Zend_Db_NestedSet - Graham Anderson

<ac:macro ac:name="unmigrated-inline-wiki-markup"><ac:plain-text-body><![CDATA[

<ac:macro ac:name="unmigrated-inline-wiki-markup"><ac:plain-text-body><![CDATA[

Zend Framework: Zend_Db_NestedSet Component Proposal

Proposed Component Name	Zend_Db_NestedSet
Developer Notes	http://framework.zend.com/wiki/display/ZFDEV/Zend_Db_NestedSet
Proposers	Graham Anderson
Zend Liaison	TBD
Revision	1.0 - 16 January 2010: Initial Draft. (wiki revision: 6)

Table of Contents

1. Overview
2. References
3. Component Requirements, Constraints, and Acceptance Criteria
4. Dependencies on Other Framework Components
5. Theory of Operation
6. Milestones / Tasks
7. Class Index
8. Use Cases
9. Class Skeletons

1. Overview

Zend_Db_NestedSet is an implementation of the modified pre-order traversal pattern for storing hierarchical data structures in a relational database. This data storage pattern allows for fast retrieval of tree-like structures that are used in common web page elements.

2. References

- Joe Celko's Trees and Hierarchies in SQL for Smarties
- original php4 nstrees

3. Component Requirements, Constraints, and Acceptance Criteria

- This component **will** support single tree or multi tree database tables.
- This component **will** allow data queried from the database table to be rebuilt into a tree-like structure suitable for use with Zend_Navigation
- This component **will** allow data to be presented either as a multi-dimensional associate array or as a recursive iterator object.
- This component **will** allow tree nodes or branches to be moved or deleted within the database

- This component **will** provide functionality to retrieve a whole tree or a branch(sub tree).
- This component **will** provide functionality to get the path, depth, siblings or immediate descendants of a specific tree node.
- This component **will not** provide retrieval or modification of data using materialised path or adjency list techniques
- This component **will** provide common interfaces that other hierarchical retrieval techniques might use.

4. Dependencies on Other Framework Components

- Zend_Db_Table

5. Theory of Operation

The component provides for fast and efficient retrieval of hierarchical data using a minimum amount of database queries.

6. Milestones / Tasks

- Milestone 1: [DONE] Initial class skeletons & proposal
- Milestone 2: [DONE] Working prototype gitHub repo
- Milestone 3: Working prototype checked into the incubator supporting use cases #3 and #4.
- Milestone 4: Unit tests exist, work, and are checked into SVN.
- Milestone 5: Initial documentation exists.

7. Class Index

- Zend_Db_TreeInterface (interface)
- Zend_Db_TreeNodeinterface (interface)
- Zend_Db_TreeBranchInterface (interface)
- Zend_Db_NestedSet
- Zend_Db_NestedSet_Node
- Zend_Db_NestedSet_Branch
- Zend_Db_NestedSet_Exception

8. Use Cases

UC-01

Get the full tree

```
$table = new Zend_Db_NestedSet('categories');

    $tree = $table->getTree()

    // or getTree($rootId) for mutli tree table
```

UC-02

Get the full tree as a multidimensional associative array

```
$table = new Zend_Db_NestedSet('categories');

    $tree = $table->getTree()->toMultiArray()

    // or getTree($rootId) for mutli tree table

    $navigation = new Zend_Navigation($tree);
```

UC-03

Get the full tree as a recursive iterator

```
$table = new Zend_Db_NestedSet('categories');

    /*
    CREATE TABLE IF NOT EXISTS `categories` (
      `categoryId` int NOT NULL AUTO_INCREMENT,
      `rootId` int NOT NULL,
      `categoryName` varchar(255) COLLATE utf8_unicode_ci NOT NULL,
      `parentId` int NOT NULL,
      `lft` int NOT NULL,
      `rgt` int NOT NULL,
      PRIMARY KEY (`categoryId`)
    ) ;

    INSERT INTO `categories` (`categoryId`, `rootId`, `categoryName`, `parentId`, `lft`, `rgt`) VALUES
        (1, 0, 'Electronics', 0, 1, 16),
        (2, 0, 'Televisions', 1, 2, 7),
        (3, 0, 'LCD', 2, 3, 4),
        (4, 0, 'Plasma', 2, 5, 6),
        (5, 0, 'Audio Equipment', 1, 8, 15),
        (6, 0, 'MP3 Players', 5, 9, 10),
        (7, 0, 'Radios', 5, 11, 12),
        (8, 0, 'CD Players', 5, 13, 14);

    */

    $tree = $table->getTree()->toIterator()

    // or getTree($rootId) for mutli tree table

    foreach($tree as $node) {
        echo str_repeat('--',$tree->getDepth()) . $node->categoryName;
    }


    */ Outputs the following

      Electronics
      --Televisions
      ----LCD
      ----Plasma
      --Audio Equipment
      ----MP3 Players
      ----Radios
      ----CD Players

    */
```

