Lazarus for the web

Michaël Van Canneyt

February 6, 2006

Abstract

In this article, the support for web programming in Free Pascal/Lazarus is explored. The support is divided in several parts: HTML, Templates, Sessions, HTTP, CGI. Each of these parts is explained, as well as the way in which they cooperate.

1 Introduction

Web programming, and more specifically CGI programming can be done in many languages, and many ways. Obviously, CGI programming can be done in Pascal, but can it also be done in a RAD way, using a GUI IDE ? An environment such as Intraweb is not yet available for Lazarus or FPC. However, the basics to create such an environment have been laid down, as will be shown in this article.

The things described here have been developed by members of the FPC team. There exist other components like PSP (Pascal Server Pages) which also allow to develop CGI applications in FPC/Lazarus with little effort. However, the RAD aspect of PSP is largely lacking, and integration with existing technologies in FCL/Lazarus is not one of the goals of PSP. The components presented here have as an explicit design goal that they should integrate with existing components in FPC.

The components to be presented here have been designed with extensibility in mind: they are not supposed to be the end point of development, rather they are the building stones with which a solid web programming environments should be built. Web programming is a broad topic, so there are a lot of components to be covered.

The following section will attempt to explain the various building blocks and how they are currently connected. All units are in the weblaz package, which can be simply installed in the Lazarus IDE.

2 Functional Architecture

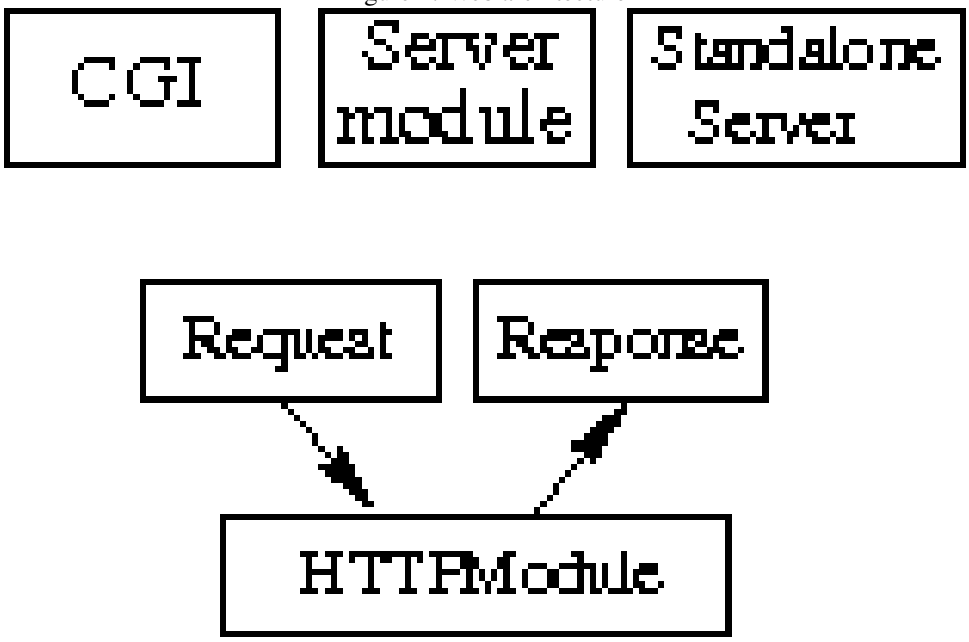
The building blocks for web programming are divided in several parts:

HTTP The various aspects of the HTTP protocol are put in this part: classes to represent a HTTP request, and a corresponding HTTP response.

HTML An implementation of a HTML DOM model: this can be used to construct HTML 4.1 compliant documents. A special writer object exists to create valid HTML in a way which hides the DOM behind it.

Sessions an abstract session definition is defined, plus the interaction with the HTTP protocol classes is defined.

Figure 1: Web architecture



Templates Complementary to the HTML DOM model, a template system was developed, allowing to separate application logic from layout. A Small extension to the HTML DOM model was implemented to be able to mix templates with the DOM objects.

HTTPModules These `TDataModule` descendents allow to program web applications in the Lazarus IDE. There are 2 variants: one for HTML-only, one for more general-purpose requests (such as sending an image or other non-html file).

