ClosureTable Documentation

Release 3.1.0

Jan Iwanow

February 13, 2015

1 Contents **3**
1.1 Installation 3
1.2 Setup your ClosureTable 3
1.3 Usage 4
1.4 Customization 7

2 Indices and tables **9**

Let me introduce the third version of my package for Laravel 4. It's intended to use when you need to operate hierarchical data in database. The package is an implementation of a well-known database design pattern called Closure Table. The third version includes many bugfixes and improvements including models and migrations generator.

CHAPTER 1

Contents

1.1 Installation

To install the package, put the following in your composer.json:

```
"require": {
        "franzose/closure-table": "dev-master"
}
```

And to `app/config/app.php`:

```
'providers' => array(
        // ...
        'Franzose\ClosureTable\ClosureTableServiceProvider',
),
```

1.2 Setup your ClosureTable

1.2.1 Create models and migrations

For example, let's assume you're working on pages. In version 3, you can just use an `artisan` command to create models and migrations automatically without preparing all the stuff by hand. Open terminal and put the following:

```
php artisan closuretable:make --entity=page
```

All options of the command:

1. `--namespace`, `-ns` *[optional]*: namespace for classes, set by `--entity` and `--closure` options, helps to avoid namespace duplication in those options
2. `--entity`, `-e`: entity class name; if namespaced name is used, then the default closure class name will be prepended with that namespace
3. `--entity-table`, `-et` *[optional]*: entity table name
4. `--closure`, `-c` *[optional]*: closure class name
5. `--closure-table`, `-ct` *[optional]*: closure table name
6. `--models-path`, `-mdl` *[optional]*: custom models path
7. `--migrations-path`, `-mgr` *[optional]*: custom migrations path

That's almost all, folks! The 'dummy' stuff has just been created for you. You will need to add some fields to your entity migration because the created 'dummy' includes just **required** `id`, `parent_id`, `position`, and `real depth` columns:

1. `id` is a regular autoincremented column
2. `parent_id` column is used to simplify immediate ancestor querying and, for example, to simplify building the whole tree
3. `position` column is used widely by the package to make entities sortable
4. `real depth` column is also used to simplify queries and reduce their number

By default, entity's closure table includes the following columns:

1. **Autoincremented identifier**
2. **Ancestor column** points on a parent node
3. **Descendant column** points on a child node
4. **Depth column** shows a node depth in the tree

It is by closure table pattern design, so remember that you must not delete these four columns.

Remember that many things are made customizable, so see *Customization* for more information.

1.3 Usage

1.3.1 Time of coding

Once your models and their database tables are created, at last, you can start actually coding. Here I will show you ClosureTable's specific approaches.

1.3.2 Direct ancestor (parent)

```
$parent = Page::find(15)->getParent();
```

1.3.3 Ancestors

```
$page = Page::find(15);
$ancestors = $page->getAncestors();
$ancestors = $page->getAncestorsWhere('position', '=', 1);
$hasAncestors = $page->hasAncestors();
$ancestorsNumber = $page->countAncestors();
```

1.3.4 Direct descendants (children)

```
$page = Page::find(15);
$children = $page->getChildren();
$hasChildren = $page->hasChildren();
$childrenNumber = $page->countChildren();

$newChild = new Page(array(
        'title' => 'The title',
```

```
        'excerpt' => 'The excerpt',
        'content' => 'The content of a child'
));

$newChild2 = new Page(array(
        'title' => 'The title',
        'excerpt' => 'The excerpt',
        'content' => 'The content of a child'
));

$page->addChild($newChild);

//you can set child position
$page->addChild($newChild, 5);

//you can get the child
$child = $page->addChild($newChild, null, true);

$page->addChildren([$newChild, $newChild2]);

$page->getChildAt(5);
$page->getFirstChild();
$page->getLastChild();
$page->getChildrenRange(0, 2);

$page->removeChild(0);
$page->removeChild(0, true); //force delete
$page->removeChildren(0, 3);
$page->removeChildren(0, 3, true); //force delete
```

1.3.5 Descendants

```
$page = Page::find(15);
$descendants = $page->getDescendants();
$descendants = $page->getDescendantsWhere('position', '=', 1);
$descendantsTree = $page->getDescendantsTree();
$hasDescendants = $page->hasDescendants();
$descendantsNumber = $page->countDescendants();
```

1.3.6 Siblings

```
$page  = Page::find(15);
$first = $page->getFirstSibling(); //or $page->getSiblingAt(0);
$last  = $page->getLastSibling();
$atpos = $page->getSiblingAt(5);

$prevOne = $page->getPrevSibling();
$prevAll = $page->getPrevSiblings();
$hasPrevs = $page->hasPrevSiblings();
$prevsNumber = $page->countPrevSiblings();

$nextOne = $page->getNextSibling();
$nextAll = $page->getNextSiblings();
$hasNext = $page->hasNextSiblings();
$nextNumber = $page->countNextSiblings();
```

```
//in both directions
$hasSiblings = $page->hasSiblings();
$siblingsNumber = $page->countSiblings();

$sibligns = $page->getSiblingsRange(0, 2);

$page->addSibling(new Page);
$page->addSibling(new Page, 3); //third position

//add and get the sibling
$sibling = $page->addSibling(new Page, null, true);

$page->addSiblings([new Page, new Page]);
$page->addSiblings([new Page, new Page], 5); //insert from fifth position
```

1.3.7 Roots (entities that have no ancestors)

```
$roots = Page::getRoots();
$isRoot = Page::find(23)->isRoot();
Page::find(11)->makeRoot();
```

1.3.8 Entire tree

```
$tree = Page::getTree();
$treeByCondition = Page::getTreeWhere('position', '>=', 1);
```

You deal with the collection, thus you can control its items as you usually do. Descendants? They are already loaded.

```
$tree = Page::getTree();
$page = $tree->find(15);
$children = $page->getChildren();
$child = $page->getChildAt(3);
$grandchildren = $page->getChildAt(3)->getChildren(); //and so on
```

1.3.9 Moving

```
$page = Page::find(25);
$page->moveTo(0, Page::find(14));
$page->moveTo(0, 14);
```

1.3.10 Deleting subtree

If you don't use foreign keys for some reason, you can delete subtree manually. This will delete the page and all its descendants:

```
$page = Page::find(34);
$page->deleteSubtree();
$page->deleteSubtree(true); //with subtree ancestor
$page->deleteSubtree(false, true); //without subtree ancestor and force delete
```

1.4 Customization

You can customize the default things in your classes created by the ClosureTable `artisan` command:

1. **Entity table name**: change `protected $table` property
2. **Closure table name**: do the same in your `ClosureTable` (e.g. `PageClosure`)
3. **Entity's ‘‘parent_id‘‘, ‘‘position‘‘, and ‘‘real depth‘‘ column names**: change return values of `getParentIdColumn()`, `getPositionColumn()`, and `getRealDepthColumn()` respectively
4. **Closure table's ‘‘ancestor‘‘, ‘‘descendant‘‘, and ‘‘depth‘‘ columns names**: change return values of `getAncestorColumn()`, `getDescendantColumn()`, and `getDepthColumn()` respectively.

CHAPTER 2

Indices and tables

- *genindex*
- *modindex*
- *search*

C

Customization, 6

I

Installation, 3

S

Setup your ClosureTable, 3

U

Usage, 4