Easy Web Controls

by Paul Warren

Wonderful though they are, Borland's WebBroker components can seem like a step backward in the world of visual programming. I just can't get used to not being able to use most of my components. True, most components are controls designed for the user interface and wouldn't make sense for a web application. But what about data presentation? Many controls are exclusively or partially used for data presentation. In this case it might make sense to be able to drop a visual component on a WebModule form.

Suppose for a moment we *could* drop a visual component on a WebModule. What could this do for us?

Let's say we wanted to create a web application in which a user could access a calendar with his or her monthly appointments highlighted. We start by creating a new web application, drop a PageProducer on the form, write a few lines of HTML including an <IMG SRC=""> tag, and drop a calendar component beside the PageProducer. A few more lines of code in an event and the web application is done.

When the application is accessed it first returns the HTML to the browser then, when the browser requests the image, it sends the image of the calendar. Is this a pipe dream or can we actually do this?

We can, and it's not difficult. So let's start creating web controls!

Wrap It Up

We first need to get around the restriction against using visual components on a web module. Any component you want to drop on a web module must be non-visual, that is, it must descend from TComponent. This means that to use a TControl descendant we have to create a wrapper around it. The wrapper will descend from TComponent and own the control we want to use.

Since we will want to create a number of these control 'wrappers' we should create a base class first. Listing 1 shows the partial declaration of a component that will own a TControl descendant. We'll include Width and Height properties, since we will almost certainly need to set the width and height of the underlying control.

I have also included a ContentAsStream property to provide output to the Response.ContentStream method. This way, when we decide how we want to provide the output, it will be easy to pass to the web browser. For the same reason, there is a ContentType property to write out whatever content type we wish to pass. The reasons for this design decision will become clearer later on; for now, let's see how we can provide useful output from our web controls.

TWinControl.PaintTo

A couple of years ago I discovered the PaintTo method introduced in the TWinControl class. PaintTo takes as parameters a handle to a device context and two coordinates. PaintTo is one of those incredibly useful methods that Borland tucks away in the VCL library here and there, with little or no documentation. If you want a control to copy itself to a bitmap then you need PaintTo.

It works by merging the device context (or canvas) passed for its own canvas temporarily and then forcing a repaint. I have used this method to give some of my components printout capabilities that would otherwise be difficult and tedious.

To see if PaintTo can help develop web controls let's try a little experiment. Place a TImage and a button on a form and in the OnClick event put the code shown in Listing 2. This should copy a dynamically created TStringGrid's Canvas to the image. Unfortunately, if you try to run the code you will get an exception claiming the

```
type
  TWebComponent = class(TComponent)
  private
    FWidth: integer;
    FHeight: integer;
    FControl: TControl;
    FContentType: string;
    function GetContentAsStream: TStream; virtual; abstract;
  protected
  public
    property ContentAsStream: TStream read GetContentAsStream;
    property ContentType: string read FContentType write FContentType;
    property Width: integer read FWidth write SetWidth default 200;
    property Height: integer read FHeight write SetHeight default 150;
  published
  end;
```

➤ Above: Listing 1

➤ Below: Listing 2

```
procedure TForm1.Button1Click(Sender: TObject);
var
  TB: TStringGrid;
  BM: TBitmap;
begin
  TB := TStringGrid.Create(nil);
  try
    BM := TBitmap.Create;
    try
      BM.Width := TB.Width;
      BM.Height := TB.Height;
      BM.Canvas.Lock;
      try
        TB.PaintTo(BM.Canvas.Handle, 0, 0);
      finally
        BM.Canvas.Unlock;
      end;
      Image1.Picture.Bitmap.Assign(BM);
    finally
      BM.Free;
    end;
  finally
    TB.Free;
  end;
end;
```

