PROFESSIONAL COLDFUSION CONTENT FOR THE COLDFUSION PROFESSIONAL

The Fusion Authority Quarterly Update

Vol. II Issue I

SPECIAL BUSINESS ISSUE:

Business practices and concerns to channel your programming skills into cash

How to Avoid Unpaid Consulting

Making Google Pay

Deconstructing the Consulting Contract

Plus — Developing Applications with Tranfer

And Much, Much, More...

VOL.II ISSUE I $14.95

ISSN 1932-0264

21 >

9 771932 026000

CONTENTS

Fusion Authority Quarterly ■ Vol. II ■ Issue I

Developing Applications with Transfer

by Mark Mandel

A Relational Database in the Object-Oriented Application

Developing all the backend code that manages the data in a database-driven, object-oriented system can be a very long and repetitive process, and the codebase can become quite immense.

At an object level, we must write all the get and set methods for each of a table's columns in the system. If the application also incorporates objects composed of structures or arrays of other objects, we must write code to manage these collections, and to match them back to the proper database tables correctly.

At the database level, consider the onerous task of writing all the SQL statements to do the basic create, read, update and delete operations on the tables.

Then factor in the code that takes the queries for the database tables and maps them to the objects themselves. This may require massaging the data that comes from the SELECT statements so that it can populate an object, or retrieving data from an object for INSERT, DELETE or UPDATE statements.

There is also the problem of system maintenance. Changing a single database table column causes changes in the object representing the table, the SQL that is used to map it, and the code that is used to map the SQL to the object. This sort of maintenance can quickly add up. Any major updates to a database structure, or the refactoring of a codebase or database, can add hours, if not days, to development time.

What if there was a way to automate this entire process, and still allow enough flexibility to match a wide variety of requirements?

Object Relational Mapping – The Solution

Object Relational Mapping (ORM) software was developed to solve exactly this problem. When configured, ORM software will generate all the SQL statements required to manage data within the database itself. It will also take the data from the database and map it to and from the objects that represent that data. This often incorporates representing and managing the relational database tables through a series of collections of child objects within a parent object, which can be a relatively complicated operation when done manually.

ORMs will often also automatically generate the objects that represent the data, and provide functionality to extend the generated object, allowing customization. An extensive suite of capabilities often manages the model aspect of a given application, depending on the specific ORM solution.

Transfer – An ORM for ColdFusion

Transfer is an ORM for ColdFusion that has been in development since 2005. It provides all of the above functionality and other features as well. At a core level, the information in Transfer's two XML configuration files allows Transfer to interact with the required database tables, generate its ColdFusion code as a series of .transfer files, and build and populate TransferObjects, which represent the data stored in the database.

TransferObjects, or the objects in your Transfer application, resolve to a CFC type, 'transfer.com. TransferObject'. While a Transfer application will often use only CFCs of this type, it is possible to provide more specific CFC typing using Decorators, covered further on in this article.

Transfer is supported on four databases: Microsoft SQL Server 2000+, MySQL 4.1+, PostgreSQL 8.1+, and Oracle 9i+.

One of Transfer's key features is a caching layer. Transfer maintains a cache of the requested TransferObjects in its system, thus greatly reducing your database traffic and speeding up your system's performance.

This caching layer is highly configurable through the object configuration file, including configuration options for the ColdFusion scope the cache resides in, the number of TransferObjects to be cached and/or the duration a TransferObject is cached. Details of the configuration elements inside the Transfer object configuration file are outside the scope of this article.

The configurable nature of Transfer gives us a wide variety of application development options and great deal of flexibility, while still providing ease of development.

Installing Transfer

As in many other ColdFusion frameworks, to install Transfer, the /transfer/ directory needs to be in the webroot of your application, or a mapping called 'transfer' must point to the Transfer directory.