UC-04

Get a branch starting at a specific node

```
$table = new Zend_Db_NestedSet('categories');

    $sql = $table->select()
                 ->where('categoryName = ?', 'Audio Equipment');

    $branch = $table->fetchBranch($sql);
```

UC-05

Move a single node or a branch

```
$table = new Zend_Db_NestedSet('categories');

    // Add new sub cateogry 'Apple' who's parent is the root 'Electronics'
    $apple = $table->addNode(
        $table->getRoot(),
        array('categoryName' => 'Apple')
    );

    // Add ipods under audio equipment
    $audio = $table->fetchNode(
        $table->select()->where('categoryName=?','Audio Equipment')
    );
    $ipods = $table->addNode($audio, array('categoryName' => 'iPods'));

    // Move ipods to 'Apple' category as the first child
    $newLocation = $table->moveNode(
        $ipods,
        $apple,
        Zend_Db_NestedSet::FIRST_CHILD
    );
```

9. Class Skeletons

Zend_Db_TreeInterface

```
<?php
interface Zend_Db_TreeInterface
{
    /**
     * Check whether tree has a root node
     *
     * @return bool
     */
    public function hasRoot();

    /**
     * Insert a new root node un a multi-tree table
     *
     * @param array $values array of column/values
     * @return Zend_Db_Tree_NodeInterface $root
     */
    public function setRootNode(array $values = array());

    /**
```

```
     * Add a new node
     *
     * @param Zend_Db_TreeNodeInterface $relation
     * @param mixed $data
     * @param string $position
     */
    public function addNode(Zend_Db_TreeNodeInterface $relation, $data, $position = null);

   /**
     * Move a node (and any descendants) of the tree to the specified location
     *
     * @param Zend_Db_Tree_NodeInterface $origin
     * @param Zend_Db_Tree_NodeInterface $destination
     * @param string $position
     */
    public function moveNode(Zend_Db_TreeNodeInterface $origin,
                             Zend_Db_TreeNodeInterface $destination,
                             $position = null);

    /**
     * Delete a node (and all its children!)
     *
     * @param Zend_Db_TreeNodeInterface $node
     */
    public function deleteNode(Zend_Db_TreeNodeInterface $node);

    /**
     * Fetch a node via a select object
     *
     * @param Zend_Db_Table_Select $select
     */
    public function fetchNode(Zend_Db_Table_Select $select);

    /**
     * Return a branch, starting at $node or the whole tree
     *
     * @param mixed Zend_Db_Table_Select|Zend_Db_TreeNodeInterface $node
     */
    public function fetchBranch($node = null);

    /**
     * Add a child node, 'last child' position by default
     *
     * @param Zend_Db_TreeNodeInterface $parent
     * @param array $data
     */
    public function addChild(Zend_Db_TreeNodeInterface $parent,
                             array $data,
                             $position = Zend_Db_NestedSet::LAST_CHILD);

    /**
     * Add a sibling node, 'next' position by default
     *
     * @param Zend_Db_TreeNodeInterface $relation
     * @param array $data
     */
    public function addSibling(Zend_Db_TreeNodeInterface $relation,
                               array $data,
```

```
                                $position = Zend_Db_NestedSet::NEXT_SIBLING);

}
```

Zend_Db_TreeBranchInterface

```
<?php
interface Zend_Db_TreeBranchInterface
{
    public function toMultiArray();
    public function toIterator();
}
```

Zend_Db_TreeNodeInterface

```
<?php
interface Zend_Db_TreeNodeInterface
{
    /**
     * Return whether or not this is a leaf node
     *
     * @return bool
     */
    public function isLeaf();

    /**
     * Returns whether or not this is a root node
     *
     * @return bool
     */
    public function isRoot();

    /**
     * Return any siblings of this node or null if none
     *
     * @return mixed|Zend_Db_Tree_BranchIterator
     */
    public function getSiblings();

    /**
     * Returns array of values representing the ancestry of this node
     *
     * @return array
     */
    public function getPath();

    /**
     * Returns the parent node
     *
     * @return Zend_Db_Tree_NodeInterface
     */
    public function getParent();

    /**
     * Returns whether this node has descendants
     *
     * @return bool
     */
    public function hasDescendants();

    /**
     * Return a branch filled with decendants of this node
     *
     * @return Zend_Db_Tree_BranchInterface
     */
    public function getDescendants();

}
```

