Free Spider Web development for Free Pascal/Lazarus

user's manual

FreeSpider Version 1.3.3
Modified: 13.Apr.2013
Author: Motaz Abdel Azeem
Home Page : http://code-sd.com/freespider

Introduction:

Free Spider is a web development package for Lazarus. You can create any web application in Linux, Windows, Mac or any other platform that already supported by FreePascal/Lazarus. Lazarus produces independent native executable files and libraries.
With *FreeSpider* you can produce CGI web applications and Apache Module web applications.

Produced FreeSpider web application is a compiled code, it is not like scripting language, no need for parsing and compilation at run-time on every request, for that reason its response is very fast.

How to create Free Spider web applications

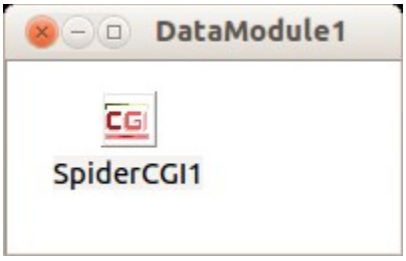
There are two types of web applications if FreeSpider: ***CGI*** and ***Apache module***:

1. CGI Web application

To create Free Spider CGI web application follow these steps:

1. In Lazarus IDE main menu select *File/New..* select *FreeSpider CGI Web Application*

2. Put *TSpiderCGI* component in the data module in *main.pas* unit , select it from *FreeSpider* page



3. Double click on *Data Module* or select *OnCreate* event and write this code on it:

```
SpiderCGI1.Execute;
```

4. Double click on *SpiderCGI1* component or select *OnRequest* event on that component and write this code on it:

```
Response.ContentType:= 'text/html; charset=UTF-8';
Response.Add('Hello world');
```

5. Change application **output directory** to your web application *cgi-bin* directory, such as */usr/lib/cgi-bin* in Linux. You can change this settings by clicking *Project menu/Project Options/Application Tab/Output settings group/Target file name*, then you can write your application name after *cgi-bin* path. Make sure *cgi-bin* directory is writable.

6. Suppose that your project name is ***first***, then you can call it from your browser like this in Linux:

```
http://localhost/cgi-bin/first
```

In windows it should be:

```
http://localhost/cgi-bin/first.exe
```

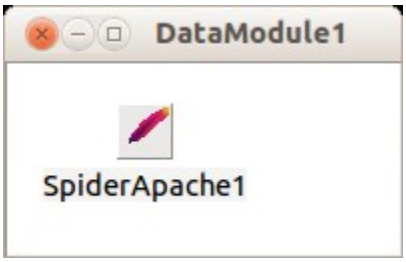
7. If you get error in browser, make sure that cgi-bin is properly configured and you have this option in it's configuration:

```
Options ExecCGI
```

2. Apache Module Web application

To create FreeSpider *Apache Module* Web application, follow these steps:

1. In Lazarus IDE main menu select *File/New..* select *FreeSpider Apache Module Web Application*
2. Put *TSpiderApache* component in the data module in ***main.pas*** unit, select it from *FreeSpider* page



3. Double click on ***SpiderApache1*** component or select *OnRequest* event on that component and write this code on it:

```
Response.ContentType:= 'text/html; charset=UTF-8';
Response.Add('Hello world');
```

4. Save your project. Change it's default name from ***mod_proj1*** to ***mod_first*** for example.
5. Open project source (***mod_first***) file, you will find these constants:

```
const

  MODULE_NAME = 'mod_proj1.so';
  MODNAME = 'apache_mod1';
  HANDLER_NAME = 'proj1-handler';
```

Change them to these settings:

```
const

  MODULE_NAME = 'mod_first.so';
  MODNAME = 'apache_first';
  HANDLER_NAME = 'first-handler';
```

Hanlder_Name value should be unique in your sever

6. make sure that your web module class name is matching this procedure in project source code:

```
function DefaultHandler(r: Prequest_rec): Integer; cdecl;
begin
   Result:= ProcessHandler(r, TDataModule1, MODULE_NAME, HANDLER_NAME, False);
end;
```

In our case it is ***TDataModule1***, if you want to change *Data Module* name, then change it back in this procedure.