control (`TStringGrid`) has no parent. This is, of course, a bit of a setback, but we could try creating a dummy form as the parent and see if that works. Listing 3 shows the new code. This time it works fine, though with some overhead.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Dummy: TForm;
  TB: TStringGrid;
  BM: TBitmap;
begin
  Dummy := TForm.Create(application);
  try
    TB := TStringGrid.Create(Dummy);
    try
      TB.Parent := Dummy;
      BM := TBitmap.Create;
      try
        BM.Width := TB.Width;
        BM.Height := TB.Height;
        BM.Canvas.Lock;
        try
          TB.PaintTo(BM.Canvas.Handle,0,0);
        finally
          BM.Canvas.Unlock;
        end;
        Image1.Picture.Bitmap.Assign(BM);
      finally
        BM.Free;
      end;
    finally
      TB.Free;
    end;
  finally
    Dummy.Free;
  end;
end;
```

➤ *Listing 3*

It seems `PaintTo` may help by giving us a bitmap representation of the control. What about non-windowed controls, they don't have a `PaintTo` method? Obviously we'll need to develop a different way of obtaining a visual representation for these controls. Before looking at this, however, let's extend our component class to handle `TWinControl` descendants. Listing 4 shows a component declaration descended from `TWebControl`. To make its purpose clear I have called it `TWebWinControl`.

```
TWebWinControl = class(TWebControl)
private
  function GetContentAsStream:
    TStream; override;
protected
public
published
end;
```

➤ *Listing 4*

The only change here is to override the `GetContentAsStream` abstract method to return the component image as a stream. The code is the same as Listing 3.

All components that descend from `TWebWinControl` will inherit the ability to return their visual representations, so we need no further code. Non-windowed controls, on the other hand, are a little more difficult. Since they don't inherit the `PaintTo` method we will have to descend them from `TWebControl` directly and supply each one with a method to provide ouput.

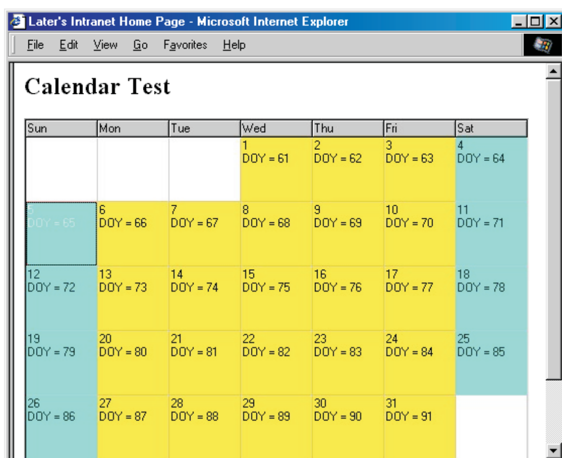
That's enough theory to get us started: let's create a component.

A Web Calendar

We'll start with a windowed control, as it already has the primary functionality. Let's encapsulate my `TRngSelCalendar`, a string grid descendant that implements a calendar with the ability to add strings to the days and select a range of days. Listing 5 shows the declaration and a sample of the methods (the full source is, as always, included on the disk).

Apart from creating and freeing the 'wrapped' `TRngSelCalendar`, the component consists exclusively of property access methods. Notice that all the property access methods do is pass the values through to the true control. This way the developer can set properties and write event code as usual.

We now have a functional semi-visual web control that will give the output in Figure 1 when accessed by a web browser. Before we take the final step to a web control that can truly be called visual, let's see how to handle a non-windowed control.



➤ *Figure 1*

A Rotatable Label

There are really two solutions to handling non-windowed controls. The first is used for third party controls where you may not have the source or the right to modify the source. As an example, we'll use a third party control that can display rotated text. Since the control is third party I can only supply the compiled unit, so you need to be using Delphi 5 for this example.

The first thing we must do is create a descendant of `TXLabel` (the third party component) and supply it with a `PaintTo` method. The code is shown in Listing 6. This is almost the same as `PaintTo` in `TWinControl` (from Controls.pas) except that I have removed all the code that draws borders, etc.

Basically the code merges the supplied device context with the controls canvas using `IntersectClipRect` and forces a repaint using the `Perform` method. It's too bad Borland didn't introduce this method earlier in the VCL hierarchy, since it would have saved us a lot of work. They could have made it virtual and overriden the method in `TWinControl` to give the complete functionality.

Now, if we create a component wrapper descended from `TWebControl`, we can create a web control to display rotated text. The

