Creating a simple webserver in Lazarus

Michaël Van Canneyt

August 27, 2011

Abstract

Free Pascal - and therefor Lazarus - can compile many TCP/IP stacks: Synapse, Indy and lnet. However, Free Pascal also ships with some simple networking components. One of them is a simple webserver. This article shows how to use it.

1 Introduction

There are times when one needs to add a small webserver to an application. For instance, if it is necessary to add a web-interface to configure and manage an application - this is especially useful when writing service applications (daemons).

There are several TCP/IP networking component suites available: Indy and Synapse are the most well-known, especially for people coming from the Delphi environment. L Net was written especially for FPC, and uses an event-driven, non-blocking, approach. Free Pascal includes since a long time some simple TCP/IP classes, which have been used recently to implement a set of components that implement the server and client side for the HTTP 1.1 protocol.

Combined with the existing fcl-web components, this can be used to create a full-fledged HTTP server with a standard installation of Free Pascal and Lazarus, which can serve files, implement a web application or host a web service for WST (The Web Services Toolkit as explained in the Lazarus book).

The component is kept architecturally simple, so it may not be suitable to build high-load and scalable web servers. But for simple web services or embedded use, it is perfectly suitable.

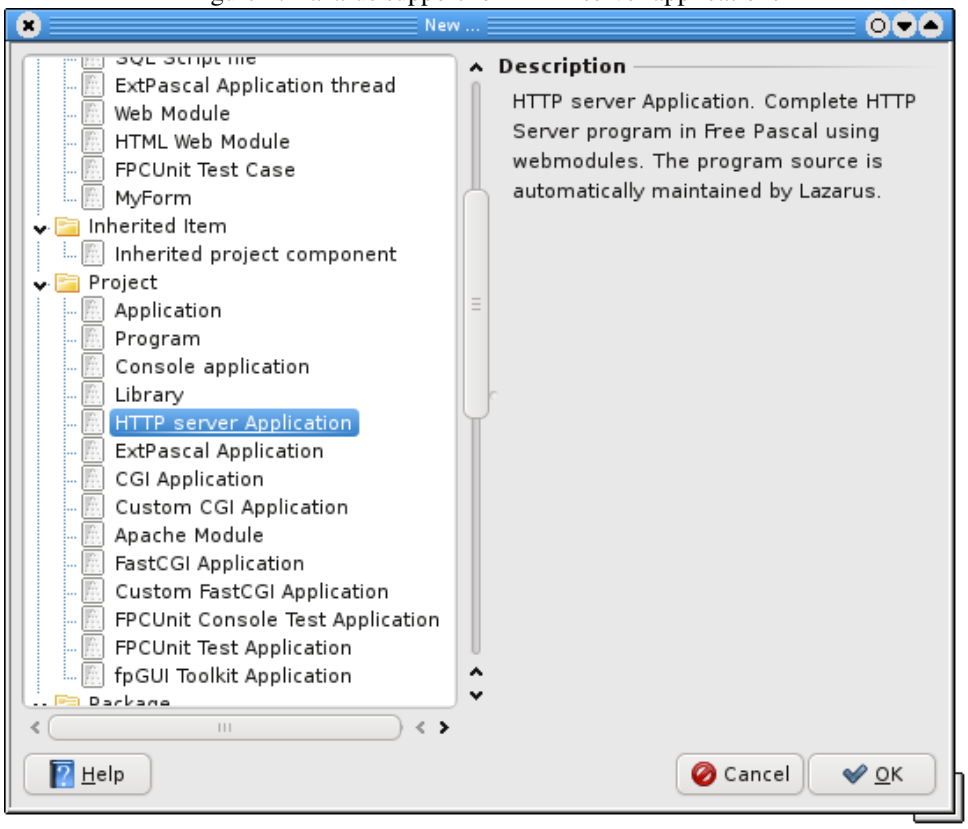
Basically, there are 2 components available:

TFPHTTPServer This is the component that does the actual work. It offers some properties to configure the server. The most important ones are the TCP/IP port number to listen to (obviously named `Port`), and the `Active` property, which starts the server. It also offers an event to actually handle requests, with the name `OnRequest` (what else?). When embedding a HTTP server in an application, this is the component one should use.