The Transfer Datasource Configuration File

The datasource configuration file provides Transfer with the required information to connect to the correct ColdFusion datasource. See Listing 1 below:

Listing 1: The Datasource Configuration File - datasource.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <datasource
     xsi:noNamespaceSchemaLocation="../../transfer/resources/xsd/datasource.xsd"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
3    <name>tBlog</name>
4    <username></username>
5    <password></password>
6 </datasource>
```

This configuration file should be fairly self-explanatory. It simply gives the name of the ColdFusion datasource, and the username and password, if required.

On line 2 of Listing 1, there is a declaration for a datasource.xsd XML Schema file. This is useful when developing with an XML editor, as it allows the editor to validate the XML and provide auto-completion and code hinting.

While the name of the datasource configuration file can be anything you like, I would suggest naming it 'datasource.xml' to make things simple.

Transfer Object Configuration File

Transfer requires information on how to map tables in the system database to objects that it is going to generate and populate, and it gets this through the object configuration file.

For an example, see Listing 2 below:

Listing 2: Object Configuration File - transfer.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<transfer xsi:noNamespaceSchemaLocation="../../../transfer/resources/xsd/transfer.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<objectDefinitions>
   <package name="user">
      <!-- User details -->
      <object name="User" table="tbl_User">
         <id name="IDUser" type="numeric"/>
            <property name="Name" type="string" column="user_Name"/>
            <property name="Email" type="string" column="user_Email"/>
      </object>
   </package>
```

```
<package name="system">
  <!-- Different categories for Blog Posts -->
  <object name="Category" table="tbl_Category">
    <id name="IDCategory" type="numeric"/>
      <property name="Name" type="string" column="category_Name"/>
      <property name="OrderIndex" type="numeric"
        column="category_OrderIndex"/>
  </object>
</package>

<package name="post">
  <!-- A Blog Post -->

  <object name="Post" table="tbl_Post" decorator="tblog.com.Post">
    <id name="IDPost" type="numeric"/>
    <property name="Title" type="string" column="post_Title"/>
    <property name="Body" type="string" column="post_Body"/>
    <property name="DateTime" type="date" column="post_DateTime"/>

    <!-- Link between a Post and the User it who wrote it -->
    <manytoone name="User">
      <link to="user.User" column="lnkIDUser"/>
     </manytoone>

     <!--  Link between a Post and its array of Comments  -->

      <onetomany name="Comment">
        <link to="post.Comment" column="lnkIDPost"/>

        <!--  Specifying the collection is an array and
              is ordered by the dateTime property of the Comment  -->
         <collection type="array">
           <order property="DateTime" order="asc"/>
         </collection>
      </onetomany>

      <!--  Link to the many Categories that Posts can fall under.  This is provided by
            an intermediate table between the Category table and the Post table.  -->
      <manytomany name="Category" table="lnk_PostCategory">
        <link to="post.Post" column="lnkIDPost"/>
        <link to="system.Category" column="lnkIDCategory"/>

        <!--  Specifying the collection is an array and is ordered by the orderIndex
              property of the Category -->
        <collection type="array">
          <order property="OrderIndex" order="asc"/>
        </collection>
      </manytomany>
  </object>
```

```
50      <!-- A comment for a blog post -->
51      <object name="Comment" table="tbl_Comment">
52        <id name="IDComment" type="numeric"/>
53        <property name="Name" type="string" column="comment_Name"/>
54        <property name="Value" type="string" column="comment_Value"/>
55        <property name="DateTime" type="date" column="comment_DateTime"/>
56      </object>
58   </package>

59 </objectDefinitions>
60 </transfer>
```

Mapping Objects to Tables

As you can see in the above example, several core XML elements allow objects to be mapped to tables:

1. *package* elements provide organization to the structure of the document, and become part of the class name used to retrieve generated objects from Transfer.
2. *object* elements map an object type to a table, and give that object a name. There may be more than one object in a package, such as the 'post' package defined in lines 21-58, which contains the Post and Comment objects.
3. *id*, a sub-element of object, defines the primary key for that object. id generates get() and set() methods corresponding to the **name** attribute of the object with that *id* element. For example, line 52 tells Transfer that getIDComment() will be generated on the Comment TransferObject.
4. *property* elements in an object map columns to properties generated on the object. The property element also generates get() and set() methods corresponding to each of its configurations. For example, line 25 tells Transfer to generate the getTitle() and setTitle() methods on the Post TransferObject.

In line 6, we have an object of type user.User. The name of this type is derived by combining the *package* element's name and the *object* element's name. This User TransferObject maps back to the tbl_User table in the database. The table's primary key column, 'IDUser', maps back to an *id* of the same name on the User object. The object has two properties, 'Name' and 'Email', which map back to the columns 'user_Name' and 'user_Email', respectively, in the tbl_User table.

