

PostgreSQL for Hollywood

Inhaltsverzeichnis

Einführung	3
Anforderungen	3
Lizenz	3
History	4
Future	5
Erste Schritte	5
Einfaches Tutorial	5
Erweitertes Tutorial	9
BLOB Tutorial	13
Funktionen	18
PG:OpenDatabase	18
PG:CloseDatabase	21
PG:SimpleUpdate	21
PG:ExtendedUpdate	23
PG:SimpleQuery	26
PG:ExtendedQuery	28
PG:ConvertBLOB	31
Interne Fehlercodes	32

Einführung



PostgreSQL for Hollywood ist eine Sammlung an Funktionen, eine Library die es der Programmiersprache Hollywood ermöglicht mit PostgreSQL- Servern zu kommunizieren. Die Library kann einfach per "INCLUDE" in bestehende Projekte eingefügt werden. Alle Variablen sind lokal deklariert, es sollte also zu keinen Konflikten mit bestehendem Programmcode kommen. Der Code ist Cross-OS einsetzbar, getestet habe ich bislang aber nur auf Windows10, Android und AmigaOS3.9(WinUAE).

Mit dem Personal Edition von HelpNDoc erstellt: [HTML-Hilfedokumente einfach erstellen](#)

Anforderungen

- **PostgreSQL for Hollywood** wurde mit Hollywood 8.0 entwickelt. In wie weit der Code abwärtskompatible zu früheren Hollywood-Versionen ist habe ich nicht getestet.
- Zugang zu einem PostgreSQL-Datenbankserver. Getestet habe ich mit PgSQL 10.1 sowie PgSQL 11.2. Die Library sollte jedoch abwärtskompatibel zu älteren PostgreSQL-Versionen sein solange diese die Protokoll-Version 3 beherrschen.
- **Einschränkung:** Es kann nur mit Servern kommuniziert werden die md5 zur Authentifizierung einsetzen. Außerdem ist mit dieser Version der Library noch keine SSL-Verbindung möglich. Solange PgSQL in der Standard-Konfiguration verwendet wird stellen diese Einschränkungen jedoch kein Problem dar.
- Grundkenntnisse der Datenbanksprache SQL.

Mit dem Personal Edition von HelpNDoc erstellt: [Vorteile eines Hilfe-Entwicklungstools](#)

Lizenz

PostgreSQL for Hollywood
Copyright © 2019 Michael Suther
Email: michaelsuther@googlemail.com

1. Ich veröffentliche den Source Code von "**PostgreSQL for Hollywood**" als Open Source unter Ausschluss jeglicher Gewährleistung und Haftung. Sie verwenden den Source Code also auf absolut eigenes Risiko.

2. Sie dürfen den Source Code gerne veröffentlichen wenn er Bestandteil Ihres eigenen Software- Projekts ist. Sie dürfen den Source Code natürlich auch in kommerziellen oder nicht OpenSource Projekten benutzen. Diese Erlaubnis bezieht sich nur auf den Source Code!
3. Es ist nicht erlaubt das Archive oder deren Inhalt zu veröffentlichen, zu verbreiten, beispielsweise als Download. Es gilt die Ausnahme in Punkt 2.

Die Programmiersprache Hollywood ist Eigentum von Andreas Falkenhahn:

<https://www.hollywood-mal.com/index.html>

PostgreSQL ist ein OpenSource Datenbank Projekt. <https://www.postgresql.org/>

PostgreSQL Markenrichtlinien / Trademark Policy :

<https://www.postgresql.org/about/policies/trademarks/>

Mit dem Personal Edition von HelpNDoc erstellt: [Kindle eBooks einfach herstellen](#)

History

Version 1.0 Final (17.05.2019)

- Seit der letzten beta habe ich Programme geschrieben die diese Library testen. Ich konnte sehr viele Bugs finden und beheben. Meist handelte es sich um kleine Fehler wie etwa nicht wieder freigegebene Ressourcen. Insgesamt wurden in den letzten Tagen viele Millionen Datenbank-Abfragen fehlerfrei abgeschlossen. Deswegen hier nun die erste Final-Version.
- Den Bindungsvariablen wurde noch der Type #SHORT hinzugefügt.

Version 1.0 beta 1 (04.05.2019)

- PG:OpenDatabase wurde überarbeitet. Die Netzwerkverbindung wird nun von der Funktion selber geöffnet.
- BLOBs (Binary Large Objects)
- Die Funktionen PG:ExtendedUpdate und PG:ExtendedQuery können nun BLOBs (Binary Large Objects) verarbeiten.
- Die Dokumentation wurde angepasst und erweitert,

Version 1.0 alpha 2 (28.04.2019)

- Funktion PG:ExtendedQuery() hinzugefügt.
- Funktion PG:ExtendedUpdate() hinzugefügt.
- Bindungsvariablen
- Empfangen der Netzwerk-Daten geändert.
- Alle Kommentare im Quellcode und in den Beispielen in englisch verfasst.

Version 1.0 alpha 1 (19.04.2019)

- Erste öffentliche Alpha-Version

- PostgreSQL Nachrichten werden noch nicht vollständig "geparst".
- Es fehlen noch viele Funktionen wie etwa: Bindungsvariablen oder BLOBs
- Einfache Abfragen/Änderungen an der Datenbank sollten funktionieren.

Mit dem Personal Edition von HelpNDoc erstellt: [Gratis PDF-Dokumentationsgenerator](#)

Future

- Eine SSL-Version

Mit dem Personal Edition von HelpNDoc erstellt: [Einfacher CHM- und Dokumentationseditor](#)

Erste Schritte

Einige kleine Beispiele die die Benutzung der Library veranschaulichen.
Wünsche viel Spaß!

Mit dem Personal Edition von HelpNDoc erstellt: [HTML-Hilfedokumente einfach erstellen](#)

Einfaches Tutorial

In diesem kleinen Tutorial werden wir eine Tabelle mit dem Namen "Personen" erstellen und mit dieser arbeiten.