Zend_Db_NestedSet

```
<?php
abstract class Zend_Db_NestedSet extends Zend_Db_Table
                                 implements Zend_Db_TreeInterface
{
 protected $_rowClass = "Zend_Db_NestedSet_Node";

 protected $_rowsetClass = "Zend_Db_NestedSet_Branch";
```

```
    const FIRST_CHILD       = 'firstChild';
    const LAST_CHILD        = 'lastChild';
    const PREVIOUS_SIBLING  = 'previousSibling';
    const NEXT_SIBLING      = 'nextSibling';

    const LEFT_KEY          = 'lft';
    const RIGHT_KEY         = 'rgt';
    const PARENT_KEY        = 'parentId';
    const ROOT_KEY          = 'rootId';

    /**
     * enable/disable table with mutliple trees
     *
     * @var unknown_type
     */
    protected $_multiRoot = false;

    /**
     * Column identifier for the "left" node value
     *
     * @var string
     */
    protected $_lftKey;

    /**
     * Column identifier for the "right" node value
     *
     * @var string
     */
    protected $_rgtKey;

    /**
     * The parent id column for fast tree re-construction
     *
     * @var string
     */
    protected $_parentKey;

    /**
     * The root id column for multi tree tables
     *
     * @var string
     */
    protected $_rootKey;

    /**
     * Constructor
     *
     * @param array $config
     */
    public function __construct($config = null)
    {
        parent::__construct($config);

        if (isset($config['leftKey'])) {
            $this->_lftKey = (string) $config['leftKey'];
        } else {
            $this->_lftKey = self::LEFT_KEY;
        }

        if (isset($config['rightKey'])) {
            $this->_rgtKey = (string) $config['rightKey'];
        } else {
            $this->_rgtKey = self::RIGHT_KEY;
        }

        if (isset($config['parentKey'])) {
            $this->_parentKey = (string) $config['parentKey'];
        } else {
```

```
            $this->_parentKey = self::PARENT_KEY;
        }

        if (isset($config['rootKey'])) {
            $this->_rootKey = (string) $config['rootKey'];
        } else {
            $this->_rootKey = self::ROOT_KEY;
        }

        if(isset($config['multiRoot'])) {
            $this->_multiRoot = (bool)$config['multiRoot'];
        }
    }

    /**
     * Set the left node key
     *
     * @param string $left
     * @return void
     */
    public function setLeftKey($left)
    {
        $this->_lftKey = (string)$left;
    }

    /**
     * Set the right key
     *
     * @param string $right
     */
    public function setRightKey($right)
    {
        $this->_rgtKey = (string)$right;
    }

    /**
     * Return name of left key column
     * @return string
     */
    public function getLeftKey()
    {
        return $this->_lftKey;
    }

    /**
     * return name of right key colum
     * @return string
     */
    public function getRightKey()
    {
        return $this->_rgtKey;
    }

    /**
     * Returns the parentID key
     *
     * @return string
     */
    public function getParentKey()
    {
        return $this->_parentKey;
    }

    /**
     * Return the rootId key name
     * @return string
     */
    public function getRootkey()
    {
        return $this->_rootKey;
```

```
    }

    /**
     * Set whether this table is multi root(tree) capables
     * @param bool $mutli
     */
    public function setMultiRoot($multi = true)
    {
        $row = $this->fetchNode($this->select()->where($this->_lftKey . '=1'));
        if($row && !$this->_multiRoot) {
            throw new Zend_Db_NestedSet_Exception('Cannot set table to multi-root.');
        }
        $this->_multiRoot = $multi;
        return $this;
    }

    /**
     * Check whether tree has a root node
     *
     * @return bool
     */
    public function hasRoot($rootId = null)
    {
        if ($this->_multiRoot && ($rootId == null)) {
            throw new Zend_Db_NestedSet_Exception('Ambiguous rootId');
        }

        $select = $this->select();
        $select->where($this->_lftKey . '=1');
        if($this->_multiRoot && !($rootId == null)) {
            $select->where($this->getRootkey() . '=?', $rootId);
        }
        $root = $this->fetchRow($select);
        if ($root === null) {
            return false;
        }
        return true;
    }

    /**
     * Set an initial or add a new root node
     *
     */
    protected function _addRoot(array $data = array())
    {
        $data[$this->_lftKey] = 1;
        $data[$this->_rgtKey] = 2;
        $data[$this->_parentKey] = 0;

        if(!$this->_multiRoot && $this->hasRoot()) {
            throw new Zend_Db_NestedSet_Exception('Not a multi-tree table.');
        }

        $root = $this->fetchNew();
        $root->setFromArray($data);
        $id = $root->save();

        $savedRoot = $this->find($id);
        $savedRoot = $savedRoot[0];
        $savedRoot->{$this->getRootkey()} = $savedRoot->{$this->_primary[1]};
        $savedRoot->save();

        return $savedRoot;
    }

    /**
     * Set a root node, typically done when setting up a single tree table
     *
     * @param array $values array of coulmn values
     * @return Zend_Db_Node_NestedSet $root node
```

```
     */
    public function setRootNode(array $values = array()) {

        if ($this->_multiRoot) {
            throw new Zend_Db_NestedSet_Exception(
                'Use addRootNode() on multi-tree tables.'
            );
        }
        return $this->_addRoot($values);
    }

    /**
     * Add a new root node to a multi-tree table
     * @param array $values
     * @return Zend_Db_Node_NestedSet $root node
     */
    public function addRootNode(array $values = array())
    {
        if(!$this->_multiRoot) {
            throw new Zend_Db_NestedSet_Exception(
                'Use setRootNode() on single tree tables.'
            );
        }
        return $this->_addRoot($values);
    }

    /**
     * Return whether the table is in multi mode or not
     *
     * @return bool
     */
    public function isMultiRoot()
    {
        return $this->_multiRoot;
    }

    /**
     * Return a root node
