Lazarus for the web: Sessions and Templates

Michaël Van Canneyt

April 10, 2006

Abstract

In this second article about Lazarus web programming, the support for sessions in Lazarus web modules is investigated.

1 Introduction

Most websites these days have some kind of session support: Either implicitly (to see how a user navigates through a site) or explicitly: requiring a user to log in to be able to navigate through certain parts of the website. The FPC/Lazarus web components provide support for simple session management using cookies which are sent to the browser: Values can be registered in a session, and they will be loaded when an active session is detected. The session variables are by default stored in ini files, but they can be stored in any desired way: it will be shown how to create a custom session storage using SQLDB: The database for session data will be a Firebird database, but this could be any other database.

In this article, it will be assumed that the weblaz package (introduced in the previous article) is installed in the Lazarus IDE, and that SQLDB is installed as well (this should be so by default as of Lazarus 0.9.14).

2 Sessions

Support for sessions is restricted to saving and loading a set of variables between client requests. To keep the interface simple, these variables are stored and retrieved as strings, and they are identified by strings. The sessions work through cookies, so the web-client (a browser) should have cookie supporte enabled, or the sessions will not work.

Sessions are defined in httpdefs, in the `TCustomSession` class. The methods and properties to manage the session are the following:

```
TCustomSession = Class(TComponent)
Public
  Procedure Terminate;
  Property TimeOutMinutes : Integer;
  Property SessionID : String;
  Property Variables[VarName : String] : String;
end;
```

They have the following meaning

Terminate Terminates the session. For instance on websites where the user is reauired to log in, this method can be called to invalidate the session when the user logs out.

TimeOutMinutes This property can be used to specify a timeout. The timeout can be used to automatically invalidate a session.

SessionID This is the unique ID that is used to identify a session. This ID is generated automatically by the session component. By default, this will be a GUID.

Variables This array property can be used to store and retrieve session variables. All variables will be saved at the end of the client request, and the stored variables will be retrieved when the session is re-opened.

Session support is introduced in `TSessionHTTPModule`, the parent class of `TFPWebModule`. This class introduces the following properties to deal with sessions:

```
TSessionHTTPModule = Class(TCustomHTTPModule)
  Property CreateSession : Boolean;
  Property Session : TCustomSession;
  Property OnNewSession : TNotifyEvent;
  Property OnSessionExpired : TNotifyEvent;
```

These properties act as follows:

CreateSession To activate sessions, the `CreateSession` property should be set to `True`. The other session-related properties will be ignored as long as this property is set to `False`

Session If this property is set to a descendent of `TCustomSession`, then that session component will be used to manage the session. If it is not set, a default (.ini file based) session will be used. This session is of type `TFPWebSession`, and is defined in unit `websession`.

OnNewSession This event is triggered when a new session is started: it can be used to set certain variables, modify the default actions, and so on.

OnSessionExpired This event is triggered when the timeout period of the session was reached. It can be used to display custom pages.

Sessions are initialized before the actual request is handled. First the session is initialized (which can trigger one of the 2 events described above) and only then the client request is handled.

To demonstrate this, a small project can be created. It stores a single value. This is not much, but shows how the session support works. The project is created using 'New' and then 'CGI Application'. A new project (name it cgisession) and a new module (name it sessionmodule) are created. The `CreateSession` property of the webmodule should be set to `True`

To the module, 3 actions are added:

SessionStarted This action will be executed when the session is started.

HaveSession This action will be executed if a session is active. This is the default action for the webmodule.

Terminate This action should be executed to terminate the session.

In the `OnNewSession` event handler, the following code is placed:

```
TSessionModule.SessionModuleNewSession(Sender: TObject);
begin
  Actions[1].Default:=False;
  Actions[0].Default:=True;
end;
```

This makes the `SessionStarted` action the default action if a new session was started. This works because the session is started before the request is handled.

The following code is placed in the `OnRequest` handler of the `SessionStarted` action.

