Calculated Fields in Data Controllers

Data Aquarium Framework dynamically parses *SELECT* statements in commands written in SQL.

The parsing engine is very simplistic and expects a single *SELECT* statement with *FROM*, *WHERE*, and *ORDER BY* clauses. The last two clauses of *SELECT* statement are optional. You can create such commands with any query builder or rely on the statements generated automatically by Code OnTime Generator.

The engine is trying to identify all fields that are available in the statement, their aliases, the table name in the *FROM* clause, the filtering condition and the sort expression. You can include multiple *JOIN* expressions to de-normalize your data and produce a data set that truly represents your data objects to hide the complexity of the normalized database schema.

The parsed information is used to dynamically create SQL statements capable of paging and sorting of very large data sets. It is also used to generate *UPDATE*, *INSERT*, and *DELETE* statements.

Data controller requires that all fields that are displayed in views are enumerated in */dataController/fields/field* node. Field list provides information about data types, primary keys, support for update and the need to have a non-empty value in the fields. All of these field properties are assisting the framework in generating dynamic SQL statements.

Let's consider a few methods of introducing calculated fields that can be used by the framework to provide reach data presentation.

METHOD 1: SIMPLE CALCULATED FIELDS

Here is sample command for *Employees* table in *Northwind* database.

```
    <command id="command1" type="Text">
      <text><![CDATA[
select
    "Employees"."EmployeeID" "EmployeeID"
    ,"Employees"."LastName" "LastName"
    ,"Employees"."FirstName" "FirstName"
    ,("Employees"."LastName" + ', ' + "Employees"."FirstName") "FullName"
    ,"Employees"."Title" "Title"
    ,"Employees"."TitleOfCourtesy" "TitleOfCourtesy"
    ,"Employees"."BirthDate" "BirthDate"
    ,"Employees"."HireDate" "HireDate"
    ,"Employees"."Address" "Address"
    ,"Employees"."City" "City"
    ,"Employees"."Region" "Region"
    ,"Employees"."PostalCode" "PostalCode"
    ,"Employees"."Country" "Country"
    ,"Employees"."HomePhone" "HomePhone"
    ,"Employees"."Extension" "Extension"
    ,"Employees"."Photo" "Photo"
    ,"Employees"."Notes" "Notes"
    ,"Employees"."ReportsTo" "ReportsTo"
    ,"ReportsTo"."LastName" "ReportsToLastName"
    ,"Employees"."PhotoPath" "PhotoPath"
from "dbo"."Employees" "Employees"
    left join "dbo"."Employees" "ReportsTo" on
        "Employees"."ReportsTo" = "ReportsTo"."EmployeeID"
]]></text>
```

This is the list of corresponding fields.