Zunächst binden wir die PostgreSQL-Library mit der Präprozessor-Anweisung **@INCLUDE** ein.

Dieser kleine Code meldet den Benutzer "test" bei der Datenbank "postgres" an.

Bitte passen Sie die Benutzerdaten Ihren Gegebenheiten an.

In diesem Beispiel läuft die Datenbank lokal auf meinem Rechner und lauscht an Port 5432. Sollte die Datenbank nicht auf Ihren Rechner laufen, sondern beispielsweise auf einem anderen Rechner im lokalen Netzwerk, müsse Sie "localhost" gegen dessen IP austauschen.

Dies gilt übrigens für alle weiteren Beispiele!

```

/*****
**
** Name:      Einfaches Tutorial
** Author:    Michael Suther
** Date:      27.04.19
** Interpreter: Hollywood 8.0
** Function:   A simple example
**
**
*****/

@INCLUDE "PostgreSQL_Lib.hws"

Block
    Local Fail, ConnectionID
    Local Errorcode
    Local Host$ = "localhost"
    Local Port = 5432
    Local Database$ = "postgres"
    Local Username$ = "test"
    Local Passwort$ = "test"

    ;registration
    Fail, ErrorCode, ConnectionID = PG:OpenDatabase(Host$, Port,
Database$, Username$, Passwort$)

    ; Evaluating the Return values
    If Fail = False
        NPrint("")
        NPrint("Connection was successfully established.")
        NPrint("-----")
    Else
        NPrint("")
        NPrint("The connection failed.")
        NPrint("")
        NPrint("Error code: ", ErrorCode)
        End
    EndIf
EndBlock

```

Nachdem wir uns erfolgreich bei der Datenbank angemeldet haben erstellen wir unsere erste eigene Tabelle.

Tabelle Personen:

Vorname	Typ: text
Name	Typ: text
Alter	Typ: int4

```

;The SQL query
Local SQL$ = "CREATE TABLE Personen (Vorname text, Name text, Alter
int4)"

;Create a new table
Fail, Errorcode = PG:SimpleUpdate(SQL$, ConnectionID)

; Evaluating the Return values
If Fail = False
    NPrint("")
    NPrint("Table was created.")
    NPrint("")
    NPrint("")
Else
    NPrint("")
    NPrint("Create Table failed.")
    NPrint("")
    NPrint("Error code: ", Errorcode)
EndIf

```

Nun wird es zeit die Tabelle mit Daten zu füllen.
Der folgende Code fügt der Tabelle drei Personen hinzu.

```

;Write data To the table
Fail, Errorcode = PG:SimpleUpdate("INSERT INTO Personen VALUES
('Michael', 'Mustermann', 34)", ConnectionID)
;Evaluating the Return values
If Fail = False
    NPrint("Database update successful - First person added.")
    NPrint("")
Else
    NPrint("INSERT INTO people failed.")
    NPrint("Error code: ", Errorcode)
EndIf
Fail, Errorcode = PG:SimpleUpdate("INSERT INTO Personen VALUES
('Thomas', 'Mustermann', 48)", ConnectionID)
;Evaluating the Return values
If Fail = False
    NPrint("Database update successful - Second person added.")
    NPrint("")
Else
    NPrint("INSERT INTO people failed.")
    NPrint("Error code: ", Errorcode)
EndIf

```

```

    Fail, Errorcode = PG:SimpleUpdate("INSERT INTO Personen VALUES
('Frank', 'Testmann', 27)", ConnectionID)
;Evaluating the Return values
If Fail = False
    NPrint("Database update successful - Third person added.")
    NPrint("")
    NPrint("")
Else
    NPrint("INSERT INTO people failed.")
    NPrint("Error code: ", Errorcode)
EndIf

```

Die folgenden Code-Zeilen beinhalten eine einfache Suche in der Datenbank.
Wir lassen uns alle Personen anzeigen die 48 Jahre alt sind.

```

;Lists all persons 48 years old.
Local SQL$ = "SELECT * FROM Personen WHERE Alter = 48"
Fail, Errorcode, fields, Result = PG:SimpleQuery(SQL$, ConnectionID)

; Evaluating the Return values
If Fail = False
    NPrint("Find entries with people who are 48 years old.")
    NPrint("")
    NPrint("Search result")
    NPrint("")
    items = TableItems(Result)
    For i = 0 To items-1 Step fields
        For a = 1 To fields
            Print(Result[i + a-1], " ")
        Next
        NPrint("")
    Next
Else
    NPrint("Query failed.")
    NPrint("Error code: ", Errorcode)
EndIf

```

Als letztes machen wir alles wieder rückgängig und löschen die Tabelle.
Wenn Sie mit der Tabelle weiter arbeiten möchten, lassen Sie den letzten Schritt einfach aus.

Das gesamte Code-Beispiel finden Sie auch im Archive als Hollywood-Script.

```

;Delete the table
Local SQL$ = "DROP TABLE Personen;"
Fail, Errorcode = PG:SimpleUpdate(SQL$, ConnectionID)

; Evaluating the Return values
If Fail = False
    NPrint("")
    NPrint("")

```



```

        NPrint("Table has been deleted")
    Else
        NPrint("DROP table failed.")
        NPrint("Error code: ", ErrorCode)
    EndIf

    WaitLeftMouse

    ;Close connection To the database.
    PG:CloseDatabase(ConnectionID)

EndBlock

```

Wenn Sie alles richtig gemacht haben, sollte die Programmausgabe wie folgt aussehen (Windows10):



```

Hollywood
Connection was successfully established.
-----
Table was created.
Database update successful - First person added.
Database update successful - Second person added.
Database update successful - Third person added.
Find entries with people who are 40 years old.
Search result
Thomas    Mustermann    40
Table has been deleted

```

Mit dem Personal Edition von HelpNDoc erstellt: [Gratis CHM-Hilfedokumentationsgenerator](#)

Erweitertes Tutorial

In diesem kleinen Tutorial beschreibe ich die Benutzung von SQL-Bindungsvariablen.