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

Var
  C : TCookie;

begin
  With AResponse.Contents do
    begin
    Add('<HTML><TITLE>Session started</TITLE><BODY>');
    Add('<H1>Session started</H1>');
    Add('A session was started.');
    If Session is TFPWebSession then
      begin
      C:=AResponse.Cookies.FindCookie((Session as TFPWebSession).SessionCookie);
      If Assigned(C) then
        begin
        Add('Session Cookie is ');
        Add('called <B>'+C.Name+'</B> ');
        Add('and has value <B>'+C.Value+'</B>.');
        end
      else
        begin
        Add('Session cookie was not yet set.');
        end;
      end;
    Add('</BODY></HTML>');
    end;
  AResponse.SendContent;
  Handled:=True;
end;
```

The above code simply shows the session cookie value if the session is of type `TFPWebSession`.

By entering the URL

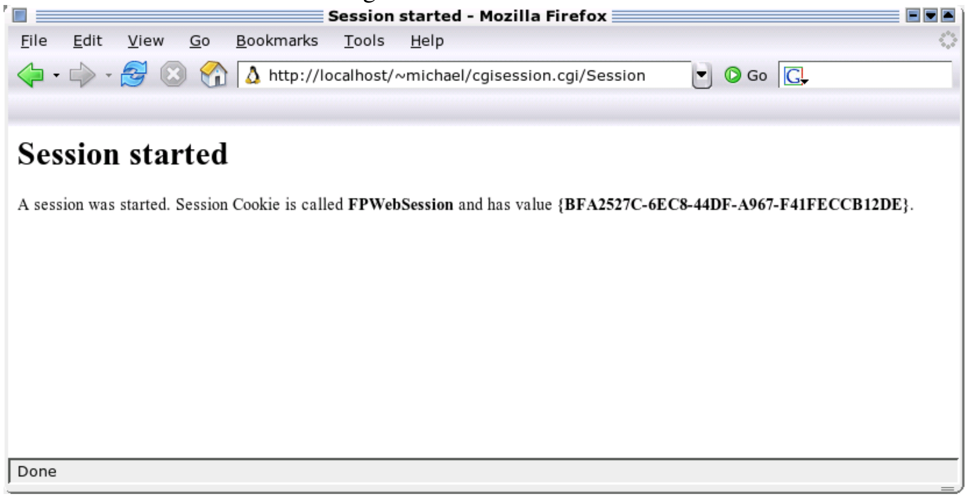
```
http://localhost/~michael/cgisession.cgi
```

In the browser, the page as shown in figure 1 on page 4 is sent to the browser.

Secondly, in the `OnRequest` handler of the `HaveSession` action, the following code is placed:

```
procedure TSessionModule.HaveSessionRequest(Sender: TObject;
```

Figure 1: Start of a session


Session started - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost/~michael/cgisession.cgi/Session
Go
Done

Session started

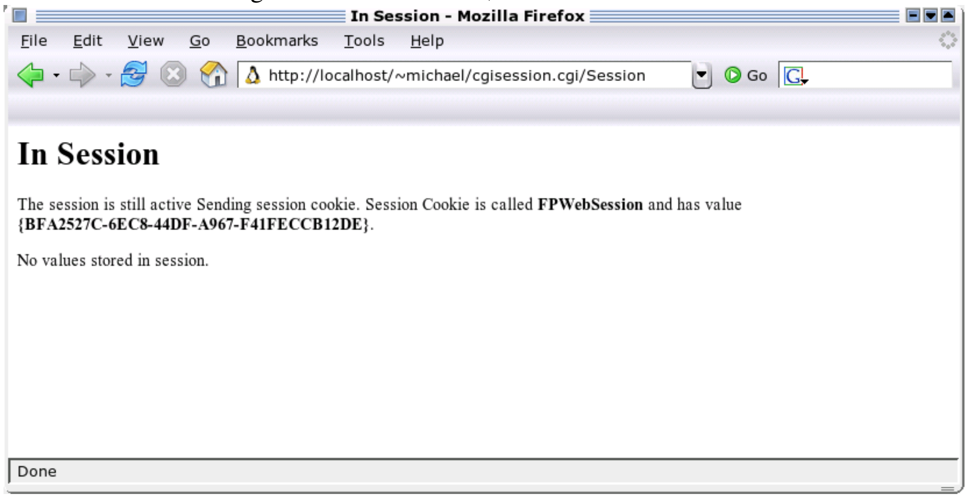
A session was started. Session Cookie is called **FPWebSession** and has value **{BFA2527C-6EC8-44DF-A967-F41FECCB12DE}**.

