



ColdFusion ORM

A guide to developing applications using ColdFusion ORM

JOHN WHISH

ColdFusion ORM

A Guide to Developing Applications using ColdFusion ORM

First Edition

Copyright © 2011, John Whish

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the author, except for the use of brief quotations in a book review.

Although the author made every effort to ensure that the information in this book was correct at press time, the author does not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause.

Cover design by Andy Beer
<http://www.flamingsquirrel.co.uk/>

Foreword

This is a book that was begging to be written.

The integration of Hibernate (an enterprise-strength object relational mapper (ORM)) into Adobe ColdFusion is, in my opinion, the most significant change to the language since the introduction of ColdFusion components. The power that all ColdFusion developers now have to rapidly develop applications is tremendous, but the complexity of Hibernate is not to be underestimated. While the documentation that comes with Adobe ColdFusion does provide a reasonable introduction to ORM, it is quite simply inadequate for those wishing to develop real world applications making use of Hibernate. In order to be successful with ColdFusion ORM you need to educate yourself, and I can think of no better resource than the book that you are holding in your (virtual) hands right now.

Who would you trust to guide you through this journey of learning ColdFusion ORM? I was pleased as punch to learn that John Whish would be taking on this important job. For nearly half a decade John has shown us that he has a knack for explaining complicated topics in an understandable way via his blog, and now he has tackled this comprehensive study with the same skill and determination. He has taken the time to share his vast knowledge of ORM with us all, covering virtually every topic one could imagine, from the most introductory to the extremely advanced.

This is a book that every ColdFusion programmer should have on his or her shelf. I know I am glad to have it on mine.

Bob Silverberg

Principal at SilverWare Consulting

<http://www.silverwareconsulting.com/>

Table of Contents

Foreword	2
Table of Contents	3
Acknowledgements	9
Preface	10
Requirements	10
What is ORM?	11
What is Hibernate?	11
Book Conventions.....	11
Chapter 1 : Getting Started	13
Configuring ORM in your Application	13
Mapping Tables to Objects	14
Allow ColdFusion to Modify the Database.....	17
ORMReload	18
Generating Database Schema	18
Data Types and Precision.....	19
Annotations	21
Chapter 2 : ORM is CRUD	24
Creating Records	24
Disabling setters	25
EntityNew.....	25
EntitySave	26
Reading data	26
EntityLoad	26
Reading Individual Records by ID	28
EntityLoadByPK.....	29
Loading Filtered Data	29
Ordering Data	30

Displaying data	30
Updating Existing Data	31
Deleting Records	33
Chapter 3 : ORM Sessions.....	35
What are ORM Sessions?	35
The FlushAtRequestEnd Setting	35
ORMFlush.....	37
ORMClearSession	38
Transactions	40
Transactions in ColdFusion 9.0.0	42
Chapter 4 : Relationships	43
One-to-Many Relationships	43
Generated Methods for Handling Relationships.....	44
The Singularname attribute	46
Adding Entities to a One-to-Many Relationship.....	47
Removing Entities from a One-to-Many Relationship	53
Choosing the Sort Order.....	54
Many-to-One Relationships	55
Many-to-Many Relationships	57
One-to-One Relationships	60
Handling Nulls.....	62
Chapter 5 : Bi-Directional Associations	65
Uni-Directional versus Bi-Directional	65
Managing Relationships	65
SQLIntegrityConstraintViolationException.....	69
The Inverse Attribute	70
Inverse and Many-to-Many	72
Chapter 6 : Cascading	76
Cascade Types.....	77

Cascade None	77
Cascade Delete Type	78
Cascade Save-update Type	79
Cascade All Type.....	80
Cascade Delete-orphan Type.....	80
Cascade All-delete-orphan Type	81
Chapter 7 : HQL	82
Getting Started With HQL	82
Using a Where Clause	83
Named Parameters.....	84
Positional Parameters.....	84
Nicer Looking HQL	85
Using Operators in HQL	85
Using Functions in HQL.....	86
Relationships in HQL	87
Selecting Properties.....	91
Chapter 8 : Debugging	94
Logging with ColdFusion Builder's Console View	94
Running ColdFusion from a Console.....	95
Viewing parameter values	96
Logging to a File	97
Viewing HQL Statements in Inline Debugging	98
Getting HQL using Hibernate Statistics	99
Chapter 9 : Computed Properties	102
Using Formulas.....	102
EntityReload	104
Where Filter	107
Chapter 10 : Event Handlers.....	110
The Global Event Handler	112

Chapter 11 : Improving Performance	116
Paging.....	116
Maxresults	116
Offset	116
Lazy Loading	117
The n+1 Selects Problem	120
Fetch="join".....	122
Fetch with HQL	123
Batch Fetching.....	124
ORM Settings	126
Chapter 12 : Caching.....	130
First-Level Cache.....	130
Second-Level Cache	130
What's Cached?	132
Caching Associated Entities	133
Managing the cache	134
ORMEvictEntity.....	134
ORMEvictCollection.....	135
ORMEvictQueries	135
Bulk Operations	136
Choosing the Cache Provider.....	136
Chapter 13 : Validation.....	138
Chapter 14 : Advanced Mapping.....	141
Mapping Inheritance	141
One Table Per Persistent Entity	142
One Table Per Persistent CFC With Discriminator	147
One Table per Hierarchy	149
HQL and Inheritance.....	150
Using Collections Instead of Arrays.....	151

Getting an Entity from a Collection	158
The Inverse Attribute	158
Chapter 15 : Schema Generation Revisited	160
DataTypes and Precision.....	160
The ormtype attribute.....	160
Specifying Length, Nullable and more.....	161
Naming Strategy	163
In-Built Naming Strategies	163
Using Your Own Naming Strategy.....	165
Chapter 16 : Tips, Tricks & Gotchas.....	167
Tips	167
Populate Using the EntityNew Constructor	167
HQL IN Clause.....	167
Refactoring to ORM with EntityToQuery	168
Detecting Changes	168
Multiple Datasources	169
Non Persisted Properties.....	169
Entities with the same name.....	170
Tricks	171
Hibernate Statistics.....	172
Gotchas.....	175
Constructor Arguments.....	175
Detached Entities.....	175
WriteDump causes ColdFusion to Crash	176
Cached MetaData.....	177
Generated Has Methods and Non-Persisted Entities.....	177
Appendix	179
Quick Mapping Reference	179
Relationships	179

Setting up an Apache Derby Database	180
References	182
Documentation.....	182
ColdFusion ORM Documentation.....	182
Hibernate Documentation	182
Publications	182
Articles	182
Hat Tips.....	182
Where Next?.....	183
And Finally... ..	185

Acknowledgements

Although I have been the one to actually put this book together, it would not have been possible with the help, support, advice and time of several other people. I would like to thank them all as they have been an important part of making this book possible. They are in (alphabetical order):

Andy Beer

<http://www.flamingsquirrel.co.uk/>

Andy designed the cover for me. If you can tell a book by its cover then this book is going to be pretty damn good!

Kev McCabe

<http://www.bigmadkev.com/>

Kev has been kind enough to donate his time and technical insight for this book and it's all the better for it.

Matt Gifford

<http://www.mattgifford.co.uk/>

Matt gave me the inspiration and encouragement required to write this book.

Seb Duggan

<http://sebduggan.com/>

Seb provided a huge amount of editorial input, proof reading and found lots of typos.

Victoria Whish

Vic supplied caffeine and flapjack and allowed me to spend my evenings doing this. I'd also like to thank her for just being her.

Although, not directly involved in this book, I'd also like to thank Bob Silverberg (<http://www.silverwareconsulting.com/>) for sharing his seemingly limitless knowledge.

A special thank you goes to Julian Halliwell (<http://www.simplicityweb.co.uk>) who not only bought this book, but kindly pointed out several typos and grammatical errors.

Preface

In my opinion ColdFusion 9 is the biggest leap forward in developing applications since ColdFusion MX6.1.

Why? Well, ColdFusion 9 comes with the Hibernate Object Relational Mapper (ORM) library baked in. In short, what this means is that you can quickly build applications without having to write lots of code to get the data in and out of the database.

ORM and Hibernate are huge topics and I won't be covering every aspect or feature but I hope it will serve as a useful guide to building applications in ColdFusion using the power of Hibernate. I intend for this book to be complimentary, rather than a replacement for the official Adobe ColdFusion or Hibernate documentation (both of which are available for free online). In addition, please do not consider the approaches I outline in subsequent pages to be the only way to build your applications. There are many approaches you could take, and you should take some time to experiment and decide which style suits your development needs best.

Read on if I've piqued your interest...

Requirements

The code samples in this book are written for ColdFusion 9 updater 1, better known as ColdFusion 9.0.1. If you're not sure which version you are running, you can easily check by logging into the ColdFusion administrator (CFIDE) and clicking the System Information icon (it's the blue "i" in the top right hand corner).

You should see something like this:

System Information	
Server Details	
Server Product	ColdFusion
Version	9,0,1,274733
Edition	Developer
Serial Number	Developer

If you don't have ColdFusion 9.0.1 you can download the free upgrade from:

http://www.adobe.com/support/coldfusion/downloads_updates.html#cf9

To use ColdFusion ORM you will also need a database. ColdFusion ORM will work with all the major database servers; if you don't have one available for experimenting with, you can use ColdFusion to create an Apache Derby database for you. Instructions on how to do this can be found in the Appendix.

What is ORM?

ORM stands for Object Relational Mapping. In short, the primary purpose of an ORM framework is to assist you by handling saving and retrieving of objects from a relational database (RDBMS). By using an ORM framework, it means that you can concentrate on developing applications without having to write lots of repetitive SQL statements to support select, insert, update and delete operations. Hibernate makes use of objects; if you're not familiar with using objects in your applications, don't worry: there are lots of code samples for you to look at.

What is Hibernate?

Under the hood, ColdFusion ORM capabilities are powered by an open source Java persistence framework called Hibernate. Much of the complexity of using Hibernate has been hidden away by the ColdFusion engineering team at Adobe, making it relatively painless and intuitive to use. Hibernate works with a number of different database engines including MySQL, Microsoft SQL Server, PostgreSQL, Oracle and Apache Derby. In fact there is some clever optimisation that Hibernate can do depending on which database server you are using. Unsurprisingly, Microsoft Access is not supported.

Hibernate has been around since 2001 and is widely adopted, with companies such as Evernote and Google using it for parts of their business. It is now part of the JBoss Community.

Book Conventions

Most of the code samples featured in this book are written in cfscript, but you are free to use tags in your code if you prefer.

Code samples are written using a fixed-width font as follows:

```
1  <cfscript>
2  Author = new Author();
3  Author.setName( "John Whish" );
4  WriteDump( Author );
5  </cfscript>
```

In addition, entity names, database table names and function names may also be shown in a fixed-width font in paragraph text.

Where code samples have changed or had code added to them, the changed sections are highlighted and bold. For example:

```
1 <cfscript>
2 Author = new Author();
3 Author.setName( "John Whish" );
4 Author.setNickname( "Aliaspooryorik" );
5 WriteDump( Author );
6 </cfscript>
```

To fit code samples into the book without spanning multiple pages, some code sections may be omitted and replaced with three dots. For example:

```
1 <cfscript>
2 Author = new Author();
3 ...
4 WriteDump( Author );
5 </cfscript>
```

The omitted code is not important for demonstration purposes, or may have been previously shown.

For formatting and readability reasons, some code may be formatted or simplified. This particularly applies to generated SQL statements.

If the code wraps onto another line then it is shown with an → in the gutter.

```
1 log4j.appender.HIBERNATECONSOLE.layout.ConversionPattern=%d
→ {MM/dd HH:mm:ss} [%t] HIBERNATE %-5p - %m%n%n
```

Script names are shown in italic as follows:

model/Category.cfc

Now that the preliminaries are out of the way, it's time to get started!

Chapter 1 : Getting Started

Adobe's engineers have done a very good job of allowing you to use and configure the ORM capabilities in your application simply by writing code in CFML. Unless you really want to, there is no need to dig into Hibernate's own configuration files.

For the purposes of this book I'll be creating a new database and using ColdFusion ORM to create the tables, columns and data for us. However, you can use ColdFusion ORM with existing databases and data if you want; I'll cover this in a bit more detail later in this chapter.

Configuring ORM in your Application

As with all ColdFusion applications that use a database, you will need to create a datasource in the ColdFusion administrator. The process for creating a datasource for use with an ORM-enabled application is exactly the same as you would normally follow.

If you are still using *Application.cfm*, now is the time to change and start using *Application.cfc* instead, as this is where you enable and configure the ORM capabilities. *Application.cfc* belongs in the root of your application.

Application.cfc

```
1  /**
2  * I am the Application.cfc for my application
3  */
4  component
5  {
6      // set the unique name of the application
7      this.name = "bookblog";
8      // set the datasource
9      this.datasource = "bookblog";
10     // turn ORM on
11     this.ormenabled = true;
12 }
```

Above is a simple *Application.cfc* for an ORM-enabled application. The two lines of interest are line 9:

```
9      this.datasource = "bookblog";
```

and line 11:

```
11     this.ormenabled = true;
```

`this.datasource` is a new feature of ColdFusion 9 that allows you to set an application-wide datasource which is then used for ORM – as well as for traditional SQL statements using `CFQUERY`. As a result, you no longer need to use the `datasource` attribute of `CFQUERY` unless you wish to use a different datasource from the application-wide datasource.

Another new setting introduced in ColdFusion 9 is `this.ormenabled` which, as the name suggests, tells ColdFusion that you want to use ORM in your application. By default this setting is `false`.

As you can see, it's very simple to set up ORM in your application: simply set the name of your datasource and enable ORM.

Mapping Tables to Objects

Once the application is set up to use ORM, we need to do the actual Object Relational Mapping part! In its basic form, we can create a CFC which has properties; those properties are then mapped to columns in a database table. CFCs that map to a database are commonly referred to as entities. If an entity has a corresponding record in the database, it is known as a persistent entity. Persistent in this situation simply means that the data is stored in a database.

I always find it's easier to learn things if I can apply them to a real-world scenario. As such, I'm going to use the example of a simple blogging application to demonstrate ORM.

You may be thinking that the first step is to create the database table so that we can map to it. However, when you are using an ORM you should really be thinking about objects and not database schemas. If, like me, you've spent years designing the database first, try to take a step back and think about objects and their relationships instead. Hibernate makes this change in thinking a little easier by being able to create database tables and relationships for you. By default ColdFusion won't change the database schema, but you can choose to allow it to do so, which is useful for development.

Mapping objects to tables can get quite complicated, but most of the time it's a relatively simple process. As I mentioned earlier, you can use the ORM capabilities of ColdFusion to map brand new CFCs to an existing database table or even to a database view.

Before we create any CFCs, it's a good idea to organise your code. CFCs that represent something are commonly considered to be part of the "business model". As such I'm going to create a folder called "model" in my application root and put my "business model" CFCs in this folder. You can go further and categorise those CFCs further by putting them into sub directories, but that is beyond the scope of this book.

The first CFC I'm going to create will represent a blog category. To do this, I've created a blank CFC in the model directory and named it *Category.cfc*. The code for *Category.cfc* is shown below:

model/Category.cfc

```
1  /**
2   * I model a blog category
3   */
4   component persistent="true" table="categories"
5   {
6       // identifier
7       property name="id" fieldtype="id"
8           generator="native" column="category_pk";
9
10      // properties
11      property name="title" length="100";
12  }
```

This little bit of code is going to change your life (or at least the way you develop applications!). There are several things here that may be new to you, so here is a breakdown.

The first three lines are a comment written using the JavaDoc style syntax (note the double asterisk on the first line):

```
1  /**
2   * I model a blog category
3   */
```

Comments written like this can be used for generating documentation: they work in exactly the same way as if you'd written your code using a `cfcomponent` tag with a `hint` attribute.

Line four is the start of the component block;

```
4   component persistent="true" table="categories"
```

As you can see there is a 'persistent' attribute. The persistent attribute tells ColdFusion that we want this CFC to be mapped to a database. If you omit the persistent attribute then ColdFusion will treat *Category.cfc* as a normal ColdFusion component.

The fourth line also has a 'table' attribute with a value of `categories`. The table attribute is used to specify the database table that you want to map this CFC to. If you don't have a table attribute, ColdFusion will assume that you are trying to map to a database table that has the same name as the CFC. The table attribute is particularly useful when dealing with legacy databases where the table names contain underscores or use a prefix, as you can have still have descriptive and meaningful entity names.

As we don't have an existing database table, I want ColdFusion to create a new table in the database. Entities normally have a singular naming convention, whereas you may wish to use a plural naming convention for database tables. So instead of using the default table name of 'Category' (the same as the CFC name), my database table will be named 'categories'. As a side note, the case of the table name may depend on which database engine you are using.

Line 6 is a comment. Lines 7 & 8 are where things get interesting. Please note, that I've used two lines for formatting reasons; you could have them all on one line.

```
6 // identifier
7 property name="id" fieldtype="id"
8   generator="native" column="category_pk";
```

In ColdFusion 9, property tags have a whole new meaning and usage. On this line we are defining an identifier (primary key in database terms) that will be used to identify a specific category.

The value of the name attribute is simply the name we want to use for the variable that will hold the id. In this case, the variable will be called 'id'.

As this property is an identifier, I need to tell ColdFusion this by using the `fieldtype` attribute. If the property is being used as an identifier then you use `fieldtype="id"`. All entity CFCs must have at least one property that has a `fieldtype` of `id`.

I want the database to be responsible for creating primary keys, so I've added the `generator` attribute with a value of 'native'. What this means is that the identifier will be generated for us by the database when we save a new entity.

The `generator` attribute accepts several values. A value of 'native' tells ColdFusion that you want to use the default method of your database engine to generate the primary key. If you need to allocate the identifier yourself, then you can use the value of 'assigned' for the `generator` attribute.

Finally we have the `column` attribute. This is used to map this property to a specific column name in the `categories` database table. The attribute is optional, and if omitted, ColdFusion will assume that the database column name is the same as the property name. As I want to employ a naming strategy for primary keys in my database of table name with a '_pk' suffix, I've included the `column` attribute for the `id` property.

Last but not least, in our `Category` CFC, we define a property of `title`:

```
11 property name="title" length="100";
```

As I haven't specified a `column` attribute, ColdFusion will try to map this property to a `title` column in the database table. As we don't yet have a column or even a database table, ColdFusion will create a `varchar` column for us named 'title' with a length of 100 characters. The number of characters for the `varchar` field in the database is dictated by the value specified in the 'length' attribute. If you omit the `length` attribute, then the column will be 255 characters long.

Allow ColdFusion to Modify the Database

If you recall, I mentioned that Hibernate could generate the database schema for us, but by default this was disabled. To tell ColdFusion to generate the database tables for us, we need to modify the *Application.cfc* we created earlier.

Application.cfc

```
1  /**
2  * I am the Application.cfc for my application
3  */
4  component
5  {
6      // set the unique name of the application
7      this.name = "bookblog";
8      // set the datasource
9      this.datasource = "bookblog";
10     // turn ORM on
11     this.ormenabled = true;
12     // allow ColdFusion to add tables and fields
13     this.ormsettings.dbcreate = "update";
14 }
```

Line 13 of the updated *Application.cfc* now has the new setting:

```
13     this.ormsettings.dbcreate = "update";
```

This setting tells ColdFusion that you want Hibernate to be able to update the database schema. This means that it will add any new tables or columns that you reference in your CFCs. However, it will not modify any existing columns.

Before we can run our code to see the magic happening, we want to create a quick test script we can call from a browser to test that the `categories` table is created in the database.

test.cfm

```
1  <cfscript>
```

```
2 ORMReload();
3 MyCategory = new model.Category();
4 WriteDump( MyCategory );
5 </cfscript>
```

If you run this script in a browser, you'll see a ColdFusion dump of a new instance of the Category entity as shown below.

component <u>model.Category</u>		
PROPERTIES	id	[empty string]
	title	[empty string]
	METHODS	

The Category instance shows the two properties we defined: `id` and `title`. If we look at the database using a tool such as the RDS dataview in ColdFusion Builder or CFIDEclipse, you'll see that the `categories` table has been created with the two columns which correspond to the entity properties. Note that the column name for the `id` property is `category_pk`

	categories
	category_pk (INT 10) required
	title (VARCHAR 100)

ORMReload

You may have noticed that, right at the top of the *test.cfm* script, I'm calling a function called `ORMReload()`. When you build applications using ORM, ColdFusion will look at the CFCs when the application starts up for the first time and configure Hibernate for you behind the scenes. As this is a relatively intensive process, ColdFusion caches the information in memory so that your applications run faster for subsequent requests. The downside of this is that if you change your persistent CFCs, ColdFusion won't necessarily pick up the changes. This is where `ORMReload()` comes in. `ORMReload()` is a new function in ColdFusion 9 that allows you to programmatically tell ColdFusion to rebuild the cached data relating to Hibernate.

Generating Database Schema

We've just seen that Hibernate can generate database tables and columns based on what is specified for each property. So far we have two columns in the `categories` database table – a

numeric primary key and a varchar(100) column – but applications require other types of data such as decimals and dates. ColdFusion has several other attributes for the property tag that give you much greater control over the generated database schema.

Data Types and Precision

Additional attributes of the property tag allow you to define column datatypes and precision. You can of course choose to set the column precision at a database level rather than adding too much meta-programming code to your CFC: the choice is yours. The blog application we are building is going to need additional entities, so now is a good time to create a *Post.cfc*.

A post is going to need properties for id, title, content, ispublished and datepublished. While we are at it, let's add properties to indicate when the post was created and updated.

model/Post.cfc

```
1  /**
2  * I model a blog post
3  */
4  component persistent="true" table="posts"
5  {
6      // identifier
7      property name="id" fieldtype="id"
8          column="post_pk" generator="native";
9
10     // properties
11     property name="title" length="150";
12     property name="content" ormtype="text";
13     property name="ispublished" ormtype="boolean";
14     property name="datepublished" ormtype="date";
15     property name="created" ormtype="timestamp";
16     property name="updated" ormtype="timestamp";
17 }
```

Reviewing the code we've created for the *Post.cfc*, we have a property for the identifier:

```
7  property name="id" fieldtype="id"
8      column="post_pk" generator="native";
```

This is the same as the *id* property used in the *Category.cfc* except that it is mapped to a database column named *post_pk*. When you specify *fieldtype="id"* ColdFusion will make the assumption that the corresponding database column should be an integer.

The title property is defined as follows:

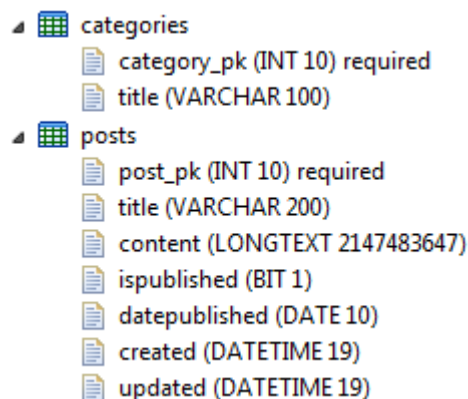
```
11 property name="title" length="150";
```

As we saw earlier, the `length` attribute dictates the number of characters. Accordingly, the database column will have a `varchar` datatype consisting of 150 characters.

```
12 property name="content" ormtype="text";
```

The 'content' property has an `ormtype` attribute. The `ormtype` attribute is used to specify the datatype of the column in the database. If you don't specify the `ormtype` attribute then the default datatype of `varchar` is used (which is why it wasn't needed for the `title` property). As this property has the datatype of `text`, I don't need to specify the `length`. Although the `ormtype` attribute is optional, I would suggest that you always use it to make your code easier to read and maintain.

The rest of the properties for 'ispublished', 'datepublished', 'created' and 'updated' – along with their corresponding `ormtypes` – are pretty straightforward, so I won't go through them. If you run the `test.cfm` script that we created earlier and then view the database schema in RDS dataview, you'll see the following:



Now that we have created some entities that are mapped to database tables, let's start to add and manipulate database records.

Just before we do that, we are going to be making a lot of code changes that we want to be picked up while we are developing. This is a common requirement, and as such it has become routine practice for many developers to add an `ORMReload()` call to the `onRequestStart()` method of *Application.cfc*. Let's do that right now before we forget.

Application.cfc

```
1 /**
2  * I am the Application.cfc for my application
```

```
3  */
4  component
5  {
6      // set the unique name of the application
7      this.name = "bookblog";
8      // set the datasource
9      this.datasource = "bookblog";
10     // turn ORM on
11     this.ormenabled = true;
12     // allow ColdFusion to add tables and fields
13     this.ormsettings.dbcreate = "update";
14
15     /**
16     * I run on each request
17     */
18     boolean function onRequestStart( targetPage )
19     {
20         // force ColdFusion to reload ORM settings
21         if ( StructKeyExists( url, "ormreload" ) )
22         {
23             ORMReload();
24         }
25         return true;
26     }
27 }
```

It's important to note that we only want to use `ORMReload()` during development, so if you do add it to your *Application.cfc* you should make it conditional, so that you don't accidentally deploy it to a production environment. In the example above I am checking for the existence of an 'ormreload' URL parameter, and if it exists, then the `ORMReload()` function is called.

Annotations

ColdFusion supports an alternative style for defining the property attributes, which is to use annotations. Annotations are used in the JavaDoc style comments and are prefixed with an `@` sign. For example, if I were to rewrite *Post.cfc* to use annotations it would look like this:

model/Post.cfc

```
1  /**
2  * I model a blog post using annotations
3  * @persistent true
4  * @table posts
5  */
```

```
6 component
7 {
8     /**
9     * @fieldtype id
10    * @column post_pk
11    * @generator native
12    */
13    property id;
14
15    /**
16    * @ormtype string
17    * @length 150
18    */
19    property title;
20
21    /**
22    * @ormtype text
23    */
24    property content;
25
26    /**
27    * @ormtype boolean
28    */
29    property ispublished;
30
31    /**
32    * @ormtype date
33    */
34    property datepublished;
35
36    /**
37    * @ormtype timestamp
38    */
39    property created;
40
41    /**
42    * @ormtype timestamp
43    */
44    property updated;
45 }
```

Annotation-based metadata is not unique to ColdFusion: you'll find it in other languages, including Java. However, there is a strong argument that what are essentially comment blocks should not be used to affect functionality. The rest of the examples in this book do not make use

of annotations, but it's an important feature and you can of course write your code this way if you prefer.

Chapter 2 : ORM is CRUD

CRUD is an acronym that stands for Create, Read, Update and Delete. This is an area where ColdFusion ORM excels, and as such you should be able to reduce the time you spend building your applications.

At the moment, we have no data in our database; let's rectify that by adding a record using the power of ColdFusion and Hibernate.

Creating Records

In the previous chapter, we created a *test.cfm* script that dumped out an instance of a Category. If you don't remember, here is the code. Note that as `ORMReload()` is now being called in *Application.cfc* I've removed it from the *test.cfm* script.

test.cfm

```
1 <cfscript>
2 MyCategory = new model.Category();
3 WriteDump( MyCategory );
4 </cfscript>
```

When you run this code and expand out the dump you'll see that there are four methods. A really nice feature of ColdFusion 9 is that when you define a property for an entity, a corresponding 'get' and 'set' method is automatically created. As we have `id` and `title` properties, then public `getId()`, `getTitle()`, `setId( id )` and `setTitle( title )` methods are created for you. These methods are often referred to as getters and setters and can be overwritten. You may also hear the terms *accessors* and *mutators* used instead of getters and setters.

component <u>model.Category</u>		
PROPERTIES	id	[empty string]
	title	[empty string]
METHODS	GETID	
	GETTITLE	
	SETID	
	SETTITLE	

The 'getters' retrieve the instance property values (the internal variable values) and the 'setters' are used to change the property values of the entity.

Disabling setters

There are a couple of things that I want to correct before we go any further. Firstly, the `id` of the `Category` entity is being generated for us by the database, so we really don't want to have a `setId()` method. You can tell ColdFusion not to automatically create the `setId()` method by adding `setter="false"` to the `id` property like so:

model/Category.cfc

```
1  /**
2   * I model a blog category
3   */
4   component persistent="true" table="categories"
5   {
6       // identifier
7       property name="id" fieldtype="id"
8           generator="native" column="category_pk"
9           setter="false";
10
11       // properties
12       property name="title" length="100";
13   }
```

If you dump out the category again, you'll see that there is now no longer a `setId()` method.

EntityNew

The second point I want to address is that I want to use the `EntityNew()` function to create a new entity instance.

test.cfm

```
1  <cfscript>
2  MyCategory = EntityNew( "Category" );
3  WriteDump( MyCategory );
4  </cfscript>
```

So why would you choose to use `EntityNew()` instead of `CreateObject()` or the `new` keyword which we had previously? Well, it's largely a case of personal choice, but there are some advantages to using `EntityNew()`. Firstly, you don't need to specify the path to your CFC, which will make refactoring easier in the future (entity names are unique across the application).

Secondly, if you see `EntityNew()` in the code, then you know that it's going to return a persistent entity as `EntityNew()` only works with persistent CFCs.

Now that we've sorted out those two quibbles let's actually create a script to insert something into the database.

EntitySave

Once you have created a new entity, you need to be able to populate it and then have ColdFusion save it to the database. This is achieved with the `EntitySave()` method.

addcategory.cfm

```
1 <cfscript>
2 MyCategory = EntityNew( "Category" );
3 MyCategory.setTitle( "Category A" );
4 EntitySave( MyCategory );
5 </cfscript>
```

I'll breakdown the code line by line. First, we create a new instance of the `Category` entity and assign it to the variable `MyCategory`:

```
2 MyCategory = EntityNew( "Category" );
```

Then we use the automatically generated `setTitle()` method to set the title:

```
3 MyCategory.setTitle( "Category A" );
```

Finally, we make use of another new function introduced in ColdFusion 9, `EntitySave()`:

```
4 EntitySave( MyCategory );
```

As you may have guessed, calling `EntitySave()` will result in the passed-in category entity's data being inserted into the database – all without you having to write a single line of SQL!

Reading data

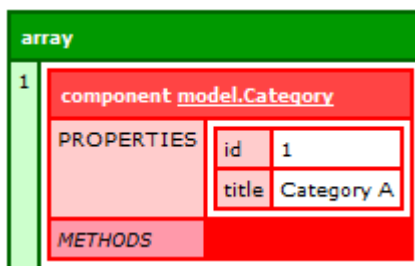
Now that we have successfully added a record to the database, we need to be able to retrieve it again. As you would expect, Hibernate has the ability to load the data from the database and populate the `Category` entity for you. There are several ways to do this; we'll start off by using the new `EntityLoad()` function.

EntityLoad

listcategories.cfm

```
1 <cfscript>
2 categories = EntityLoad( "Category" );
3 writeDump( categories );
4 </cfscript>
```

`EntityLoad()` will retrieve all the database records for the named entity passed to it. Above I've passed in the value of 'category' so if you run the code, you will get an array of populated Category entities.



array											
1	<table border="1"><thead><tr><th colspan="2">component <u>model.Category</u></th></tr></thead><tbody><tr><td>PROPERTIES</td><td><table border="1"><tr><td>id</td><td>1</td></tr><tr><td>title</td><td>Category A</td></tr></table></td></tr><tr><td>METHODS</td><td></td></tr></tbody></table>	component <u>model.Category</u>		PROPERTIES	<table border="1"><tr><td>id</td><td>1</td></tr><tr><td>title</td><td>Category A</td></tr></table>	id	1	title	Category A	METHODS	
component <u>model.Category</u>											
PROPERTIES	<table border="1"><tr><td>id</td><td>1</td></tr><tr><td>title</td><td>Category A</td></tr></table>	id	1	title	Category A						
id	1										
title	Category A										
METHODS											

As there is only one category record in the database, then the array only has one element.

