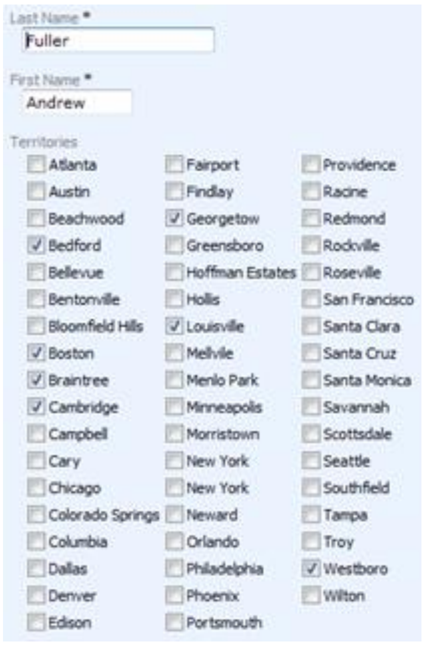
Business Rules: ControllerAction Attribute

We continue discussion of business rules started with posts about RowBuilder attribute. We will create a method that will handle processing of employee territories that were check-marked by user in *editForm1*.



Last Name *
Fuller

First Name *
Andrew

Territories

☐ Atlanta	☐ Fairport	☐ Providence
☐ Austin	☐ Findlay	☐ Racine
☐ Beachwood	☑ Georgetow	☐ Redmond
☑ Bedford	☐ Greensboro	☐ Rockville
☐ Bellevue	☐ Hoffman Estates	☐ Roseville
☐ Bentonville	☐ Hollis	☐ San Francisco
☐ Bloomfield Hills	☑ Louisville	☐ Santa Clara
☑ Boston	☐ Mellvile	☐ Santa Cruz
☑ Braintree	☐ Menlo Park	☐ Santa Monica
☑ Cambridge	☐ Minneapolis	☐ Savannah
☐ Campbell	☐ Morristown	☐ Scottsdale
☐ Cary	☐ New York	☐ Seattle
☐ Chicago	☐ New York	☐ Southfield
☐ Colorado Springs	☐ Neward	☐ Tampa
☐ Columbia	☐ Orlando	☐ Troy
☐ Dallas	☐ Philadelphia	☑ Westboro
☐ Denver	☐ Phoenix	☐ Wilton
☐ Edison	☐ Portsmouth	

Field *Territories* is a placeholder field of *String* type. Business rule method *PrepareExistingEmployeeRow()* retrieves a list of territories that an employee is responsible for.

Now it is time to implement saving of new checked territories and deletion of unchecked ones.

HANDLING OF CUSTOM FIELDS

We will need two methods in our business rule class to process *Territories* field value.

First method will hide that fact of changes to *Territories* field value from Data Aquarium Framework.

C#:

```
[ControllerAction("Employees", "editForm1", "Update", ActionPhase.Before)]
protected void BeforeEmployeeUpdate(int employeeId,
    FieldValue territories, FieldValue lastName)
{
    territories.Modified = false;
    lastName.Modified = true;
}
```

VB:

```
<ControllerAction("Employees", "editForm1", "Update", ActionPhase.Before)> _
Protected Sub BeforeEmployeeUpdate(ByVal employeeId As Integer, _
    ByVal territories As FieldValue, ByVal lastName As FieldValue)
    territories.Modified = False
    lastName.Modified = True
End Sub
```

The name of the method implies that we want this method to execute just before an employee update. But method name means nothing to the framework. Data Aquarium Framework relies on attribute *ControllerAction* to decide if it has to invoke a business rule method.

Our *ControllerAction* attribute specifies that method *BeforeEmployeeUpdate()* shall execute when command *Update* is issued in form *editForm1* of data controller *Employees*. There is also an execution phase information. We want this method to be invoked *before* the framework tries to perform an update.

There are three parameters: *employeeId*, *territories*, and *lastName*.

Data Aquarium Framework assumes that parameter names are matching the values of data fields specified for *editForm1* in the data controller descriptor *~/Controllers/Employees.xml*.

The native data field value information is held in *FieldValue* class instances. If you specify a parameter of type *FieldValue* then you gain access to its properties *OldValue*, *NewValue, Value*, and *Modified*.

If you don't care about the nature of changes of the field then you can simply specify an exact type that is matched to the data field definition in data controller descriptor.

We know that *employeeId* has not changed and that is why we specify *System.Int32* as the type of the field. We do want to perform certain manipulations with the state of changes to *territories* and *lastName* fields. That requires *FieldValue* type.

The order of the fields and name capitalization does not have to match the definitions in data controller descriptor. Specify only the fields that you need.

