cakephp-upload

Release 4.0.0

WSS Software

Jun 28, 2020

PREFACE

CHAPTER

ONE

WELCOME

CakePHP DataTables is a plugin for the 4th version of CakePHP. The plugin was designed for automatize the the implementation of DataTable jQuery library into CakePHP as easy as possible. As the Cake's old members and users says, **Automagic**. Here you will find almost everything that you will need to use the plugin.

The CakePHP DataTables cookbook is an editable documentation project based on the CakePHP cookbook to be more intuitive. Notice the pencil icon button fixated against the right wall; it will direct you to the GitHub online editor of the active page, allowing you to contribute any additions, deletions, or corrections to the documentation.

1.1 Getting Help

If you're stuck, you can send an email to creator.

1.2 First Steps

Basically if you understand how CakePHP and DataTables library works and understand the basics concepts of CakePHP DataTables, you will able to use the plugin without large problems. It's easy! So it's recommended that you read *CakePHP DataTables at a Glance* and after this the *Installation Guide*.

After you've finished the Installation Guide, you can learn more about specifics items:

- *Baking DataTables*.
- *Configuring a DataTable table*.
- *Using the plugin resources load*.

CHAPTER

TWO

CAKEPHP DATATABLES AT A GLANCE

CakePHP DataTables plugin follows the same principles of CakePHP, it is designed to make common web-development tasks simple, and easy. It was developed to be light, clean code, fast and easy to use.

The goal of this overview is to introduce the general concepts and conventions of the plugin, and also explain how DataTables library params, configurations, functions and others items are implemented in the plugin.

2.1 Conventions Over Configuration

CakePHP DataTables was build between two web giants, CakePHP and DataTables jQuery library. To make the code consistent it follow a basics rules to preserve the coherence on the CakePHP side as well as on the DataTables side. This makes it easier to use the plugin, because it is intuitive. See some examples below.

2.1.1 Plugin defaults

You can configure all the plugin defaults inside your configuration file such as **app.php**. For more info about this, go to *configuration* section.

2.1.2 DataTables library options

All library options has its name preserved in plugin using CamelCase format. When it is on a second or third level on configuration tree such as **language.info**, the set method is **setLanguageInfo($value)**, and all its params respects the format expected by the library, so, if you read the library documentation, you will know how to pass the param in the right way.

2.1.3 DataTables library callbacks

All library callbacks also has its name preserved in plugin using CamelCase format. But in set method it will be prefixed with **callback** word, so, for example, the callback **createdRow** set will be **callbackCreatedRow($bodyOrParams)**. This may sound a little weird when a certain library callback ends with the word **callback**, like **footerCallback**, because it setter will be **callbackFooterCallback($bodyOrParams)**. But this is to become more standardized!

2.2 Plugin Request Cycle

Below you can see the plugin request cycle to render a table. This diagram is a more simplified version of CakePHP Request Cycle.

The plugin request cycle starts with a user requesting a page that contain a table configured to render. At a high level each plugin request goes through the following steps:

- **Request 1 - Rendering the table structure.**
    1. If user pass a *configuration* for `options`, `columns` and/or `query` in `Controller` action, the plugin `Component` will get the original objects, overwrite it and store on session with a unique key for that url.
    2. On `View`, in the place that the user call the table render, the plugin will render a primitive html table structure. **Note:** If exist some custom `options`, `columns` and/or `query`, the plugin will merge it with the default configuration to use in next steps.
    3. If auto resources isn't disabled, the plugin will load all configured library dependencies.
    4. Yet in `View`, if user call some table render, the plugin will generate the jQuery script of all tables that need to have their settings created.
    5. The response will be delivered to user. Every first load, so as well every changes on table filter, will trigger a **Request 2 or subsequent** to load data inside the table.
- **Request 2..n - Requesting table data.**
    1. The request contain all it filters passed by **GET** or **POST**. The `Controller` will use this to get the right data intended by user to render in table. **Note:** If exist some custom `options`, `columns` and/or `query`, the plugin will merge it with the default configuration to use in next steps.
    2. The `View` will apply the user custom render for each column of each row, if it not exists for one or more column, the plugin will try do a auto render for it. This auto render will stay blinking as a warning for user.
    3. The `View` will compile all this information in a json response that the DataTables library will use to feed the table.

CHAPTER

THREE

INSTALLATION

CakePHP DataTables plugins has a few requirements:

- The same minimum environment of CakePHP 4.
- json PHP extension.
- DataTables library and it plugins (if required).
- jQuery 1 or 3.

Tip: You can use the *Local Resources* to load all the DataTables libraries dependencies and jQuery.

Note: The main function of this plugin is create dynamic HTML tables using DataTables library, so, doesn't make sense use it this without a configured data source in your application, because you need data. DataTables plugin will require it and its respective ORM classes. To see more about data source and ORM classes, go to this link.

3.1 Installation steps

Before use the plugin you will need do somethings to install and configure it.

3.1.1 Requiring plugin using composer

You need load the plugin inside your application using composer's `require` command:

```
composer require wsssoftware/cakephp-datatables:^4.0
```

3.1.2 Loading plugin

Load the plugin by adding the following statement in your project's `src/Application.php`:

```
public function bootstrap(): void
{
    parent::bootstrap();
    $this->addPlugin('DataTables');
    //OR
    $this->addPlugin(\DataTables\Plugin::class);
}
```

OR using cake shell in `bin folder`:

```
cake plugin load DataTables
```

3.1.3 Loading helper

Load the helper by adding the following statement in your project's `src/View/AppView.php`:

```
use Cake\View\View;
use DataTables\View\Helper\DataTablesHelper;

/**
 * Application View
 *
 * @property DataTablesHelper DataTables
 */
class AppView extends View
{

    /**
     * Initialization hook method.
     *
     * @return void
     */
    public function initialize(): void
    {
        $this->loadHelper('DataTables.DataTables');
    }
```

3.1.4 Loading component

Tip: This step is not mandatory, but if you need to edit any DataTables configuration inside a controller for a specific action, you will need to load the component.