Bitte beachten Sie, Bindungsvariablen können in der SQL-Sprache nicht immer oder überall eingesetzt werden!

Okay, los geht es.

Zunächst melden wir uns bei der Datenbank an.
Die Benutzerdaten müssen Sie natürlich Ihren Gegebenheiten anpassen.

```

@INCLUDE "PostgreSQL_Lib.hws"

Block
    Local Fail, fields, SQL$, i, a

```

```

Local Errorcode
Local Host$ = "localhost"
Local Port = 5432
Local Database$ = "postgres"
Local Username$ = "test"
Local Passwort$ = "test"
Local Bind = {}
Local Bind1 = {}
Local Result = {}

;registration
Fail, Errorcode, ConnectionID = PG:OpenDatabase(Host$, Port,
Database$, Username$, Passwort$)

; Evaluating the Return values
If Fail = False
    NPrint("")
    NPrint("Connection was successfully established.")
    NPrint("-----")
Else
    NPrint("")
    NPrint("The connection failed.")
    NPrint("")
    NPrint("Error code: ", Errorcode)
End
EndIf

```

Wie bereits im Tutorial zuvor erstellen wir eine einfache Tabelle.

Tabelle Personen:

Vorname	Typ: text
Name	Typ: text
Alter	Typ: int4

```

;The SQL query
Local SQL$ = "CREATE TABLE Personen (Vorname text, Name text, Alter
int4)"

;Create a new table
Fail, Errorcode = PG:SimpleUpdate(SQL$, ConnectionID)

; Evaluating the Return values
If Fail = False
    NPrint("")
    NPrint("Table was created.")
    NPrint("")
    NPrint("")
Else

```

```

NPrint("")
NPrint("Create Table failed.")
NPrint("")
NPrint("Error code: ", ErrorCode)

EndIf

```

Nun wird es Zeit die Tabelle mit Daten zu füllen.

Der folgende Code fügt der Tabelle drei Personen hinzu.

An dieser Stelle benutzen wir auch das erste mal die Bindungsvariablen.

```

;Create binding variables
InsertItem(Bind, #STRING)           ;SQL variable $1
InsertItem(Bind, "Michael")
InsertItem(Bind, #STRING)           ;SQL variable $2
InsertItem(Bind, "Mustermann")
InsertItem(Bind, #INTEGER)          ;SQL variable $3
InsertItem(Bind, 34)

InsertItem(Bind, #STRING)           ;SQL variable $4
InsertItem(Bind, "Thomas")
InsertItem(Bind, #STRING)           ;SQL variable $5
InsertItem(Bind, "Mustermann")
InsertItem(Bind, #INTEGER)          ;SQL variable $6
InsertItem(Bind, 48)

InsertItem(Bind, #STRING)           ;SQL variable $7
InsertItem(Bind, "Frank")
InsertItem(Bind, #STRING)           ;SQL variable $8
InsertItem(Bind, "Testmann")
InsertItem(Bind, #INTEGER)          ;SQL variable $9
InsertItem(Bind, 27)

;Write data To the table
SQL$ = "INSERT INTO Personen VALUES ($1, $2, $3)"
Fail, ErrorCode = PG:ExtendedUpdate(Bind, SQL$, ConnectionID)
If Fail = False
    NPrint("Database update successful - First person added.")
    NPrint("")
Else
    NPrint("INSERT INTO people failed.")
    NPrint("Error code: ", ErrorCode)
EndIf
SQL$ = "INSERT INTO Personen VALUES ($4, $5, $6)"
Fail, ErrorCode = PG:ExtendedUpdate(Bind, SQL$, ConnectionID)
If Fail = False
    NPrint("Database update successful - Second person added.")
    NPrint("")
Else
    NPrint("INSERT INTO people failed.")
    NPrint("Error code: ", ErrorCode)
EndIf

```

```

SQL$ = "INSERT INTO Personen VALUES ($7, $8, $9)"
Fail, Errorcode = PG:ExtendedUpdate(Bind, SQL$, ConnectionID)
If Fail = False
    NPrint("Database update successful - Third person added.")
    NPrint("")
    NPrint("")
Else
    NPrint("INSERT INTO people failed.")
    NPrint("Error code: ", Errorcode)
EndIf

```

Die folgenden Code-Zeilen beinhalten eine einfache Suche in der Datenbank.
Wir lassen uns alle Personen anzeigen die 48 Jahre alt sind.
Auch hier benutzen wir wieder eine Bindungsvariable.

```

;Lists all persons 48 years old.
InsertItem(Bind1, #INTEGER)      ;SQL variable $1
InsertItem(Bind1, 48)

SQL$ = "SELECT * FROM Personen WHERE Alter = $1"
Fail, Errorcode, fields, Result = PG:ExtendedQuery(Bind1, SQL$,
ConnectionID)

; Evaluating the Return values
If Fail = False
    NPrint("Find entries with people who are 48 years old.")
    NPrint("")
    NPrint("Search result")
    NPrint("")
    items = TableItems(Result)
    For i = 0 To items-1 Step fields
        For a = 1 To fields
            Print(Result[i + a-1], " ")
        Next
        NPrint("")
    Next
Else
    NPrint("Query failed.")
    NPrint("Error code: ", Errorcode)
EndIf

```

Als letztes machen wir alles wieder rückgängig und löschen die Tabelle.
Wenn Sie mit der Tabelle weiter arbeiten möchten, lassen Sie den letzten Schritt einfach aus.

