

Contents

- [Transactions and Transfer](#)
 - [The Problem](#)
 - [Nested Transactions](#)
 - [Cache Synchronisation](#)
- [The Solution](#)
 - [Direct Execution](#)
 - [Aspect Oriented Programming](#)

Transactions and Transfer

The following gives an outline on how to manage Transactions with Transfer, some of the issues that are faced, and how they are resolved with Transfer.

The Problem

There are two issues I was trying to solve with Transfer Transaction support, so let's look at a common Transfer Transaction scenario and see what the problems are. Say we have a [TransferObject](#) named 'Foo', and we have a Service, named 'FooService'. We may have some code that looks like:

```
<cffunction name="saveFoo" hint="saves Foo, and its children" access="public" returnType="void" output="false" >
  <cfargument name="foo" hint="The foo" type="Foo" required="Yes" >
  <cfset var children = arguments.foo.getFooChildrenArray() />
  <cfset var child = 0 />

  <cftransaction>
    <cfset getTransfer().save(arguments.foo, false) />
    <cfloop array="#children#" index="child" >
      <cfset getTransfer().save(child, false) />
    </cfloop>
  </cftransaction>
</cffunction>
```

In which we save Foo, and its many FooChildren, and we wrap it in a <cftransaction>, so that if anything goes wrong, the whole set of data gets rolled back.

Now, there are two issues with this scenario that need to be addressed:

Nested Transactions

You can't nest the saveFoo() call inside another Transaction block. i.e. code that looks like:

```
<cffunction name="saveFooParent" hint="saving a Foo Parent" access="public" returnType="void" output="false" >
  <cfargument name="fooParent" hint="The foo" type="FooParent" required="Yes" >
  <cftransaction>
    <cfif arguments.fooParent.hasFoo() >
      <cfset saveFoo(arguments.fooParent.getFoo(), false) />
    </cfif>
    <cfset getTransfer().save(arguments.fooParent, false) />
  </cftransaction>
</cffunction>
```

This will throw an error, as ColdFusion won't allow you to nest <cftransaction> blocks. We could write something similar to what Transfer already does, and pass a boolean to the save method to tell it whether or not to use an inner transaction, but this is cumbersome, and depending on your application architecture, you may not be in a position to know if a given method is in a transaction.

Cache Synchronisation

Cache synchronisation is a real problem when database data rolls back, and the Transfer cache stays the same.

i.e. If we look at the saveFoo() method above, if the data on Foo has been updated, but something goes wrong when saving the children, then the data for Foo gets rolled back along with everything else, but the cache stays the same, which can be a very bad thing, as your object data is totally out of sync with your database (and not when you want it to be).

The Solution

So what is the solution? The solution is to use the new [Transfer Transaction Object!](#)

The Transaction object provides both:

1. Nested Transaction support.
 - If a Transaction wraps another Transaction, the inner transaction just becomes merged with the outer, as if it was just one big Transaction.
2. Transfer cache synchronisation
 - If anything goes wrong in your Transaction, then Transfer will automatically discard any object whose data was modified during that Transaction.

So let's look at how we can use the [Transaction](#) object. First of all, how do we get it? Very simply, it's available from the [TransferFactory](#), so can now go:

```
<cfscript>
  transaction = application.transferFactory.getTransaction();
</cfscript>
```

And get access to the Transaction object. On a side note, the reason it is accessible from the TransferFactory, and not Transfer, is because it is not specifically tied to Transfer. You can use this Transaction object with anything that requires a <cftransaction> wrapped around it.

There are three different ways you can use the Transaction object, and will look at them all, in regards to the example we gave above.