Load the component by adding the following statement in your project's `src/Controller/AppController.php`:

```
use Cake\Controller\Controller;
use DataTables\Controller\Component\DataTablesComponent;

/**
 * Application Controller
 *
 * @property DataTablesComponent DataTables
 */
class AppController extends Controller
{

    /**
     * Initialization hook method.
     *
     * @return void
     */
```

(continues on next page)

(continued from previous page)

```
    public function initialize(): void
    {
        $this->loadComponent('DataTables.DataTables');
    }
```

3.1.5 Setting the script renderer

You must to call the *View::fetch()* with the script block name passed as parameter to plugin render tables scripts. The plugin use the same script block of *Local Resources*:: that by default is *script*. Is recommended that you call the fetch method above the **</body>** close tag like example below:

```
    ...

    <?= $this->fetch('script') ?>
</body>
</html>
```

Tip: If you want to use the *Local Resources* class to load yours library dependencies files, you must to have the *View::fetch()* with the css block name passed as parameter. You can change the block name, but by default is *css*. Is recommend that you call the fetch method above the **</head>** close tag like example below:

```
...

    <?= $this->fetch('css') ?>
</head>
```

CHAPTER

FOUR

CONFIGURATION

DataTables.columnAutoGeneratedWarning *Default:* **true** - If true the plugin will show each autogenerated column with a blinking yellow color to alert the developer that it was autogenerated.

DataTables.libraries This config subsection is for control as default what `css` and `scripts` from DataTables library and it dependencies will be loaded.

DataTables.libraries.enabled *Default:* **true** - If true, will enable the `css` and `scripts` load, if false, will disable it.

DataTables.libraries.autoload *Default:* **true** - If true, it will load the library plugins automatically when one is used.

DataTables.libraries.cssBlock *Default:* **css** - This config is used by plugin to know in what fetch block the css files tag will be rendered.

DataTables.libraries.scriptBlock *Default:* **script** - This config is used by plugin to know in what fetch block the script files tag will be rendered. The plugin script code for each table will be rendered inside this block too.

DataTables.libraries.theme *Default:* **LocalResourcesConfig::THEME_BASE** - This configuration will decide what theme files must be loaded. The options are:

- **LocalResourcesConfig::THEME_BASE** - With this option the default DataTables library theme will be loaded.
- **LocalResourcesConfig::THEME_BOOTSTRAP3** - With this option the Bootstrap 3 theme will be loaded.
- **LocalResourcesConfig::THEME_BOOTSTRAP4** - With this option the Bootstrap 4 theme will be loaded.
- **LocalResourcesConfig::THEME_FOUNDATION** - With this option the Foundation theme will be loaded.
- **LocalResourcesConfig::THEME_JQUERY_UI** - With this option the jQuery UI theme will be loaded.
- **LocalResourcesConfig::THEME_SEMANTIC_UI** - With this option the Semantic UI theme will be loaded.

DataTables.libraries.loadThemeLibrary *Default:* **false** - If this config is true, the plugin will load the theme library if it exists, like Bootstrap 3 *css* and *script* files.

DataTables.libraries.jquery *Default:* **LocalResourcesConfig::JQUERY_NONE** - With this config you can choose what version of *jQuery* load or don't load it. The options are:

- **LocalResourcesConfig::JQUERY_NONE** - With this option no jQuery file will be loaded.
- **LocalResourcesConfig::JQUERY_1** - With this option the jQuery 1 will be loaded.
- **LocalResourcesConfig::JQUERY_3** - With this option the jQuery 3 will be loaded.

DataTables.libraries.plugins.autoFill *Default:* **false** - If true will load the plugin *Auto Fill* files.

DataTables.libraries.plugins.buttons *Default:* **false** - If true will load the plugin *Buttons* files.

DataTables.libraries.plugins.colReorder *Default:* **false** - If true will load the plugin *Col Reorder* files.

DataTables.libraries.plugins.fixedColumns *Default:* **false** - If true will load the plugin *Fixed Columns* files.

DataTables.libraries.plugins.fixedHeader *Default:* **false** - If true will load the plugin *Fixed Header* files.

DataTables.libraries.plugins.keyTable *Default:* **false** - If true will load the plugin *Key Table* files.

DataTables.libraries.plugins.responsive *Default:* **false** - If true will load the plugin *Responsive* files.

DataTables.libraries.plugins.rowGroup *Default:* **false** - If true will load the plugin *Row Group* files.

DataTables.libraries.plugins.rowReorder *Default:* **false** - If true will load the plugin *Row Reorder* files.

DataTables.libraries.plugins.scroller *Default:* **false** - If true will load the plugin *Scroller* files.

DataTables.libraries.plugins.searchPanes *Default:* **false** - If true will load the plugin *Search Panes* files.

DataTables.libraries.plugins.select *Default:* **false** - If true will load the plugin *Select* files.

DataTables.StorageEngine.class *Default:* **CacheStorageEngine::class** - This says to plugin what class will do the cache. Must be a implementation of *StorageEngineInterface*.

DataTables.StorageEngine.forceCache *Default:* **false** - If true will force the cache even when debug is on.

DataTables.resources.templates *Default:* **ROOT . DS . 'templates' . DS . 'data_tables' . DS** - This is the folder that the plugin will save baked template files and where the plugin will find it when necessary.

DataTables.resources.twigCacheFolder *Default:* **CACHE . DS . 'data_tables' . DS . 'twig' . DS** - The plugin use *Twig* to generate script callback functions, so it need a cache folder.

DataTables.Cache._data_tables_config_bundles_ A CakePHP cache config that plugin use to store some things and *CacheStorageEngine* is included.

CHAPTER

FIVE

TABLES

We need to do some things to the table work like a charm and practically automatic, so on section learning more, you can see some topics that will teach you how use it. It's easy, don't be afraid!

5.1 Learning more

5.1.1 DataTables

class DataTables\Table**DataTables**

Classes inherited from **DataTables** are the classes that has two methods very important that is called to apply the application business rules to a *ConfigBundle* from a table. They are saved on `src/DataTables/` folder and are postfixed with *DataTables*, so, Categories DataTables class will be named *CategoriesDataTables*. You can easily *bake* this class using the CakePHP bake shell.