```
  ARequest: TRequest; AResponse: TResponse;
  var Handled: Boolean);

Var
  C : TCookie;
  V : String;
  WS : TFPWebSession;

begin
  With AResponse.Contents do
    begin
    Add('<HTML><TITLE>In Session</TITLE><BODY>');
    Add('<H1>In Session</H1>');
    Add('The session is still active');
    if Session is TFPWebSession then
      begin
      WS:=Session as TFPWebSession
      C:=AResponse.Cookies.FindCookie(WS.SessionCookie);
      If Assigned(C) then
        begin
        Add('Sending session cookie. Session Cookie is ');
        Add('called <B>'+C.Name+'</B> ');
        Add('and has value <B>'+C.Value+'</B>.');
        end;
      end;
    V:=Session.Variables['Value'];
    If (V<>'') then
      Add('<P>Stored session value: <B>'+V+'</B>.')
    else
      Add('<P>No values stored in session.');
    V:=ARequest.QueryFields.Values['Value'];
    If V<>'' then
      begin
      Add('<P>Storing new session value: <B>'+V+'</B>.');
      Session.Variables['Value']:=V;
```

Figure 2: Active session, no variables in session


In Session - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost/~michael/cgisession.cgi/Session
Go
Done

In Session

The session is still active Sending session cookie. Session Cookie is called **FPWebSession** and has value **{BFA2527C-6EC8-44DF-A967-F41FECCB12DE}**.

No values stored in session.

```
      end;
    end;
    Add('</BODY></HTML>');
    end;
end;
```

The above again shows the session cookie, and displays the value of the session variable `Value`, if it exists.

If a variable named `Value` is passed on to the CGI application by the browser, then it's value is stored in the session.

If the previous URL is invoked a second time, the result of the above code is page as shown in figure 2 on page 5.

By entering the URL

```
http://localhost/~michael/cgisession.cgi?Value=ABC
```

The variable `Value` is set to the value `ABC` and the page depicted in figure 3 on page 6 is sent to the browser.

Changing the URL to

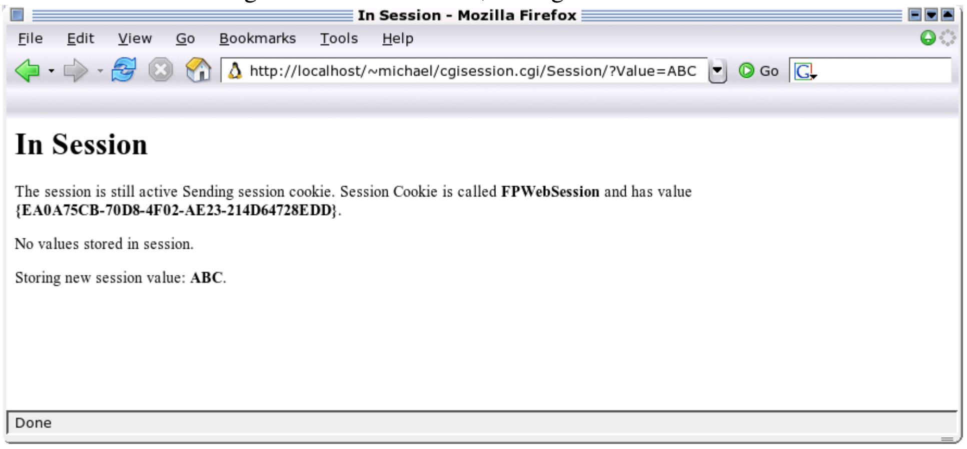
```
http://localhost/~michael/cgisession.cgi?Value=ABC2
```

will change the value to `ABC2` and will send the page as shown in figure 4 on page 6 to the browser.

Finally, entering the following code in the `OnRequest` handler of the `Terminate` action will terminat the session:

