

Contents

- [Managing Relationships and Compositions](#)
 - [Sample Database](#)
 - [Table: tbl_User](#)
 - [Table: tbl_Post](#)
 - [Table: tbl_Comment](#)
 - [Table: tbl_Category](#)
 - [Table: lnk_PostCategory](#)
 - [Where Are the relationships?](#)
 - [OneToMany or ManyToOne?](#)
 - [ManyToOne](#)
 - [Database Tables](#)
 - [Transfer.xml Configuration](#)
 - [Sample Code](#)
 - [OneToMany](#)
 - [Database Tables](#)
 - [Transfer.xml Configuration](#)
 - [Sample Code](#)
 - [ManyToMany](#)
 - [Database Tables](#)
 - [Transfer.xml Configuration](#)
 - [Sample Code](#)

Managing Relationships and Compositions

While we manage relationships in our physical database using Primary Keys and Foreign Keys, in Transfer we define our relationships in terms of Objects.

Transfer does not use the concept of foreign keys, rather it uses *Object Composition*, meaning that relationships are created by defining Objects that contain other Objects.

So in Transfer a relationship is manifested as an Object that exists *inside* another Object. For example, a Blog Post is written by a User. In our physical database the Blog Post table may have a foreign key that points back to the User table. With Transfer we'd have a Blog Post object that *contains* a User object.

Since relationships always map back to a foreign key in your database, let's start by looking at a sample database, and from there we will look at how it is represented with Objects via composition.

Sample Database

This database is used by the tBlog sample application, which can be [downloaded here](#). It is comprised of 5 tables, each of which is required to support the blog application:

Table: tbl_User

Purpose: Each blog post is owned by a single user. Those user records are stored in this table.

Column	Datatype	Purpose
--------	----------	---------

IDUser	numeric	Primary Key (auto-generated)
User_Name	string	The name of the User
User_Email	string	The email address of the User

Table: tbl_Post

Purpose: The records for individual blog posts are stored in this table.

Column	Datatype	Purpose
IDPost	numeric	Primary Key (auto-generated)
InkIDUser	numeric	Foreign Key back to tbl_User, links the Post with its corresponding User
post_Title	string	The title of the Post
post_Body	string	The contents of the Post
post_DateTime	datetime	The date/time that the Post was added

Table: tbl_Comment

Purpose: Each blog post can have multiple comments associated with it. The records for those comments are stored in this table.

Column	Datatype	Purpose
IDComment	numeric	Primary Key (auto-generated)
InkIDPost	numeric	Foreign Key back to tbl_Post, links the Comment with its corresponding Post

comment_Name	string	The name associated with the Comment
comment_Value	string	The text of the Comment
comment_DateTime	datetime	The date/time that the Post was added

Table: tbl_Category

Purpose: Blog posts can be assigned to categories. The records that describe each category are stored in this table.

Column	Datatype	Purpose
IDCategory	numeric	Primary Key (auto-generated)
category_Name	string	The name of the Category
category_OrderIndex	string	Used for sorting Categories for output
comment_DateTime	datetime	The date/time that the Post was added

Table: lnk_PostCategory

Purpose: Each blog post can be assigned to multiple categories, and each category can have multiple blog posts assigned to it. The records that keep track of the link between a single blog post and a single category are stored in this table.

Column	Datatype	Purpose
lnkIDPost	numeric	Foreign Key back to tbl_Post
lnkIDCategory	numeric	Foreign Key back to tbl_Category

Where Are the relationships?

As mentioned above, relationships in Transfer are always manifested in your database as foreign keys. So, if we want to find the relationships in our database we look for foreign keys. The sample database contains four foreign keys:

tbl_post.lnkIDUser implements a one-to-many relationship between Users and Posts.

tbl_comment.lnkIDPost implements a one-to-many relationship between Posts and Comments.

lnk_postcategory.lnkIDPost plus *lnk_postcategory.lnkIDCategory* implement a many-to-many

relationship between Posts and Categories.