5.1.1.1 Methods

There are two methods, the *config* and *rowRenderer*, one to set configs and other to define how data will be rendered.

Config

DataTables\Table\DataTables**::config**(*ConfigBundle $configBundle*)

This method has a *ConfigBundle* passed as param. Inside it you will be able to set the columns, DataTables library options and special Query conditions for the table. Its structure is:

```
/**
 * Will implements all the table configuration.
 *
 * @param \DataTables\Table\ConfigBundle $configBundle
 * @return void
 */
public function config(ConfigBundle $configBundle) {
}
```

Row Renderer

DataTables\Table\DataTables::**rowRenderer**(*DataTablesView $appView*, *EntityInterface $entity*, *Renderer $renderer*)

This method has a *DataTablesView*, *Entity* and *Renderer* objects passed as param. Inside it the developer will be able to define how each row column will be rendered. DataTablesView is inheritance from *AppView* and has access to all helpers and *View* methods. The entity has all needle dada, the developer can even do *if* and others conditionals as required by business rule. Renderer is a object that will storage each column value set by the developer. Each row render will call this method and the non set columns will be autogenerated with a warning. Its structure is:

```
/**
 * @param \DataTables\View\DataTablesView $appView
 * @param \Cake\Datasource\EntityInterface $entity
 * @param \DataTables\Table\Renderer $renderer
 * @return void
 */
public function rowRenderer(DataTablesView $appView, EntityInterface $entity,␣
↪Renderer $renderer) {
}
```

5.1.2 Baking DataTables Classes

The plugin provide the bake command to create the *DataTables* class files that you will need to create and customize your table. As default the class name will be the same as database table name, but you can change this passing the option *–table* in command to inform what is the table name used in that DataTables class.

If the table and the class have the same name you must enter:

```
cake bake data_tables config Categories
```

Where in sample *Categories* is the database table name. This will result in a *CategoriesDataTables* file that will be saved on `src/DataTables/CategoriesDataTables.php` and it datasource is the *categories* table.

If you want want use a different table you must enter:

```
cake bake data_tables config MyCategories --table=Categories
```

Where in sample *MyCategories* is the html table name. This will result in a *MyCategoriesDataTables* file that will be saved on `src/DataTables/MyCategoriesDataTables.php` and it datasource is the *categories* table. Inside the class you will see a attribute that will tell the plugin that it have a different database table name from class name:

```
/**
 * Class MyCategoriesDataTables
 */
class MyCategoriesDataTables extends DataTables {


protected $_ormTableName = 'Categories';
```

5.1.3 ConfigBundle

class DataTables\Table**ConfigBundle**(*string $checkMd5*, *string $dataTablesFQN*)

ConfigBundle is a class that store a table configuration. It will be used to draw the table html, to create the JavaScript, and to get the table data.

5.1.3.1 What is inside

Inside this object, you will be able to access a *Columns* attribute that is a instance of `DataTables\Table\Columns`. With it, is possible manage the columns from your table. Other is *Options* that is a instance of `DataTables\Table\Option\MainOption`. With it, you can use almost all configurations and options from DataTables library. Finally we have the *Query* that is a instance of `DataTables\Table\QueryBaseState`. It is a object that allow you have control over the table query, such as select some more columns from database table or do a *where* conditions for all the DataTables config.

5.1.3.2 How it works

Every time that you ask for a rendering of a table, the plugin will create a instance of this class using as base one DataTables class that you pass as parameter. After constructed the instance, the plugin will call the method config of the selected DataTables class a this will apply the application business rules to the current *ConfigBundle*. When debug isn't on, they will do this long process only on first request, because after get the final result of instance the plugin will save it on cache. In the next time, if cache exists and it is valid yet, the cache will be read and will return a *ConfigBundle* that was generated previously. The plugin uses md5 to check plugin version and class content, so if the developer do some change on the class, or the plugin is updated, the cache is automatic invalidated.

5.1.3.3 Learning more

Columns

class DataTables\Table**Columns**(*ConfigBundle $configBundle*)

Inside the ConfigBundle we have a Columns object from class `DataTables\Table\Columns`. Its function is manage the columns from table, so, you can add, edit, remove, list and other methods related with columns.

Basic Usage

In short, you can use two kind of column, database and non database columns. Non database columns are used to some custom business rules based in the entity data or in a simple case, make a action column. The database column in general will show the data for a specific item from database.

Adding database column

DataTables\Table\Columns**::addDatabaseColumn**(*string $dataBaseField*, *?int $index = null*)

This method will add a new database column to the table, but first it will check if the column exists in table schema, both for the table itself and for its relations. Its name must be unique for the table. As in CakePHP, the column name words must be separated by underscore and the table name (optional) in CamelCase format, both separated by a dot. This method will return a **Column** object that you can use to set some configurations for the created column. You can pass the index to put the column in a specific position. Example of usage:

```php
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
   $configBundle->Columns->addDatabaseColumn('id');

   $configBundle->Columns->addDatabaseColumn('created');
   $configBundle->Columns->addDatabaseColumn('modified');

   // Adding a column BelongsTo
   $configBundle->Columns->addDatabaseColumn('Categories.name');

   // Adding a column HasMany
   $configBundle->Columns->addDatabaseColumn('Tags.name');

}
```

Note: On each column added, the plugin automatically will select the field in query and will put contain associations if needed, so, the query will be build automatically according to your configuration. **BelongsTo** and **HasMany** associations will follow the same logical from CakePHP ou result entity.

Adding a custom database column

DataTables\Table\Columns::**addCustomDatabaseColumn**(*FunctionExpression $functionExpression*, *string $asName*, *?int $index = null*)

With this method, you will able of do some custom sql finds like *CONCAT*, *SUM* and others. You must provide a *FunctionExpression* instance that is the result of one of many methods that you can find in *FunctionsBuilder* that you can get in *$configBundle->Columns->func()* method. Example of usage:

```php
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $func = $configBundle->Columns->func()->concat(['id' => 'identifier', ' created␣
↪in: ', 'created' => 'identifier']);
    $configBundle->Columns->addCustomDatabaseColumn($func, 'custom_field');

}
```

Note: When you use this method with joined tables you will need to join it manually with the *$configBundle->Query* object. With this *DataTablesTableQueryBaseState* object, you will can join the new table with a contain method or others joins methods.