You may notice that the `id` property of the Category entity has a value, although I didn't set one. This is because in *Category.cfc*, I specified in the generator attribute that I wanted the database to generate the primary key for me.

To add some more records I'm going to modify and run the *addcategory.cfm* script I wrote earlier. My new categories will be imaginatively named, 'Category B' and 'Category C'.

Now when I run the *listcategories.cfm* script I get the following output:

array											
1	<table><tr><td colspan="2">component <u>model.Category</u></td></tr><tr><td>PROPERTIES</td><td><table><tr><td>id</td><td>1</td></tr><tr><td>title</td><td>Category A</td></tr></table></td></tr><tr><td>METHODS</td><td></td></tr></table>	component <u>model.Category</u>		PROPERTIES	<table><tr><td>id</td><td>1</td></tr><tr><td>title</td><td>Category A</td></tr></table>	id	1	title	Category A	METHODS	
component <u>model.Category</u>											
PROPERTIES	<table><tr><td>id</td><td>1</td></tr><tr><td>title</td><td>Category A</td></tr></table>	id	1	title	Category A						
id	1										
title	Category A										
METHODS											
2	<table><tr><td colspan="2">component <u>model.Category</u></td></tr><tr><td>PROPERTIES</td><td><table><tr><td>id</td><td>2</td></tr><tr><td>title</td><td>Category B</td></tr></table></td></tr><tr><td>METHODS</td><td></td></tr></table>	component <u>model.Category</u>		PROPERTIES	<table><tr><td>id</td><td>2</td></tr><tr><td>title</td><td>Category B</td></tr></table>	id	2	title	Category B	METHODS	
component <u>model.Category</u>											
PROPERTIES	<table><tr><td>id</td><td>2</td></tr><tr><td>title</td><td>Category B</td></tr></table>	id	2	title	Category B						
id	2										
title	Category B										
METHODS											
3	<table><tr><td colspan="2">component <u>model.Category</u></td></tr><tr><td>PROPERTIES</td><td><table><tr><td>id</td><td>3</td></tr><tr><td>title</td><td>Category C</td></tr></table></td></tr><tr><td>METHODS</td><td></td></tr></table>	component <u>model.Category</u>		PROPERTIES	<table><tr><td>id</td><td>3</td></tr><tr><td>title</td><td>Category C</td></tr></table>	id	3	title	Category C	METHODS	
component <u>model.Category</u>											
PROPERTIES	<table><tr><td>id</td><td>3</td></tr><tr><td>title</td><td>Category C</td></tr></table>	id	3	title	Category C						
id	3										
title	Category C										
METHODS											

As expected, an array of Category entities is returned; but what if we just wanted to get the 'Category B' Category entity? You could simply reference the element we want from the array like so:

```
1 <cfscript>
2 categories = EntityLoad( "Category" );
3 writeDump( categories[ 2 ] );
4 </cfscript>
```

Whilst that does indeed get the Category entity we want, we really don't want to have to get all categories from the database just to get at one of them. It's also important to note that although we are referencing the second element in the array, it doesn't necessarily mean it will be the Category with an id of 2.

Reading Individual Records by ID

To retrieve a single record by its id, we can use the additional optional arguments of `EntityLoad()` to pass in the id of the Category entity we want to retrieve:

```
1 <cfscript>
2 MyCategory = EntityLoad( "Category", 2, True );
```

```
3 writeDump( MyCategory );  
4 </cfscript>
```

The second argument passed to the `EntityLoad()` function is the id (primary key) of the Category entity we want to load. The third argument tells ColdFusion that we don't want an array, just a singular instance. If we omitted the third argument (or set it to false) we would get an array with one element, rather than just the object.

EntityLoadByPK

ColdFusion 9 also provides a more semantic way of loading an entity by id, which is to use the `EntityLoadByPK()` function:

```
1 <cfscript>  
2 MyCategory = EntityLoadByPK( "Category", 2 );  
3 writeDump( MyCategory );  
4 </cfscript>
```

This is my preferred method, as I find it much easier to read and also means that I can't forget to include the third argument that the previous `EntityLoad()` example required.

All of the previous three examples will produce:

component model.Category		
PROPERTIES	id	2
	title	Category B
METHODS		

Loading Filtered Data

A common requirement is to be able to retrieve a subset of data based on certain criteria. In SQL this is usually done by using the where clause. For example, in a blog application you'd want to get all the posts flagged as published. ColdFusion ORM provides a simple way to do this using the `EntityLoad()` method:

```
1 <cfscript>  
2 // load published posts  
3 posts = EntityLoad( "Post", { ispublished=true } );  
4 </cfscript>
```

Previously when I used `EntityLoad()` to retrieve a single `Category` entity, the second argument was a simple value. This time I'm passing in a struct. In the example above, the struct has one key of 'ispublished' with a value of true. ColdFusion will look at this struct and match the key name to the property name and, of course, match the value. The `posts` variable will contain an array of all the `Post` entities that have a value of true for the `ispublished` property. The struct can have multiple keys, allowing you to filter on multiple properties.

Ordering Data

As well as filtering data you may also want to order it. `EntityLoad()` accepts a third argument, which allows you to specify the sort order. It's worth noting that I need to pass in the name of the property that I want to order by and not the database column name.

```
1 <cfscript>
2 filter = { ispublished=true };
3 posts = EntityLoad( "Post", filter, "datepublished desc, title asc" );
4 </cfscript>
```

If you wanted to sort by multiple properties then you would use a comma-delimited list just as you would in an SQL statement. Of course, you may want to return all records but still have them ordered. To do that, you simply pass in an empty struct for the second argument:

```
1 <cfscript>
2 posts = EntityLoad( "Post", {}, "datepublished desc, title asc" );
3 </cfscript>
```

The 'asc' keyword is optional, as ascending is the default sort order, but I have included it for consistency.

Displaying data

So far all the examples I've shown have used `writeDump()` (the cfscript version of the `cfDump` tag) to display the persistent entities, but clearly you'd have some pretty upset customers if you did that in a real application.

Here's a simple script that lists the blog categories ordered by title:

```
1 <!--- get category entities order by title --->
2 <cfset categories = EntityLoad( "Category", {}, "title" )>
3
4 <cfoutput>
```

```
5      <ul>
6          <cfloop array="#categories#" index="theCategory">
7              <li>#theCategory.getTitle()#</li>
8          </cfloop>
9      </ul>
10 </cfoutput>
```

The code above is pretty straightforward. As we saw earlier, each Category entity has a `getTitle()` method, so all we are doing is looping through the array of Category entities and calling the `getTitle()` method of each Category in the array.

Updating Existing Data

The next part of the CRUD acronym is Update. We have already created records in the database, so we need to be able to change that data.

If we were doing this in straight SQL, we'd simply write a query where we update certain fields and use a where clause to determine which record or records are updated. If we wanted to update an existing category record, we'd use something like:

```
1 <cfquery>
2     update CATEGORIES set
3         title = 'new title'
4     where id = 1
5 </cfquery>
```

When you're working with persistent entities, it works differently. Before you can modify any of the data, you first need to load the entity from the database. Once you've done that then you can use the setter methods to change the data. Here's an example:

updatecategory.cfm

```
1 <cfscript>
2 MyCategory = EntityLoadByPK( "Category", 1 );
3 MyCategory.setTitle( "new title" );
4 </cfscript>
```

The first line of code gets the Category with an id of 1. The second line sets the title to 'new title'. That's all you need to do to update the corresponding record in the database.

After you've run this code, if you run *listcategories.cfm* you'll see that the title of the original Category has changed from 'Category A' to 'new title':

array										
1	component <u>model.Category</u> <table><tr><td rowspan="2">PROPERTIES</td><td>id</td><td>1</td></tr><tr><td>title</td><td>new title</td></tr><tr><td colspan="2">METHODS</td><td></td></tr></table>		PROPERTIES	id	1	title	new title	METHODS		
PROPERTIES	id	1								
	title	new title								
METHODS										
2	component <u>model.Category</u> <table><tr><td rowspan="2">PROPERTIES</td><td>id</td><td>2</td></tr><tr><td>title</td><td>Category B</td></tr><tr><td colspan="2">METHODS</td><td></td></tr></table>		PROPERTIES	id	2	title	Category B	METHODS		
PROPERTIES	id	2								
	title	Category B								
METHODS										
3	component <u>model.Category</u> <table><tr><td rowspan="2">PROPERTIES</td><td>id</td><td>3</td></tr><tr><td>title</td><td>Category C</td></tr><tr><td colspan="2">METHODS</td><td></td></tr></table>		PROPERTIES	id	3	title	Category C	METHODS		
PROPERTIES	id	3								
	title	Category C								
METHODS										

As the eagle-eyed amongst you may have noticed, the strange thing about the code I used to update the Category entity is that we didn't use `EntitySave()` to persist the changes to the database. So why and how did the record in the database get updated? The short answer is that ColdFusion ORM, by default, will notice any changes made to a persistent entity (one which has been loaded from the database) any save those changes back to the database automatically.

Let me put it another way: if you use ColdFusion ORM to retrieve and populate an entity from the database and then change something, that change will be saved to the database at the end of the page request regardless of whether you used `EntitySave()`.

This process of monitoring persistent entities for changes, and subsequently committing those changes back to the database, is called automatic dirty checking.

With this newfound knowledge, you may be wondering if we needed to use `EntitySave()` in the *addcategory.cfm* example. The answer is yes. The difference is that, with a new entity, it hasn't been saved to the database yet, and so is not persisted until we explicitly tell ColdFusion to insert it in the database by using `EntitySave()`.

I'll revisit this later on, as you can change the default behaviour of ColdFusion, but it's important to know that this is how it works "out of the box".

Deleting Records

As with the update example, when you want to delete a record from the database, you will need to load the entity first before you can delete it. To tell ColdFusion to delete the corresponding database record for the entity you use the new `EntityDelete()` function.

deletecategory.cfm

```
1 <cfscript>
2 MyCategory = EntityLoadByPK( "Category", 1 );
3 EntityDelete( MyCategory );
4 </cfscript>
```

The code is largely self-explanatory so I won't go into much detail. Suffice to say, we load the category with an id of 1 and then tell ColdFusion to delete it by passing the entity into the `EntityDelete()` method.

One of the things that you'll need to be careful with in ColdFusion ORM is that although I've flagged the entity referenced by `MyCategory` for deletion, the entity will still be available in memory for the remainder of the request. In fact, it may not be deleted from the database until the request has ended. Consider the following example:

```
1 <cfscript>
2 MyCategory = EntityLoadByPK( "Category", 1 );
3 EntityDelete( MyCategory );
4
5 MyCategory.setTitle( "New Title" );
6 </cfscript>
```

If you ran this script, you may expect an error to occur when you try to call the `setTitle()` method on the entity after it has been deleted using `EntityDelete()`. In fact, the in-memory version of `MyCategory` will be modified, but the record will still be deleted from the database. This could cause some confusing behaviour in your application. Because of this you may want to consider deleting the `MyCategory` variable. For example:

```
1 <cfscript>
2 MyCategory = EntityLoadByPK( "Category", 1 );
3 EntityDelete( MyCategory );
4
5 StructDelete( variables, "MyCategory" );
6 </cfscript>
```

By doing this, if you try to modify the persistent entity that has been left in memory after it has been flagged for deletion from the database, then you'll get an error. You don't need to do this, but it may save you some head-scratching later.

Chapter 3 : ORM Sessions

You may be wondering why I'm talking about ORM Sessions before looking at associations (relationships) between entities. Understanding how and when ColdFusion updates and inserts data into the database is an important part of working with Hibernate and I wanted to address this before we get distracted with the more exciting functionality. Before I go any further it is important to note that ORM sessions are not related in any way with the ColdFusion session scope, and they should not be confused with each other.

What are ORM Sessions?

By default, ColdFusion ORM is configured so that Hibernate opens a new ORM session at the start of the request, and subsequently closes it at the end of the request. Any entities that are loaded during the page request are attached to the ORM session. Unlike the ColdFusion session scope, an ORM session only exists for the duration of the page request. All ORM operations happen within the context of an ORM session.

In the *ORM is CRUD* chapter, I demonstrated that when you change any of the instance data of an existing persistent entity, the changes are automatically saved to the database at the end of the request – whether you call `EntitySave()` or not. The ORM session that is opened at the start of the request keeps track of any changes to persistent entities. When the ORM session is closed at the end of the request, the modified entities are persisted back to the database (if they haven't been already). This behaviour isn't ideal for several reasons:

- You may have invalid data in your object that you don't want to save inadvertently
- You aren't in control of when the data is persisted
- Someone reading your code might not realise that the changes will be saved

The first point is crucial, particularly as those changes may not be saved until the request has ended. As a result, you wouldn't notice until the next time the entity is loaded from the database, or – even worse – it may be your client that notices!

The FlushAtRequestEnd Setting

It's not uncommon for developers to populate objects with data and then validate the object. Normally the developer would then choose to persist that object's state only if the validation

passed. With the default ColdFusion 9 ORM settings, if you populate a persistent entity with invalid data, that data will be persisted regardless of whether the validation passed or failed.

For the reasons stated above, I recommend that you override the default behaviour of ColdFusion ORM so that changes are not automatically saved at the end of the request. To do this you need to set the `this.ormsettings.flushAtRequestEnd` flag in *Application.cfc* to false.

Application.cfc

```
1  /**
2  * I am the Application.cfc for my application
3  */
4  component
5  {
6      // set the unique name of the application
7      this.name = "bookblog";
8      // set the datasource
9      this.datasource = "bookblog";
10     // turn ORM on
11     this.ormenabled = true;
12     // allow ColdFusion to add tables and fields
13     this.ormsettings.dbcreate = "update";
14     // turn off automatic ORM session flushing
15     this.ormsettings.flushAtRequestEnd = false;
16     ...
17 }
```

Now that the default behaviour has been disabled, we need to tell ColdFusion that you do want to save the changes to your persistent entity. So how would you do that? Well, the obvious answer would seem to be to use `EntitySave()` like so:

updatecategory.cfm

```
1  <cfscript>
2  MyCategory = EntityLoadByPK( "Category", 1 );
3  MyCategory.setTitle( "new title" );
4  EntitySave( MyCategory );
5  </cfscript>
```

If you run this code, you'll find that the category's title did **not** change in the database. That's right, even though we have used `EntitySave()`, it has no effect on the database. Just as before `EntitySave()` is only for new entities that we want to insert into the database.

There are two possible ways to persist the change of the category's title. They are:

- To use `ORMFlush()`

- To use transactions

ORMFlush

When you call `ORMFlush()` in your code, you are telling ColdFusion that you want to save any dirty (modified) persistent entities that you have in the current request. This is effectively what ColdFusion was doing for you at the end of page requests before you disabled the default behaviour using the `this.ormsettings.flushAtRequestEnd` configuration setting.

If I replace the `EntitySave()` call with `ORMFlush()` in the previous example, then the changes will be saved to the database.

updatecategory.cfm

```
1 <cfscript>
2 MyCategory = EntityLoadByPK( "Category", 1 );
3 MyCategory.setTitle( "new title" );
4 ORMFlush();
5 </cfscript>
```

This seems like a good solution to the problem we had, but there is still a potential problem. Remember that we didn't want to inadvertently save any changes to the database? Well, if we use `ORMFlush()`, then any previously modified Entities in the request will also be saved. Let me demonstrate with a slightly more complicated example that has some basic validation:

```
1 <cfscript>
2 // get category 1
3 MyCategoryA = EntityLoadByPK( "Category", 1 );
4 MyCategoryA.setTitle( "" );
5
6 // do some validation
7 if ( MyCategoryA.getTitle() != "" )
8 {
9     // flush session as entity valid
10    ORMFlush();
11 }
12
13 // get category 2
14 MyCategoryB = EntityLoadByPK( "Category", 2 );
15 MyCategoryB.setTitle( "new title" );
16
17 // do some validation
18 if ( MyCategoryB.getTitle() != "" )
```

```
19 {  
20     // flush session as entity valid  
21     ORMFlush();  
22 }  
23 </cfscript>
```

If we read through this code, you'll see that I load a Category with an id of 1 and set the title to an empty string:

```
3 MyCategoryA = EntityLoadByPK( "Category", 1 );  
4 MyCategoryA.setTitle( "" );
```

Next we do some simple validation to see if the category's title is blank, if it isn't, then we can call `ORMFlush()` to save it:

```
7 if ( MyCategoryA.getTitle() != "" )  
8 {  
9     // flush session as entity valid  
10    ORMFlush();  
11 }
```

As the category's title is set to an empty string, then the `ORMFlush()` is not called.

As execution of the script continues, the Category with an id of 2 is loaded and the category's title is set:

```
13 // get category 2  
14 MyCategoryB = EntityLoadByPK( "Category", 2 );  
15 MyCategoryB.setTitle( "new title" );
```

This time, as the category's title is valid, the simple validation will pass and `ORMFlush()` will be called:

```
18 if ( MyCategoryB.getTitle() != "" )  
19 {  
20     // flush session as entity valid  
21     ORMFlush();  
22 }
```

The moment that the second `ORMFlush()` is called to save the valid Category entity (with an id of 2), the invalid Category entity (with an id of 1) is also saved. Clearly, we need a better way.

ORMClearSession

You can use `ORMClearSession()` to remove all the entities from the current ORM session (remember, ORM sessions are not the same as the ColdFusion session scope) without persisting new or changed entities to the database.

By using `ORMClearSession()`, we can solve the problem of changed entities being accidentally saved to the database. If we update the previous code to use `ORMClearSession()` it will look something like this:

```
1  <cfscript>
2  // get category 1
3  MyCategoryA = EntityLoadByPK( "Category", 1 );
4  MyCategoryA.setTitle( "" );
5
6  // do some validation
7  if ( MyCategoryA.getTitle() != "" )
8  {
9      // flush session as entity valid
10     ORMFlush();
11 }
12 else
13 {
14     // clear the ORM session of modified entities
15     ORMClearSession();
16 }
17
18 // get category 2
19 MyCategoryB = EntityLoadByPK( "Category", 2 );
20 MyCategoryB.setTitle( "new title" );
21
22 // do some validation
23 if ( MyCategoryB.getTitle() != "" )
24 {
25     // flush session as entity valid
26     ORMFlush();
27 }
28 else
29 {
30     // clear the ORM session of modified entities
31     ORMClearSession();
32 }
33 </cfscript>
```

This time, when the second Category passes validation and `ORMFlush()` is called, only the changes made to the second Category entity will be saved. This is because we told ColdFusion

to discard the first modified Category entity from the ORM session because it was invalid, by using `ORMClearSession()`.

Unfortunately, although this is an improvement, it still isn't a perfect solution. If you forget to use `ORMClearSession()`, any subsequent `ORMFlush()` calls will persist modified changes.

Enter transactions!

Transactions

When dealing with updating and inserting data into databases it is a good idea use transactions as they allow you to control when and if data is persisted. This applies equally whether you are using ORM, CFQUERY or pure Java. By wrapping code inside a transaction block you can control what data is persisted to the database, and when.

Taking the previous example and changing it to use transactions we end up with:

```
1  <cfscript>
2  transaction
3  {
4      // get category 1
5      MyCategoryA = EntityLoadByPK( "Category", 1 );
6      MyCategoryA.setTitle( "" );
7
8      // do some validation
9      if ( MyCategoryA.getTitle() != "" )
10     {
11         transaction action="commit";
12     }
13     else
14     {
15         transaction action="rollback";
16     }
17 }
18
19 transaction
20 {
21     // get category 2
22     MyCategoryB = EntityLoadByPK( "Category", 2 );
23     MyCategoryB.setTitle( "new title" );
24
25     // do some validation
26     if ( MyCategoryB.getTitle() != "" )
```

```
27 {  
28     transaction action="commit";  
29 }  
30 else  
31 {  
32     transaction action="rollback";  
33 }  
34 }  
35 </cfscript>
```

If we run this script, Category 1 will not be updated but Category 2 will. If the entity fails validation, then the transaction is rolled back and nothing is committed. If the entity is valid, then it is persisted to the database. You should also note that there are two separate transactions, one for each entity. As each update is wrapped in its own transaction we don't need to worry about previously modified persistent entities accidentally being persisted later in the request.

In ColdFusion, when you end a transaction it is automatically committed if you haven't explicitly called a rollback. As a result, in the example above, we could remove the lines that call the commit on the transaction:

```
1 <cfscript>  
2 transaction  
3 {  
4     // get category 1  
5     MyCategoryA = EntityLoadByPK( "Category", 1 );  
6     MyCategoryA.setTitle( "" );  
7  
8     // do some validation  
9     if ( MyCategoryA.getTitle() == "" )  
10    {  
11        // validation failed  
12        transaction action="rollback";  
13    }  
14 }  
15  
16 transaction  
17 {  
18     // get category 2  
19     MyCategoryB = EntityLoadByPK( "Category", 2 );  
20     MyCategoryB.setTitle( "new title" );  
21  
22     // do some validation  
23     if ( MyCategoryB.getTitle() == "" )  
24    {  
25        // validation failed
```

```
26     transaction action="rollback";
27   }
28 }
29 </cfscript>
```

This code works exactly the same; however, when you read this code, it is not clear that the transaction will be committed if the conditional rollback isn't fired. The choice is yours whether you want to write extra code and provide more clues for other developers as to the code's intent.

Transactions in ColdFusion 9.0.0

Whilst upgrading to ColdFusion 9.0.1 is recommended, it is worth noting that the way ORM transactions work in ColdFusion 9.0.1 has changed since the original 9.0.0 release. If you're stuck with using ColdFusion 9.0.0, then you should manage Hibernate transactions like this:

```
1  <cfscript>
2  // begin transaction
3  ORMTransaction = ORMGetSession().beginTransaction();
4
5  // get category 1
6  MyCategory = EntityLoadByPK( "Category", 1 );
7  MyCategory.setTitle( "" );
8
9  // do some validation
10 if ( MyCategory.getTitle() != "" )
11 {
12     ORMTransaction.commit();
13 }
14 else
15 {
16     ORMTransaction.rollback();
17 }
18 // clear the ORM session
19 ORMGetSession().clear();
20 </cfscript>
```

If you use the ColdFusion 9.0.0 style ORM transactions, then you will need to include a call to the commit method or it won't persist to the database. You'll also need to call `ORMGetSession().clear()` at the end of your transaction block, otherwise the changes may bleed into a subsequent commit.

Chapter 4 : Relationships

So far we have looked at simple isolated entities; however, in a real application there will be relationships between your entities. Consider a blogging application, we would want to be able to add blog posts to a category. In turn those blog posts can have associated comments and an author. This is where relationships come into play.

There are several types of relationship, but the two main ones are one-to-many and many-to-one.

One-to-Many Relationships

An example of a one-to-many relationship is where one Author can have many Posts. Before we can set up a relationship between a Post entity and an Author entity we need to create the *Author.cfc*.

model/Author.cfc

```
1  /**
2  * I model an Author
3  */
4  component persistent="true" table="authors"
5  {
6      property name="id" fieldtype="id" setter="false"
7          column="author_pk" generator="native";
8
9      property name="firstname" length="50";
10     property name="lastname" length="50";
11 }
```

You should be familiar with the code here; if not, it may be worth revisiting the *Mapping Tables to Objects* section earlier in the book.

Now that we have an Author entity, we can define the one-to-many relationship with our existing *Post.cfc*:

model/Author.cfc

```
1  /**
2  * I model an Author
3  */
4  component persistent="true" table="authors"
5  {
```

```
6     property name="id" fieldtype="id" setter="false"
7         column="author_pk" generator="native";
8
9     property name="firstname" length="50";
10    property name="lastname" length="50";
11
12    // one Author can have many Posts
13    property name="posts" fieldtype="one-to-many"
14        cfc="Post" fkcolumn="fk_authorid" type="array";
15 }
```

I have added a property for posts. This time the `fieldtype` attribute is used to specify that I want a 'one-to-many' relationship:

```
13     property name="posts" fieldtype="one-to-many"
```

On the following line is the `cfc` attribute which defines which CFC we want to have the relationship with; in this instance it is the *Post.cfc* so we enter a value of 'Post':

```
14         cfc="Post" fkcolumn="fk_authorid" type="array";
```

The `fkcolumn` attribute is used to say which column name in the `posts` database table will be used to store the Author id. Finally, the `type` attribute is set to `array`, because we want to return associated *Post* entities as an array.

It is important to note that the `cfc` attribute must refer to another entity. If you link to a normal CFC (one without the persistent attribute), then you will get an error. In the example above I'm assuming that the *Post.cfc* is in the same directory (or package) as the *Author.cfc*. If the *Post.cfc* were in a different directory, I would need to use the dotted path syntax.

If this all seems like a lot to remember, then don't worry too much: I often find myself looking this up; the important thing is to understand what the attributes do. It is worth noting that in *Post.cfc* we do not reference the `fk_authorid` database column. This is because Hibernate is handling the relationships between database tables for you so you don't need to in your code.

Generated Methods for Handling Relationships

If we dump out an instance of the *Author* entity and expand out the methods section, you'll see that ColdFusion has automatically created some new methods to handle the relationship between the *Author* and *Post* entities.

component <u>model.Author</u>		
PROPERTIES	id	[empty string]
	firstname	[empty string]
	lastname	[empty string]
	posts	[empty string]
METHODS	ADDPSTS	
	GETFIRSTNAME	
	GETID	
	GETLASTNAME	
	GETPOSTS	
	HASPOSTS	
	REMOVEPOSTS	
	SETFIRSTNAME	
	SETLASTNAME	
	SETPOSTS	

These new methods allow you to add, remove and check for the existence of associated Post entities. The methods are:

The `addPosts()` method – allows you to add a Post entity to the Author entity. The method expects a Post entity to be passed in as the only argument. It does not return a value.

The `getPosts()` method – used to retrieve all the associated Post entities as an array.

The `hasPosts()` method – used to determine if the Author entity has any associated Post entities. It returns a boolean value. If you pass in an optional Post entity, then you can check to see if the passed Post entity is already associated to the Author. If you don't pass in any arguments, it will return true if the Author has any associated Post entities.

The `removePosts()` method – allows you to remove a Post entity from the Author. You will need to pass in the Post entity that you want to be removed. If you pass in a Post entity that is not associated to the Author, then Hibernate will ignore it and not throw an error. This method does not return a value.

The `setPosts()` method – used to associate one or more Post entities to the Author. The Post entities will need to be passed in as an array. This method does not return a value.

At the moment, these methods are not named very semantically. The `addPosts()` and `removePosts()` method only allow you to add or remove one Post at a time, so, they really should be called `addPost()` and `removePost()` respectively. Thankfully we can correct this by making use of the `singularname` attribute.

The Singularname attribute

The `singularname` attribute can be used on one-to-many and many-to-many relationships to provide better names for the generated methods. I'll update the `posts` property in `Author.cfc` to use the `singularname` attribute.

model/Author.cfc

```
1  /**
2  * I model an Author
3  */
4  component persistent="true" table="authors"
5  {
6      property name="id" fieldtype="id" setter="false"
7          column="author_pk" generator="native";
8
9      property name="firstname" length="50";
10     property name="lastname" length="50";
11
12     // one Author can have many Posts
13     property name="posts" fieldtype="one-to-many"
14         cfc="Post" fkcolumn="fk_authorid" type="array"
15         singularname="post";
16 }
```

Now when the entity is dumped out the method names have changed to be `addPost()`, `removePost()` and `hasPost()`.

component <u>model.Author</u>		
PROPERTIES	id	[empty string]
	firstname	[empty string]
	lastname	[empty string]
	posts	[empty string]
METHODS	ADDPOST	
	GETFIRSTNAME	
	GETID	
	GETLASTNAME	
	GETPOSTS	
	HASPOST	
	REMOVEPOST	
	SETFIRSTNAME	
	SETLASTNAME	
	SETPOSTS	

A nice touch is that `setPosts()` and `getPosts()` are still named using the plural, which, as they deal with arrays of entities makes semantic sense.

Adding Entities to a One-to-Many Relationship

Now that the one-to-many relationship has been created between the Author and Post entities, it's time to see it in action.

onetomany.cfm

```
1 <cfscript>
2 transaction
3 {
4     // create a new Author
5     SomeAuthor = EntityNew( "Author" );
6     SomeAuthor.setFirstName( "John" );
7     SomeAuthor.setLastName( "Whish" );
8
9     // create a new Post
10    SomePost = EntityNew( "Post" );
11    SomePost.setTitle( "Post 1" );
12    SomePost.setContent( "Lorem ipsum dolor amet." );
13    SomePost.setIsPublished( true );
```

```
14     SomePost.setDatePublished( Now() );
15
16     // add the Post to the Author
17     SomeAuthor.addPost( SomePost );
18
19     // save the new Author
20     EntitySave( SomeAuthor );
21
22     // save the new Post
23     EntitySave( SomePost );
24 }
25
26 WriteDump( SomeAuthor );
27 </cfscript>
```

If we look at this code, I first create a new instance of an Author entity and populate it:

```
4     // create a new Author
5     SomeAuthor = EntityNew( "Author" );
6     SomeAuthor.setFirstName( "John" );
7     SomeAuthor.setLastName( "Whish" );
```

Next I get a new instance of a Post entity and populate it:

```
9     // create a new Post
10    SomePost = EntityNew( "Post" );
11    SomePost.setTitle( "Post 1" );
12    SomePost.setContent( "Lorem ipsum dolor amet." );
13    SomePost.setIsPublished( true );
14    SomePost.setDatePublished( Now() );
```

Now that the Author and the Post entities have been created, I can assign the Post to the Author using the generated addPost() method:

```
16     // add the Post to the Author
17     SomeAuthor.addPost( SomePost );
```

Finally I tell ColdFusion to persist the Post and Author entities:

```
19     // save the new Author
20     EntitySave( SomeAuthor );
21
22     // save the new Post
23     EntitySave( SomePost );
```

Under the hood Hibernate will generate and execute the SQL statements required to persist the new Author and Post, and also the relationship between them. Hibernate is clever enough to

understand that it needs the primary key from the authors database table for the new Author and to use that id to set the foreign key relationship in the posts table.

