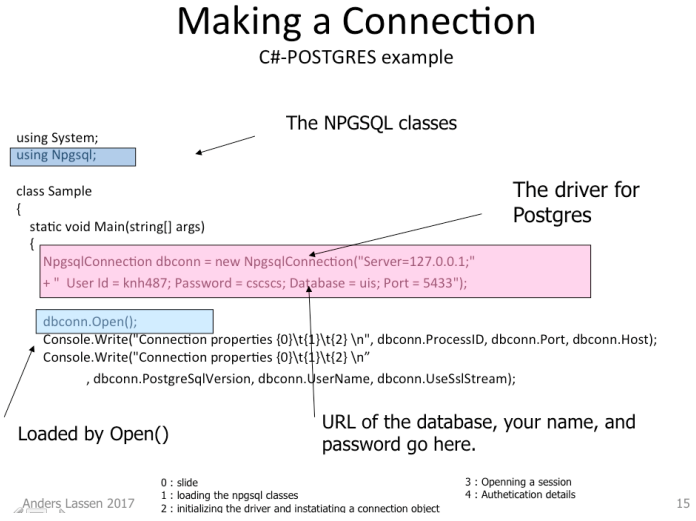
One document comment is that when my laptop is used without external keyboard the komma (','), period ('.'), demicolon, and the colon is replaced with '-', until i make a second pass. Thus, the reader may fill in the missing punctuation.

Making a connection.

A C# (CSHARP) program consist of a number of files. One of these files has a main(). The main is the first method to be called (Illogical but my current understanding). Anyway, the using-directive loads class libraries for the program. Using visual studio or xamarin- program extensions must be loaded or checked as well. The NPGSQL library must be referenced/loaded in order to ude the call-level-interface NPGSQL to access POSTGRES.



Figur slide 15 Making a connection.

Instaitating npgsqlconnection to the variable dbconn is an intialization with authentication. The driver against POSTGRES is implicit, as far as I know it is a POSTGRES specific driver. Authentication as needed, in this example server, userid, password, database and port number is specified in the string.

The open-statement opens the connection and established a session on the server DBMS. The authentication determines the cluster of database objects reachable with the users granted rights. For example a table that is updated through the connection requires that the user has been granted UPDATE on the table. A stored procedure that is to be executed requires that the user has been granted EXECUTE on the stored procedure. And so forth.

The connection remains open until it is closed or timed out. A connection can be reused. In some cases a connection pool is maintained by the driver or web-server whereby a pool of connections can be 'shared'. A website has many parallel request and one way to reduce the reconnection overhead is to maintain a set of connections. JDBC is example of a protocol that can implement connection-pooling[look for some refence I know oracle-web-server-interface has a connection pool-administration at least in 2007].

Commandline input

```
Console.WriteLine("1. Add account.");
Console.WriteLine("Enter choice: ");
string input = Console.ReadLine();
int number;
Int32.TryParse(input, out number);
```

```
int number;
if(!Int32.TryParse(input, out number))
{
    //no, not able to parse, repeat, throw exception, use fallback value?
}
```

```
int input = Convert.ToInt32(Console.ReadLine());
```

```
String input = Console.ReadLine();
int selectedOption;
if(int.TryParse(input, out selectedOption))
{
         switch(selectedOption)
         {
                 case 1:
                          //your code here.
                          break;
                 case 2:
                         //another one.
                         break;
                 //. and so on, default..
         }

}
else
{
        //print error indicating non-numeric input is unsupported or something more meaningful.
}
```

```
Int32 _userInput;
if(Int32.TryParse (Console.Readline(), out _userInput) {// do the stuff on userInput}
```

```
            int balance = 10000;
            int retrieve = 0;
            Console.Write("Hello, write the amount you want to retrieve: ");
            retrieve = Convert.ToInt32(Console.ReadLine());
```

```
int op = 0;
string in = string.Empty;
do
{
      Console.WriteLine("enter choice");
      in = Console.ReadLine();
} while (!int.TryParse(in, out op));
```

```
      Console.WriteLine("1. Add account.");
      Console.WriteLine("Enter choice: ");
      int choice=int.Parse(Console.ReadLine());
```

[user4655759 2015 eller stackoverflow 2015] user4655759 font TheLethalCoder yagohaw Xenon Alex Bell. User input commands in Console application. stackoverflow.com. Various Responses 2014-2015. URL20170326 http://stackoverflow.com/questions/24443827/reading-an-integer-from-user-input.

