**CakePHP Database Configuration**

**public** $default = **array**(

'datasource' => 'Database/Mysql',

'persistent' => **false**,

'host' => 'localhost',

'port' => '',

'login' => 'cakeBlog',

'password' => 'c4k3-rUl3Z',

'database' => 'cake\_blog\_tutorial',

'schema' => '',

'prefix' => '',

'encoding' => 'utf8'

);

Once you’ve saved your new database.php file, you should be able to open your browser and see the CakePHP welcome page.

**Optional Configuration**

The security salt is used for generating hashes. Change the default Security.salt value in

/app/Config/core.php. The replacement value should be long, hard to guess and be as random

as you can make it:

/\*\*

\* A random string used in security hashing methods.

\*/

Configure::write('Security.salt', 'pl345e-P45s\_7h3\*S@l7!');

The cipher seed is used for encrypt/decrypt strings. Change the default Security.cipherSeed value

by editing /app/Config/core.php. The replacement value should be a large random integer:

/\*\*

\* A random numeric string (digits only) used to encrypt/decrypt strings.

\*/

Configure::write('Security.cipherSeed', '7485712659625147843639846751');

**A Note on mod\_rewrite**

Occasionally new users will run into mod\_rewrite issues. For example if the CakePHP welcome page looks

a little funny (no images or CSS styles), it probably means mod\_rewrite is not functioning on your system.

Please refer to one of the sections below about URL rewriting for your webserver to get you up and running:

**URL Rewriting**

**Apache and mod\_rewrite (and .htaccess)**

While CakePHP is built to work with mod\_rewrite out of the box–and usually does–we’ve noticed that a few users struggle with getting everything to play nicely on their systems.

Here are a few things you might try to get it running correctly. First look at your httpd.conf. (Make sure you are editing the system httpd.conf rather than a user- or site-specific httpd.conf.)

These files can vary between different distributions and Apache versions. You may also take a look at http://wiki.apache.org/httpd/DistrosDefaultLayout for further information.

1. Make sure you are loading mod\_rewrite correctly. You should see something like:

LoadModule rewrite\_module libexec/apache2/mod\_rewrite.so

**Create Model**

The Model class is the bread and butter of CakePHP applications. By creating a CakePHP model that will interact with our database.

CakePHP’s model class files go in /app/Model, and the file we’ll be creating will be saved to

/app/Model/Post.php.

App::uses('AppModel', 'Model');

**class Post extends** AppModel {}

Naming conventions are very important in CakePHP. By naming our model Post, CakePHP can automatically infer that this model will be used in the PostsController, and will be tied to a database table called posts.

App::uses('AppModel', 'Model'); is path of Parent Class AppModel. App::uses(’AppModel’, ’Model’) ensures that the model is loaded when it is needed.

**Associations: Linking Models Together**

One of the most powerful features of CakePHP is the ability to link relational mapping provided by the model. In CakePHP, the links between models are handled through associations.

**Relationship Types**

The four association types in CakePHP are: hasOne, hasMany, belongsTo, and hasAndBelongsToMany (HABTM).

|  |  |  |
| --- | --- | --- |
| Relationship | Association Type | Example |
| one to one  one to many  many to one  many to many | hasOne  hasMany  belongsTo  hasAndBelongsToMany | A user has one profile.  A user can have multiple recipes.  Many recipes belong to a user.  Recipes have, and belong to, many ingredients. |

To further clarify which way around the associations are defined in the models: If the table of the model contains the foreign key (other\_model\_id), the relation type in this model is always a Model belongsTo OtherModel relation!

