Working with Simple Associations using CakePHP

by Ahsanul Bari Anupom Syam | December 2008 | MySQL Open Source PHP

Database relationship is hard to maintain even for a mid-sized PHP/MySQL application, particularly, when multiple levels of relationships are involved because complicated SQL queries are needed. CakePHP offers a simple yet powerful feature called 'object relational mapping' or ORM to handle database relationships with ease.In CakePHP, relations between the database tables are defined through association—a way to represent the database table relationship inside CakePHP. Once the associations are defined in models according to the table relationships, we are ready to use its wonderful functionalities. Using CakePHP's ORM, we can save, retrieve, and delete related data into and from different database tables with simplicity, in a better way—no need to write complex SQL queries with multiple JOINs anymore!

In this article by Ahsanul Bari and Anupom Syam, we will have a deep look at various types of associations and their uses. In particular, the purpose of this article is to learn:

- How to figure out association types from database table relations
- How to define different types of associations in CakePHP models
- How to utilize the association for fetching related model data
- How to relate associated data while saving

There are basically 3 types of relationship that can take place between database tables:

- one-to-one
- one-to-many
- many-to-many

The first two of them are simple as they don't require any additional table to relate the tables in relationship. In this article, we will first see how to define associations in models for one-to-one and one-to-many relations. Then we will look at how to retrieve and delete related data from, and save data into, database tables using model associations for these simple associations.

Defining One-To-Many Relationship in Models

To see how to define a one-to-many relationship in models, we will think of a situation where we need to store information about some authors and their books and the relation between authors and books is one-to-many. This means an author can have multiple books but a book belongs to only one author (which is rather absurd, as in real life scenario a book can also have multiple authors). We are now going to define associations in models for this one-to-many relation, so that our models recognize their relations and can deal with them accordingly.

Time for Action: Defining One-To-Many Relation

1. Create a new database and put a fresh copy of CakePHP inside the web root. Name the database whatever you like but rename the cake folder to relationship. Configure the database in the new Cake installation.
2. Execute the following SQL statements in the database to create a table named authors,

```
CREATE TABLE `authors` (
`id` int( 11 ) NOT NULL AUTO_INCREMENT PRIMARY KEY ,
`name` varchar( 127 ) NOT NULL ,
`email` varchar( 127 ) NOT NULL ,
`website` varchar( 127 ) NOT NULL
);
```

3. Create a books table in our database by executing the following SQL commands:

```
CREATE TABLE `books` (
`id` int( 11 ) NOT NULL AUTO_INCREMENT PRIMARY KEY ,
`isbn` varchar( 13 ) NOT NULL ,
`title` varchar( 64 ) NOT NULL ,
`description` text NOT NULL ,
`author_id` int( 11 ) NOT NULL
)
```

4. Create the Author model using the following code (/app/models/authors.php):

```
<?php
class Author extends AppModel
{
var $name = 'Author';
var $hasMany = 'Book';
}
?>
```

5. Use the following code to create the Book model (/app/models/books.php):

```
<?php
class Book extends AppModel
{ var $name = 'Book';
var $belongsTo = 'Author';
}
?>
```

6. Create a controller for the Author model with the following code: (/app/controllers/authors_controller.php):

```
<?php
class AuthorsController extends AppController {
var $name = 'Authors';
var $scaffold;
}
?>
```

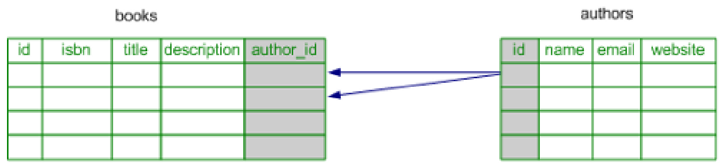
7. Use the following code to create a controller for the Book model (/app/controllers/books_controller.php):

```
<?php
class BooksController extends AppController {
var $name = 'Books';
var $scaffold;
}
?>
```

8. Now, go to the following URLs and add some test data: http://localhost/relationship/authors/ and http://localhost/relationship/books/

What Just Happened?

We have created two tables: authors and books for storing author and book information.



