Simple Base Persistent (ORM) Object for CF9 Now Available

Posted At : September 7, 2009 9:38 PM | Posted By : Bob Silverberg
Related Categories: ColdFusion, CF ORM Integration, BPO

I mentioned in an earlier blog post that I've been working on a Base Persistent Object (BPO) for ColdFusion 9 that can be extended by other components to provide them with certain behaviours. In order to add the kind of flexible behaviour that I desire in my BPO, I compose a number of other objects into it, which it then uses to accomplish things such as automatic validation and returning population and validation failures to whomever called its methods. I realized as I was preparing it for public consumption that it is a much larger task that I first envisioned to document and describe all of its intricacies and dependencies. I have decided, therefore to start by releasing a Simple Base Persistent Object, which can be used as a standalone object. It does not accomplish all that my full-featured BPO does, but it works as is, and will hopefully provide some ideas for what a BPO can do and how a BPO can be used.

For those who want to skip the explanation and just see the code, it can be downloaded from the Base Persistent Object RIAForge Project.

The Methods

The point of the BPO is that it defines methods which then become available to the persistent objects which extend it. The Simple BPO contains the following methods:

- populate()
- save()
- delete()
- configure()

The populate() method

The populate() method is easily the most useful method in the Simple BPO. Simply stated, it can be used to automatically populate an object's properties and many-to-one relationships from user-submitted data. It accepts two arguments:

1. *data*, which is a structure containing data to be used to populate the object. For example, one might pass the contents of the *form* scope into this argument.
2. *propList*, which is an optional array containing a list of properties to be populated. If this is passed in then only those properties listed in the array are populated.

When called it does the following:

- For each property of the object (or each property in the *propList*, if provided), it looks for a matching key in the *data* struct. When a matching key is found:
    - If the value of that key is an empty string and the property supports nulls, it calls the corresponding setter method passing in a null. This can be useful if one needs to set a property such as a date or a number to null.
    - Otherwise it calls the corresponding setter method passing in the value of the key.
- For each many-to-one relationship of the object:
    - It attempts to find a matching key by looking for one of the following keys in the *data* struct:
        - A key that matches the *name* of the relationship.
        - A key that matches the *fkcolumn* of the relationship.
    - If it finds a match it then attempts to load an object for the relationship using the value found in the key.
        - If it obtains an object, it calls the corresponding setter method passing in the object.
        - If not, and the relationship supports nulls, it calls the corresponding setter method passing in a null.

For example, let's say we have a User object like this:

```
<cfcomponent persistent="true" entityname="User" table="tblUser"
 extends="SimpleBasePersistentObject">
 <cfproperty name="UserId" fieldtype="id" generator="native" />
 <cfproperty name="UserName" />
 <cfproperty name="UserPass" />
 <cfproperty name="UserGroup" cfc="UserGroup" fieldtype="many-to-one"
  fkcolumn="UserGroupId" />
</cfcomponent>
```

We could then create and populate a User object from a struct like this:

```
<cfset User = EntityNew("User").init() />
<cfset data = StructNew() />
<cfset data.UserName = "Bob" />
<cfset data.UserPass = "myPass" />
<cfset data.UserGroup = 19 />
<cfset User.populate(data) />
<cfset EntitySave(User) />
```

Some things to note about the sample above:

- We wouldn't normally be populating the *data* struct manually as show above - it would usually already be available to us as the result of a form post.
- The value *19* above, which is being set into the *UserGroup* key of the *data* struct, corresponds to the id of the UserGroup object that we want to relate to this User.
- Rather than set the value *19* into the *UserGroup* key, we could set it into a *UserGroupId* key of the *data* struct. That will work as well because *UserGroupId* is the value of the *fkcolumn* attribute of the relationship.

Updating the example above to show how the populate() method might be called using the form scope, would look something like this:

```
<cfset User = EntityNew("User").init() />
<cfset User.populate(form) />
```

```
<cfset EntitySave(User) />
```

In addition to the ability to populate both properties and many-to-one relationships, the populate() method has one extra feature. I have found that it is common to want to*cleanse* input from a user via the HTMLEditFormat() function. For this reason, the populate() method supports automatic cleansing of data. This can be specified either for all properties of the component, or for individual properties. To turn it on for all properties, simply add a *cleanseInput="true"* attribute to your cfcomponent tag, like so:

```
<cfcomponent persistent="true" entityname="User" table="tblUser"

 extends="SimpleBasePersistentObject" cleanseInput="true">

 <cfproperty name="UserId" fieldtype="id" generator="native" />

 <cfproperty name="UserName" />

 <cfproperty name="UserPass" />

 <cfproperty name="UserGroup" cfc="UserGroup" fieldtype="many-to-one"

  fkcolumn="UserGroupId" />

</cfcomponent>
```