Das gesamte Code-Beispiel finden Sie auch im Archive als Hollywood-Script.

```

;Delete the table
SQL$ = "DROP TABLE Personen;"

```

```

Fail, Errorcode = PG:SimpleUpdate(SQL$, ConnectionID)

; Evaluating the Return values
If Fail = False
    NPrint("")
    NPrint("")
    NPrint("Table has been deleted")
Else
    NPrint("DROP table failed.")
    NPrint("Error code: ", ErrorCode)
EndIf

WaitLeftMouse

;Close connection To the database.
PG:CloseDatabase(ConnectionID)
EndBlock

```

Wenn Sie alles richtig gemacht haben, sollte die Programmausgabe wie folgt aussehen (AmigaOS 3.9):



```

Hollywood
-----
Connection was successfully established.
Table was created.
Database update successful - First person added.
Database update successful - Second person added.
Database update successful - Third person added.
Find entries with people who are 40 years old.
Search result
Thomas Mustermann 40
Table has been deleted

```

Mit dem Personal Edition von HelpNDoc erstellt: [Hilfdateien für das Qt Help-Framework erstellen](#)

BLOB Tutorial

In diesem Tutorial beschreibe ich die Benutzung von BLOBs (Binary Large Objects),

Direkt vorweg, ich bin kein Freund, kein Befürworter von großen Binärdaten in einer Datenbank. Es ist meistens sinnvoller diese auf dem Filesystem zu belassen und in der DB nur einen Verweis darauf zu speichern.

Nun aber los.

Als erstes Verbinden wir uns wieder mit der Datenbank,

```

@INCLUDE "PostgreSQL_Lib.hws"

Block
    Local Fail, fields, SQL$, i, a, f$

```

```

Local Errorcode, BrushID
Local Host$ = "localhost"
Local Port = 5432
Local Database$ = "postgres"
Local Username$ = "test"
Local Passwort$ = "test"
Local Bind = {}
Local Bind1 = {}
Local Result = {}

;registration
Fail, ErrorCode, ConnectionID = PG:OpenDatabase(Host$, Port,
Database$, Username$, Passwort$)

; Evaluating the Return values
If Fail = False
    NPrint("")
    NPrint("Connection was successfully established.")
    NPrint("-----")
Else
    NPrint("")
    NPrint("The connection failed.")
    NPrint("")
    NPrint("Error code: ", ErrorCode)
    End
EndIf

```

Nun laden wir ein Bild, welches wir später in die Datenbank schreiben.

```

;Load an image file
f$ = FileToString("picture.png")

```

Als nächstes erstellen wir eine Tabelle mit folgenden Spalten:

Tabelle Personen:

Vorname	Typ: text
Name	Typ: text
Alter	Typ: int4
Bild	Typ: bytea

Beachten Sie bitte: PostgreSQL benötigt zum speichern von BLOBs eine Spalte vom Typ bytea.

```

;Create a new table
Local SQL$ = "CREATE TABLE Personen (Vorname text, Name text, Alter
int4, Bild bytea)"
Fail, Errorcode = PG:SimpleUpdate(SQL$, ConnectionID)

```

```

; Evaluating the Return values
If Fail = False
    NPrint("")
    NPrint("Table was created.")
    NPrint("")
    NPrint("")
Else
    NPrint("")
    NPrint("Create Table failed.")
    NPrint("")
    NPrint("Error code: ", ErrorCode)
EndIf

```

Nun werden wir der Tabelle eine Zeile hinzufügen.
 Neu im Code unten ist die Bindungsvariable 4 vom Typ #BLOB.
 Der String enthält das komplette Bild in Form von Raw-Data.

```

;Create binding variables
InsertItem(Bind, #STRING);SQL variable $1
InsertItem(Bind, "Michael")
InsertItem(Bind, #STRING);SQL variable $2
InsertItem(Bind, "Mustermann")
InsertItem(Bind, #INTEGER);SQL variable $3
InsertItem(Bind, 48)
InsertItem(Bind, #BLOB) ;SQL variable $4
InsertItem(Bind, f$)

;Write data To the table
SQL$ = "INSERT INTO Personen VALUES ($1, $2, $3, $4)"
Fail, Errorcode = PG:ExtendedUpdate(Bind, SQL$, ConnectionID)
If Fail = False
    NPrint("Database update successful")
    NPrint("")
    NPrint("")
Else
    NPrint("INSERT INTO Personen failed.")
    NPrint("Error code: ", ErrorCode)
EndIf

```

Nun lesen wir die gesamte Zeile wieder aus der Datenbank.
 Wie immer enthält die Tabelle "Result" alle Daten der Zeile wie Vorname, Name, Alter und natürlich auch das Bild.

```

;Read data from the database
SQL$ = "SELECT * FROM Personen;"
Fail, Errorcode, fields, Result = PG:SimpleQuery(SQL$, ConnectionID)
If Fail = False

```

```

        NPrint("")
        NPrint("Data read successfully")
        NPrint("")
    Else
        NPrint("Query failed.")
        NPrint("Error code: ", ErrorCode)
    EndIf

```

Der nun folgende Code bedarf wieder etwas Erklärung.

PostgreSQL sendet das BLOB leider in einem eigenen Format, welches zu validen Binärdaten konvertiert werden muss.

Diese Aufgabe erledigt die Funktion PG:ConvertBLOB.

Im Beispiel unten übergeben wir der Funktion folgende Ausdrücke:

- Der erste Ausdruck (fields) besagt die Anzahl der Spalten.
- Der zweite Ausdruck (column) besagt in welcher Spalte sich das BLOB befindet (beginnend mit 0).
- Der dritte Ausdruck (Result) enthält natürlich die Daten die wir vorhin empfangen haben.

Sollten Sie einmal Tabellen mit mehreren BYTEA-Spalten haben, müssen sie PG:ConvertBLOB für jede Spalte Separat ausführen.

```

;Convert the BLOB data into a usable format
Fail, ErrorCode, Result = PG:ConvertBLOB(fields, 3, Result)
If Fail = False
    NPrint("")
    NPrint("BLOB conversion successful")
    NPrint("")
Else
    NPrint("ConvertBLOB failed")
    NPrint("Error code: ", ErrorCode)
EndIf

```

Nachdem wir das BLOB erfolgreich ins Originalformat konvertiert haben speichern wir es auf der Festplatte.