     * @param $roodId
     * @return Zend_Db_Node_NestedSet
     */
    public function getRoot($rootId = null)
    {
        if($this->_multiRoot && $rootId == null) {
            throw new Zend_Db_NestedSet_Exception('Invalid rootId arguement.');
        }
        if(!$this->_multiRoot) {
            $rootId = 0;
        }
        $sql = $this->select()
                    ->where($this->getLeftKey() . '= ?', 1);
        if($this->_multiRoot) {
            $sql->where($this->getRootkey() . '= ?', $rootId);
        }
        return $this->fetchRow($sql);
    }

    /**
     * Shift the left and right values of nodes to make room for the addition of
     * a new node. Calls to this method should be wrapped in a transaction.
     *
     * @param int $delta
     * @param int $value the node value to compare against
     * @param string|int $rootId the rootId in mutli tree tables
     * @return void
     */
    public function applyDeltaShift($delta, $value, $rootId = null)
    {
        if($this->_multiRoot && !$rootId) {
            throw new Zend_Db_NestedSet_Exception('You must specify a rootId.');
```

```
        }

        $lftKey = $this->getLeftKey();
        $rgtKey = $this->getRightKey();

        $leftWhere = $this->_db->quoteInto($lftKey . '>=?', $value);
        $rightWhere = $this->_db->quoteInto($rgtKey . '>=?', $value);

        if ($this->_multiRoot) {
            $rootClause = ' AND ' . $this->_db->quoteInto(
                                       $this->getRootkey() . '=?',$rootId
                                   );
            $leftWhere .= $rootClause;
            $rightWhere .= $rootClause;
        }

        $this->update(
            array(
                $lftKey => new Zend_Db_Expr($lftKey . '+' . $delta)
            ),
            $leftWhere
        );

        $this->update(
            array(
                $rgtKey => new Zend_Db_Expr($rgtKey . '+' . $delta)
            ),
            $rightWhere
        );
    }

    /**
     * Change the lef/right values of nodes between a range to allow for moving
     * branches. Any calls to this should be wrapped in a transaction. Auto-commit
     * is bad mkay.
     *
     * @param Zend_Db_Table_Row $node
     * @param int $delta
     * @param string|int $rootId the rootId in mutli tree tables
     * @return void
     */
    public function applyDeltaRange($node, $delta, $rootId = null)
    {
        if($this->_multiRoot && !$rootId) {
            throw new Zend_Db_NestedSet_Exception('You must specify a rootId.');
        }

        $lftKey = $this->getLeftKey();
        $rgtKey = $this->getRightKey();

        $leftWhere = array(
            $this->_db->quoteInto($lftKey . '>=?', $node->$lftKey),
            $this->_db->quoteInto($lftKey . '<=?', $node->$rgtKey)
        );
        $rightWhere = array(
            $this->_db->quoteInto($rgtKey . '>=?', $node->$lftKey),
            $this->_db->quoteInto($rgtKey . '<=?', $node->$rgtKey )
        );

        if ($this->_multiRoot) {
            $rootClause = $this->_db->quoteInto(
                $this->getRootkey() . '=?',$rootId
            );
            $leftWhere[] = $rootClause;
            $rightWhere[] = $rootClause;
        }

        $this->update(array(
                $lftKey => new Zend_Db_Expr( $lftKey . '+' . $delta )
                ),
```

```
            $leftWhere
        );

    $this->update(array(
            $rgtKey => new Zend_Db_Expr( $rgtKey . '+' . $delta )
            ),
            $rightWhere
        );
}

/**
 * Add a new node, optionally with a position, defaults to adding new node
 * as a last(right most) child.
 *
 * @param Zend_Db_Table_Row $relation
 * @param mixed $data
 * @param string $position
 * @return Zend_Db_Table_Row $row
 * @todo check if save() performs a db commit
 */
public function addNode(Zend_Db_TreeNodeInterface $relation, $data,
                        $position = null)
{
    if (is_null($position)) {
        $position = self::LAST_CHILD;
    }

    $row = $this->fetchNew()->setFromArray($data);

    if ($position == self::PREVIOUS_SIBLING ||
        $position == self::NEXT_SIBLING)
    {
        $parent = $relation->{$this->getParentKey()};
    } else {
        $parent = $relation->{$this->_primary[1]};
    }
    $row->{$this->getParentKey()} = $parent;

    if ($this->_multiRoot) {
        $row->{$this->getRootkey()} = $relation->{$this->getRootkey()};
    }

    $lftKey = $this->getLeftKey();
    $rgtKey = $this->getRightKey();

    switch ((string)$position) {
        case self::FIRST_CHILD:
            $row->$lftKey = $relation->$lftKey + 1;
            $row->$rgtKey = $relation->$lftKey + 2;
            break;
        case self::LAST_CHILD:
            $row->$lftKey = $relation->$rgtKey;
            $row->$rgtKey = $relation->$rgtKey + 1;
            break;
        case self::PREVIOUS_SIBLING:
            $row->$lftKey = $relation->$lftKey;
            $row->$rgtKey = $relation->$lftKey + 1;
            break;
        case self::NEXT_SIBLING:
            $row->$lftKey = $relation->$rgtKey + 1;
            $row->$rgtKey = $relation->$rgtKey + 2;
            break;
     default:
        break;
    }

    try {
        $this->_db->beginTransaction();

        $rootId = null;
```

```
        if ($this->_multiRoot) {
            $rootId = $relation->{$this->getRootkey()};
        }
        $this->applyDeltaShift(2, $row->$lftKey, $rootId);

        $row->save();
        $this->_db->commit();
    } catch (Exception $e) {
        $this->_db->rollBack();
        throw new Zend_Db_NestedSet_Exception($e->getMessage());
    }

    return $row;
}

/**
 * Move a branch of the tree to the specified location
 *
 * @param Zend_Db_TreeNodeInterface $branchOrigin
 * @param Zend_Db_TreeNodeInterface $branchDestination
 * @return array $newBranchValues
 */
public function moveNode(Zend_Db_TreeNodeInterface $origin,
                         Zend_Db_TreeNodeInterface $destination,
                         $position = null )
{
    if(is_null($position)) {
        $position = self::LAST_CHILD;
    }