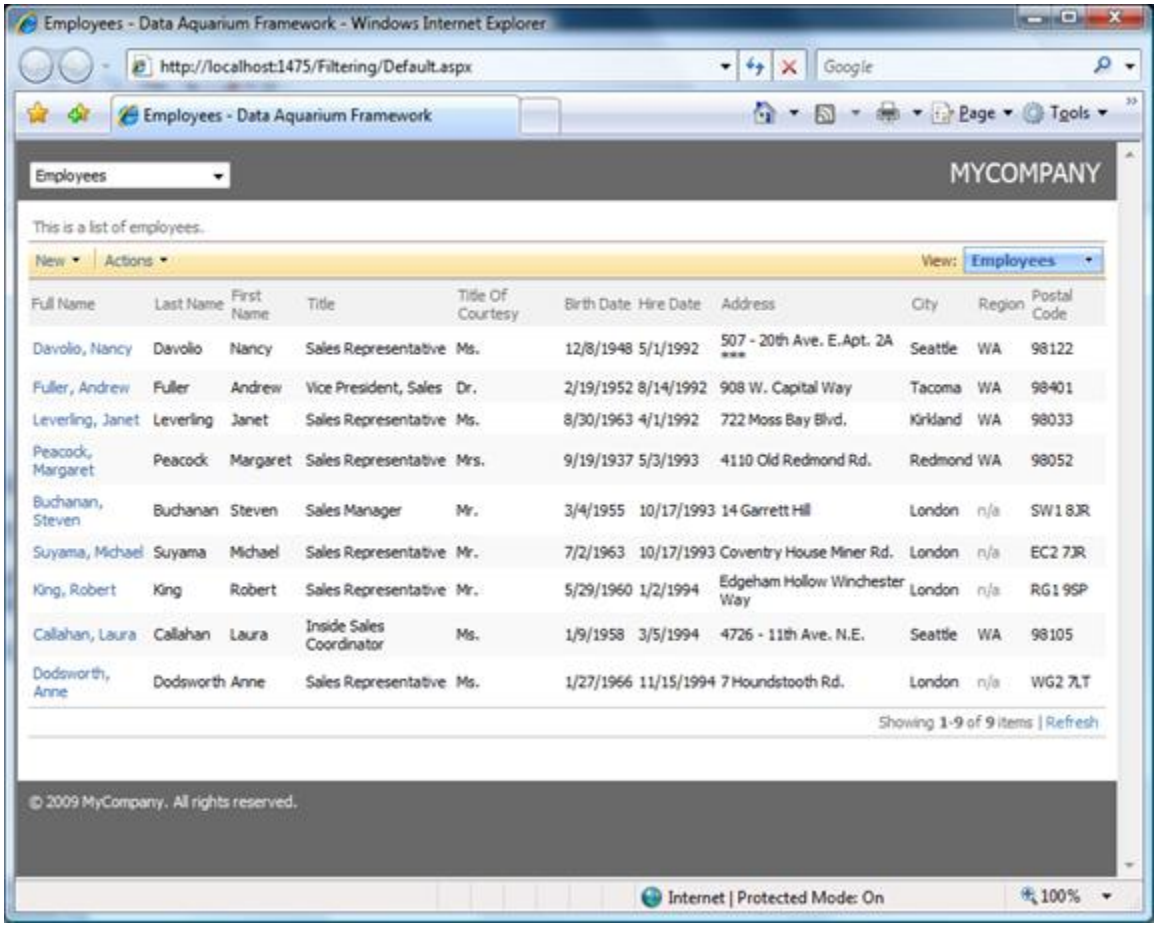
```
<fields>
  <field name="EmployeeID" type="Int32" allowNulls="false"
isPrimaryKey="true"
        label="Employee#" readOnly="true" />
```

```
  <field name="FullName" type="String" allowNulls="true" label="Full Name"
        readOnly="true"/>
  <field name="LastName" type="String" allowNulls="false" label="Last Name"
/>
  <field name="FirstName" type="String" allowNulls="false" label="First Name"
/>
  <field name="Title" type="String" label="Title" />
  <field name="TitleOfCourtesy" type="String" label="Title Of Courtesy" />
  <field name="BirthDate" type="DateTime" label="Birth Date" />
  <field name="HireDate" type="DateTime" label="Hire Date" />
  <field name="Address" type="String" label="Address" />
  <field name="City" type="String" label="City" />
  <field name="Region" type="String" label="Region" />
  <field name="PostalCode" type="String" label="Postal Code" />
  <field name="Country" type="String" label="Country" />
  <field name="HomePhone" type="String" label="Home Phone" />
  <field name="Extension" type="String" label="Extension" />
  <field name="Photo" type="Byte[]" onDemand="true" sourceFields="EmployeeID"
        onDemandHandler="EmployeesPhoto" onDemandStyle="Thumbnail"
allowQBE="false"
        allowSorting="false" label="Photo" />
  <field name="Notes" type="String" allowQBE="false" allowSorting="false"
label="Notes" />
  <field name="ReportsTo" type="Int32" label="Reports To">
    <items style="Lookup" dataController="Employees"
newDataView="createForm1" />
  </field>
  <field name="ReportsToLastName" type="String" readOnly="true"
        label="Reports To Last Name" />
  <field name="PhotoPath" type="String" label="Photo Path" />
</fields>
```

