ADO Datenbank SQL Dump

Nun noch ein zweites Beispiel wo sicherlich einige Redundanzen in der Realisierung des XML Exportes existieren, aber vielleicht wird gerade dadurch klar das es sinnvoll ist einige ADO Interfaces mit einander zu verbinden und eine Vereinfachungen für den Zugriff auf Teilbereiche zu realisieren. Für kleinere Projekte ist es sicherlich nicht erforderlich, wie ich mit diesen beiden Beispielen andeuten möchte, jedoch bei größeren Projekten wünsche ich mir eine Klassenstruktur die auf Connection, Record (Table), Fields, Field, ... setzt.

ADODBDump.lpr Pascal (10,11 kByte) 29.12.2013 12:46

```
program ADODBDump;
{$APPTYPE CONSOLE}
{$mode objfpc}{$H+}
uses
  Classes, // for TFileStream
  SysUtils, ActiveX, Variants,
  Interfaces, // used packages: LazActiveX
  adodb_6_1_tlb, // Microsoft ActiveX Data Objects 2.x-Objektbibliothek
  adox_6_0_tlb; // ADO Extensions 2.5 for DDL and Security Library Reference

function IIF(b: boolean; sTrue: string; sFalse: string = ''): string;
begin if b then result := sTrue else result := sFalse; end;

procedure CreateDBDump(const Filename: string);
var
  cn: _Connection;
  cat: _Catalog;
  sTable: Widestring;

  function GetAsUTF8String(const value: OleVariant): UTF8String;
  var
    index, lowVal, highVal: integer;
    oleArray: PSafeArray;
    oleObj: OleVariant;
  begin
    try
      case TVarData(Value).vType of
        varEmpty:
          result := '';
        varNull:
          result := 'NULL';
        varSmallint, varInteger, varByte, varError:
          result := IntToStr(Value);
        varSingle, varDouble, varCurrency:
          result := FloatToStr(Value);
        varDate:
          result := '''' + DateTimeToStr(Value) + '''';
        varOleStr, varStrArg, varString:
          begin
            result := UTF8Encode(Widestring(Value));
```

```
            result := StringReplace(result, '\', '\\', [rfReplaceAll]);
            result := StringReplace(result, '''', '\''', [rfReplaceAll]);
            result := StringReplace(result, #13#10, '\n', [rfReplaceAll]);
            result := StringReplace(result, '&', '&amp;', [rfReplaceAll]);
            result := StringReplace(result, '<', '&lt;', [rfReplaceAll]);
            result := '''' + StringReplace(result, '>', '&gt;',
[rfReplaceAll]) + '''';
          end;
        varBoolean:
          if Value then
            result := 'true'
          else
            result := 'false';
        varDispatch, varVariant, varUnknown, varTypeMask:
          begin
            VarAsType(Value, varOleStr);
            result := UTF8Encode(Widestring(Value));
          end;
        else
          if VarIsArray(Value) then
          begin
            VarArrayLock(Value);
            index := VarArrayDimCount(Value);
            lowVal := VarArrayLowBound(Value, index);
            highVal := VarArrayHighBound(Value, index);
            oleArray := TVariantArg(Value).pArray;

            for index := lowVal to highVal do
            begin
              SafeArrayGetElement(oleArray, @index, oleObj);
              result := result + GetAsUTF8String(oleObj) + #13#10;
            end;

            VarArrayUnlock(Value);
            Delete(result, length(result) - 1, 2);
          end
          else
            result := ''; // varAny, varByRef
      end;
    except
      result := '#error#'
    end;
  end;

  function GetTypeAsUTF8String(const field: Field): UTF8String;
  var s: UTF8String;
  begin
    result := '';
    if cn.Provider = 'SQLOLEDB.1' then
      case field.Type_ of
```

```
    adSmallInt:          result := 'SmallInt';