    $lftKey = $this->getLeftKey();
    $rgtKey = $this->getRightKey();
    $rootKey = $this->getRootkey();

    if($this->_multiRoot &&
       !($origin->$rootKey == $destination->$rootKey))
    {
        throw new Zend_Db_NestedSet_Exception(
            'Both origin and destination must be in the same tree.'
        );
    }

    $updateParent = false;
    switch ((string)$position) {
        case self::FIRST_CHILD:
            $nodeDestination = $destination->$lftKey + 1;
            $updateParent = true;
            break;
        case self::LAST_CHILD:
            $nodeDestination = $destination->$rgtKey;
            $updateParent = true;
            break;
        case self::PREVIOUS_SIBLING:
            $nodeDestination = $destination->$lftKey;
            break;
        case self::NEXT_SIBLING:
        default:
            $nodeDestination = $destination->$rgtKey + 1;
            break;
    }

    $branchWidth = ($origin->$rgtKey - $origin->$lftKey) + 1;

    $this->_db->beginTransaction();

    try {
        $rootId = null;
        if ($this->_multiRoot) {
            $rootId = $origin->$rootKey;
        }
```

```
        $this->applyDeltaShift($branchWidth, $nodeDestination, $rootId);

        if ( $origin->$lftKey >= $nodeDestination ) {
            $origin->$lftKey += $branchWidth;
            $origin->$rgtKey += $branchWidth;
        }

        $range =  $nodeDestination - $origin->$lftKey;
        $this->applyDeltaRange($origin, $range, $rootId);

        $this->applyDeltaShift(-$branchWidth, ($origin->$rgtKey + 1), $rootId);

        if($updateParent) {
            $parent = $destination->{$this->_primary[1]};
        } else {
            $parent = $destination->{$this->getParentKey()};
        }

        $this->update(array(
                $this->getParentKey() => $parent
            ),
            $this->_primary[1] . '=' . $origin->{$this->_primary[1]}
        );

        $this->_db->commit();

    } catch (Exception $e) {
        $this->_db->rollBack();
        throw new Zend_Db_NestedSet_Exception($e->getMessage());
    }


    $newBranchValues = array('left'  => $origin->$lftKey + $range,
                             'right' => $origin->$rgtKey + $range);

    if ($origin->$lftKey <= $nodeDestination) {
        $newBranchValues['left'] -= $branchWidth;
        $newBranchValues['right']-= $branchWidth;
    }

    return $newBranchValues;
}

/**
 * Delete a node (and all its children!)
 *
 * @param Zend_Db_Table_Row $node
 * @return int $rowsDeleted
 * @todo add option to move children to nodes position
 */
public function deleteNode(Zend_Db_TreeNodeInterface $node) {

    $lftKey = $this->getLeftKey();
    $rgtKey = $this->getRightKey();

    try {
        $this->_db->beginTransaction();
        $rowsDeleted = $this->delete(array(
            $this->_db->quoteInto($lftKey . '>=?', $node->$lftKey),
            $this->_db->quoteInto($rgtKey . '<=?', $node->$rgtKey))
        );
        $this->applyDeltaShift(
            ($node->$lftKey - $node->$rgtKey - 1),
            $node->$rgtKey + 1
        );
        $this->_db->commit();
    } catch (Exception $e) {
        $this->_db->rollBack();
        throw new Zend_Db_NestedSet_Exception($e->getMessage());
    }
```

```
        return $rowsDeleted;
    }

    /**
     * Fetch a node via a select object
     *
     * @param Zend_Db_Table_Select $select
     * @return Zend_Db_table_Row $row
     */
    public function fetchNode(Zend_Db_Table_Select $select, $rootId = null) {
        if($this->_multiRoot && !$rootId) {
            throw new Zend_Db_NestedSet_Exception('You must specify a rootId.');
        }
        return $this->fetchRow($select);
    }

    /**
     * Return a branch, starting at $node, ordered on left value, depth first
     *
     * @param mixed Zend_Db_Table_Select|Zend_Db_TreeNodeInterface $node
     * @return Zend_Db_TreeNodeInterface $rowset
     */
    public function fetchBranch($node = null, $rootId = null)
    {
        if($this->_multiRoot && !$rootId) {
            throw new Zend_Db_NestedSet_Exception('You must specify a rootId.');
        }
        if($node instanceof Zend_Db_Table_Select) {
            $node = $this->fetchNode($node);
        }

        $lftKey = $this->getLeftKey();
        $rgtKey = $this->getRightKey();

        if (null == $node) {
            $nodeSelect = $this->select()->where($lftKey . '=?', 1);
            if ($this->_multiRoot) {
                $nodeSelect->where($this->getRootkey() . '=?', $rootId);
            }
            $node = $this->fetchRow($nodeSelect);
        }

        $select = $this->select()
                        ->where($lftKey . '>=?', $node->$lftKey)
                        ->where($rgtKey . '<=?', $node->$rgtKey)
                        ->order($lftKey . ' ASC');

        if($this->_multiRoot) {
            $select->where($this->getRootkey() . '=?', $rootId);
        }

        return $this->fetchAll($select);
    }

    /**
     * Return the full tree
     *
     * @param int $rootId
     * @return Zend_Db_TreeBranchInterface
     */
    public function getTree($rootId = null)
    {
        return $this->fetchBranch(null, $rootId);
    }

    /**
     * Add a child node to the 'last child' position
     *
     * @param Zend_Db_NodeInterface $destination
```

```
     * @param Zend_Db_NodeInterface $origin
     * @return int $id of new node
     */
    public function addChild(Zend_Db_TreeNodeInterface $parent,
                             array $data,
                             $position = self::LAST_CHILD)
    {
        return $this->addNode($parent, $data, $position);
    }

    /**
     * Add a sibling node, 'next' position by default
     *
     * @param Zend_Db_NodeInterface $relation
     * @param array $data
     * @return int $id of new node
     */
    public function addSibling(Zend_Db_TreeNodeInterface $relation,
                               array $data,
                               $position = self::NEXT_SIBLING)
    {
```

```
        return $this->addNode($relation, $data, $position);
    }
}
```