Example by [White 2013].

```
namespace ConsoleApplicationCSharp1
{
   class Program
   {
      static void Main(string[] args)
      {
            String command;
            Boolean quitNow = false;
            while(!quitNow)
            {
                 command = Console.ReadLine();
                 switch (command)
                 {
                      case "/help":
                         Console.WriteLine("This should be help.");
                          break;

                       case "/version":
                          Console.WriteLine("This should be version.");
                          break;

                          case "/quit":
                             quitNow = true;
                             break;

                          default:
                             Console.WriteLine("Unknown Command " + command);
                             break;
                 }
            }
       }
   }
}
```

[White 2013] Ken White. User input commands in Console application. stackoverflow.com. Response by Ken White 20130802. URL20170326 http://stackoverflow.com/questions/18007246/user-input-commands-in-console-application.

C# i et server environment

En One of the comments to this paragraph is the the author-me is a novice in C# programming. This is the same for many of the students. One advice in programming is to seperate configurational pains from programming solution. There is a lot of learning to get running.

For at lave en connection og på toppen af den en session ind mod en Postgres server fra C#, benyttes benyttes et postgres udvidelsee til kilde-bibliotek 'Npgsql'. Foruden Npgsql benyttes system- fordi der skrives til standard out.

```
using System;
using Npgsql;
```

Using – direktivet benyttes til at referere et klassebibliotek eller definere kode I en 'read only' tilstand. Så her er det klassebibliotekerne system og npgsql som refereres denne programkode.

```
class Sample
{
    static void Main(string[] args)
```

Et C# projekt består af mange filer og en af filerne har en klasse med en main() procedure. Det er den der afvikles først og argumenterne – her args er et array som modtager alle parametre fra et kald miteksempel('her', 'er', 'jeg) vil overføre tre argumenter, 'her', 'er', 'jeg'.

NpgsqlConnection

`NpgsqlConnection` er en klasse som andre klasser nedarver fra. Blandt andet vigtige klasser som begintransaction, createcommand og container. ---Initializes a new instance of the NpgsqlConnection class. public NpgsqlConnection(NpgsqlConnectionStringBuilder builder). Initializes a new instance of NpgsqlConnection with the given strongly-typed connection string. public NpgsqlConnection(string connectionString). Initializes a new instance of NpgsqlConnection with the given connection string. –[npgsql 2017]

[npgsql 2017] states that the default port is port 5432.

--- BeginDbTransaction(IsolationLevel). Begins a database transaction with the specified isolation level. Open(). Opens a database connection with the property settings specified by the ConnectionString. ---[npgsql 2017]

```
// Connect to PostgreSQL database
NpgsqlConnection dbconn = new NpgsqlConnection("Server=127.0.0.1; User Id = your_kuid " + " P
assword = your_postgres_password; Database = uis; Port = 5433");
dbconn.Open();
```

So the first linie instatieaties a connection object and the second line opens a database connection with the registered properties-

```
// Define a query returning a single row result set
NpgsqlCommand query = new NpgsqlCommand("SELECT * FROM public.classes", dbconn);
```

From the npgsqlcommand class a number of classes inherits form this, for example CreateParameter(), public NpgsqlCommand(string cmdText, NpgsqlConnection connection). Initializes a new instance of the NpgsqlCommand class with the text of the query and a NpgsqlConnection.--

Getters and setters DbConnection, DbTransaction, Parameters, Transaction-- getters and setters--

methods

- Cancel() Attempts to cancel the execution of a NpgsqlCommand.
- Clone() Create a new command based on this one.

- CreateDbParameter() Creates a new instance of an System.Data.Common.DbParameter object.
- CreateParameter() Creates a new instance of a NpgsqlParameter object.
- Dispose(Boolean) Releases the resources used by the NpgsqlCommand.
- ExecuteDbDataReader(CommandBehavior) Executes the command text against the connection.
- ExecuteDbDataReaderAsync(CommandBehavior, CancellationToken) Executes the command text against the connection.
- ExecuteNonQuery() Executes a SQL statement against the connection and returns the number of rows affected.
- ExecuteNonQueryAsync(CancellationToken) Asynchronous version of ExecuteNonQuery()
- ExecuteReader() Executes the CommandText against the Connection, and returns an DbDataReader. Remarks Unlike the ADO.NET method which it replaces, this method returns a Npgsql-specific DataReader.
- ExecuteReader(CommandBehavior) Executes the CommandText against the Connection, and returns an DbDataReader using one of the CommandBehavior values.
- ExecuteScalar() Executes the query, and returns the first column of the first row in the result set returned by the query. Extra columns or rows are ignored.
- ExecuteScalarAsync(CancellationToken) Asynchronous version of ExecuteScalar()
- Prepare() Creates a prepared version of the command on a PostgreSQL server.
- Unprepare() Unprepares a command, closing server-side statements associated with it. Note that this only affects commands explicitly prepared with Prepare(), not automatically prepared statements.

```
// Execute the query and obtain a result set
NpgsqlDataReader dr = query.ExecuteReader();
```