The first line of the method resets *Modified* property to hide any field changes. Here is how *Territories* field was defined in data controller descriptor:

```
        <command id="command1" type="Text">
            <text>
                <![CDATA[
select
    "Employees"."EmployeeID" "EmployeeID"
    ,"Employees"."LastName" "LastName"
    ,"Employees"."FirstName" "FirstName"
. . . . . . . . . . . . . . . . .
    ,"ReportsTo"."LastName" "ReportsToLastName"
    ,"Employees"."PhotoPath" "PhotoPath",
    ,null "Territories" <<< Territories Field Definition
from "dbo"."Employees" "Employees"
    left join "dbo"."Employees" "ReportsTo" on
        "Employees"."ReportsTo" = "ReportsTo"."EmployeeID"
]]>
            </text>
        </command>
```

We are returning *null* as a field value. The field is not based on any actual database table field. The framework does not know that and will try to update *null* with a new value. Resetting *Modified* property to *false* will prevent any update attempts that would happen by default otherwise.

We do want the framework to perform an SQL update for any other fields that might have changed. If the only field that have changed is *Territories* then no update will be performed. That is why we always "pretend" that field *lastName* has been changed. An alternative is to write more code and execute an update on our own. You will have to call *PreventDefault()* method of your business rules class to prevent any default update logic from execution if you do your own updates.

POST-ACTION BUSINESS LOGIC

Here is how we will update the territories that an employee is responsible for.

C#:

```
[ControllerAction("Employees", "editForm1", "Update", ActionPhase.After)]
protected void AfterEmployeeUpdate(FieldValue territories, int employeeId)
{
    string[] newTerritories =
        Convert.ToString(territories.NewValue).Split(',');
    string[] oldTerritories =
        Convert.ToString(territories.OldValue).Split(',');
    // remove "unchecked" territories
    foreach (string territoryId in oldTerritories)
        if (!(String.IsNullOrEmpty(territoryId)) &&
            (Array.IndexOf(newTerritories, territoryId) == -1))
        {
            EmployeeTerritories et =
                EmployeeTerritories.SelectSingle(employeeId, territoryId);
            et.Delete();
        }
    // add new "checked" territories
    foreach (string territoryId in newTerritories)
        if (!(String.IsNullOrEmpty(territoryId)) &&
            (Array.IndexOf(oldTerritories, territoryId) == -1))
        {
            EmployeeTerritories et = new EmployeeTerritories();
            et.EmployeeID = employeeId;
            et.TerritoryID = territoryId;
            et.Insert();
        }
}
```

VB:

```
<ControllerAction("Employees", "editForm1", "Update", ActionPhase.After)> _
Protected Sub AfterEmployeeUpdate(ByVal territories As FieldValue, _
    ByVal employeeId As Integer)
    Dim newTerritories As String() = _
        Convert.ToString(territories.NewValue).Split(",")
    Dim oldTerritories As String() = _
        Convert.ToString(territories.OldValue).Split(",")
    ' remove "unchecked" territores
    For Each territoryId In oldTerritories
        If Not String.IsNullOrEmpty(territoryId) AndAlso _
            Array.IndexOf(newTerritories, territories) = -1 Then
            Dim et As EmployeeTerritories = _
                EmployeeTerritories.SelectSingle(employeeId, territoryId)
            et.Delete()
        End If
    Next
    ' add new "checked" territories
    For Each territoryId In newTerritories
        If Not String.IsNullOrEmpty(territoryId) AndAlso _
            Array.IndexOf(oldTerritories, territories) = -1 Then
            Dim et As EmployeeTerritories = New EmployeeTerritories()
            et.EmployeeID = employeeId
            et.TerritoryID = territoryId
            et.Insert()
        End If
    Next
End Sub
```

Parameter *territories* is defined as a variable of type *FieldValue* since we want to know the previous value of the field. We split new and old value of the field by comma.

Next we scan old territories and if any territory is not in the list of new ones then we delete the territory with the help of *EmployeeTerritories* class that was automatically generated for us by ASP.NET code generator Code OnTime Generator.

Finally we scan new territories and for any "new" territories that were not in the list of old ones we are executing an employee territory insertion. Again we do that we the help of automatically generated business object *EmployeeTerritories*.

CONCLUSION

A simple and easy to understand server programming model for business rules is a crown jewel of Data Aquarium Framework. Business rules are built on top custom action handlers and provide a truly simple and easy to use programming mode to enhance your AJAX ASP.NET application with server code.

In our next post we will discuss programmatic data filtering via business rules.

Code OnTime LLC

http://www.codeontime.com