The output from the *onetomany.cfm* script will be:

component <u>model.Author</u>																				
PROPERTIES	id	1																		
	firstname	John																		
	lastname	Whish																		
	posts	array																		
	1	<table><tr><th colspan="2">component <u>model.Post</u></th></tr><tr><td rowspan="7">PROPERTIES</td><td>id</td><td>1</td></tr><tr><td>title</td><td>Post 1</td></tr><tr><td>content</td><td>Lorem ipsum dolor sit amet, consectetur adipiscing elit.</td></tr><tr><td>ispublished</td><td>YES</td></tr><tr><td>datepublished</td><td>{ts '2011-07-26 00:00:00'}</td></tr><tr><td>created</td><td>2011-09-03 21:17:21.0</td></tr><tr><td>updated</td><td>2011-09-03 21:17:23.0</td></tr><tr><td colspan="2">METHODS</td></tr></table>	component <u>model.Post</u>		PROPERTIES	id	1	title	Post 1	content	Lorem ipsum dolor sit amet, consectetur adipiscing elit.	ispublished	YES	datepublished	{ts '2011-07-26 00:00:00'}	created	2011-09-03 21:17:21.0	updated	2011-09-03 21:17:23.0	METHODS
component <u>model.Post</u>																				
PROPERTIES	id	1																		
	title	Post 1																		
	content	Lorem ipsum dolor sit amet, consectetur adipiscing elit.																		
	ispublished	YES																		
	datepublished	{ts '2011-07-26 00:00:00'}																		
	created	2011-09-03 21:17:21.0																		
	updated	2011-09-03 21:17:23.0																		
METHODS																				
METHODS																				

In the previous example we only added one Post to the Author but we could easily add several new posts to the Author we've already created by updating the *onetomany.cfm* script.

onetomany.cfm

```

1  <cfscript>
2  transaction
3  {
4      // get an existing Author
5      SomeAuthor = EntityLoadByPK( "Author", 1 );
6
7      // create a new Post
8      SomePost = EntityNew( "Post" );
9      SomePost.setTitle( "Post A" );
10     SomePost.setContent( "Lorem ipsum dolor amet." );
11     SomePost.setIsPublished( true );
12     SomePost.setDatePublished( Now() );
13
14     // add the Post to the Author
15     SomeAuthor.addPost( SomePost );
16

```

```
17      // save the new Post
18      EntitySave( SomePost );
19
20      // create another new Post
21      SomePost = EntityNew( "Post" );
22      SomePost.setTitle( "Post B" );
23      SomePost.setContent( "Lorem ipsum dolor amet." );
24      SomePost.setIsPublished( true );
25      SomePost.setDatePublished( Now() );
26
27      // add the Post to the Author
28      SomeAuthor.addPost( SomePost );
29
30      // save the new Post
31      EntitySave( SomePost );
32  }
33
34  WriteDump( SomeAuthor );
35  </cfscript>
```

The output from running this would be:

component <u>Author</u>																																												
PROPERTIES	id	1																																										
	firstname	John																																										
	lastname	Whish																																										
	posts	array 1 component <u>Post</u> PROPERTIES <table><tr><td>id</td><td>1</td></tr><tr><td>title</td><td>Post 1</td></tr><tr><td>content</td><td>Lorem ipsum dolor sit amet, consectetur elit.</td></tr><tr><td>ispublished</td><td>YES</td></tr><tr><td>datepublished</td><td>{ts '2011-07-27 00:00:00'}</td></tr><tr><td>created</td><td>[empty string]</td></tr><tr><td>updated</td><td>[empty string]</td></tr></table> METHODS 2 component <u>Post</u> PROPERTIES <table><tr><td>id</td><td>2</td></tr><tr><td>title</td><td>Post A</td></tr><tr><td>content</td><td>Lorem ipsum dolor sit amet, consectetur elit.</td></tr><tr><td>ispublished</td><td>YES</td></tr><tr><td>datepublished</td><td>{ts '2011-07-27 14:03:05'}</td></tr><tr><td>created</td><td>[empty string]</td></tr><tr><td>updated</td><td>[empty string]</td></tr></table> METHODS 3 component <u>Post</u> PROPERTIES <table><tr><td>id</td><td>3</td></tr><tr><td>title</td><td>Post B</td></tr><tr><td>content</td><td>Lorem ipsum dolor sit amet, consectetur elit.</td></tr><tr><td>ispublished</td><td>YES</td></tr><tr><td>datepublished</td><td>{ts '2011-07-27 14:03:05'}</td></tr><tr><td>created</td><td>[empty string]</td></tr><tr><td>updated</td><td>[empty string]</td></tr></table> METHODS	id	1	title	Post 1	content	Lorem ipsum dolor sit amet, consectetur elit.	ispublished	YES	datepublished	{ts '2011-07-27 00:00:00'}	created	[empty string]	updated	[empty string]	id	2	title	Post A	content	Lorem ipsum dolor sit amet, consectetur elit.	ispublished	YES	datepublished	{ts '2011-07-27 14:03:05'}	created	[empty string]	updated	[empty string]	id	3	title	Post B	content	Lorem ipsum dolor sit amet, consectetur elit.	ispublished	YES	datepublished	{ts '2011-07-27 14:03:05'}	created	[empty string]	updated	[empty string]
	id	1																																										
	title	Post 1																																										
	content	Lorem ipsum dolor sit amet, consectetur elit.																																										
	ispublished	YES																																										
	datepublished	{ts '2011-07-27 00:00:00'}																																										
	created	[empty string]																																										
updated	[empty string]																																											
id	2																																											
title	Post A																																											
content	Lorem ipsum dolor sit amet, consectetur elit.																																											
ispublished	YES																																											
datepublished	{ts '2011-07-27 14:03:05'}																																											
created	[empty string]																																											
updated	[empty string]																																											
id	3																																											
title	Post B																																											
content	Lorem ipsum dolor sit amet, consectetur elit.																																											
ispublished	YES																																											
datepublished	{ts '2011-07-27 14:03:05'}																																											
created	[empty string]																																											
updated	[empty string]																																											
METHODS																																												

It is worth noting that although we only added two posts to the author, the output shows that the author actually has three posts. This is because when we initially created the author, we also

allocated a post to the new author. As such, subsequent `addPost()` calls, simply adds more Post entities and doesn't affect any Post entities that are already associated with that Author.

If you wanted to replace any existing relationships between the Author and existing posts then you could use the `setPosts()` method.

```
1  <cfscript>
2  transaction
3  {
4      // get an existing Author
5      SomeAuthor = EntityLoadByPK( "Author", 1 );
6
7      // create a new Post
8      SomePostC = EntityNew( "Post" );
9      SomePostC.setTitle( "Post A" );
10     SomePostC.setContent( "Lorem ipsum dolor amet." );
11     SomePostC.setIsPublished( true );
12     SomePostC.setDatePublished( Now() );
13
14     // create another new Post
15     SomePostD = EntityNew( "Post" );
16     SomePostD.setTitle( "Post D" );
17     SomePostD.setContent( "Lorem ipsum dolor amet." );
18     SomePostD.setIsPublished( true );
19     SomePostD.setDatePublished( Now() );
20
21     // save the posts
22     EntitySave( SomePostC );
23     EntitySave( SomePostD );
24
25     // set the Author posts
26     postsArray = [ SomePostC, SomePostD ];
27     SomeAuthor.setPosts( postsArray );
28 }
29
30 WriteDump( SomeAuthor );
31 </cfscript>
```

The output from this script will be:

component <u>Author</u>																														
PROPERTIES	id	1																												
	firstname	John																												
	lastname	Whish																												
	posts	array 1 component <u>Post</u> PROPERTIES <table><tr><td>id</td><td>4</td></tr><tr><td>title</td><td>Post A</td></tr><tr><td>content</td><td>Lorem ipsum dolor sit amet, consectetur elit.</td></tr><tr><td>ispublished</td><td>YES</td></tr><tr><td>datepublished</td><td>{ts '2011-07-27 14:12:11'}</td></tr><tr><td>created</td><td>[empty string]</td></tr><tr><td>updated</td><td>[empty string]</td></tr></table> METHODS 2 component <u>Post</u> PROPERTIES <table><tr><td>id</td><td>5</td></tr><tr><td>title</td><td>Post D</td></tr><tr><td>content</td><td>Lorem ipsum dolor sit amet, consectetur elit.</td></tr><tr><td>ispublished</td><td>YES</td></tr><tr><td>datepublished</td><td>{ts '2011-07-27 14:12:11'}</td></tr><tr><td>created</td><td>[empty string]</td></tr><tr><td>updated</td><td>[empty string]</td></tr></table> METHODS	id	4	title	Post A	content	Lorem ipsum dolor sit amet, consectetur elit.	ispublished	YES	datepublished	{ts '2011-07-27 14:12:11'}	created	[empty string]	updated	[empty string]	id	5	title	Post D	content	Lorem ipsum dolor sit amet, consectetur elit.	ispublished	YES	datepublished	{ts '2011-07-27 14:12:11'}	created	[empty string]	updated	[empty string]
	id	4																												
title	Post A																													
content	Lorem ipsum dolor sit amet, consectetur elit.																													
ispublished	YES																													
datepublished	{ts '2011-07-27 14:12:11'}																													
created	[empty string]																													
updated	[empty string]																													
id	5																													
title	Post D																													
content	Lorem ipsum dolor sit amet, consectetur elit.																													
ispublished	YES																													
datepublished	{ts '2011-07-27 14:12:11'}																													
created	[empty string]																													
updated	[empty string]																													
METHODS																														

As you can see, the Post entities that were allocated to the Author have been replaced with the new Posts. The previously associated post records will still be in the database, but the `fk_authorid` column will have a null value.

Removing Entities from a One-to-Many Relationship

Adding entities to a relationship is only one part of building applications; you'll need to be able to remove them as well.

removepost.cfm

```

1 <cfscript>
2 transaction

```

```
3 {
4     // get an existing Author
5     SomeAuthor = EntityLoadByPK( "Author", 1 );
6
7     // get an existing Post
8     SomePost = EntityLoadByPK( "Post", 1 );
9
10    SomeAuthor.removePost( SomePost );
11 }
12
13 WriteDump( SomeAuthor );
14 </cfscript>
```

The above code is all that you will need to remove the Post with an id of 1 from the one-to-many relationship with the Author. It is worth noting that Hibernate doesn't delete the Post, it simply sets the `fk_authorid` foreign key column in the `posts` table to null. If you pass a Post entity into the `Author.removePost()` method that wasn't associated as part of the one-to-many relationship, then Hibernate will ignore it and not run an update statement against the `posts` table for that record. It won't throw an error.

Choosing the Sort Order

You'll probably want to choose the default sort order of the blog posts returned by the `getPosts()` method. You can do this by using the optional `orderby` attribute, which accepts a comma-delimited list of property names. In a blogging application we'd want to order the associated posts by descending date order. Here is the updated *Author.cfc* with the added `orderby` attribute:

model/Author.cfc

```
1 /**
2  * I model an Author
3  */
4 component persistent="true" table="authors"
5 {
6     property name="id" fieldtype="id" setter="false"
7         column="author_pk" generator="native";
8
9     property name="firstname" length="50";
10    property name="lastname" length="50";
11
12    // one Author can have many Posts
13    property name="posts" fieldtype="one-to-many"
14        cfc="Post" fkcolumn="fk_authorid" type="array"
```

```
15     singularname="post"
16     orderby="datepublished desc";
17 }
```

Now that you have added the `orderby` attribute, whenever you use the `getPosts()` method the array of `Post` entities will be ordered by descending `datepublished`.

Many-to-One Relationships

The `Author` entity knows about its relationship with the `Post` entity, but it would be useful if the `Post` entity knew about its relationship to the `Author` entity. We could then ask a `Post` entity who its author was. As we know, one author can have many posts, so the reverse relationship is that many posts can have one author. In other words the `Post` entity has a many-to-one relationship with the `Author` entity. Let's add this relationship to the *Post.cfc*.

model/Post.cfc

```
1  /**
2   * I model a blog post
3   /
4   component persistent="true" table="posts"
5   {
6       // identifier
7       property name="id" fieldtype="id" setter="false"
8           column="post_pk" generator="native";
9
10      // properties
11      property name="title" length="150";
12      property name="content" ormtype="text";
13      property name="ispublished" ormtype="boolean";
14      property name="datepublished" ormtype="date";
15      property name="created" ormtype="timestamp";
16      property name="updated" ormtype="timestamp";
17
18      // many Post entities can have one Author entity
19      property name="Author" fieldtype="many-to-one" cfc="Author"
20          fkcolumn="fk_authorid";
21  }
```

As you can see, setting up a many-to-one relationship is similar to the one-to-many relationship we used in the *Author.cfc*. The only differences are; the `fieldtype` attribute value is 'many-to-one' and we haven't used `type="array"`, as a `Post` will only ever have one `Author`.

With the addition of this relationship, ColdFusion will automatically generate three new methods, which we can use to control the new one-to-many relationship we've just created. The methods are:

The `setAuthor()` method – allows you to set the Author of a Post entity. The method expects an Author entity to be passed in as the only argument. It does not return a value.

The `getAuthor()` method – returns the associated Author entity for the Post entity. This method has no arguments. If the Post entity does not have an associated Author entity, then a null value is returned.

The `hasAuthor()` method – returns a boolean value to indicate if the Post entity has an associated Author entity.

Let's put these new methods to the test:

getauthor.cfm

```
1 <cfscript>
2 // load a Post
3 SomePost = EntityLoadByPK( "Post", 1 );
4
5 // has it got an Author?
6 if ( SomePost.hasAuthor() )
7 {
8     // show the Author
9     writeDump( SomePost.getAuthor() );
10 }
11 </cfscript>
```

As expected, this simple script will dump out the Author object if the loaded Post entity has an Author. It is important to use the `hasAuthor()` method to check that the Post has an Author before calling the Post's `getAuthor()` method – otherwise you'll get an error.

Using the new `setAuthor()` method is straightforward:

setauthor.cfm

```
1 <cfscript>
2 transaction
3 {
4     // load a Post
5     SomePost = EntityLoadByPK( "Post", 1 );
6     // load an Author
7     SomeAuthor = EntityLoadByPK( "Author", 1 );
```

```
8
9     // set the relationship
10    SomePost.setAuthor( SomeAuthor );
11  }
12 </cfscript>
```

The *setauthor.cfm* script will cause Hibernate to run an update statement on the *posts* table. If the Author was already allocated to the Post then Hibernate is clever enough to realise this and will not run the update statement.

Many-to-Many Relationships

An example of when you would use a many-to-many relationship is a post that can have many categories; likewise, one category can have many posts.

In the actual applications that I have written, I rarely use many-to-many relationships. Instead I prefer to have an intermediary entity between the Post and Category entities that acts as an associative entity. The reason I do it that way is so that I can store additional information about the relationship in the intermediary object. For example, I may wish to record when the relationship was created. This is the same principle as a junction table performs at a database level.

For the purposes of learning I'm going to use a many-to-many relationship between my Post entity and a new Category entity. I want the relationship to be bi-directional, so we'll add the relationship to both the Post and Category CFCs:

Firstly we need to create the *Category.cfc*.

model/Category.cfc

```
1  /**
2   * I model a Category
3   */
4  component persistent="true" table="categories"
5  {
6      // identifier
7      property name="id" fieldtype="id" setter="false"
8          column="category_pk" generator="native";
9
10     // properties
11     property name="title" ormtype="string" length="100";
12
13     // many categories can have many posts
14     property name="posts" fieldtype="many-to-many" cfc="Post"
```

```

15     type="array" singularname="post" orderby="datepublished desc"
16     linktable="posts_categories";
17 }

```

The majority of the code for the *Category.cfc* is pretty simple. The posts property is where we set up the many-to-many relationship to the *Post.cfc*. As expected the `fieldtype` attribute is used to define the type of relationship, with the `cfc`, `type`, `singularname` and `orderby` attributes being used as they are in a one-to-many relationship.

In a relational database, a many-to-many relationship requires the use of a join (or link) table. In the `linktable` attribute, you specify the name of the join table. As before Hibernate can generate this table for you if you have allowed it to.

The link table will need to have two columns, one to store the id of the Category entity, and the second to store the Post entity id. By default these columns will be named using the convention:

`EntityName_PrimaryKeyColumn`

This means that the `posts_categories` database table will have columns named `Category_category_pk` and `Post_post_pk`. If you want to use your own naming convention, then you should add the `fkcolumn` and `inversejoincolumn` attributes.

model/Category.cfc

```

1  /**
2  * I model a Category
3  */
4  component persistent="true" table="categories"
5  {
6      // identifier
7      property name="id" fieldtype="id" setter="false"
8         column="category_pk" generator="native";
9
10     // properties
11     property name="title" ormtype="string" length="100";
12
13     // many categories can have many posts
14     property name="posts" fieldtype="many-to-many" cfc="Post"
15         type="array" singularname="post" orderby="datepublished desc"
16         linktable="posts_categories"
17         fkcolumn="fk_categoryid" inversejoincolumn="fk_postid";
18 }

```

The `fkcolumn` and `inversejoincolumn` attributes are used to name the two columns in the join table. The `fkcolumn` attribute sets the column name in the database table we will be using to

hold the category id in. The `inversejoincolumn` attribute sets the name of the column that will hold the post id.

We now need to set up the relationship from the other side:

model/Post.cfc

```
1  /**
2  * I model a blog post
3  */
4  component persistent="true" table="posts"
5  {
6      // identifier
7      property name="id" fieldtype="id" setter="false"
8          column="post_pk" generator="native";
9
10     // properties
11     ...
12
13     // many Post entities can have one Author entity
14     property name="Author" fieldtype="many-to-one" cfc="Author"
15         fkcolumn="fk_authorid";
16
17     // define a many-to-many relationship with Category
18     property name="categories" fieldtype="many-to-many"
19         cfc="Category"
20         type="array" singularname="Category"
21         fkcolumn="fk_postid" inversejoincolumn="fk_categoryid"
22         linktable="posts_categories";
23 }
```

I've added a new property of categories and defined it as a many-to-many relationship with the Category entity. We want to get an array of categories back, so we set the `type` attribute value to array. By using the `singularname` attribute we can have more semantically accurate names for the methods. The `fkcolumn`, `inversejoincolumn` and `linktable` attributes match those in the Category entity.

If we look at the methods that are now available to us, you'll see that the Post entity methods for controlling the relationship with Category are:

The `addCategory()` method – allows you to add a Category entity to the Post entity. The method expects a Category entity to be passed in as the only argument. It does not return a value.

The `hasCategory()` method – used to determine if the `Post` entity has any associated `Category` entities. It returns a boolean value. If you pass in an optional `Category` entity, then you can check to see if the passed `Category` entity is already associated to the `Post`. If you don't pass in any arguments, then it will return `true` if the `Post` has any associated `Category` entities.

The `removeCategory()` method – allows you to remove a `Category` entity from the `Post`. You will need to pass in the `Category` entity that you want to be removed. If you pass in a `Category` entity that is not associated with the `Post`, then Hibernate will ignore it and not throw an error. This method does not return a value.

Likewise the methods in the `Category` entity for controlling the relationship are:

The `addPost()` method – used to add `Post` entities.

The `hasPost()` method – used to check for the existence of any or a particular `Post` entity.

The `removePost()` method – used to remove specific `Post` entities from the relationship.

The usage of these methods is the same as with many-to-one relationships, so I won't provide code samples. If you need to, you can refer to the previous examples.

One-to-One Relationships

The one-to-one relationship is useful in situations where you have an entity that can have only have a maximum of one occurrence of a related entity. In our blog application there isn't really a good example of where a one-to-one relationship would be used; so I'm going to contrive one just so you can see it in action!

I'm going to add a `Profile` entity. Each `Author` will have one `Profile` and one `Profile` will have one `Author`. Here is the new `Profile.cfc`:

model/Profile.cfc

```
1  /**
2   * I model an Author Profile
3   */
4   component persistent="true" table="profiles"
5   {
6       // identifier
7       property name="id" fieldtype="id" setter="false"
8       column="profile_pk" generator="native";
```

```
9
10 // properties
11 property name="nickname" ormtype="string" length="50";
12 property name="bio" ormtype="text";
13
14 // one-to-one relationship with Author
15 property name="author" fieldtype="one-to-one" cfc="Author"
16     fkcolumn="fk_authorid";
17 }
```

In the *Profile.cfc* there are the usual properties that are now familiar; the lines we are interested in are:

```
14 // one-to-one relationship with Author
15 property name="author" fieldtype="one-to-one" cfc="Author"
16     fkcolumn="fk_authorid";
```

As you'd expect, the `fieldtype` attribute has a value of 'one-to-one' and the `cfc` attribute references the `Author` entity. We also have an `fkcolumn` attribute that defines the column name to create in the profiles table to store the author id.

Now that we have a `Profile` entity, we can create the relationship to it from the `Author` entity:

model/Author.cfc

```
1 /**
2  * I model an Author
3  */
4 component persistent="true" table="authors"
5 {
6     // identifier
7     property name="id" fieldtype="id" generator="native"
8         column="author_pk" setter="false";
9
10    // properties
11    property name="firstname" length="50";
12    property name="lastname" length="50";
13
14    // one author can have many Posts
15    property name="posts" fieldtype="one-to-many" cfc="Post"
16        fkcolumn="fk_authorid" type="array" singularname="post"
17        orderby="datepublished desc";
18
19    // define a one-to-one relationship with Profile
20    property name="profile" fieldtype="one-to-one" cfc="Profile"
21        mappedby="author";
```

As you'd expect, the `fieldtype` attributes value is set to 'one-to-one' and the `cfc` attribute points to the `Profile.cfc`.

There is a new attribute of `mappedby`. The `mappedby` attribute is used to define the property name in the `Profile` entity used to manage the relationship. It is important to note that the value in the `mappedby` attribute is case sensitive and must match the case of the one-to-one property name defined in the *Profile.cfc*.

The one-to-one relationship will create the following methods for the `Author` entity:

`getProfile()` – returns the associated `Profile` entity (if any).

`hasProfile()` – returns true is there is an associated `Profile` entity.

`setProfile()` – used to set the associated `Profile` entity. It does not return a value.

On the other side of the relationship the `Profile` entity has the following methods:

`getAuthor()` – returns the associated `Author` entity (if any).

`hasAuthor()` – returns true is there is an associated `Author` entity.

`setAuthor()` – used to set the associated `Author` entity. It does not return a value.

These methods work the same as we've seen in the many-to-one example so I won't go into detail about them.

The way that I've defined the relationship above means that the `Author id` is stored as a foreign key in the `Profile` database table. As a result, if I wanted to, then I could assign an `Author` profile to a different `Author`. In this scenario it's unlikely that you would ever want to do that, but at least I have the option to do this in future if I so wished. It would be more applicable if you consider a one-to-one relationship between an `Author` and an office cubicle, as the author may change cubicle and another author would want to use the vacated cubicle.

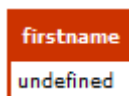
ColdFusion does support an alternative way to set up a one-to-one relationship where both the `Author` and the `Profile` share the same primary key. I don't find that technique particularly flexible (and the syntax is horrible!), so unless you have a specific need to do it that way I'd recommend using the approach I've demonstrated.

Handling Nulls

Because Hibernate is a Java library we have a problem that we're not used to in the ColdFusion world: null values. A null value in simple terms means undefined or unknown. For example if I run the following script:

```
1 <cfscript>
2 Author = EntityNew( "Author" );
3 WriteDump( var=Author.getFirstname(), label="firstname" );
4 </cfscript>
```

I'll get this output to the browser.



If you have debugging turned on then you'll probably also see the exception message:

Element VAR is undefined in ATTRIBUTES.

This happens because no value has been set for the `firstname` property. ColdFusion ORM allows you to set a default value for your properties, by using the `default` attribute.

model/Author.cfc

```
1 /**
2  * I model an Author
3  */
4 component persistent="true" table="authors"
5 {
6     property name="id" fieldtype="id" setter="false"
7         column="author_pk" generator="native";
8
9     property name="firstname" default="";
10    property name="lastname" default="";
11    ...
12 }
```

This time when I run the script I no longer get an undefined error. Unfortunately the `default` attribute is quite limited, as you can only use simple values. If I try to set a default value which is dynamic, for example `'#Now()#'`, then it will cause the error:

Expression in cfproperty value must have a constant value.

The null problem applies to all properties, which means that you will have the same issue with relationships. You will probably want to have an empty array as the default value for a one-to-many relationship. Unfortunately, it's not possible to do this using the `default` attribute.

Due to this limitation you should create an `init()` method to set the defaults. The `init()` method is called automatically by ColdFusion, which treats it as the constructor.

```
1  /**
2  * I model an Author
3  */
4  component persistent="true" table="authors"
5  {
6      property name="id" fieldtype="id" setter="false"
7          column="author_pk" generator="native";
8
9      property name="firstname";
10     property name="lastname";
11
12     property name="posts" fieldtype="one-to-many" cfc="Post"
13         fkcolumn="fk_authorid" singularname="post";
14
15     /**
16     * I am the constructor
17     */
18     any function init()
19     {
20         variables.firstname = "";
21         variables.lastname = "";
22         variables.posts = [];
23     }
24
25 }
```

By using the `init()` method, I can still set the instance variables (property values) for the `firstname` and `lastname` properties as empty string. I've also set the `posts` variable to be an empty array. As the `init()` method is called before ColdFusion populates the entity from the database, any default values you set in the `init()` method will be overwritten.

Chapter 5 : Bi-Directional Associations

We now have a one-to-many relationship between Author and Post, and we also have a many-to-one relationship between Post and Author. Both objects know about the relationship. This is known as a bi-directional relationship.

Uni-Directional versus Bi-Directional

Before I get into the nuances, I should point out that it is perfectly valid to have a one-way relationship (uni-directional), where only one side of the relationship knows about the other entity. For example, does a Comment entity really need to know which Post entity it belongs to for your application to work? A better example of where a uni-directional relationship may be more appropriate than a bi-directional relationship is in a shopping cart, where a product is part of many Orders. You probably wouldn't want to ask a product which orders it belongs to, but you would want to ask an order which products it is composed of.

Managing Relationships

When using bi-directional relationships, you will need to set both sides of the relationship if you want the in-memory versions of your entities to know about each other.

As we've already seen, when you update a persistent entity it is stored in memory until it is committed (if at all) to the database. So if you just set one side of a bi-directional relationship, only that side of the relationship will know about it until it is saved in the database and subsequently retrieved.

The following script creates a new Author and new Post. The `setAuthor()` method of the Post entity is called, but the `addPost()` method of Author is not called.

```
1 <cfscript>
2 transaction
3 {
4     SomeAuthor = EntityNew( "Author" );
5     SomeAuthor.setFirstName( "John" );
6     SomeAuthor.setLastName( "Whish" );
7
8     // get a Post
9     SomePost = EntityNew( "Post" );
10    SomePost.setTitle( "Post 1" );
11    SomePost.setContent( "Lorem ipsum dolor amet." );
```

A Guide to Developing Applications Using ColdFusion ORM

```
12 SomePost.setIsPublished( true );
13 SomePost.setDatePublished( Now() );
14
15 SomePost.setAuthor( SomeAuthor );
16
17 EntitySave( SomePost );
18 EntitySave( SomeAuthor );
19 }
20
21 writeDump( SomeAuthor );
22 writeDump( SomePost );
23 </cfscript>
```

If you run this script, then the Author and Post will both be persisted in the database and the foreign key in the posts database table will be set correctly. However, the in-memory versions of these entities will look like this:

component <u>Author</u>		
PROPERTIES	id	31
	firstname	John
	lastname	Whish
	posts	[empty string]
	METHODS	

component <u>Post</u>															
PROPERTIES	id	31													
	title	Post 1													
	content	Lorem ipsum dolor sit amet, consectetur elit.													
	ispublished	YES													
	datepublished	{ts '2011-07-29 20:32:46'}													
	created	[empty string]													
	updated	[empty string]													
	Author	<table><tr><th colspan="3">component <u>Author</u></th></tr><tr><td rowspan="5">PROPERTIES</td><td>id</td><td>31</td></tr><tr><td>firstname</td><td>John</td></tr><tr><td>lastname</td><td>Whish</td></tr><tr><td>posts</td><td>[empty string]</td></tr><tr><td colspan="2">METHODS</td></tr></table>	component <u>Author</u>			PROPERTIES	id	31	firstname	John	lastname	Whish	posts	[empty string]	METHODS
component <u>Author</u>															
PROPERTIES	id	31													
	firstname	John													
	lastname	Whish													
	posts	[empty string]													
	METHODS														
METHODS															

In the screenshot above you can see that the Author entity does not know about the Post entity, but the Post entity knows about the Author entity. This clearly shows that although the relationship has been correctly stored in the database, the in-memory version of the Author entity does not reflect this.

To overcome this issue, you need to set both sides of the relationship. A quick update to the script will sort that out:

```
1  <cfscript>
2  transaction
3  {
4      // create a new Author
5      SomeAuthor = EntityNew( "Author" );
6      SomeAuthor.setFirstName( "John" );
7      SomeAuthor.setLastName( "Whish" );
8
9      // create a new Post
10     SomePost = EntityNew( "Post" );
11     SomePost.setTitle( "Post 1" );
12     SomePost.setContent( "Lorem ipsum dolor amet." );
13     SomePost.setIsPublished( true );
14     SomePost.setDatePublished( Now() );
15
16     // add the Author to the Post
17     SomePost.setAuthor( SomeAuthor );
18     // add the Post to the Author
19     SomeAuthor.addPost( SomePost );
20
21     // save the new Author
22     EntitySave( SomeAuthor );
23
24     // save the new Post
25     EntitySave( SomePost );
26 }
27 WriteDump( SomeAuthor );
28 WriteDump( SomePost );
29 </cfscript>
```

This time when we run the script we will see that both entities now know about the relationship:

component <u>Author</u>																							
PROPERTIES	id	32																					
	firstname	John																					
	lastname	Whish																					
	posts	array 1 <table border="1"> <thead> <tr> <th colspan="2">component <u>Post</u></th> </tr> </thead> <tbody> <tr> <td rowspan="8">PROPERTIES</td> <td>id</td> <td>32</td> </tr> <tr> <td>title</td> <td>Post 1</td> </tr> <tr> <td>content</td> <td>Lorem ipsum dolor sit amet, consectetur elit.</td> </tr> <tr> <td>ispublished</td> <td>YES</td> </tr> <tr> <td>datepublished</td> <td>{ts '2011-07-29 20:46:22'}</td> </tr> <tr> <td>created</td> <td>[empty string]</td> </tr> <tr> <td>updated</td> <td>[empty string]</td> </tr> <tr> <td>Author</td> <td>[see cfc1 for Author details]</td> </tr> <tr> <td colspan="2">METHODS</td> </tr> </tbody> </table>	component <u>Post</u>		PROPERTIES	id	32	title	Post 1	content	Lorem ipsum dolor sit amet, consectetur elit.	ispublished	YES	datepublished	{ts '2011-07-29 20:46:22'}	created	[empty string]	updated	[empty string]	Author	[see cfc1 for Author details]	METHODS	
	component <u>Post</u>																						
PROPERTIES	id	32																					
	title	Post 1																					
	content	Lorem ipsum dolor sit amet, consectetur elit.																					
	ispublished	YES																					
	datepublished	{ts '2011-07-29 20:46:22'}																					
	created	[empty string]																					
	updated	[empty string]																					
	Author	[see cfc1 for Author details]																					
METHODS																							
METHODS																							

component <u>Post</u>															
PROPERTIES	id	32													
	title	Post 1													
	content	Lorem ipsum dolor sit amet, consectetur elit.													
	ispublished	YES													
	datepublished	{ts '2011-07-29 20:46:22'}													
	created	[empty string]													
	updated	[empty string]													
	Author	component <u>Author</u> <table border="1"> <thead> <tr> <th colspan="2">component <u>Author</u></th> </tr> </thead> <tbody> <tr> <td rowspan="4">PROPERTIES</td> <td>id</td> <td>32</td> </tr> <tr> <td>firstname</td> <td>John</td> </tr> <tr> <td>lastname</td> <td>Whish</td> </tr> <tr> <td>posts</td> <td> array 1 [see cfc1 for Post details] </td> </tr> <tr> <td colspan="2">METHODS</td> </tr> </tbody> </table>	component <u>Author</u>		PROPERTIES	id	32	firstname	John	lastname	Whish	posts	array 1 [see cfc1 for Post details]	METHODS	
	component <u>Author</u>														
PROPERTIES	id	32													
	firstname	John													
	lastname	Whish													
	posts	array 1 [see cfc1 for Post details]													
METHODS															
METHODS															

Having to set both sides of the relationship means that we have in-memory data integrity, but it also means having to write extra code. To alleviate the tedium of doing this every time you are working with a bi-directional relationship, you can create 'convenience' methods by overriding the generated `addPost()` method of the Author entity so that it calls the `setAuthor()` method of the Post entity.

model/Author.cfc

```
1  /**
2  * I model an Author
3  */
4  component persistent="true" table="authors"
5  {
6      property name="id" fieldtype="id" setter="false"
7          column="author_pk" generator="native";
8
9      property name="firstname" length="50";
10     property name="lastname" length="50";
11
12     // one author can have many Posts
13     property name="posts" fieldtype="one-to-many" cfc="Post"
14         fkcolumn="fk_authorid" type="array" singularname="post"
15         orderby="datepublished desc";
16
17     /**
18     * Manages both sides of the relationship
19     */
20     void function addPost( required Post )
21     {
22         if( !hasPost() )
23         {
24             variables.posts = [];
25         }
26         ArrayAppend( variables.posts, arguments.Post );
27         arguments.Post.setAuthor( this );
28     }
29
30 }
```

By doing this I now only have to call the `addPost()` method on the Author entity and no longer need also to call the `setAuthor()` method of the Post entity. If you override the generated `addPost()` method, then it makes sense to override the `removePost()` method as well.