By [npgsql 2017b] query execute reader returs a npgsql specific DataReader object. This may be the resultset or the iterator object to navigate up and down.

Class NpgsqlDataReader. Reads a forward-only stream of rows from a data source. [npgsql 2017c]- Forward only means the resultset is not navigable up and down and presumably not updatable. Of classes that inherits (functionalty and clad variables) from npgsqldatareader are – getdata. ReadAsync(), NextResultAsync(), ToString(), GetType().

Of properties noted- The most used is difficult to say- the methos getitem etc are the used ones-

- Depth Gets a value indicating the depth of nesting for the current row. Always returns zero.
- FieldCount Gets the number of columns in the current row.
- HasRows Gets a value that indicates whether this DbDataReader contains one or more rows.
- IsClosed Gets a value indicating whether the data reader is closed.
- Item[Int32] Gets the value of the specified column as an instance of System.Object.
- Item[String] Gets the value of the specified column as an instance of System.Object.
- RecordsAffected Gets the number of rows changed, inserted, or deleted by execution of the SQL statement.
- Statements Returns details about each statement that this reader will or has executed.

Of methods noted-

- Close() Closes the NpgsqlDataReader object.
-
- Dispose(Boolean) Releases the resources used by the NpgsqlDataReader.
- GetBoolean(Int32) Gets the value of the specified column as a Boolean.
- GetBytes(Int32, Int64, Byte[], Int32, Int32) Reads a stream of bytes from the specified column, starting at location indicated by dataOffset, into the buffer, starting at the location indicated by bufferOffset.
- GetChar(Int32) Gets the value of the specified column as a single character.
- GetColumnSchema() Returns schema information for the columns in the current resultset.
- GetDataTypeName(Int32) Gets the data type information for the specified field. This will be the PostgreSQL type name (e.g. int4) as in the pg_type table, not the .NET type (see GetFieldType(Int32) for that).
- GetDate(Int32) Gets the value of the specified column as an NpgsqlDate, Npgsql's provider-specific type for dates.
- GetDecimal(Int32) Gets the value of the specified column as a System.Decimal object.

- GetFloat(Int32) Gets the value of the specified column as a single-precision floating point number.
- GetName(Int32) Gets the name of the column, given the zero-based column ordinal.
- GetOrdinal(String) Gets the column ordinal given the name of the column.
- GetValues(Object[]) Populates an array of objects with the column values of the current row.
- NextResult() Advances the reader to the next result when reading the results of a batch of statements.
- Read() Advances the reader to the next record in a result set.
-

So here are a number of getters all specific to different data types.This is an indicator of the impedance-problem. Every language has its own structures so each element must be converted to the host language data format.

The next observation is that there are getters and there are no setters- using prepared statements both getters and setters are necessary- so this is a readonly resultset iterator.

getFloat, getName, getDate are usual methods.

Read() reads next row- but interestingly nextResult() get the next statement from a multistatement query.

```
while (dr.Read())
Console.Write("{0}\t{1}\t{2} \n", dr[0], dr[1], dr[2]); Console.WriteLine("Press any key to exit.");
Console.ReadKey(); //keep console window open in debug mode
```

We now know that the loop above reads the next row in the current statement and writes to the system-console. The method [] is specified in the documentation as

Item[Int32]

Gets the value of the specified column as an instance of System.Object.

```
public override object this[int ordinal] { get; }


public override object this[string name] { get; }
```

The last line is the function overload of the funtion []() a special function. It has a funtion overload with the string parameter use to name the columns with the attribute name. The exmple above uses an index to address fields.

Conclusion- An updatable resultset operator must be chosen.