7. At projects options/Compiler Options, add FreeSpider directory in *other unit files*

At Code generation uncheck *Relocatable* option (*-WR*) if it is checked.

Remove this line if you find it in project's source code:

```
  Application.Title:='Spider Apache Module';
```

This line is automatically added when you modify project options.

8. Compile your project and copy produced library into Apache module directory, e.g. in Ubuntu:

```
 sudo cp libmod_first.so /usr/lib/apache2/modules/mod_first.so
```

In Windows copy ***mod_first.dll*** library file to any directory or leave it in it's original project folder.

9. Open /etc/apache2/apache2.conf file and add this configuration (Ubunto):

```
LoadModule apache_first /usr/lib/apache2/modules/mod_first.so

<Location /first>
     SetHandler first-handler
</Location>
```

In Windows:

```
LoadModule apache_first c:\projects\firstapache\mod_first.dll

<Location /first>
     SetHandler first-handler
</Location>
```

10. Restart Apache web server. e.g. in Ubuntu:

```
sudo /etc/init.d/apache2 restart
```

11. In your browser type this URL:

```
http://localhost/first
```

Notes:

At each time when you have modified your Apache module web application you need to deploy it again and restart Apache. In Windows you should stop Apache before copying the new library, then start it again after that.

It is better to start your development of any *FreeSpider* web application as ***CGI***, because it is easier to deploy, re-deploy, and test. You don't need restart Apache in every new version of your web application.

When you need to deploy your web application in its final production server, it is better to produce Apache Module library of your web application. Apache Module Library can handle a lot of concurrent requests, and uses less memory and CPU than CGI version. In the next topics you will find how to maintain two versions of your web application: ***CGI*** and ***Apache Module*** at the same time.

Apache Module is very sensitive and it work inside Apache memory, so that if you miss-configured it, it could prevent Apache service from start. CGI is working outside web server memory so that it is more safer in low traffic web applications.

Thread pooling is a feature which makes your database-web application response more faster. This feature can be used only with Apache Module, you can read about it later in this manual.

CGI web application can work in many standard web servers like: Apache, NGNIX and IIS, while Apache Module is designed only for Apache web server.

If you have 32 bit of Apache web server, then you should compile Apache Module using 32 bit Lazarus, and 64 bit Lazarus for 64 bit Apache web server.

Request object:

Request Object of *TSpiderCGI / TSpiderApache* **OnRequest** event contains user request information, such as query fields that has been called with web application's URL like this:

CGI application:

```
http://localhost/cgi-bin/first?name=Mohammed&address=Khartoum
```

Apache module application:

```
http://localhost/first?name=Mohammed&address=Khartoum
```

Then you can access this query information like this:

```
  UserName:= Request.Query('name');
  Address:= Request.Query('address');
```

Also you can access it from Query's String List:

```
  UserName:= Request.QueryFields.values['name'];
  Address:= Request.QueryFields.values['address'];
```

In *Query* you will find the data that has been sent in the URL or submitted in a from fields using GET method.

If you are using POST method in HTML form, you can read it using *Form* method in Request object:

```
  Login:= Request.Form('login');
  Password:= Request.Form('password');
```

Also you can use *ContentFields* string list to access all posted data:

```
  Login:= ContentFields.values['login'];
  Password:= ContentFields.values['password'];
```

Request object also contains some user's browser information like *UserAgent*, and *RemoteAddress* of the client and ***WebServerSoftware*** information.

Response object:

Response object of *TSpiderCGI' / TSpiderApache OnRequest* event represents the HTML response that will be shown in user's browser after requesting the URL.

You can add HTML text response using *Add* method. You can use *Add* method many times to build the

whole HTML page. *Add* method accumulates HTML text in a string list, then at the end of *OnRequest* event all these HTML that has been added will be displayed in the browser at once.