```
procedure TSessionModule.TerminateRequest(Sender: TObject;
  ARequest: TRequest; AResponse: TResponse;
  var Handled: Boolean);
begin
  Session.Terminate;
  With AResponse.Contents do
    begin
```

Figure 3: Active session, setting variable `Value`


In Session - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost/~michael/cgisession.cgi/Session/?Value=ABC
Go

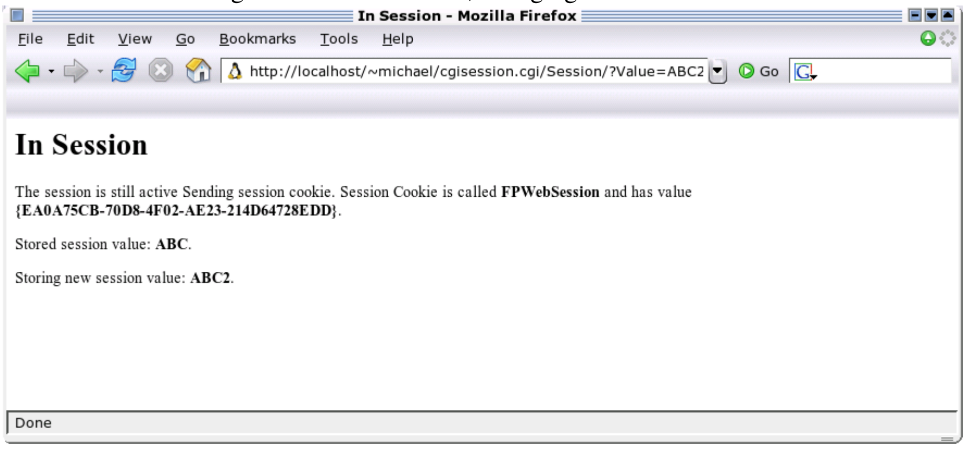
In Session

The session is still active Sending session cookie. Session Cookie is called **FPWebSession** and has value **{EA0A75CB-70D8-4F02-AE23-214D64728EDD}**.

No values stored in session.

Storing new session value: **ABC**.

Figure 4: Active session, changing variable `Value`


In Session - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost/~michael/cgisession.cgi/Session/?Value=ABC2
Go

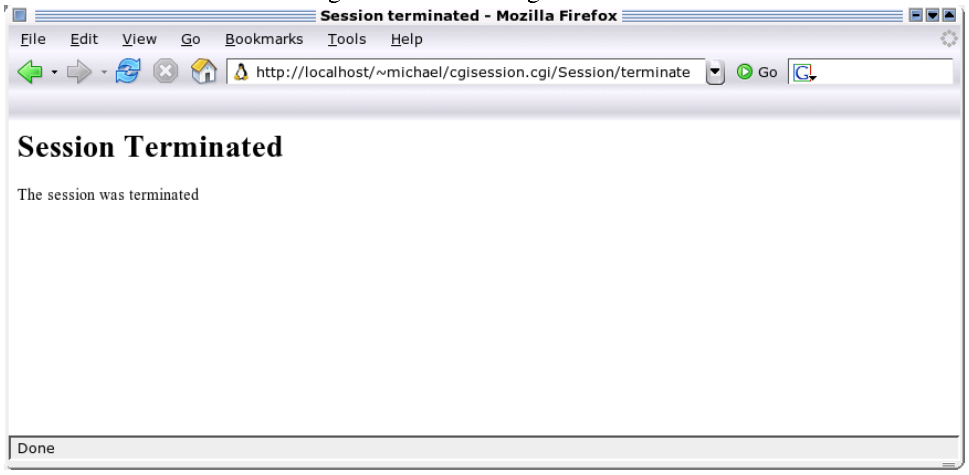
In Session

The session is still active Sending session cookie. Session Cookie is called **FPWebSession** and has value **{EA0A75CB-70D8-4F02-AE23-214D64728EDD}**.

Stored session value: **ABC**.

Storing new session value: **ABC2**.

Figure 5: Terminating the session


Session terminated - Mozilla Firefox

http://localhost/~michael/cgisession.cgi/Session/terminate

Session Terminated

The session was terminated

Done