A foreign-key named author_id is added to the books table to establish the one-to-many relation between authors and books. Through this foreign-key, an author is related to multiple books, as well as, a book is related to one single author.

> By Cake convention, the name of a foreign-key should be underscored, singular name of target model, suffixed with _id.

Once the database tables are created and relations are established between them, we can define associations in models. In both of the model classes, Author and Book, we defined associations to represent the one-to-many relationship between the corresponding two tables. CakePHP provides two types of association: hasMany and belongsTo to define one-to-many relations in models.

These associations are very appropriately named:

- As an author 'has many' books, Author model should have hasMany association to represent its relation with the Book model.
- As a book 'belongs to' one author, Book model should have belongsTo association to denote its relation with the Author model.

In the Author model, an association attribute $hasMany is defined with the value Book to inform the model that every author can be related to many books. We also added a $belongsTo attribute in the Book model and set its value to Author to let the Book model know that every book is related to only one author.

After defining the associations, two controllers were created for both of these models with scaffolding to see how the associations are working.

Retrieving Related Model Data in One-To-Many Relation

Once the associations are in place, we can retrieve related data very easily without any trouble. In fact, we don't need to do anything at all—CakePHP's ORM will automatically fetch all the related model data using association. We will work on the previous code-base to see how the related model data can be retrieved.

Time for Action: Retrieving Related Model Data

1. Take out scaffolding from both of the controllers: AuthorsController (/app/controllers/authors_controller.php) **and** BooksController (/app/controllers/books_controller.php).

2. Modify the AuthorsController like the following:

```
<?php
class AuthorsController extends AppController {
var $name = 'Authors';
function index() {
$this->Author->recursive = 1;
$authors = $this->Author->find('all');
$this->set('authors', $authors);
}
}
?>
```

3. Create a view file for the /authors/index action (/app/views/authors/index.ctp):

```
<?php foreach($authors as $author): ?>
<h2><?php echo $author['Author']['name'] ?></h2>
<hr />
<h3>Book(s):</h3>
<ul>
<?php foreach($author['Book'] as $book): ?>
<li><?php echo $book['title'] ?></li>
<?php endforeach; ?>
</ul>
<?php endforeach; ?>
```

4. Write down the following code inside the BooksController:

```
<?php
class BooksController extends AppController {
var $name = 'Books';
function index() {
$this->Book->recursive = 1;
$books = $this->Book->find('all');
$this->set('books', $books);
}
}
?>
```

5. Create a view file for the action /books/index (/app/views/books/index.ctp):

```
<table>
<thead>
<th>ISBN</th><th>Title</th><th>Author</th>
</thead>
<?php foreach($books as $book): ?>
<tr>
<td><?php echo $book['Book']['isbn'] ?></td>
<td><?php echo $book['Book']['title'] ?></td>
<td><?php echo $book['Author']['name'] ?></td>
</tr>
<?php endforeach; ?>
</table>
```

6. Point your browser to the following links and see how related books and authors show up:
   http://localhost/relationship/authors/
   http://localhost/relationship/books/

What Just Happened?

The first line in the index() action of the AuthorsController sets the model attribute $recursive to 1.