TContentProducers component classes which can be used on `HTTP` modules to produce valid HTML in a number of ways, from a number of sources.

CGI Is a set of units which handles all aspects of a CGI application. Basically, it sets up the HTTP request and response objects, and makes sure the response is sent back to the webserver. There is no offical RFC for CGI implementations, but the units implement most of the unofficial CGI RFC, plus some apache extensions (wincgi is also planned).

Currently not implemented are units to be able to write Apache loadable Modules, or modules for IIS. Implementation of these units is planned. Also planned is a HTTP server, implemented in Object pascal, which could use modules to handle requests.

The architecture is depicted schematically in figure 1 on page 2. When a client requests a URL that must be handled by the web components, the enclosing application (CGI, Apache Module, Stand-alone server) prepares a pair of request and response instances. Then, it determines what module should handle the request (more about this later). If need be, an instance is created, and the request and response variables are passed to it. When the module has handled the request, the response is sent back to the client.

The `HTTPDatamodule` class is central in this design: it handles the request. In fact, the class has only one method: `HandleRequest`.

To handle the request, various descendents of this datamodule have been created, as well as a number of components that make producing HTML from datasets (or just in code) eas-

ier, likewise auxiliary components for session management and handling HTML templates have been provided.

3 Unit structure

The functionality outlined above is distributed along various units, each unit taking a part of the functionality. They are outlined below:

HttpDefs This is the basic unit for HTTP management. It contains the `TRequest` and `TResponse` classes. These contain properties for all headers defined in the HTTP 1.1 protocol, plus some auxiliary classes for cookies and uploaded files. The base session definition is also in this unit. The response and request classes are nearly completely self-contained, although many methods can be overridden.

HtmlDefs contains an enumerated type containing all valid HTML 4.1 tags, plus an array describing the various attributes of each of these tags. The HTML entities are also described, and and some functionality to transform strange characters to HTML entities is provided.

HtmlElements contains a series of HTML DOM elements: for each HTML tag, a DOM element is defined which has as properties all valid attributes for this tag.

HtmlWriter is a component which allows to produce HTML in a procedural way, mimicing what would be done in e.g. PHP. It constructs a HTML document as defined in the htmlelements unit, which can then be streamed. This allows to write structurally correct HTML. It is used by a lot of the HTML content producers.

fpHttp contains the basic `TCustomHTTPModule` definition, as well as a factory for registering and creating `TCustomHTTPModule` instances. It also contains basic definitions for HTTP actions.

WebSession contains a session-aware descendent of `TCustomHTTPModule`: `TSessionHTTPModule`, and has functionality to create and maintain sessions using `.ini` files and cookies.

fpTemplate Template handling support.

fpWeb Contains a version of a general purpose HTML module which can be used in the Lazarus IDE. It's a descendent of the `TSessionHTTPModule` class.

custcgi Contains descendent of the general `TCustomApplication` class, called `TCustomCGIApplication`, which handles the CGI protocol: it prepares a `TRequest` variable and a `TResponse` variable, which it then passes to an abstract `HandleRequest` method. It has no knowledge of `TCustomHTTPModule` at all. In principle, using this class, a CGI application can be built.

fpCGI contains a descendent of the `TCustomCGIApplication` class which will use the HTTP Module factory to create an `TCustomHTTPModule` instance, depending on the URL with which it was invoked. This is the application class as it is used in the Lazarus IDE.

fpHTML contains various HTML content producers, as well as a `TCustomHTTPModule` descendent which can only produce HTML content: it cannot be used to send image or other non-html files. This module can be created in the Lazarus IDE.

In the following sections, the base objects will be discussed, and also how they can be used.

4 Request and Response

The httpdefs unit contains the basic classes used for web programming. These classes describe the client's (browser's) request, and the server's response to the request. Both response and request consist of 2 parts, namely the headers and the data. Both classes descend from a common ancestor which contains most headers. Similar classes exist in the Delphi websnap package: the names have been kept different, so as to allow future development of descendents of the FPC classes which match the interface of Delphi's classes.

The following main classes are defined in the httpdefs unit:

THTTPHeader The common ancestor of `TRequest` and `TResponse`. For all variables allowed in the HTTP headers, a property with the same name exists. It contains methods for loading all variables from a stream or from a stringlist.