Class NpgsqlDataAdapter This class represents an adapter from many commands: select, update, insert and delete to fill System.Data.DataSet[npgsql 2017d]. This looks right

[Khan 2013] explains with ADO net library that ExecuteNonQuery should be used for INSERT UPDATE and DELETE- ExecuteReader should be used for SQL and stored procedures - ExecuteScalar reurns first column of first row and can tolerate a non scalar (meaning one value of a

simple type like a number og a text-

For now updatable resultsets is defered

[Khan 2013] snippets

```
string name = "Mudassar Khan";
string city = "Pune";
string constring = ConfigurationManager.ConnectionStrings["constr"].ConnectionString;
using (SqlConnection con = new SqlConnection(constring))
{
  using (SqlCommand cmd = new SqlCommand("INSERT INTO Persons (Name, City)
VALUES (@Name, @City)", con))
  {
    cmd.CommandType = CommandType.Text;
    cmd.Parameters.AddWithValue("@Name", name);
    cmd.Parameters.AddWithValue("@City", city);
    con.Open();
    int rowsAffected = cmd.ExecuteNonQuery();
    con.Close();
  }
}
```

```
string name = "Mudassar Khan";
string city = "Pune";
string constring = ConfigurationManager.ConnectionStrings["constr"].ConnectionString;
using (SqlConnection con = new SqlConnection(constring))
{
  using (SqlCommand cmd = new SqlCommand("UPDATE Persons SET City = @City
WHERE Name = @Name", con))
  {
    cmd.CommandType = CommandType.Text;
    cmd.Parameters.AddWithValue("@Name", name);
    cmd.Parameters.AddWithValue("@City", city);
    con.Open();
    int rowsAffected = cmd.ExecuteNonQuery();
    con.Close();
  }
}
string name = "Mudassar Khan";
string constring = ConfigurationManager.ConnectionStrings["constr"].ConnectionString;
using (SqlConnection con = new SqlConnection(constring))
{
  using (SqlCommand cmd = new SqlCommand("DELETE FROM Persons WHERE Name =
@Name", con))
  {
    cmd.CommandType = CommandType.Text;
    cmd.Parameters.AddWithValue("@Name", name);
    con.Open();
```

```
    int rowsAffected = cmd.ExecuteNonQuery();
    con.Close();
  }
}
```

What happens when I use ExecuteNonQuery for SELECT statement?

ExecuteNonQuery will work flawlessly for SELECT SQL Query or Stored Procedure but that will simply execute the query and do nothing. Even if you use it you will not throw any error but the Rows Affected will be negative i.e. -1.

```
--
string constring = ConfigurationManager.ConnectionStrings["constr"].ConnectionString;
using (SqlConnection con = new SqlConnection(constring))
{
  using (SqlCommand cmd = new SqlCommand("SELECT * FROM Persons", con))
  {
    cmd.CommandType = CommandType.Text;
    cmd.Parameters.AddWithValue("@Name", name);
    cmd.Parameters.AddWithValue("@City", city);
    con.Open();
    int rowsAffected = cmd.ExecuteNonQuery();
    con.Close();
  }
}
```

```
string name = "Mudassar Khan";
string constring = ConfigurationManager.ConnectionStrings["constr"].ConnectionString;
using (SqlConnection con = new SqlConnection(constring))
{
  using (SqlCommand cmd = new SqlCommand("SELECT City FROM Persons WHERE Name=@Name", con))
  {
    cmd.CommandType = CommandType.Text;
    cmd.Parameters.AddWithValue("@Name", name);
    con.Open();
    object o = cmd.ExecuteScalar();
    if (o != null)
    {
      string city = o.ToString();
    }
    con.Close();
  }
}
```

```
using (SqlConnection con = new SqlConnection(constring))
{
  using (SqlCommand cmd = new SqlCommand("SELECT Name, City FROM Persons WHERE Name=@Name", con))
  {
    cmd.CommandType = CommandType.Text;
    cmd.Parameters.AddWithValue("@Name", name);
    con.Open();
```

```
        object o = cmd.ExecuteScalar();
        if (o != null)
        {
            string city = o.ToString();
        }
        con.Close();
    }
}
```

Can we use ExecuteScalar for INSERT, UPDATE and DELETE Statements?
Yes you can. But since INSERT, UPDATE and DELETE Statements return no value you will not get any value returned from the Query as well as you will not get the Rows Affected like you get in ExecuteNonQuery.
--

```
string name = "Mudassar Khan";
string city = "Pune";
string constring = ConfigurationManager.ConnectionStrings["constr"].ConnectionString;
using (SqlConnection con = new SqlConnection(constring))
{
    using (SqlCommand cmd = new SqlCommand("UPDATE Persons SET City = @City WHERE Name = @Name", con))
    {
        cmd.CommandType = CommandType.Text;
        cmd.Parameters.AddWithValue("@Name", name);
        cmd.Parameters.AddWithValue("@City", city);
        con.Open();
        object o = cmd.ExecuteScalar();
        con.Close();
    }
}
```