Adding non database column

DataTables\Table\Columns::**addNonDatabaseColumn** (*string $label*, *?int $index = null*)

This method will add a new non database column to the table. Its name must be a alphanumeric string and unique for the the table. You can use this method to create columns with custom values as a sum or concatenate of two database fields or a action column. This method will return a **Column** object that you can use to set some configurations for the created column. You can pass the index to put the column in a specific position. Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
   $configBundle->Columns->addNonDatabaseColumn('total');
   $configBundle->Columns->addNonDatabaseColumn('action');
}
```

Note: When you use the a non database column, its **searchable** and **orderable** options will be disabled automatically.

Changing the index of a column

DataTables\Table\Columns::**changeColumnIndex** (*string $columnName*, *int $index*)

By default, the column index will follow the created order, if you want change the order, you can use this method in your *DataTables* class or in controller using the *DataTables Component*. Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Columns->addDatabaseColumn('name');
    $configBundle->Columns->addDatabaseColumn('created');
    $configBundle->Columns->addDatabaseColumn('modified');
    $configBundle->Columns->addNonDatabaseColumn('action');
    $configBundle->Columns->addDatabaseColumn('id');

    $configBundle->Columns->changeColumnIndex('id', 0);
}
```

Or:

```
/**
 * A example of controller action.
 */
public function controllerAction()
{
    $columns = $this->DataTables->getColumns('Products')
    $columns->changeColumnIndex('id', 0);
}
```

Getting created columns

DataTables\Table\Columns::**getColumns**()

If you need get all configured columns, you can call this method, it will return a array will all **column** objects. Example of usage:

```
/**
 * A example of controller action.
 */
public function controllerAction()
{
    $columns = $this->DataTables->getColumns('Products')

    $columnTitles = [];
    foreach ($columns->getColumns() as $column) {
        $columnTitles[] = $column->getTitle();
    }
}
```

Getting a column

DataTables\Table\Columns::**getColumn**(*string $columnName*)

If you need get a specific configured column, you can call this method, it will return a **column** object for the requested column name. Example of usage:

```
/**
 * A example of controller action.
 */
public function controllerAction()
{
    $columns = $this->DataTables->getColumns('Products')

    $columnTitle = $columns->getColumn('id')->getTitle();
}
```

Getting a column by index

DataTables\Table\Columns::**getColumnByIndex**(*int $index*)

If you need get a specific configured column, you can call this method, it will return a **column** object for the requested column index. Example of usage:

```
/**
 * A example of controller action.
 */
public function controllerAction()
{
    $columns = $this->DataTables->getColumns('Products')

    $columnTitle = $columns->getColumnByIndex(0)->getTitle();
}
```

Getting a column index by name

DataTables\Table\Columns::**getColumnIndexByName**(*string $columnName*)

If you need know the index of a specific column, you can provide the column name to this method and it will return the index. Example of usage:

```
/**
 * A example of controller action.
 */
public function controllerAction()
{
    $columns = $this->DataTables->getColumns('Products')

    $columnIndex = $columns->getColumnIndexByName('Products.created');
}
```

Getting a column name by index

DataTables\Table\Columns::**getColumnNameByIndex**(*int $index*)

If you need to know what the column name for a specific index, you can call this method passing a index number for it. Example of usage:

```
/**
 * A example of controller action.
 */
public function controllerAction()
{
    $columns = $this->DataTables->getColumns('Products')

    $columnName = $columns->getColumnNameByIndex(0);
}
```

Deleting a column

DataTables\Table\Columns::**deleteColumn**(*string $columnName*)

If you need remove a specific column from table, you can call this method passing a column name for it. Example of usage:

```
/**
 * A example of controller action.
 */
public function controllerAction()
{
    $columns = $this->DataTables->getColumns('Products')

    $columns->deleteColumn('action')
}
```

Deleting all columns

DataTables\Table\Columns::**deleteAllColumns**()

If you want remove all created columns, you can call this method. Example of usage:

```
/**
 * A example of controller action.
 */
public function controllerAction()
{
    $columns = $this->DataTables->getColumns('Products')

    $columns->deleteAllColumns()
    $columns->Columns->addDatabaseColumn('id');
    $columns->Columns->addDatabaseColumn('name');
    $columns->Columns->addDatabaseColumn('created');
    $columns->Columns->addDatabaseColumn('modified');
    $columns->Columns->addNonDatabaseColumn('action');
}
```

Learning more

Column

class DataTables\Table**Column**(*string $name*, *bool $database = true*, *array $columnSchema = []*, *string $associationPath = ''*)

After create a column with respective methods, a object column will be storage in columns array and returned in the method to be configured.

Basic Usage

We have some basics configuration about the column and its behavior. In this section you can discovery how to customize and configure your column after create it.

Getting the column name

DataTables\Table\Column::**getName**()

With this method you can get the column name if needed. Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $column = $configBundle->Columns->addDatabaseColumn('id');

    // Will return 'ModelName.id'
    $columnName = $column->getName();

    $column = $configBundle->Columns->addNonDatabaseColumn('action');
```

(continues on next page)

(continued from previous page)

```
    // Will return 'action'
    $columnName = $column->getName();
}
```

Getting the column name

`DataTables\Table\Column::getName()`

With this method you can get the column name if needed. Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $column = $configBundle->Columns->addDatabaseColumn('id');

    // Will return 'ModelName.id'
    $columnName = $column->getName();

    $column = $configBundle->Columns->addNonDatabaseColumn('action');

    // Will return 'action'
    $columnName = $column->getName();
}
```

Checking if is database column

`DataTables\Table\Column::isDatabase()`

This method will return true if the column is a database column. Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $columns = $configBundle->Columns->getColumns();

    $databaseCount = 0;
    foreach ($columns as $column) {
        if ($column->isDatabase()) {
            $databaseCount++;
        }
    }
}
```

Getting column schema

DataTables\Table\Column::**getColumnSchema**(*string $name = null*)

If the field is database, this method will return the column schema. Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $columnSchema = $configBundle->Columns->addDatabaseColumn('id')->
↪getColumnSchema();

    if ($columnSchema['type'] === 'integer') {
        // do something.
    }
}
```

Cell type

Change the cell type created for the column - either TD cells or TH cells.