Aber unter einem anderen Namen, damit wir überprüfen können ob das Bild die Transfers fehlerfrei überstanden hat.

```

;Save image To hard disk
StringToFile(Result[3], "test.png")

```

Als letztes lassen wir uns die gesamten Daten anzeigen und löschen die Tabelle wieder.

```

;Show the whole record
BrushID = LoadBrush(Nil, "test.png")
NPrint("")

```



```

NPrint("Vorname : ", Result[0])
NPrint("Name      : ", Result[1])
NPrint("Alter      : ", Result[2])
DisplayBrush(BrushID, 20,180, {Width = 100, Height = 100})

;Delete the table
SQL$ = "DROP TABLE Personen;"
Fail, Errorcode = PG:SimpleUpdate(SQL$, ConnectionID)
; Evaluating the Return values
If Fail = False
    Locate(0,350)
    NPrint("Table has been deleted")
Else
    NPrint("DROP table failed.")
    NPrint("Error code: ", ErrorCode)
EndIf

WaitLeftMouse

;Close connection To the database.
PG:CloseDatabase(ConnectionID)

```

EndBlock

Die Programmausgabe sollte dann so aussehen:



Das Tutorial sowie mein Kunstwerk "picture.png" befindet sich natürlich auch im Archiv.

Mit dem Personal Edition von HelpNDoc erstellt: [Kindle eBooks mit Leichtigkeit generieren](#)

Funktionen

Auf den folgenden Seiten finden Sie eine detaillierte Beschreibung der Funktionen.

Mit dem Personal Edition von HelpNDoc erstellt: [Gratis Hilfeverfassungswerkzeug](#)

PG:OpenDatabase

Bezeichnung

PG:OpenDatabase -- stellt eine neue Verbindung zur einer PostgreSQL Datenbank bereit.

Übersicht

Fail, ErrorCode, ConnectionID = PG:OpenDatabase(Host\$, Port, Database\$, Username\$, Password\$)

Beschreibung

Die Funktion stelle eine Verbindung zu einer PostgreSQL Datenbank her. Das Argument **Database\$** ist der Name der Datenbank mit welcher Sie sich verbinden möchten. Das Argument **Username\$** gibt an mit welchem PostgreSQL - Usernamen Sie sich anmelden möchten. **Password\$** ist logischer weise das Passwort des Users. **Host\$** ist die Adresse des PgSQL-Servers und **Port** natürlich die Port-Nummer an welchem der Server "lauscht".

Bitte beachten Sie: Diese Library unterstützt aktuell nur die MD5 Authentifikation.

Wenn die Verbindung erfolgreich war ist **Fail = False**, ansonsten bei einem Fehler **True**. Sollte ein Fehler auftreten, enthält **ErrorCode** entweder einen dreistelligen internen Fehlercode oder einen fünfstelligen PostgreSQL Fehlercode.

Die Library internen Fehlercodes finden Sie [hier](#).
Die Fehlercodes der PostgreSQL Datenbank sind [hier](#) beschrieben.

Eingaben

Host\$
Adresse des PgSQL-Servers. Beispiele: "localhost" oder "www.meinedb.de" oder "127.0.0.1"

Port
Der Port an dem Der Server lauscht. PgSQL Standard ist 5432

Database\$
Name der Datenbank

Username\$
PostgreSQL User

Password\$
Passwort des Users

Rückgabewerte

Fail
True wenn ein Fehler aufgetreten ist.
False, also 0 wenn alles funktioniert hat.

ErrorCode
Einen dreistelligen internen Fehlercode oder einen fünfstelligen PostgreSQL Fehlercode.

Ansonsten 0

ConnectionID

Der Netzwerk Identifikator.

Beispiel

```

/*****
**
** Name:          1_OpenDatabase
** Author:        Michael Suther
** Date:          01.05.19
** Interpreter:   Hollywood 8.0
** Function:      Open and close a PostgreSQL DB
**
**
*****/

@INCLUDE "PostgreSQL_Lib.hws"

Block

    Local Fail, ConnectionID
    Local Errorcode
    Local Host$ = "localhost"
    Local Port = 5432
    Local Database$ = "postgres"
    Local Username$ = "test"
    Local Passwort$ = "test"

    ;registration
    Fail, ErrorCode, ConnectionID = PG:OpenDatabase(Host$, Port,
Database$, Username$, Passwort$)

    ; Evaluating the Return values
    If Fail = False
        NPrint("Connection was successfully established.")
    Else
        NPrint("")
        NPrint("The connection failed.")
        NPrint("")
        NPrint("Error code: ", ErrorCode)
    EndIf

    WaitLeftMouse

    ;Close connection To the database.
    PG:CloseDatabase(ConnectionID)

EndBlock

```

PG:CloseDatabase

Bezeichnung

PG:CloseDatabase -- beendet die Verbindung zu Datenbank

Übersicht

PG:CloseDatabase(ConnectionID)

Beschreibung

Diese Funktion meldet den Benutzer von der Datenbank ab und beendet die Netzwerkverbindung die zuvor mit **OpenConnection()** geöffnet wurde.

Eingaben

ConnectionID
ID der Netzwerkverbindung

Beispiel

Siehe [PG:OpenDatabase](#)

PG:SimpleUpdate

Bezeichnung

PG:SimpleUpdate -- Führt ein Update, eine Änderung in der Datenbank aus.

Übersicht

Fail, ErrorCode = PG:SimpleUpdate(SQL\$, ConnectionID)

Beschreibung

PG:SimpleUpdate kann nur Aktionen wie etwa "**DROP TABLE**" oder "**CREATE TABLE**" in der Datenbank ausführen. Es ist nicht möglich damit Daten zu lesen.
Für Abfragen wie "**SELECT**" benutzen Sie bitte [PG:SimpleQuery](#) oder [PG:ExtendedQuery](#).

Eingaben

SQL\$

Dieser String muss die SQL-Abfrage enthalten.

ConnectionID

ID der Netzwerkverbindung

Rückgabewerte

Fail

True wenn ein Fehler aufgetreten ist.**False** wenn alles funktioniert hat.