**class User extends** AppModel

{

**public** $hasOne = 'Profile';

**public** $hasMany = **array**(

'Recipe' => **array**(

'className' => 'Recipe',

'conditions' => **array**('Recipe.approved' => '1'),

'order' => 'Recipe.created DESC'

)

Associations are defined by creating a class variable named after the association you are defining. In the above example, the first instance of the word ‘Recipe’ is what is termed an ‘Alias’. This is an identifier for the relationship, and can be anything you choose. Usually, you will choose the same name as the class that it references. However, aliases for each model must be unique across the app. For example, it is appropriate to have:

**class User extends** AppModel

{

**public** $hasMany = **array**('MyRecipe' => **array**('className' => 'Recipe',));

**public** $hasAndBelongsToMany = **array**('MemberOf' => **array**('className' => 'Group',));

}

**class Group extends** AppModel

{

**public** $hasMany = **array**('MyRecipe' => **array**('className' => 'Recipe',));

**public** $hasAndBelongsToMany = **array**('Member' => **array**('className' => 'User',));

}

CakePHP will automatically create links between associated model objects. So for example in your User model you can access the Recipe model as:

$this->Recipe->someFunction(); //Like CRUD

There are two ways to describe this relationship in your model files. The simplest method is to set the $hasOne attribute to a string containing the class name of the associated model, as we’ve done above. If you need more control, you can define your associations using array syntax. For example, you might want to limit the association to include only certain records.

**class User extends** AppModel

{

**public** $hasOne = **array**(

'Profile' => **array**('className' => 'Profile',

'conditions' => **array**('Profile.published' => '1'),

'dependent' => **true**));

}

Possible keys for hasOne association arrays include:

• className: the class name of the model being associated to the current model. If you’re defining a

‘User hasOne Profile’ relationship, the className key should equal ‘Profile’.

• foreignKey: the name of the foreign key found in the other model. This is especially handy if you need to define multiple hasOne relationships. The default value for this key is the underscored, singular name of the current model, suffixed with ‘\_id’. In the example above, it would default to ‘user\_id’.

• conditions: an array of find()-compatible conditions or SQL strings such as array(‘Profile.approved => true)

• fields: A list of fields to be retrieved when the associated model data is fetched. Returns all fields by default.

• order: an array of find()-compatible order clauses or SQL strings such as array(‘Profile.last\_name’ => ‘ASC’)

• dependent: When the dependent key is set to true, and the model’s delete() method is called with the cascade parameter set to true, associated model records are also deleted. In this case, we set it true so that deleting a User will also delete her associated Profile.

**belongsTo**

Now that we have Profile data access from the User model, let’s define a belongsTo association in the Profile model in order to get access to related User data. The belongsTo association is a natural complement to the hasOne and hasMany associations: it allows us to see the data from the other direction.

belongsTo: the current model contains the foreign key.

|  |  |
| --- | --- |
| Relation | Schema |
| Banana belongsTo Apple  Profile belongsTo User  Mentor belongsTo Doctor | bananas.apple\_id  profiles.user\_id  mentors.doctor\_id |

Tip: If a model(table) contains a foreign key, it belongsTo the other model(table).

We can also define a more specific relationship using array syntax:

**class Profile extends** AppModel

{

**public** $belongsTo = **array**('User' => **array**('className' => 'User','foreignKey' => 'user\_id'));

}

Possible keys for belongsTo association arrays include:

• className: the class name of the model being associated to the current model. If you’re defining a ‘Profile belongsTo User’ relationship, the className key should equal ‘User’.

• foreignKey: the name of the foreign key found in the current model. This is especially handy if you need to define multiple belongsTo relationships. The default value for this key is the underscored, singular name of the other model, suffixed with \_id.

• conditions: an array of find() compatible conditions or SQL strings such as

array(’User.active’ => true)

• type: the type of the join to use in the SQL query. The default is ‘LEFT’, which may not fit your

needs in all situations. The value ‘INNER’ may be helpful (when used with some conditions) when

you want everything from your main and associated models or nothing at all.

• fields: A list of fields to be retrieved when the associated model data is fetched. Returns all fields by default.

