Currying Connections

These are some of the techniques used to connect our various databases at work. This approach is taken because there was a notorious connection leak issue that brought down our 5-college database servers during registration when there is moderate traffic on the portal. Additionally, it simplifies and eliminates thousands of lines of code in our code base.

Why Curry?

Currying allows us to combine together expressions and execute them at one time.

In our code base, there appears to be 4 common patterns when doing database-related operations. **Scalar**, which executes a command and returns the first column of the first row of the query. **Non-query**, which returns the rows affected, typically, this is executed for updates or deletes. **Rows/Row**, here we expect rows to be returned, the former approach was to use the data adapter from ADO.Net to fill the the dataset/data table and acquire data from it.

Here is a typical flow of how a database connection runs.

1. **Scalar**: Connect to DB -> Execute SQL Cmd -> Supply Needed Parameters -> Execute Scalar -> Acquire First Column of First Row or Null
2. **Non-Query**: Connect to DB -> Execute SQL Cmd -> Supply Needed Parameters -> Execute Non-query (updates, deletes, ect) -> Rows Affected
3. **Ds**: Connect to DB -> Execute SQL Cmd -> Supply Needed Parameters -> Use ADO API's to fill data rows -> Return DataSet
4. **Ds**: Connect to DB -> Execute SQL Cmd -> Supply Needed Parameters -> Iterate through Reader -> Return Dictionary<string, object>
5. **Row**: Same as *Rows*, but we only use one.

With currying, there is a simple way to handle these operations and guarantee no connection leaks.

Take, for instance, an example of a sql stored procedure execution with a parameter input of uid and getting back the row for that user.

Here is the the curried version . . .

```
Db
  .Jics
  .Proc("spGetFwkUser")
  .Param("@uid", userid)
  .Row();
```

to the non-curried version . . .

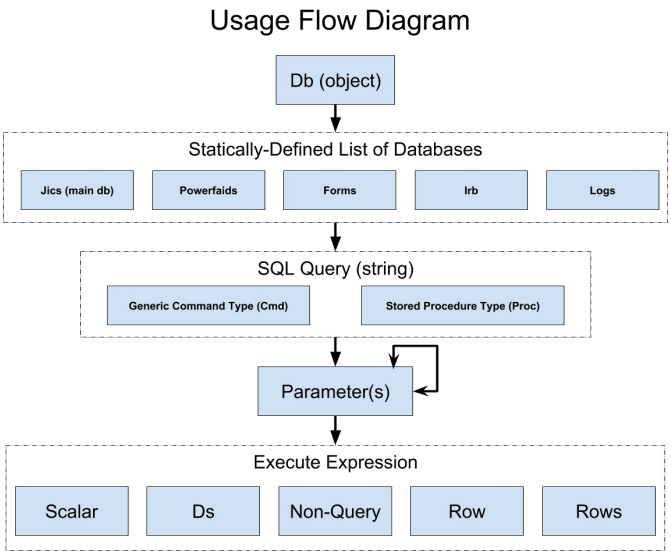
```
using (var conn = new SqlConnection(<... conn for jics db ...>))
{
    using(var proc = conn.CreateCommand())
    {

      proc.CommandType = CommandType.StoredProcedure;
      proc.CommandText = "spGetFwkUser";
      proc.Parameters.Add("@uid", uid);

      conn.Open();

      var rdr = proc.ExecuteReader();

      if (!rdr.HasRows)
          return null;
      else
      {
          var user = new FwkUsers();
          while (rdr.Read())
          {
              // parse
              // and
              // extract
              // data
              // . . .
          }
          return user;
      }

    }
}
```

We use .Rows() which returns a Dictionary<string, object> that represents a row (where the key is the column's name--always of type string, the value of that column and is whatever object it is). In many cases, this allows us to forward the data model straight to the front-end as json.

In this example, the non-curried version was fairly clean and the connection is guaranteed to close. However, when dealing with very bad code where the connection is not closed, the connection is not properly wrapped around a using clause, and the logic continues for 90 or more lines, the logic becomes difficult to keep track of. Unfortunately, our codebase had many instances of this behavior. We used Currying to simplify the code and guaranteed that the database connections were managed properly.

Flow

The following is how this database currying will flow.



1. The expression starts off by calling **Db** which has a statically defined list of database connections.
2. Choose between the list of statically defined databases: **Jics** (our main db), **Powerfaids** (finnancial aid), **Forms** (our generic form solution database), **Irb** (inter-college science form information), **Logs** (trace and log information)
3. Input the SQL query as a string, choose between **Cmd** (a generic SQL query) or **Proc** (a SQL stored procedure).
4. Supply the appropriate parameter(s) to the prepared statement from step 3.
5. Execute the expression and get data back. **Scalar** returns first column of first row as an object. **Ds** returns a DataSet, an ADO.Net object. **Non-Query** this will return the rows affected by the call. **Row** this will return a Dictionary<string, object> (like Json) where the key is the column name and the value is the object; returns null when no rows are available. **Rows** similar to row, except it will return a array of Dictionary<string, object> object.