THTTPApplication This is a `TCustomApplication` descendant: it uses a `THTTPServer` component to listen to requests, and then hands the requests over to the FCL-Web request dispatcher mechanism. This component should be used when creating a web service hosting application. It offers the same properties as the `THTTPServer` component, to configure the behavior of the server.

In fact, the latter component is integrated in the development version of Lazarus: in the 'File - New' dialog, one can choose 'HTTP server application', as shown in figure 1 on page 2. (the LazWebExtra package must be installed for this)

Figure 1: Lazarus support for HTTP server applications



2 A simple webserver

The simplest webserver can be implemented with very few lines of code:

```
program myserver;

uses
  SysUtils, fphttpapp, fpwebfile;

Const
  MyPort = 8080;
  MyDocumentRoot = '/home/michael/public_html';
  MyMiMeFile = '/etc/mime.types';

begin
  RegisterFileLocation('files',MyDocumentRoot);
  MimeTypesFile:=MyMimeFile;
  Application.Initialize;
  Application.Port:=MyPort;
  Application.Title:='My HTTP Server';
  Application.Run;
end.
```

In fact, this source code is not very different from what Lazarus generates if a 'HTTP Server application' project is started.

Note the unit fpwebfile in the uses clause. This unit contains a complete and ready-to-use FCL-Web module (`TFPCustomFileModule`) that serves files. It is not necessary to create an instance of this module explicitly. fcl-web will take care of this. All that needs to be done, is tell fcl-web which location should be mapped to an actual directory: this is the purpose of the first line of code:

```
  RegisterFileLocation('files',MyDocumentRoot);
```

This tells fcl-web that a location as

```
http://localhost/files/myfile.html
```

Must be translated to a file on disk with the name

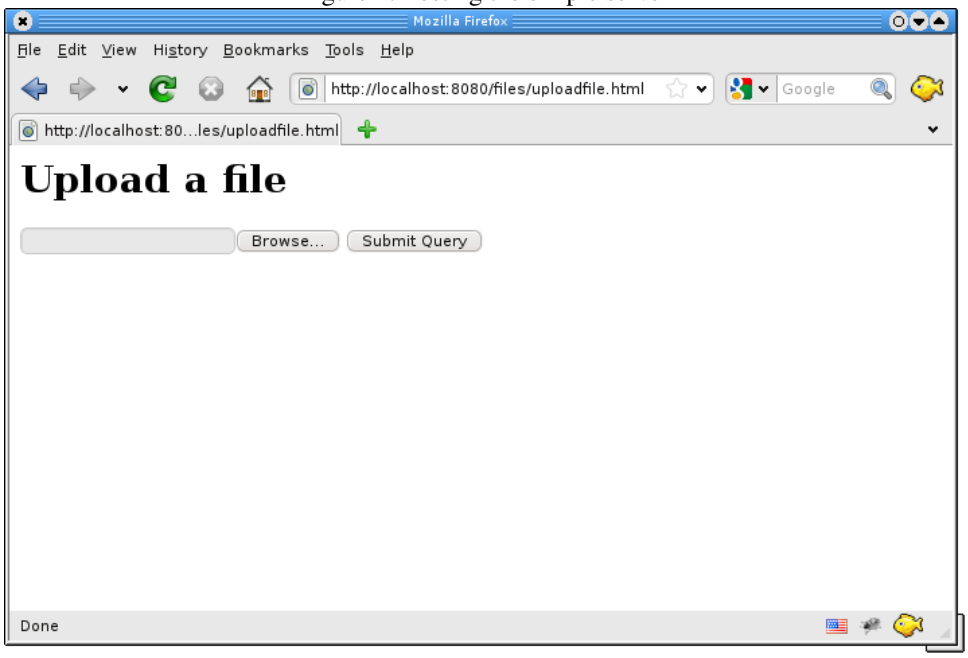
```
/home/michael/public_html/myfile.html
```

This is much like the 'Alias' directive in the Apache webserver configuration. Obviously, the actual directory should exist on the computer where the program is run. It is possible to register several locations.