• order: an array of find() compatible order clauses or SQL strings such as array(’User.username’ => ’ASC’)

• counterCache: If set to true, the associated Model will automatically increase or decrease the “[singular\_model\_name]\_count” field in the foreign table whenever you do a save() or delete().

If it’s a string, then it’s the field name to use. The value in the counter field represents the number of related rows. You can also specify multiple counter caches by defining an array. See MultiplecounterCache.

• counterScope: Optional conditions array to use for updating counter cache field.

Once this association has been defined, find operations on the Profile model will also fetch a related User record if it exists:

//Sample results from a $this->Profile->find() call.

**Array**

(

[Profile] => **Array**

(

[id] => 12

[user\_id] => 121

[skill] => Baking Cakes

[created] => 2007-05-01 10:31:01

)

[User] => **Array**

(

[id] => 121

[name] => Gwoo the Kungwoo

[created] => 2007-05-01 10:31:01

)

)

**counterCache - Cache your count()**

This feature helps you cache the count of related data. Instead of counting the records manually via find(’count’), the model itself tracks any addition/deletion towards the associated $hasMany model and increases/decreases a dedicated integer field within the parent model table.

The name of the field consists of the singular model name followed by a underscore and the word “count”:

my\_model\_count

Here are some more examples:

|  |  |  |
| --- | --- | --- |
| Model | Associated Model | Example |
| User  Image  BlogEntry | Image  ImageComment  BlogEntryComment | users.image\_count  images.image\_comment\_count  blog\_entries.blog\_entry\_comment\_count |

**class ImageComment extends** AppModel

{

**public** $belongsTo = **array**('Image' => **array**('counterCache' => **true**,));

}

From now on, every time you add or remove a ImageComment associated to Image, the number within image\_comment\_count is adjusted automatically.

**CounterScope**

You can also specify counterScope. It allows you to specify a simple condition which tells the model when to update (or when not to, depending on how you look at it) the counter value.

**class ImageComment extends** AppModel

{

**public** $belongsTo = **array**('Image' => **array**('counterCache' => 'active\_comment\_count', //custom field name

// only count if "ImageComment" is active = 1

'counterScope' => **array**(

'ImageComment.active' => 1)));

}

**Multiple counterCache**

Since 2.0, CakePHP has supported having multiple counterCache in a single model relation. It is also possible to define a counterScope for each counterCache. Assuming you have a User model and a Message model, and you want to be able to count the amount of read and unread messages for each user.

|  |  |  |
| --- | --- | --- |
| Model | Field | Description |
| User  User  Message | users.messages\_read  users.messages\_unread  messages.is\_read | Count read Message  Count unread Message  Determines if a Message is read or not. |

With this setup, your belongsTo would look like this:

**class Message extends** AppModel

{

**public** $belongsTo = **array**('User' => **array**('counterCache' =>

**array**(

'messages\_read' => **array**('Message.is\_read' => 1),

'messages\_unread' => **array**('Message.is\_read' => 0)

)));

}

**HasMany**

Next step: defining a “User hasMany Comment” association. A hasMany association will allow us to fetch a user’s comments when we fetch a User record.

When keying your database tables for a hasMany relationship, follow this convention:

**hasMany:** the other model contains the foreign key.

|  |  |
| --- | --- |
| Relation | Schema |
| User hasMany Comment  Cake hasMany Virtue  Product hasMany Option | Comment.user\_id  Virtue.cake\_id  Option.product\_id |

We can define the hasMany association in our User model at /app/Model/User.php using the string syntax as follows:

**class User extends** AppModel {

**public** $hasMany = 'Comment';

}

We can also define a more specific relationship using array syntax:

**class User extends** AppModel

{

**public** $hasMany = **array**(

'Comment' => **array**(

'className' => 'Comment',

'foreignKey' => 'user\_id',

'conditions' => **array**('Comment.status' => '1'),

'order' => 'Comment.created DESC',

'limit' => '5',

'dependent' => **true**

));

}

Possible keys for hasMany association arrays include:

• className: the class name of the model being associated to the current model. If you’re defining a ‘User hasMany Comment’ relationship, the className key should equal ‘Comment.’