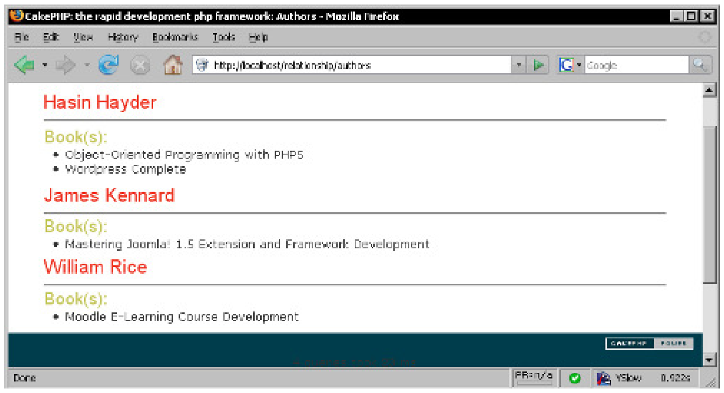
```
$this->Author->recursive = 1;
```

This special attribute is an integer that specifies the number of levels we want our model to fetch associated model data in the next find('all') or find('first') operations. In the AuthorsController, we set it to 1 for the Author model. This means the subsequent Author->find('all') operation will return all associated model data that are related directly to the Author model. The result of the function call Author->findAll() is then stored in a variable $authors. The returned result is an array containing all authors and their book information. This array looks like the following:

```
Array
(
[0] => Array
(
[Author] => Array
(
[id] => 1
[name] => Author Name
...
)
[Book] => Array
(
[0] => Array
(
[id] => 1
[isbn] => 1847192564
...
[author_id] => 1
)
[1] => Array
(
[id] => 3
...
)
)
)
[1] => Array
...
...
)
```

Every element of this array contains information about one particular author and his/her related books. Each of these elements has two associated arrays: one is 'an array' containing the author information and the other is 'an array of array' (containing multiple books data as Author to Book association is hasMany) containing information about the books related to that particular author.

We can easily traverse through all the elements of this array and print out all the authors and their related books data. We passed $authors to the view file, and in the view file, we used two loops (first loop is to for each author and second loop is for each related book to this particular author) to print out the author and its related book information.


CakePHP: the rapid development php framework: Authors - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/relationship/authors
Google
Done

Hasin Hayder

Book(s):

- Object-Oriented Programming with PHP5
- Wordpress Complete

James Kennard

Book(s):

- Mastering Joomla! 1.5 Extension and Framework Development

William Rice

Book(s):

- Moodle E-Learning Course Development

Similarly, in BooksController, Book->recursive is set to 1 before calling the Book->find('all') function.

Book->find('all') function also returns a similar array. Every element of this array contains two associated arrays, one contains the book data and the other contains one (not multiple, as Book to Author association is belongsTo) related author data. The returned array looks something like the following:

```
Array
(
[0] => Array
(
[Book] => Array
(
[id] => 1
[isbn] => 1847192564
...
)
[Author] => Array
(
[id] => 1
[name] => Author Name
...
)
)
[1] => Array
...
...
)
```

The result of Book->find('all') is forwarded to the view. In the view file, we looped through the array and printed out all books' information with their related authors' names.

To have a better understanding of how $recursive works, let's assume:

- We have a new model named Chapter.
- And Book is related to Chapter through a hasMany association.

We can now set $recursive to different values based on the amount of data we want back from a Author->find('all') call:

- $recursive = 0: Only authors' data is returned.
- $recursive = 1: Authors' and their associated books.
- $recursive = 2: Author, authors' associated books, and books' related chapters.

In this way, we can fetch out related data of multiple levels by setting the value of $recursive to our desired level.

Always set $recursive = 0, if you don't need to fetch any associated model data.

By default, $recursive is set to 1. In the last example, it was not required to explicitly set its value to 1.

Saving Related Model Data in One-To-Many Relation

Associations can also make saving related data pretty easy. We will now create a simple program that can save a book and can relate the book with an author at the time of saving. We will add some code to the previous code-base to put in the save functionality.

Time for Action: Saving Related Model Data

1. In the BooksController (/app/controllers/books_controller.php), add the FormHelper and write a new action add(),

```
<?php
class BooksController extends AppController {
var $name = 'Books';
var $helpers = array('Form' );
function index() {
$this->Book->recursive = 1;
$books = $this->Book->find('all',
array('fields' =>
array('Book.isbn','Book.title','Author.name')
)
);
$this->set('books', $books);
}
function add() {
if (!empty($this->data)) {
$this->Book->create();
$this->Book->save($this->data);
$this->redirect(array('action'=>'index'));
}
$authors = $this->Book->Author->generateList();
$this->set('authors', $authors);
}
}
```

```
?>
```

2. Create a view for the action '/books/add' (/app/views/add.ctp)

```
<?php echo $form->create('Book');?>
<fieldset>
<legend>Add New Book</legend>
<?php
echo $form->input('isbn');
echo $form->input('title');
echo $form->input('description');
echo $form->input('author_id');
?>
</fieldset>
<?php echo $form->end('Submit');?>
```

3. Point your browser to the following URL and add a book:
   http://localhost/relationship/books/add

What Just Happened?