To turn it on for an individual property, add the *cleanseInput="true"* attribute to the cfproperty tag, like so:

```
<cfcomponent persistent="true" entityname="User" table="tblUser"

 extends="SimpleBasePersistentObject">

 <cfproperty name="UserId" fieldtype="id" generator="native" />

 <cfproperty name="UserName" />

 <cfproperty name="UserPass" cleanseInput="true" />

 <cfproperty name="UserGroup" cfc="UserGroup" fieldtype="many-to-one"

  fkcolumn="UserGroupId" />

</cfcomponent>
```

Note that you can also turn off cleansing for individual properties when it has been turned on for the whole component, like so:

```
<cfcomponent persistent="true" entityname="User" table="tblUser"

 extends="SimpleBasePersistentObject" cleanseInput="true">

 <cfproperty name="UserId" fieldtype="id" generator="native" />

 <cfproperty name="UserName" cleanseInput="false" />

 <cfproperty name="UserPass" />

 <cfproperty name="UserGroup" cfc="UserGroup" fieldtype="many-to-one"

  fkcolumn="UserGroupId" />

</cfcomponent>
```

The populate() method relies on certain conventions to work, namely:

- The name of the keys in the *data* struct must match the *name* attribute of the corresponding cfproperty tag.
- All properties are considered to allow nulls by default. To specify a property (or relationship) as not allowing nulls, a *notnull="true"* attribute must be specified. This is a standard attribute of the cfproperty tag.
- The name of the many-to-one relationship must be the same as the *entityname* attribute of the corresponding object.

The save() method

The save() method in the Simple BPO does nothing more than call EntitySave(), passing in the current object. This means that rather than writing code like so:

```
<cfset User = EntityNew("User").init() />

<cfset User.populate(form) />

<cfset EntitySave(User) />
```

We can write:

```
<cfset User = EntityNew("User").init() />

<cfset User.populate(form) />

<cfset User.save() />
```

This doesn't really add any functionality, but it does hide the implementation of the save() method, so one could theoretically change it in the future without having to change the calling routine. In my full BPO the save() method does more, namely validating the object and returning any failure messages to the calling routine.

The delete() method

Like the save() method, the delete() method in the Simple BPO does nothing more than call EntityDelete(), passing in the current object. This means that rather than writing code like so:

```
<cfset User = EntityLoadByPK("User",1) />

<cfset EntityDelete(User) />
```

We can write:

```
<cfset User = EntityLoadByPK("User",1) />

<cfset User.delete() />
```

As with the save() method, this doesn't add any functionality, but it does hide the implementation of the delete() method. Also, if one is going to use User.save(), for consistency I think one should use User.delete().

I originally included a call to ormFlush() after the EntitySave() and EntityDelete() calls, because I generally run with flushatrequestend="false" in my ormsettings in Application.cfc, because of how I do validations. I removed those calls because this is the *Simple* BPO. At this point I'm not sure what the overhead is for calling ormFlush() immediately after a save/delete, so maybe it makes sense to put them back in, as a convenience.

The configure() method

The configure() method in the BPO does nothing at all. It is meant to be overridden in the objects that extend it. It allows one to specify some code that will always be executed when an object is instantiated (any time the init() method is called). The reason I like using a configure() method is that it means I don't have to override the init() method in any of my persistent objects. I know that any code that is in the init() method of the BPO will always be executed, and I don't have to worry about the signature of the init() method nor have to remember to call super.init() if I want to change the initialization behaviour of an object.

Summary

This Simple Base Persistent Object is a work in progress. There are certainly other behaviours that could be added to it, most notably validation. It is a common practice to include validation code inside business objects, and you can see an excellent example of that in **Paul Marcotte's Metro**. I have no issues with Paul's approach, in fact I think what he's developed is quite remarkable, but I prefer to keep all of my validation rules external to the object itself, which is why I developed and use **ValidateThis**.

My full BPO makes use of ValidateThis, as well as a number of other objects, and I hope to be able to find the time to give it some TLC and release and describe it in the not-too-distant future.

Although I am releasing this via RIAForge, at this point the BPO really this just represents a bunch of ideas, many of which could be improved upon. I already have some ideas for improvement, which I've added to the **issue tracker** on the RIAForge project. I welcome any discussion, feedback, and suggestions about it. My hope is that we, as a community, can come up with an excellent BPO that the community as a whole can benefit from.