• foreignKey: the name of the foreign key found in the other model. This is especially handy if you need to define multiple hasMany relationships. The default value for this key is the underscored, singular name of the actual model, suffixed with ‘\_id’.

• conditions: an array of find() compatible conditions or SQL strings such as array(‘Comment.visible’ => true)

• order: an array of find() compatible order clauses or SQL strings such as array(‘Profile.last\_name’ => ‘ASC’)

• limit: The maximum number of associated rows you want returned.

• offset: The number of associated rows to skip over (given the current conditions and order) before fetching and associating.

• dependent: When dependent is set to true, recursive model deletion is possible. In this example, Comment records will be deleted when their associated User record has been deleted.

• exclusive: When exclusive is set to true, recursive model deletion does the delete with a deleteAll() call, instead of deleting each entity separately. This greatly improves performance, but may not be ideal for all circumstances.

• finderQuery: A complete SQL query CakePHP can use to fetch associated model records. This should be used in situations that require highly customized results. If a query you’re building requires a reference to the associated model ID, use the special {$\_\_cakeID\_\_$} marker in the query. For example, if your Apple model hasMany Orange, the query should look something like this: SELECT Orange.\* from oranges as Orange WHERE Orange.apple\_id = {$\_\_cakeID\_\_$};

**hasAndBelongsToMany (HABTM)**

All right. At this point, you can already call yourself a CakePHP model associations professional. You’re already well versed in the three associations that take up the bulk of object relations.

Let’s tackle the final relationship type: hasAndBelongsToMany, or HABTM. This association is used when you have two models that need to be joined up, repeatedly, many times, in many different ways.

The main difference between hasMany and HABTM is that a link between models in HABTM is not exclusive. For example, we’re about to join up our Recipe model with an Ingredient model using HABTM. Using tomatoes as an Ingredient for my grandma’s spaghetti recipe doesn’t “use up” the ingredient. I can also use it for a salad Recipe.

HABTM requires a separate join table that includes both model names.

|  |  |
| --- | --- |
| Relationship | HABTM Table Fields |
| Recipe HABTM  Ingredient->  Cake HABTM Fan ->  Foo HABTM Bar -> | ingredients\_recipes.id, ingredients\_recipes.ingredient\_id,  ingredients\_recipes.recipe\_id  cakes\_fans.id, cakes\_fans.cake\_id, cakes\_fans.fan\_id  bars\_foos.id, bars\_foos.foo\_id, bars\_foos.bar\_id |

Make sure primary keys in tables cakes and recipes have “id” fields as assumed by convention. If they’re different than assumed, they must be changed in model’s primaryKey.

Once this new table has been created, we can define the HABTM association in the model files. We’re going to skip straight to the array syntax this time:

**class Recipe extends** AppModel

{

**public** $hasAndBelongsToMany = **array**('Ingredient' =>**array**(

'className' => 'Ingredient',

'joinTable' => 'ingredients\_recipes',

'foreignKey' => 'recipe\_id',

'associationForeignKey' => 'ingredient\_id',

'unique' => **true**,

'conditions' => '',

'fields' => '',

'order' => '',

'limit' => '',

'offset' => '',

'finderQuery' => '',

'with' => ''

));

}

Possible keys for HABTM association arrays include:

• className: the class name of the model being associated to the current model. If you’re defining a ‘Recipe HABTM Ingredient’ relationship, the className key should equal ‘Ingredient’.

• joinTable: The name of the join table used in this association (if the current table doesn’t adhere to the naming convention for HABTM join tables).

• with: Defines the name of the model for the join table. By default CakePHP will auto-create a model for you. Using the example above it would be called IngredientsRecipe. By using this key you can override this default name. The join table model can be used just like any “regular” model to access the join table directly. By creating a model class with such name and filename, you can add any custom behavior to the join table searches, such as adding more information/columns to it.