```
  TWebCalendar = class(TWebWinControl)
  private
    FOnNeedStrings: TNeedStrings;
    function GetBlockWeekends: boolean;
    function GetBlockedColor: TColor;
    function GetFixedHeader: boolean;
    function GetRangeColor: TColor;
    function GetStartDate: TDateTime;
    function GetEndDate: TDateTime;
    function GetCalendarDate: TDateTime;
    procedure SetHeight(Value: integer);
    procedure SetWidth(Value: integer);
    procedure SetBlockWeekends(Value: Boolean);
    procedure SetBlockedColor(Value: TColor);
    procedure SetCalendarDate(Value: TDateTime);
    procedure SetFixedHeader(Value: Boolean);
    procedure SetRangeColor(Value: TColor);
    procedure SetStartDate(Value: TDateTime);
    procedure SetEndDate(Value: TDateTime);
  protected
    procedure DoNeedStrings(Sender: TObject; ACol, ARow:
      Integer; ADate: TDateTime; var Value: TStringList);
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property Width: integer read FWidth
      write SetWidth default 200;
    property Height: integer read FHeight
      write SetHeight default 150;
    property BlockWeekends: Boolean read GetBlockWeekends
      write SetBlockWeekends default false;
    property BlockedColor: TColor read GetBlockedColor
      write SetBlockedColor default clGray;
    property CalendarDate: TDateTime read GetCalendarDate
       write SetCalendarDate stored false;
    property FixedHeader: Boolean read GetFixedHeader
      write SetFixedHeader default True;
    property RangeColor: TColor read GetRangeColor
      write SetRangeColor default clBlue;
    property StartDate: TDateTime read GetStartDate
      write SetStartDate;
    property EndDate: TDateTime read GetEndDate
      write SetEndDate;
    property OnNeedStrings: TNeedStrings read FOnNeedStrings
      write FOnNeedStrings;
    property ContentType;
  end;
constructor TWebCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  // set default properties
  FContentType := 'image/jpeg';
  FWidth := 200;
  FHeight := 150;
  // create underlying calendar
  FControl := TRngSelCalendar.Create(nil);
  // set calendar properties
  FControl.Width := FWidth;
  FControl.Height := FHeight;
  (FControl as TRngSelCalendar).OnNeedStrings :=
     DoNeedStrings;
end;
destructor TWebCalendar.Destroy;
begin
  FControl.Free;
  inherited Destroy;
end;
procedure TWebCalendar.DoNeedStrings(Sender: TObject; ACol,
  ARow: Integer; ADate: TDateTime; var Value: TStringList);
begin
  if Assigned(FOnNeedStrings) then
    FOnNeedStrings(Sender, ACol, ARow, ADate, Value);
end;
procedure TWebCalendar.SetHeight(Value: integer);
begin
  // make sure to change the calendar properties when
  // changing the components properties
  if Value <> FHeight then begin
    FHeight := Value;
    (FControl as TRngSelCalendar).Height := FHeight;
  end;
end;
procedure TWebCalendar.SetWidth(Value: integer);
begin
  // make sure to change the calendar properties when
  // changing the components properties
  if Value <> FWidth then begin
    FWidth := Value;
    (FControl as TRngSelCalendar).Width := FWidth;
  end;
end;
function TWebCalendar.GetBlockWeekends: boolean;
begin
  Result := (FControl as TRngSelCalendar).BlockWeekends;
end;
procedure TWebCalendar.SetBlockWeekends(Value: Boolean);
begin
  (FControl as TRngSelCalendar).BlockWeekends := Value;
end;
```

➤ *Listing 5*