Obviously when you override a generated method to handle both sides of the relationship, make sure that you only do this on one side of the relationship; otherwise you'll get caught in an infinite loop.

SQLIntegrityConstraintViolationException

You may recall that when I ran the previous code example, Hibernate generated and executed the following SQL statements:

```
1  Hibernate:
2      insert into AUTHORS
3          (firstname, lastname)
4      values
5          (?, ?)
6  Hibernate:
7      insert into POSTS
8          (title, content, ispublished, datepublished, created, updated,
9      →  fk_authorid)
10     values
11         (?, ?, ?, ?, ?, ?, ?)
12  Hibernate:
13      update POSTS set
14          fk_authorid=?
15      where post_pk=?
```

What Hibernate does is to insert the Author record and then insert the Post record into the database. When the record is inserted into the posts table, the `fk_authorid` is set as a null value. After the insert, Hibernate then saves the relationship by running an update statement and setting the `fk_authorid` column of the posts table to the Author id.

This isn't ideal: first, because it's not very efficient; and second, because a database administrator (DBA) may want to enforce data integrity in the database and define the `fk_authorid` column so that it does not allow nulls.

If we did have an `fk_authorid` column of the posts table that didn't allow null values, I would get an error if I tried to run the SQL we saw above. The error message will be similar to the following:

```
Root cause :java.sql.SQLIntegrityConstraintViolationException: Column
'FK_AUTHORID' cannot accept a NULL value.
```

This error will be thrown when Hibernate tries to execute the insert statement on the posts table as no value is being passed for the `fk_authorid` column.

Thankfully, there is a solution to this problem and for the inefficiency of running an insert followed by an update on the same table.

The Inverse Attribute

When you have bi-directional relationships, it is important to set one side of the relationship as the 'owner'. This dictates the SQL statements that Hibernate generates and executes to persist relationships in the database. You use the inverse attribute to determine which side of the relationship is in control. Without it your application will be executing unnecessary SQL statements, or (worse) throwing errors due to database foreign keys not allowing null values.

By adding `inverse="true"` to one side of the relationship you determine which side of the relationship is responsible for persisting the relationship in the database. This will eliminate the unnecessary SQL statement. The side that you set the `inverse="true"` flag on is delegating responsibility to the other side of the relationship. The inverse attribute does not affect the in-memory relationship between the objects; it only comes into effect when persisting to the database.

The inverse attribute should be added to the property defined as a 'one-to-many'. In our Author to Post relationship, we need to add the inverse attribute to the 'posts' property of Author.cfc.

model/Author.cfc

```
1  /**
2   * I model an Author
3   */
4   component persistent="true" table="authors"
5   {
6       property name="id" fieldtype="id" setter="false"
7           column="author_pk" generator="native";
8
9       property name="firstname" length="50";
10      property name="lastname" length="50";
11
12      // one author can have many Posts
13      property name="posts" fieldtype="one-to-many" cfc="Post"
14          fkcolumn="fk_authorid" type="array" singularname="post"
15          orderby="datepublished desc"
16          inverse="true";
17
18      /**
19       * Manages both sides of the relationship
20       */
21      void function addPost( required Post Post )
22      {
23          param name="variables.posts" default="#ArrayNew( 1 )#";
24          if( !hasPost( arguments.Post ) )
25          {
26              ArrayAppend( variables.posts, arguments.Post );
```

```
27         arguments.Post.setAuthor( this );
28     }
29 }
30
31 }
```

Running the same code will now generate the following SQL statements:

```
1  Hibernate:
2      insert into AUTHORS
3          (firstname, lastname)
4      values
5          (?, ?)
6  Hibernate:
7      insert into POSTS
8          (title, content, ispublished, datepublished, created, updated,
9      →  fk_authorid)
10     values
11         (?, ?, ?, ?, ?, ?, ?)
```

As you can see there is now no update operation performed against the posts. We therefore don't have to worry about foreign key columns not allowing null values, and we also have better performance.

A word of caution: if you do set the wrong side of the relationship, your changes may not be persisted to the database at all!

Inverse and Many-to-Many

When working with many-to-many relationships you will still need to use the inverse attribute to determine which side of the relationship is in control of persistence to the database. You should also make use of convenience methods so that you don't have to set both sides of the relationship manually as we saw earlier in the chapter.

With a many-to-many relationship you can choose which side is in control of the relationship. I'm going to add the inverse attribute to the categories property of the Post entity.

model/Post.cfc

```
1  /**
2   * I model a blog post
3   */
4  component persistent="true"
5  {
```

```
6     property name="id" fieldtype="id" setter="false"
7         column="post_pk" generator="native";
8
9     property name="title";
10    property name="content";
11
12    // define a many-to-one relationship with Author
13    property name="Author" fieldtype="many-to-one" cfc="Author"
14        fkcolumn="fk_authorid";
15
16    // define a many-to-many relationship with Category
17    property name="categories" fieldtype="many-to-many"
18        cfc="Category"
19        type="array" singularname="Category"
20        fkcolumn="fk_postid" inversejoincolumn="fk_categoryid"
21        linktable="CategoryPosts"
22        inverse="true";
23 }
```

As this is now set as the inverse side, I'm also going to add convenience methods to the *Post.cfc*:

model/Post.cfc

```
1  /**
2   * I model a blog post
3   */
4  component persistent="true"
5  {
6      property name="id" fieldtype="id" setter="false"
7          column="post_pk" generator="native";
8
9      property name="title";
10     property name="content";
11
12     // define a many-to-one relationship with Author
13     property name="Author" fieldtype="many-to-one"
14         cfc="Author" fkcolumn="fk_authorid";
15
16     // define a many-to-many relationship with Category
17     property name="categories" fieldtype="many-to-many"
18         cfc="Category"
19         type="array" singularname="Category"
20         fkcolumn="fk_postid" inversejoincolumn="fk_categoryid"
21         linktable="CategoryPosts"
22         inverse="true";
23 }
```

```

22
23  /**
24  * I add to both sides of the relationship
25  */
26  void function addCategory( required Category Category )
27  {
28      param name="variables.categories" default="#ArrayNew( 1 )#";
29      if ( !hasCategory( arguments.Category ) )
30      {
31          ArrayAppend( variables.categories, arguments.Category );
32          arguments.Category.addPost( this );
33      }
34  }
35
36  /**
37  * I remove from both sides of the relationship
38  */
39  void function removeCategory( required Category Category )
40  {
41      if( hasCategory( arguments.Category ) )
42      {
43          ArrayDelete( variables.categories, arguments.Category );
44          arguments.Category.removePost( this );
45      }
46  }
47
48  /**
49  * I set from both sides of the relationship
50  */
51  void function setCategories( required array categories )
52  {
53      var Category = "";
54      // loop through existing Categories
55      for ( Category in variables.categories )
56      {
57          if ( !ArrayContains( arguments.categories, Category ) )
58          {
59              Category.removePost( this );
60          }
61      }
62      // loop through passed Categories
63      for ( Category in arguments.categories )
64      {
65          if ( !Category.hasPost( this ) )
66          {

```

```
67         Category.addPost( this );
68     }
69 }
70 variables.categories = arguments.categories;
71 }
72
73 }
```

The `setCategories()` method is more complicated than we've seen before. This is because we need to check for existing Categories in the relationship that need to be removed. In the first loop, I'm iterating through the currently associated Categories; in the second loop I'm iterating through the categories we want to be associated to the Post entity.

Chapter 6 : Cascading

Database foreign key constraints provide the ability for modifications to be passed onto related data. For example, if you delete a blog post the database can delete any other associated records, such as comments. This is known as cascading and it is something that Hibernate supports. By default cascading is disabled, which means that you have to handle changes to related entities yourself. To enable cascading, you simply add the optional cascade attribute.

Before I get into the different types of cascading I'm going to add a new Comment persistent entity.

model/Comment.cfc

```

1  /**
2  * I model a Comment
3  */
4  component persistent="true" table="comments"
5  {
6      property name="id" fieldtype="id" setter="false"
7          column="comment_pk" generator="native";
8
9      property name="postedby" ormtype="string" length="100";
10     property name="comment" ormtype="text";
11     property name="dateposted" ormtype="timestamp";
12 }
```

In the *Post.cfc*, I'm going to create a one-to-many relationship with the new Comment persistent entity:

model/Post.cfc

```

1  /**
2  * I model a blog post
3  */
4  component persistent="true" table="posts"
5  {
6      property name="id" fieldtype="id" setter="false"
7          column="post_pk" generator="native";
8
9      property name="title";
10     property name="content";
11
12     property name="comments" fieldtype="one-to-many" cfc="Comment"
13     fkcolumn="fk_postid" singularname="comment"
```

```
14         orderby="dateposted desc";  
15  
16         ...  
17  
18     }
```

Cascade Types

There are various possible values for the cascade attribute that you can choose from. Each type will cause changes to be cascaded when certain operations occur, or not at all. They are:

- none
- delete
- save-update
- all
- delete-orphan
- all-delete-orphan

I'll outline the usage of each in turn, and demonstrate what each of them does with some practical examples.

Cascade None

At the moment, I don't have a cascade attribute for the one-to-many relationship. If you don't specify a cascade type, then effectively you are setting the cascade option to 'none' and ColdFusion will not automatically update any other records in the database. Up until now, all the code examples for relationships we've looked at so far in this book have been set like this.

Often when you are working with objects, then one object can be dependent on the existence of another object, and without it, the object ceases to exist. A real world example of this exists in a blogging application where a Comment is associated to a Post. If you delete the Post then the Comment is orphaned and has no context, and therefore is no longer required.

When you delete a record in the posts database table that has associated records in the comments database table, Hibernate will set a null value for the foreign key column. This means that you'll have orphaned records in the comments database table, or worse, if the foreign key column does not allow null values you'll get an error.

Because of this, if we did want to delete a Post persistent entity then, we would need to load the Post, call the `getComments()` method and then iterate through the array of Comment persistent entities deleting each one in turn before we could delete the Post persistent entity. This seems like a lot of work just to delete a record from the database and this is the problem that cascade delete addresses.

Cascade Delete Type

Cascade delete means that any database records that are dependent (reference the primary key) on the record you're deleting will be deleted automatically. By simply adding the cascade attribute with a value of 'delete' to the Post entity's one-to-many relationship with the Comment entity, we no longer need to worry about having to delete associated comments ourselves. Here is the updated *Post.cfc*:

model/Post.cfc

```
1  /**
2   * I model a blog post
3   */
4   component persistent="true"
5   {
6       property name="id" fieldtype="id" setter="false"
7           column="post_pk" generator="native";
8
9       property name="title";
10      property name="content";
11
12      property name="comments" fieldtype="one-to-many" cfc="Comment"
13          fkcolumn="fk_postid" singularname="comment"
14          orderby="dateposted desc"
15          cascade="delete";
16
17      ...
18
19  }
```

With this in place, the code to delete a Post with associated Comments is as simple as:

```
1  <cfscript>
2  transaction
3  {
4      Post = EntityLoadByPK( "Post", 1 );
5      EntityDelete( Post );
6  }
```

```
7 </cfscript>
```

Cascade Save-update Type

In several of the examples we've looked at in this book, we've often been creating new transient entities and the calling `EntitySave()` on them to make them persistent before adding them as an association to an existing persistent entity. For example, if we have an instance of a `Post` persistent entity and we add a new `Comment` to it, we'd use the following code:

```
1 <cfscript>
2 transaction
3 {
4     Comment = EntityNew( "Comment" );
5     Comment.setPostedBy( "A.N. Other" );
6     Comment.setDateposted( Now() );
7     Comment.setComment( "Nice post!" );
8
9     // make transient entity persistent
10    EntitySave( Comment );
11
12    Post = EntityLoadByPK( "Post", 2 );
13
14    // associate the Comment to the Post
15    Post.addComment( Comment );
16 }
17 </cfscript>
```

As we are associating a new transient entity (the `Comment`), to an already persistent entity (the `Post`), it is pretty safe to assume that we want to persist the new `Comment` to the database, so we really shouldn't need to call `EntitySave()` explicitly. We can achieve this behaviour by adding a cascade attribute with the value of 'save-update'.

Model/Post.cfc

```
1 /**
2  * I model a blog post
3  */
4 component persistent="true"
5 {
6     property name="id" fieldtype="id" setter="false"
7         column="post_pk" generator="native";
8
9     property name="title";
10    property name="content";
```

```
11
12     property name="comments" fieldtype="one-to-many" cfc="Comment"
13         fkcolumn="fk_postid" singularname="comment"
14         orderby="dateposted desc"
15         cascade="save-update";
16
17     ...
18
19 }
```

With this cascade option set up, we can amend the previous example so that `EntitySave()` is no longer required.

```
1  <cfscript>
2  transaction
3  {
4      Comment = EntityNew( "Comment" );
5      Comment.setPostedBy( "A.N. Other" );
6      Comment.setDateposted( Now() );
7      Comment.setComment( "Nice post!" );
8
9      Post = EntityLoadByPK( "Post", 2 );
10
11      // associate the Comment to the Post
12      // and persist new Comment entity
13      Post.addComment( Comment );
14  }
15  </cfscript>
```

If we run this script, then as expected a new record will be inserted into the `comments` database table.

Cascade All Type

As we've seen the cascade options of `delete` and `save-update` are very useful tools to have in our toolbox, but a lot of the time we will want to have the behaviour of both of them applied to our relationships. In these cases you can use a value of `all` for the cascade attribute. By doing this, transient entities that are associated to a persistent entity will be saved automatically, and orphaned records will be deleted automatically.

Cascade Delete-orphan Type

With the `delete` cascade type, we saw that records in the database are automatically deleted when the owning persistent entity is deleted. The `delete-orphan` cascade type takes this one stage further and deletes records when they are no longer associated to a persistent entity. To give you an example, if a `Post` has a `Comment` and you remove the association between the `Post` and the `Comment`, then that `Comment` will be deleted even though you haven't deleted the `Post`.

In code terms, the following will result in the `Comment` being deleted from the database:

```
1 <cfscript>
2 transaction
3 {
4     Comment = EntityLoadByPK( "Comment", 6 );
5     Post = EntityLoadByPK( "Post", 2 );
6     Post.removeComment( Comment );
7 }
8 </cfscript>
```

If I had run this code with the `cascade` attribute value set to `delete` instead of `delete-orphan` then the `Comment` would still exist in the database, but the foreign key value would have been set to a null value. Because of this behaviour, I would suggest that you use `delete-orphan` with care.

Cascade All-delete-orphan Type

The final option we have for the `cascade` attribute is `all-delete-orphan`. If you use this cascade type, then it will apply the behaviour both the `all` type and the `delete-orphan` type.

Chapter 7 : HQL

HQL is an acronym for Hibernate Query Language. It has an SQL-like syntax, but instead of querying database tables and columns it allows you to query your entities and properties instead.

Before I go any further I'd like to point out that HQL is not intended to replace SQL. HQL is very useful and I encourage you to use and learn it, but if your application requires complex SQL (for your reports, for example) then write it in SQL using good old `cfquery`. As a general rule of thumb: if you don't need to be using rich business objects with behaviour, then consider SQL. Whether you're generating a report or simply populating a drop down list, `cfquery` and SQL could well be the best option.

Getting Started With HQL

We've already looked at getting an array of Post entities from the database using the `EntityLoad()` function. We can do the same thing using HQL to get exactly the same result. As a refresher here is a simple example of retrieving all Post entities using `EntityLoad()`:

```
1 <cfscript>
2 allPosts = EntityLoad( "Post" );
3 </cfscript>
```

If we write that using HQL it will look like:

```
1 <cfscript>
2 allPosts = ORMExecuteQuery( "from Post" );
3 </cfscript>
```

The `ORMExecuteQuery()` function allows you to execute HQL statements and will return the result. Inside the `ORMExecuteQuery()` I have an HQL statement which looks a little bit like SQL, except that I haven't used 'select' like I would in SQL.

If you just say "from Post", ColdFusion will return an array of Post persistent entities, just as it would from the `EntityLoad()` function. Although this statement looks a lot like SQL it is important to realise that I am referencing the entity name (Post) and not the database table name (posts).

Something that you need to be careful about in HQL is that entity names are case sensitive. If your entity names are not in the correct case, you'll get a nasty error message. For example, the following will cause an error:

```
1 <cfscript>
2 // will cause an error as 'Post' is case sensitive
3 allPosts = ORMExecuteQuery( "from post" );
4 </cfscript>
```

However, keywords are not case sensitive so the following will work:

```
1 <cfscript>
2 allPosts = ORMExecuteQuery( "FROM Post" );
3 </cfscript>
```

Using a Where Clause

If you wanted to be able to retrieve a subset of all Post entities then, just as you would with SQL, you simply add 'where' and 'order by' clauses to your HQL statement. To refresh your memory, here is the EntityLoad() we would use to get an array of posts. I'm using the variable of form.ispublished to filter the results, so I'll get all the published posts ordered by the date they were published.

```
1 <cfscript>
2 param name="form.ispublished" default="1";
3
4 posts = EntityLoad(
5     "Post",
6     { ispublished=form.ispublished },
7     "datepublished desc"
8 );
9 </cfscript>
```

Written as HQL we get:

```
1 <cfscript>
2 param name="form.ispublished" default="1";
3
4 posts = ORMExecuteQuery( "
5     from Post
6     where ispublished = #form.ispublished#
7     order by datepublished desc"
8 );
9 </cfscript>
```

The HQL now looks even more like SQL, and just like SQL it is susceptible to SQL injection attacks. With SQL statements you should be using cfqueryparam to protect yourself from SQL injection attacks; a similar approach should be used with HQL.

Named Parameters

As we saw above, inline parameters should be avoided due to SQL injection attacks. Another reason not to use inline parameters is because the database engine you're using won't be able to create an execution plan, which can lead to performance issues.

Instead of using inline parameters, you can use named parameters in your HQL statements:

```
1  <cfscript>
2  param name="form.ispublished" default="1";
3
4  posts = ORMExecuteQuery( "
5      from Post
6      where ispublished = :ispublished
7      order by datepublished desc",
8      { ispublished = form.ispublished }
9  );
10 </cfscript>
```

You may notice that I'm using a second argument of `ORMExecuteQuery()` to pass in a struct of values for the named parameters. In the example above I have used a named parameter of 'ispublished' in the HQL, so I need to pass in a key of `ispublished` with the required value.

Positional Parameters

An alternative approach to named parameters is to use positional parameters.

```
1  <cfscript>
2  param name="form.ispublished" default="1";
3
4  posts = ORMExecuteQuery( "
5      from Post
6      where ispublished = ?
7      order by datepublished desc",
8      [ form.ispublished ]
9  );
10 </cfscript>
```

This time, the second argument passed to `ORMExecuteQuery()` is an array instead of a struct. If you need additional parameters, then you simply add more elements to the array; however, they must be in the same order that they are referenced in the HQL. Personally I think this is harder to read than using named parameters, but you may prefer to write your code this way.

Nicer Looking HQL

As ColdFusion developers, we have been spoiled by being able to write nicely formatted SQL statements neatly wrapped in a `cfquery` tag. With the ColdFusion 9.0.1 updater, you can now write HQL statements the same way. This makes them much easier to read and write, particularly if you have conditional statements in your where clause. If I were to rewrite the previous example that used `ORMExecuteQuery()` to use `cfquery` instead, it would look like this:

```
1 <cfquery name="posts" dbtype="hql">
2     from Post
3     where ispublished = <cfqueryparam value="1">
4     order by datepublished desc
5 </cfquery>
```

You should note that the `cfquery` tag has a `dbtype` attribute which has a value of 'hql'. If you don't specify this then ColdFusion will assume you have written an SQL statement and will try to run it directly against the database, bypassing Hibernate. As we are placing our HQL inside a `cfquery` tag, then we can also take advantage of the `cfqueryparam` tag. You do not need to specify the `sqltype` attribute when using `cfqueryparam` with HQL as Hibernate already knows the datatypes of your properties.

I think you will agree that that is much nicer than the `ORMExecuteQuery()` version, and this is one place where I prefer to use tags instead of `cfscript`. As an added bonus, if you use `cfquery` to execute your HQL statements, the HQL will appear in the ColdFusion page debugging – which it won't if you use `EntityLoad()` or `ORMExecuteQuery()`.

Using Operators in HQL

Something that we couldn't do with `EntityLoad()`, and which we do all the time in SQL statements, is to use binary comparison operators and logical operations. This is where HQL offers real benefits over using `EntityLoad()`.

Below is an example of an HQL query we might have to allow a visitor to search blog posts for a list of keywords:

```
1 <cfquery name="posts" dbtype="hql">
2     from Post
3     where ispublished = <cfqueryparam value="1">
4     <cfloop list="#form.keywords#" index="word" delimiters=" ">
5         and title like <cfqueryparam value="%#word#%">
6     </cfloop>
```

```
7     order by datepublished desc
8 </cfquery>
```

Trying to do this with `ORMExecuteQuery()` would be hard work; in a `cfquery` block it's relatively simple. So, spare a moment for all the poor developers of certain other languages who have to build their SQL statements by concatenating strings and manually code against SQL injection attacks!

HQL supports a huge number of operators, including many that you will have used in SQL statements. These include:

Binary comparison operators, such as:

`=, >=, <=, <>, !=`

Logical operations, such as:

`and, or, not`

Extended comparison operators, such as:

`in, not in, between, is null, is not null, like`

As the usage of these is the same as for SQL statements, I won't demonstrate them. You can find a full list of supported operators in the online Hibernate documentation at:

<http://docs.jboss.org/Hibernate/core/3.3/reference/en/html/queryhql.html>

Using Functions in HQL

Hibernate also allows you to use functions in your HQL statements. A lot of these come from the SQL world and include:

Aggregate functions, such as:

`avg(), sum(), min(), max(), count()`

String functions, such as:

`substring(), trim(), lower(), upper(), length(), locate(), concat()`

Mathematical functions, such as:

`abs(), sqrt(), mod()`

Data type functions, such as:

```
coalesce(), nullif()
```

You can also use case statements.

The usage of these functions is a big subject in itself, so I'm afraid I won't be demonstrating them within this book. For now it is useful to know that most functions and operators that you are currently using in your SQL statements are transferable to HQL.

Relationships in HQL

If you remember, I created a many-to-one relationship between the Post entity and the Author entity. In other words, many posts can have one author.

If we wanted to get all the posts for a specific author, we could do something like:

```
1 <cfscript>
2 Author = EntityLoadByPK( "Author", 2 );
3 posts = Author.getPosts();
4 </cfscript>
```

This will of course work, but what if I only wanted to get all the published posts for the author? This is where HQL helps us out.

```
1 <cfquery name="posts" dbtype="hql">
2     from Post
3     where Author.id = <cfqueryparam value="2">
4         and ispublished = <cfqueryparam value="1">
5 </cfquery>
```

Did you notice the `Author.id` reference in the where clause? One of the really nice things about using HQL (which you can't do with SQL) is that Hibernate already knows about the relationships between your entities (assuming you have defined them, of course!). Therefore instead of having to define the relationship in the HQL as you would in an SQL statement, we can simply refer to the relationship between a post entity and the author entity. Under the hood, ColdFusion ORM will translate this into a join between the database tables and generate the following SQL statement for you:

```
1 select post0_.*
2 from POSTS post0_
3 where post0_.fk_authorid=?
4 and post0_.ispublished=?
```

Hibernate is also clever enough to know that you only want data from the `posts` table and so doesn't reference the `authors` table at all. As a result this query will typically run faster than an SQL statement which has a join in it.

When you are referencing other entities in your HQL, you will need to use the correct case. If I had tried to reference the Author entity using all lowercase then I would have got an error:

Error while executing the Hibernate query.

```
org.Hibernate.QueryException: Unable to resolve path [author.id], unexpected token [author] [ from Post where author.id = ? and ispublished = ? ]
```

The previous example took advantage of the many-to-one relationship we created between the Post and Author entities. If you wanted to only get the authors who had written a post then you can make use of the one-to-many relationship to query against the Author entity. Here's how you could do it:

```
1 <cfquery name="publishedauthors" dbtype="hql">
2     from Author
3     where size( posts ) > <cfqueryparam value="0">
4 </cfquery>
```

Once again we need to be aware of case sensitivity. When we referred to the posts collection in the *Author.cfc* we named the property 'posts' using all lowercase characters, and we need to be consistent with the case in the HQL statement. Using the wrong case will result in an error:

The system has attempted to use an undefined value, which usually indicates a programming error, either in your code or some system code.
Null Pointers are another name for undefined values.

The HQL statement above when executed will return an array of Author entities which have one or more associated Post entities as expected, but what if we had an Author who had written a post but it hadn't been published? At the moment that Author would appear in the array. The obvious answer would seem to be the following:

```
1 <!--- this will cause an error --->
2 <cfquery name="posts" dbtype="hql">
3     from Author
4     where size( posts ) > <cfqueryparam value="0">
5         and posts.ispublished = <cfqueryparam value="1">
6 </cfquery>
```

Sadly, this will result in the error:

```
org.Hibernate.QueryException: illegal attempt to dereference collection
The Illegal attempt to dereference collection error
```

The error messages Hibernate provides are not particularly helpful and this is one that you will probably see at some point. The basic problem is that Hibernate treats the referenced posts as a collection and can't filter on them in the same way that you'd expect in SQL.

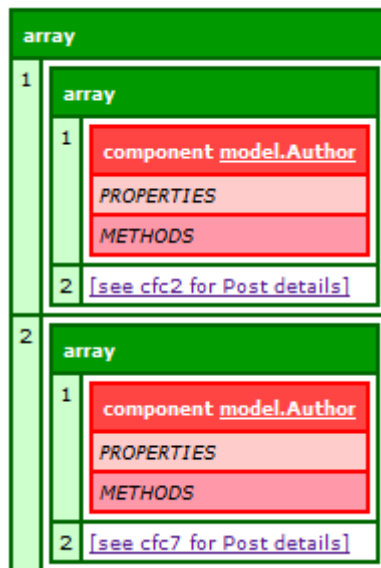
The solution is to use a join, which means that you can filter on properties of the joined Post entity like so:

```
1 <cfquery name="publishedauthors" dbtype="hql">
2   from Author as a
3   join a.posts as p
4   where size( a.posts ) > <cfqueryparam value="0">
5     and p.ispublished = <cfqueryparam value="1">
6   group by a
7 </cfquery>
```

As Hibernate already knows about the relationship, when you use a join in HQL, you don't need to say which properties are used for the join as you would in SQL. As such I can simply reference posts by doing 'join a.posts as p'. I've also aliased the Author entity and posts collection so that I can refer to them in the where clause. Finally I needed to add a 'group by' clause so that I only have unique Author entities returned.

As you can see, in HQL I don't need to reference each individual property of the Author entity; I can just tell Hibernate to group by the Author entity itself by doing group by a.

We now have the desired result of an array of Authors who have had posts published, but we also have a new problem. The result from this HQL when executed is to return an array of arrays containing the Author entities and also the Post entities.



As all we want is to get an array of the Author entities, we can use the select statement to choose what we want to return.

```
1 <cfquery name="publishedauthors" dbtype="hql">
2     select a
3     from Author as a
4     join a.posts as p
5     where size( a.posts ) > <cfqueryparam value="0">
6     and p.ispublished = <cfqueryparam value="1">
7     group by a
8 </cfquery>
```

We finally have what we want, an array of Author entities who have one or more published posts.

array	
1	component <u>model.Author</u> PROPERTIES METHODS
2	component <u>model.Author</u> PROPERTIES METHODS

The HQL we ended up with was more complex than it needed to be, but I thought it was worthwhile to demonstrate some of the issues you may face when writing HQL. For comparison, we could have got the same result by writing the HQL in a few different ways.

By using an EXISTS condition:

```
1 <cfquery name="publishedauthors" dbtype="hql">
2     from Author a
3     where exists (
4         select Author
5         from Post
6         where ispublished = <cfqueryparam value="1">
7         and a = Author
8     )
9 </cfquery>
```

Or, using an IN condition:

```
1 <cfquery name="publishedauthors" dbtype="hql">
2     from Author a
3     where a in (
4         select Author
5         from Post
6         where ispublished = <cfqueryparam value="1">
7     )
8 </cfquery>
```

An even simpler version would be to use the DISTINCT clause:

```
1 <cfquery name="publishedauthors" dbtype="hql">
2     select distinct p.Author
3     from Post p
4     where p.ispublished = <cfqueryparam value="1">
5 </cfquery>
```

Selecting Properties

So far our HQL statements have been returning arrays of objects. This doesn't mean that you can only return objects; you can choose just to return certain properties by using the select clause.

```
1 <cfquery name="result" dbtype="hql">
2     select id, title
3     from Post
4     order by title
5 </cfquery>
```

If I dump the result out you will see the following:

array							
1	<table><tr><th colspan="2">array</th></tr><tr><td>1</td><td>2</td></tr><tr><td>2</td><td>Post 1</td></tr></table>	array		1	2	2	Post 1
array							
1	2						
2	Post 1						
2	<table><tr><th colspan="2">array</th></tr><tr><td>1</td><td>14</td></tr><tr><td>2</td><td>Post 1</td></tr></table>	array		1	14	2	Post 1
array							
1	14						
2	Post 1						
3	<table><tr><th colspan="2">array</th></tr><tr><td>1</td><td>13</td></tr><tr><td>2</td><td>Post 1</td></tr></table>	array		1	13	2	Post 1
array							
1	13						
2	Post 1						

As you can see we have an array of arrays. Working with arrays of arrays is not very enjoyable, and there is the added problem of not knowing which element references which property. A solution to this is to return an array of structs instead.

```
1 <cfquery name="result" dbtype="hql">
2   select new map( id as id, title as title )
3   from Post
4   order by title
5 </cfquery>
```

When I dump this out, we get the following:

array							
1	<table><tr><th colspan="2">struct</th></tr><tr><td>id</td><td>2</td></tr><tr><td>title</td><td>Post 1</td></tr></table>	struct		id	2	title	Post 1
struct							
id	2						
title	Post 1						
2	<table><tr><th colspan="2">struct</th></tr><tr><td>id</td><td>14</td></tr><tr><td>title</td><td>Post 1</td></tr></table>	struct		id	14	title	Post 1
struct							
id	14						
title	Post 1						
3	<table><tr><th colspan="2">struct</th></tr><tr><td>id</td><td>13</td></tr><tr><td>title</td><td>Post 1</td></tr></table>	struct		id	13	title	Post 1
struct							
id	13						
title	Post 1						

This is much easier to work with, but you need to be aware that although it looks like an array of ColdFusion structs, it's actually an array of Java HashMaps. Java HashMaps are case sensitive, so you need to bear this in mind. If you try the following:

```
1 <cfoutput>
2 #result[ 1 ].title#
3 </cfoutput>
```

You will get the error message:

Element TITLE is undefined in a Java object of type class java.util.HashMap.