TRequest A descendent of `THTTPHeader` which has properties for cookies, POST or GET method query variables, and properties for handling uploaded files.

TResponse A descendent of `THTTPHeader` which has properties for cookies, and functionality to send the response to the client.

TCustomSession an abstract class which represents a web session: it simply introduces the possibility of storing variables across sessions. This is an abstract class, from which a descendent must be created.

Other than these main classes, the following auxiliary classes exist:

TUploadedFile a `TCollectionItem` descendant representing an uploaded file.

TUploadedFiles a `TCollection` descendant representing the uploaded files. The `Files` property of the `TRequest` class is a `TUploadedFiles` instance.

TCookie a `TCollectionItem` descendant representing a cookie which can be sent to the client. The various fields (expires, secure) for a cookie are implemented as the properties of this collectionitem.

TCookies a `TCollection` descendant representing cookies sent to the client. The `Cookies` property of the `TResponse` class is a `TCookies` instance.

The `TRequest` and `TResponse` classes provide storage for all headers, and can be used as-is. The instantiating routines only need to fill in all properties before passing the `TRequest` on.

To create a response, the most important properties of the `TResponse` class are the following:

```
Property Code: Integer;
Property CodeText: String;
Property ContentStream : TStream;
Property Content : String;
property Contents : TStrings;
property Cookies: TCookies;
Property HeadersSent : Boolean;
Property ContentSent : Boolean;
```

The meaning of these properties should be obvious:

Code This is the HTTP returncode which should be sent back to the client browser. After the HTTP headers have been sent, this property should no longer be changed. Standard, this is a 200 response code (all OK)

CodeText Is the Code in plain text. After the HTTP headers have been sent, this property should no longer be changed.

Content the content to be sent to the browser, as a string. Setting this will also set the content of the `Contents` property.

Contents the content to be sent to the browser, as a stringlist. Setting this will also set the `Content` property.

Contentstream a stream whose content should be sent to the client. If this property is set, then both `Content` and `Contents` will be ignored; The content of the stream will be sent as-is to the client. This can be used to send images to the browser. The stream instance will be freed of when the request instance is destroyed.

Cookies The collection of cookies to be sent back. Changing the cookies after the HTTP headers have been sent has no effect, as the cookie definitions are sent in the HTTP headers.

HeadersSent is `True` if the headers have already been (partially) sent to the client browser.

ContentSent is `True` if the content has already been (partially) sent to the client browser.

Other than these properties, the usual HTTP header properties can be set. Notably, the `ContentType` property should be set. By default it is set to `'text/html'`, so if something else than HTML is to be sent to the client, this property should be set correctly.

to send the headers, the `SendHeaders` method can be used. To send the content, the `SendContents` method can be used: if the headers have not yet been sent they will be sent first.

5 The `TCustomCGIApplication` class

The custcgi unit contains a descendent of the `TCustomApplication` class, called `TCustomCGIApplication`. This class overrides the various methods of `TCustomApplication` to handle the parsing of the HTTP variables: It instantiates 2 descendants of `TRequest` and `TResponse`, and invokes an abstract `HandleRequest` method, passing these instances to the method. After the request was handled, the result is sent back to the server. It also overrides the `HandleException` method, and transforms an exception to a suitable error page.

The main methods in this class are, first of all, the `Initialize` method:

```
Procedure TCustomCGIApplication.Initialize;

begin
  StopOnException:=True;
  Inherited;
  FRequest:=TCGIRequest.CreateCGI(Self);
  InitRequestVars;
  FOutput:=TIOStream.Create(iosOutput);
  FResponse:=TCGIResponse.CreateCGI(Self,Self.FOutput);
end;
```

As can be seen, a `TCGIRequest` and `TCGIResponse` instance are created. The `InitRequestVars` method handles the parsing of the GET or POST variables. Setting `StopOnException` to `True` makes sure that the `Run` method stops after an exception was caught. (for GUI programs, this is generally set to `False`)

The `DoRun` method is very simple:

```
procedure TCustomCGIApplication.DoRun;
begin
  HandleRequest(FRequest,FResponse);
  If Not FResponse.ContentSent then
    begin
    FResponse.SendContent;
    end;
  Terminate;
end;
```

It calls the abstract `HandleRequest` method. after the request was handled, the content is sent to the webserver, if it was not yet sent.