Note that when talking about foreign keys in a relational database we only use the term one-to-many. In terms of your physical database Transfer's concepts of *OneToMany* and *ManyToOne* are identical. You cannot differentiate between a *OneToMany* and a *ManyToOne* in your physical database - they are both implemented via a foreign key.

That means that when creating a Transfer relationship from a one-to-many, as a result of a foreign key in our database, we need to decide whether to model that to Transfer as a *OneToMany* or a *ManyToOne*. Note that this decision must be made. You cannot define both a *OneToMany* and a *ManyToOne* to Transfer for a single foreign key in your database. More on how to make this decision can be found below.

The many-to-many is more straightforward. A many-to-many in your database, implemented via a linking table, becomes a *ManyToMany* in Transfer. The one exception to this is if you are storing data in your linking table that is not just the two foreign keys. If you are storing additional data in your linking table you cannot use Transfer's built-in *ManyToMany* support. You must model that as a combination of *ManyToOne* and/or *OneToMany* relationships. A simple rule to follow is that if you have more than 2 columns in your linking table, not including a auto-generated primary key, you cannot use a *ManyToMany* relationship.

OneToMany or ManyToOne?

So, how does one decide whether to model a one-to-many in a database as a *OneToMany* or a *ManyToOne*?

First it is important to understand the difference between a *OneToMany* and *ManyToOne* in Transfer, quoting from another part of the wiki:

OneToMany composition is useful when you wish for TransferObjects on both sides of the relationship to see each other, or for the Parent to have a collection of the child objects attached to it.

A *ManyToOne* collection is useful when, either for application design, or performance reasons, you only want an Objects to load one side of the relationship, and not generate a collection of Objects.

For example, we have a series of Comments on a Blog Post, we want the Post to have a collection of Comments on it, so we would use a *OneToMany* (we assume we won't have too many Comments per Blog Post).

However, if we had a Product which could be included in hundreds of Orders, we would build our Order object with a *ManyToOne* to Product, so that an Order would have a Product attached, but the Product would have no need to be aware of its Orders.

In other words, you can think of *OneToMany* as **OneWithMany**, and *ManyToOne* as one of **ManyWithOne**.

ManyToOne

Database Tables

tbl_User

Column	Datatype	Purpose
IDUser	numeric	Primary Key (auto-generated)

User_Name	string	The name of the User
User_Email	string	The email address of the User

tbl_Post

Column	Datatype	Purpose
IDPost	numeric	Primary Key (auto-generated)
lnkIDUser	numeric	Foreign Key back to tbl_User, links the Post with its corresponding User
post_Title	string	The title of the Post
post_Body	string	The contents of the Post
post_DateTime	datetime	The date/time that the Post was added

Transfer.xml Configuration

```

<package name= "user" >
  <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= "post" >
  <object name= "Post" table= "tbl_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" />
    <manytoone name= "User" >
      <link to= "user.User" column= "lnkIDUser" />
    </manytoone>
  </object>
</package>

```

Note that we place the *ManyToOne* declaration in the object whose table contains the foreign key. We tell transfer the name of the column that represents the foreign key, and the name of the object whose table contains the corresponding primary key.

Sample Code

Setting a ManyToOne

```
<cfscript>
post = transfer.new( "post.Post" );

post.setTitle( "My Title" );
post.setBody( "The body of my post" );
post.setDateTime(Now());

user = transfer.get( "user.User" ,1);

post.setUser(user);

transfer.save(post);
</cfscript>
```

In order to set the User for a Post, we must first get the Transfer Object that corresponds to the User, and then pass that into the setUser() method of the Post object.

Getting a ManyToOne

```
<cfscript>
post = transfer.get( "post.Post" ,1);

if (post.hasUser()) {
    user = post.getUser();
}
</cfscript>
```

In order to retrieve the User from a Post, we simply call the getUser() method, which returns a User Transfer Object to us. Note that if it is possible that the User may not be set for the Post (e.g., if the InkIDUser allows NULLs), it is wise to check for the existence of a User first by calling hasUser().