This can be useful as TH cells have semantic meaning in the table body, allowing them to act as a header for a row (you may wish to add scope='row' to the TH elements using columns.createdCell option).

Source: DataTables library: columns.cellType.

Set method

DataTables\Table\Column::**setCellType**(*?string $cellType*)

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Columns->addDatabaseColumn('id')->setCellType('th');
}
```

Get method

DataTables\Table\Column::**getCellType**()

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $columns = $configBundle->Columns->getColumns();
```

(continues on next page)

(continued from previous page)

```
    $databaseCount = 0;
    foreach ($columns as $column) {
        if ($column->getCellType() === 'th') {
            // do something.
        }
    }
}
```

Class name

Quite simply this option adds a class to each cell in a column, regardless of if the table source is from DOM, Javascript or Ajax. This can be useful for styling columns.

Source: DataTables library: columns.className.

Set method

`DataTables\Table\Column::`**`setClassName`**`(`*?string $className*`)`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Columns->addDatabaseColumn('id')->setClassName('full-width');
}
```

Get method

`DataTables\Table\Column::`**`getClassName`**`()`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $columns = $configBundle->Columns->getColumns();

    $databaseCount = 0;
    foreach ($columns as $column) {
        if ($column->getClassName() === 'full-width') {
            // do something.
        }
    }
}
```

Content padding

The first thing to say about this property is that generally you shouldn't need this!

Having said that, it can be useful on rare occasions. When DataTables calculates the column widths to assign to each column, it finds the longest string in each column and then constructs a temporary table and reads the widths from that. The problem with this is that "mmm" is much wider then "iiii", but the latter is a longer string - thus the calculation can go wrong (doing it properly and putting it into an DOM object and measuring that is horribly slow!). Thus as a "work around" we provide this option. It will append its value to the text that is found to be the longest string for the column - i.e. padding.

Source: DataTables library: columns.contentPadding.

Set method

`DataTables\Table\Column::`**`setContentPadding`**`(`*`?string $contentPadding`*`)`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Columns->addDatabaseColumn('id')->setContentPadding(
↪'aaaaaaaaaaaaaaaaaaaa');
}
```

Get method

`DataTables\Table\Column::`**`getContentPadding`**`()`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $columns = $configBundle->Columns->getColumns();

    $databaseCount = 0;
    foreach ($columns as $column) {
        if (strlen($column->getContentPadding()) >= 15) {
            // do something.
        }
    }
}
```

Content padding

The first thing to say about this property is that generally you shouldn't need this!

Having said that, it can be useful on rare occasions. When DataTables calculates the column widths to assign to each column, it finds the longest string in each column and then constructs a temporary table and reads the widths from that. The problem with this is that "mmm" is much wider then "iiii", but the latter is a longer string - thus the calculation can go wrong (doing it properly and putting it into an DOM object and measuring that is horribly slow!). Thus as a "work around" we provide this option. It will append its value to the text that is found to be the longest string for the column - i.e. padding.

Source: DataTables library: columns.contentPadding.

Set method

`DataTables\Table\Column::`**`setContentPadding`**`(`*`?string $contentPadding`*`)`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Columns->addDatabaseColumn('id')->setContentPadding(
↪'aaaaaaaaaaaaaaaaaaaa');
}
```

Get method

`DataTables\Table\Column::`**`getContentPadding`**`()`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $columns = $configBundle->Columns->getColumns();

    $databaseCount = 0;
    foreach ($columns as $column) {
        if (strlen($column->getContentPadding()) >= 15) {
            // do something.
        }
    }
}
```

Callback Created Cell

This is a callback function that is executed whenever a cell is created (Ajax source, etc) or read from a DOM source. It can be used as a complement to columns.render allowing modification of the cell's DOM element (add background colour for example) when the element is created (cells may not be immediately created on table initialisation if deferRender is enabled, or if rows are dynamically added using the API (rows.add()).

This is the counterpart callback for rows, which use the createdRow option.

Source: DataTables library: columns.createdCell.

Learning more: *Understanding plugin callbacks*.

Set method

`DataTables\Table\Column::`**`callbackCreatedCell`**`(`*$bodyOrParams = []*`)`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Columns->addDatabaseColumn('id')->callbackCreatedCell('alert(
↪"created");');
}
```

Order Data

Allows a column's sorting to take either the data from a different (often hidden) column as the data to sort, or data from multiple columns.

A common example of this is a table which contains first and last name columns next to each other, it is intuitive that they would be linked together to multi-column sort. Another example, with a single column, is the case where the data shown to the end user is not directly sortable itself (a column with images in it), but there is some meta data than can be sorted (e.g. file name) - note that orthogonal data is an alternative method that can be used for this.

Source: DataTables library: columns.orderData.

Set method

`DataTables\Table\Column::`**`setOrderData`**`(`*$orderData*`)`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Columns->addDatabaseColumn('id')->setOrderData([0, 1]);
}
```

Get method

`DataTables\Table\Column::`**`getOrderData`**`()`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $columns = $configBundle->Columns->getColumns();

    $databaseCount = 0;
    foreach ($columns as $column) {
        if ($column->getOrderData() === [0, 1]) {
            // do something.
        }
    }
}
```

Order Data Type

DataTables' primary order method (the ordering feature) makes use of data that has been cached in memory rather than reading the data directly from the DOM every time an order is performed for performance reasons (reading from the DOM is inherently slow). However, there are times when you do actually want to read directly from the DOM, acknowledging that there will be a performance hit, for example when you have form elements in the table and the end user can alter the values. This configuration option is provided to allow plug-ins to provide this capability in DataTables.

Please note that there are no columns.orderDataType plug-ins built into DataTables, they must be added separately. See the DataTables sorting plug-ins page for further information.

Source: DataTables library: columns.orderDataType.

Set method

`DataTables\Table\Column::`**`setOrderDataType`**`(`*`?string $orderDataType`*`)`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Columns->addDatabaseColumn('id')->setOrderDataType('dom-checkbox');
}
```