Instead you need to do this:

```
1 <cfoutput>
2 #result[ 1 ][ "title" ]#
3 </cfoutput>
```

Alternatively you could make the property aliases in your HQL uppercase like so:

```
1 <cfquery name="result" dbtype="hql">
2     select new map( id as ID, title as TITLE )
3     from Post
4     order by title
5 </cfquery>
```

By doing this, you will be able to use either of the following:

```
1 <cfoutput>
2 #result[ 1 ].title#
3 #result[ 1 ][ "TITLE" ]#
4 </cfoutput>
```

Being able to select specific properties from an entity is a useful technique, particularly when you need to improve performance, as only the requested data is retrieved from the database. You can also use functions with properties. For example:

```
1 <cfquery name="result" dbtype="hql">
2     select new map( count( p ) as PostCount )
3     from Post p
4     where p.ispublished = <cfqueryparam value="1">
5 </cfquery>
```

Chapter 8 : Debugging

Hibernate and ColdFusion do an excellent job of abstracting all interaction with the database. The down side to this is that the SQL statements that are being generated and executed are hidden away from you. Being able to view the SQL statements that Hibernate uses is useful for troubleshooting, performance tuning and understanding what Hibernate is doing.

Fortunately, you can log the SQL to either a file or console without too much difficulty. If you are running ColdFusion Builder then you can add your server to the Servers view. Once you have done that you can start and stop your ColdFusion server from within ColdFusion Builder. The added benefit of doing this is that you can see other information that ColdFusion outputs in the console view of ColdFusion Builder along with the SQL.

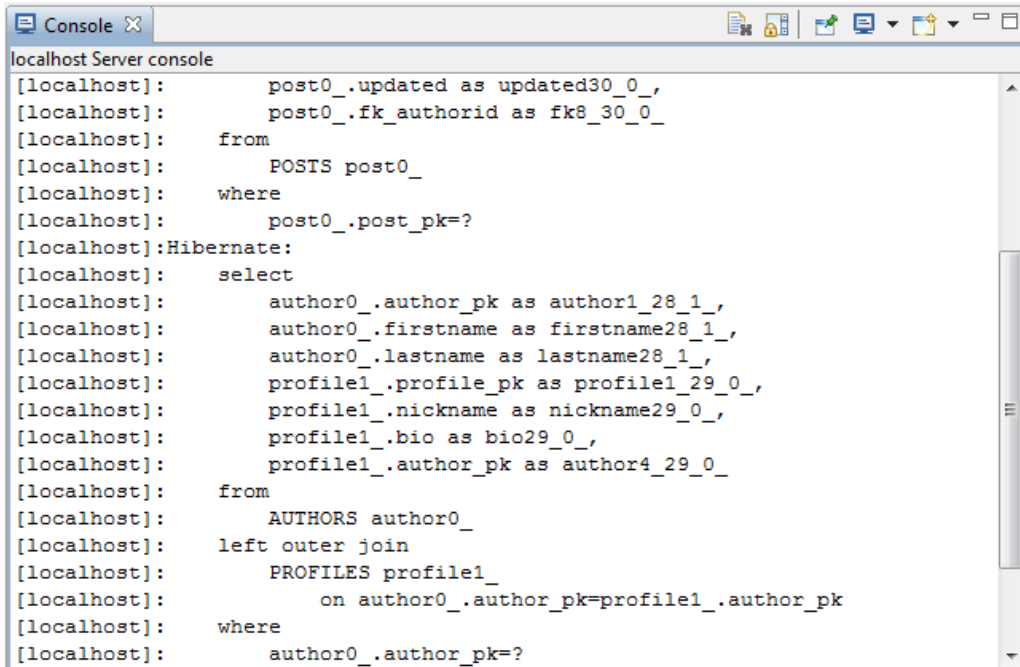
Logging with ColdFusion Builder's Console View

ColdFusion can write the SQL statements that Hibernate is executing to the console. To enable this you simply need to add the `this.ormsettings.logSQL` flag in *Application.cfc* with a value of `true`.

Application.cfc

```
1  /**
2   * I am the Application.cfc for my application
3   */
4   component
5   {
6       // set the unique name of the application
7       this.name = "orm_application";
8       // set the datasource
9       this.datasource = "orm_application";
10      // turn ORM on
11      this.ormenabled = true;
12      // allow ColdFusion to add tables and fields
13      this.ormsettings.dbcreate = "update";
14      // enable SQL logging to the console
15      this.ormsettings.logSQL = true;
16
17      ...
18  }
```

Once you've enabled SQL logging, you will be able to use the console view to watch in real-time the SQL statements as they are executed. An example of the type of output you might see is shown below.

The image shows a screenshot of the 'Console' window in ColdFusion Builder. The window title is 'Console X'. The content area shows the 'localhost Server console' output. The output consists of a series of lines, each starting with '[localhost]:'. The lines represent an SQL query being executed by Hibernate. The query is a SELECT statement that joins the 'POSTS' table with the 'AUTHORS' and 'PROFILES' tables. The query includes various aliases and conditions, such as 'post0_.updated as updated30_0_', 'post0_.fk_authorid as fk8_30_0_', 'author0_.author_pk as author1_28_1_', 'author0_.firstname as firstname28_1_', 'author0_.lastname as lastname28_1_', 'profile1_.profile_pk as profile1_29_0_', 'profile1_.nickname as nickname29_0_', 'profile1_.bio as bio29_0_', and 'profile1_.author_pk as author4_29_0_'. The query also includes a 'left outer join' between 'AUTHORS' and 'PROFILES' based on 'author0_.author_pk=profile1_.author_pk'. The query ends with 'where author0_.author_pk=?'.

```
[localhost]: post0_.updated as updated30_0_,
[localhost]: post0_.fk_authorid as fk8_30_0_
[localhost]: from
[localhost]: POSTS post0_
[localhost]: where
[localhost]: post0_.post_pk=?
[localhost]:Hibernate:
[localhost]: select
[localhost]:     author0_.author_pk as author1_28_1_,
[localhost]:     author0_.firstname as firstname28_1_,
[localhost]:     author0_.lastname as lastname28_1_,
[localhost]:     profile1_.profile_pk as profile1_29_0_,
[localhost]:     profile1_.nickname as nickname29_0_,
[localhost]:     profile1_.bio as bio29_0_,
[localhost]:     profile1_.author_pk as author4_29_0_
[localhost]: from
[localhost]:     AUTHORS author0_
[localhost]: left outer join
[localhost]:     PROFILES profile1_
[localhost]:         on author0_.author_pk=profile1_.author_pk
[localhost]: where
[localhost]:     author0_.author_pk=?
```

Like `ORMReload()`, the `this.ormsettings.logSQL` is useful in development but should not be used in production, as having logging enabled has a negative impact on the performance of your server.

Running ColdFusion from a Console

Of course, you may not wish to, or be able to, use ColdFusion Builder as your day-to-day editor and want to use something else such as CFEclipse, Dreamweaver or even Homesite. This isn't an issue as you can still view the SQL statements by choosing to run ColdFusion from the command line/terminal.

To start ColdFusion so that you can view the console output, you use commands similar to the ones shown below. Please note that the paths may vary depending on the operating system, and the type of install.

Mac users should enter something similar to the following into a terminal window:

```
1 sudo /Applications/ColdFusion9/runtime/bin/jrun -start coldfusion
```

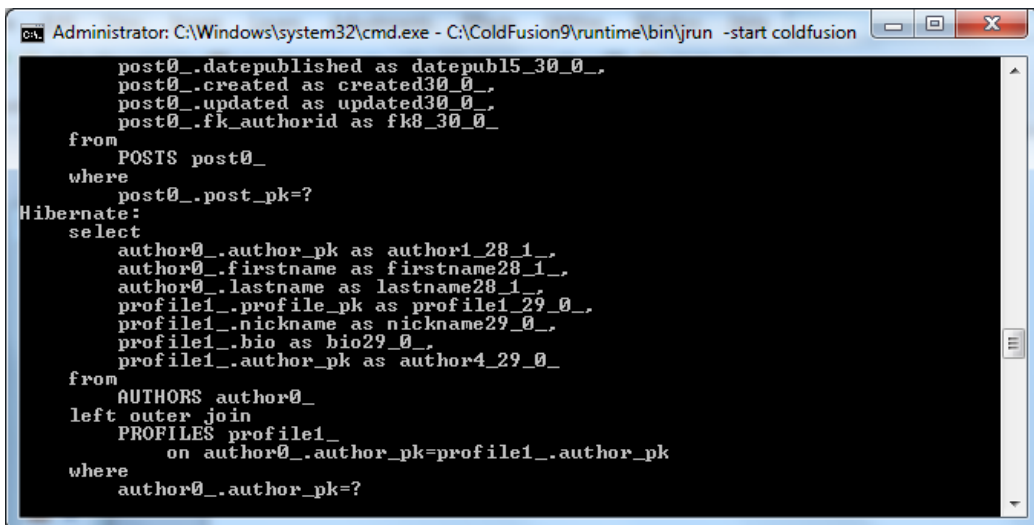
Windows users should enter something similar to the following into a command prompt:

```
1 C:\ColdFusion9\runtime\bin\jrun -start coldfusion
```

Linux users should enter something similar to the following into a terminal window:

```
1 sudo /opt/ColdFusion9/runtime/bin/jrun -start coldfusion
```

When you start ColdFusion from the console, you will be able to view in real time the SQL statements as they are executed. An example of the type of output you might see is shown below:

A screenshot of a Windows command prompt window. The title bar reads "Administrator: C:\Windows\system32\cmd.exe - C:\ColdFusion9\runtime\bin\jrun -start coldfusion". The window has a black background with white text. The text displays SQL query results for a Hibernate query. It shows columns like datepublished, created, updated, fk_authorid, and various aliases for authors and profiles. The output is formatted with tabs and spaces to align columns.

```
Administrator: C:\Windows\system32\cmd.exe - C:\ColdFusion9\runtime\bin\jrun -start coldfusion

post0_.datepublished as datepubl5_30_0_,
post0_.created as created30_0_,
post0_.updated as updated30_0_,
post0_.fk_authorid as fk8_30_0_
from
  POSTS post0_
where
  post0_.post_pk=?
Hibernate:
select
  author0_.author_pk as author1_28_1_,
  author0_.firstname as firstname28_1_,
  author0_.lastname as lastname28_1_,
  profile1_.profile_pk as profile1_29_0_,
  profile1_.nickname as nickname29_0_,
  profile1_.bio as bio29_0_,
  profile1_.author_pk as author4_29_0_
from
  AUTHORS author0_
left outer join
  PROFILES profile1_
    on author0_.author_pk=profile1_.author_pk
where
  author0_.author_pk=?
```

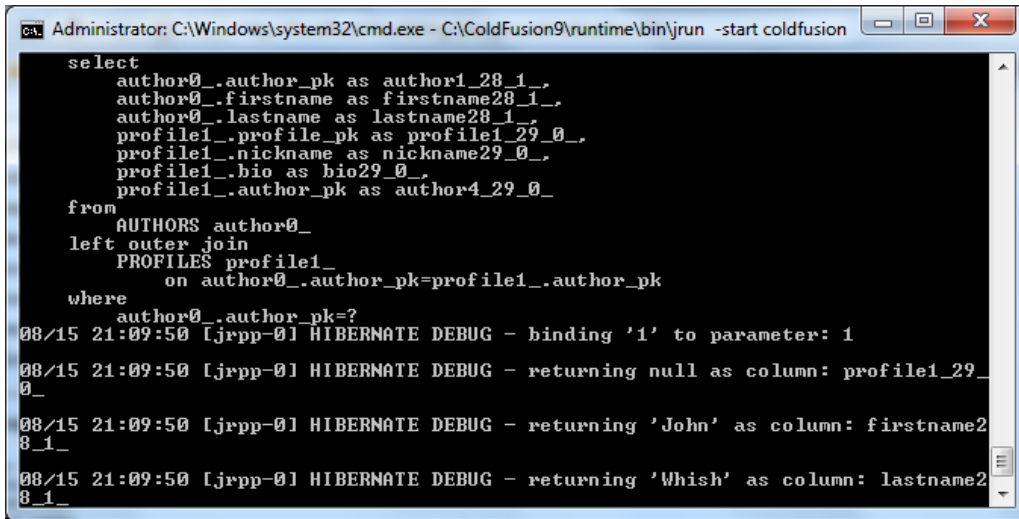
Viewing parameter values

As you may have spotted from the screenshots of the executed SQL statements, the parameter values are not shown. It can be useful to see the parameter values, and this can be achieved with a small tweak to the *log4j.properties* configuration file, which you'll find as part of your ColdFusion install.

If you open up the *log4j.properties* configuration file in a suitable editor, then find the following line:

```
1 #log4j.logger.org.hibernate.type=DEBUG
```

Remove the # prefix and restart the ColdFusion server. You will now see the parameter values, albeit in a rather verbose format, whether you're using ColdFusion Builder's console view or running ColdFusion from the console.



```
Administrator: C:\Windows\system32\cmd.exe - C:\ColdFusion9\runtime\bin\jrun -start coldfusion

select
  author0_.author_pk as author1_28_1_,
  author0_.firstname as firstname28_1_,
  author0_.lastname as lastname28_1_,
  profile1_.profile_pk as profile1_29_0_,
  profile1_.nickname as nickname29_0_,
  profile1_.bio as bio29_0_,
  profile1_.author_pk as author4_29_0_
from
  AUTHORS author0_
left outer join
  PROFILES profile1_
    on author0_.author_pk=profile1_.author_pk
where
  author0_.author_pk=?
08/15 21:09:50 [jrpp-0] HIBERNATE DEBUG - binding '1' to parameter: 1
08/15 21:09:50 [jrpp-0] HIBERNATE DEBUG - returning null as column: profile1_29_0_
08/15 21:09:50 [jrpp-0] HIBERNATE DEBUG - returning 'John' as column: firstname28_1_
08/15 21:09:50 [jrpp-0] HIBERNATE DEBUG - returning 'Whish' as column: lastname28_1_
```

Logging to a File

Occasionally it is useful to be able to log to a file. This can be achieved by adding the following to the *log4j.properties* configuration file:

```
1 log4j.appender.HIBERNATECONSOLE=org.apache.log4j.RollingFileAppender
2 log4j.appender.HIBERNATECONSOLE.File=../hibernatesql.log
3 log4j.appender.HIBERNATECONSOLE.Append=true
4 log4j.appender.HIBERNATECONSOLE.MaxFileSize=500KB
5 log4j.appender.HIBERNATECONSOLE.MaxBackupIndex=3
6 log4j.appender.HIBERNATECONSOLE.layout=org.apache.log4j.PatternLayout
7 log4j.appender.HIBERNATECONSOLE.layout.ConversionPattern=%d{MM/dd HH:mm:ss}
→ [%t] HIBERNATE %-5p - %m%n%n
```

When you restart the ColdFusion server, Hibernate SQL statements will be logged to a file named *hibernatesql.log*, which will automatically start afresh once the log file size reaches 500 KB. This is the same behaviour as ColdFusion uses for other logging operations such as the *exceptions.log* and *application.log*.

If you are using CFEclipse, you can use the Tail view to watch this file.

Viewing HQL Statements in Inline Debugging

I mentioned in the HQL chapter that in ColdFusion 9.0.1 the ability to use the `cfquery` tag for HQL statements was added. You can therefore write your HQL queries like so:

```
1 <cfquery name="myquery" dbtype="hql">
2     from Post
3     where id = <cfqueryparam value="#variables.id#">
4 </cfquery>
```

When you wrap your HQL statements inside a `cfquery` tag you can see the HQL statements that are being executed in your ColdFusion debugging output. This is really useful if you're trying to track down a bug on a production server, as you can temporarily enable ColdFusion page debugging without needing to restart the ColdFusion server, and you can also restrict the output to a specific IP address.

Below is a screenshot of what the inline debugging output looks like. It is important to note that you are seeing the HQL statements and not the SQL statements.

SQL Queries

```
publishedauthors (Datasource=, Time=593ms, Records=0) in C:\inetpub\wwwroot\bookblog\playground\hql.cfm @ 19:35:25.025

    select a
    from Author as a
    join a.posts as p
    where size( a.posts ) > ?
    and p.ispublished = ?
    group by a

Query Parameter Value(s) -
Parameter #1(CF_SQL_CHAR) = 0
Parameter #2(CF_SQL_CHAR) = 1

publishedauthors (Datasource=, Time=15ms, Records=0) in C:\inetpub\wwwroot\bookblog\playground\hql.cfm @ 19:35:30.030

    select distinct p.Author
    from Post p
    where p.ispublished = ?

Query Parameter Value(s) -
Parameter #1(CF_SQL_CHAR) = 1
```

If you really feel like you don't want to be using any ColdFusion tags inside your model, then you can choose to write the previous query entirely in `cfscript`.

```
1 <cfscript>
2 q = new Query();
3 q.setDBType( "hql" );
4 q.setSQL( " from Post where id = :id " );
5
6 q.addParam( name="id", value=1 );
7
8 myquery = q.execute().getResult();
```

```
9 </cfscript>
```

It's a bit more verbose than the tag version, but still allows you to view the HQL statements. The output of this query in your ColdFusion debugging output is shown below:

SQL Queries

```
qryname47654 (Datasource=, Time=15ms, Records=0) in C:\ColdFusion9\CustomTags\com\adobe\coldfusion\base.cfc @ 20:12:51.051
    from Post where id = ?
Query Parameter Value(s) -
Parameter #1(CF_SQL_CHAR) = 1
```

As you can see, the output is very similar; however, you may have noticed that there are two subtle differences between the cfscript version and the tag version. The first is that in the cfscript version the query has a random name. You can solve this by naming the query manually.

```
1 <cfscript>
2 q = new Query();
3 q.setDBType( "hql" );
4 q.setName( "myquery" );
5 q.setSQL( " from Post where id = :id " );
6
7 q.addParam( name="id", value=1 );
8
9 myquery = q.execute().getResult();
10 </cfscript>
```

The second difference is that ColdFusion does not report correctly the name of the script where the code resides. Any HQL (and SQL) queries that you execute using the cfscript version will report as being located in the *CustomTags\com\adobe\coldfusion\base.cfc* file. For my ColdFusion installation it shows as:

```
C:\ColdFusion9\CustomTags\com\adobe\coldfusion\base.cfc
```

Getting HQL using Hibernate Statistics

Another way to view the HQL statements is to take advantage of the Hibernate's own in-built statistics functionality. This approach requires a little more work, but is a powerful technique and provides you with lots of data, and can be controlled programmatically.

You can access and enable Hibernate statistics via the `ORMGetSessionFactory()` method like so:

```
1 <cfscript>
2 Statistics = ORMGetSessionFactory().getStatistics();
```

```
3 Statistics.setStatisticsEnabled( true );
4 </cfscript>
```

Once you have enabled statistics, you can retrieve quite a lot of metrics from Hibernate. We'll have a look at some of these later, but for now we are interested in getting the HQL statements which are available via the `getQueries()` method.

An example of how you could choose to employ this functionality is:

```
1 <cfscript>
2 Statistics = ORMGetSessionFactory().getStatistics();
3 Statistics.setStatisticsEnabled( true );
4
5 posts = ORMExecuteQuery( "from Author where id = :id", { id=1 } );
6
7 q = new Query();
8 q.setDBType( "hql" );
9 q.setSQL( " from Post where id = :id " );
10
11 q.addParam( name="id", value=1 );
12
13 comments = q.execute().getResult();
14 </cfscript>
15
16 <cfquery name="comments" dbtype="hql">
17     from Comment
18     where id = <cfqueryparam value="1">
19 </cfquery>
20
21 <cfdump var="#Statistics.getQueries()#" label="getQueries()">
```

Running this script will result in the following output:

getQueries - array	
1	from Post where id = ?
2	from Author where id = :id
3	from Comment where id = ?

SQL Queries

```
qryname21770 (Datasource=, Time=0ms, Records=0) in C:\ColdFusion9\CustomTags\com\adobe\coldfusion\base.cfc @ 20:37:01.001
    from Post where id = ?

Query Parameter Value(s) -
Parameter #1(CF_SQL_CHAR) = 1

comments (Datasource=, Time=15ms, Records=0) in C:\inetpub\wwwroot\bookblog\playground\getQueries.cfm @ 20:37:01.001
    from Comment
    where id = ?

Query Parameter Value(s) -
Parameter #1(CF_SQL_CHAR) = 1
```

I've also shown the ColdFusion debugging output in the screenshot for comparison. You should note that the output from the `getQueries()` method has three HQL statements. The inline ColdFusion debugging output only shows two HQL statements. The HQL execute using `ORMExecuteQuery()` is not shown. However using the `getQueries()` method means that we are unable to see the values of the parameters being passed in.

I'll leave you to choose which of the solutions I've shown you in this chapter best suits your needs and style of coding.

Chapter 9 : Computed Properties

Computed properties are a useful weapon to have in your arsenal as they can solve some complex problems easily. They come in two flavours, formulas and filters.

Using Formulas

Imagine that we want to be able to get the number of posts associated to a specific Author. This is quite easy by using the `ArrayLen()` function, but doing this is far from optimal in performance terms. We may also only want to get the number of published posts for a specific Author. This is much harder to ascertain without doing loops or writing custom HQL statements. This is where formulas come into play.

We can solve the example of getting the number of published posts from an instance of an Author persistent entity by adding a new property to the *Author.cfc* which makes use of the formula attribute.

model/Author.cfc

```
1  /**
2   * I model an Author
3   */
4  component persistent="true" table="authors"
5  {
6      property name="id" fieldtype="id" generator="native"
7          column="author_pk" setter="false";
8      property name="firstname" length="50";
9      property name="lastname" length="50";
10
11     // get count of active posts
12     property name="publishedpostcount" setter="false"
13         formula="select count(*)
14             from Posts p
15             where author_pk = p.fk_authorid
16             and p.ispublished = 1";
17
18     // one author can have many Posts
19     property name="posts" fieldtype="one-to-many" cfc="Post"
20         fkcolumn="fk_authorid" type="array" singularname="post"
21         orderby="datepublished desc"
22         inverse="true";
```

```
23
24     ...
25
26 }
```

In the example above I've added the `publishedpostcount` property with a `formula` attribute. The value of the `formula` attribute is an SQL statement that gets a count of all the published posts for the current Author. Note that I've referenced the `author_pk` column of the `authors` table although the `authors` table is not mentioned in the SQL statement.

Hibernate uses the SQL statement in the `formula` attribute to create a sub-select. Calling a simple `EntityLoad( "Author" )` will result in Hibernate generating the following SQL statement:

```
1  Hibernate:
2      select author0_.*,
3          (select count(*)
4            from Posts p
5           where author0_.author_pk = p.fk_authorid
6             and p.ispublished = 1) as formula3_0_
7      from AUTHORS author0_
```

ColdFusion will automatically create a `getPublishedPostCount()` method for the Author entity so that you can access the value.

There are a few things that you should be aware of when using formulas. First, the `formula` attribute only accepts SQL statements and will not work with HQL.

Second, since the formula is used to create a sub-select, it can only return one row and one value. If you do use a sub-select which returns more than one row you will get the error:

```
Subquery returns more than 1 row
Root cause :java.sql.SQLException: Subquery returns more than 1 row
```

Third, the value of the property is only populated when the persistent entity is loaded. So if you added a new published post to the Author entity, the value returned by the `getPublishedPostCount()` method will be incorrect. I'll demonstrate what I mean with the following code:

```
1  <cfoutput>
2  <cftransaction>
3      <cfset Author = EntityLoadByPK( "Author", 5 )>
4
5      1. count when loaded = #Author.getPublishedPostCount()#<br />
6  </cfoutput>
```

```
7      <cfset Post = EntityNew( "Post" )>
8      <cfset Post.setTitle( "New Post" )>
9      <cfset Post.setIsPublished( true )>
10     <cfset Author.addPost( Post )>
11
12     <!---
13     Note: new post will be saved by cascade
14     --->
15 </cftransaction>
16
17 2. count after commit = #Author.getPublishedPostCount()#<br />
18
19 <cfquery name="postcount" dbtype="hql">
20     select count(*)
21     from Post
22     where Author = <cfqueryparam value="#Author#">
23     and ispublished = 1
24 </cfquery>
25
26 3. hql query count = #postcount[ 1 ]#<br />
27 4. count after hql = #Author.getPublishedPostCount()#<br />
28
29 </cfoutput>
```

This code is pretty straightforward. I get a persistent Author entity and add a new Post entity to it. When the transaction closes, the new Post entity is persisted (as the Author to Post entity relationship is set up to cascade saves). The rest of code is there just to output the values from the calls to the `getPublishedPostCount()` method and the HQL statement. The output from running this code is:

1. count when loaded = 7
2. count after commit = 7
3. hql query count = 8
4. count after hql = 7

As you can see the HQL statement is returning the correct value, but the values returned from the `getPublishedPostCount()` are not, even after the HQL has executed. This is the expected behaviour, but it can catch you out if you're not aware of it.

EntityReload

Another new function introduced in ColdFusion 9 is `EntityReload()`. This function allows you reload the entity's data from the database. If I update the previous script to include an

EntityReload call we can get the correct values returned from the `getPublishedPostCount()` method.

```
1 <cfoutput>
2 <cftransaction>
3   <cfset Author = EntityLoadByPK( "Author", 5 )>
4
5   1. count when loaded = #Author. getPublishedPostCount()#<br />
6
7   <cfset Post = EntityNew( "Post" )>
8   <cfset Post.setTitle( "New Post" )>
9   <cfset Post.setIsPublished( true )>
10  <cfset Author.addPost( Post )>
11
12  <!---
13  Note: new post will be saved by cascade
14  --->
15 </cftransaction>
16
17 2. count after commit = #Author. getPublishedPostCount()#<br />
18
19 <cfset EntityReload( Author )>
20
21 3. count after reloading = #Author.getPublishedPostCount()#<br />
22 </cfoutput>
```

This time, when the script is run it will return the following output:

1. count when loaded = 8
2. count after commit = 8
3. count after reloading = 9

We now have the correct value after the reload, but it does come with a small performance cost. In order for Hibernate to refresh the Author entity's data, it needs to run the SQL query again.

If you want to avoid having to run the SQL statement again, you can manually update the instance value of the `publishedpostcount` property by overriding the `addPost()` method of the *Author.cfc*.

model/Author.cfc

```
1 /**
2  * I model an Author
3  */
4 component persistent="true" table="authors"
5 {
```

```
6      property name="id" fieldtype="id" generator="native"
7          column="author_pk" setter="false";
8      property name="firstname" length="50";
9      property name="lastname" length="50";
10
11      // get count of active posts
12      property name="publishedpostcount" setter="false"
13          formula="select count(*)
14              from Posts p
15              where author_pk = p.fk_authorid
16              and p.ispublished = 1";
17
18      // one author can have many Posts
19      property name="posts" fieldtype="one-to-many" cfc="Post"
20          fkcolumn="fk_authorid" type="array" singularname="post"
21          orderby="datepublished desc"
22          inverse="true";
23
24      ...
25
26      void function addPost( required Post Post )
27      {
28          param name="variables.posts" default="#ArrayNew( 1 )#";
29          if( !hasPost( arguments.Post ) )
30          {
31              ArrayAppend( variables.posts, arguments.Post );
32              arguments.Post.setAuthor( this );
33              /*
34              update publishedpostcount count.
35              Use ArrayLen instead of simple
36              increment in case other methods have
37              changed variables.posts
38              */
39              variables.publishedpostcount = ArrayLen( variables.posts );
40          }
41      }
42  }
```

Now when I run the previous script I get the following output:

1. count when loaded = 9
2. count after commit = 10
3. count after reloading = 10

The `EntityReload()` is now no longer required as the count after the commit is correct. If you choose to use this technique instead of using `EntityReload()`, you will of course need to override the `setPosts()` and `removePost()` methods in the *Author.cfc* to adjust the count accordingly.

It is still possible for the value after an `EntityReload()` to return a different value as `EntityReload()` gets the values from the database, whereas when we update the value in code, it is the in-memory version. This could happen when you add a post to the Author persistent entity but don't save it (for example if it failed validation). The in-memory version of the Author would have the new post, but the database would not. Alternatively, another process may have modified the database.

Where Filter

As we've already seen, when you created a one-to-many or many-to-many association, then ColdFusion will automatically create a method allowing you to return all associated entities as an array or struct. While this is extremely useful, there will be many situations when you'll only want to retrieve a subset of those persistent entities. This is where you can use the where filter.

If you wanted to get all the published posts from the Author persistent entity, you could add a where attribute to the relationship to filter out the unpublished posts:

model/Author.cfc

```
1  /**
2   * I model an Author
3   */
4   component persistent="true" table="authors"
5   {
6       property name="id" fieldtype="id" setter="false"
7           column="author_pk" generator="native";
8
9       property name="firstname" length="50";
10      property name="lastname" length="50";
11
12      // get count of active posts
13      property name="publishedpostcount" setter="false"
14          formula="select count(*)
15              from Posts p
16              where author_pk = p.fk_authorid
17              and p.ispublished = 1";
18  }
```

```
19 // get published posts
20 property name="publishedposts" fieldtype="one-to-many" cfc="Post"
21     fkcolumn="fk_authorid" orderby="datepublished desc"
22     where="ispublished = 1";
23
24 // one author can have many Posts
25 property name="posts" fieldtype="one-to-many" cfc="Post"
26     fkcolumn="fk_authorid" type="array" singularname="post"
27     orderby="datepublished desc"
28     inverse="true";
29
30 ...
31 }
```

The new publishedposts property is essentially a one-to-many relationship with an additional where attribute. Like the formula attribute, the where attribute accepts an SQL statement. You can have multiple clauses in the SQL statement if you wish, and even sub queries. For example, if you wanted to get all posts for the Author which have at least one category, then you could use the following SQL in the where attribute:

```
19 // get published posts
20 property name="publishedposts" fieldtype="one-to-many" cfc="Post"
21     fkcolumn="fk_authorid" orderby="datepublished desc"
22     where="ispublished = 1
23         and post_pk IN (
24             select pc.fk_postid
25             from posts_categories pc
26         )";
```

By running a simple script such as:

```
1 <cfscript>
2 Author = EntityLoadByPK( "Author", 2 );
3 publishedposts = Author.getpublishedposts();
4 WriteOutput( ArrayLen( publishedposts ) );
5 </cfscript>
```

Hibernate will generate and execute these SQL statements:

```
1 Hibernate:
2     select author0_.*
3     from AUTHORS author0_
4     where author0_.author_pk=?
5 Hibernate:
6     select publishedp0_.*
7     from POSTS publishedp0_
```

```
8      where (
9          publishedp0_.ispublished = 1
10         and publishedp0_.post_pk IN (
11             select pc.fk_postid
12             from posts_categories pc
13         )
14     )
15     and publishedp0_.fk_authorid=?
16     order by publishedp0_.datepublished desc
```

The first SQL statement returns the data from the authors database table; the second retrieves the data from the posts database table. Interestingly, the second statement only runs when I call the `ArrayLen()` function and not when I call the `getPublishedPosts()` method of the Author entity. Hibernate does not load and populate the Post entities until they are actually required.