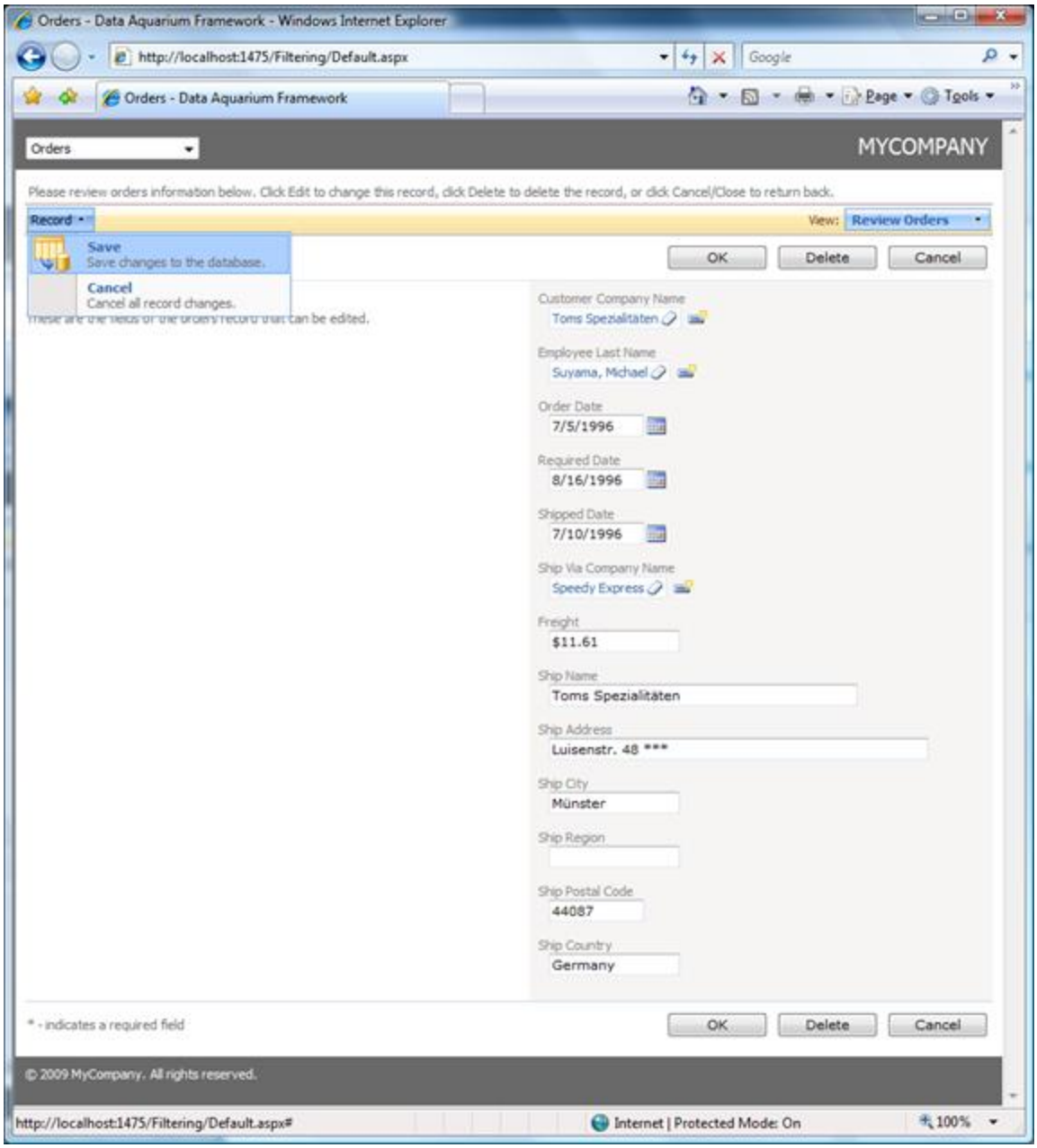
These definitions were automatically produced by Code OnTime Generator but you should have little difficulty in making your own changes when needed.

You have probably noticed that *FullName* field was entered manually in SQL text and in the list of fields. The value of this field is a composition of *LastName* and *FirstName*.

This is how the field looks when displayed in a browser.