ExecuteReader
ExecuteReader is strictly used for fetching records from the SQL Query or Stored Procedure i.e. SELECT Operation.
Example would be fetching Name City for all records in the Person Table

---

```
string constring = ConfigurationManager.ConnectionStrings["constr"].ConnectionString;
using (SqlConnection con = new SqlConnection(constring))
{
    using (SqlCommand cmd = new SqlCommand("SELECT Name, City FROM Persons", con))
    {
        cmd.CommandType = CommandType.Text;
        con.Open();
        SqlDataReader dr = cmd.ExecuteReader();
        while (dr.Read())
        {
            string name = dr["Name"].ToString();
            string city = dr["City"].ToString();
            Response.Write("Name: " + name);
            Response.Write("City: " + city);
        }
        con.Close();
    }
```

```
}
```

[Khan 2013] Mudassar Ahmed Khan. Difference between ExecuteReader ExecuteScalar and ExecuteNonQuery. aspsnippets.com. Posted by Mudassar Ahmed Khan 20130905. https://www.aspsnippets.com/Articles/Difference-between-ExecuteReader-ExecuteScalar-and-ExecuteNonQuery.aspx

[npgsql 2017d] http://www.npgsql.org/api/Npgsql.NpgsqlDataAdapter.html. npgsql.org. url 20170325. npsql site. API documentation. Class NpgsqlDataReader class.

[npgsql 2017c] http://www.npgsql.org/api/Npgsql.NpgsqlDataReader.html url 20170325. npsql site. API documentation. Class NpgsqlDataReader class.

[npgsql 2017b] http://www.npgsql.org/api/Npgsql.NpgsqlCommand.html url 20170325. npsql site. API documentation. NpgsqlCommand class.

[npgsql 2017a] http://www.npgsql.org/api/Npgsql.NpgsqlConnection.html url 20170325. npsql site. API documentation. NpgsqlConnection class.

Contributors npgsql

Current active contributors to Npgsql are:

- Shay Rojansky
- Emil Lenngren
- Francisco Figueiredo Jr.
- Kenji Uno

Past contributors to Npgsql:

- Jon Asher
- Josh Cooley
- Federico Di Gregorio
- Jon Hanna
- Chris Morgan
- Dave Page
- Glen Parker
- Brar Piening
- Hiroshi Saito

sdfas

```
    string sql = "SELECT * FROM tbl_student WHERE studentname = @param";
      NpgsqlCommand command = new NpgsqlCommand(sql,conn);
      command.Parameters.Add("@param", textBox_studentname.Text); //add the parameter into the sql command
```

```
      DataTable dt = new DataTable();
      //load the DataReader object of the command to the DataTable
      dt.Load(NpgsqlDataReader reader = command.ExecuteReader(CommandBehavior.CloseConnection));
      GridView1.DataSource = dt;
```

[stackoverflow 2015] Du Tran- ow to use SQL parameters with NpgsqlDataAdapter? stackoverflow.com. Posted by user2939293. Response by Du Tran. 201502 URL20170325 http://stackoverflow.com/questions/23413359/how-to-use-sql-parameters-with-npgsqldataadapter