Finally, when you build your object configuration file, please remember that XML is case-sensitive, while names of objects (in quotes) may be either upper- or lower-cased. Whatever names you give, keep them consistent or you'll get caught in the XML case-sensitivity trap! It can often be handy to use an XML editor to validate your XML against the XML Schema that is provided for the object configuration file.

I would suggest naming this file 'transfer.xml'; however, there is no restriction on what the name of this file should be.

Mapping Objects to Tables - Object Composition

In terms of object-oriented programming, composition represents a "has-a" relationship, where a single object is made up of multiple sub-objects and the composed sub-objects have a closely-related or dependent relationship with their parent. In terms of a pure OO definition, sub-objects can have no independent existence without their parent; however, in this case, this is largely dependent on your application design. (For more information on composition, see our definition in the Object-Oriented Lexicon, FAQU Issue 2). In Listing 2, the Post TransferObject is composed of a User TransferObject representing the author of the Post, an array of Comment TransferObjects representing all the comments that have been made on the Post, and a second array of Category TransferObjects representing the Categories that the Post belongs to.

As we see in Listing 2, Transfer can also generate and manage object composition, and this is done through several XML elements in the configuration file.

The names of these elements echo the relationships that are found in a relational database: one-to-many, many-to-one, and many-to-many.

Mapping Objects to Tables – ManyToOne Composition

The use of a *ManyToOne* element allows an instance of a TransferObject to contain a single TransferObject. In line 29 of Listing 2, the Post object is configured to contain a single User TransferObject. While this may seem like a one-to-one relationship, at a database level this is actually many-to-one, as many Posts can share the one User.

This many-to-one composition on a TransferObject is controlled by the lnkIDUser foreign key on the tbl_Post table, which points to the primary key of IDUser on the tbl_User table. This also generates the getUser(), setUser(), removeUser() and hasUser() methods on the Post object for managing the User record associated with the Post.

Mapping Objects to Tables – OneToMany Composition

A *OneToMany* element allows an instance of a Transfer object to contain multiple TransferObjects as a collection of objects, which can be set as either an ordered or unordered array, or as a struct. In line 33 of Listing 2, the Post object contains an array of Comments ordered by the date they were created.

This one-to-many composition is controlled by the foreign key lnkIDPost on the tbl_Comment table, which points to the primary key IDPost on the tbl_Post table. The array of Comments is ordered by the Comment object's DateTime property.

This configuration will generate getComment(), getCommentArray() , containsComment(), findComment() and sortComment() methods on the Post object to manage the Comments within a Post, and setParentPost(), getParentPost(), removeParentPost() and hasParentPost() methods to manage the Comment's connection to its parent Post.

Mapping Objects to Tables – ManyToMany Composition

A *ManyToMany* element allows an instance of a TransferObject to contain multiple child TransferObjects, including multiple instances of the same type of TransferObject. It also allows child TransferObjects to belong to more than one parent. These child TransferObjects can be contained in an ordered or unordered array, or a struct, similar to a OneToMany collection. On Line 41 of Listing 2, a Post can contain multiple Categories, and since the relationship is many-to-many, different Posts can all belong to the same Category.

This many-to-many composition is controlled at the database level through the interim table lnk_PostCategory, which contains a foreign key to the tbl_Post table, lnkIDPost, and a foreign key to the tbl_Category table, lnkIDCategory. The array is also configured so that it is ordered by the 'OrderIndex' found on the Category.

This configuration will generate the getCategory(), getCategoryStruct(), containsCategory(), sortCategory(), addCategory(), removeCategory(), findCategory() and clearCategory() functions on the Post TransferObject to manage the Post's connection to specific Categories.

Mapping Objects to Tables – Lazy Loading

Transfer also supports lazy loading of composite objects. This means that one-to-many, many-to-one and many-to-many collections can be configured to only load their data when it is requested, a feature that can be very important for application performance.

By default, Transfer does not lazily load its composite elements. To configure lazy loading, add the attribute **lazy** to the composite element, and set its value to 'true'.