In the add() action of the BooksController, we first checked if there is any form data sent back from the view. When no data is returned from its view, a form is displayed to take user input. We have text inputs inside the form to take book information. As well as, we have a select-list to select a related author. This select-list has options for all the authors, from where one single author can be selected. CakePHP model function generateList() is used to create this select-list.

Author->generateList() returns an array with author's id=>name as key=>value pairs like the following:

```
Array
(
[1] => Author 1
[2] => Author 2
[3] => ...
)
```

From or through a model class, all model classes related to that model are accessible. Like, Author model can be accessed through Book model like Book->Author and vice-versa.

This function is very handy for creating HTML select tags. We passed the returned result from the function call Book->Author->generateList() to the view through $authors variable. In the view file FormHelper's input() method is used, $form->input('author_id') – this input() method cleverly creates a select tag with options for every author- where names are used as tag-labels and ids are set as tag-values. The output of $form->input('author_id') is something like the following:

```
<select>
<option value="1">Author 1</option>
<option value="2">Author 2</option>
<option value="3">Author 3</option>
...
</select>
```

When the submit button of the form is clicked, all form data is sent back to the BooksController's add() action. Then the form data is passed to the Book->save() function that stores the book data into the database and automatically relates the newly added book with the selected author (saves the selected author's id in the author_id field of books table). When the save operation is done, the controller redirects to index() action and the newly added book can be seen in the list of books.

The prototype of the generateList() function looks like the following:

```
generateList(string $conditions, string $order, int $limit, string
$keyPath, string $valuePath)
```

$conditions, $order, and $limit parameters are just as we would use in the find() function. By default, the generateList() function takes id and name fields of the model respectively as keys and values of the returned array. We can change this behavior using $keyPath and $valuePath parameters.

If we want to get an array of first 10 authors with id and last_name fields of the Author model as key/value pairs and is ordered by the last_name field, we could make use the $order, $limit, $keyPath, and $valuePath parameters like the following:

```
$this->Author->generateList(
   null,

  'last_name ASC',

  10,

  '{n}.Author.initial',

  '{n}.Author.last_name'
);
```

Adding More than One Association of the Same Type

In previous examples, the only association that the Author model has is a hasMany association with the

Book model. This association was created by the following line of code:

```
var $hasMany = 'Book';
```

Here, a string is used to define the hasMany association. It is also possible to do the same thing using an associative array like the following:

```
var $hasMany = array(
'Book' => array(
'className' => 'Book',
)
);
```

Likewise, for the Book model we can do the following:

```
var $belongsTo = array(
'Author' => array(
'className' => 'Author',
)
);
```

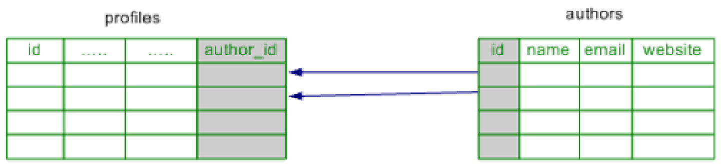
This alternative method is handy if we want to add more than one association of the same type or we need to customize some of the relationship characteristics.

If we have another model Tutorial and our Author model also has a hasMany association with the Tutorial model, we can use the array approach to define these two associations in the Author model:

```
var $hasMany = array(
'Book' => array(
'className' => 'Book',
),
'Tutorial' => array(
'className' => 'Tutorial',
)
);
```

One-To-One Relation

Defining associations for one-to-one relation is very much the same as defining association for a one-to-many relation. Let's look at an example to see how a one-to-one relation can be defined in models. Let's assume, we have a profiles table and the relation between the authors table and the profiles table is one-to-one. To establish this relation, we created a profiles table with a foreign-key author_id.