This is example of a binded or prepared statement where the parameters are binded after the statementobject is created- The numbering is quite usual

```
 private void UpdateRecord()
      {
            try
            {
                  NpgsqlConnection conn = Connection.getConnection();
                  conn.Open();

                  NpgsqlCommand cmd = new NpgsqlCommand("update info set \"Fname\" = :FirstName, set \"Lname\" = :LastName,
set \"Address\" = :Address," +
                        "set \"City\" = :City, set \"State\" = State, set \"Zip\" = :Zip," +
                        "set \"PhoneNumber\" = :PhoneNumber where \"LicenceNumber\" = '" + LicenseID + "' ;", conn);

                  cmd.Parameters.Add(new NpgsqlParameter("FirstName", NpgsqlTypes.NpgsqlDbType.Text));
                  cmd.Parameters.Add(new NpgsqlParameter("LastName", NpgsqlTypes.NpgsqlDbType.Text));
                  cmd.Parameters.Add(new NpgsqlParameter("Address", NpgsqlTypes.NpgsqlDbType.Text));
                  cmd.Parameters.Add(new NpgsqlParameter("City", NpgsqlTypes.NpgsqlDbType.Text));
                  cmd.Parameters.Add(new NpgsqlParameter("State", NpgsqlTypes.NpgsqlDbType.Text));
                  cmd.Parameters.Add(new NpgsqlParameter("Zip", NpgsqlTypes.NpgsqlDbType.Text));
                  cmd.Parameters.Add(new NpgsqlParameter("PhoneNumber", NpgsqlTypes.NpgsqlDbType.Text));
                  cmd.Parameters[0].Value = txtFirstName.Text;
                  cmd.Parameters[1].Value = txtLastName.Text;
                  cmd.Parameters[2].Value = txtAddress.Text;
                  cmd.Parameters[3].Value = txtCity.Text;
                  cmd.Parameters[4].Value = cboState.Text;
                  cmd.Parameters[5].Value = txtZip.Text;
                  cmd.Parameters[6].Value = mtxtPhoneNumber.Text;

                  cmd.ExecuteNonQuery();
                  conn.Close();
            }

            catch (Exception ex)
            {
                  MessageBox.Show(ex.ToString());
            }
```

My own concluded example of a binded update can seen below- For testing purposes the command is wrapped in a transaction- therefore the code can de rerun any number of times-

```
using System;
using Npgsql;

class Sample
{
```

```
    static void Main(string[] args)
    {
        // Connect to PostgreSQL  database
        //NpgsqlConnection dbconn = new NpgsqlConnection("Server=127.0.0.1; User Id = knh487; " + " Password = fjkghdfkjhg; Databas
e = uis; Port = 5433");
        NpgsqlConnection dbconn = new NpgsqlConnection("Server=127.0.0.1; User Id = postgres; " + " Database = postgres; Port = 543
2");

        dbconn.Open();
        Console.Write("Connection properties {0}\t{1}\t{2} \n", dbconn.ProcessID, dbconn.Port, dbconn.Host);
        Console.Write("Connection properties {0}\t{1}\t{2} \n", dbconn.PostgreSqlVersion, dbconn.UserName, dbconn.UseSslStream);

        // Start a transaction as it is required to work with result sets (cursors) in PostgreSQL
        NpgsqlTransaction tran = dbconn.BeginTransaction();
        //tran.Rollback();


        // Define a query returning a single row result set
        NpgsqlCommand query = new NpgsqlCommand("SELECT * FROM public.classes", dbconn);
        //NpgsqlCommand query = new NpgsqlCommand("SELECT * FROM public.classes", dbconn);
        //NpgsqlCommand query = new NpgsqlCommand("SELECT * FROM classes", dbconn);

        // Execute the query and obtain a result set
        NpgsqlDataReader dr = query.ExecuteReader();
        Console.Write("depth {0}\t recordsaffected {1}\t fieldcount {2} \n", dr.Depth, dr.RecordsAffected, dr.FieldCount);
        Console.Write("depth {0}\t hasRows {1}\t{2} \n", dr.Depth, dr.HasRows, dr.Statements);


        while (dr.Read())
            Console.Write("{0}\t{1}\t{2}\t{3}\t{4} \n", dr[0], dr[1], dr[2], dr[3], dr[4]);

        dr.Close();
        Console.WriteLine("1-Press any key to continue.");
        Console.ReadKey(); //keep console window open in debug mode


        NpgsqlCommand cmdu = new NpgsqlCommand("UPDATE classes SET \"numguns\" = :NumGuns" +
              " WHERE \"class\" = :Class ;", dbconn);
        cmdu.Parameters.Add(new NpgsqlParameter("NumGuns", NpgsqlTypes.NpgsqlDbType.Integer));
        cmdu.Parameters.Add(new NpgsqlParameter("Class", NpgsqlTypes.NpgsqlDbType.Text));
        cmdu.Parameters[0].Value = 6;
        cmdu.Parameters[1].Value = "Kongo";
        cmdu.ExecuteNonQuery();



        NpgsqlCommand query2 = new NpgsqlCommand("SELECT * FROM public.classes", dbconn);
        NpgsqlDataReader dr2 = query2.ExecuteReader();
        while (dr2.Read())
            Console.Write("{0}\t{1}\t{2}\t{3}\t{4} \n", dr2[0], dr2[1], dr2[2], dr2[3], dr2[4]);

        dr2.Close();
        Console.WriteLine("2-Press any key to continue.");
        Console.ReadKey(); //keep console window open in debug mode

        Console.WriteLine("F-Press any key to exit.");
        Console.ReadKey(); //keep console window open in debug mode

        tran.Rollback();
        dbconn.Close();
    }
}
```