• foreignKey: the name of the foreign key found in the current model. This is especially handy if

you need to define multiple HABTM relationships. The default value for this key is the underscored, singular name of the current model, suffixed with ‘\_id’.

• associationForeignKey: the name of the foreign key found in the other model. This is especially handy if you need to define multiple HABTM relationships. The default value for this key is the underscored, singular name of the other model, suffixed with ‘\_id’.

• unique: boolean or string **keepExisting**.

– If true (default value) CakePHP will first delete existing relationship records in the foreign keys table before inserting new ones. Existing associations need to be passed again when updating.

– When false, CakePHP will insert the specified new relationship records and leave any existing

relationship records in place, possibly resulting in duplicate relationship records.

– When set to keepExisting, the behavior is similar to true, but with an additional check so that if any of the records to be added are duplicates of an existing relationship record, the existing relationship record is not deleted, and the duplicate is ignored. This can be useful if, for example, the join table has additional data in it that needs to be retained.

• conditions: an array of find()-compatible conditions or SQL string. If you have conditions on an associated table, you should use a ‘with’ model, and define the necessary belongsTo associations on it.

• fields: A list of fields to be retrieved when the associated model data is fetched. Returns all fields by default.

• order: an array of find()-compatible order clauses or SQL strings

• limit: The maximum number of associated rows you want returned.

• offset: The number of associated rows to skip over (given the current conditions and order) before fetching and associating.

• finderQuery: A complete SQL query CakePHP can use to fetch associated model records. This should be used in situations that require highly customized results.

Once this association has been defined, find operations on the Recipe model will also fetch related Tag records if they exist:

// Sample results from a $this->Recipe->find() call.

**Array**

(

[Recipe] => **Array**(

[id] => 2745

[name] => Chocolate Frosted Sugar Bombs

[created] => 2007-05-01 10:31:01

[user\_id] => 2346

)

[Ingredient] => **Array**(

[0] => **Array**

(

[id] => 123

[name] => Chocolate

)

[1] => **Array**

(

[id] => 124

[name] => Sugar

)

[2] => **Array**

(

[id] => 125

[name] => Bombs

)

))

Remember to define a HABTM association in the Ingredient model if you’d like to fetch Recipe data when using the Ingredient model.

Note: HABTM data is treated like a complete set. Each time a new data association is added, the complete set of associated rows in the database is dropped and created again so you will always need to pass the whole data set for saving. For an alternative to using HABTM, see hasMany through (The Join Model).

**Saving Related Model Data (HABTM)**

// in the controller:

$this->set('tags', $this->Recipe->Tag->find('list'));

// in the view:

$this->Form->input('Tag');

**hasMany through (The Join Model)**

It is sometimes desirable to store additional data with a many-to-many association. Consider the following Student hasAndBelongsToMany Course Course hasAndBelongsToMany Student In other words, a Student can take many Courses and a Course can be taken by many Students. This is a simple many-to-many association demanding a table such as this:

id | student\_id | course\_id

Now what if we want to store the number of days that were attended by the student on the course and their final grade? The table we’d want would be:

// Student.php

**class Student extends** AppModel {

**public** $hasMany = **array**(

'CourseMembership'

);

}

// Course.php

**class Course extends** AppModel

{

**public** $hasMany = **array**('CourseMembership');

}

// CourseMembership.php

**class CourseMembership extends** AppModel

{

**public** $belongsTo = **array**( 'Student', 'Course' );

}

**Creating and Destroying Associations on the Fly**

Sometimes it becomes necessary to create and destroy model associations on the fly. This may be for any number of reasons:

• You want to reduce the amount of associated data fetched, but all your associations are on the first level of recursion.

• You want to change the way an association is defined in order to sort or filter associated data.