    adInteger:           result := 'Int';
    adSingle:            result := 'Real';
    adDouble:            result := 'Float';
    adCurrency:          result := 'Money';
    adDate:              result := 'DateTime';
    adBoolean:           result := 'Bit';
    adVariant:           result := 'Sql_Variant';
    adUnsignedTinyInt:   result := 'TinyInt';
    adBigInt:            result := 'BigInt';
    adGUID:              result := 'UniqueIdentifier';
    adBinary:            result := 'Binary';
    adChar:              result := 'Char';
    adWChar:             result := 'NChar';
    adNumeric:           result := 'Numeric';
    adDBTimeStamp:       result := 'DateTime';
    adVarChar:           result := 'VarChar';
    adLongVarChar:       result := 'Text';
    adVarWChar:          result := 'NVarChar';
    adLongVarWChar:      result := 'NText';
    adVarBinary:         result := 'VarBinary';
    adLongVarBinary:     result := 'Image';
    else                 result := 'Unknown';
    end
  else //if Provider = 'Microsoft.Jet.OLEDB.4.0' then
    case field.Type_ of
    adSmallInt:          result := 'Integer';
    adInteger:           result := 'Integer';
    adSingle:            result := 'Single';
    adDouble:            result := 'Double';
    adCurrency:          result := 'Currency';
    adDate:              result := 'Date';
    adBoolean:           result := 'YesNo';
    adUnsignedTinyInt:   result := 'Byte';
    adGUID:              result := 'ReplicationID';
    adNumeric:           result := 'Decimal';
    adDBTimeStamp:       result := 'DateTime';
    adVarChar:           result := 'Text';
    adLongVarChar:       result := 'Memo';
    adVarWChar:          result := 'Text';
    adLongVarWChar:      result := 'Memo';
    adVarBinary:         result := 'ReplicationID';
    adLongVarBinary:     result := 'OLEObject';
    else                 result := 'Unknown';
    end;
  case field.Type_ of
  adBinary, adBSTR, adChar, adWChar, adVarChar, adVarWChar:
    result := result + '(' + IntToStr(field.DefinedSize) + ')';
  adSingle, adDouble, adCurrency, adNumeric, adVarNumeric:
    begin
```

```
        result := result + '(' + FloatToStr(field.Precision) + ',' +
FloatToStr(field.NumericScale) + ')';
        if (field.Attributes and adParamSigned)=0 then
          result := result + ' unsigned';
      end;
    end;
    if (field.Attributes and adFldIsNullable)=0 then
      result := result + ' NOT NULL';
    s :=
GetAsUTF8String(cat.Tables[sTable].Columns[field.Name].Properties['Default']
.Value);
    if s<>'' then
      result := result + ' DEFAULT ' + s;
    if field.Properties['ISAUTOINCREMENT'].Value = 'True' then
      result := result + ' AUTO_INCREMENT';
  end;

var
  rsTables, rsTable, rsKeys: _Recordset;
  s, pk, fk, sFields, sValues, index: UTF8String;
  f: TFileStream;
  i: integer;
begin
//connect db
  cn := CoConnection.Create;
  cn.Open('Provider=Microsoft.Jet.OLEDB.4.0;Data Source=' + Filename, '',
'', 0);
  if cn.State = adStateOpen then
  begin
    cat := CoCatalog.Create;
    cat.Set_ActiveConnection_(cn);
    f := TFileStream.Create(ChangeFileExt(Filename, '-dump.sql'),
fmOpenWrite or fmCreate);
    rsTables := CoRecordset.Create;
    rsKeys := CoRecordset.Create;
    rsTable := CoRecordset.Create;
    try
      rsTables := cn.OpenSchema(adSchemaTables, EmptyParam, EmptyParam);
      while not (rsTables.EOF_) do
      begin
        if (rsTables.Fields['TABLE_TYPE'].Value = 'TABLE') then // only
tables
        begin
          sTable := rsTables.Fields['TABLE_NAME'].Value;