Example:

```
  Response.Add('Hello world<br>');
  Response.Add('This has been written in <b>Object Pascal</b>');
```

TSpiderAction component

TSpiderCGI is needed for any CGI application in Free Spider web application, and TspiderApache is needed for any FreeSpider's Apache module web application, but both components can handle only one request/response (*action*) which has been requested from browser e.g:

```
http://localhost/cgi-bin/first
or
http://localhost/first
```

TSpiderAction components can handle more additional request/response actions, for example suppose that you have a web application which has many HTML forms, each form represent different request, for example an e-mail system needs these request/response actions: *register user, login, logout, view Inbox, send e-mail*, etc. You can call these requests like this:

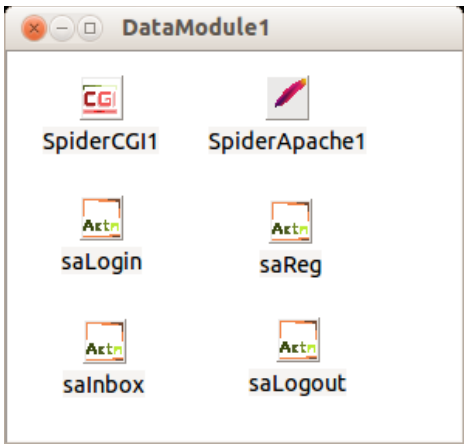
```
http://localhost/cgi-bin/mail/login

http://localhost/cgi-bin/mail/logout

http://localhost/cgi-bin/mail/reg

http://localhost/cgi-bin/mail/inbox
```

/*login*, /*logout*, /*reg*, and /*inbox* parts are called *Paths*. Each *TSpiderAction* component can handle one Path. In this case you will need 4 *TSpiderAction* components.



TSpiderAction's *Request* and *Response* objects are the same like TSpiderCGI / TSpiderApache Request/Response objects.

OnRequest event of *TSpiderAction* is typical to *TSpiderCGI's* / TSpiderApache *OnRequest* event. The main difference is that *TSpiderCGI / TSpiderApache OnRequest* is called when no path information, only CGI executable name or web module name, while *TSpiderAction's OnRequest* is called when the user requests a URL that contains path after web application name, for example: */cgi-bin/mail/inbox*

In every request, only one action will be triggered and its code will be executed.

TSpiderTable component

TSpiderTable component generates HTML table, either manually or from a dataset.

This is an example of manually populated HTML table:

```
SpiderTable1.SetHeader(['ID', 'Name', 'Telephone Number']);
SpiderTable1.AddRow('', ['1', 'Mohammed', '01223311']);
SpiderTable1.AddRow('#FFEEDD', ['2', 'Ahmed', '01754341']);
SpiderTable1.AddRow('#FFDDEE', ['3', 'Omer', '045667890']);
Response.Add(SpiderTable1.Contents);
```

This will generate a table in a browser like this:

ID	Name	Telephone Number
1	Mohammed	01223311
2	Ahmed	01754341
3	Omer	045667890

Note that *SetHeader* method overrides *ColumnsCount* property. You can leave *ColumnsCount* property unchanged and let SpiderTable set it from the number of passed columns parameters. In the previous example *ColumnsCount* will be set to 3 after calling *SetHeader* method.

Other mode for *TSpiderTable* is by linking it to *DataSet*. When you set a *DataSet* property either at design time or at run time you will get a HTML table from that *DataSet. e.g:*

```
SpiderTable1.DataSet:= ATable;
ATable.Open;
```

```
Response.Add(SpiderTable1.Contents);
```

TSpiderTable component contains two events:

OnDrawHeader: This event will be triggered when drawing Table's header
OnDrawDataCell: This event will be triggered when drawing each data cell

Both events contains *CellData* and *BgColor* properties for being drawn cell. Web Developer can change them according to specific data values.

TSpiderForm

TSpiderForm component generates HTML form to be used for entering data to be submitted to a web action.
TSpiderForm important properties are:

1. *Method*: by default it is POST. Post method can transmit more data than Get method and data that are being sent is not displayed in the URL. GET method sends data field values in the URL like in our previous examples.

2. *Action*: represents receiver web action that handles this form data. It's format is: (*server/alias/web application/path).* Example: */cgi-bin/mail/login*

3. *ExtraParam*: contains additional form properties such as Java Script code, or encode type like file upload *multi part* type which will talk about later in uploading files section.

4. *PutInTable*: by default it is *True*. If it is set to *true*, it puts table fields and labels (*AddText*) in a table of two columns: first column contains the text that has been added by *AddText*, and second column is the field that has been added by *AddInput*. If you call *AddInput* without *AddText*; then this input will be displayed in the first column, the same like button on the example below.