**class Leader extends** AppModel

{

**public** $hasMany = **array**(

'Follower' => **array**(

'className' => 'Follower',

'order' => 'Follower.rank'));

}

**class Follower extends** AppModel

{

**public** $name = 'Follower';

}

**public function** some\_action() {

// This fetches Leaders, and their associated Followers

$this->Leader->find('all');

// Let's remove the hasMany...

$this->Leader->unbindModel(

**array**('hasMany' => **array**('Follower')));

// Now using a find function will return

// Leaders, with no Followers

$this->Leader->find('all');

// NOTE: unbindModel only affects the very next

// find function. An additional find call will use

// the configured association information.

// We've already used find('all') after unbindModel(),

// so this will fetch Leaders with associated

// Followers once again...

$this->Leader->find('all');

}

Removing or adding associations using bind- and unbindModel() only works for the next find operation unless the second parameter has been set to false. If the second parameter has been set to false, the bind remains in place for the remainder of the request.

Here’s the basic usage pattern for unbindModel():

$this->Model->unbindModel(

**array**('associationType' => **array**('associatedModelClassName'))

);

Now that we’ve successfully removed an association on the fly, let’s add one.

**public function** another\_action() {

// There is no Leader hasMany Principles in

// the leader.php model file, so a find here

// only fetches Leaders.

$this->Leader->find('all');

// Let's use bindModel() to add a new association

// to the Leader model:

$this->Leader->bindModel(

**array**('hasMany' => **array**(

'Principle' => **array**(

'className' => 'Principle'))));

// Now that we're associated correctly,

// we can use a single find function to fetch

// Leaders with their associated principles:

$this->Leader->find('all');

}

There you have it. The basic usage for bindModel() is the encapsulation of a normal association array inside an array whose key is named after the type of association you are trying to create:

$this->Model->bindModel(

**array**('associationName' => **array**(

'associatedModelClassName' => **array**(

// normal association keys go here...

))));

**Retrieving Your Data**

As stated before, one of the roles of the Model layer is to get data from multiple types of storage. The CakePHP Model class comes with some functions that will help you search for this data, sort it, paginate it, and filter it. The most common function you will use in models is Model::find().

**Find**

find(string $type = ’first’, array $params = array())

Find is the multifunctional workhorse of all model data-retrieval functions. $type can be ’all’,

’first’, ’count’, ’list’, ’neighbors’ or ’threaded’, or any custom finder you can define.

Keep in mind that $type is case-sensitive. Using an upper case character (for example, All) will not produce the expected results.

$params is used to pass all parameters to the various types of find(), and has the following possible keys by default, all of which are optional:

**array**(

'conditions' => **array**('Model.field' => $thisValue), //array of conditions

'recursive' => 1, //int

//array of field names

'fields' => **array**('Model.field1', 'DISTINCT Model.field2'),

//string or array defining order

'order' => **array**('Model.created', 'Model.field3 DESC'),

'group' => **array**('Model.field'), //fields to GROUP BY

'limit' => n, //int

'page' => n, //int

'offset' => n, //int

'callbacks' => **true** //other possible values are false, 'before', 'after'

)

**find(‘first’)**

find(’first’, $params) will return one result. You’d use this for any case where you expect only

one result. Below are a couple of simple (controller code) examples:

**public function** some\_function()

{

// ...

$semiRandomArticle = $this->Article->find('first');

$lastCreated = $this->Article->find('first', **array**(

'order' => **array**('Article.created' => 'desc')));

$specificallyThisOne = $this->Article->find('first', **array**(

'conditions' => **array**('Article.id' => 1)));

// ...

}

In the first example, no parameters at all are passed to find, so no conditions or sort order will be used. The format returned from find(’first’) call is of the form:

**Array**

(

[ModelName] => **Array**

(

[id] => 83

[field1] => value1

[field2] => value2

[field3] => value3

)

[AssociatedModelName] => **Array**

(

[id] => 1

[field1] => value1

[field2] => value2

[field3] => value3

))

**find(‘count’)**

find(’count’, $params) returns an integer value. Below are a couple of simple (controller code)

examples:

**public function** some\_function()

{

// ...