Get method

DataTables\Table\Column::**getOrderDataType**()

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $columns = $configBundle->Columns->getColumns();

    $databaseCount = 0;
    foreach ($columns as $column) {
        if ($column->getOrderDataType() === 'dom-checkbox') {
            // do something.
        }
    }
}
```

Order Sequence

You can control the default ordering direction, and even alter the behaviour of the order handler (i.e. only allow ascending sorting etc) using this parameter.

Source: DataTables library: columns.orderSequence.

Set method

DataTables\Table\Column::**setOrderSequence**(*array $orderSequence = []*)

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Columns->addDatabaseColumn('id')->setOrderSequence(["desc", "asc",
↪"asc"]);
}
```

Get method

DataTables\Table\Column::**getOrderSequence**()

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
```

(continues on next page)

(continued from previous page)

```
    $columns = $configBundle->Columns->getColumns();

    $databaseCount = 0;
    foreach ($columns as $column) {
        if (in_array('asc', $column->getOrderSequence())) {
            // do something.
        }
    }
}
```

Orderable

Using this parameter, you can remove the end user's ability to order upon a column. This might be useful for generated content columns, for example if you have 'Edit' or 'Delete' buttons in the table.

Note that this option only affects the end user's ability to order a column. Developers are still able to order a column using the order option or the order() method if required.

Source: DataTables library: columns.orderable.

Set method

`DataTables\Table\Column::setOrderable(?bool $orderable)`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Columns->addDatabaseColumn('id')->setOrderable(false);
}
```

Checker method

`DataTables\Table\Column::isOrderable()`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $columns = $configBundle->Columns->getColumns();

    $databaseCount = 0;
    foreach ($columns as $column) {
        if ($column->isOrderable()) {
            // do something.
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Searchable

Using this parameter, you can define if DataTables should include this column in the filterable data in the table. You may want to use this option to disable search on generated columns such as 'Edit' and 'Delete' buttons for example.

Source: DataTables library: columns.searchable.

Set method

`DataTables\Table\Column::`**`setSearchable`**`(`*?bool $searchable*`)`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Columns->addDatabaseColumn('id')->setSearchable(false);
}
```

Checker method

`DataTables\Table\Column::`**`isSearchable`**`()`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $columns = $configBundle->Columns->getColumns();

    $databaseCount = 0;
    foreach ($columns as $column) {
        if ($column->isSearchable()) {
            // do something.
        }
    }
}
```

Title

The titles of columns are typically read directly from the DOM (from the cells in the THEAD element), but it can often be useful to either override existing values, or have DataTables actually construct a header with column titles for you (for example if there is not a THEAD element in the table before DataTables is constructed). This option is available to provide that ability.

Please note that when constructing a header, DataTables can only construct a simple header with a single cell for each column. Complex headers with colspan and rowspan attributes must either already be defined in the document, or be constructed using standard DOM / jQuery methods.

Source: DataTables library: columns.title.

Set method

`DataTables\Table\Column::setTitle(string $title)`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Columns->addDatabaseColumn('id')->setTitle(__('Code'));
}
```

Get method

`DataTables\Table\Column::getTitle()`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $columns = $configBundle->Columns->getColumns();

    $databaseCount = 0;
    foreach ($columns as $column) {
        if ($column->getTitle() === 'Id') {
            // do something.
        }
    }
}
```

Type

When operating in client-side processing mode, DataTables can process the data used for the display in each cell in a manner suitable for the action being performed. For example, HTML tags will be removed from the strings used for filter matching, while sort formatting may remove currency symbols to allow currency values to be sorted numerically. The formatting action performed to normalise the data so it can be ordered and searched depends upon the column's type.

Source: DataTables library: columns.type.

Set method

`DataTables\Table\Column::setType(?string $type)`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Columns->setType('id')->setTitle('num');
}
```

Get method

`DataTables\Table\Column::getType()`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $columns = $configBundle->Columns->getColumns();

    $databaseCount = 0;
    foreach ($columns as $column) {
        if ($column->getType() === 'nul') {
            // do something.
        }
    }
}
```

Visible

DataTables and show and hide columns dynamically through use of this option and the column().visible() / columns().visible() methods. This option can be used to get the initial visibility state of the column, with the API methods used to alter that state at a later time.

This can be particularly useful if your table holds a large number of columns and you wish the user to have the ability to control which columns they can see, or you have data in the table that the end user shouldn't see (for example a database ID column).

Source: DataTables library: columns.visible.

Set method

`DataTables\Table\Column::setVisible(?bool $visible)`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Columns->addDatabaseColumn('id')->setVisible(false);
}
```

Checker method

`DataTables\Table\Column::isVisible()`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $columns = $configBundle->Columns->getColumns();

    $databaseCount = 0;
    foreach ($columns as $column) {
        if ($column->isVisible()) {
            // do something.
        }
    }
}
```

Options

class DataTables\Table\Option**MainOption**(*ConfigBundle $configBundle*, *string $url*)

This is the object instance inside the *ConfigBundle* that is responsible for the management of all DataTables library options. Using it, you will be able to customize almost all options available on the library.

The possibilities are many, so, to you understand better how it works and know all the possibilities, go to the *Options* section of *The library in the plugin*.

Learning more

Options

Learning more

Features

trait DataTables\Table\Option\Section**FeaturesOptionTrait**

This is a implementation of section Features of the library documentation. All the methods of this section are contained inside of *FeaturesOptionTrait* and implemented inside of MainOption object that can be accessed on attribute *Options* of *ConfigBundle* instance.

Basic Usage

Here you will learn how to use the methods **get** and **set** for each option of this section.

Auto Width

Enable or disable automatic column width calculation. This can be disabled as an optimisation (it takes a finite amount of time to calculate the widths) if the tables widths are passed in using columns.width.

Default: *true*

Source: DataTables library: autoWidth.

