCGI` as generic as possible we will save the file under a unique file name in a temporary directory. This unique name is stored as a CGI form variable `TEMPFILE` for calling applications to access. The original name is also extracted and saved as the variable `FILENAME` (see Figure 2).

`TCGI` is now capable of working whether a form is submitted with URL encoding or as a multipart stream. Calling applications can access CGI session variables and form variables in the same way for either case. If a file was submitted the caller can access the file, move it, process it or do whatever is required.

Next Time

Parsing multipart form data turned out to be a fair bit more difficult than I thought it would be, but I have already found many uses for file uploads. They are not restricted to the web or intranet either.



➤ *Figure 2*

Next time we'll develop a useful application using the extended `TCGI` component. Running on what I refer to as an *infranet* I'll show you how file uploads can be used to process files on your standalone PC as well as on the web or a corporate intranet.

Postscript

Shortly after writing this article I found out Netscape 3 doesn't follow the rules laid out in RFC 1867. Specifically, when the browser can't determine the content of a file being uploaded the `Content-type:` line should default to `application/octet-stream`. MS Internet Explorer does this correctly. Netscape fails to include the `Content-type:` line at all, causing my code to fail for unregistered file types. Rather than try to accommodate Netscape's aberrant behavior now I will defer a correction until the next article.

Paul Warren runs HomeGrown Software Development in Langley, British Columbia, Canada and can be contacted at hg_soft@uniserve.com