Actually, this is enough to start programming CGI programs. All that needs to be done in Lazarus is to create a descendent of `TCustomCGIApplication`.

If the WebLaz package is installed in the Lazarus IDE, it is sufficient to choose 'Custom CGI Application' from the **'File–New'** dialog. Lazarus then creates a new project file with the following code in it:

```
program dumpcgi;

{$mode objfpc}{$H+}

uses
  Classes,SysUtils,httpDefs,custcgi,cgiApp;

Type
  TCGIApp = Class(TCustomCGIApplication)
  Public
    Procedure HandleRequest(ARequest : Trequest; AResponse : TResponse); override
  end;

Procedure TCGIApp.HandleRequest(ARequest : Trequest; AResponse : TResponse);

begin
  // Your code here
end;

begin
  With TCGIApp.Create(Nil) do
    try
      Initialize;
      Run;
    finally
      Free;
    end;
end.
```

All that needs to be done is to insert the code in `HandleRequest`, and to use the

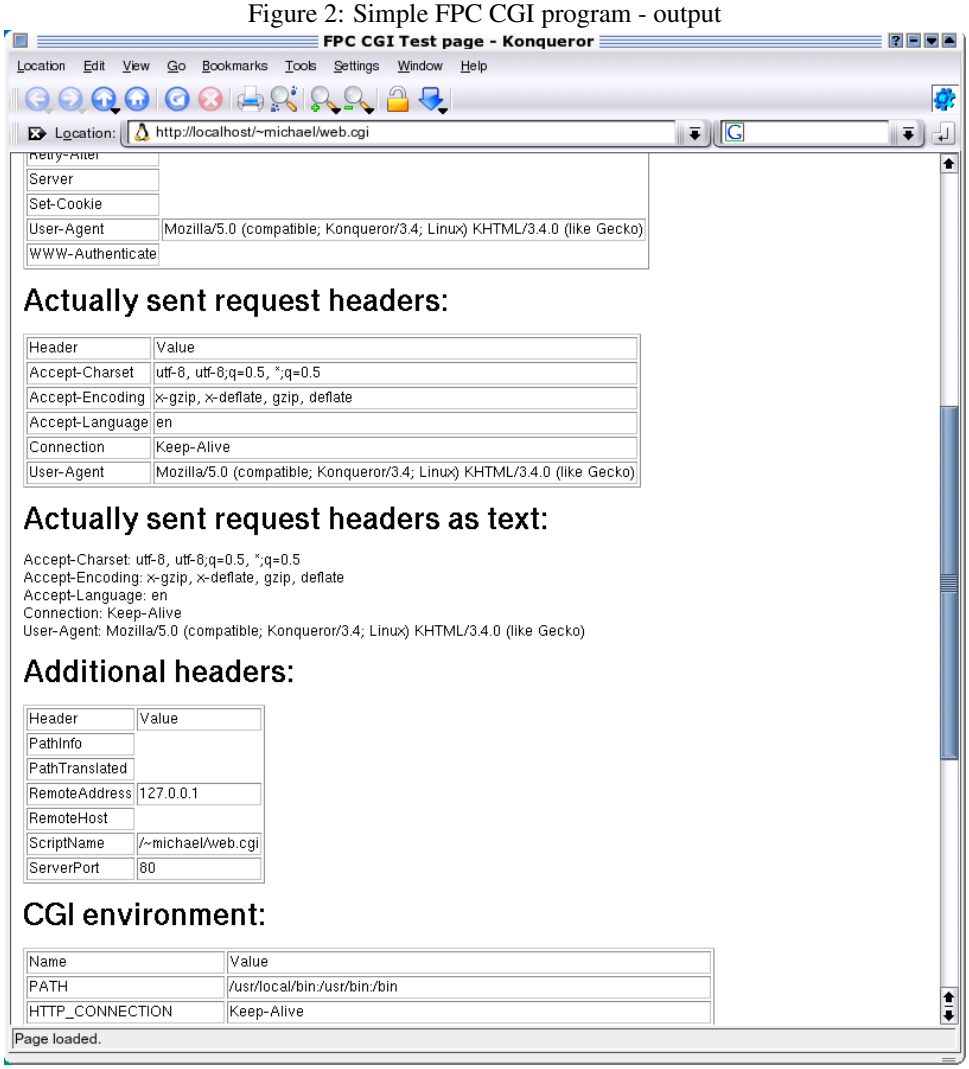
ARequest and AResponse instances to create a web page.

