FPC - FREE PASCAL TUTORIALS

sqlite 2

SQLite is a fast file-based SQL database. The units used in FPC is async, so a lot of the operations you don't wait on, instead you give it a function and when its done it will fire it.

Subpages (6): 1) Opening and Closing 2) Basic Query 3) Asynchronous Query 4) Inserts & Last ID 5) Inserting Unsafe Data 6) Fetching Rows

Updates
- Just the Begining
- Untitled Post
- Why Google Sites?

Distributed Units
- base64
- fcl-image
- sqlite 2
- sqlite-3

GUI Units

Sitemap

Distributed Units > sqlite 2 >

1) Opening and Closing

<- Back to sqlite   Next: 2) Basic Query ->

(TODO)

```
program test;

uses sqlite, sqlitedb, strings, classes;

var
  Db: TSQLite;
  Sql: String;
begin
  Db := TSQLite.Create('test.db');
  Db.Free;
end.
```

<- Back to sqlite   Next: Basic Query ->

Updates
Just the Begining
Untitled Post
Why Google Sites?
Distributed Units
base64
fcl-image
sqlite 2
sqlite-3
GUI Units
Sitemap

Distributed Units > sqlite 2 >

2) Basic Query

<- Opening and Closing Next: Asynchronous Query ->

Because SQLite is integrated in with Asynchronous IO, its makes basic queries a bit more complex. Most of the time you'd want to wait for a database call to finish before you proceed. To work around this we will make a loop checking it the query has been completed.

Now if you are familiar with SQL in general you can't create the same table over and over. You'll get a SQL error. This unit will tell you there is some error but it would tell you that specially.

```
{$mode objfpc}{$h+}
program test;

uses sqlite, sqlitedb, strings, classes;

var
  Db: TSQLite;
  Sql: String;
begin
  Db := TSQLite.Create('test.db');

  { Its very important that all queries end with a ";" }
  Sql := 'CREATE TABLE members ('
    + 'members_id INT,'
    + 'username VARCHAR(32),'
    + 'passwd VARCHAR(32)'
    + ');';
  { Call the query }
  Db.Query(Sql, nil);

  { Wait till query is completed }
  while Db.IsComplete(Sql) = False do
  begin
    { Do Nothing But Wait... }

    { If there was a parsing error this would loop forever }
    if Db.LastError <> 0 then
    begin
      { There was a parse error }
      break;
    end;
  end;

  { Sql Error? }
  if Db.LastError <> 0 then
  begin
    WriteLn('There was an error:');
    WriteLn(Db.LastErrorMessage);
  end else begin
    WriteLn('Created Table!');
  end;

  Db.Free;
end.
```

[<- Opening and Closing](#) [Next: Asynchronous Query ->](#)

Distributed Units > sqlite 2 >

3) Asynchronous Query

<- Basic Query Next: Inserts & Last ID ->

Async queries are much suited for interfaced applications. They allow expensive calls to be processed in the background while your application keeps on kicking. Now if your needing that query before anything else can happen you want synchronized queries and can review the "Basic Query"

```
{$mode objfpc}{$h+}
program test;

uses sqlite, sqlitedb, strings, classes;

type
  TDbCallback = Object
    Db: TSQLite;
    procedure OnQueryComplete(Sender: Tobject);
  end;

procedure TDbCallback.OnQueryComplete(Sender: TObject);
begin
  { Sql Error? }
  if Db.LastError <> 0 then
  begin
    WriteLn('There was an error:');
    WriteLn(Db.LastErrorMessage);
  end else begin
```

```
    WriteLn('Created Table!');
  end;
end;

var
  Db: TSQLite;
  Sql: String;
  Callback: TDbCallback;
begin
  Db := TSQLite.Create('test.db');
  Callback.Db := Db;

  { Its very important that all queries end with a ";" }
  Sql := 'CREATE TABLE members ('
    + 'members_id INT,'
    + 'username VARCHAR(32),'
    + 'passwd VARCHAR(32)'
    + ');';

  { Assign the async database call }
  Db.OnQueryComplete := @Callback.OnQueryComplete;

  { Call the query }
  Db.Query(Sql, nil);

  {Does an async call! lalalal }

  Db.Free;
end.
```

<- Basic Query   Next: Inserts & Last ID ->

4) Inserts & Last ID

<- Asynchronous Query

Next: Inserting User Data ->

Inserts are straight forward, however you want to very careful with the data you insert into the SQLite database. It is not properly filtered you'll have a big problem with users destroying your data. In the next segment we'll go over sanitizing user data.