Set method

DataTables\Table\Option\Section\FeaturesOptionTrait::**setAutoWidth**(*bool $autoWidth*)

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Options->setAutoWidth(true);
    // or
    $configBundle->Options->setAutoWidth(false);
}
```

Check method

DataTables\Table\Option\Section\FeaturesOptionTrait::**isAutoWidth**()

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    if ($configBundle->Options->isAutoWidth()) {
        // Do something.
    }
}
```

Defer Render

By default, when DataTables loads data from an Ajax or Javascript data source (ajax and data respectively) it will create all HTML elements needed up-front. When working with large data sets, this operation can take a not-insignificant amount of time, particularly in older browsers such as IE6-8. This option allows DataTables to create the nodes (rows and cells in the table body) only when they are needed for a draw.

As an example to help illustrate this, if you load a data set with 10,000 rows, but a paging display length of only 10 records, rather than create all 10,000 rows, when deferred rendering is enabled, DataTables will create only 10. When the end user then sorts, pages or filters the data the rows needed for the next display will be created automatically. This effectively spreads the load of creating the rows across the life time of the page.

Note that when enabled, it goes without saying that not all nodes will always be available in the table, so when working with API methods such as columns().nodes() you must take this into account. Below shows an example of how to use jQuery delegated events to handle such a situation.

Default: *false*

Source: DataTables library: deferRender.

Set method

DataTables\Table\Option\Section\FeaturesOptionTrait::**setDeferRender**(*bool $deferRender*)

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Options->setDeferRender(true);
    // or
    $configBundle->Options->setDeferRender(false);
}
```

Check method

`DataTables\Table\Option\Section\FeaturesOptionTrait::`**`isDeferRender`**`()`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    if ($configBundle->Options->isDeferRender()) {
        // Do something.
    }
}
```

Info

When this option is enabled, Datatables will show information about the table including information about filtered data if that action is being performed. This option allows that feature to be enabled or disabled.

Note that by default the information display is shown below the table on the left, but this can be controlled using dom and CSS).

Default: *true*

Source: DataTables library: info.

Set method

`DataTables\Table\Option\Section\FeaturesOptionTrait::`**`setInfo`**`(`*bool $info*`)`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Options->setInfo(true);
    // or
    $configBundle->Options->setInfo(false);
}
```

Check method

`DataTables\Table\Option\Section\FeaturesOptionTrait::`**`isInfo`**`()`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
```

(continues on next page)

(continued from previous page)

```
{
    if ($configBundle->Options->isInfo()) {
        // Do something.
    }
}
```

Length Change

When pagination is enabled, this option will control the display of an option for the end user to change the number of records to be shown per page. The options shown in the list are controlled by the lengthMenu configuration option.

Note that by default the control is shown at the top left of the table. That can be controlled using dom and CSS.

If this option is disabled (false) the length change input control is removed - although the page.len() method can still be used if you wish to programmatically change the page size and pageLength can be used to specify the initial page length. Paging itself is not affected.

Additionally, if pagination is disabled using the paging option, this option is automatically disabled since it has no relevance when there is no pagination.

Default: *true*

Source: DataTables library: lengthChange.

Set method

`DataTables\Table\Option\Section\FeaturesOptionTrait::`**`setLengthChange`**`(`*bool $lengthChange*`)`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Options->setLengthChange(true);
    // or
    $configBundle->Options->setLengthChange(false);
}
```

Check method

`DataTables\Table\Option\Section\FeaturesOptionTrait::`**`isLengthChange`**`()`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    if ($configBundle->Options->isLengthChange()) {
```

(continues on next page)

(continued from previous page)

```
        // Do something.
    }
}
```

Ordering

Enable or disable ordering of columns - it is as simple as that! DataTables, by default, allows end users to click on the header cell for each column, ordering the table by the data in that column. The ability to order data can be disabled using this option.

Note that the ability to add or remove sorting of individual columns can be disabled by the columns.orderable option for each column. This parameter is a global option - when disabled, there are no sorting actions applied by DataTables at all.

Default: *true*

Source: DataTables library: ordering.

Set method

`DataTables\Table\Option\Section\FeaturesOptionTrait::`**`setOrdering`**(*bool $ordering*)

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Options->setOrdering(true);
    // or
    $configBundle->Options->setOrdering(false);
}
```

Check method

`DataTables\Table\Option\Section\FeaturesOptionTrait::`**`isOrdering`**()

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    if ($configBundle->Options->isOrdering()) {
        // Do something.
    }
}
```

Paging

DataTables can split the rows in tables into individual pages, which is an efficient method of showing a large number of records in a small space. The end user is provided with controls to request the display of different data as the navigate through the data. This feature is enabled by default, but if you wish to disable it, you may do so with this parameter.

Default: *true*

Source: DataTables library: paging.

Set method

`DataTables\Table\Option\Section\FeaturesOptionTrait::`**`setPaging`**`(`*bool $paging*`)`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Options->setPaging(true);
    // or
    $configBundle->Options->setPaging(false);
}
```

Check method

`DataTables\Table\Option\Section\FeaturesOptionTrait::`**`isPaging`**`()`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    if ($configBundle->Options->isPaging()) {
        // Do something.
    }
}
```

Processing

Enable or disable the display of a 'processing' indicator when the table is being processed (e.g. a sort). This is particularly useful for tables with large amounts of data where it can take a noticeable amount of time to sort the entries.

Default: *false*

Source: DataTables library: processing.

Set method

DataTables\Table\Option\Section\FeaturesOptionTrait::**setProcessing**(*bool $processing*)

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Options->setProcessing(true);
    // or
    $configBundle->Options->setProcessing(false);
}
```

Check method

DataTables\Table\Option\Section\FeaturesOptionTrait::**isProcessing**()

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    if ($configBundle->Options->isProcessing()) {
        // Do something.
    }
}
```

Processing

Enable or disable the display of a 'processing' indicator when the table is being processed (e.g. a sort). This is particularly useful for tables with large amounts of data where it can take a noticeable amount of time to sort the entries.

Default: *false*

Source: DataTables library: processing.

Set method

DataTables\Table\Option\Section\FeaturesOptionTrait::**setProcessing**(*bool $processing*)

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
```

(continues on next page)

(continued from previous page)

```
{
    $configBundle->Options->setProcessing(true);
    // or
    $configBundle->Options->setProcessing(false);
}
```

Check method

`DataTables\Table\Option\Section\FeaturesOptionTrait::`**`isProcessing`**`()`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    if ($configBundle->Options->isProcessing()) {
        // Do something.
    }
}
```

ScrollX

Enable horizontal scrolling. When a table is too wide to fit into a certain layout, or you have a large number of columns in the table, you can enable horizontal (x) scrolling to show the table in a viewport, which can be scrolled.