ErrorCode

Einen dreistelligen internen Fehlercode oder einen fünfstelligen PostgreSQL Fehlercode.
Ansonsten **False**.

Beispiel

Dieser Code erstellt folgende Tabelle:

- Tabellen-Name: Kunde
- Die Spalte "vorname" vom Typ "text"
- Die Spalte "name" vom Typ "text"

```

/*****
**
** Name:      2_DBUpdate_CreateTable
** Author:    Michael Suther
** Date:      01.05.19
** Interpreter: Hollywood 8.0
** Function:   Creates a new table in a PgSQL DB
**
**
*****/

@INCLUDE "PostgreSQL_Lib.hws"

Block
    Local Fail, ConnectionID, SQL$
    Local Errorcode
    Local Host$ = "localhost"
    Local Port = 5432
    Local Database$ = "postgres"
    Local Username$ = "test"
    Local Passwort$ = "test"

    ;registration
    Fail, ErrorCode, ConnectionID = PG:OpenDatabase(Host$, Port,
Database$, Username$, Passwort$)

    ; Evaluating the Return values

```

```

If Fail = False
    NPrint("Connection was successfully established.")
Else
    NPrint("")
    NPrint("The connection failed.")
    NPrint("")
    NPrint("Error code: ", ErrorCode)
    WaitLeftMouse
End
EndIf

;The SQL query
Local SQL$ = "CREATE TABLE Kunde (vorname text, name text)"

;Create a new table
Fail, ErrorCode = PG:SimpleUpdate(SQL$, ConnectionID)

; Evaluating the Return values
If Fail = False
    NPrint("")
    NPrint("Table was created.")
Else
    NPrint("")
    NPrint("Create Table failed.")
    NPrint("")
    NPrint("Error code: ", ErrorCode)
EndIf

WaitLeftMouse

;Close connection To the database.
PG:CloseDatabase(ConnectionID)
End
EndBlock

```

Mit dem Personal Edition von HelpNDoc erstellt: [Funktionsreicher Hilfefgenerator](#)

PG:ExtendedUpdate

Bezeichnung

PG:ExtendedUpdate -- Führt ein Update, eine Änderung in der Datenbank aus unter Verwendung von Bindungsvariablen.

Übersicht

Fail, ErrorCode = PG:ExtendedUpdate(Bind{Table}, SQL\$, ConnectionID)

Beschreibung

PG:ExtendedUpdate kann nur Aktionen wie etwa **"DROP TABLE"** oder **"CREATE TABLE"** in der Datenbank ausführen. Es ist nicht möglich damit Daten zu lesen.

Für Abfragen wie **"SELECT"** benutzen Sie bitte [PG:SimpleQuery](#) oder [PG:ExtendedQuery](#).

Im Gegensatz zu **PG:SimpleUpdate()** verwendet **PG:ExtendedUpdate()** aber Bindungsvariablen, welche vor sogenannten "SQL-Injection" schützen. SQL-Code und Benutzereingaben werden der Datenbank getrennt übergeben.

Bindungsvariablen müssen der Tabelle **Bind** übergeben werden!

Aufbau/Beispiel:

Index 0: Typ	; #String, #Integer, #Double, #Float, #Short oder #BLOB
index 1: Inhalt	; Index 0 und Index 1 entsprechen somit der SQL-Variable \$1
Index 2: Typ	; #String, #Integer, #Double, #Float, #Short oder #BLOB
index 3: Inhalt	; Index 2 und Index 3 entsprechen somit der SQL-Variable \$2

... und immer so weiter, je nachdem wie viele Variablen Sie benötigen.

Eingaben

Bind

Ist eine Tabelle die Ihre Bindungsvariablen enthalten muss.

SQL\$

Dieser String muss die SQL-Abfrage enthalten.

ConnectionID

ID der Netzwerkverbindung

Rückgabewerte

Fail

True wenn ein Fehler aufgetreten ist.

False wenn alles funktioniert hat.

ErrorCode

Einen dreistelligen internen Fehlercode oder einen fünfstelligen PostgreSQL Fehlercode. Ansonsten **False**.

Beispiel

```

/*****
**
** Name:      ExtendedUpdate
** Author:    Michael Suther
** Date:      01.05.19
** Interpreter: Hollywood 8.0
** Function:   Shows the use of a binding variable
**
*****/

```



```

**                                                                    **
*****/

@INCLUDE "PostgreSQL_Lib.hws"

Block
    Local Fail, fields, SQL$, i, a, ConnectionID
    Local Errorcode
    Local Host$ = "localhost"
    Local Port = 5432
    Local Database$ = "postgres"
    Local Username$ = "test"
    Local Passwort$ = "test"
    Local Bind = {}
    Local Result = {}

    ;registration
    Fail, ErrorCode, ConnectionID = PG:OpenDatabase(Host$, Port,
Database$, Username$, Passwort$)

    ; Evaluating the Return values
    If Fail = False
        NPrint("")
        NPrint("Connection was successfully established.")
    Else
        NPrint("")
        NPrint("The connection failed.")
        NPrint("")
        NPrint("Error code: ", ErrorCode)
    EndIf

    ;Create binding variables ($1)
    InsertItem(Bind, #STRING)
    InsertItem(Bind, "Kai")

    ;Write another customer In the database.
    SQL$ = "INSERT INTO Kunde VALUES ($1 , 'Mustertest', 200)"

    Fail, Errorcode = PG:ExtendedUpdate(Bind, SQL$, ConnectionID)

    If Fail = False
        NPrint("Database update successful")
    Else
        NPrint("Database update failed.")
        NPrint("Error code: ", ErrorCode)
    EndIf

    WaitLeftMouse

    ;Close connection To the database.
    PG:CloseDatabase(ConnectionID)
EndBlock

```

PG:SimpleQuery

Bezeichnung

PG:SimpleQuery -- Führt eine SQL-Abfrage aus.