OneToMany

Database Tables

tbl_Post

Column	Datatype	Purpose
IDPost	numeric	Primary Key (auto-generated)
InkIDUser	numeric	Foreign Key back to tbl_User, links the Post with its corresponding User
post_Title	string	The title of the Post

post_Body	string	The contents of the Post
post_DateTime	datetime	The date/time that the Post was added

tbl_Comment

Column	Datatype	Purpose
IDComment	numeric	Primary Key (auto-generated)
lnkIDPost	numeric	Foreign Key back to tbl_Post, links the Comment with its corresponding Post
comment_Name	string	The name associated with the Comment
comment_Value	string	The text of the Comment
comment_DateTime	datetime	The date/time that the Post was added

Transfer.xml Configuration

```

<package name= "post" >
  <object name= "Post" table= "tbl_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" />

    <onetomany name= "Comment" >
      <link to= "post.Comment" column= "lnkIDPost" />
      <collection type= "array" >
        <order property= "DateTime" order= "asc" />
      </collection>
    </onetomany>
  </object>

  <object name= "Comment" table= "tbl_Comment" >
    <id name= "IDComment" type= "numeric" />
    <property name= "Name" type= "string" column= "comment_Name" />
    <property name= "Value" type= "string" column= "comment_Value" />
    <property name= "DateTime" type= "date" column= "comment_DateTime" />
  </object>

```

```
</package>
```

Note that we place the *OneToMany* declaration in the object whose table **does not contain** the foreign key. We refer to that object as the Parent. We tell Transfer to link back to the object whose table contains the foreign key. We refer to that object as the Child. We must specify the name of the column in the Child table that represents the foreign key.

Because a *OneToMany* creates a collection, we need to tell Transfer whether that collection should be an array or a structure. If it is to be an array, we can optionally specify a sort sequence for the items in that array. This sort sequence must point to a property in the Child object.

We can also ask Transfer for create the collection as a structure, for example:

```
<onetomany name= "Comment" >
  <link to= "post.Comment" column= "lnkIDPost" />
  <collection type= "struct" >
    <key property= "DateTime" />
  </collection>
</onetomany>
```

In this case we're asking for a struct, the keys of which are the values of the DateTime property in each of the Child objects. Note that the key values must be unique for a given Parent.

Sample Code

Setting a OneToMany

```
<cfscript>
  comment = transfer.new( "post.Comment" );

  comment.setName( "Bob Silverberg" );
  comment.setValue( "The content of my comment" );
  comment.setDateTime(Now());

  post = transfer.get( "post.Post" ,1);

  comment.setParentPost(post);

  transfer.save(comment);
</cfscript>
```

In order to set the Post for a Comment (which is the same as adding a Comment to a Post), we must first get the Transfer Object that corresponds to the Post, and then pass that into the setParentPost() method of the Comment object.

Removing a OneToMany

```
<cfscript>
  comment = transfer.get( "post.Comment" ,1);

  comment.removeParentPost();

  transfer.save(comment);
</cfscript>
```

That will set the value of the foreign key in tbl_Comment to NULL, and will remove the Comment object from the collection stored in the Post object.

Getting a Parent

```

<cfscript>
  comment = transfer.get( "post.Comment" ,1);

  if (comment.hasParentPost()) {
    post = comment.getParentPost();
  }
</cfscript>

```

In order to retrieve the Post for a Comment, we simply call the `getParentPost()` method, which returns a Post Transfer Object to us. Note that if it is possible that the Post may not be set for the Comment (e.g., if the `InkIDPost` allows NULLs), it is wise to check for the existence of a Post first by calling `hasParentPost()`. In our example application this would be unnecessary, as it doesn't make sense to have a Comment that doesn't belong to a Post.

Getting a Collection

```

<cfscript>
  post = transfer.get( "post.Post" ,1);

  comments = post.getCommentArray();
</cfscript>

```

That will return an array that contains one Comment Transfer Object for each Comment that exists for the given Post.

Getting a Child

```

<cfscript>
  post = transfer.get( "post.Post" ,1);

  comment = post.getComment(1);
</cfscript>

```

That will return a Comment Transfer Object that is the first Child found in the Post's collection of Comments. Note that this will throw an error if the child requested does not exist in the collection.