The webutil unit contains a method DumpRequest, which can be used to create a HTML page which shows the contents of the request instance:

```
procedure TMyWeb.HandleRequest(ARequest: TRequest;
                               AResponse: TResponse);

  Procedure AddNV(Const N,V : String);

  begin
    AResponse.Contents.Add('<TR><TD>'+N+
                           '</TD><TD>'+V+
                           '</TD></TR>');
  end;

Var
  I,P : Integer;
  N,V : String;

begin
  With AResponse.Contents do
    begin
    BeginUpdate;
    Try
      Add('<HTML><TITLE>FPC CGI Test page</TITLE><BODY>');
      DumpRequest(ARequest,AResponse.Contents);
      Add('<H1>CGI environment:</H1>');
      Add('<TABLE BORDER="1">');
      Add('<TR><TD>Name</TD><TD>Value</TD></TR>');
      For I:=1 to GetEnvironmentVariableCount do
        begin
        V:=GetEnvironmentString(i);
        P:=Pos('=',V);
        N:=Copy(V,1,P-1);
        system.Delete(V,1,P);
        AddNV(N,V);
        end;
      Add('</TABLE>');
      Add('</BODY></HTML>');
    Finally
      EndUpdate;
    end;
    end;
end;
```

The TRequest.Contents is filled with a HTML page that contains the dump of the request, and then the environment variables passed to the CGI script, are dumped in a table. The output of this simple CGI program is shown in figure figure 2 on page 8 Very simple and small CGI programs can be made like this. It would be very easy to make a file upload or download program with techniques like this: the tedious details of handling the request variables and sending the response are taken care of, so the programmer can concentrate on web logic and creating content. There is little overhead, since the Lazarus streaming system is not included, and there is no definition of a TCustomHTTPModule is not included either.

Figure 2: Simple FPC CGI program - output



However, more is needed, and this is where the fphttp unit comes in.

6 Web modules and Web Actions

To create a web system where more functionality is included in 1 binary, and to enable a more RAD approach to web programming, the system of `TCustomHTTPModule` is set up.

The `fphttp` unit contains the definition of `TCustomHTTPModule`:

```
TCustomHTTPModule = Class(TDataModule)
Procedure HandleRequest(ARequest: TRequest;
                        AResponse: TResponse);virtual;abstract;
end;
```

It also contains the following overloaded procedures:

```
Procedure RegisterHTTPModule
    (ModuleClass : TCustomHTTPModuleClass);
Procedure RegisterHTTPModule
  (Const ModuleName : String;
   ModuleClass : TCustomHTTPModuleClass);
```

These routines register a module in the Module Factory, using `TModuleFactory`:

```
TModuleFactory = Class
  Function FindModule(AModuleName : String) : TModuleItem;
  Function ModuleByName(AModuleName : String) : TModuleItem;
  Function IndexOfModule(AModuleName : String) : Integer;
  Property Modules [Index : Integer]: TModuleItem;
end;
```

This factory keeps a list of registered modules, and has some methods to look up modules. The module definitions are stored in a `TModuleItem` class:

```
Property ModuleClass : TCustomHTTPModuleClass;
Property ModuleName : String;
```

The purpose of this factory is to be able to create a module on-demand, depending on the URL asked by the client. This is best illustrated by an example. Supposing a CGI application mycgi.cgi is located somewhere on the webserver, and that the URL for the CGI is as follows:

```
http://www.mylocation.org/cgi-bin/mycgi.bin
```

Then the following URL could be used to select a particular `TCustomHTTPModuleClass`:

```
http://www.mylocation.org/cgi-bin/mycgi.bin/mymodule
```

If the CGI application detects this URL, then it should use the module factory to locate module `mymodule` and let this module handle the request. For example, the FPC bug-tracker and contributed units could be implemented in 2 different modules, and united in one CGI application, and selecting one or the other could be done by entering the correct URL:

```
http://www.freepascal.org/cgi-bin/fpc.cgi/bugs
http://www.freepascal.org/cgi-bin/fpc.cgi/contrib
```

if the modules were used in an apache loadable module to handle a certain location, then they could be reached by e.g. the following URLs:

```
http://www.freepascal.org/fpc/bugs
http://www.freepascal.org/fpc/contrib
```

(Note that support for Apache modules is not yet implemented, it's only in the planning stage)

This principle of automatically 'dissecting' an URL to execute a certain action can be extended by introducing webactions (much as in Delphi's websnap). A webaction is a particular action to be undertaken when a given URL needs to be handled by a module.