Übersicht

Fail, ErrorCode, fields, Result(Table) = PG:SimpleQuery(SQL\$, ConnectionID)

Beschreibung

Es werden nur Abfragen unterstützt die keine Inhalte der Datenbank verändern. Möchten Sie beispielsweise Einträge einer Tabelle verändern, benutzen Sie bitte [PG:SimpleUpdate](#) oder [PG:ExtendedUpdate](#).

Fields und Result kurz erklärt:

Angenommen fields enthält den Wert 3. Das würde bedeuten das Ihre Abfrage Ergebnisse in drei Spalten brachte.

Result[0], Result[1] und Result[2] sind somit ein Datensatz, also eine Zeile der Tabelle.

Enthält Result noch weitere Einträge würde der zweite Datensatz bei Result[3] beginnen und bei [5] enden. usw. usw....

Eingaben

SQL\$

Dieser String muss die SQL-Abfrage enthalten.

ConnectionID

ID der Netzwerkverbindung

Rückgabewerte

Fail

True wenn ein Fehler aufgetreten ist.

False wenn alles funktioniert hat.

ErrorCode

Einen dreistelligen internen Fehlercode oder einen fünfstelligen PostgreSQL Fehlercode. Ansonsten **False**.

fields

Enthält die Anzahl der Spalten einer Tabelle

Result

Result ist eine Tabelle die alle Ergebnisse der Abfrage enthält. Beginnend beim Index 0

Beispiel

Dieses Beispiel listet alle Kunden welche "Mustermann" heißen.

```

/*****
**
** Name:          5_DBQuery
** Author:        Michael Suther
** Date:          27.04.19
** Interpreter:   Hollywood 8.0
** Function:      Performs a search in the Customer table.
**               Lists all customers named "Mustermann".
**
**
*****/

@INCLUDE "PostgreSQL_Lib.hws"

Block
    Local Fail, ConnectionID, i, a, SQL$
    Local Errorcode
    Local Host$ = "localhost"
    Local Port = 5432
    Local Database$ = "postgres"
    Local Username$ = "test"
    Local Passwort$ = "test"
    Local fields
    Local Result = {}

    ;registration
    Fail, ErrorCode, ConnectionID = PG:OpenDatabase(Host$, Port,
Database$, Username$, Passwort$)

    ; Evaluating the Return values
    If Fail = False
        NPrint("Connection was successfully established.")
        NPrint("")
    Else
        NPrint("")
        NPrint("The connection failed.")
        NPrint("")
        NPrint("Error code: ", ErrorCode)
    EndIf

    ;Show all customers with surname Mustermann
    Local SQL$ = "SELECT * FROM Kunde WHERE name LIKE 'Mustermann'"
    Fail, Errorcode, fields, Result = PG:SimpleQuery(SQL$, ConnectionID)

    ; Evaluating the Return values
    If Fail = False
        items = TableItems(Result)
        For i = 0 To items-1 Step fields

```

```

        For a = 1 To fields
            Print(Result[i + a-1], " ")
        Next
        NPrint("")
    Next
Else
    NPrint("Query failed.")
    NPrint("Error code: ", ErrorCode)
EndIf

WaitLeftMouse

;Close connection To the database.
PG:CloseDatabase(ConnectionID)
End
EndBlock

```

Mit dem Personal Edition von HelpNDoc erstellt: [Gratis Webhilfegenerator](#)

PG:ExtendedQuery

Bezeichnung

PG:ExtendedQuery -- Ermöglicht SQL-Abfragen unter Verwendung von Bindungsvariablen.

Übersicht

Fail, ErrorCode, fields, Result(Table) = PG:ExtendedQuery(Bind{Table}, SQL\$, ConnectionID)

Beschreibung

Wie mit **PG:SimpleQuery()** sind mit dieser Funktion auch nur Abfragen möglich die keine Datenbank-Inhalte verändern.

Möchten Sie beispielsweise Einträge einer Tabelle verändern, benutzen Sie bitte [PG:SimpleUpdate](#) oder PG:[ExtendedUpdate](#).

Im Gegensatz zu **PG:SimpleQuery()** verwendet **PG:ExtendedQuery()** aber Bindungsvariablen, welche vor sogenannten "SQL-Injection" schützen. SQL-Code und Benutzereingaben werden der Datenbank getrennt übergeben.

Bindungsvariablen müssen der Tabelle **Bind** übergeben werden!

Aufbau/Beispiel:

Index 0: Typ	; #String, #Integer, #Double, #Float, #Short oder #BLOB
index 1: Inhalt	; Index 0 und Index 1 entsprechen somit der SQL-Variable \$1
Index 2: Typ	; #String, #Integer, #Double, #Float, #Short oder #BLOB
index 3: Inhalt	; Index 2 und Index 3 entsprechen somit der SQL-Variable \$2

... und immer so weiter, je nachdem wie viele Variablen Sie benötigen.

Fields und Result kurz erklärt:

Angenommen fields enthält den Wert 3. Das würde bedeuten das Ihre Abfrage Ergebnisse in drei Spalten brachte.

Result[0], Result[1] und Result[2] sind somit ein Datensatz, also eine Zeile der Tabelle.

Enthält Result noch weitere Einträge würde der zweite Datensatz bei Result[3] beginnen und bei [5] enden. usw. usw....

Eingaben

Bind

Ist eine Tabelle die Ihre Bindungsvariablen enthalten muss.

SQL\$

Dieser String muss die SQL-Abfrage enthalten.

ConnectionID

ID der Netzwerkverbindung

Rückgabewerte

Fail

True wenn ein Fehler aufgetreten ist.

False wenn alles funktioniert hat.

