Tree class Documentation

Introduction

This package is a set of three classes which handle the creation and manipulation of a Tree structure. It consists of the Tree class, the Tree_Node class and the Tree_NodeCollection class.

Author

This code was written by Richard Heyes.

API Reference

Tree class

Methods

bool hasChildren()
This method returns true or false indicating whether this Tree has any child nodes.

Tree factory(Tree_Factory_Iterator it)
This method provides a way to create Tree structures from Various data sources. Two provided methods classes are:

- Tree_Factory_List
- Tree_Factory_FileSystem

The first creates a Tree structure from an array of "paths", whilst the second creates a tree structure mapped to a given directory structure on disk. Eg:

```
/**
* Create a structure from an array of "paths". Optional second
* arg to Tree_Factory_List
* is the separator (defaults to "/").
*/
$rit = new Tree_Factory_List(array('foo/bar/jello',
                                   'foo/bar',
                                   'blaat/ping/plod/monkey'));
$tree = Tree::factory($rit);
```

Or:

```
/**
* Create a structure from a filesystem path. Recurses down
* through directories
*/
$rit = new Tree_Factory_FileSystem('./');

$tree = Tree::factory($rit);
```

object *createFromList(array data [, string separator])*

The list class as you can see will take an array of “paths” like that shown below and produce a tree structure from it.

```
$data[] = 'node1/node1_1/node1_1_1';
$data[] = 'node1/node1_2/node1_2_1';
$data[] = 'node2/node2_1/node2_1_1';
```

The separator can be defined by the second optional argument, and it defaults to “/”. The above example data would create a tree like this:

```
+-node1
|  +-node1_1
|  |  +-node1_1_1
|  +-node1_2
|     +-node1_2_1
+-node2
   +-node2_1
      +-node2_1_1
```

The filesystem class simply maps to the given directory.

Important:

The factory method is not limited to these two classes. Any class that implements the Tree_Factory_Iterator interface (located in Tree/Factory/Iterator.php) can be given to the Tree::factory() method. This allows you to make your own Factory classes. The Tree_Factory_Iterator inherits from RecursiveIterator, so all methods defined in these two Interfaces must be implemented.

int merge(object $obj...)
This method merges the given object(s) with the current Tree. The objects given to the method can be either Tree objects or Tree_Node objects. A Tree_Node object will simply be added To the current tree, whilst a Tree will have all its immediate Nodes copied to the current tree.

Tree_Node class

Methods

Constructor([mixed $tag])
The constructor simply takes the tag data as its sole argument. Tag data is simply the data assigned to the node. It can be any type (string, array, object etc) and defaults to NULL.

void setTag(mixed $tag)
An accessor method to set the tag data of this node.

mixed getTag()
An accessor method which returns the tag data of this node.

object &prevSibling()
This method returns a reference to the previous node in this

nodes parent node collection. If there is no such node then NULL is returned.

object &nextSibling()
This method returns a reference to the next node in this nodes parent node collection. If there is no such node then NULL is returned.

v*oid remove()*
This method removes this node from its parent node collection. If this node has not been added to a Tree or Tree_Node, then this method will do nothing.

object getParent()
Returns the parent Tree_Node object if there is one.

o*bject getTree()*
Returns the encompassing Tree object.

bool hasChildren()
Returns true or false indicating whether this node has any child nodes.

integer depth()
Returns the depth of this node in the tree. This depth is zero based, so root nodes will have a depth of 0 (zero), child nodes of root nodes will have a depth of 1 (one) and so on.

bool isChildOf(object $parent)
Returns true/false as to whether the given Tree_Node object is the parent of the current object (ie the instance of which the method is called).

moveTo(object &$newParent)
This moves this node and all its child nodes to the given Tree or Tree_Node object.

copyTo(object &$newParent)
This copies the node and all its child nodes to the given Tree or Tree_Node object. Technically, new nodes are created with copies of the tag data, since that is the only distinct item between nodes.

Tree_NodeCollection class

Methods

o*bject &add(object &$node)*
This method adds a new node to the node collection and returns a reference to the node inside the collection. This return value can then be used to add child nodes if required. The argument should be a Tree_Node object. Example:

$myTree->nodes->add(new Tree_Node('my tag data'))

In the above the tree structure *$myTree* has a property *nodes*

which is a Tree_NodeCollection object. The *add()* method is being called with a new *Tree_Node* object as its argument.

o*bject &firstNode()*
This method returns a reference to the first node in this node collection. If the collection is empty, NULL is returned.

object &lastNode()
This method returns a reference to the last node in this node collection. If the collection is empty, NULL is returned.

o*bject &removeNodeAt(int $index)*
This method removes the node at the specified (zero based) index. It returns the removed node, or NULL if there was no node at the given index.

b*ool removeNode(object &$node [, bool $search])*
This method attempts to remove the supplied node from the Node collection. Nodes are compared using internal UIDs. The second *$search* argument determines whether the method will search recursively down the tree for the supplied node. TRUE is returned on success, FALSE otherwise.

i*nt indexOf(object $node)*
Returns the (zero based) index of the supplied node object in the node collection. Uses internal UIDs to compare the nodes. If not found, NULL is returned.

int count([bool $search])
This method returns the number of child nodes in this Node collection. Optionally acts recursively and returns the cumulative count, based on the $search argument.

array getFlatList()
This method returns a single dimensional indexed array of all the nodes in the node collection. It acts recursively, and returns references to the nodes.

void traverse(callback $function[, mixed $data])
Traverses the node collection applying a function to each and every node. The function name given (though this can be anything you can supply to call_user_func(), not just a name) should take a single argument which is the node object (Tree_Node class) and this is passed by reference. You can then access the nodes data by using the getTag() method. The traversal goes from top to bottom, left to right (ie same order as what you get from getFlatList()). Optionally, this method can take a second argument, which is user supplied data which gets passed to the callback function. Nothing is done with it, it simply gets passed through.

object &search(mixed $data [, bool $strict])
Searches the node collection for a node with a tag matching what you supply. This is a simple "tag == your data"

comparison (=== if strict option is applied), more advanced comparisons can be made using the traverse() method with your own callback function. This function returns an array of matching Tree_Node objects.

o*bject moveTo(object &$newParent)*
Move the nodes in this node collection to be child nodes of the given Tree or Tree_Node object.

o*bject copyTo(object &$newParent)*
Copies the nodes in this node collection to be child nodes of the given Tree or Tree_Node object.

Example

See the example.php script for sample code.

License

This code is distributed under the BSD license. Wishlist fulfilment is of course always appreciated:

Richard Heyes http://www.phpguru.org/wishlist