The same field is immediately displayed when you select an employee in the lookup field. Notice the value *Employee Last Name* field.



Similar modifications can be done to other related data controllers.

The calculated expressions must remain simple. Otherwise you are risking to confuse the powerful but rather simple-minded *SQL* parser of the framework. High performance comes at a price!

METHOD 2: USE DATA VIEWS FOR COMPLEX CALCULATIONS

Complex formulas can be easily hidden in the views.

For example, create the SQL view as shown in example.

```
create view EmployeeView as
    select
        EmployeeID,
        LastName + ', ' + FirstName as FullName
    from
        Employees
```

Modify the command from the previous sample to look like the one below. Notice the changes in field *FullName* and joined view *EmployeeView*.

```
    <command id="command1" type="Text">
      <text><![CDATA[
select
    "Employees"."EmployeeID" "EmployeeID"
    ,"Employees"."LastName" "LastName"
    ,"Employees"."FirstName" "FirstName"
    ,EmployeeView.FullName
    ,"Employees"."Title" "Title"
    ,"Employees"."TitleOfCourtesy" "TitleOfCourtesy"
    ,"Employees"."BirthDate" "BirthDate"
    ,"Employees"."HireDate" "HireDate"
    ,"Employees"."Address" "Address"
    ,"Employees"."City" "City"
    ,"Employees"."Region" "Region"
    ,"Employees"."PostalCode" "PostalCode"
    ,"Employees"."Country" "Country"
    ,"Employees"."HomePhone" "HomePhone"
    ,"Employees"."Extension" "Extension"
    ,"Employees"."Photo" "Photo"
    ,"Employees"."Notes" "Notes"
```

```
    ,"Employees"."ReportsTo" "ReportsTo"
    ,"ReportsTo"."LastName" "ReportsToLastName"
    ,"Employees"."PhotoPath" "PhotoPath"
from "dbo"."Employees" "Employees"
    left join "dbo"."Employees" "ReportsTo" on
        "Employees"."ReportsTo" = "ReportsTo"."EmployeeID"
    inner join EmployeeView on
        Employees.EmployeeID = EmployeeView.EmployeeID
]]></text>
    </command>
```

Run the program and observe that the result is identical.

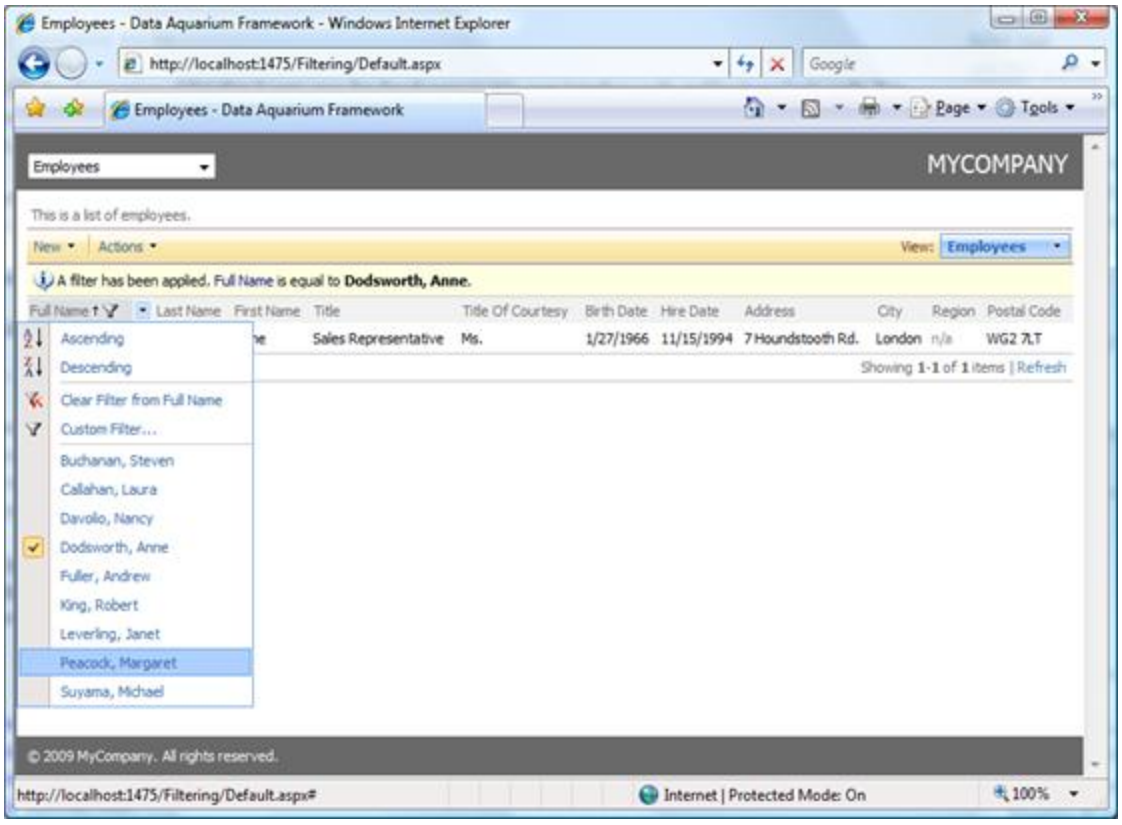
Additional views can be linked to the main (updatable) table in the *FROM* clause and allow use of calculated values. This approach is preferred to the previous one since it brings the business logic into the database where it belongs.