One of the things that can be confusing about using the where filter is that ColdFusion will generate the `addPublishedPosts()`, `hasPublishedPosts()` and `removePublishedPosts()` methods as you'd expect when creating a one-to-many relationship. However, these methods do not take into account the where clause. This can be a real problem as you could have the following code:

```
1  <cfscript>
2  Author = EntityLoadByPK( "Author", 5 );
3  </cfscript>
4
5  <cfoutput>
6  Has published posts = #Author.hasPublishedPosts()#<br />
7  Array length = #ArrayLen( Author.getPublishedPosts() )#<br />
8  </cfoutput>
```

Running this script will produce contradictory values:

```
Has published posts = YES
Array length = 0
```

As such you should be careful when using this technique, as the automatically generated method names don't really reflect the nature of what the method does.

Chapter 10 : Event Handlers

ColdFusion has something called Event Handlers built in. If you use any of the methods in *Application.cfc*, you already have some familiarity with events being fired at certain points in an application's lifecycle.

Before you can use any of the interception points, you need to enable event handling in your application. As with most ORM settings, this is done in *Application.cfc*.

Application.cfc

```
1  /**
2   * I am the Application.cfc for my application
3   */
4  component
5  {
6      ...
7      this.ormsettings.eventhandling = true;
8      ...
9  }
```

Now that event handling is enabled, you can make use of the following interception points:

`preLoad()` – This event occurs before the entity is populated from the database.

`postLoad()` – This event occurs after the entity is populated from the database.

`preInsert()` – This event occurs before the entity's data is inserted into the database.

`postInsert()` – This event occurs after the entity's data is inserted into the database.

`preUpdate()` – This event occurs before the corresponding database record is updated in the database.

`postUpdate()` – This event occurs after the corresponding database record is updated in the database.

`preDelete()` – This event occurs before the corresponding database record is deleted from the database.

`postDelete()` – This event occurs before the corresponding database record is deleted from the database.

None of these methods returns a value and any of them can be added directly into your entity CFC. You can add any combination of these methods, or none at all. If one of these methods doesn't exist, then ColdFusion will ignore it and carry on.

There is a known bug with ColdFusion 9.0.1 which means that the `preUpdate()` event fires twice, when it should only fire once. This has been fixed in the Cumulative Hotfix 1 (CHF1) and is included in Cumulative Hotfix 2 (CHF2) for ColdFusion 9.0.1.

More information about Cumulative Hotfix 2 (CHF2) for ColdFusion 9.0.1 is available at:

http://kb2.adobe.com/cps/918/cpsid_91836.html

In the rest of the chapter, I'll be dealing with some simple examples of using event handlers, but with a little imagination you'll see that they can be pretty powerful. For example you could use them for detailed auditing, where you would track when a change was made, who made the change and even what the changes were.

One use of event handlers that I use quite a lot in my applications is for keeping track of when an entity was created and updated.

You may be thinking that you can do this anyway by simply calling setters on the entity when you create or update the entity. This is true, but you'll need to do this everywhere in your code that you create or update the entity. It's much easier – and reduces the amount of code you need to write – to use event handlers.

Before we can track when an entity has been updated or created, we will need two new properties to hold this data, I'm going to call them `created` and `updated` respectively. I don't want the values to be altered from outside the entity, so I'm going to disable the setters.

model/Post.cfc

```
1  /**
2   * I model a Blog Post
3   */
4  component persistent="true" table="posts"
5  {
6      // the id
7      property name="id" fieldtype="id" setter="false"
8          column="post_pk" generator="native";
9
10     ...
11
12     property name="created" ormtype="timestamp" setter="false";
13     property name="updated" ormtype="timestamp" setter="false";
14 }
```

```
15     ...
16
17 }
```

Now that we have a created and updated property, we can add the event handlers. As we are interested in having a time stamp for when the entity is created and also when it is updated, we need to use the `preInsert()` and `preUpdate()` methods.

model/Post.cfc

```
1  /**
2   * I model a Blog Post
3   */
4  component persistent="true" table="posts"
5  {
6     ...
7
8     void function preInsert()
9     {
10         variables.created = Now();
11         variables.updated = variables.created;
12     }
13     void function preUpdate()
14     {
15         variables.updated = Now();
16     }
17 }
```

That's it! As event handling is enabled in our application, these new methods will be called automatically at the appropriate moment in the lifecycle without us having to write any additional code. The only thing that you need to be aware of is that the methods must be public. I've not actually set the access type to public in the above code, as the default access type is public in ColdFusion. You don't even need to set a return type of void as this is also the default, but I prefer to add it to make the code more descriptive; as always, the choice is yours.

As you can see, it is very easy to use event handlers in your code. However, if we wanted to have every entity track when it was created or updated, it would become rather tedious to add the `preInsert()` and `preUpdate()` methods to every single entity.

The Global Event Handler

If you want to have the same events fire on all the entities in your application, you should consider using the global event handler. The global event handler is a place where you can add

code that will fire whenever any entity in your application is loaded or persisted. To enable global event handling, you simply have to set the path to the component (CFC) that you want to use as the global event handler.

Application.cfc

```
1  /**
2  * I am the Application.cfc for my application
3  */
4  component
5  {
6      ...
7      this.ormsettings.eventhandling = true;
8      this.ormsettings.eventhandler = "model.aop.GlobalEventHandler";
9      ...
10 }
```

Clearly we now need to create the CFC that we are pointing to! As ColdFusion is expecting to find specific methods, and for those specific methods to accept specific arguments, you are required to implement an interface. Don't worry if you're not familiar with the concept of interfaces (they are not widely adopted in dynamic languages like ColdFusion) – all the interface is doing is making sure that your *GlobalEventHandler.cfc* has the correct methods and signatures.

Your *GlobalEventHandler.cfc* will need to have the `implements` attribute set to "cfide.orm.IEventHandler". This is mandatory; if you don't add this, you'll get an error:

```
model.aop.GlobalEventHandler is not a valid ORM event handler.
It must implement interface cfide.orm.IEventHandler
```

Below is my global event handler. I've added in some basic auditing which simply logs events to a file. You'll also notice that the `preInsert()` and `preUpdate()` methods now expect an entity to be passed in. I've also added a check to ensure that the passed entity has public `setCreated()` or `setUpdated()` methods before calling them.

The `logEvent()` method is not a built-in method; I've just added it to handle the logging.

model/aop/GlobalEventHandler.cfc

```
1  /**
2  * Application wide event handler for ORM operations
3  */
4  component implements="cfide.orm.IEventHandler"
5  {
```

```
6 public void function preLoad( any entity )
7 {
8     logEvent( "preload", arguments.entity );
9 }
10
11 public void function postLoad( any entity )
12 {
13     logEvent( "postload", arguments.entity );
14 }
15
16 public void function preInsert( any entity )
17 {
18     logEvent( "preinsert", arguments.entity );
19     var timestamp = now();
20     if ( StructKeyExists( arguments.entity, "setCreated" ) )
21     {
22         arguments.entity.setcreated( timestamp );
23     }
24     if ( StructKeyExists( arguments.entity, "setUpdated" ) )
25     {
26         arguments.entity.setupdated( timestamp );
27     }
28 }
29
30 public void function postInsert( any entity )
31 {
32     logEvent( "postinsert", arguments.entity );
33 }
34
35 public void function preUpdate( any entity, Struct oldData )
36 {
37     if ( StructKeyExists( arguments.entity, "setUpdated" ) )
38     {
39         arguments.entity.setupdated( Now() );
40     }
41     logEvent( "preupdate", arguments.entity );
42 }
43
44 public void function postUpdate( any entity )
45 {
46     logEvent( "postupdate", arguments.entity );
47 }
48
49 public void function preDelete( any entity )
50 {
```

```
51     logEvent( "predelete", arguments.entity );
52 }
53
54 public void function postDelete( any entity )
55 {
56     logEvent( "postdelete", arguments.entity );
57 }
58
59 private void function logEvent( required string event, required CFC )
→ {
60     var message = "Entity: #getMetaData( cfc ).fullname#, Event: #event#";
61     WriteLog( type="event", file="ormevent.log", text=message );
→ }
62 }
63
64
```

Now, whenever we load or persist any entity in your application to the database, you'll have a log of the events. But wait, we've still got `preInsert()` and `preUpdate()` methods defined in the *Post.cfc* – what is going to happen? Well, if a method exists both in a persistent CFC and in the global event handler, then they are both fired. The method in the persistent CFC is called first, followed by the corresponding method in the global event handler.

You may have noticed that the `preUpdate()` method has two arguments. The second argument is a struct of the data in the state it was when the entity was loaded from the database. This struct of "old" data can be really useful if you wanted to build versioning into your application so you could roll back to a previous version. I didn't show the "olddata" argument in the `preUpdate()` method in *Post.cfc* for simplicity, but the argument is still passed in and can be used.

Chapter 11 : Improving Performance

Reading and writing to databases is a common cause of performance bottlenecks in applications, particularly as the number of records grows over time. With ColdFusion ORM these problems can still exist, so it's worth being aware of techniques that can remedy these issues.

Paging

Maxresults

When you are working with large recordsets, you will typically want to break them up into chunks and employ paging to display them. Doing this with ColdFusion ORM is relatively simple.

You can't use database proprietary syntax to limit the number of results returned; instead you should use the `maxresults` argument. Here are some examples:

```
1 <cfscript>
2 result = ORMExecuteQuery( "from Author", false, { maxresults=2 } );
3
4 result = EntityLoad( "Author", {}, "", { maxresults=2 } );
5 </cfscript>
6
7 <cfset ormoptions = { maxresults=2 }>
8
9 <cfquery name="result" dbtype="hql" ormoptions="#ormoptions#">
10     from Author
11 </cfquery>
```

All of the above will return the same result. Hibernate will automatically use the correct syntax for the database engine you're using. If you were using MySQL, which uses the `LIMIT` keyword, the generated SQL would be:

```
1 Hibernate:
2     select author0_.*
3     from AUTHORS author0_ limit ?
```

Offset

The second part of using paging is to be able to choose the offset (or startrow). Imagine you have an application where you are displaying 5 records at a time. On the second 'page', you'll

want to retrieve the next 5 records starting at the sixth record. To do this you would set the offset value to 5, telling Hibernate to skip the first 5 records.

```
1 <cfscript>
2 result = ORMExecuteQuery( "from Author", false, { maxresults=5, offset=5 }
→ );
3
4 result = EntityLoad( "Author", {}, "", { maxresults=5, offset=5 } );
→ </cfscript>
5
6 <cfset ormoptions = { maxresults=5, offset=5 }>
7
8 <cfquery name="result" dbtype="hql" ormoptions="#ormoptions#">
9     from Author
10 </cfquery>
11
```

Lazy Loading

We've seen in previous chapters that when dealing with relationships between entities Hibernate will often run more than one SQL statement to get the data. For optimal performance you may want to reduce the number of database requests, or delay loading the data into memory until it's required. ColdFusion ORM gives you fine grained control over how Hibernate constructs its SQL queries.

Hibernate is set by default to use lazy loading (which is the same as adding a `lazy` attribute with the value of `true`). This means that it will only load the data for the entity requested and not for any of the associated entities. If any of the related entities are required, then Hibernate will need to run another SQL statement. This may sound inefficient, but Hibernate doesn't know what data you are going to need, so instead of retrieving lots of data from the database and creating potentially large object graphs in memory, Hibernate will only load from the database the data that it needs when it needs it. So in the example of `Post.getAuthor().getName()`, Hibernate will run a query against the `posts` database table and then run another query against the `authors` database table.

In the simple one-to-many relationship we created between an `Author` entity and the `Post` entity, we will often use code similar to the following:

```
1 <cfdump var="----- pre load -----" output="console">
2
3 <cfset Author = EntityLoadByPK( "Author", 5 )>
4
```

```
5 <cfdump var="----- pre haspost -----" output="console">
6
7 <cfif Author.hasPost(>
8
9     <cfdump var="----- pre getpost -----" output="console">
10
11     <cfoutput>Post count: #ArrayLen( Author.getPosts() )#<br></cfoutput>
12
13     <cfdump var="----- pre loop getpost -----" output="console">
14
15     <cfloop array="#Author.getPosts()#" index="Post">
16         <cfoutput>#Post.getID()#<br></cfoutput>
17     </cfloop>
18
19 </cfif>
```

I've added in some `cfdump` tags to show at which points the SQL is being executed. If I run this code, Hibernate will execute the following three SQL statements:

```
1 ----- pre load -----
2 Hibernate:
3     select author0_.*
4     from AUTHORS author0_
5     where author0_.author_pk=?
6 ----- pre haspost -----
7 Hibernate:
8     select count(post_pk)
9     from POSTS
10    where fk_authorid =?
11 ----- pre getpost -----
12 Hibernate:
13     select posts0_.*
14     from POSTS posts0_
15    where posts0_.fk_authorid=?
16     order by posts0_.datepublished desc
17 ----- pre loop getpost -----
```

To improve performance, I could create a new computed property that uses the `formula` attribute to get a count of the associated posts as I demonstrated in the '*Using Formulas*' section of Chapter 9. Alternatively, I can change this behaviour by adding a `lazy` attribute with a value of `false` to the relationship in *Author.cfc*:

model/Author.cfc

```
1  /**
2  * I model an Author
3  */
4  component persistent="true" table="authors"
5  {
6      property name="id" fieldtype="id" setter="false"
7          column="author_pk" generator="native";
8
9      ...
10
11     // one author can have many Posts
12     property name="posts" fieldtype="one-to-many" cfc="Post"
13         fkcolumn="fk_authorid" type="array" singularname="post"
14         orderby="datepublished desc" inverse="true"
15         lazy="false";
16
17     ...
18
19 }
```

This time when we run the same script Hibernate runs two SQL statements:

```
1  ----- pre load -----
2  Hibernate:
3      select author0_.*
4      from AUTHORS author0_
5      where author0_.author_pk=?
6  Hibernate:
7      select posts0_.*
8      from POSTS posts0_
9      where posts0_.fk_authorid=?
10     order by posts0_.datepublished desc
11 ----- pre haspost -----
12 ----- pre getpost -----
13 ----- pre loop getpost -----
```

The other behaviour change that setting the `lazy` attribute to `false` had was on when the second SQL statement ran. It ran directly after the `EntityLoadByPK()` and before the `hasPost()` method was called. In fact, if the script only had the `EntityLoadByPK()` in it, Hibernate would still run the same SQL statements.

All of the above applies to one-to-one, one-to-many, many-to-many and many-to-one relationships. If you are working with one-to-many or many-to-many relationships, then the `lazy` attribute also accepts a third value of `'extra'`. Extra lazy is similar to setting the relationship up with the `lazy` attribute set to `'true'` (which is the default behaviour). The difference is that when

you try to load the associated posts, Hibernate will only load the primary key of each post. Running the script again with the relationship set up as extra lazy produces the following SQL statements:

```
1  ----- pre load -----
2  Hibernate:
3      select author0_.*
4      from AUTHORS author0_
5      where author0_.author_pk=?
6  ----- pre haspost -----
7  Hibernate:
8      select count(post_pk)
9      from POSTS
10     where fk_authorid =?
11 ----- pre getpost -----
12 Hibernate:
13     select count(post_pk)
14     from POSTS
15     where fk_authorid =?
16 ----- pre loop getpost -----
17 Hibernate:
18     select posts0_.*
19     from POSTS posts0_
20     where posts0_.fk_authorid=?
21     order by posts0_.datepublished desc
```

The purpose of extra lazy is to reduce the amount of data that needs to be loaded into memory until it is required. In my example, you can see that Hibernate was clever enough to know that it didn't need to retrieve all the data from the posts table until I started to loop through the array.

If you try to use extra lazy loading with one-to-one or many-to-one relationships, then you'll get an error similar to:

```
Cannot convert lazy value extra specified for property Author in cfc model.Post
to a boolean.
```

The n+1 Selects Problem

When you are dealing with one-to-many and many-to-many relationships then you can get into the dreaded n+1 selects problem. So what is the 'n+1 selects problem'?

Consider the following code, which loads all the Authors and iterates through them displaying the first name and number of posts for each Author.

```
1 <cfscript>
2 authors = EntityLoad( "Author" );
3 </cfscript>
4
5 <cfoutput>
6 <ul>
7 <cfloop array="#authors#" index="Author">
8     <li>#Author.getFirstname()# - #ArrayLen( Author.getPosts() )#</li>
→ </cfloop>
9 </ul>
10 </cfoutput>
11
```

Code like this is commonly used in applications; but it creates a performance issue, since what Hibernate does is to run a separate SQL statement for each Author to get the number of posts. So, you are running one statement to get the Authors and then one SQL statement per Author. Put another way, you have one select to get the Authors. If you have 20 authors, then you'll have to run 20 (one for each author) additional select statements to get the number of posts. The number of required SQL statements is expressed as $n+1$.

It is also worth looking at what happens when we approach this relationship from the other side. This time we are going to load all the posts and then iterate through them displaying the author.

```
1 <cfscript>
2 posts = EntityLoad( "Post" );
3 </cfscript>
4
5 <cfoutput>
6 <ul>
7 <cfloop array="#posts#" index="Post">
8     <li>#Post.getTitle()# by #Post.getAuthor().getFirstname()#</li>
9 </cfloop>
10 </ul>
11 </cfoutput>
```

When this code runs, an SQL statement will be run to get the Post, then another SQL statement will be executed for each Author. Hibernate is clever enough to only run a new SQL statement if the Author data hasn't been loaded already.

In the example above, if the same Author wrote all of those posts, only two SQL statements will be run. However, if you had 50 posts, each written by a different Author, you would be running 51 SQL statements – which is far from ideal.

There are several strategies that you can use to solve this problem.

Fetch="join"

You can set up many-to-one relationships with an additional `fetch` attribute. If you don't specify the `fetch` attribute then ColdFusion defaults to `fetch="select"`. The default works well for most situations as data is only loaded when we need it. In the example of iterating through authors to get the number of posts, this is not a good strategy. By switching the `fetch` strategy to `join` the number of trips to the database server can be reduced.

Model/Post.cfc

```
1  /**
2  * I model a blog post
3  */
4  component persistent="true"
5  {
6      property name="id" fieldtype="id" setter="false"
7          column="post_pk" generator="native";
8
9      property name="title";
10     property name="content";
11
12     property name="Author" fieldtype="many-to-one" cfc="Author"
13         fkcolumn="fk_authorid"
14         fetch="join";
15
16     ...
17 }
```

Now when we run the script to list the Authors, the SQL that is executed is:

```
1  Hibernate:
2      select
3          this_.post_pk as post1_121_1_,
4          this_.title as title121_1_,
5          this_.content as content121_1_,
6          this_.ispublished as ispublis4_121_1_,
7          this_.datepublished as datepubl5_121_1_,
8          this_.created as created121_1_,
9          this_.updated as updated121_1_,
10         this_.fk_authorid as fk8_121_1_,
11         author2_.author_pk as author1_119_0_,
12         author2_.firstname as firstname119_0_,
13         author2_.lastname as lastname119_0_
14     from
```

```
15         POSTS this_  
16     left outer join  
17         AUTHORS author2_  
18         on this_.fk_authorid=author2_.author_pk
```

This is much more efficient than running multiple SQL statements, as we've told Hibernate that we want to load the Author at the same time as we load the posts. Of course, the downside is that if we now load any Post persistent entity, Hibernate will also retrieve and load into memory the associated Author regardless of whether we need that data. Using the fetch attribute is really an all or nothing approach. It is most commonly used for one-to-one relationships; for other types of relationship you'll want to look at how your data is used before you decide on the best strategy. Because of this you should consider using HQL to give you more flexibility.

Fetch with HQL

By using a fetch join in HQL, you can initialize all the associated entities along with the parent, all via an HQL statement. The code shown below allows you to load all the posts and authors in one go. Note the inclusion of the `fetch` keyword in the HQL statement.

```
1  <cfscript>  
2  posts = ORMExecuteQuery( "select p from Post p join fetch p.Author a" );  
→  </cfscript>  
3  
4  <cfoutput>  
5  <ul>  
6  <cfloop array="#posts#" index="Post">  
7      <li>#Post.getAuthor().getFirstname()#</li>  
8  </cfloop>  
9  </ul>  
10 </cfoutput>  
11
```

Using an HQL statement like this will override any `lazy` or `fetch` attributes you have defined in *Post.cfc*. Running the code above will result in the following SQL statement being run:

```
1  Hibernate:  
2      select  
3          post0_.post_pk as post1_342_0_,  
4          author1_.author_pk as author1_340_1_,  
5          post0_.title as title342_0_,  
6          post0_.content as content342_0_,  
7          post0_.ispublished as ispublis4_342_0_,  
8          post0_.datepublished as datepubl5_342_0_,
```

```
9         post0_.created as created342_0_,
10        post0_.updated as updated342_0_,
11        post0_.fk_authorid as fk8_342_0_,
12        author1_.firstname as firstname340_1_,
13        author1_.lastname as lastname340_1_
14    from
15        POSTS post0_
16    inner join
17        AUTHORS author1_
18        on post0_.fk_authorid=author1_.author_pk
```

Using HQL is more work for you to do as a developer, but means that you can leave the relationship as lazy and then use a `fetch join` when you know that you will need to load associated entities.

Batch Fetching

Batch fetching is a useful strategy when dealing with large recordsets, as it tells Hibernate to retrieve a number of records in one go. If we revisit the example I used before of loading posts and iterating through them calling the `getAuthor()` method, we saw that when the relationship between `Post` and `Author` is lazy then Hibernate will execute one SQL statement for each new `Author`. If we add the `batch` attribute to the `Author.cfc` then we can tell Hibernate that we want to load 'n' Authors at once.

```
1  /**
2   * I model an Author
3   */
4   component persistent="true" table="authors" batchsize="5"
5   {
6       ...
7   }
```

With the `batchsize` attribute set to 5, Hibernate will now load 5 authors at a time from the database. So when we run the following code:

```
1  <cfscript>
2  posts = EntityLoad( "Post" );
3  </cfscript>
4
5  <cfoutput>
6  <ul>
7  <cfloop array="#posts#" index="Post">
8      <li>#Post.getTitle()# by #Post.getAuthor().getFirstname()#</li>
```

```
9      </cfloop>
10    </ul>
11  </cfoutput>
```

Hibernate will execute the following SQL statements:

```
1  Hibernate:
2      select this_.*
3      from POSTS this_
4  Hibernate:
5      select author0_.*
6      from AUTHORS author0_
7      where author0_.author_pk in (?, ?, ?, ?, ?)
8  HIBERNATE DEBUG - binding '3' to parameter: 1
9  HIBERNATE DEBUG - binding '2' to parameter: 2
10 HIBERNATE DEBUG - binding '4' to parameter: 3
11 HIBERNATE DEBUG - binding '5' to parameter: 4
12 HIBERNATE DEBUG - binding '6' to parameter: 5
13 Hibernate:
14      select author0_.*
15      from AUTHORS author0_
16      where author0_.author_pk in (?)
17 HIBERNATE DEBUG - binding '1' to parameter: 1
```

As I set the `batchsize` attribute to 5, Hibernate needed to run a third SQL statement to retrieve the sixth Author from the database. If I'd set the `batchsize` attribute higher, then Hibernate wouldn't have needed the additional SQL statement. The value you should use for the `batchsize` depends on your application; there are examples on the Hibernate forums of people using a value of 100 without adverse effects.

Setting the `batchsize` attribute works for many-to-one and many-to-many relationships. For one-to-many relationships, you should set the `batchsize` attribute on the property instead. If we were loading all the Authors and iterating through them calling the `getPosts()` method, we could optimise this by adding the `batchsize` attribute to the `posts` property in `Author.cfc`.

```
1  /**
2   * I model an Author
3   */
4  component persistent="true" table="authors"
5  {
6      ...
7      property name="posts" fieldtype="one-to-many" cfc="Post"
8          fkcolumn="fk_authorid" singularname="post"
9          orderby="datepublished desc" inverse="true" cascade="all"
10         batchsize="100";
```

```
11     ...
12 }
```

When we call the `getPosts()` method of the `Author` object, Hibernate will now load the associated `Authors` in batches as demonstrated in the SQL logged below:

```
1  Hibernate:
2      select author0_.*
3      from AUTHORS author0_
4  Hibernate:
5      select posts0_.*
6      from POSTS posts0_
7      where posts0_.fk_authorid in (?, ?, ?, ?, ?, ?)
8      order by posts0_.datepublished desc
```

ORM Settings

When you first run an ORM-enabled application, ColdFusion does quite a bit of work behind the scenes for you. Whilst the performance is perfectly good it doesn't hurt to improve start up times where possible. By using the `ormsettings` in *Application.cfc* you should be able to reduce the configuration time.

`this.ormsettings.cfclocation`

I always use this setting in my application and urge you to do the same. If you don't use the `cfclocation` configuration key, then ColdFusion will search all the subdirectories of your application looking for CFCs that are persistent. If you have lots of directories and CFCs in your application, this can add a significant delay to your application's startup time.

The `cfclocation` configuration key accepts either a string value or an array. Some examples of the usage are:

```
1  /**
2   * I am the Application.cfc
3   */
4  component
5  {
6      this.datasource = "mydsn";
7      this.ormenabled = true;
8      // search for persistent cfc's in the model directory and sub directories
9      this.ormenabled.cfclocation = "model";
10 }
```

Using an array of locations:

```
1  /**
2  * I am the Application.cfc
3  */
4  component
5  {
6      this.datasource = "mysdn";
7      this.ormenabled = true;
8      // search the following locations
9      this.ormenabled.cfclocation = [ "blog/model/", "common/users/" ];
10 }
```

Using CFCs outside the webroot with a mapping:

```
1  /**
2  * I am the Application.cfc
3  */
4  component
5  {
6      this.datasource = "mysdn";
7      this.ormenabled = true;
8      this.mappings[ "/model" ] = ExpandPath( "../model/" );
9      // search the following locations
10     this.ormenabled.cfclocation = this.mappings[ "/model" ];
11 }
```

Some users have reported issues with `cfclocation` not working with relative paths. As a result I always use a mapping, even though my CFCs are in sub folders of the application, like so:

```
1  /**
2  * I am the Application.cfc
3  */
4  component
5  {
6      this.datasource = "mysdn";
7      this.ormenabled = true;
8      this.mappings[ "/model" ] = getDirectoryFromPath(
→  getCurrentTemplatePath() ) & "model";
9      // search the following locations
10     this.ormenabled.cfclocation = this.mappings[ "/model" ];
11 }
```

this.ormsettings.dialect

ColdFusion will attempt to detect the database server you are using automatically so it can use the correct dialect. It is a good idea to specify this setting even if performance is not of utmost importance, especially if you are using MySQL.

Example:

```
1  /**
2  * I am the Application.cfc
3  */
4  component
5  {
6      this.datasource = "mydsn";
7      this.ormenabled = true;
8      this.ormsettings.dialect = "MySQLwithInnoDB";
9  }
```

The supported values are:

- DB2
- DB2AS400
- DB2OS390
- Derby
- PostgreSQL
- MySQL
- MySQLwithInnoDB
- MySQLwithMyISAM
- Oracle8i
- Oracle9i
- Oracle10g
- Sybase
- SybaseAnywhere
- MicrosoftSQLServer

- Informix

this.ormsettings.useDBForMapping

By default this is enabled, which means that ColdFusion will inspect the database to get the datatypes of columns and also the primary and foreign keys of each table. If you set the value to false, you will need to provide all of this information – which Hibernate requires in order to function – in your CFCs. Turning this setting off will improve your application startup time, particularly if you have lots of database tables and columns.

this.ormsettings.logSQL

This may be fairly obvious, but please make sure that you either delete or disable this configuration key in production environments. The same goes for any erroneous `ORMReload()` calls you may have left in your *Application.cfc*.

Chapter 12 : Caching

First-Level Cache

Hibernate has a certain level of caching built-in by default, which happens automatically within the current ORM session. The following block of code will help explain this.

```
1 <cfscript>
2 A = EntityLoadByPK( "Author", 1 );
3 B = EntityLoadByPK( "Author", 1 );
4 </cfscript>
```

If you ran the code above, the first `EntityLoadByPK()` would cause an SQL statement to be run to get the data from the database. When the second `EntityLoadByPK()` occurs, Hibernate knows it has already loaded the Author entity from the database and will not execute a second SQL statement. Instead, it uses the Author data that is in the first-level cache.

Second-Level Cache

ColdFusion has had the ability to cache queries for a long time by simply adding the `cachedwithin` attribute to `cfquery`. ColdFusion ORM has similar in-built caching capabilities that you can utilise in your applications – this is called the Secondary Level Cache. By default the Secondary Level Cache is disabled, so you will need to enable it in your *Application.cfc*.

Application.cfc

```
1 /**
2  * I am the Application.cfc for my application
3  */
4 component
5 {
6     // set the unique name of the application
7     this.name = "orm_application";
8     // set the datasource
9     this.datasource = "orm_application";
10    // turn ORM on
11    this.ormenabled = true;
12    this.ormSettings.secondaryCacheEnabled = true;
13 }
```

Now that the Secondary Level Cache is available to you, you need to tell ColdFusion which entities you want to be able to cache. To do this you need to add the `cacheuse` attribute to your component metadata.

Post.cfc

```
1  /**
2  * I model a blog post
3  */
4  component persistent="true" table="posts" cacheuse="transactional"
5  {
6      property name="id" fieldtype="id" setter="false"
7          column="post_pk" generator="native";
8
9      ...
10 }
```

By adding the `cacheuse` attribute to your component metadata, when you next request a `Post` entity from the database using `EntityLoad()` or `EntityLoadByPK()`, ColdFusion will first look for it in your secondary level cache before it queries the database.

There are four possible values for the `cacheuse` attribute (I've used 'transactional' in my example). The available types are:

The 'read-only' Concurrency Strategy

You should use `read-only` only if the data is never updated. It is the fastest of all the options but, as the name suggests, is only useful for static data. If you try to update a persistent entity that has `cacheuse` attribute set to `read-only` then you'll get an error similar to:

An exception occurred when committing the transaction. The root cause of this exception was: `java.lang.UnsupportedOperationException: Can't write to a readonly object`. The specific sequence of files included or processed is:
C:\inetpub\wwwroot\bookblog\playground\update-post.cfm, line: 4

The 'nonstrict-read-write' Concurrency Strategy

The second fastest type is the `nonstrict-read-write` option. This strategy does not employ transaction isolation at a database level, so doesn't guarantee that the cache and the database are in sync. If your data is rarely updated and it is crucial to have the best possible performance from your application, you may want to consider this option.

The 'read-write' Concurrency Strategy

The read-write type is similar to the nonstrict-read-write option above, but uses read isolation at a database level. Because of this it doesn't perform quite as well as the previous two types, but does guarantee that the cache matches the database. It is not suitable for clustered environments.

The 'transactional' Concurrency Strategy

The transactional option is the safest type. Although this has the worst performance of the available options, it still performs really well. It also differs from all the other strategies because it updates the cache when the data is updated. All the other types will just evict the entity from the second level cache; the cache is only re-populated when the next read occurs. As such this strategy will only run an SQL update statement and not require a subsequent SQL select query on the next request for the entity.

When in doubt I tend to go for the transactional option as it still has good performance with added safety for your data integrity. I do encourage you to try out the different options under load in your application and choose the most suitable for your needs.