Zend_Db_NestedSet_Branch

```
<?php
class Zend_Db_NestedSet_Branch extends Zend_Db_Table_Rowset_Abstract
                               implements Zend_Db_TreeBranchInterface
{

    /**
     * Zend_Db_Tree_NestedSet_NodeIterator
     *
     * @var RecursiveIteratorIterator
     */
    protected $_iterator = null;

    /**
     * Constructor, passes config to parent constructor
     *
     * @param array $config
     */
    public function __construct(array $config)
    {
        parent::__construct($config);
    }

    /**
     * Return the rowset data as a multi-dimensional array
     *
     * @todo fallback to recursion to build the array if no parent column
     * @return array $tree
     */
    public function toMultiArray()
    {
        $tree = array();
        $ref = array();
        $tableInfo = $this->getTable()->info();
        $primary = $tableInfo['primary'][1];
        $parent = $this->getTable()->getParentKey();

        // TODO fallback to recursion to build the array if no parent column
        if(!in_array($parent, $tableInfo['cols'])) {
            throw new RuntimeException(
                'Cannot build mutli-dimensional array. Table has no parent column.'
            );
        }

        foreach ($this->_data as $node) {
                $node['children'] = array();
            if (isset($ref[$node[$parent]])) {
                $ref[$node[$parent]]['children'][$node[$primary]] = $node;
                $ref[$node[$primary]] = & $ref[$node[$parent]]['children'][$node[$primary]];
            } else {
                $tree[$node[$primary]] = $node;
                $ref[$node[$primary]] = & $tree[$node[$primary]];
            }
        }
        return $tree;
    }

    /**
     * Return the rowset as recursive iterator tree
```

```
 *
 * @todo Fallback to recursion to build iterator if no parent column
 * @return RecursiveIteratorIterator
 */
public function toIterator()
{
    $tree = null;
    $ref = array();

    $tableInfo = $this->getTable()->info();
    $primary = $tableInfo['primary'][1];
    $parent = $this->getTable()->getParentKey();

    // TODO fallback to recursion to build the array at this point
    if(!in_array($parent, $tableInfo['cols'])) {
        throw new RuntimeException(
            'Cannot build mutli-dimensional array. Table has no parent column.'
        );
    }

    foreach ($this->_data as $value) {
     if ( isset($ref[$value[$parent]]) ) {
            $node = new $this->_rowClass(
                        array(
                            'table'    => $this->_table,
                            'data'     => $value,
                            'stored'   => $this->_stored,
                            'readOnly' => $this->_readOnly
                        )
                    );
            $ref[$value[$parent]]->addChild($node);
            $ref[$value[$primary]] = $node;
     } else {
            $node = new $this->_rowClass(
                        array(
                            'table'    => $this->_table,
                            'data'     => $value,
                            'stored'   => $this->_stored,
                            'readOnly' => $this->_readOnly
                        )
                    );
           $tree = $node;
           $ref[$value[$primary]] = $node;
     }
    }
    $this->_iterator = new RecursiveIteratorIterator(
                           $tree, RecursiveIteratorIterator::SELF_FIRST
                       );
```

```
        return $this->_iterator;
    }
}
```