```
    Add('<HTML><TITLE>Session terminated</TITLE><BODY>');
    Add('<H1>Session Terminated</H1>');
    Add('The session was terminated');
    Add('</BODY></HTML>');
    end;
end;
```

The session can be ended by entering the query

```
http://localhost/~michael/cgisession.cgi/terminate
```

resulting in the page shown in figure 5 on page 7.

3 Custom session storage

The default session storage mechanism is implemented using .ini files. This is a leightweight implementation, which does not require a lot of code, thus making the resulting CGI application very small and fast.

The default session is implemented in the `TFPWebSession` class and is implemented in the `websession` unit. Per session, a .ini file is written, with the session cookie value as the name (a GUID). The public interface of `TFPWebSession` contains the following properties:

Cached if `False`, then after a variable is set, the whole .ini file is written to disk. This hurts performance, but avoids data loss.

SessionCookie The name of the cookie to be used when managing the session. By default, this is `FPWebSession`.

SessionDir The name of the directory where the .ini files are written. By default this is the temporary directory (actually, the value of the `GlobalSessionDir` variable). Obviously, this directory should be writable by the CGI process.

Now, many CGI applications will want to use a database. Since the database is present anyway, it is probably a good idea to store the session data in the database as well.

For this a `TSQLDBSession` component will be developed. This component can be used to store the data in a database. For the demo application, a Firebird database was used. In the database, 2 tables are used:

SESSIONS This table keeps the list of active sessions. It also keeps the session parameters, the start time, last seen time and the timeout. It can be created with the following SQL Statement:

```
CREATE TABLE SESSIONS (
 SE_ID INT NOT NULL,
 SE_SESSION CHAR(38) NOT NULL,
 SE_STARTED TIMESTAMP DEFAULT 'NOW' NOT NULL,
 SE_LASTSEEN TIMESTAMP DEFAULT 'NOW' NOT NULL,
 SE_TIMEOUT INT DEFAULT 15 NOT NULL,<
 CONSTRAINT PK_SESSIONS PRIMARY KEY (SE_ID),
 CONSTRAINT U_SESSIONS UNIQUE (SE_SESSION)
);
```

The `SE_SESSION` field is used to store the session ID (a GUID). The `SE_STARTED` field can be used for determining when the session started. The `SE_LASTSEEN` field keeps the time of the last client request. `SE_TIMEOUT` field keeps the timeout in minutes. The `SE_ID` field is an autonumber field, which will be filled by a trigger. Using an integer field as the primary key keeps the indexes smaller and faster than when the `SE_SESSION` field was used for the primary kay.

SESSIONDATA This table keeps the actual data. It can be created with the following SQL statement:

```
CREATE TABLE SESSIONDATA (
  SD_SESSION_FK INT NOT NULL,
  SD_NAME VARCHAR(64) NOT NULL,
  SD_VALUE VARCHAR(1024),
  CONSTRAINT U_SESSIONDATA UNIQUE (SD_SESSION_FK,SD_NAME),
  CONSTRAINT R_SESSIONDATA_SESSIONS
    FOREIGN KEY (SD_SESSION_FK)
    REFERENCES SESSIONS(SE_ID) ON DELETE CASCADE
);
```

The meaning of the fields should be obvious. The reference to the `SESSIONS` table is cascading, meaning that if the session is deleted from the `SESSIONS` table, all data associated with it is deleted too.

Now that the structure of the database is clear, the component can be coded.

The `TCustomSession` class has 5 abstract methods that must be overridden, they are shown below (there are other methods, but these are not shown. The reader is referred to the source files on the CD accompagnying this issue):

```
TSQLDBSession = Class(TCustomSession)
Public
  Function GetSessionVariable(VarName : String) : String;
  procedure SetSessionVariable(VarName : String;
                             const AValue: String);
  Procedure UpdateResponse(AResponse : TResponse);
  Procedure InitSession(ARequest : TRequest;
                      OnNewSession, OnExpired: TNotifyEvent);
  Procedure InitResponse(AResponse : TResponse);
  Procedure Terminate;