There is an optional cachename attribute that you can also use to specify a name for the key where the data is stored in the cache; by default the cachename is the name of the entity.

What's Cached?

When you use `EntityLoad()` or `EntityLoadByPK()` to retrieve a specific entity, if you have added the `cacheuse` attribute to the CFC ColdFusion will look in the cache first; if it doesn't exist, then ColdFusion will get the data from the database and store it in the cache.

The following code examples will all use the cached version, if available, before hitting the database.

```
1 a = EntityLoadByPK( "Post", 1 );
2 b = EntityLoad( "Post", 1, true );
3 c = EntityLoad( "Post", { id=1 }, true );
4 d = EntityLoad( "Post", { id=1 } ); // returns an array
```

ColdFusion will only cache the actual entity that you are loading; it does not cache associated entities. In the example above, the Post entity with an id of 1 will be cached, but the associated entities such as Author, Category etc are not in the cache. If you need to retrieve the Author of the Post, Hibernate will need to get it from the database (assuming that the Author has not been previously cached). This may seem strange at first; but if you consider that otherwise Hibernate could potentially end up loading up your entire database into the cache, you'll appreciate that this is a sensible approach!

However, you need to be aware that if you're using an HQL statement the cache is not used. This is not surprising as you could be using custom HQL, but it does mean we have to do some extra work if we want to cache results from HQL.

If you prefer to use `ORMExecuteQuery()` for your HQL, then the optional `queryoptions` argument should be used.

```
1 e = ORMExecuteQuery(  
2     " from Post where id = :id ",  
3     { id=1 },  
4     { cacheable=true, cachename='post', timeout=60 }  
5 );
```

If you're using `cfquery` to run your HQL then you can use the optional `ormoptions` argument to control caching.

```
1 <cfset options = { cacheable=true, cachename='post', timeout=60 }>  
2 <cfquery name="q" dbtype="hql" ormoptions="#options#">  
3     from Post where id = <cfqueryparam value="1">  
4 </cfquery>
```

By using `cfquery` or `ORMExecuteQuery()`, you get a much finer level of control over how the caching is employed.

This code will return an array with one element containing the Post with an id of 1. As you can see, I'm passing a structure into the `queryoptions` argument. The `queryoptions` argument accepts the following keys and values:

- `ignorecase` - boolean
- `maxResults` - integer
- `cacheable` - boolean
- `cachename` - string
- `timeout` - integer

Of these `cacheable`, `cachename` and `timeout` apply to caching. The only one that you need to use is `cacheable` with a value of `true`.

Caching Associated Entities

There are times when you will want to cache the associated entities for one-to-many or many-to-many relationships. This can be achieved adding the `cacheuse` attribute to a property that defines a relationship. For example, if we wanted to cache the associated posts of an Author entity we could use the following code:

model/Author.cfc

```
1 component persistent="true" table="authors"
2 {
3     ...
4     property name="posts" fieldtype="one-to-many" cfc="Post"
5         fkcolumn="fk_authorid" singularname="post"
6         orderby="datepublished desc"
7         inverse="true" cascade="all"
8         cacheuse="nonstrict-read-write";
9     ...
10 }
```

Managing the cache

The caching strategies we've already looked at are an excellent way to improve the performance of your applications without you needing to manage the cache directly. However, there may be times when an external process alters the database and you will want to be able to remove items from the cache. ColdFusion ORM gives you full control of the cache: unlike cached queries, you can choose to evict items from the cache by class, name or even id.

It is worth noting that a side effect of calling `ORMReload()` in your application is that it will evict all items from the cache.

ORMEvictEntity

`ORMEvictEntity()` allows you to evict (remove) all items from the cache of a certain entity. For example if you wanted to evict all cached Author entities you would use:

```
1 <cfscript>
2 ORMEvictEntity( "Author" );
3 </cfscript>
```

If you have only updated a specific Author entity, then this is a brute force approach to managing the cache. Instead you can evict a specific entity from the cache by id:

```
1 <cfscript>
2 ORMEvictEntity( "Author", 1 );
```

```
3 </cfscript>
```

If the item existed in the cache, it will be evicted; if it isn't, then ColdFusion will continue execution of the script without throwing an error.

The `ORMEvictEntity()` function will only evict entities that have been loaded using the `EntityLoadByPK()` or `EntityLoad()` functions.

ORMEvictCollection

When you use `ORMEvictEntity()`, only the instance data of the named entity is cleared from the cache. If you have set up caching on a one-to-many or many-to-many relationship, any associated entities can still remain in the cache. In the example of an `Author` entity that has cached posts, when you call `ORMEvictEntity()` on the `Author` entity, only the `Author` is evicted from the cache and not the associated `Posts`. This is where you would use `ORMEvictCollection()`.

```
1 <cfscript>
2 // evict all Author entities from the cache
3 ORMEvictEntity( "Author" );
4 // evict all associated Post entities from the cache
5 ORMEvictCollection( "Author", "posts" );
6 </cfscript>
```

If you were just evicting a single `Author` from the cache, you can choose to only evict the associated posts for the specific `Author`:

```
1 <cfscript>
2 // evict the Author from the cache
3 ORMEvictEntity( "Author", 1 );
4 // evict associated Post entities from the cache
5 ORMEvictCollection( "Author", "posts", 1 );
6 </cfscript>
```

ORMEvictQueries

If you use HQL in your application and have chosen to cache the result, you will need to use the `ORMEvictQueries()` function to evict it from the cache. The usage is simple; if you had used the following code to execute and cache the result from the HQL statement:

```
1 <cfscript>
2 result = ORMExecuteQuery(
3     " from Post where id = :id ",
```

```
4      { id=1 },
5      { cacheable=true, cachename="mycachename", timeout=60 }
6    );
7  </cfscript>
```

You would then use the value specified as the cachename to evict it from the cache:

```
1  <cfscript>
2    ORMEvictQueries( "mycachename" );
3  </cfscript>
```

You can also use `ORMEvictQueries()` to evict all cached HQL queries by not specifying a cache name:

```
1  <cfscript>
2    // evict all cached HQL results
3    ORMEvictQueries();
4  </cfscript>
```

Bulk Operations

If you find that you are deleting or updating objects inside a loop, you may want to consider using batch operations. As we've already seen, when you update or delete an existing entity you first have to load it into the session (or first level) cache. If you do this with lots of objects you will quickly use up memory. By using HQL you can do bulk updates or deletes, much as you would with SQL.

The following will update Authors:

```
1  <cfset foo = ORMExecuteQuery( "update Author set active = false" )>
```

The following will delete Authors:

```
1  <cfset foo = ORMExecuteQuery( "delete from Author where active = false" )>
→
```

Hibernate also supports bulk inserts, but you will need to dig into Hibernate's configuration files to do it successfully.

Of course, there is no reason why you can't fall back to using SQL, particularly for the bulk inserts. With inserts you don't have the added complication of evicting updated/deleted entities when using the secondary level cache.

Choosing the Cache Provider

ColdFusion ORM caching uses Ehcache under the hood by default. However, if you would prefer to use a different caching provider then you can use the `cacheprovider` setting to specify a different one.

Application.cfc

```
1  /**
2  * I am the Application.cfc for my application
3  */
4  component
5  {
6      // set the unique name of the application
7      this.name = "orm_application";
8      // set the datasource
9      this.datasource = "orm_application";
10     // turn ORM on
11     this.ormenabled = true;
12     this.ormSettings.secondaryCacheEnabled = true;
13     this.ormSettings.cacheprovider = "JBossCache";
14 }
```

It is worth noting that all the caching strategies (transactional, read-write, nonrestrict-read-write and read-only) are supported by Ehcache, but may not be by other caching providers.

Chapter 13 : Validation

ColdFusion 9 builds on the simple datatype enforcement of previous versions of ColdFusion. The `cfproperty` tag now accepts two additional attributes: `validate` and `validateparams`.

The `validate` and `validateparams` attributes are used when you call the generated setter for the property. If you are familiar with using the `IsValid()` in your applications, you'll be able to translate this knowledge to the new capabilities, although the `validate` attribute does not accept all of the validation types of `IsValid()`.

Rather than list all the supported types, I have chosen to demonstrate the usage by applying validation to the *Author.cfc*.

```
1  /**
2   * I model an Author
3   */
4   component persistent="true" table="authors"
5   {
6       // the id
7       property name="id" fieldtype="id" setter="false"
8           column="author_pk" generator="native";
9
10      property name="firstname"
11          ormtype="string" length="50"
12          validate="string" validateparams="{minLength=2, maxLength=50}";
13
14      property name="lastname"
15          ormtype="string" length="50"
16          validate="string" validateparams="{minLength=2, maxLength=50}";
17
18      property name="email"
19          ormtype="string" length="150"
20          validate="email";
21
22      ...
23
24  }
```

I've added `validate` and `validateparams` attributes to the `firstname`, `lastname` and `email` properties. When you try to pass an invalid value to the `setFirstname()`, `setLastname()` or `setEmail()` methods, ColdFusion will throw an error so you will need to wrap a try and catch around calling code.

The code below is a simple form for adding new Authors, which submits to itself. I've opted to store any error messages in an array, but you can implement error reporting any way that you wish.

```
1  <cfscript>
2  author = EntityNew( "Author" );
3  errors = [];
4
5  if ( StructCount( form ) )
6  {
7      try
8      {
9          author.setFirstname( form.firstname );
10     }
11     catch( any e )
12     {
13         ArrayAppend( errors, "Firstname: " & e.message );
14     }
15     try
16     {
17         author.setLastName( form.lastname );
18     }
19     catch( any e )
20     {
21         ArrayAppend( errors, "LastName: " & e.message );
22     }
23
24     if ( !ArrayLen( errors ) )
25     {
26         // no errors so save
27         transaction
28         {
29             EntitySave( author );
30         }
31     }
32 }
33 </cfscript>
34
35 <!--- start the view --->
36 <cfoutput>
37
38 <cfif ArrayLen( errors )>
39 <ul>
40 <cfloop array="#errors#" index="ndx">
41     <li>#ndx#</li>
```

```
42 </cfloop>
43 </ul>
44 </cfif>
45
46 <form action="addauthor.cfm" method="post">
47
48     <label for="firstname">First Name</label>
49     <input type="text" name="firstname" id="firstname"
50         value="#HtmlEditFormat( Author.getFirstName() )#">
51
52     <label for="lastname">Last Name</label>
53     <input type="text" name="lastname" id="lastname"
54         value="#HtmlEditFormat( Author.getLastName() )#">
55
56     <input type="submit" name="submit" id="submit"
57         value="add author">
58
59 </form>
60
61 </cfoutput>
```

I personally don't use the inbuilt validation capabilities as it is quite limited: I prefer to use validation frameworks. There are two free-to-use validation frameworks for ColdFusion:

- ValidateThis
<http://www.validatethis.org/>
- HyRule
<http://hyrule.riaforge.org/>

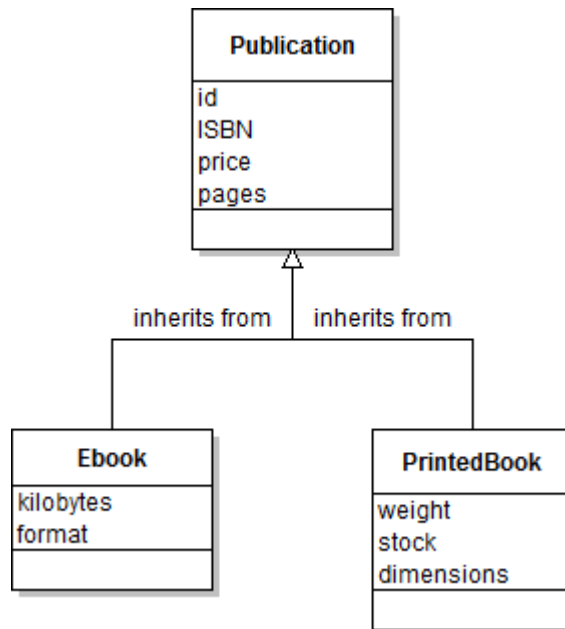
Chapter 14 : Advanced Mapping

Mapping Inheritance

Inheritance is a common (although often over-used) feature of writing Object Oriented code. ColdFusion ORM offers you three ways to store data in the database when using inheritance in your application.

I don't want to go too far off topic, but if you're not familiar with how inheritance works, then it can be used when you want to have several 'child' CFCs that share certain properties and functions. What you do is to create a 'parent' CFC that the 'child' CFC extends (inherits from). The parent CFC has all the common properties and functions that you want the child CFCs to have. The relationship between the parent and child is often referred to as an 'IS-A' relationship. For example, you could say that a Cat is-a more specialised version of an Animal. You could represent this in code by having a parent *Animal.cfc*, which *Cat.cfc* inherits from. You can then add a *Dog.cfc* that also extends *Animal.cfc*.

As I've used a blog as basis for the code samples in this book, I'm going to give the Authors a book deal to publish a collection of their blog posts. The book will be available as a printed paperback and also as an ebook. As the ebook and printed paperback have a lot of common properties, this sounds like a good case for inheritance. Below is a simplified UML diagram of the relationship:



The diagram shows that the 'parent' Publication has properties that are common to both Ebook and PrintedBook. PrintedBook and Ebook then have their own specific properties.

One Table Per Persistent Entity

As the title suggests, this approach means that each persistent Entity will have a corresponding table in the database. This means, for the example shown in the UML diagram above, that the database will need to have three database tables corresponding to the three CFCs.

To set up your application to use one table per persistent CFC, you will need to create the 'parent' *Publication.cfc* using the following code:

Publication.cfc

```
1  /**
2  * I model a Publication
3  */
4  component persistent="true" table="publications"
5  {
6      property name="id" fieldtype="id" setter="false"
7          column="publication_pk" generator="native";
8
9      property name="ISBN" ormtype="string" length="13";
10     property name="price" ormtype="big_decimal" scale="2";
```

```
11     property name="pages" ormtype="int";
12 }
```

The code used for *Publication.cfc* is the same as we've seen before. You don't need to do anything different to create the parent persistent entity.

Now that we have the parent entity, we can create the two child CFCs – *Ebook.cfc* and *PrintedBook.cfc* – both of which need to extend *Publication.cfc*.

EBook.cfc

```
1  /**
2  * I model an Ebook - inherits from Publication.cfc
3  */
4  component persistent="true" table="ebooks"
5      extends="Publication" joincolumn="publication_pk"
6  {
7      property name="kilobytes" ormtype="int";
8      property name="format" ormtype="string" length="20";
9  }
```

As we are using one table per persistent entity, then although the *Publication* entity has a table attribute, we need to have a table attribute to specify the table name for this entity. The extends attribute of the component references the parent CFC from which this CFC will inherit.

Finally, the joincolumn attribute is used to reference the column name of the primary key of the *Publication.cfc*.

You should note that the *EBook.cfc* does not have an identifier property. This is because the Ebook entity shares the same identifier as the *Publication* entity.

The *PrintedBook* entity will use similar code to the *Ebook* entity; you just add the appropriate properties for the subclass.

PrintedBook.cfc

```
1  /**
2  * I model a Printed Book - inherits from Publication.cfc
3  */
4  component persistent="true" table="printedbooks"
5      extends="Publication" joincolumn="publication_pk"
6  {
7      property name="weight" ormtype="int";
8      property name="stock" ormtype="int";
9      property name="dimensions" ormtype="string" length="30";
10 }
```

CRUD operations with persistent entities that extend another persistent entity are similar to how I demonstrated in the *ORM is CRUD* chapter. However, there are some subtle differences.

```
1  <cfscript>
2  // Create a new PrintedBook
3  transaction
4  {
5      PrintedBook = EntityNew( "PrintedBook" );
6
7      // inherited properties
8      PrintedBook.setTitle( "The Blog Post Collection" );
9      PrintedBook.setISBN( "1234567890120" );
10     PrintedBook.setPrice( 19.99 );
11     PrintedBook.setPages( 128 );
12
13     // PrintedBook properties
14     PrintedBook.setWeight( 500 );
15     PrintedBook.setStock( 20 );
16     PrintedBook.setDimensions( "w:17cm h:20cm d:1cm" );
17
18     EntitySave( PrintedBook );
19 }
20 // Create a new Ebook
21 transaction
22 {
23     Ebook = EntityNew( "Ebook" );
24
25     // inherited properties
26     Ebook.setTitle( "The Blog Post Collection" );
27     Ebook.setISBN( "1234567890121" );
28     Ebook.setPrice( 9.99 );
29     Ebook.setPages( 128 );
30
31     // Ebook properties
32     Ebook.setKilobytes( 1084 );
33     Ebook.setFormat( "EPUB" );
34
35     EntitySave( Ebook );
36 }
37 </cfscript>
```

As you can see from the code example above, when you create a new child entity you also have access to the getters and setters of the parent entity, but the calling code you write does not need to refer to the parent entity at all.

To list all the Ebook entities that are persisted in the database, you can use either `EntityLoad()` or `HQL`. You can also choose to list all Publications (all EBooks and PrintedBooks) by loading the parent Publication entity.

```
1 <cfscript>
2 publications = EntityLoad( "Publication" );
3
4 writeDump( publications );
5 </cfscript>
```

The output from this script is:

array								
1	component <u>model.PrintedBook</u> extends <u>model.Publication</u>							
	PROPERTIES	<table><tr><td>weight</td><td>500</td></tr><tr><td>stock</td><td>20</td></tr><tr><td>dimensions</td><td>w:17 cm h:20cm d:1cm</td></tr></table>	weight	500	stock	20	dimensions	w:17 cm h:20cm d:1cm
weight	500							
stock	20							
dimensions	w:17 cm h:20cm d:1cm							
	METHODS							
2	component <u>model.EBook</u> extends <u>model.Publication</u>							
	PROPERTIES	<table><tr><td>kilobytes</td><td>1084</td></tr><tr><td>format</td><td>EPUB</td></tr></table>	kilobytes	1084	format	EPUB		
kilobytes	1084							
format	EPUB							
	METHODS							

The first thing that you probably notice about this is that you can't see any of the inherited properties. This is just a limitation of `cfDump`: the data and properties are there.

At first glance the one table per persistent entity seems like an excellent approach, but in reality it does have its drawbacks.

When Hibernate retrieves data from the database, the generated SQL statements obviously need to use joins as we have linking three database tables. They need to be outer joins because of the way that the data is being stored: this is apparent when you see how the data is stored in the database. Shown below is the data and structure of the tables in the database:

Publications Database Table

PUBLICATIONS - query					
	ISBN	PAGES	PRICE	PUBLICATION_PK	TITLE
1	1234567890120	128	19.99	3	The Blog Post Collection
2	1234567890121	128	9.99	4	The Blog Post Collection

Printedbooks Database Table

PRINTEDBOOKS - query				
	DIMENSIONS	PUBLICATION_PK	STOCK	WEIGHT
1	w:17cm h:20cm d:1cm	3	20	500

Ebooks Database Table

EBOOKS - query			
	FORMAT	KILOBYTES	PUBLICATION_PK
1	EPUB	1084	4

The publications database table has one record per subclass, whereas the printedbooks and ebooks database tables will only have a record if the saved entity is of the corresponding type.

While the data stored in the database is properly normalised, if you look at the SQL that Hibernate generates you can see that, as well as the left outer joins, a case statement is also required to determine the entity type to map to:

```

1  Hibernate:
2      select
3          publicatio0_.*,
4          publicatio0_1_.*
5          publicatio0_2_.*
6      case
7          when publicatio0_1_.publication_pk is not null then 1
8          when publicatio0_2_.publication_pk is not null then 2
9          when publicatio0_.publication_pk is not null then 0
10     end as clazz_
11  from
12      PUBLICATIONS publicatio0_
13  left outer join
14      EBOOK publicatio0_1_
15      on publicatio0_.publication_pk=publicatio0_1_.publication_pk

```

```
16 left outer join
17     PRINTEDBOOK publicatio0_2_
18     on publicatio0_.publication_pk=publicatio0_2_.publication_pk
```

Using joins and case statements in SQL has a negative effect on performance. Eliminating the need for the case statement is easy to achieve by using a discriminator column.

One Table Per Persistent CFC With Discriminator

You can use a discriminator column to store the entity type. Setting this up is simple.

```
1  /**
2   * I model a Publication
3   */
4  component persistent="true" table="publications"
5     discriminatorcolumn="type"
6  {
7     property name="id" fieldtype="id" setter="false"
8         column="publication_pk" generator="native";
9     ...
10 }
```

In the example above I've added a `discriminatorcolumn` attribute, the value of which I've set to 'type'. The value you enter for this attribute is the name of the database column which you wish to use to store the entity type. I've chosen to use a column called 'type', but you can use a different naming convention if you prefer.

In each of the subclasses, you need to add the `discriminatorvalue` attribute. The value you enter for this attribute is what is stored in the database. You can use any value, but I think it makes sense to use the entity name. If the size of your database tables is of concern, you may wish to use a single character to save a bit of space.

EBook.cfc

```
1  /**
2   * I model an Ebook - inherits from Publication.cfc
3   */
4  component persistent="true" table="ebooks"
5     extends="Publication" joincolumn="publication_pk"
6     discriminatorvalue="EBOOK"
7  {
8     ...
9  }
```

PrintedBook.cfc

```

1  /**
2  * I model an Printed Book - inherits from Publication.cfc
3  */
4  component persistent="true" table="printedbooks"
5      extends="Publication" joincolumn="publication_pk"
6      discriminatorvalue="PRINTEDBOOK"
7  {
8      ...
9  }

```

If we examine the database now it looks like this:

Publications Database Table

PUBLICATIONS - query						
	ISBN	PAGES	PRICE	PUBLICATION_PK	TITLE	TYPE
1	1234567890120	128	19.99	3	The Blog Post Collection	PRINTEDBOOK
2	1234567890121	128	9.99	4	The Blog Post Collection	EBOOK

Printedbooks Database Table

PRINTEDBOOKS - query				
	DIMENSIONS	PUBLICATION_PK	STOCK	WEIGHT
1	w:17cm h:20cm d:1cm	3	20	500

Ebooks Database Table

EBOOKS - query			
	FORMAT	KILOBYTES	PUBLICATION_PK
1	EPUB	1084	4

Hibernate no longer needs a case statement to determine entity type:

```

1  Hibernate:
2      select
3          publicatio0_.*,
4          publicatio0_1_.*,
5          publicatio0_2_.*,
6      from
7          PUBLICATIONS publicatio0_
8      left outer join

```

```
9      EBOOK publicatio0_1_  
10      on publicatio0_.publication_pk=publicatio0_1_.publication_pk  
11  left outer join  
12      PRINTEDBOOK publicatio0_2_  
13      on publicatio0_.publication_pk=publicatio0_2_.publication_pk
```

The relationship between the entities is correctly normalised in the database but, because of this, we need to have left outer joins to link the tables. SQL statements that use multiple tables generally perform slower than SQL statements on single database tables, so for increased performance we should consider using one table to store the data for all Ebook and PrintedBook entities.

One Table per Hierarchy

ColdFusion ORM allows you to map an entire hierarchy to a single database table. This approach performs faster than the previous two and simplifies the database schema, but does mean you are de-normalising some of the data.

To use this strategy, the parent Publication entity is the same as for the other types of inheritance mapping:

Publication.cfc

```
1  /**  
2  * I model a Publication  
3  */  
4  component persistent="true" table="publications"  
5      discriminatorcolumn="type"  
6  {  
7      property name="id" fieldtype="id" setter="false"  
8          column="publication_pk" generator="native";  
9      ...  
10 }
```

The subclasses need to reference the table name of the parent entity. In the example we're using, this needs to be the publications database table. As we are only using a single table, we no longer require the joincolumn attribute that we used for the other inheritance mapping strategies.

EBook.cfc

```
1  /**  
2  * I model an Ebook - inherits from Publication.cfc  
3  */  
4  component persistent="true" table="publications"
```

```

5     extends="Publication" discriminatorvalue="EBOOK"
6     {
7         ...
8     }

```

PrintedBook.cfc

```

1  /**
2  * I model an Printed Book - inherits from Publication.cfc
3  */
4  component persistent="true" table="publications"
5      extends="Publication" discriminatorvalue="PRINTEDBOOK"
6  {
7      ...
8  }

```

The database schema used to store the relationships is now:

PUBLICATIONS - query										
	DIMENSIONS	FORMAT	ISBN	KILOBYTES	PAGES	PRICE	PUBLICATION_PK	STOCK	TITLE	WEIGHT
1	w:17 cm h:20cm d:1cm	[empty string]	1234567890120	[empty string]	128	19.99	1	20	Greatest Posts	PRINTEDBOOK
2	[empty string]	EPUB	1234567890121	1084	128	9.99	2	[empty string]	Greatest Posts	EBOOK

HQL and Inheritance

When you write HQL statements, you can use the special property `class` to filter by specific entity types. For example, imagine that we added a new subclass of `AudioBook`, but we had a need to get all publications that cost less than 10 pounds, excluding `AudioBooks`. We could use the following HQL:

```

1  <cfscript>
2  result = ORMExecuteQuery( "
3      from Publication as p
4      where p.class IN ( EBook, PrintedBook )
5      and price < 10
6      " );
7  </cfscript>

```

One thing to note is that you don't need to wrap quotes around the entity names. The HQL statement will produce the same result whether you use quotes or not. However, if you misspell or use the wrong case for the entity name, the HQL will not throw an error if you use quotes. If you omit the quotes, you'll get an error similar to:

Error while executing the Hibernate query.

```
com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: Unknown column
'printedbook' in 'where clause' The error occurred in
C:\inetpub\wwwroot\bookblog\playground\inheritance_hql.cfm: line 2
1 : <cfscript>
2 : result = ORMExecuteQuery( "
3 :   from Publication as p
4 :   where p.class IN ( EBook, printedbook )
```

I prefer to not use quotes, so that I do get errors. This is useful as it means the problem will be picked up during development, rather than having an application which does not error but produces the wrong results.

Using Collections Instead of Arrays

When we looked at the many-to-many relationship previously between Post and Category, we were dealing with arrays. This works well in most circumstances, but ColdFusion also allows you to return a struct of entities instead of an array.

To demonstrate this, I'm going to introduce a Permission entity that will have a many-to-many relationship with the Author entity.

model/Permission.cfc

```
1  /**
2   * I model a permission
3   */
4   component persistent="true" table="permissions"
5   {
6       // identifier
7       property name="id" fieldtype="id" generator="native"
8         column="permission_pk";
9
10      // properties
11      property name="title" length="6";
12      property name="description" length="100";
13
14      // relationships
15      property name="authors" fieldtype="many-to-many" cfc="Author"
16        singularname="author" linktable="authors_permissions"
17        type="array" inverse="true";
18  }
```

The many-to-many relationship with the Author entity is defined just as we did earlier, and will use an array.

Now that I have created the *Permission.cfc* we can add the many-to-many relationship to the Author entity. This time, instead of using an array, I'm going to use a collection (struct) instead.

model/Author.cfc

```
1  /**
2  * I model an Author
3  */
4  component persistent="true" table="authors"
5  {
6      // identifier
7      property name="id" fieldtype="id" setter="false"
8          column="author_pk" generator="native";
9
10     // properties
11     property name="firstname" length="50";
12     property name="lastname" length="50";
13
14     // one author can have many Posts
15     property name="posts" fieldtype="one-to-many" cfc="Post"
16         fkcolumn="fk_authorid" type="array" singularname="post"
17         orderby="datepublished desc"
18         inverse="true";
19
20     // define a many-to-many relationship with Permission
21     property name="permissions" fieldtype="many-to-many"
22         cfc="Permission" singularname="permission"
23         linktable="authors_permissions"
24         type="struct" structkeycolumn="title"
25         structkeytype="string";
26 }
```

The syntax I have used here is different to that which I used in the *Permission.cfc*. The relevant line of code is:

```
24     type="struct" structkeycolumn="title"
25     structkeytype="string";
```

Instead of the `type` attribute being an array, it's defined as a struct. There are also two new attributes, `structkeycolumn` and `structkeytype`. The first of these, `structkeycolumn`, is used to identify the column name to use in the `authors_permissions` database table to store the struct (collection) key name. The second attribute, `structkeytype`, dictates the data type for the key name.

If we dump out an instance of the modified Author, we'll see that we have the following methods:

The `addPermission()` method – used to add permissions to the Author entity. When you call the `addPermission()` method you will need to pass in the key name as well as the Permission entity.

function ADDPERMISSION

Arguments:

Name	Required	Type	Default
key	Required	Any	
permission	Required	Permission	

ReturnType: void

The `getPermissions()` method – used to return a struct of all associated permissions for the Author entity.

function GETPERMISSIONS

Arguments: none

ReturnType: struct

The `hasPermission()` method – used to check for a specific permission. Unlike with array-based many-to-many relationships, this method accepts a string (the key name) instead of a Permission entity.

function HASPERMISSION

Arguments:

Name	Required	Type	Default
key	Optional	Any	

ReturnType: boolean

The `removePermission()` method – used to remove a specific permission. Unlike with array-based many-to-many relationships, this method accepts a string (the key name) instead of a Permission entity.

function REMOVEPERMISSION

Arguments:

Name	Required	Type	Default
key	Required	Any	

ReturnType: boolean

The `setPermissions()` method – used to associate multiple Permission entities in one go. Unlike with array-based many-to-many relationships, this method accepts a struct of Permission entities.

function SETPERMISSIONS

Arguments:

Name	Required	Type	Default
permissions	Required	struct	

ReturnType: void

The method names are as we would expect, but the method signatures are different from the ones we used earlier for the array based many-to-many relationship between the Category and Post entities.

To see the usage of the new methods, let's give an author some permissions:

```
1 <cfscript>
2 transaction
3 {
4     // create a new Author
5     Author = EntityNew( "Author" );
6     Author.setFirstname( "John" );
7     Author.setLastname( "Whish" );
8
9     // create a 'Create' Permission
10    Create = EntityNew( "Permission" );
11    Create.setTitle( "Create" );
12    Create.setDescription( "can create posts" );
13
14    // create a 'Delete' Permission
15    Delete = EntityNew( "Permission" );
16    Delete.setTitle( "Delete" );
17    Delete.setDescription( "can delete posts" );
18
```

```
19 // add the permissions to the Author
20 Author.addPermission( Create.getTitle(), Create );
21 Author.addPermission( Delete.getTitle(), Delete );
22
23 // add the Author to the Permission
24 Create.addAuthor( Author );
25 Delete.addAuthor( Author );
26
27 // make entities persistent
28 EntitySave( Author );
29 EntitySave( Create );
30 EntitySave( Delete );
31 }
32 </cfscript>
```

This is all pretty much as we would expect. I've created an Author and two new Permission entities; namely 'create' and 'delete'. However, when I assign the permissions to the author entity, I am using the following code:

```
19 // add the permissions to the Author
20 Author.addPermission( Create.getTitle(), Create );
21 Author.addPermission( Delete.getTitle(), Delete );
```

With an array I'd only need to pass in the Permission entity. As I am using a collection based association, I need to pass it two arguments; the key name and the entity respectively. The key name is the value of the property referenced when defining the relationship; which is why I'm passing in `Create.getTitle()` as the first argument.