Example of generating form:

```
SpiderForm1.Clear;
SpiderForm1.Action:= Request.RootURI + '/reg';
SpiderForm1.AddText('Enter Subscriber ID: ');
SpiderForm1.AddInput(itText, 'id', '');

SpiderForm1.AddText('Enter Subscriber Name: ');
SpiderForm1.AddInput(itText, 'subname', '');
```

```
  SpiderForm1.AddText('Address: ');
  SpiderForm1.AddInput(itText, 'address', '');

  SpiderForm1.AddText('Telephone Number: ');
  SpiderForm1.AddInput(itText, 'telephone', '');

  SpiderForm1.AddInput(itSubmit, 'add', 'Add');

  Response.Add(SpiderForm1.Contents);
```

This will generate a form in user's browser like this one:

Enter Subscriber ID:

Enter Subscriber Name:

Address:

Telephone Number:

Add

Note:

If you put form's actions at design time like: */cgi-bin/mail/reg* this will make your web application not portable across operating systems and web technologies, e.g:

/cgi-bin/mail/reg will work for Linux only

/cgi-bin/mail.exe/reg will work for windows only

/mail/reg will work with Apache Module only

To prevent this dependency you should use ***Request.RootURI*** property at run time :

```
  SpiderForm1.Action:= Request.RootURI + '/reg';
```

RootURI will contains current web application name/alias . It is a good practice to make your source code a platform and a technology independent.

You can set the same name for different inputs, like check box, for example suppose that you have many groups, and you need to let the user to select a group or many groups:

```
SpiderForm1.AddText('Group 1');
SpiderForm1.AddInput(itCheckbox, 'group', '1');
SpiderForm1.AddText('Group 2');
SpiderForm1.AddInput(itCheckbox, 'group', '2');
SpiderForm1.AddText('Group 3');
SpiderForm1.AddInput(itCheckbox, 'group', '3');
```

In this case you should use *Group ID* as value of check box, and you need to read it like this:

```
for i:= 0 to Request.ContentFields.Count - 1 do
  if Request.ContentNames(i) = 'group' then
  begin
     Response.Add('<br/>You have selected group # : ' + Request.ContentValues(i));
  end;
```

TSpiderPage

TSpiderPage component enables web developer/designer to design HTML pages using any HTML editor such as Office Writer. The designer should embed tags inside HTML to enable dynamic contents to be shown. Tags can be like this *@tag;*

You can replace these tags later in *TSpiderPage's OnTag* event.
TSpiderPage enables the developer to separate web application user interface design from compiled code. If there are any change in the design are later needed, the developer may not need to recompile the web application, instead he/she needs to change only external HTML pages.
HTML pages can be put in the same directory with CGI application, for example *cgi-bin* directory.

ExtraParams

You will find *ExtraParams* property in many FreeSpider objects, such as *Form* and *Table*. On this property you can put any additional HTML property such as css class name or java script code:

```
SpiderTable2.TableExtraParams:= 'class=”myStyle2”';
```

Please refer to SpiderSample for a complete example of using CSS in FreeSpider.

HTML Tags

You can use some of ready HTML tags in *Response* object instead of writing them manually. For example if you need to display line break in HTML you could do it like;

```
	Response.Add('<br/>');
```

Or using this simple method:

```
	Response.NewLine;
```

Also you can display many lines, for example 10:

```
	Response.NewLine(10);
```

To put a hyper link you can use this ready method:

```
	Response.AddHyperLink('http://code.sd', 'My Site');
```

Also you can display a table without using TspiderTable:

```
	Response.NewTable;
	Response.NewTableRow;
	Response.PutTableData('First Cell');
	Response.PutTableData('Second Cell');
	Response.CloseTableRow;
	Response.CloseTable;
```

This is an example of a paragraph tag (P):

```
	Response.AddParagraph('This is my paragraph');
```

There are many other HTML tags in *Response* object. More tags can be added later in newer version of FreeSpider package.