ErrorCode

Einen dreistelligen internen Fehlercode oder einen fünfstelligen PostgreSQL Fehlercode. Ansonsten **False**.

fields

Enthält die Anzahl der Spalten einer Tabelle

Result

Result ist eine Tabelle die alle Ergebnisse der Abfrage enthält. Beginnend beim Index 0

Beispiel

```

/*****
**
** Name:      1_OpenDatabase
** Author:    Michael Suther
** Date:      27.04.19
** Interpreter: Hollywood 8.0
** Function:   Lists all customers with a specific id.
**
**
**
*****/
@INCLUDE "PostgreSQL_Lib.hws"

```

Block

```

Local Fail, fields, SQL$, i, a
Local Errorcode
Local Host$ = "localhost"
Local Port = 5432
Local Database$ = "postgres"
Local Username$ = "test"
Local Passwort$ = "test"
Local Bind = {}
Local Result = {}

;registration
Fail, Errorcode, ConnectionID = PG:OpenDatabase(Host$, Port,
Database$, Username$, Passwort$)

; Evaluating the Return values
If Fail = False
    NPrint("")
    NPrint("Connection was successfully established.")
Else
    NPrint("")
    NPrint("The connection failed.")
    NPrint("")
    NPrint("Error code: ", Errorcode)
EndIf

;Create binding variables ($1)
InsertItem(Bind, #INTEGER)
InsertItem(Bind, 20)

;Show all customers with ID 20
SQL$ = "SELECT vorname, name FROM Kunde WHERE id=$1"

Fail, Errorcode, fields, Result = PG:ExtendedQuery(Bind, SQL$,
ConnectionID)

If Fail = False
    items = TableItems(Result)
    For i = 0 To items-1 Step fields
        For a = 1 To fields
            Print(Result[i + a-1], " ")
        Next
        NPrint("")
    Next
Else
    NPrint("Query failed.")
    NPrint("Error code: ", Errorcode)
EndIf

WaitLeftMouse

```

```

;Close connection To the database.
PG:CloseDatabase(ConnectionID)
EndBlock

```

Mit dem Personal Edition von HelpNDoc erstellt: [Gratis EPub-Hersteller](#)

PG:ConvertBLOB

Bezeichnung

PG:ConvertBLOB -- Konvertiert ein empfangenes "Binary Large Object" zu validen Binärdaten.

Übersicht

Fail, ErrorCode, Result(Table) = PG:ConvertBlob(fields, column, Result(Table))

Beschreibung

PostgreSQL sendet BLOBs im **bytea HEX-Format** weshalb diese nicht ohne weiteres nutzbar sind. Sie müssen die Daten zuerst konvertieren.

Sollte Ihrer Tabelle mehrere Spalten vom Typ **bytea** haben, so müssen Sie PG:ConvertBLOB für jede Spalte aufrufen.

Bitte beachten Sie:

Im PostgreSQL **bytea HEX-Format** wird ein Byte durch zwei Bytes dargestellt. Deshalb ist zum empfangen eines 5 Megabyte großen Bildes 10 Megabyte freier Speicher nötig, zuzüglich nochmals 5 Megabyte für die Konvertierung.

Das bedeutet:

Zum empfangen eines 5 Megabyte großen BLOB benötigt der Client mindestens 15 Megabyte freien Arbeitsspeicher.

Amiga-Rechner mit wenig Arbeitsspeicher kommen da sicherlich schnell an ihre Grenzen.

Eingaben

fields

Anzahl der Tabellenspalten. Diese erhalten Sie auch von **PG:SimpleQuery** und **PG:ExtendedQuery**

column

Die Nummer der Spalte in welcher sich die Binär-Daten befinden. (Beginnend mit 0)

Result(Table)

Diese Tabelle enthält alle Ergebnisse einer Abfrage. Wird von **PG:SimpleQuery** und **PG:ExtendedQuery** bereitgestellt.

Rückgabewerte

Fail

True wenn ein Fehler aufgetreten ist.
False wenn alles funktioniert hat.

ErrorCode

Einen dreistelligen internen Fehlercode oder einen fünfstelligen PostgreSQL Fehlercode.
 Ansonsten **False**.

Result(Table)

Die Tabelle mit allen Daten und dem konvertierten BLOB:

Beispiel

Bitte schauen Sie sich das [BLOB-Tutorial](#) an.

Mit dem Personal Edition von HelpNDoc erstellt: [Hilfedokumente einfach erstellen](#)

Interne Fehlercodes

Eine Liste der PostgreSQL-Fehlercodes finden Sie [hier](#).

PG:OpenDatabase()

Fehlercode	Bedeutung
100	Verbindungsdaten konnten nicht gesendet werden. (SendData)
101	Authentication Request konnte nicht empfangen werden. (ReceiveData)
102	Authentication Methode ist nicht MD5.
103	Passwort Message konnte nicht gesendet werden. (SendData)
104	Authentication Antwort konnte nicht empfangen werden. (ReceiveData)
900	Fehler Übertragung / Nicht bekannte PostgreSQL-Antwort

PG:SimpleUpdate()

Fehlercode	Bedeutung
105	Simple query konnte nicht gesendet werden. (SendData)
106	Keine Antwort nach erfolgreichem Query. (ReceiveData)
901	Fehler Übertragung / Nicht bekannte PostgreSQL-Antwort

PG:SimpleQuery()

Fehlercode	Bedeutung
107	Simple query konnte nicht gesendet werden. (SendData)
108	Keine Antwort nach erfolgtem Query. (ReceiveData)
902	Fehler Übertragung / Nicht bekannte PostgreSQL-Antwort

PG:ExtendedQuery()

Fehlercode	Bedeutung
109	Query konnte nicht gesendet werden. (SendData)
110	Keine Antwort nach erfolgtem Query. (ReceiveData)
111	Bindungsvariable: Falscher Typ
112	Keine Bindungsvariable angegeben
903	Fehler Übertragung / Nicht bekannte PostgreSQL-Antwort

PG:ExtendedUpdate()

Fehlercode	Bedeutung
113	Query konnte nicht gesendet werden. (SendData)
114	Keine Antwort nach erfolgtem Query. (ReceiveData)
115	Bindungsvariable: Falscher Typ
116	Keine Bindungsvariable angegeben
904	Fehler Übertragung / Nicht bekannte PostgreSQL-Antwort

PG:ConvertBLOB()

Fehlercode	Bedeutung
117	Es wurde keine gültige byta Spalte gefunden.