```
type
  TRotatedLabel = class(TXLabel)
  public
    procedure PaintTo(DC: HDC;
      X, Y: Integer);
  end;
procedure TRotatedLabel.PaintTo(
  DC: HDC; X, Y: Integer);
var
  SaveIndex: Integer;
begin
  SaveIndex := SaveDC(DC);
  MoveWindowOrg(DC, X, Y);
  IntersectClipRect(DC, 0, 0,
   Width, Height);
  Perform(WM_ERASEBKGND, DC, 0);
  Perform(WM_PAINT, DC, 0);
  RestoreDC(DC, SaveIndex);
end;
```

➤ *Listing 6*

code is shown in Listing 7. The property access methods are the same as in `TWebCalendar`.

You may have noticed the `Output` property and wondered what that is all about. Don't we already supply the output in a stream?

Really Visual Web Controls

I wanted to be able to use these components the same way we use all our visual components, but that really isn't possible since a web module is not a form and can't own controls. However, there is a way we can sort of beat the system. Since we have a bitmap representation of the control, why not pass it to a property editor? This way we can see the component at design-time. We don't even need to write a property editor, since one already exists for type `TBitmap`.

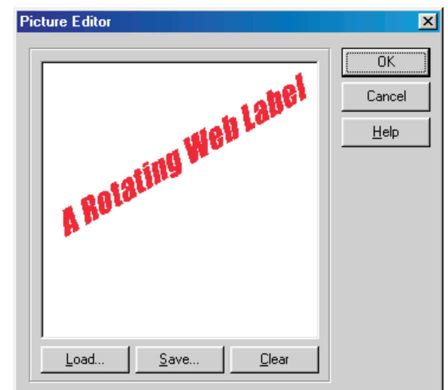
All we need to do is declare a `published` property called `Output` and provide access methods for it. Since we will want all our components to work the same way we will add the property to the ancestor class `TWebControl`. In the unlikely event that we derive a class that doesn't output a graphic representation we'll have to supply a graphic or override the methods to do nothing: a small price to pay.

After installing the `TWebRotatedLabel` component we can set its properties and view the output by clicking the ellipsis next to the output method. The result can be seen in Figure 2. Pretty cool if I do say so myself. Now let's look at the second method of creating non-windowed web controls.

A Web Chart

I created `TXYChart` several years ago to do some limited statistical charting. I didn't need the bells and whistles available in most charting packages and I certainly didn't want to interface with Excel or some other application. I needed to supply printed output, though,

➤ *Figure 2*



```
  TWebRotatedLabel = class(TWebControl)
  private
    function GetContentAsStream: TStream; override;
    { ...see disk for full declaration... }
    function GetOutput: TBitmap; override;
  protected
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    { ...see disk for full declaration... }
  end;
constructor TWebRotatedLabel.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FContentType := 'image/jpeg';
  FWidth := 200;
  FHeight := 150;
  FControl := TRotatedLabel.Create(nil);
  FControl.Width := FWidth;
  FControl.Height := FHeight;
  (FControl as TRotatedLabel).Caption := 'Rotating Label';
end;
destructor TWebRotatedLabel.Destroy;
begin
  FControl.Free;
  inherited Destroy;
end;
function TWebRotatedLabel.GetOutput: TBitmap;
begin
  Result := TBitmap.Create;
  try
    Result.Width := FControl.Width;
    Result.Height := FControl.Height;
    Result.Canvas.Lock;
    try
      (FControl as
TRotatedLabel).PaintTo(Result.Canvas.Handle, 0, 0);
    finally
      Result.Canvas.Unlock;
    end;
  except
    Result.Free;
    raise;
  end;
end;
function TWebRotatedLabel.GetContentAsStream: TStream;
var
  Jpg: TJpegImage;
  S: TMemoryStream;
begin
  Jpg := TJpegImage.Create;
  try
    Jpg.Assign(Output);
    S := TMemoryStream.Create;
    Jpg.SaveToStream(S);
    S.Position := 0;
    Result := S;
  finally
    Jpg.Free;
  end;
end;
```

➤ *Listing 7*

so I took advantage of the PaintTo idea and created my own PaintTo method as described above.

This, then, is the second way to handle non-windowed controls: include a PaintTo method as part of a control that you want to use as a web control. This can, of course, be done in an ancestor class, so all derived controls inherit the ability to reproduce themselves. As a by-product you have an easy way of printing these controls.

I haven't reproduced any of the code for TWebChart here since it is all the same. I have, however, included the source to my charts unit on the disk. I recently added a TPieChart class to the charts unit and you can see the output in Figure 3.

Using Web Controls

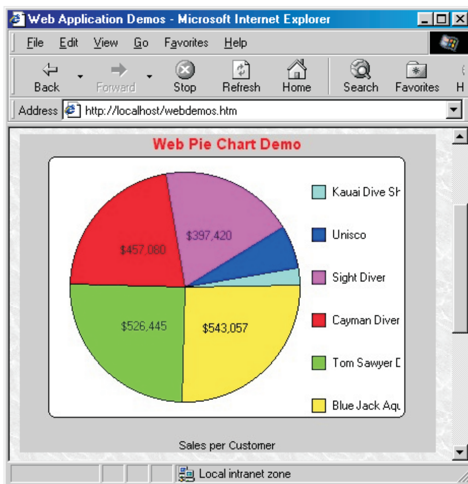
Now let us see if these new web controls can really simplify