Direct Execution

The first one is the simplest:

```
transaction.execute(component, methodName, [arguments])
```

This simply executes a given method (even private ones!) on a given component, with an optional struct of arguments, and automatically wraps the method call *in a transaction!*

In the instance of above, we would have to change our code to look something like:

```
<cffunction name="saveFoo" hint="saves Foo, and its children" access="public" returnType="void" output="false" >
  <cfargument name="foo" hint="The foo" type="Foo" required="Yes" >
  <cfset getTransaction().execute( this , "_saveFoo" , arguments) >
</cffunction>

<cffunction name="_saveFoo" hint="saves Foo, and its children" access="public" returnType="void" output="false" >
  <cfargument name="foo" hint="The foo" type="Foo" required="Yes" >
  <cfset var children = arguments.foo.getFooChildrenArray() />
  <cfset var child = 0 />

  <cfset getTransfer().save(arguments.foo) />
  <cfloop array="#children#" index="child" >
    <cfset getTransfer().save(child) />
  </cfloop>
</cffunction>
```

```
</cffunction>
```

So the above code would execute the method `_saveFoo`, wrapped out in our nesting-safe `Transaction`. Pretty cool, but a bit cludgy as we have to implement a second method.

This type of `Transaction` handling is best suited for `Transient` objects, such as inside `Decorators`

Aspect Oriented Programming

I think we can do better, and in fact we can, using the second method.

```
transaction.advise(component, function )
```

For those of you who are familiar with `Aspect Oriented Programming (AOP)`, this method takes the component, and the function itself, and wraps a nesting-safe `Transaction` advise around the function that has been passed in. If that was a little high level, to give an example, I could change the `FooService` to be:

```
<cffunction name="init" hint="saves Foo, and its children" access="public" returnType="void" output="false" >
  <cfargument name="transaction" hint="The transaction object" type="transfer.com.sql.transaction.Transaction" required="Yes" >
  <cfset arguments.transaction.advise( this , saveFoo) />
</cffunction>

<cffunction name="saveFoo" hint="saves Foo, and its children" access="public" returnType="void" output="false" >
  <cfargument name="foo" hint="The foo" type="Foo" required="Yes" >
  <cfset var children = arguments.foo.getFooChildrenArray() />
  <cfset var child = 0 />

  <cfset getTransfer().save(arguments.foo) />
  <cfloop array="#children#" index="child" >
    <cfset getTransfer().save(child) />
  </cfloop>
</cffunction>
```

So what happens at `init()` time, the original `saveFoo()` is wrapped up in a nesting-safe `transaction`, and you don't have to write much extra code, except to remove the old `<cftransaction>` tags, and tell the `Transaction` object what methods to advise. Much better than the first way, and it works on both private and public methods.

So the above, is very nice, however, what happens if we have a lot of functions we want to `advise()`? Well, the third way of using the `Transaction` Object is handy for that!

```
transaction.advise(component, regex, [debug])
```

With this approach, we can apply the `Transaction` advise to all methods, both public and private whose name matches the given regular expression.

For example, if we want to apply `Transaction` advice to every method that starts with 'save' in our `FooService`, we would just need to have our `FooService` written like:

```
<cffunction name="init" hint="saves Foo, and its children" access="public" returnType="void" output="false" >
  <cfargument name="transaction" hint="The transaction object" type="transfer.com.sql.transaction.Transaction" required="Yes" >
  <cfset arguments.transaction.advise( this , "^save" ) />
</cffunction>

<cffunction name="saveFoo" hint="saves Foo, and its children" access="public" returnType="void" output="false" >
  <cfargument name="foo" hint="The foo" type="Foo" required="Yes" >
  <cfset var children = arguments.foo.getFooChildrenArray() />
  <cfset var child = 0 />

  <cfset getTransfer().save(arguments.foo) />
  <cfloop array="#children#" index="child" >
    <cfset getTransfer().save(child) />
  </cfloop>
</cffunction>
```

And we are done. If we ever add a method that has 'save' at the beginning of its name, such as the `saveFooParent()` from before, then it will automatically be wrapped up in a `Transaction`, even if the method is private.

If the 'debug' argument is set to true, the `Transaction` object will `<cftrace>` all the methods that are advised, so you can see if your method is being wrapped up in a `Transaction` or not.

AOP type of `Transaction` handling is best suited for `Singletons`, such as a `Service Layer`, due to the code generation that occurs.