An important example below is the use of a stored procedure and referenced cursors defined on the database- This example is used in som application- for example at a work experince at a telephone company in copenhagen the online booking system had a java application using referenced cursors like this throughout this particular system-

I did not make this work the cursor returns the cursor name – Maybee theis example will help (https://www.mail-archive.com/npgsql-hackers@gborg.postgresql.org/msg00020.html)

```
-- Procedure that returns a single result set (cursor)
  CREATE OR REPLACE FUNCTION show_cities() RETURNS refcursor AS $$
   DECLARE
     ref refcursor;                                    -- Declare a cursor variable
   BEGIN
     OPEN ref FOR SELECT city, state FROM cities;   -- Open a cursor
     RETURN ref;                                        -- Return the cursor to the caller
   END;
   $$ LANGUAGE plpgsql;
```

```
using System;
```

```
using System.Data;
using Npgsql;

class Sample
{
    static void Main(string[] args)
    {
            // Connect to a PostgreSQL database
            NpgsqlConnection conn = new NpgsqlConnection("Server=127.0.0.1;User Id=postgres; " +
                    "Password=pwd;Database=postgres;");
            conn.Open();

            // Start a transaction as it is required to work with result sets (cursors) in PostgreSQL
            NpgsqlTransaction tran = conn.BeginTransaction();

            // Define a command to call show_cities() procedure
            NpgsqlCommand command = new NpgsqlCommand("show_cities", conn);
            command.CommandType = CommandType.StoredProcedure;

            // Execute the procedure and obtain a result set
            NpgsqlDataReader dr = command.ExecuteReader();

            // Output rows
            while (dr.Read())
                Console.Write("{0} \t {1} \n", dr[0], dr[1]);

            tran.Commit();
            conn.Close();
        }
    }
```

[sqlines 2017] PostgreSQL and C# - Working with Result Sets - Npgsql .NET Data Provider. www.sqlines.com. URL20170325 http://www.sqlines.com/postgresql/npgsql_cs_result_sets.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using Npgsql;
using NpgsqlTypes;

namespace PostgresCsharpTests
{
    public partial class PostgresCsharpTestsMainForm : Form
    {

        #region Connection

        NpgsqlConnectionStringBuilder   m_ConnectionString  = new NpgsqlConnectionStringBuilder();
        private const string            DBHost              = "localhost";
        private const string            DBName              = "testdb";
        private const string            DBUserName          = "testuser";
        private const string            DBPassword          = "pw";
        private const string            DBTable             = "testtable";
        private NpgsqlDataAdapter       m_DataAdapter       = new NpgsqlDataAdapter();
        private NpgsqlConnection        m_Connection        = null;

        #endregion

        public DataSet m_DataSet = new DataSet();
```

```
        public PostgresCsharpTestsMainForm()
        {
            InitializeComponent();
            InitializeDataConnection();
        }

        private void InitializeDataConnection()
        {
            m_ConnectionString.Clear();
            m_ConnectionString.Host = DBHost;
            m_ConnectionString.Database = DBName;
            m_ConnectionString.UserName = DBUserName;
            m_ConnectionString.Password = DBPassword;
            m_ConnectionString.PreloadReader = true;
            m_ConnectionString.UseExtendedTypes = false;

            m_Connection = new NpgsqlConnection(m_ConnectionString);

            SetupDataConnection();
        }

        private void SetupDataConnection()
        {
            m_Connection.Open();
            m_DataSet.Reset();
            string sQuery = "SELECT * FROM " + DBTable;
            m_DataAdapter = new NpgsqlDataAdapter(sQuery, m_Connection);
            m_DataAdapter.Fill(m_DataSet, DBTable);
            this.MainFormDataGridView.DataSource = m_DataSet;
            this.MainFormDataGridView.DataMember = DBTable;
            m_Connection.Close();
        }

        private int SubmitDataSetChanges()
        {
            m_Connection.Open();
            NpgsqlCommandBuilder cb = new NpgsqlCommandBuilder(m_DataAdapter as NpgsqlDataAdapter);
            m_DataAdapter.InsertCommand = cb.GetInsertCommand();
            m_DataAdapter.DeleteCommand = cb.GetDeleteCommand();
            m_DataAdapter.UpdateCommand = cb.GetUpdateCommand();
            DataTable table = m_DataSet.Tables[DBTable];
            int response = m_DataAdapter.Update(table);
            m_Connection.Close();
            return response;
        }

        private void MainFrom_Button_SubmitChanges_Click(object sender, EventArgs e)
        {
            SubmitDataSetChanges();
        }