For example, let's set the configuration of the Comment collection found inside a Post (Listing 2, line 33) to lazily load:

Listing 3: Lazy Loading of the Comment Collection:

```
<object name="Post" table="tbl_Post">
  ...
  <onetomany name="Comment" lazy="true">
    ...
  </onetomany>
</object>
```

In the configuration shown in Listing 3, the Comment of the Post will only be loaded if any of the methods generated for the OneToMany configuration element, such as getCommentArray(), are called.

Mapping Objects to Tables – Decorators

Where a generic transfer.com.TransferObject is not acceptable, or where you need to add a high level of custom ColdFusion code to an object, you might want to set a Decorator in the Object Configuration file.

A Decorator, based on the Decorator design pattern, is a CFC that extends transfer.com.TransferDecorator. When it's configured, an instance of the Decorator containing the generated TransferObject is returned instead of the TransferObject requested by Transfer.

Transfer allows the specified Decorator to automatically extend all the public methods of the generated TransferObject it contains. This means that the CFC designated as Decorator has access to all the methods that are generated for the TransferObject, but also has the ability to overload those methods, and overwrite or extend the default functionality. This is very similar to the usual inheritance hierarchy; however, in this case it is functionality that is provided by Transfer, and not by an 'extends' attribute on a CFC.

On line 23 of Listing 2, the Post TransferObject is configured with a Decorator of type 'tblog.com.Post' and therefore, when a request is made for the Post TransferObject, a CFC of type 'tblog.com.Post' is returned.

Listing 4: A Post Decorator CFC

```
<cfcomponent hint="Post Decorator" extends="transfer.com.TransferDecorator"
   output="false">
  <cffunction name="getNumberOfComments" access="public"
     hint="Custom function for retrieving how many Comments a Post has"
     returntype="numeric">
     <cfreturn ArrayLen(getCommentArray())>
  </cffunction>
</cfcomponent>
```

The Decorator in Listing 4 leverages the getCommentArray() method generated by Transfer for the Post TransferObject. Transfer makes this method available to the Decorator CFC so that it is able to return the number of Comments the Post has.

Using Transfer

Once the configuration files have been written, you will be able to take full advantage of Transfer's ability to manipulate the data held in the database.

Using Transfer - The TransferFactory

To begin using Transfer, we must create a Singleton of the transfer.TransferFactory CFC. This means that there will be only one instance of the TransferFactory in the entire application. Usually, you'll create an instance of the TransferFactory within the ColdFusion Application scope, but this can depend on your application design.

The TransferFactory CFC takes three arguments on its init function:

1. The relative path to the datasource configuration file, often called 'datasource.xml'
2. The relative path to the Transfer object configuration file, often called 'transfer.xml'
3. The relative path to where Transfer will write the .transfer files.

For example:

```
application.transferFactory = createObject("component", "transfer.TransferFactory").init(
  "/tblog/resources/xml/datasource.xml", "/tblog/resources/xml/transfer.xml",
  "/tblog/definitions");
```

The TransferFactory CFC has two public methods that allow interaction with Transfer.

The first is getDatasource(), which simply returns a CFC with the properties of the configured ColdFusion datasource from the Datasource configuration file. This CFC is useful for custom SQL that is not generated by Transfer, and therefore needs the datasource details for the CFQUERY to run properly. See Listing 5 below.

Listing 5: The getDatasource() Method

```
1 datasource = application.transferFactory.getDatasource();
2 <cfquery name="qPosts" datasource="#datasource.getName()#"
    username="#datasource.getUsername()#" password="# datasource.getUsername()#">
3   <!--- query here --->
4 </cfquery>
```

On line 2 of Listing 5, the Datasource CFC provides the information to execute a CFQUERY tag.

The second method is getTransfer(), which returns the transfer.com.Transfer CFC. This CFC is the primary gateway to the main functionality of Transfer, and is covered in the following sections.

Using Transfer - Creating a New Object

To create an object we will invoke the new() method with the TransferObject's class name as an argument. This class name is derived from the *package* element names and the *object* name, as defined in the configuration XML.