Now, in both of the model classes, Author and Profile, we have to define associations for this relation. The belongsTo association is always defined in the corresponding model of the table with the foreign-key and hence in the Profile table we have to define a belongsTo association. On the other hand, an author can only have one associated profile—to define the relation from Author model's perspective, we can use a hasOne association.

To resemble this relation, in the Author model a $hasOne attribute should be defined:

```
var $hasOne = array(
'Profile' => array(
'className' => 'Profile',
)
);
```

As well as, a $belongsTo attribute should be defined in the Profile model:

```
var $belongsTo = array(
'Profile' => array(
'className' => 'Author',
)
);
```

In both 'one-to-one' and 'one-to-many' relations, the model that has the foreign-key in its underlying table uses the belongsTo association. The only difference between these two relations is: in the other model, the hasOne association should be used instead of the hasMany.

Customizing Association Characteristics

Defining associations using array has another benefit—we can include some key-value pairs in the associative array to customize the relationship.

A customized hasMany association that defines the relation between authors and books may look like the following:

```
var $hasMany = array(
     'Book' => array(
         'className'     => 'Book',
         'foreignKey'    => 'author_id',
         'conditions'    => 'Book.status = 1',
'fields'      => array('isbn', 'title'),
'order'        => 'Book.released DESC',

      )

 );
```

Let's see how some of these key/value pairs can be used to manipulate a relation:

- className: The name of the associated model.
- foreignKey: The name of the foreign-key involved in this particular relation. In case of belongsTo, it is the name of the foreign-key found in the current model. In case of hasOne or hasMany associations, it holds the foreign-key name found in the corresponding table of the other model. Default is underscored, singular name of the target model (in case of belongsTo, the current model, for hasOne/hasMany—the other model), suffixed with _id.
- conditions: A SQL fragment of WHERE clause to filter the related model records. To fetch all the related books that has status = 1, conditions can be set as 'Book.status = 1'.
- fields: An array of fields specifies the fields to be retrieved when the associated model data is fetched. If not set, it returns all the fields.
- order: A SQL fragment of ORDER BY clause that defines the sorting order for associated records returned. It is not applicable for 'hasOne' and 'belongsTo' associations as there is only one related record.

About the Author :

Ahsanul Bari

Ahsanul Bari is a web application developer from Dhaka, Bangladesh. After graduating from North South University with a bachelor's degree in Computer Science, he has been involved in developing various web applications for local businesses. At a very early stage of his career he felt the need for tools and techniques to build structured and maintainable web applications. That is when he found out about CakePHP. It was love at first sight and he decided to use CakePHP for his future projects. He never had to look back, and from then on he has been heavily using CakePHP for all kinds of projects. Most notably, using CakePHP, he developed an ERP solution for companies involved in urban and land development.

Apart from that, he has also 'irregularly' contributed to the CakePHP Documentation Team. He is also an 'irregular' blogger (http://ahsanity.com and http://ahsanity.wordpress.com). Just when people start to think that he has given up blogging, he is known to write a post from nowhere! Among his friends and colleagues, he is known as a fanboy for CakePHP.

Currently he is working at Trippert Labs, where he has been involved in making a travel-based blogging system, http://www.trippert.com.

Anupom Syam

Anupom Syam is a web application developer from Dhaka, Bangladesh. He started programming back in 1998 in C when he was a high school kid. In his early university years, he met Java and fell in love immediately. Through the years he has become proficient in various aspects of Java (ME, SE, and EE). Early in his career he was engaged mainly in building localized mobile applications. Over time his interest in web technologies grew and he did not hesitate to jump onto the Web 2.0 bandwagon. Over the last five years he has been working with different startups and building web/mobile applications. He currently works as a Development Engineer at Trippert, Inc. where he has been involved in developing a travel-based blogging system http://www.trippert.com (which is developed using CakePHP) as the lead back-end programmer.

He loves to build rich-client web apps with JavaScript/AJAX in the front end and CakePHP/RoR/MySQL in the back end. He still uses Java heavily for his personal fun-time projects. He also maintains blogs: http://anupom.wordpress.com and http://syamantics.com. Besides programming he is interested in many things, ranging from the most recent scientific discoveries to ancient Vedic philosophies.