Zend_Db_NestedSet_Node

```
<?php
class Zend_Db_NestedSet_Node extends Zend_Db_Table_Row_Abstract
                            implements Zend_Db_TreeNodeInterface,
                                       RecursiveIterator,
                                       Countable
{

    const NODE_PATH_SEPARATOR = ':';

    /**
     * List of child nodes
     *
     * @var Zend_Db_Tree_NestedSet_NodeIterator
     */
 public $_children = null;

 /**
  * The parent node
  *
  * @var Zend_Db_Tree_NodeInterface
  */
 protected $_parent = null;

 /**
  * Setup the node
  *
  * @todo Enable writing to the db from a node object under certain
  *        conditions. We set nodes read only because we cannot have unique
  *        left and right values, manually altering a node could possibly
  *        corrupt a tree.
  * @param array $config
  */
 public function __construct(array $config = array())
 {
  parent::__construct($config);
  if(isset($config['children']) && is_array($config['children'])) {
           $this->addChildren($config['children']);
  } else {
      $this->_children = array();
  }
  $this->setReadOnly(true);
 }

 /**
  * Set the parent node
  *
  * @param Zend_Db_TreeNodeInterface $node
  */
 public function setParent(Zend_Db_TreeNodeInterface $node)
 {
     $this->_parent = $node;
 }

 /**
  * Get the parent node
  *
  * @return Zend_Db_Tree_NodeInterface
  */
```

```
public function getParent()
{
    return $this->_parent;
}

/**
 * Get's the table to which this node is associated with
 *
 * @return Zend_Db_Tree_NestedSet
 * @see Zend_Db_Table_Row_Abstract#getTable()
 */
public function getTable()
{
 return parent::getTable();
}

/**
 * return whether or not this is a leaf node
 *
 * @return bool
 */
   public function isLeaf()
   {
       $lft = $this->_table->getLeftKey();
       $rgt = $this->_table->getRightKey();
       return (($this->$rgt - $this->$lft) == 1) ? true : false;
   }

   /**
    * Return whether or not this is a root node
    *
    * @return bool
    */
   public function isRoot()
   {
    $lft = $this->_table->getLeftKey();
       return ($this->$lft == 1) ? true : false;
   }

   /**
    * Return whether or not this node has children
    *
    * @return bool
    */
   public function hasChildren()
   {
       return $this->count() > 0;
   }

   /**
    * Adds a child table row object
    *
    * @param unknown_type $child
    */
   public function addChild($child)
   {
       if($child instanceof Zend_Db_TreeNodeInterface) {
           $child->setParent($this);
           $this->_children[] = $child;
        } else {
            throw new RuntimeException('Not instance of Zend_Db_Tree_NodeInterface');
        }
   }

   /**
    * Adds a set of table row objects as children
    *
    * @param unknown_type $children
    */
   public function addChildren($children)
```

```
    {
        if(is_array($children)) {
            // its an array, just attach each array member
            foreach ($children as $child) {
                if($child instanceof Zend_Db_TreeNodeInterface) {
                    $this->addChild($child);
                    continue;
                }
                if (is_array($child)) {
                    $child = $this->getTable()->createRow($child);
                    $this->addChild($child);
                } else {
                    throw new RuntimeException('Not array or not instance of Zend_Db_NodeInterface');
                }
            }
        } else {
            throw new RuntimeException('Nodes supplied as unsupported format.');
        }
    }

    /**
     * Get the container of children
     *
     * @return mixed|Zend_Db_Tree_NestedSet_BranchIterator|null
     */
    public function getChildren()
    {
        if(isset($this->_children[$this->key()])) {
            $children = $this->_children[$this->key()];
            return $children;
        }
        return null;
    }

    /**
     * Completes interface
     */
    public function count()
    {
        $count =  count($this->_children);
        return $count;
    }

    /**
     * Completes interface
     */
    public function current()
    {
        return current($this->_children);
    }

    /**
     * Completes interface
     */
    public function key()
    {
        return key($this->_children);
    }

    /**
     * Completes interface
     */
    public function next()
    {
        next($this->_children);
    }

    /**
     * Completes interface
     */
```

```
    public function rewind()
    {
        reset($this->_children);
    }

    /**
     * Completes interface
     */
    public function valid()
    {
        $key = key($this->_children);
        $current = current($this->_children);
        $valid = current($this->_children) !== false;
        return $valid;
    }

    /**