Published
    Property Connection : TSQLConnection Read FConnection Write SetConnection;
    Property SessionCookie : String Read FSessionCookie Write FSessionCookie;
end;
```

The 2 properties are the SQLDB connection component reference, and the `SessionCookie` wich is the name of the cookie to be used when managing the session.

The methods to be overridden are the following:

GetSessionVariable This method is the `Read` specifier of the `Variables` property of `TCustomSession`. It should return the value of the named variable, or return an empty string if there is no variable with the given name.

SetSessionVariable This method is the `Read` specifier of the `Variables` property of `TCustomSession`. It should set the value of the named variable, creating the variable if needed.

InitResponse This method is called when the response for the web-browser is initialized: at this point, no HTTP headers or content will have been sent: headers can still be added: at this point, a cookie can and should be sent.

InitSession This is called when the session should be initialized. The request is passed, as well as 2 events: one when a new session is started, one when the session has expired.

UpdateResponse this is called after the request was handled: it can be used to update the response (although this is dangerous, as it can happen that the response headers were already sent).

Terminate called when a session should be terminated. This should clean up all data associated with the session.

For faster operation, the session variables will be stored in a stringlist, as name=value pairs. When the session is initialized, the variables are loaded from the database. When the list is updated, then the added variables will be stored in the database as well.

The first method that will be called is the `InitSession` method. It's coded as follows:

```
procedure TSQLDBSession.InitSession(ARequest: TRequest; OnNewSession,
  OnExpired: TNotifyEvent);

Var
  L   : TDateTime;
  T   : Integer;
  S : String;

begin
  CheckSession;
  S:=ARequest.CookieFields.Values[SessionCookie];
  If (S<>'') and GetSessionData(S,L,T) then
    begin
    If ((Now-L)>(T/(24*60))) then
```

```
      begin
      If Assigned(OnExpired) then
        OnExpired(Self);
      DeleteSession(S);
      S:='';
      end
    else
      SID:=S;
    end;
  If (S='') then
    begin
    If Assigned(OnNewSession) then
      OnNewSession(Self);
    GetSessionID;
    CreateSession(SID);
    FVariables.Clear;
    end
  else
    begin
    UpdateSessionTime(SID);
    LoadSessionData(SID,FVariables);
    end;
  FSessionStarted:=True;
end;
```

The `CheckSession` method will see if a database connection is present, and will initialize the SessionCookie property, if it was not set. After this the session data is retrieved from the database: the `GetSessionData` will return `True` if data for the session was found. If so, the data is checked: the variable `L` will contain the last seen time. The variable `T` contains the timeout of the session. If the session timed out, then the `OnExpired` handler is called, and the session is deleted from the database (in `DeleteSession`)

If at this point, there is no active session, a session is started: a session ID is created (using `GetSessionID`), and the session is created in the databas (using `CreateSession`). The list of session variables (in `FVariables`, a `TStrings` instance) is cleared.

If there is an active session, the last session time is set in `UpdateSessionTime`. The call to `LoadSessionData` will load the list of variables with values found in the database.

The `InitResponse` call will be called after the session is started. It's a very simple method, all it does is send a cookie with the session ID:

```
procedure TSQLDBSession.InitResponse(AResponse: TResponse);
Var
  C : TCookie;
begin
  If FSessionStarted then
    begin
    C:=AResponse.Cookies.Add;
    C.Name:=SessionCookie;
    C.Value:=SID;
     end
  else If FTerminated then
    begin
    C:=AResponse.Cookies.Add;
    C.Name:=SessionCookie;
    C.Value:='';
    end;
end;
```

The code is self-explanatory.

The `UpdateResponse` method is not used, except for committing the transaction in the database:

```
procedure TSQLDBSession.UpdateResponse(AResponse: TResponse);
begin
  If Assigned(FTransaction) and FTransaction.Active then
    FTransaction.Commit;
end;
```

Since this is called after the request was handled, all variables will be in the database at this point.

When a variable is requested, it is simply retrieved from the stringlist:

```
function TSQLDBSession.GetSessionVariable(
  VarName: String): String;
begin
  CheckSession;
  Result:=FVariables.Values[VarName];
end;
```

When a variable needs to be set, it is either inserted in the database or it's value is updated, using the `InsertVariable` or `UpdateVariable` calls.