CGI life cycle

CGI applications has a very short life cycle. The application will be uploaded into server memory and executed starting from user's request (click/refresh, enter URL, etc) until being served (until user gets result on browser). This time normally can be less than 1 seconds according to simplicity of action's event handler (*OnRequest*). For that reason Free Pascal/Lazarus CGI web applications is easy to develop, no memory leaks will occur, the developer shouldn't be worry about freeing resources, because these resources will be reserved for a short time and freed automatically when this CGI application finished execution. These resources can be memory allocations, files, sockets, etc. This behavior of CGI application grantees more reliability and more uptime for the web application.

In addition to it's short cycle, Free Pascal CGI application run in it's own memory space, not inside web server's memory, for that reason a buggy CGI application will not affect web server (for example Apache web server). Deployment or replacing new version of a Free Spider web application in the web server does not require restarting that web server.

User Session

User session in Free Spider web application is handled by using cookies. Cookies are hidden data that web application can store in users browsers private data to let the browser send it in the next requests as a definition to web application of current user and session. The behavior of stateless life cycle makes it difficult to web applications to link between *first, second, third* … requests. For that reason we need to put some data in each web browser to keep session tracking. This methodology is handled by *Response's SetCookie* method. This is an example of setting cookies in user browser :

```
  Response.SetCookie('sessionid', '2', '/');
```

Cookies path parameter represents the scope of this cookies. If it is "/" that means all web applications in this server can access this value. Some times you need to have a large web application system which falls into many projects and then executables, these executables can serve the same user using only one login by using cookies and the global path "/".

Previous cookies will be erased when the user closes the browser.
If you want to set more/less expiration time for cookies you can provide an *expiration* parameter like:

```
  Response.SetCookie('sessionid', '2', '/', Now + 1);
```

In this case, cookies will last for one day. You can set it to hours, or minutes, but at first you should know your time zone in GMT. For example if your country time zone is (GMT + 3), then you should deduct 3 hours from current time, after that you can set expiration time:

```
    Response.SetCookie('sessionid', '2', '/', Now -
      EncodeTime(3, 0, 0, 0) + EncodeTime(0, 5, 0, 0));
```

This will make the *sessionid* cookie field last for 5 minutes

Cookies can be read using *Request's GetCookie* method like this:

```
  Response.Add(Request.GetCookie('sessionid'));
```

File upload/Download

To upload files into web application, you need to change HTML form's encode type to *multipart/form-data* using *TSpiderForm's ExtraParam* property. You can put this value at design time or at run time:

```
  SpiderForm1.ExtraParam:= 'enctype="multipart/form-data"';
```

Example:

```
    Response.Add('<h2>File Upload sample</h2>');
    SpiderForm1.AddInput(itFile, 'upload');
    SpiderForm1.AddInput(itSubmit, 'upload', 'Upload');
    Response.Add(sfUpload.Contents);
```

Then you can receive this file using *Request's ContentFiles* property, then save it in a web server directory, put it in a database or send it back to the user as this example:

```
    Response.ContentType:= Request.ContentFiles[0].ContentType;
    Response.CustomHeader.Add('Content-Disposition: filename="' +
      Request.ContentFiles[0].FileName + '"');
    Response.Content.Add(Request.ContentFiles[0].FileContent);
```

Smart Module Loading Design

You can create a web application that contains many Data Modules, each data Module can contain Free Spider *Actions*, *Pages*, *Tables*, and *Forms* components, and their required datasets and any other objects.
The idea of *Smart module loading* helps developers to splits the design and development of large web applications into smaller pieces. Also it minimizes the loading of unnecessary components in memory,

only one module will be loaded per request.

The main module that contains *SpiderCGI* / ***SpiderApache*** will be created regardless of requested path, plus additional data module that contains this SpiderAction/path will be created too. That means if you have a web application that has 100 different paths, you can put them in 10 modules for example, a maximum of two modules will be loaded into memory each time the user requests a page or action.