$total = $this->Article->find('count');

$pending = $this->Article->find('count', **array**(

'conditions' => **array**('Article.status' => 'pending')));

$authors = $this->Article->User->find('count');

$publishedAuthors = $this->Article->find('count', **array**(

'fields' => 'DISTINCT Article.user\_id',

'conditions' => **array**('Article.status !=' => 'pending')));

// ...

}

**find(‘list’)**

find(’list’, $params) returns an indexed array, useful for any place where you would want a list,

such as for populating input select boxes. Below are a couple of simple (controller code) examples:

**public function** some\_function() {

// ...

$allArticles = $this->Article->find('list');

$pending = $this->Article->find('list', **array**(

'conditions' => **array**('Article.status' => 'pending')

));

$allAuthors = $this->Article->User->find('list');

$allPublishedAuthors = $this->Article->find('list', **array**(

'fields' => **array**('User.id', 'User.name'),

'conditions' => **array**('Article.status !=' => 'pending'),

'recursive' => 0

));

// ...

}

**find(‘threaded’)**

find(’threaded’, $params) returns a nested array, and is appropriate if you want to use the

parent\_id field of your model data to build nested results. Below are a couple of simple (controller code) examples:

**public function** some\_function()

{

// ...

$allCategories = $this->Category->find('threaded');

$comments = $this->Comment->find('threaded', **array**(

'conditions' => **array**('article\_id' => 50)));

// ...

}

A better way to deal with nested data is using the Tree behavior

In the above code example, $allCategories will contain a nested array representing the whole category structure. The results of a call to find(’threaded’) will be of the following form:

**Array**

(

[0] => **Array**

(

[ModelName] => **Array**

(

[id] => 83

[parent\_id] => **null**

[field1] => value1

[field2] => value2

[field3] => value3

)

[AssociatedModelName] => **Array**

(

[id] => 1

[field1] => value1

[field2] => value2

[field3] => value3

)

[children] => **Array**

(

[0] => **Array**

(

[ModelName] => **Array**

(

[id] => 42

[parent\_id] => 83

[field1] => value1

[field2] => value2

[field3] => value3

)

[AssociatedModelName] => **Array**

(

[id] => 2

[field1] => value1

[field2] => value2

[field3] => value3

)

[children] => **Array**

(

)

)

...

)

))

**find(‘neighbors’)**

find(’neighbors’, $params) will perform a find similar to ‘first’, but will return the row before

and after the one you request. Below is a simple (controller code) example:

**public function** some\_function() {

$neighbors = $this->Article->find(

'neighbors',

**array**('field' => 'id', 'value' => 3)

);

}

**Array**

(

[prev] => **Array**

(

[ModelName] => **Array**(

[id] => 2

[field1] => value1

[field2] => value2

...

)

[AssociatedModelName] => **Array**

(

[id] => 151

[field1] => value1

[field2] => value2

...

)

)

[next] => **Array**

(

[ModelName] => **Array**

(

[id] => 4

[field1] => value1

[field2] => value2

...

)

[AssociatedModelName] => **Array**

(

[id] => 122

[field1] => value1

[field2] => value2

...

)

))

Note how the result always contains only two root elements: prev and next. This function does not honor a model’s default recursive var. The recursive setting must be passed in the parameters on each call.

**Creating custom find types**

The find method is flexible enough to accept your custom finders. This is done by declaring your own types in a model variable and by implementing a special function in your model class.

A Model Find Type is a shortcut to find() options. For example, the following two finds are equivalent.

$this->User->find('first');

$this->User->find('all', **array**('limit' => 1));

The following are core find types:

• first

• all

• count

• list

• threaded

• neighbors