METHOD 3: THE POWER OF ROWBUILDER

Business rules provide a complete control over the field value calculation when you need it. You can find an example of *RowBuilder* attribute usage in business rules at http://blog.codeontime.com/2009/02/business-rules-rowbuilder-attribute.html.

Everything has a price though. High performance user-defined sorting and filtering is supported in the framework via dynamic *SQL* statements. Calculated values provided by business rules are not a part of SQL operations and thus make it impossible to have these great features in the user interface.

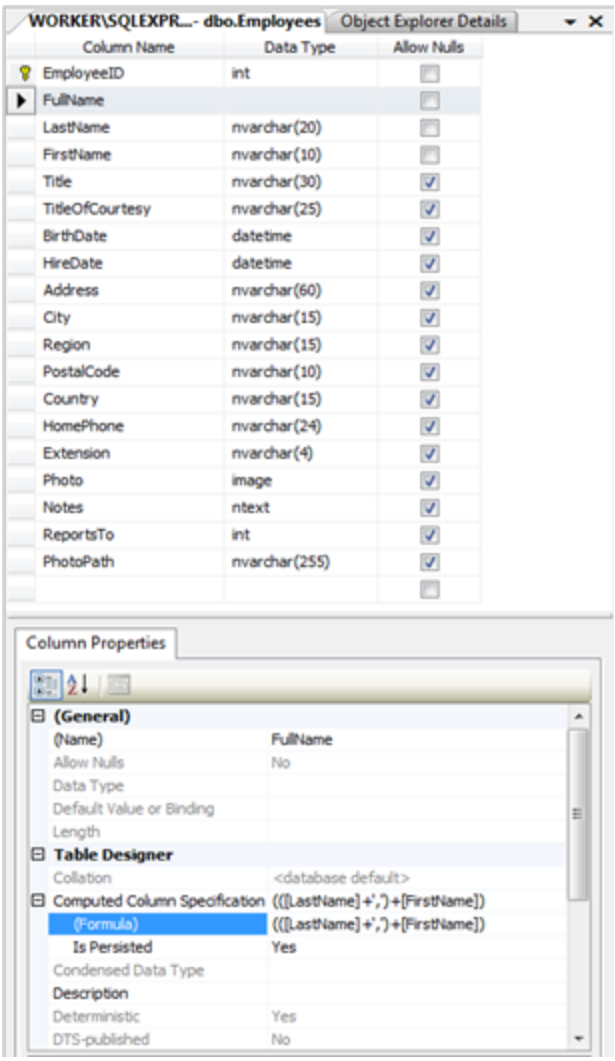
The previous two methods do provide these capabilities as you can see here.