For example:

Listing 6: Example of the new() Method

```
1 transfer = application.transferFactory.getTransfer();
2 post = transfer.new("post.Post");
```

In Listing 6, line 2, we are requesting a new instance of the Post TransferObject defined in Listing 2. We pass in the argument of 'post.Post' because it is a combination of the **name** attribute on the package configuration element (Listing 2, line 21) and the name of the Post *object* element (Listing 2, line 23).

The code in Listing 6 will generate the required ColdFusion code for the Post TransferObject, and return a new transfer.com.TransferObject or perhaps a new Decorator, depending on the configuration, along with all the methods required to manage the object. Therefore, once the Post TransferObject has been created, values can be set and retrieved on it.

Listing 7: Example of setting values on a TransferObject

```
post.setTitle(form.title);
post.setBody(form.Body);
```

Listing 8: Example of getting values from a TransferObject

```
<h2><a href="displayPost.cfm?ID=#post.getIDPost()#">#post.getTitle()#</a></h2>
```

In Listing 7, the Title and Body properties of the Post TransferObject are being set for form values.

In Listing 8, the IDPost and Title properties from the Post Object are being used to display the Post's Title and a link to the Post in HTML.

Using Transfer – Saving an Object

Persisting the properties of an object to the database is a very simple operation, that can be done via the create(), update() and save() methods that are found on the Transfer CFC.

To explicitly insert a record into the database, we invoke the create() method, passing the TransferObject to be inserted as an argument.

For example, let's now insert a User from the configuration in Listing 2 into the database:

Listing 9: Using the create() Method to Insert a Record

```
1  transfer = application.transferFactory.getTransfer();
2  user = transfer.new("user.User");

3  user.setName(form.name);
4  user.setEmail(form.email);

5  transfer.create(user);
```

In Listing 9, the Name and Email properties of the User TransferObject are set on lines 3 and 4, respectively. When the object is passed to the Transfer CFC's create() method, Transfer inserts the User into the tbl_User table, as configured in Listing 2, line 6, with the columns populated with the values stored in the User TransferObject. Transfer generates the required SQL INSERT statement and executes it against the configured database.

To update a record in the database, we call the update() method with the TransferObject to be updated. Transfer will generate the appropriate SQL UPDATE statement to update the values in the designated database table.

Here's an example that will update the Categories that a Post TransferObject contains:

Listing 10: Example of Using the update() Method

```
1  transfer = application.transferFactory.getTransfer();
2  //clear out the categories
3  post.clearCategory();

4  //loop through the categories and add them back in
5  categories = listToArray(form.category);
6  len = ArrayLen(categories);
7  for(counter = 1; counter lte len; counter = counter + 1)
8  {
9      category = transfer.get("system.Category", categories[counter]);
10     post.addCategory(category);
11  }
12 transfer.update(post);
```

In Listing 10, when calling the update() method with the Post as the argument, Transfer will update column values of the tbl_Post table with the values from the Post TransferObject. Transfer will also update the values of the *ManyToMany* table in Listing 2, line 41, with the primary key stored in the Post TransferObject and the primary key stored in the Categories added to the Post in Listing 10, line 9.

While these are specific methods for explicitly inserting or updating data within the database, Transfer also provides a save() method that automatically determines whether the create() or update() method is necessary. This will depend on whether or not the TransferObject has been persisted in the database.

The save() method works exactly like the create() and update() methods:

Listing 11: An Example of the save() Method

```
transfer = application.transferFactory.getTransfer();
post = transfer.new("post.Post");
post.setTitle(form.title);
post.setBody(form.Body);
transfer.save(post);
```

In Listing 11, the save() method will intelligently call the create() method on the Transfer CFC to insert the new Post TransferObject, which has not yet been persisted, into the database. This contributes to code reuse, as the same ColdFusion can be used to create a new TransferObject or to update a TransferObject that is already persisted in the database.

Using Transfer - Retrieving an Object

To retrieve a TransferObject from the database by its primary key, we invoke the get() method, passing the class name and primary key value of the TransferObject as arguments.