     * Return the path to this node as an array of column values, the column
     * value to return cannot be primary key, left or right identifiers.
     *
     * @param string $column
     * @return Zend_Db_TreeBranchInterface $path collection of path nodes
     */
    public function getPath($column = null)
    {
        $select = $this->_table->select();
        $tableName = $this->_table->info(Zend_Db_Table::NAME);
        $lft = $this->_table->getLeftKey();
        $rgt = $this->_table->getRightKey();
        $primary = $this->_table->info(Zend_Db_Table::PRIMARY);

        if($column == null ||
           $column == $primary[1] ||
           $column == $lft ||
           $column == $rgt)
        {
            throw new Zend_Db_NestedSet_Exception(
                'Column cannot be null, primary, left or right.'
            );
        }
        if (!in_array($column, $this->_table->info(Zend_Db_Table::COLS))) {
            throw new Zend_Db_NestedSet_Exception('Unknown column.');
        }

        $select->from(array('p' => $tableName),
                      array($column))
              ->join(array('n' => $tableName),
                     "n.{$lft} BETWEEN p.{$lft} AND p.{$rgt} " .
                     "AND n.{$column} = {$this->_table->getAdapter()->quote($this->$column)}")
              ->order("p.{$lft}");

        return $this->_table->fetchAll($select);
    }

    /**
     * Reutrns whether the node has descendant nodes
     *
     * @return bool
     */
    public function hasDescendants()
    {
        $lft = $this->_table->getLeftKey();
        $rgt = $this->_table->getRightKey();
        return (($this->$rgt - $this->$lft) > 1) ? true : false;
    }

    /**
     * Return the immediate descendants for this node. As an example, this would
     * be useful for getting all items in the next level of a category or menu.
     *
```

```
 * @todo get immediate descendants only or option to get all?
 * @param string $column used for display/label e.g. categoryName
 * @return Zend_Db_TreeBranchInterface
 */
public function getDescendants($column = null)
{
    $select = $this->_table->select();
    $tableName = $this->_table->info(Zend_Db_Table::NAME);
    $lft = $this->_table->getLeftKey();
    $rgt = $this->_table->getRightKey();
    $parent = $this->_table->getParentKey();
    $primary = $this->_table->info(Zend_Db_Table::PRIMARY);

    if($column == null ||
       $column == $primary[1] ||
       $column == $lft ||
       $column == $rgt)
    {
        throw new Zend_Db_NestedSet_Exception(
            'Column cannot be null, primary, left or right.'
        );
    }
    if (!in_array($column, $this->_table->info(Zend_Db_Table::COLS))) {
        throw new Zend_Db_NestedSet_Exception('Unknown column.');
    }

     $subStr="(SELECT n.{$column}, (COUNT(p.{$column}) - 1) AS depth
               FROM {$tableName} AS n,
               {$tableName} AS p
               WHERE n.{$lft} BETWEEN p.{$lft} AND p.{$rgt}
               AND n.{$column} = {$this->_table->getAdapter()->quote($this->$column)}
               GROUP BY n.{$column}
               ORDER BY n.{$lft})";
     $subSelect = new Zend_Db_Expr($subStr);

     $select->setIntegrityCheck(false);
     $select->from(array('n' => $tableName),
                   array(
                       "n.{$column}",
                       "n.{$primary[1]}",
                       "n.{$lft}",
                       "n.{$rgt}",
                       "n.{$parent}",
                       "(COUNT(p.{$column})-(st.depth+1)) as depth")
                   )
            ->from(array('p' => $tableName),
                   array())
            ->from(array('sp'=> $tableName),
                   array())
            ->from(array('st'=> $subSelect),
                   array())
            ->where("n.{$lft} BETWEEN p.{$lft} AND p.{$rgt}
                     AND n.{$lft} BETWEEN sp.{$lft} AND sp.{$rgt}
                     AND sp.{$column} = st.{$column}
                     AND depth = 1")
            ->group("n.{$column}")
            ->having('depth = 1')
            ->order("n.{$lft}");

     return $this->_table->fetchAll($select);
}

/**
 * Return all the siblings of this node.
 *
 * @return Zend_Db_TreeBranchInterface
 */
public function getSiblings()
{
    $select = $this->_table->select();
```

```
        $parent = $this->_table->getParentKey();
        $select->where($parent . '=?', $this->$parent);

        if ($this->_table->isMultiRoot()) {
            $root = $this->_table->getRootKey();
            $select->where($root . '=?', $this->$root);
        }

        $lft = $this->_table->getLeftKey();
        $select->where($lft . '!=?', $this->$lft)
               ->order($lft);
```

```
        return $this->_table->fetchAll($select);
    }
}
```

]]></ac:plain-text-body></ac:macro>
]]></ac:plain-text-body></ac:macro>