This method will reduce memory consumption because only required objects will be created, and will make the response more faster because creation objects in memory takes time.
To make a *smart module loading* design follow these steps:

1. Create new *Free Spider* web application the same as above examples
2. Add *New* Data Module
3. Call *RegisterClass* procedure to register this new data module class. Suppose that this new data module class is named *TdmMod2*, then you should write this code in Data Module 2 Initialization section:

```
initialization
  {$I mod2.lrs}
  RegisterClass(TdmMod2);
```

4. Add this unit name in main Data Module *uses* clause
5. Put Free Spider *TSpiderAction* components and define a new path
6. At main Data Module *OnCreate* event write this code:

```
  SpiderCGI1.AddDataModule('TdmMod2', ['/path2', '/path3']);
  SpiderCGI1.Execute;
```

It if it Apache module, then write this in your main Data Module OnCreate event:

```
  SpiderApache1.AddDataModule('TdmMod2', ['/path2', '/path3']);
```

If you have two components (SpiderApache and SpiderCGI) in the same web application then write:

```
  SpiderCGI1.AddDataModule('TdmMod2', ['/path2', '/path3']);
  SpiderApache1.AddDataModule('TdmMod2', ['/path2', '/path3']);
  SpiderCGI1.Execute;
```

Paths parameter should contain all the paths that exist in this new Data Module, if you forget to add any path, then FreeSpider will not find it.

You can find this method in *SpiderSample* web application at www.code.sd.

Note:

You can put any global variables/objects inside public part of main Data Module, like database connection objects, socket components, and any other object that are needed in other data modules, because main data module is always created on every request.

Thread pooling

With Apache Module, you can make it faster in response when you are using database by using *Thread pooling*. By default *thread pooling* is disabled, on each request a new instance of web module will be created and destroyed on each user request. If you turn *Thread pooling* to *True,* created data module will not be destroyed, instead it will be a reusable resource, another requests could use it. This method will give you a ready, initialized and connected database components, also you can leave lookup tables open to be reused in another requests. If you use variables or any objects that are declared inside data module private or public sections, next call for this data module will get their last values, so that you have to be careful on using these variables and objects, don't forget to initialize the variables and objects if you need to.

To turn *Thread pooling* on in your FreeSpider Apache Module, go to the project source code, locate DefaultHander function and write change it's default code from:

```
Result:= ProcessHandler(r, TDataModule1, MODULE_NAME, HANDLER_NAME, False);
```

to

```
Result:= ProcessHandler(r, TDataModule1, MODULE_NAME, HANDLER_NAME, True);
```

In the first default option there is no thread pooling, and web modules will be created and freed upon every request, on the second option, thread pooling will be used.

Note that there is no link between requests from the same user, if the same user do multiple requests from the browser, FreeSpider may give different web module each time, so that cookies are still used to identify user session.

If you have a code in Data Modules *onCreate* event, it will be triggered only when there is no reusable web module in the thread pool, so that don't rely on this event to execute code that is re queried for each user request.

If you are using smart module design, only the main module will get benefit of thread pooling, and other data modules will be created and freed each time, so that put your database connection component and lookups on main module.

After one minute of last request to your apache module, all thread pooled data modules will be freed.

Maintaining two versions of FreeSpider: CGI and Apache Module

You can have two versions of executables: CGI executable and Apache Module library in the same FreeSpider Web application. To do this, suppose that you have already FreeSpider CGI application and you need to add Apache Module version. Follow these simple steps:

1. In Lazarus click Create new *FreeSpider Apache Module* Web application

2. Go to Project/Remove from project and remove main unit and close it from editor

3. Save your project in the same directory with your current CGI web application, and name it differently than CGI project, e.g. add mod_ to project name, like *mod_first*

4. Go to File/Open and open ***main.pas*** file which contains Data Module for your current CGI web application, and click Project/Add Editor file to project

5. If you have additional Data Modules, then repeat step 4 with each one

6. Drop ***TSpiderApache*** component in your main Data Module, and link its OnRequest method with current existing ***TSpiderCGI*** component, make them point to the same code.

7. Add ***FreeSpider*** directory into this project library path

8. Modify the configuration ***constants*** of Apache Module project source code to a proper one

9. Compile your project, and you should get a Apache Module library version of your old web application

Note:
You can go back to your CGI project to resume development and and you can go back to Apache project to produce new version of your web application.

Diagram

To understand the internal structure of FreeSpider package please download this diagram document

Performance

FreeSpider Apache Module can handle more traffic than CGI version and consumes less resources.

You can refer to FreeSpider web page to see the detailed performance test that has been done to FreeSpider applications.

Motaz Abdel Azeem
www.code.sd