```
procedure TSQLDBSession.SetSessionVariable(VarName: String;
                                           AValue: String);

Var
  I : Integer;

begin
  CheckSession;
  I:=FVariables.IndexOfName(VarName);
  If (I=-1) then
    begin
    if (AValue<>'') then
      InsertVariable(SID,VarName,AValue)
    end
  else
    UpdateVariable(SID,VarName,AValue);
  FVariables.Values[VarName]:=AValue;
end;
```

This mechanism ensures that a minimal number of queries is run on the database.

When the session is terminated, the `Terminate` method is called:

```
procedure TSQLDBSession.Terminate;
begin
  FTerminated:=True;
  FSessionStarted:=False;
  DeleteSession(SID);
  SID:='';
  FVariables.Clear;
end;
```

The `DeleteSession` method deletes all session data from the database.

The session data is loaded from the database in the `GetSessionData` method, which simply runs a query (the exact SQL statement can be found in the `SGetSessionData` constant) on the database:

```
Function TSQLDBSession.GetSessionData(Const ASID : String;
                                      Var Lastseen : TDateTime;
                                      Var ATimeOut : Integer)
                                      : Boolean;

begin
  With FQuery do
    begin
    SQL.Text:=Format(SGetSessionData,[ASID]);
    Open;
    Try
      Result:=Not (EOF AND BOF);
      if Result then
        begin
        LastSeen:=FieldByName('SE_LASTSEEN').AsDateTime;
        ATimeOut:=FieldByName('SE_TIMEOUT').AsInteger;
        end;
    Finally
      Close;
    end;
    end;
end;
```

The `FQuery` object (of class TSQLQuery) is created when the session object is created. It is configured in the `CheckSession` method:

```
procedure TSQLDBSession.CheckSession;
begin
  If Not Assigned(FConnection) then
    raise ESession.CreateFmt(SErrNoConnection,[Name]);
  If (SessionCookie='') then
    SessionCookie:=SFPWebSession;
  FConnection.Connected:=True;
  If FQuery.Database<>FConnection then
    begin
    FQuery.Database:=FConnection;
    FTransaction.Database:=FConnection;
    end;
end;
```

This method is called when a session is started. It simply sets up the query and transaction component for use with the connection.

The session variables are loaded in the `LoadSessionData` method:

```
Procedure TSQLDBSession.LoadSessionData(Const ASID : String;
                                        List : TStrings);

Var
  FN,FV : TField;

begin
  List.Clear;
  With FQuery do
    begin
    SQL.Text:=Format(SGetVariables,[ASID]);
    Open;
    Try
      FN:=Fields.FieldByName('SD_NAME');
      FV:=Fields.FieldByName('SD_VALUE');
      While not EOF do
        begin
        List.Add(FN.AsString+'='+FV.AsString);
        Next;
        end;
    Finally
      Close;
    end;
    end;
end;
```

The method is self explanatory.

All other methods are simple query methods: they format an insert/update/delete query, and execute it on the database. The interested reader is referred to the sources in the `sqldbsession` unit.

To use this session, there are 2 options: register it in Lazarus, drop it on the session webmodule, and set the `Session` property of the webmodule.

To avoid having to register it in the IDE, it will be created instead in code in the `OnCreate` method of the session datamodule:

```
procedure TSessionModule.SessionModuleCreate(Sender: TObject);

Var
  S :TSQLDBSession;

begin
  S:=TSQLDBSession.Create(Self);
  S.Connection:=FPCDB;
  Session:=S;
end;
```

The `FPCDB` component is a `TIBConnection` component which was configured to work on a database that has the tables as indicated above. This little piece of code is all that is needed to change from a .ini file based session management to a database session management. Recompiling the CGI application and entering the appropriate URL's in a browser will convince us that the session management still works, only now using a database.

4 Conclusion

In this second article about web programming, a non-visual aspect of web-programming was examined: Session management, or keeping the state of the client who is browsing a website. It was shown that the web support of FPC/Lazarus supports sessions by default, and that with a little effort, sessions can be managed using a database as the back-end. No fancy output was created in this article: this is the subject of the next articles about web-programming, where the support for creating HTML will be examined.