          s := 'DROP TABLE IF EXISTS `' + sTable + '`;'#13#10 +
               'CREATE TABLE IF NOT EXISTS `' + sTable + '` (';
          f.Write(Pchar(s)^, Length(s));

          // table properties
          sFields := '';
```

```
          rsTable.Open(sTable, cn, adOpenForwardOnly, adLockReadOnly,
adCmdTable);
          for i := 0 to rsTable.Fields.Count-1 do
          begin
            sFields := sFields + '`' + rsTable.Fields[i].Name + '`' +
IIF(i<rsTable.Fields.Count-1, ',');
            s := #13#10'  `' + rsTable.Fields[i].Name + '`: ' +
GetTypeAsUTF8String(rsTable.Fields[i]) + IIF(i<rsTable.Fields.Count-1, ',');
            f.Write(Pchar(s)^, Length(s));
          end;

          // primary key & index
          pk := '';
          index := '';
          rsKeys := cn.OpenSchema(adSchemaIndexes, VarArrayOf([Unassigned,
Unassigned, Unassigned, Unassigned, sTable]), EmptyParam);
          while not rsKeys.EOF_ do
          begin
            if (rsKeys.Fields['PRIMARY_KEY'].Value) then
              pk := pk + IIF(pk<>'', ' ,') + '`' +
GetAsUTF8String(rsKeys.Fields['COLUMN_NAME'].Value) + '`'
            else
              index := index + IIF(index<>'', ' ,') + '`' +
GetAsUTF8String(rsKeys.Fields['COLUMN_NAME'].Value) + '`';
            rsKeys.MoveNext;
          end;
          rsKeys.Close;
          if pk <> '' then
          begin
            s := ','#13#10'  PRIMARY KEY (' + pk + ')';
            f.Write(PChar(s)^, Length(s));
          end;

          // foreign keys
          fk := '';
          rsKeys := cn.OpenSchema(adSchemaForeignKeys,
VarArrayOf([UnAssigned, UnAssigned, sTable]), EmptyParam);
          while not rsKeys.EOF_ do
          begin
            s := '';
            if TVarData(rsKeys.Fields['PK_COLUMN_NAME'].Value).vType <>
varNull then
              s := 'FOREIGN KEY (`' +
GetAsUTF8String(rsKeys.Fields['PK_COLUMN_NAME'].Value) + '`)';
            if TVarData(rsKeys.Fields['FK_TABLE_NAME'].Value).vType <>
varNull then
              s := s + ' REFERENCES `' +
GetAsUTF8String(rsKeys.Fields['FK_TABLE_NAME'].Value) + '`';
            if TVarData(rsKeys.Fields['FK_COLUMN_NAME'].Value).vType <>
varNull then
```

```
            s := s + ' (`' +
GetAsUTF8String(rsKeys.Fields['FK_COLUMN_NAME'].Value) + '`)';
            fk := fk + IIF(fk<>'', ', ') + s;
            rsKeys.MoveNext;
          end;
          rsKeys.Close;

          if fk <> '' then
          begin
            s := ','#13#10 + '  ' + fk;
            f.Write(PChar(s)^, Length(s));
          end;

          s := ');' + #13#10#13#10;
          f.Write(PChar(s)^, Length(s));

          sFields := 'INSERT INTO `' + sTable + '` (' + sFields + ')';
          // values
          while not rsTable.EOF_ do
          begin
            sValues := '';
            for i := 0 to rsTable.Fields.Count-1 do
            begin
              s := GetAsUTF8String(rsTable.Fields[i].Value);
              sValues := sValues + IIF(sValues<>'', ', ') + s;
            end;
            s := sFields + ' VALUES (' + sValues + ');'#13#10;
            f.Write(Pchar(s)^, Length(s));
            rsTable.MoveNext;
          end;
          rsTable.Close;
          s := #13#10;
          f.Write(Pchar(s)^, Length(s));
        end;
        rsTables.MoveNext;
      end;
      rsTables.Close;
    finally
      rsTable := nil;
      rsKeys := nil;
      rsTables := nil;;
      f.Free;
    end;
    cat := nil;
    cn.Close;
  end;
  cn := nil;
end;

begin
```

```
  CreateDBDump(ParamStr(1));
end.
```

Autor: Udo Schmal, veröffentlicht: 25.12.2013, letzte Änderung: 29.08.2024