To retrieve a Post TransferObject with all its Comments, Categories and User, we might do the following:

Listing 12: Using the get() Method

```
1 transfer = application.transferFactory.getTransfer();
2 post = transfer.get("post.Post", url.id);
```

Like the new() method, get() will generate all the appropriate ColdFusion for the Post TransferObject and any of its composite children, and return the appropriate Decorator or TransferObject. It will also generate the required SELECT statement to populate the generated TransferObject with its required data. In Listing 12, line 2, the Post TransferObject is queried on aspects of its composite elements. Now let's look at Listing 13, which displays the details of all the Comments in the Post that has been retrieved:

Listing 13: Example of Composition Methods

```
<ul>
<!--- show all the comments in the post --->
<cfscript>
  comments = post.getCommentArray();
     len = ArrayLen(comments);
</cfscript>
<cfloop from="1" to="#len#" index="counter">
  <cfset comment = comments[counter]>
     <li>
     #comment.getName()# wrote on
     #DateFormat(comment.getDateTime(), "dd mmm yy")#
     #TimeFormat(comment.getDateTime(), "hh:mm:ss tt")#
     :<br/>
     #comment.getValue()#
     </li>
</cfloop>
</ul>
```

In Listing 13, the Post TransferObject contains all the Comment TransferObjects within it as an array, which can be retrieved and looped through.

Sometimes retrieving a TransferObject by its primary key is not appropriate, and therefore it is also possible to retrieve a TransferObject from the database by a unique property value. This is done by calling the readByPropertyValue() method, taking the class name of a TransferObject, a property name, and a property value as arguments, and retrieving the TransferObject by the given property and its value for that class.

Here is an example of the readByProperty() Method:

```
post = getTransfer().readByProperty("post.Post", "Title", "My first Post");
```

In this example, we retrieve a Post TransferObject by its Title property, whose value is "My First Post".

We can also retrieve a TransferObject by multiple property values that translate to an unique record, by using the readByPropertyMap() method. This method takes the class name of the TransferObject and a struct with keys corresponding to the properties of a TransferObject and the value of the key used for retrieval.

Listing 14: Example of readByPropertyMap()

```
1  map = StructNew();
2  map.title = "My First Post";
3  post = getTransfer().readByPropertyMap("post.Post", map);
```

In Listing 14, we again retrieve the Post TransferObject by its Title property. However, the property to retrieve is set from the map struct, seen on line 1, and its value of "My First Post", set on line 2.

Finally, it is possible to retrieve TransferObjects using a WHERE statement via the readByWhere() method. This method takes the class name of the TransferObject as an argument and a WHERE statement that contains property names in curly braces ('{}'), which are then translated into the column names for that class. Here is some sample code that uses the readByWhere() method:

```
post = getTransfer().readByWhere("post.Post", "{Title} = 'My First Post'");
```

In this example, we retrieve the Post TransferObject by its Title property, with the value of "My First Post", which is set via the WHERE statement used to retrieve the Post TransferObject.

If any of these methods do not find a TransferObject in the database, they will return a new object of the same class as the requested TransferObject. This contributes to code reuse, as we can run the same ColdFusion when creating a new TransferObject and when updating Transfer Objects that are already persisted in the database.

Using Transfer – Deleting an Object

Deleting a TransferObject from the database is a very simple operation. Simply invoke the delete() method, passing in the TranferObject to be deleted as an argument, and the record represented by the TransferObject will be deleted from the database.

For example:

Listing 15: Deleting a User TransferObject

```
1  transfer = application.transferFactory.getTransfer();
2  user = transfer.get("user.User", form.userid);
3  transfer.delete(user);
```

In Listing 15, we retrieve the User TransferObject from the database on line 2, and then delete the User from the database on line 3. Transfer generates the required DELETE statement to delete the User from the table.

Using Transfer – List Queries

Transfer also allows simple list operations on tables. By default, these list queries alias the column names of the tables with the configured property names and allow for ordering by a single object property.

To return a list of all records for a given class, call the list() method with the TransferObject class as an argument. The following code uses the list() method:

```
query = getTransfer().list("post.Post");
```

This example will return a query of all the records in the tbl_Post table, as configured in Listing 2, line 23 for the "post.Post" TransferObject.

We can also return a list of records, filtered by a given property value. Invoke the ListByProperty() method, while passing in the class of the TransferObject, and the name and value of the property to filter, as arguments.