The `TCustomWebActions` class is a collection of `TCustomWebAction`:

```
TCustomWebActions = Class(TCollection)
  Function ActionByName(AName : String) : TCustomWebAction;
  Function FindAction(AName : String): TCustomWebAction;
  Function IndexOfAction(AName : String) : Integer;
  Property OnGetAction : TGetActionEvent;
  Property Actions[Index : Integer] : TCustomWebAction;
end;
```

The action itself is defined as follows:

```
TCustomWebAction = Class(TCollectionItem)
  Property Name : String;
  Property ContentProducer : THTTPContentProducer;
  Property Default : Boolean;
  Property BeforeRequest : TRequestEvent;
  Property AfterResponse : TResponseEvent;
end;
```

The `Name` property is used to identify the action. The `Default` property specifies that this is the action that should be executed if no action could be determined from the URL. The `ContentProducer` property allows to couple a `THTTPContentProducer` to the action. This component (which will be explained later) will then handle the request further.

Modules that descend from `TCustomHTTPModuleClass` have a property of type `TCustomWebActions` which allows to handle various actions in the same module: actions can be created in the Lazarus Object Inspector, and events can be associated with them to create HTML content.

The advantage of this system is that various actions which are similar can be executed in the same `TCustomHTTPModuleClass`, without having to code a system to differentiate between various actions.

To illustrate the above, the idea of the bugtracker can be used again. The following URLs would invoke different actions in the module for the bugtracker:

```
http://www.freepascal.org/cgi-bin/fpc.cgi/bugs/browse
http://www.freepascal.org/cgi-bin/fpc.cgi/bugs/show?id=52
http://www.freepascal.org/cgi-bin/fpc.cgi/bugs/edit?id=53
http://www.freepascal.org/cgi-bin/fpc.cgi/bugs/delete?id=54
```

The first URL would invoke the bugs module, and invoke the `browse` action, which would show a list of bugs. The second URL, would invoke the `show` action, which could show 1 record from the bugs, namely the bug with ID 52. The third URL would invoke the `edit` action for record 52. This could either emit an edit form or apply edits. Lastly, the last entry could delete ID 54.

The following section shows how to use this system in Lazarus, using a `TFPWebModule`.

7 Modules in Lazarus

In Lazarus, the support for web modules is split out in 2 modules. The first is `TFPWebModule`. It publishes following properties:

```
Property Actions : TFPWebActions;
Property ActionVar : String;
Property BeforeRequest : TRequestEvent;
Property OnRequest : TWebActionEvent;
Property AfterResponse : TResponseEvent;
Property OnGetAction : TGetActionEvent;
Property Template : TFPTemplate;
Property OnGetParam : TGetParamEvent;
Property OnTemplateContent : TGetParamEvent;
```

These properties have the following meaning:

Actions This is the collection with the various actions for this module. The actions are of type `TFPWebAction`.

ActionVar The content of this request variable will be used to determine which action should be executed, if the URL does not specify an action. By default the value is 'Action'. This means that the following URL's are equivalent, they will execute the same action:

```
http://www.freepascal.org/cgi-bin/fpc.cgi/bugs/browse
http://www.freepascal.org/cgi-bin/fpc.cgi/bugs/?Action=browse
```

BeforeRequest This event is executed by the module before the request is handled.

AfterResponse This event is executed by the module after the request was handled.

OnGetAction This event can be set to influence the choice of the action: it should return the name of an action to be executed.

Template This property specifies a template which will be parsed and used as the basis for the output, replacing template variables as it goes along. If this property is specified, then the actions are bypassed.

OnGetParam This event is executed when a template variable's value needs to be retrieved.