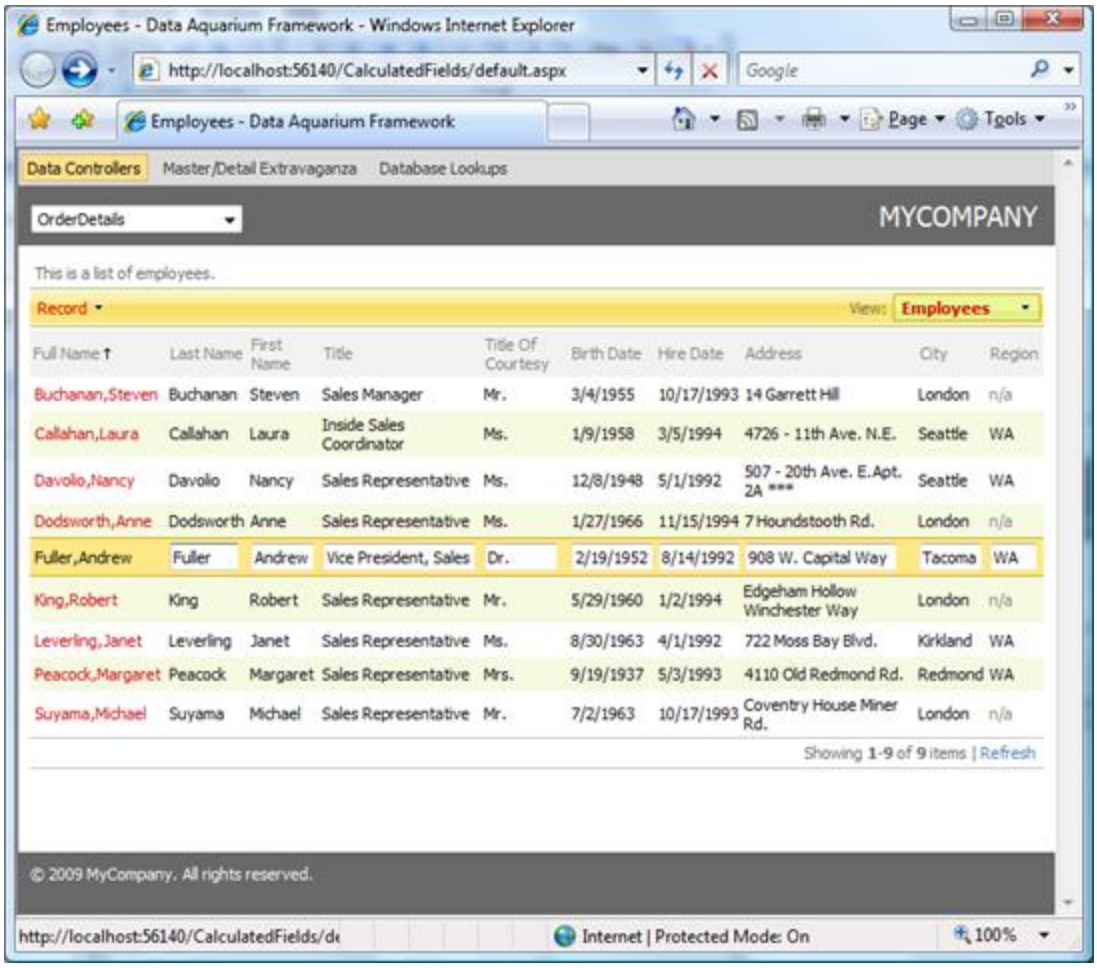
Fields calculated by methods marked with *RowBuilder* attribute will not sort or filter.

METHOD 4: CALCULATED DATABASE FIELDS.