ManyToMany

Database Tables

tbl_Post

Column	Datatype	Purpose
IDPost	numeric	Primary Key (auto-generated)
InkIDUser	numeric	Foreign Key back to tbl_User, links the Post with its corresponding User
post_Title	string	The title of the Post

post_Body	string	The contents of the Post
post_DateTime	datetime	The date/time that the Post was added

tbl_Category

Column	Datatype	Purpose
IDCategory	numeric	Primary Key (auto-generated)
category_Name	string	The name of the Category
category_OrderIndex	string	Used for sorting Categories for output
comment_DateTime	datetime	The date/time that the Post was added

lnk_PostCategory

Column	Datatype	Purpose
lnkIDPost	numeric	Foreign Key back to tbl_Post
lnkIDCategory	numeric	Foreign Key back to tbl_Category

Transfer.xml Configuration

```

<package name= "post" >
  <object name= "Post" table= "tbl_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" />

    <manytomany name= "Category" table= "lnk_PostCategory" >
      <link to= "post.Post" column= "lnkIDPost" />
      <link to= "system.Category" column= "lnkIDCategory" />
      <collection type= "array" >
        <order property= "OrderIndex" order= "asc" />
      </collection>
    </manytomany>
  </object>
</package>

```

```

<package name= "system" >
  <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>

```

Note that we place the *ManyToMany* declaration in the object from which we wish to navigate. Because *ManyToMany* is not bi-directional, we must choose one object which will be the starting point for manipulating the relationship. We will refer to that object as the Parent. We will refer to the object that represents the other side of the *ManyToMany* relationship as the Child. We tell transfer the name of the table in our database that is used to record the links between the two objects. We also need to tell transfer the names of the foreign key columns in that table, as well as which object each foreign key points to. Note that we must record the link to the Parent object first in the configuration file.

Because a *ManyToMany* creates a collection, we need to tell Transfer whether that collection should be an array or a structure. If it is to be an array, we can optionally specify a sort sequence for the items in that array. This sort sequence must point to a property in the Child object.

We can also ask Transfer for create the collection as a structure, for example:

```

<manytomany name= "Category" table= "lnk_PostCategory" >
  <link to= "post.Post" column= "lnkIDPost" />
  <link to= "system.Category" column= "lnkIDCategory" />
  <collection type= "struct" >
    <key property= "OrderIndex" />
  </collection>
</manytomany>

```

In this case we're asking for a struct, the keys of which are the values of the OrderIndex property in each of the Child objects. Note that these values must be unique for a given Parent.

Sample Code

Setting a ManyToMany

```

<cfscript>
  post = transfer.new( "post.Post" );

  post.setTitle( "My Title" );
  post.setBody( "The body of my post" );
  post.setDateTime(Now());

  category = transfer.get( "system.Category" ,1);

  post.addCategory(category);

  transfer.save(post);
</cfscript>

```

In order to add a Category to a Post, we must first get the Transfer Object that corresponds to the Category, and then pass that into the addCategory() method of the Post object.

Removing a ManyToMany

```

<cfscript>
  post = transfer.get( "post.Post" ,1);

```

```

category = transfer.get( "system.Category" ,1);

post.removeCategory(category);

transfer.save(post);
</cfscript>

```

In order to remove a Category from a Post, we must first get the Transfer Object that corresponds to the Category, and then pass that into the removeCategory() method of the Post object.

Clearing all ManyToManys

```

<cfscript>
post = transfer.get( "post.Post" ,1);

post.clearCategory();

transfer.save(post);
</cfscript>

```

That will remove all Categories from the Post.

Getting a Collection

```

<cfscript>
post = transfer.get( "post.Post" ,1);

categories = post.getCategoryArray();
</cfscript>

```

That will return an array that contains one Category Transfer Object for each Category that exists for the given Post.

Getting a Child

```

<cfscript>
post = transfer.get( "post.Post" ,1);

category = post.getCategory(1);
</cfscript>

```

That will return a Category Transfer Object that is the first Child found in the Post's collection of Category. Note that this will throw an error if the child requested does not exist in the collection.