OnTemplateContent This event is executed for the special template variable `'Content'`.

The various actions in the `Actions` property of `TFPWebModule` are of type `TFPWebAction`, a descendent of `TFPCustomWebaction`. They have the following additional properties:

```
Property Content : String;
Property Contents : TStrings;
Property OnRequest: TWebActionEvent;
Property Template : TFPTemplate;
```

The meaning of these properties is again quite simple:

Content Can be set to a string which will be sent back as the HTML page. Can be used for instance for a 'See you again' kind of page when someone e.g. logs out.

Contents same as `Content`, only as a stringlist.

Template same as `Content` or `Contents`, but this is treated as a template to be parsed.

OnRequest this event can be used to handle the request if this action is invoked.

To demonstrate all this, a simple CGI application can be created. To do this, in the **'File-New'** dialog, the 'CGI application' item can be used. A cgi project will be created, and a FPWebModule will be created automatically, which will be renamed as `TCookieModule`. In the 'OnRequest' event handler of the `TCookieModule`, the following code can be placed:

```
procedure TCookieModule.ShowCookies(Sender: TObject;
  ARequest: TRequest; AResponse: TResponse; var Handled: Boolean);

Var
  C : TCookie;

begin
  With AResponse.Contents do
    begin
    Add('<HTML><TITLE>Cookie test</TITLE><BODY>');
    Add('<H1>Cookie test page</H1>');
    if (ARequest.Cookie='') then
      begin
      Add('Your browser did not offer a cookie for this site.');
      Add('A cookie named FPCWebCookie has been offered to your'
      Add('browser, please accept it.');
      C:=AResponse.Cookies.Add;
      C.Name:='FPCWebCookie';
      C.Value:='FPCRulez';
      Add('<P>After accepting the cookie, reload this page');
      end
    else
      begin
      Add('Your browser offered a cookie for this site:<BR>');
      Add('<PRE>');
      Add(ARequest.Cookie);
      Add('</PRE>');
      end;
    Add('</BODY></HTML>')
    end;
  Handled:=True;
end;
```

The routine checks whether a cookie `FPCWebCookie` was offered by the browser. If it was not, then a cookie with this name is added to the response. The code for this is self-explanatory. If a cookie was offered, the content of the cookie is shown.

In the initialization code of the unit, the register call which was automatically inserted by Lazarus, is changed to:

```
initialization
  {$I wmcookie.lrs}
  RegisterHTTPModule('cookie', TCookieModule);
end.
```

Placing this CGI application (named web.cgi) in a location where the webserver finds it, e.g. the user's html directory, the cookie module is executed with the following URL:

```
http://localhost/~michael/web.cgi/cookie
```

The first time this is executed, the browser will be offered a cookie. The second time, the value of the cookie will be displayed.

To show the execution of an action, and the meaning of a default action, a new module is created. Two actions are created: One is called 'info', and it's `Default` property is set to 'True'. The other is called 'info2'. The `Actionvar` property of the main module is set to `'Do'`. Both actions get their 'Contents' property filled with some simple HTML code, enough to distinguish which action was execution.

The following URL's will execute the first ('info') action:

```
http://localhost/~michael/web.cgi/tmainmodule/info
http://localhost/~michael/web.cgi/tmainmodule?do=info
http://localhost/~michael/web.cgi/tmainmodule
```

The first one is a URL which points straight to the first action. The second uses the `ActionVar` property of the module. The last one uses the `Defafult` property of the first module. To execute the second action, the following URLs can be used:

```
http://localhost/~michael/web.cgi/tmainmodule/info2
http://localhost/~michael/web.cgi/tmainmodule?do=info2
```

Obviously, the content produced here is not very exciting. The idea of the above it to illustrate the ideas underpinning the modules and actions.

8 Conclusion

In this first article, the basics of the web support in FPC/Lazarus have been put forward. This is just the surface of what is possible, and only the basics of the web system have been shown : Further possibilities will be explained in subsequent articles: Sessions, use of templates, use of HTML producing components. It should be noted that the web components of FPC/Lazarus are a recent development, and are still under development. The planning for the near future is to implement support for Apache modules and an embedded HTTP server, allowing to write stand-alone webservers. After that, more elaborate support for webservices and webapplications should be implemented.