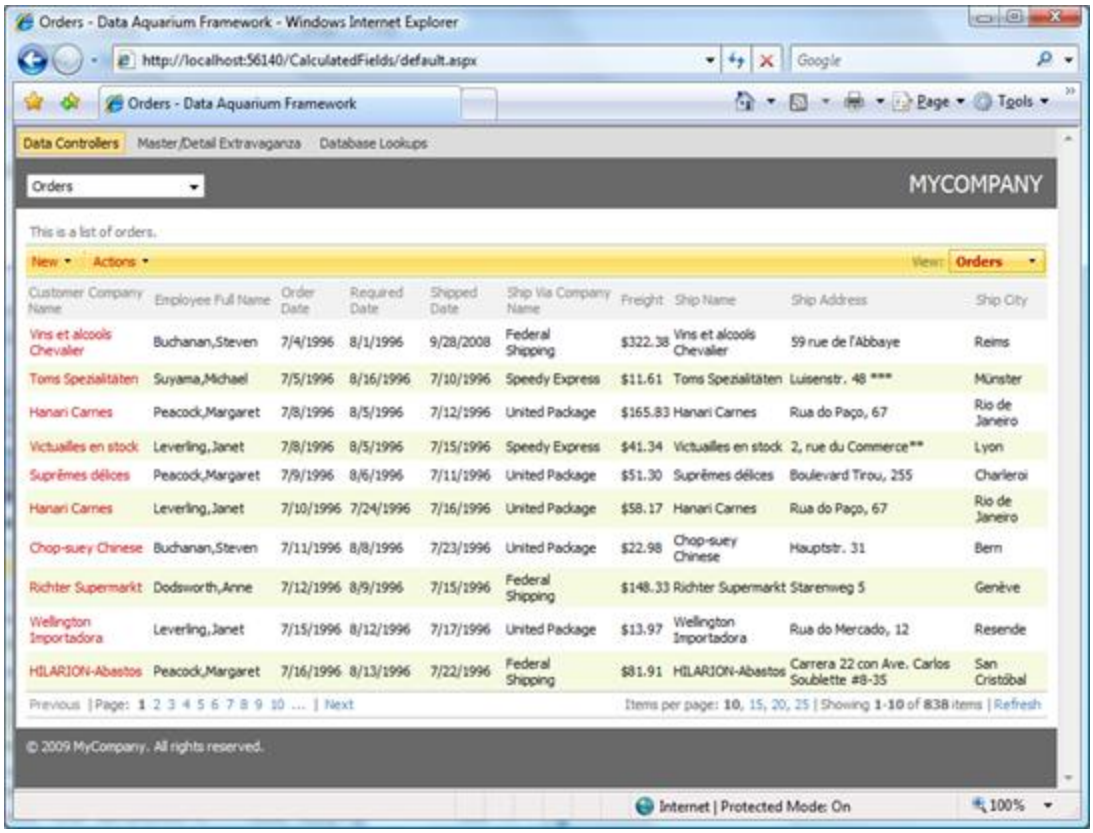
Modern database servers support server-side calculated fields to provide maximum efficiency. For example, *Microsoft SQL Server* supports computed fields. The screen shot below shows the computed field *FullName* as presented in *SQL Management Studio*.



Here is the *Northwind* sample that shows the computed field in action. Notice that the field is automatically recognized by Code OnTime Generator and marked as read-only.



The advantage of server-side calculated fields is that such fields are also recognized in any related objects. Here is the order management view that shows the computed *Employee Full Name* field. You can view sort, filter, lookup, and do any other user interface operation that you can do with any other standard field.



METHOD 5: CALCULATE STORED VALUES WITH BUSINESS RULES

You may choose to create placeholder database fields and calculate the value of such fields with the help of business rules by marking methods with *ControllerAction* attribute.

The placeholder calculated fields are stored in your database but are not designed to be changed by users. Make sure to mark such fields as read-only.

This method provides you with enough control to do just about anything when calculating the values and preserves your ability to allow user-defined sorting and filtering. You can augment your calculations with stored procedures, business logic written in other languages, call external web services… The list can go on.

Read about *ControllerAction* attribute at http://blog.codeontime.com/2009/02/business-rules-controlleraction.html.

CONCLUSION

Numerous options to introduce calculated values are supported in Data Aquarium Framework applications. Methods 4 and 5 are most flexible. Use server-side calculated/computed fields defined as *SQL* expressions or create physical placeholder fields and develop custom business rules with the power of *.NET Framework*.

Code OnTime LLC

http://www.codeontime.com