This property can be true which will allow the table to scroll horizontally when needed (recommended), or any CSS unit, or a number (in which case it will be treated as a pixel measurement).

Default: *false*

Source: DataTables library: scrollX.

Set method

`DataTables\Table\Option\Section\FeaturesOptionTrait::`**`setScrollX`**`(`*bool $scrollX*`)`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Options->setScrollX(true);
    // or
    $configBundle->Options->setScrollX(false);
}
```

Check method

DataTables\Table\Option\Section\FeaturesOptionTrait::**isScrollX**()

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    if ($configBundle->Options->isScrollX()) {
        // Do something.
    }
}
```

ScrollY

Enable vertical scrolling. Vertical scrolling will constrain the DataTable to the given height, and enable scrolling for any data which overflows the current viewport. This can be used as an alternative to paging to display a lot of data in a small area (although paging and scrolling can both be enabled at the same time if desired).

The value given here can be any CSS unit, or a number (in which case it will be treated as a pixel measurement) and is applied to the table body (i.e. it does not take into account the header or footer height directly).

Source: DataTables library: scrollY.

Set method

DataTables\Table\Option\Section\FeaturesOptionTrait::**setScrollY**(*?string $scrollY*)

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Options->setScrollY('200px');
    // or
    $configBundle->Options->setScrollY('200em');
}
```

Get method

DataTables\Table\Option\Section\FeaturesOptionTrait::**getScrollY**()

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
```

(continues on next page)

(continued from previous page)

```
{
    if ($configBundle->Options->getScrollY() === '200px') {
        // Do something.
    }
}
```

Searching

This option allows the search abilities of DataTables to be enabled or disabled. Searching in DataTables is "smart" in that it allows the end user to input multiple words (space separated) and will match a row containing those words, even if not in the order that was specified (this allow matching across multiple columns).

Please be aware that technically the search in DataTables is actually a filter, since it is subtractive, removing data from the data set as the input becomes more complex. It is named "search" here, and else where in the DataTables API for consistency and to ensure there are no conflicts with other methods of a similar name (specific the filter() API method).

Note that if you wish to use the search abilities of DataTables this must remain true - to remove the default search input box whilst retaining searching abilities (for example you might use the search() method), use the dom option.

Default: *true*

Source: DataTables library: searching.

Set method

`DataTables\Table\Option\Section\FeaturesOptionTrait::`**`setSearching`**`(`*bool $searching*`)`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Options->setSearching(true);
    // or
    $configBundle->Options->setSearching(false);
}
```

Check method

`DataTables\Table\Option\Section\FeaturesOptionTrait::`**`isSearching`**`()`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    if ($configBundle->Options->isSearching()) {
```

(continues on next page)

(continued from previous page)

```
        // Do something.
    }
}
```

Server Side

DataTables has two fundamental modes of operation:

- Client-side processing - where filtering, paging and sorting calculations are all performed in the web-browser.
- Server-side processing - where filtering, paging and sorting calculations are all performed by a server.

By default DataTables operates in client-side processing mode, but can be switched to server-side processing mode using this option. Server-side processing is useful when working with large data sets (typically >50‘000 records) as it means a database engine can be used to perform the sorting etc calculations - operations that modern database engines are highly optimised for, allowing use of DataTables with massive data sets (millions of rows).

When operating in server-side processing mode, DataTables will send parameters to the server indicating what data it needs (what page, what filters are applied etc), and also expects certain parameters back in order that it has all the information required to display the table. The client-server communication protocol DataTables uses is detailed in the DataTables documentation.

Fixed in: *true*

Source: DataTables serverSide: serverSide.

Set method

`DataTables\Table\Option\Section\FeaturesOptionTrait::`**`setServerSide`**`(`*bool $serverSide*`)`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Options->setServerSide(true);
    // or
    $configBundle->Options->setServerSide(false);
    // this will throw a exception
}
```

Check method

`DataTables\Table\Option\Section\FeaturesOptionTrait::`**`isServerSide`**`()`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    if ($configBundle->Options->isServerSide()) {
        // Do something.
    }
}
```

State Save

Enable or disable state saving. When enabled aDataTables will store state information such as pagination position, display length, filtering and sorting. When the end user reloads the page the table's state will be altered to match what they had previously set up.

Data storage for the state information in the browser is performed by use of the localStorage or sessionStorage HTML5 APIs. The stateDuration indicated to DataTables which API should be used (localStorage: 0 or greater, or sessionStorage: -1).

To be able to uniquely identify each table's state data, information is stored using a combination of the table's DOM id and the current page's pathname. If the table's id changes, or the page URL changes, the state information will be lost.

Please note that the use of the HTML5 APIs for data storage means that the built in state saving option will not work with IE6/7 as these browsers do not support these APIs. Alternative options of using cookies or saving the state on the server through Ajax can be used through the stateSaveCallback and stateLoadCallback options.

Default: *false*

Source: DataTables serverSide: stateSave.

Set method

`DataTables\Table\Option\Section\FeaturesOptionTrait::`**`setStateSave`**`(`*bool $stateSave*`)`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    $configBundle->Options->setStateSave(true);
    // or
    $configBundle->Options->setStateSave(false);
}
```

Check method

`DataTables\Table\Option\Section\FeaturesOptionTrait::isStateSave()`

Example of usage:

```
/**
 * @param \DataTables\Table\ConfigBundle $configBundle
 */
public function config(ConfigBundle $configBundle): void
{
    if ($configBundle->Options->isStateSave()) {
        // Do something.
    }
}
```

Query

CakePHP has a few system requirements:

Local Resources

5.1.4 Rendering

CakePHP has a few system requirements:

5.1.5 Customizing

CakePHP has a few system requirements:

5.1.6 Understanding callbacks

CHAPTER

SIX

CALLBACKS

PHP NAMESPACE INDEX

d

INDEX

M

R

S