Function Overview:

Db.Query(Sql, nil);

Db.LastInsertRow();

Example

```
program test;

uses sqlite, sqlitedb, strings, classes;

var
  Db: TSQLite;
  Sql: String;
begin
  Db := TSQLite.Create('test.db');

  { Its very important that all queries end with a ";" }
  Sql := 'INSERT INTO members (username, passwd) VALUES("joseph", "success");';

  { Call the query }
  Db.Query(Sql, nil);

  { Wait till query is completed }
  while Db.IsComplete(Sql) = False do
  begin
    { Do Nothing But Wait... }

    { If there was a parsing error this would loop forever }
    if Db.LastError <> 0 then
    begin
      { There was a parse error }
      break;
    end;
  end;

  { Sql Error? }
  if Db.LastError <> 0 then
  begin
    WriteLn('There was an error:');
    WriteLn(Db.LastErrorMessage);
  end else begin
    WriteLn('Row Inserted:');
    WriteLn(Db.LastInsertRow());
  end;

  Db.Free;
end.
```

<- Asynchronous Query   Next: Inserting User Data ->

5) Inserting Unsafe Data

<- Inserts & Last ID   Next: Fetching Rows ->

Currently filtering composes of adding quotes around a string, and filtering out and single quote to two single quotes. Two consecutive single quotes tell SQLite to ignore the quote for escaping. Funny enough Pascal does the same thing! Maybe the world is connected a little bit closer then you think.

Filtering the Data

```
program test;

uses sqlite, sqlitedb, strings, classes;

var
  Db: TSQLite;
  Sql: String;
begin
  Db := TSQLite.Create('test.db');

  { Its very important that all queries end with a ";" }
  Sql := 'INSERT INTO members (username, passwd) ' +
    'VALUES(' + Pas2SQLStr('O''Dona')  + ', ' + Pas2SQLStr('success')  + ');';
  { Output:
      INSERT INTO members (username, passwd) VALUES('O''Dona', 'success');
  }

  { Call the query }
  Db.Query(Sql, nil);
```

```
    { Wait till query is completed }
    while Db.IsComplete(Sql) = False do
    begin
      { Do Nothing But Wait... }

      { If there was a parsing error this would loop forever }
      if Db.LastError <> 0 then
      begin
        { There was a parse error }
        break;
      end;
    end;

    { Sql Error? }
    if Db.LastError <> 0 then
    begin
      WriteLn('There was an error:');
      WriteLn(Db.LastErrorMessage);
    end else begin
      WriteLn('Row Inserted!');
      WriteLn(Db.LastInsertRow());
    end;

    Db.Destroy;
  end.
```

<- Back to Inserts & Last ID   Next: Fetching Rows ->

Distributed Units > sqlite 2 >

6) Fetching Rows

<- Back to Inserting Unsafe Data   Next: (TODO) ->

Fetching a row or much less many rows is a bit of a pain. As of right now there is no direct interface to column names to row data. So you have to iterate through all your columns before you can figure out its true position in the row of data. This is the reason why this example is a bit excessive, but it gets the job done!

Example:

```
program test;

uses sqlite, sqlitedb, strings, classes, contnrs, sysutils;

var
  Db: TSQLite;
  Sql: String;
  Columns: TStringList;
  i: Integer;
  HashNames: TFPStringHashTable;
  HashIndex: Integer;
begin
  Db := TSQLite.Create('test.db');

  { Its very important that all queries end with a ";" }
  Sql := 'SELECT * FROM members;';

  { Call the query }
  Db.Query(Sql, nil);

  { Wait till query is completed }
  while Db.IsComplete(Sql) = False do
  begin
    { Do Nothing But Wait... }

    { If there was a parsing error this would loop forever }
    if Db.LastError <> 0 then
    begin
      { There was a parse error }
      break;
    end;
  end;

  { Sql Error? }
  if Db.LastError <> 0 then
  begin
    WriteLn('There was an error:');
    WriteLn(Db.LastErrorMessage);
  end else begin
      { Generate a hash to refer to the field names }
      HashNames := TFPStringHashTable.Create;

      { Get the field names }
      for i := 0 to Db.List_FieldName.count - 1 do
      begin
        HashNames[Db.List_FieldName.Strings[i]] := IntToStr(i);
      end;

      { Grab the Rows }
      for i := 0 to Db.List_Field.count - 1 do
      begin
        Columns := TStringList(Db.List_Field.items[i]);
        HashIndex := StrToInt(HashNames.Items['username']);
        WriteLn(i, ' -> username: ', SQL2PasStr(Columns.Strings[HashIndex]), ' ');
      end;
  end;

  Db.Destroy;
end.
```

<- Back to Inserting Unsafe Data

Next: (TODO) ->