For example, the following code will return a query of all posts with the Title of "My First Post":

```
query = getTransfer().listByProperty("post.Post", "Title", "My First Post");
```

Transfer also allows us to query for a list of records, filtered by a set of property values. We do this by invoking the listByPropertyMap() method, which takes two arguments: the class name of the TransferObject and a struct with key-value pairs corresponding to the properties of a TransferObject and the value of the key to filter by.

Listing 16: An example of the listByPropertyMap() Method

```
map = StructNew();
map.title = "my first post";
query = getTransfer().listByPropertyMap("post.Post", map);
```

Like the example before it, Listing 16 also returns a query for all Posts with the Title "My First Post". However, here the struct passed in to the listByPropertyMap() method delineates that the query be filtered by its Title, by the key 'title' and the value found therein.

Finally, we can retrieve list queries by a SQL WHERE statement with the listByWhere() method. This method takes the class name of the TransferObject as an argument and a SQL WHERE statement that can contain property names in curly braces ('{}'), which are then translated into the proper column names for that class.

Listing 17: Using the listByWhere() Method

```
query = getTransfer().listByWhere("post.post",
   {date} > #CreateODBCDateTime("1/1/2000")#);
```

In Listing 17, Transfer runs a query that returns all Posts created after January 1, 2000, by passing a SQL WHERE statement to that effect as an argument to the listByWhere() method.

Other Functionality

Transfer is not limited to the features discussed in this article. It offers a wide variety of other functionality to help map objects to database tables and manage the model space of application development.

These features include, but are not limited to:

- Primary Key Management - A full suite of tools for managing the value of the primary key of a TransferObject.
- Nullable Properties - Set properties to have NULL values, so that they will be entered into the database as NULL.
- Configurable Caching - A wide range of caching options allowing you to configure caching at the object level.

- An event model in which observers can be set - You can capture create, update, and delete events in Transfer by setting CFCs as listeners to Transfer events.
- Object.clone() - Clone a TransferObject; changes made to the clone aren't reflected in the cache until the clone is saved.
- Automatic Cache Synchronization - If TransferObjects have been discarded from the cache, or are clones, calling the create(), update(), save() or delete() methods on them will automatically synchronise them with their cached version.
- Configurable conditions on collections - Filter out certain TransferObjects from composite collections.
- Ignore on insert and/or update - Configure properties to be ignored when generating INSERT and/or UPDATE SQL statements.
- Refresh on insert and/or update - Configure properties to refresh their values after an INSERT and/or UPDATE.
- Custom UDFs within the XML Configuration - Place custom CFML within the XML object configuration.
- Get TransferObject Metadata from Transfer - Retrieve metadata outlining the structure of a TransferObject class.
- Transferdoc - A tool to introspect TransferObject classes and document the methods that are generated on them.

Conclusion

Transfer is a powerful, highly configurable ORM solution that can greatly reduce the amount of time spent developing the model aspect of an application, assist in mapping that model to a given database, and give the developer a high degree of flexibility to facilitate a wide variety of development requirements.

It generates the CFCs that represent your data and the SQL statements that map the objects to your database and update that data. It handles object composition, generates basic list queries and provides a configurable caching layer that gives you a large increase in performance in your application.

Transfer's configuration and extension possibilities, including the object configuration file and the use of Decorators, means that while Transfer generates many aspects of an application for you, you still have a high degree of control over your application, and no limitation on extending the generated base.

These resources will help you learn more about Transfer:

The Transfer home page: http://www.compoundtheory.com/transfer/

The RIAForge home page: http://transfer.riaforge.org

The Transfer mailing list: http://groups.google.com/group/transfer-dev/

The CompoundTheory blog has several articles on Transfer: http://www.compoundtheory.com

All of the code examples in this article were derived from the blog example that can be download at: http://www.compoundtheory.com/?action=transfer.examples

Mark Mandel is a Senior Developer at NGA.net and has been working with ColdFusion for a number of years, even at his very own dot com bomb back in the late 90's. More recently he has become very active within the ColdFusion open-source community, authoring several projects, including Transfer ORM and JavaLoader. Mark can often be found blogging at http://www.compoundtheory.com, posting on the ColdFusion mailing lists, or generally causing havoc in the #coldfusion channel on Dalnet irc network. When he's not too busy coding he enjoys training in martial arts in a wide variety of disciplines and reading way too much fantasy literature.