    }
}
```

[github 2014] HNpgsqlDataAdapter.Update on DataTable throws InvalidCastException if table contains a row of extended- github.com- Posted by digital-void 201409

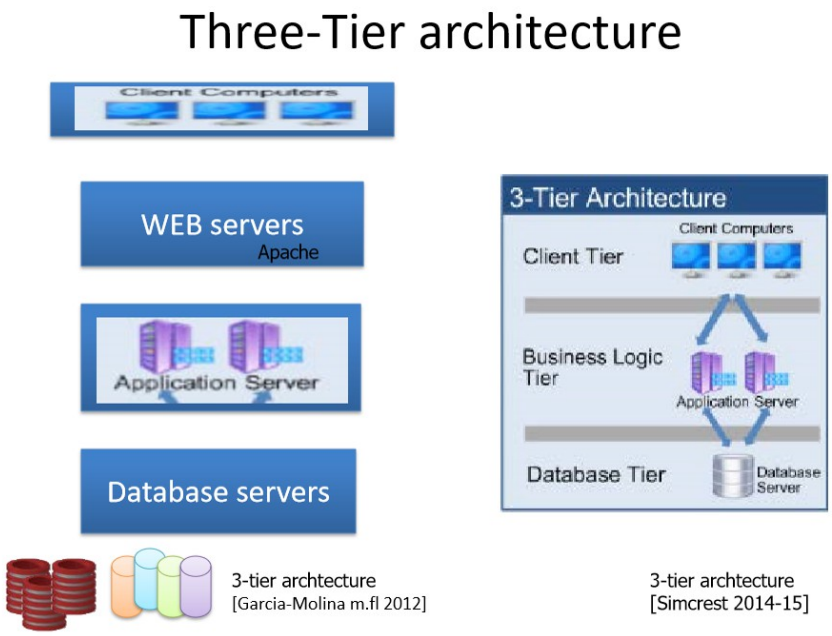
URL20170325 https://github.com/npgsql/npgsql/issues/352

Three tier architecture

Kodning I et server environment er et miljø med mange teknologier, mange databaser. Mange applikationer, sikkerhedlag. Interne applikationer uden en webadgang / uden en flerlagsadgang – altså direkte mod databasen kaldet client-server. Når programmering I et server-environment skal går igennem flere lag af maskiner og arkitektur er en af modellerne som ofte vises three-tier-architecture (da. Tre-lags-arkitektur). Der vises to eksempler I Figur slide three-tier-architecture.

Three-tier-architecture består grundlæggende nederst af et databaselag hvor der er tilgang til databasen samt alle de procedurer som er lagret I datbasen for kaldes persitent-stored-modules (da permanente/lagrede database-moduler). Både database kode, men også kompilerede java eller anden højniveau kode kan lagres på en database som funktionalitet. Database laget er hele domænets maskinpark af databaser som kan tilgåes fra applikationslaget.

Det mellemste lag af three-tier-architecture er applikationslaget. Det er direkte på webserver, altså den webserver man tilgår gennem sikkerhedslaget, hvor apache web-serveren (fra Apache foundation....) har haft stor udbredelse fordi har været opensource og nemt tilgængelig fra unix-linux-osx med videre. Og også derfor er webserveren I figuren til venstre medtaget med annoteringen 'apache'. Men applikationslaget er hele domænets maskinpark af applikationsservere.



Figur slide three-tier-architecture. Der er flere modeller for beskrivelse af three-tiearchitecture. Til venstre med et fler-lag på applikationslaget.

Det øverste lag I three-tier-arkitekturen er clientlaget, som er domænet af alle maskiner og sine browsere som kan tilgå applikationslaget på indenfor eller udenfor en eventuel sikkerheds server (en. Firewall) men som tilgår webserveren og derigennem applikationslaget.

Populært sagt ligger filerne på applikationslaget og derfra tilgåes database laget. Filerne på applikationslaget sender HTML, CSSS, XML, javascript og meget mere ud til clientlaget I en typisk web-applikation.