I can test to see how ColdFusion is storing the relationship with a simple script:

```
1 <cfscript>
2 Author = EntityLoadByPK( "Author", 5 );
3 writeDump( var=Author.getPermissions(), top=2 );
4 </cfscript>
```

Running the code above produces the following output:

struct			
Create	component <u>model.Permission</u>		
	PROPERTIES	id	11
		title	Create
		description	can create posts
		authors	array
1		component <u>model.Author</u>	
	METHODS		
Delete	component <u>model.Permission</u>		
	PROPERTIES	id	12
		title	Delete
		description	can delete posts
		authors	array
1		[see cfc2 for Author details]	
	METHODS		

Using a collection instead of an array has a few advantages. If I were writing a real application, I'd want to be able to check to see if the Author has a required permission. As the `getPermissions()` method of the Author entity returns a struct, I can use ColdFusion's struct functions to check if a specific key exists, rather than having to pass in an entity to the `hasPermission()` method. There are several ways to do this:

```

1  <cfscript>
2  Author = EntityLoadByPK( "Author", 5 );
3  </cfscript>
4
5  <cfoutput>
6  Author Permissions (by title):<br />
7  #StructKeyList( Author.getPermissions() )#<hr />
8  Author has 'Create' Permission:<br />
9  #Author.hasPermission( "Create" )#<hr />
10 Struct key 'Create' exists:<br />
11 #StructKeyExists( Author.getPermissions(), "Create" )#<hr />
12 </cfoutput>

```

Running this script will produce:

```
Author Permissions (by title):
Create,Delete
-----
Author has 'Create' Permission:
YES
-----
Struct key 'Create' exists:
YES
-----
```

A word of caution: once again, you need to be careful of the case you use. If I change my script to use different cases for the key names and run it again, you'll see that the checks do not match:

```
Author Permissions (by title):
Create,Delete
-----
Author has 'Create' Permission:
YES
-----
Author has 'CREATE' Permission:
NO
-----
Author has 'create' Permission:
NO
-----
Struct key 'Create' exists:
YES
-----
Struct key 'CREATE' exists:
NO
-----
Struct key 'create' exists:
NO
-----
```

Because of this you must make sure that you always use the correct case when working with Hibernate collections.

Getting an Entity from a Collection

If you wanted to retrieve a specific Permission, you could of course use the key name to access it:

```
1 <cfscript>
2 Author = EntityLoadByPK( "Author", 5 );
3
4 // get the 'Create' permission
5 WriteDump( var=Author.getPermissions()[ "Create" ], top=1 );
6 // alternative syntax
7 WriteDump( var=Author.getPermissions().get( "Create" ), top=1 );
8 </cfscript>
```

Both of these techniques will dump out the Permission entity.

The Inverse Attribute

As my relationship is using a collection on one side and an array on the other, ColdFusion will throw an error when persisting the relationship if you don't set the inverse attribute on the side using an array. In the Author many-to-many Permission relationship I have added the inverse attribute to the *Permission.cfc*:

```
1 /**
2  * I model a permission
3  */
4 component persistent="true" table="permissions"
5 {
6     ...
7
8     // relationships
9     property name="authors" fieldtype="many-to-many" cfc="Author"
10         singularname="author" linktable="authors_permissions"
11         type="array" inverse="true";
12 }
```

As the keyname must be the same value as the title property, you may want to consider overwriting the `addPermission()` method of *Author.cfc* to do the work for you:

Model/Author.cfc

```
1  /**
2  * I model an Author
3  */
4  component persistent="true" table="authors"
5  {
6      ...
7
8      /**
9      *I override the generated addPermission method
10     */
11     void function addPermission( required Permission )
12     {
13         permissions[ arguments.Permission.getTitle() ] = arguments.Permission;
14     }
15 }
```

Chapter 15 : Schema Generation Revisited

DataTypes and Precision

The ormtype attribute

As you've seen in the earlier code example, we can use the `ormtype` attribute to control the datatypes used for creating the database columns. The `ormtype` attribute accepts the following values:

- `big_decimal`
- `binary`
- `blob`
- `boolean`
- `char`
- `character`
- `clob`
- `date`
- `double`
- `float`
- `int`
- `integer`
- `long`
- `serializable`
- `short`
- `string`

- text
- timestamp
- true_false
- yes_no

You may notice that these potential values don't necessarily correspond to the available data types your database server supports. This is because, for the most part, Hibernate is database agnostic and so generic names are used for the datatype values. If these datatypes are not specific enough for your particular need and database engine, you can use the `sqltype` attribute instead to specify an exact or database-specific datatype.

Specifying Length, Nullable and more...

ColdFusion supports several other property attributes to allow you to have more control over the database schema generation. These are:

- column
- dbdefault
- length
- notnull
- precision
- scale
- sqltype
- unique
- uniquekey

The column attribute

The `column` attribute allows you to specify a name for the column if it is different from the property name. By default, the column is the same as the property name. I use this attribute a lot as I like my columns in the database to be prefixed per table. This allows me to see easily which table a column belongs to in queries, and also to do some neat things by querying the schema information stored in the database. Of course, it's entirely up to you how you prefer to work.

Below is a simple example where I'm using a naming convention for primary keys where I prefix the column name with 'pk_' followed by the table name.

Post.cfc

```
1  /**
2  * I model a blog post
3  */
4  component persistent="true"
5  {
6      property name="id" fieldtype="id" setter="false"
7          generator="native" column="pk_post";
8      ...
9  }
```

I should point out that while ColdFusion will generate tables and columns for you, it is important that you review your database schema and then refine and tune it before you use it in production.

There are several other attributes you can use to control the generated database schema. These are:

Dbdefault attribute

The `dbdefault` attribute allows you to specify a default value to use in the database if no value is set. The value is evaluated as a native expression by the database server you are using, so you can't use a ColdFusion function such as `Now()`.

If you want to use a fixed value, you will need to quote it. For example:

```
1  property name="title" dbdefault="'untitled'";
```

Length attribute

A numeric value that allows you to specify the number of characters for string based fields. If you don't specify the number of characters then the database field will be 255 characters long.

Example:

```
1  property name="title" length="100";
```

NotNull attribute

Accepts a boolean value which specifies if the database field can have null values. By default this is false, meaning the field will accept null values. Example:

```
1  property name="title" notnull="true";
```

Precision attribute

Accepts an integer value that specifies the number of digits in a number (including digits after the decimal point).

Scale attribute

Accepts an integer value that specifies the number of digits to the right of the decimal point.

Sqltype attribute

Allows you to have fine grained control of database specific data types. For example you may wish to use a MONEY datatype.

Unique attribute

Accepts a boolean value that specifies whether the database field should only accept unique values.

Uniquekey attribute

Allows you to specify a name for the unique constraint.

Naming Strategy

It's quite common to use different naming conventions for your entities and the database schema those entities map to. So far in this book I've used singular names for the entities and plural names for database tables. Your preference may be to prefix table names, or to use underscores for database columns instead of camel-case – whereas you wouldn't want underscores for property names.

In-Built Naming Strategies

ColdFusion comes with an in-built naming strategy that converts camel-case names to upper case names with underscores. To use this strategy, all you need to do is add another setting to the *ormsettings* structure in *Application.cfc*:

Application.cfc

```
1  /**
2   * I am the Application.cfc for my application
3   */
4  component
```

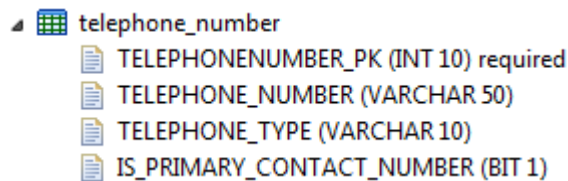
```
5 {
6     // set the unique name of the application
7     this.name = "orm_application";
8     // set the datasource
9     this.datasource = "orm_application";
10    // turn ORM on
11    this.ormenabled = true;
12    this.ormsettings.namingstrategy = "smart";
13    ...
14 }
```

If I create a new TelephoneNumber entity (which doesn't have the table attribute):

model/TelephoneNumber.cfc

```
1 /**
2  * I model a Telephone number
3  */
4 component persistent="true"
5 {
6     // identifier
7     property name="TelephoneNumberID" fieldtype="id"
8         column="telephonenumber_pk" generator="native";
9
10    // properties
11    property name="TelephoneNumber" ormtype="string" length="50";
12    property name="TelephoneType" ormtype="string" length="10";
13    property name="IsPrimaryContactNumber" ormtype="boolean";
14 }
```

With the smart naming strategy enabled, the TelephoneNumber entity shown above will automatically create a corresponding table in the database of:



The screenshot shows a database table named 'telephone_number' with the following columns:

- TELEPHONENUMBER_PK (INT 10) required
- TELEPHONE_NUMBER (VARCHAR 50)
- TELEPHONE_TYPE (VARCHAR 10)
- IS_PRIMARY_CONTACT_NUMBER (BIT 1)

For the table name, ColdFusion has used the camelCase of the entity name to work out where to insert an underscore when naming the database table.

As I specified the column name I wanted to use for the identifier, ColdFusion just converted it to uppercase for the corresponding database column. For the other properties, as with the table

name, the camelCase used in the property name has been translated to underscores for the column names.

In case you are wondering why the column names are uppercase and the table name is lowercase, this is because I'm using MySQL which doesn't use uppercase table names.

Using Your Own Naming Strategy

Let's take a scenario in which your company has a naming policy where database table names are lowercase, plural and prefixed with 'tbl_'. Also, column names need to be lowercase and prefixed with 'col_'.

Instead of having to add the table attribute and/or column attributes to each CFC, you could choose to create your own naming strategy instead.

Setting up a naming strategy is straightforward. First, you will need to create a CFC that contains the logic for mapping the entity and property names to the database table and column names.

model/aop/NamingStrategy.cfc

```
1  component implements="CFIDE.orm.INamingStrategy"
2  {
3      /**
4       * transform entity name to table name. This method
5       * is called for each entity as ColdFusion attempts to
6       * map it to a database table. The argument value is the
7       * entity name.
8       */
9      public string function getTableName( string tableName )
10     {
11         return "tbl_" & LCase( arguments.tableName ) & "s";
12     }
13
14     /**
15     * transform property name to column name. This method
16     * is called for each entity property as ColdFusion attempts to
17     * map it to a database column. The argument value is the
18     * property name.
19     */
20     public string function getColumnName( string columnName )
21     {
22         return "col_" & LCase( arguments.columnName );
23     }
24 }
```

25 }

The *NamingStrategy.cfc* above contains two methods: `getTableName()` and `getColumnName()`. Both of these methods need to be present, as the *NamingStrategy.cfc* implements the `CFIDE.orm.INamingStrategy` interface. I won't describe the usage of these methods as the code is self-explanatory. Needless to say, an Entity named 'Author' will map to the database table 'tbl_authors'. A property named 'firstname' will map to the column name 'col_firstname'.

Being able to define your own naming strategy is a powerful feature and could save you a lot of typing.

Chapter 16 : Tips, Tricks & Gotchas

This chapter is really just a place for me to mention anything I think may be useful that doesn't have a natural home in any of the previous chapters.

Tips

Populate Using the EntityNew Constructor

In all the examples in this book I've been using the generated setters to populate new entities. However, you can do it all in one call, by using the second argument of the EntityNew() function.

```
1 <cfscript>
2 // get and populate a new Author
3 Author = EntityNew( "Author", { firstname="John", lastname="Whish" } );
→ EntitySave( Author );
4 </cfscript>
5
```

The second argument of EntityNew() accepts a struct. The keys in the struct you pass in need to match the property names. This technique can save you a lot of typing and, if you name your form fields to match property names, you can populate directly from the form struct. For example:

```
1 <cfscript>
2 // get and populate a new Author from the form scope
3 Author = EntityNew( "Author", form );
4 EntitySave( Author );
5 </cfscript>
```

HQL IN Clause

When you are writing HQL statements with ORMExecuteQuery(), you may want to pass a list of values to an IN clause. You can do this by using named parameters and an array:

```
1 <cfscript>
2 result = ORMExecuteQuery(
3     "from Author where id IN ( :idlist )",
4     { idlist=[1,2,3] }
5 )
```

```
5 );  
6 </cfscript>
```

Refactoring to ORM with EntityToQuery

If you have a legacy code base and want to start using ColdFusion ORM in your application, then the `EntityToQuery()` function can be very useful. `EntityToQuery()` is used to convert an array of persistent entities into a ColdFusion Query object. This means that you can use ORM to get data from the database, but still use views, tags and functions that accept a query, such as `cfspreadsheet` and `cfchart`.

Legacy display pages

Legacy applications will probably have lots of pages that loop through a ColdFusion Query and display the results. Instead of having to update all your views to loop through arrays of objects, you can use `EntityToQuery()` to convert the array of entities into a ColdFusion Query object.

```
1 <cfscript>  
2 // get array of Author entities  
3 authors = EntityLoad( "Author" );  
4 // convert to a query  
5 qAuthors = EntityToQuery( authors );  
6 </cfscript>  
7  
8 <!--- existing legacy code which uses a query --->  
9 <ul>  
10 <cfoutput query="qAuthors">  
11 <li>#qAuthors.firstname# #qAuthors.lastname#</li>  
12 </cfoutput>  
13 </ul>
```

Query of Queries with EntityToQuery

Another use of `EntityToQuery()` is that when you convert your array of persistent entities into a ColdFusion Query Object, you can then run Query of Queries on the result. Query of Queries are not very efficient, but it is useful to know that you can do this, although it only works in ColdFusion 9.0.1.

Detecting Changes

Sometimes you want to know if you have anything in the Hibernate Session that has been modified, and will therefore be persisted back to the database. Hibernate has a useful

`isDirty()` method which returns true if modifications have been made to persistent entities in the current Hibernate session.

```
1 <cfscript>
2 Author = EntityLoadByPK( "Author", 1 );
3 writeDump( ORMGetSession().isDirty() );
4 Author.setFirstname( "Foo" );
5 writeDump( ORMGetSession().isDirty() );
6 </cfscript>
```

Running the code above will output 'NO YES' to the browser.

You could use this functionality to create a safety net before transactions close to prevent any accidental changes to the database.

Multiple Datasources

ColdFusion 9.0.1 introduced the ability to use multiple datasources with ORM. This can be useful if you have another database that holds data you need in your application. This is not unusual if you have multiple systems in your organisation that could store exchange rates, employee or company data.

To make a persistent CFC use an alternative datasource then you just need to add the datasource attribute to the component metadata.

```
1 /**
2  * I model an Employee
3  */
4 component persistent="true" table="tbl_employees"
5     datasource="legacyapp"
6 {
7     // identifier
8     property name="id" fieldtype="id" setter="false"
9         column="pk_employeeid" generator="native";
10
11     property name="employeeid" ormtype="string";
12 }
```

Non Persisted Properties

There may be times when you are working with persistent entities, but you have a property that you don't want to be saved to the database. A good example of this would be for a User entity that has a password property. When a new user signs up, you'll probably want to ask them to

confirm the password. One way to check that the two passwords match is to compare them in the User entity. The problem is that if we create a new passwordconfirm property, then the value of this will be persisted to the database.

To stop the passwordconfirm property being persisted in the database then you can add the optional persistent attribute with a value of false. The *User.cfc* may look something like:

User.cfc

```
1  /**
2  * I model a User
3  */
4  component persistent="true" table="users"
5  {
6      property name="id" fieldtype="id" setter="false"
7          generator="native" column="user_pk";
8
9      property name="username";
10     property name="password";
11     property name="passwordconfirm" persistent="false";
12
13     /**
14     * I perform a case sensitive comparison
15     */
16     boolean function passwordsmatch()
17     {
18         return compare( variables.password, variables.passwordconfirm ) == 0;
19     }
20 }
```

Entities with the same name

As your code base grows, you may find that you have a situation where you have two or more entities which have the same name. This presents a problem as entity names need to be unique in your application. Instead of having to rename one of them, you can use the entityname attribute instead.

If I had an application which already had a User entity and needed to add a second User entity, I could solve the naming collision as follows:

User.cfc

```
1  /**
2  * I model a User
```

```
3  */
4  component persistent="true" entityname="Customer"
5  {
6      property name="id" fieldtype="id" setter="false"
7          generator="native" column="user_pk";
8
9      ...
10 }
```

When you do this, you can use all the ORM functions for this entity by passing in the value specified in the `entityname` attribute.

Tricks

When I'm working on an application that uses ORM I often use the console for viewing the SQL statements that Hibernate is executing. However, when you look at the activity it can become hard to distinguish which SQL statements belong to which request. A really simple way to make it a bit easier to read is to use the `writeDump()` function in the `onRequestStart()` method of *Application.cfc*:

Application.cfc

```
1  /**
2   * I am the Application.cfc for my application
3   */
4  component
5  {
6      ...
7      boolean function onRequestStart( targetPage )
8      {
9          if ( CGI.SERVER_NAME == "localhost" )
10         {
11             writeDump( var="***** new request *****", output="console" );
12         }
13         ...
14     }
15 }
```

As the `writeDump()` is outputting to the console, you see it inline with the debugging output from Hibernate. I also occasionally put these `writeDump()` calls into my code while developing to see at which point Hibernate is executing the SQL statements – although you need to be careful not to leave them in when deploying to production.

Hibernate Statistics

In the *Debugging* Chapter, we used Hibernate Statistics to retrieve the HQL statements that were being executed. By using the in-built lifecycle of *Application.cfc* you can enable this for your entire application. This can be useful for ad-hoc debugging in production.

```
1  /**
2   * I am the Application.cfc for my application
3   */
4  component
5  {
6      ...
7      boolean function onRequestStart( targetPage )
8      {
9          var SessionFactory = ORMGetSessionFactory();
10         var request.stats = SessionFactory.getStatistics();
11         request.stats.setStatisticsEnabled( true );
12         ...
13     }
14
15     void function onRequestEnd()
16     {
17         if ( request.stats.getStatisticsEnabled() )
18         {
19             WriteDump( request.stats.getQueries() );
20         }
21     }
22 }
23 }
```

If you were to use this to help debug an application in production you'd probably want to restrict it to a specific IP address or similar, so that it wouldn't appear for all visitors.

The Hibernate Statistics engine can provide you with much more information than simply showing the HQL statements. The following are some examples of a few of the methods exposed by the Statistics API.

```
1  WriteOutput("isStatisticsEnabled: #hs.isStatisticsEnabled()#<br>");
2
3  WriteDump(var=hs.getEntityNames(), label="getEntityNames");
4
5  WriteDump(var=hs.getCollectionRoleNames(),
6  →  label="getCollectionRoleNames");
7
7  WriteOutput("getQueryExecutionCount: #hs.getQueryExecutionCount()#<br>");
```

```
→
8  WriteOutput("getQueryExecutionMaxTime:
9  #hs.getQueryExecutionMaxTime()#<br>");
→
10 WriteOutput("getEntityDeleteCount: #hs.getEntityDeleteCount()#<br>");
11
→ WriteOutput("getEntityFetchCount: #hs.getEntityFetchCount()#<br>");
12
13 WriteOutput("getEntityLoadCount: #hs.getEntityLoadCount()#<br>");
14
15 WriteOutput("getEntityInsertCount: #hs.getEntityInsertCount()#<br>");
16
17 WriteOutput("getEntityUpdateCount: #hs.getEntityUpdateCount()#<br>");
→
18 WriteOutput("getQueryCacheHitCount: #hs.getQueryCacheHitCount()#<br>");
19
→ WriteOutput("getQueryCacheMissCount: #hs.getQueryCacheMissCount()#<br>");
20
21 WriteOutput("getQueryCachePutCount: #hs.getQueryCachePutCount()#<br>");
→
22 WriteOutput("getSecondLevelCacheHitCount:
23 #hs.getSecondLevelCacheHitCount()#<br>");
→
24 WriteOutput("getSecondLevelCacheMissCount:
25 #hs.getSecondLevelCacheMissCount()#<br>");
→
26 WriteOutput("getSecondLevelCachePutCount:
27 #hs.getSecondLevelCachePutCount()#<br>");
→
28
29
→
30
31
→
```

I've included a screenshot below of what the output of the code above could be.

isStatisticsEnabled: YES

getEntityNames - array	
1	Category
2	Permission
3	Post
4	EBook
5	Author
6	Publication
7	Comment
8	Profile
9	PrintedBook

getCollectionRoleNames - array	
1	Post.comments
2	Author.posts
3	Author.permissions
4	Post.categories
5	Author.publishedposts

getQueryExecutionCount: 6
getQueryExecutionMaxTime: 230
getEntityDeleteCount: 0
getEntityFetchCount: 0
getEntityLoadCount: 32
getEntityInsertCount: 0
getEntityUpdateCount: 0
getQueryCacheHitCount: 0
getQueryCacheMissCount: 0
getQueryCachePutCount: 0
getSecondLevelCacheHitCount: 0
getSecondLevelCacheMissCount: 0
getSecondLevelCachePutCount: 32

If you really want to dig down into the statistics, you can even get data about entities using the `getEntityStatistics()` method and collections using the `getCollectionStatistics()` method.

Gotchas

Constructor Arguments

When you create your own `init()` method for persistent entities, you need to make sure that it either; does not expect any arguments, or, if you do use arguments that they are optional. If you don't then you'll get an error when using functions such as `EntityLoad()`.

Detached Entities

The detached entity issue is a common gotcha that developers new to ColdFusion ORM will encounter. It is usually caused by developers storing persistent entities in the ColdFusion session scope (not to be confused with Hibernate sessions).

You can store persistent entities in the ColdFusion session scope, but it does limit what you can do with them. Consider the following code:

```
1  <cfscript>
2  if ( IsNull( session.Author ) )
3  {
4      session.Author = EntityLoadByPk( "Author", 1 );
5      writeOutput( "loaded from database" );
6  }
7  writeDump( session.Author.getFirstname() );
8  </cfscript>
```

The first time that this code runs, it will get the Author from the database. The second time it runs (assuming the session hasn't expired), the code will still run and display the Author's first name.

The first symptom of detached entities occurs when you try to make changes to the Author.

```
1  <cfscript>
2  if ( !IsNull( session.Author ) )
3  {
4      session.Author.setFirstname( "Vicky" );
5  }
6  </cfscript>
```

When the code above runs, no errors will be thrown, but the database will not be updated. This is because Hibernate is no longer aware of this object, so won't track changes made to it. You may think that using `EntitySave()` would solve the problem, but it won't.

The second symptom of detached entities is that you can't call any of the Author's methods that use associated data that is lazy loaded. For example, this code will fail:

```
1 <cfscript>
2 posts = session.Author.getPosts();
3 </cfscript>
```

The code above will result in the error message:

```
org.hibernate.LazyInitializationException: failed to lazily initialize a
collection of role: Author.posts, no session or session was closed
```

This is because the posts are lazy loaded, so they are only retrieved from the database when they are requested. The reason that we could call the `getFirstname()` method was that the actual Author data had already been loaded from the database.

You can 're-attach' the detached entity the Hibernate session in a number of different ways. The first is to simply reload the entity from the database:

```
1 <cfscript>
2 Author = EntityLoadByPK( "Author", session.Author.getID() );
3 posts = Author.getPosts();
4 </cfscript>
```

Another technique is to use the `EntityLoadByExample()` function:

```
1 <cfscript>
2 Author = EntityLoadByExample( session.Author );
3 posts = Author.getPosts();
4 </cfscript>
```

The third method is to use the `EntityMerge()` function:

```
1 <cfscript>
2 Author = EntityMerge( session.Author );
3 posts = Author.getPosts();
4 </cfscript>
```

All of the above techniques will work: however, I'd recommend that you do not store persistent entities in the ColdFusion session scope. It is much better simply to store the id in the ColdFusion session scope and load the entity on each request. If you are using the secondary cache, then you don't need to worry about any extra database requests.

WriteDump causes ColdFusion to Crash

`writeDump()` and `cfDump` can cause a Java Heap exception when you try to dump an entity. This is because when you dump an entity, you are also dumping out all the associated entities as well. Potentially, you could actually end up trying to load your entire database into memory as ColdFusion attempts to dump it out! This happens regardless of whether you are using lazy loading or not. The easiest way to stop this happening is to always use the `top` attribute to restrict the depth that ColdFusion retrieves.

```
1 <cfscript>
2 Author = EntityLoadByPK( "Author", 1 );
3 // dump out using the top attribute
4 WriteDump( var=Author, top="2" );
</cfscript>
```

Cached MetaData

For performance reasons, ColdFusion caches metadata. This means that you may make changes to your code and then dump out the entity, but not see the changes. Your code will probably still run as expected, but it can be confusing when debugging. As a result it is best not to trust the methods shown when dumping out entities. The best way to clear the cache is to restart ColdFusion.

Generated Has Methods and Non-Persisted Entities

When working with the `hasXYZ()` method that is generated for certain relationships, you can get incorrect results. Consider the following code:

```
1 <cfscript>
2 Author = EntityNew( "Author" );
3
4 PostA = EntityNew( "Post" );
5 PostA.setTitle( "Post A" );
6
7 Author.addPost( PostA );
8
9 PostB = EntityNew( "Post" );
10 PostB.setTitle( "Post B" );
11
12 writeDump( Author.hasPost( PostA ) );
13 writeDump( Author.hasPost( PostB ) );
14 </cfscript>
```

When you run this code, the output you'll see in your browser is:

YES YES

This is clearly wrong, as we did not add PostB to the Author. The reason why this happens is that ColdFusion ORM uses the identifier of the entity to check if the relationship exists. As neither PostA nor PostB has been saved to the database, they both have a null value for the identifier. This problem does not happen after the entities have been persisted to the database as the identifier property is set to the primary key stored in the database.

You can negate this issue by handling the key generation yourself and using a GUID instead of getting the database to do it. Alternatively, if you want to use numeric or database generated identifiers, you can create a persistent property which is populated by a GUID and override the `hasXYZ()` method to use this property instead.

Appendix

Quick Mapping Reference

Relationships

one-to-many mapping

The minimal required attributes for a one-to-many mapping are: name, fieldtype and cfc:

```
1 property name="posts" fieldtype="one-to-many" cfc="Post";
```

I consider the following optional attributes to be worth adding to all one-to-many mappings: fkcolumn, singularname, inverse and orderby. For example:

```
1 property name="posts" fieldtype="one-to-many" cfc="Post"
2     fkcolumn="fk_authorid" singularname="Post" inverse="true"
3     orderby="title";
```

many-to-one mapping

The minimal required attributes for a many-to-one mapping are: name, fieldtype and cfc:

```
1 property name="Author" fieldtype="many-to-one" cfc="Author";
```

If you are mapping to an existing database or want control over foreign key names, you should add the fkcolumn attribute. An example is:

```
1 property name="Author" fieldtype="many-to-one" cfc="Author"
2     fkcolumn="fk_authorid";
```

many-to-many mapping

The minimal required attributes for a many-to-many mapping are: name, fieldtype and cfc:

```
1 property name="categories" fieldtype="many-to-many" cfc="Category";
```

If you are mapping to a legacy database or want to use your own naming conventions for tables and columns, you should add the fkcolumn, inversejoincolumn and linktable attributes. I think it is also worth adding the singularname and type attributes as well.

```
1 property name="categories" fieldtype="many-to-many" cfc="Category"
2     type="array" singularname="Category"
```

```
3 fkcolumn="fk_postid" inversejoincolumn="fk_categoryid"  
4 linktable="CategoryPosts";
```

Setting up an Apache Derby Database

To create a Derby database, you need to log into your ColdFusion administrator (CFIDE) and select “Data Sources” from the “Data & Services” menu. Type in the name of your datasource, select “Apache Derby Embedded” from the dropdown list and press the “Add” button.

Data & Services > Data Sources

Add and manage your data source connections and Data Source Names (DSNs). You use a DSN to connect ColdFusion to a variety of data sources.

Add New Data Source

Data Source Name	bookblog
Driver	Apache Derby Embedded
Add	

On the following screen you need to enter the path to where you want your database to be created and, of course, make sure you tick the “create database” option.

Data & Services > Datasources > Apache Derby Embedded

Apache Derby Embedded : bookblog_

CF Data Source Name	bookblog	
Database Folder	c:\Datasources\bookblog	Browse Server
Create Database	☒	
Description		
Show Advanced Settings Submit Cancel		

ColdFusion will create the folder for you, so don't need to create it yourself. If the folder already exists, you will get an error. Your database is now set up and ready to use.

References

Documentation

ColdFusion ORM Documentation

http://help.adobe.com/en_US/ColdFusion/9.0/Developing/WSD628ADC4-A5F7-4079-99E0-FD725BE9B4BD.html

Hibernate Documentation

<http://www.hibernate.org/docs>

Publications

Java Persistence with Hibernate

Second Edition of Hibernate in Action
Christian Bauer and Gavin King

Articles

An Introduction to ColdFusion ORM

<http://ria.dzone.com/articles/coldfusion-orm>

Hat Tips

There are many talented bloggers out there whose knowledge I have drawn upon to write this book. They are in alphabetical order.

Barney Boisvert

<http://www.barneyb.com>

Brian Rinaldi

<http://remotesynthesis.com>

Brian Kotek

<http://www.briankotek.com>

Bob Silverberg

<http://www.silverwareconsulting.com>

Dan Vega

<http://www.danvega.org>

Marc Esher

<http://blog.mxunit.org>

Mark Mandel

<http://www.compoundtheory.com>

Manju Kiran

<http://www.manjukiran.net>

Raymond Camden

<http://www.coldfusionjedi.com>

Rupesh Kumar

<http://www.rupeshk.org>

Sam Farmer

<http://samfarmer.com>

Where Next?

Hopefully after reading this book you'll feel that you have a good understanding of how to build ORM powered applications in ColdFusion. However, as I stated at the start Hibernate is a huge subject and there is a wealth of functionality that I haven't covered.

I highly recommend joining the CF-ORM Google group where you'll find a lot of clever, helpful developers who can assist you as well as a rich archive of previous questions and answers.

<http://groups.google.com/group/cf-orm-dev>

If you want to learn more about Hibernate, Object Oriented Programming or ColdFusion then you may be interested in the following publications.

Java Persistence with Hibernate

Second Edition of Hibernate in Action

Christian Bauer and Gavin King

November, 2006 | 880 pages

ISBN: 1-932394-88-5

<http://www.manning.com/bauer2/>

Object-Oriented Programming in ColdFusion

Matt Gifford

October, 2010 | 192 pages

ISBN: 1-847196-32-2

<https://www.packtpub.com/object-oriented-programming-in-coldfusion/>

Adobe ColdFusion 9 Web Application Construction Kit, Volume 1: Getting Started

ISBN: 0-321-66034-X

<http://www.forta.com/books/032166034X/>

Adobe ColdFusion 9 Web Application Construction Kit, Volume 2: Application Development

ISBN: 0-321-67919-9

<http://www.forta.com/books/0321679199/>

Adobe ColdFusion 9 Web Application Construction Kit, Volume 3: Advanced Application Development

ISBN: 0-321-69448-1

<http://www.forta.com/books/0321679202/>

ColdFusion 9 Developer Tutorial

John Farrar

ISBN: 978-1-849690-24-9

<https://www.packtpub.com/coldfusion-9-developer-tutorial/book>

And Finally...

Thank you for reading my book; I really hope that you find it useful. It took a lot of hard work and many months to put together, but all that effort would be wasted if people didn't read it!

If you would like to contact me regarding omissions, corrections or any other matters, then please do so via my website at:

<http://www.aliaspooryorik.com/>

Kind regards,

John Whish

ColdFusion 9 comes with the Hibernate Object Relational Mapper (ORM) library baked in. This means that you can quickly build applications without having to write lots of code to get the data in and out of the database.

This book is a guide aimed at anyone who wants to develop ORM enabled applications with ColdFusion.

Just reviewed @aliaspooryorik upcoming awesome book and you'll all want a copy when he releases it!

@bigmadkev

Kev McCabe - Adobe Community Professional