The second line of code tells the file serving module where the mime.types file is. This file (installed by default on all Unix systems) is used to map file extensions to mime-types when setting the `Content-Type` HTTP header. This HTTP header tells the browser what kind of file it is getting from the server - so it can decide what to do with it. On windows, such a file must be provided to the program. A typical Windows Apache installation contains such a file, it can be used for the `TFPCustomFileModule` as well.

If the above program is started, the browser can be used to display files that are in the registered file locations, as can be seen in figure 2 on page 4.

Figure 2: Testing the simple server



3 Embedding a webserver

The above example is very simple, but is also limited. Other than serve web-requests, it cannot do anything else at all. Using the `THTTPServer` component, however, it is possible to embed webserver functionality in an existing application. This is not harder than the previous example. To show this, a GUI application can be built with a Memo component, and 2 buttons. One button to start the server (let's call it `Bstart`), one to stop the server (call it `BStop`. The memo will be used to log the requests (let it be called `MLog`).

The Application will simple serve files, so a `TFPCustomFileModule` instance is needed to actually serve files. This instance can be created and configured in the `OnCreate` event of the main form:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  RegisterFileLocation('files','/home/michael/public_html');
  MimeTypesFile:='/etc/mime.types';
  FHandler:=TFPCustomFileModule.CreateNew(Self);
  FHandler.BaseURL:='files/';
end;
```

The first lines of this code are the same as in the first example. The last 2 lines create the module and tell it from what location it should serve files. Note the use of the `CreateNew` constructor: this prevents the Run-Time Library from trying to find a form file for the module. Since there isn't any, the use of `Create` would raise an exception.

If everything was simple, the Start button would create an instance of `TFPHTTPServer` and set `Active` to `True`:

```
procedure TMainForm.BStartClick(Sender: TObject);
begin
```

```
  MLog.Lines.Add('Starting server');
  FServer:=TFPHTTPServer.Create(Self);
  FServer.Port:=8080;
  FServer.OnRequest:=@DoHandleRequest;
  FServer.Active:=True;
end;
```

Unfortunately, this would freeze the application as soon as the start button was pressed: the line

```
  FServer.Active:=True;
```

does not return until the server stops. This would mean that the GUI becomes unresponsive, which is not very desirable.

Instead, the server should run in it's own thread, and the Start button should create such a thread. The following thread implementation will run the HTTP server:

```
THTTPServerThread = Class(TThread)
Private
  FServer : TFPHTTPServer;
Public
  Constructor Create(APort : Word;
         Const OnRequest : THTTPServerRequestHandler);
  Procedure Execute; override;
  Procedure DoTerminate; override;
  Property Server : TFPHTTPServer Read FServer;
end;
```

The constructor creates the `TFPHTTPServer` instance and sets the `Port` property and assigns the `onRequest` handler:

```
constructor THTTPServerThread.Create(APort: Word;
   const OnRequest: THTTPServerRequestHandler);
begin
  FServer:=TFPHTTPServer.Create(Nil);
  FServer.Port:=APort;
  FServer.OnRequest:=OnRequest;
  Inherited Create(False);
end;
```

The execute methods simply sets `Active`, and waits till it returns. After that, it frees the server instance.

```
procedure THTTPServerThread.Execute;
begin
  try
    FServer.Active:=True;
  Finally
    FreeAndNil(FServer);
  end;
end;
```

When the thread is terminated (by an external thread), it should stop the server:

```
procedure THTTPServerThread.DoTerminate;
begin
  inherited DoTerminate;
  FServer.Active:=False;
end;
```

Using this thread, the `OnClick` handler of the start button is simple:

```
procedure TMainForm.BStartClick(Sender: TObject);
begin
  MLog.Lines.Add('Starting server');
  FServer:=THTTPServerThread.Create(8080,@DoHandleRequest);
end;
```

Note that it passes a form method: `DoHandleRequest` to handle the requests. Care must be taken: This event handler will be called in the context of the server thread.

Likewise, the the `OnClick` handler of the stop button just stops the thread:

```
procedure TMainForm.BStopClick(Sender: TObject);
begin
  MLog.Lines.Add('Stopping server');
  FServer.Terminate;
end;
```

All that needs to be done is to implement the `DoHandleRequest` method:

```
procedure TMainForm.DoHandleRequest(Sender: TObject;
  var ARequest: TFPHTTPConnectionRequest;
  var AResponse: TFPHTTPConnectionResponse);
begin
  FURL:=Arequest.URL;
  FServer.Synchronize(@ShowURL);
  FHandler.HandleRequest(ARequest,AResponse);
end;
```

The method saves the URL, and then calls `ShowURL`. Because the `DoHandleRequest` is called by the thread, and it must update the GUI, it is necessary to protect it with `Synchronize`, which will make sure that `ShowURL` is called in the context of the main (GUI) thread. The `ShowURL` method is extremely simple:

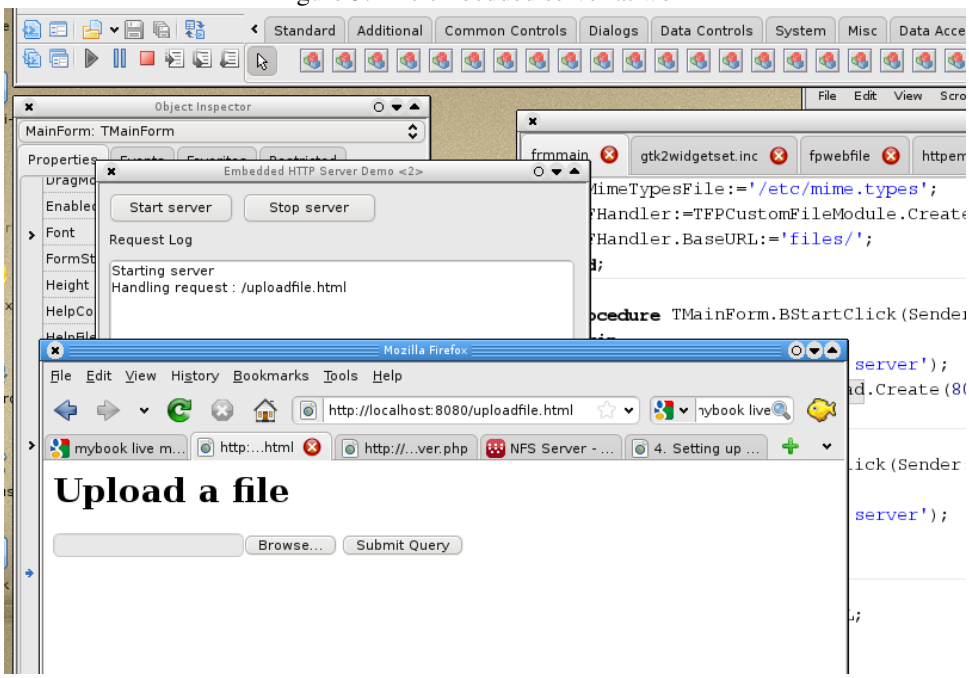
```
procedure TMainForm.ShowURL;

begin
  MLog.Lines.Add('Handling request : '+FURL);
end;
```

With this, the demonstration application is finished. It can now be run, and pressing the 'Start' button will start the HTTP server. What happens if an URL is entered in the browser, is shown in figure 3 on page 7: the page is shown in the browser, and a log entry is shown on the form.

Note that the URL which is entered, does not contain the location 'files' as in the first demo. The reason is that the application does not need to decide which FCL-Web module is needed to handle the request: all requests are handled by the file serving module. When this module was created in the `OnCreate` handler of the main form, the line

Figure 3: The embedded server at work




```
FHandler.BaseURL:='files/';
```

told the module that it should look in the registered location called 'files'.

4 conclusion

In this article, the use of the standard FPC functionality for the HTTP protocol was demonstrated: with a few lines of code, one can easily create an application that can act as a webserver. Not all aspects of the available functionality have been covered: the client side (getting files) has not yet been covered. Support for Web services (using WST) has also not been treated. This will be left for a future contribution.