```
procedure TWebModule1.WebModule1RootAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := PageProducer1.Content;
end;
procedure TWebModule1.WebModule1BackGroundAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.ContentType := WebRotatedLabel1.ContentType;
  Response.ContentStream := WebRotatedLabel1.ContentAsStream;
end;
procedure TWebModule1.WebModule1ChartAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  i: integer;
begin
  // add some data for demo purposes
  for i := 0 to 9 do WebXYChart1.AddData(i, i*i);
  Response.ContentType := WebXYChart1.ContentType;
  Response.ContentStream := WebXYChart.ContentAsStream;
end;
```

➤ *Above: Listing 8*

➤ *Below: Listing 9*

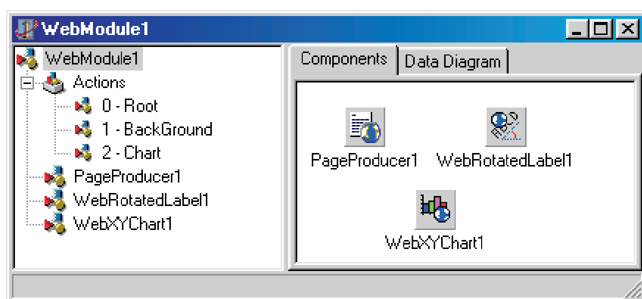
```
<HTML>
<HEAD>
<title>Web Controls Demo</title>
</HEAD>
<BODY Background="http://127.0.0.1/cgi-bin/project1.exe/Background">
<h1>Web Controls Demo</h1>
```



➤ *Figure 3*

application programming with WebBroker. After all, that was the intention from the start. To demonstrate using web controls, open a new CGI web module. Add three Action Items: Root, Background and Chart. Make Root the default and set PathInfo for the other two to /Background and /Chart respectively. In the OnAction events put the code in Listing 8.

➤ *Figure 4*



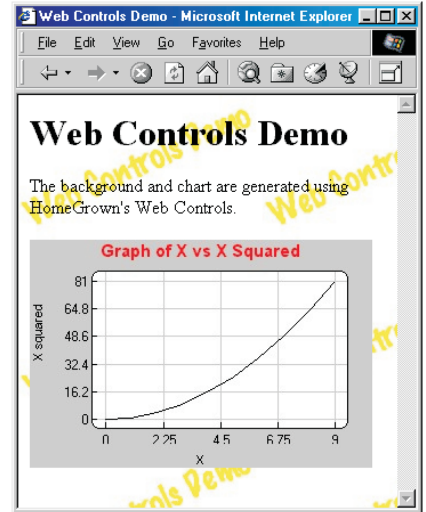
Next, add a TPageProducer to the web module. Put the html shown in Listing 9 in the strings property.

The background and chart are generated using HomeGrown's Web Controls:

```
<P><IMG SRC=
  "http://127.0.0.1/cgi-bin/
  project1.exe/Chart">
</BODY>
</HTML>
```

Now add a TWebRotatedLabel. Set the Caption property to Web Controls Demo. Set Relief to True, Angle to 25, Color to clSilver and set the font to something pleasing. I used 24 point Marking Pen in yellow. To see the result of these settings check the Output property.

Finally, drop a WebXYChart on the form and set the size to 300x200. I used ChartType := ctLine and Grid := gdBoth, but this is just my taste. Again, check the Output property to see the result. You can see the web module in Figure 4. I think you'll agree this is more like the visual application design we are all used to!



➤ *Figure 5*

When you compile the application and access it with a link to

```
<A HREF="http://127.0.0.1/
  cgi-bin/project1.exe">
```

you will see the page in Figure 5. Please note that the URLs shown above and in the listings refer to a local web server: you will need to make sure the URLs reflect your own server for this demo to run.

Conclusions

We now have three examples of visual web controls at work. More importantly, we have a class framework that makes using most visual components possible and fairly simple. Although many controls make little sense in a web application, enough of them are useful that I think visual components are an improvement over the 'out of the box' WebBroker.

While little user interaction is available, the possibilities for data presentation are only limited by our imaginations. There may be other types of data such as text, sound and multimedia that can be handled using web components and html methods. If I find new and useful techniques I'll be back to share them with you.

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