Redactor 10 Docs

How to install

Redactor is a javascript program that helps you to create word-processed texts on the web. Redactor uses the jQuery library.

Redactor requires jQuery 1.9.1 or higher. Redactor also requires a modern browser with full Javascript and HTML5 support. Redactor has been tested in and fully supports following browsers:

- Latest Chrome (desktop and mobile)
- Latest Firefox (desktop only)
- Latest Safari (desktop and mobile)
- Latest Opera (webkit)
- Internet Explorer 11

To install Redactor, place the following code between the `<head></head>` tags:

```
<link rel="stylesheet" href="/js/redactor/redactor.css" />
<script src="/js/redactor/redactor.js"></script>
```

If your Redactor download is placed in a different folder, don't forget to change file's paths.
You can call Redactor using the following code:

```
<script type="text/javascript">
$(function()
{
    $('#content').redactor();
});
</script>
```

This code may also be placed between the `<head></head>` tags or at any other place on the page.

Finally, you need to place a `<textarea>` element with ID "content" (or with the ID that you set during Redactor's call). This element will be replaced by the visual representation of the Redactor.

```
<textarea id="content" name="content"></textarea>
```

Gathering everything:

```
<!DOCTYPE html>
```

```
<html>
    <head>
        <title>Redactor</title>
        <meta charset="utf-8">

        <script src="/js/jquery-2.0.1.min.js"></script>

        <link rel="stylesheet" href="/js/redactor/redactor.css" />
        <script src="/js/redactor/redactor.js"></script>

        <script type="text/javascript">
        $(function()
        {
            $('#content').redactor();
        });
        </script>
    </head>
    <body>
        <textarea id="content" name="content"></textarea>
    </body>
</html>
```

Get and save content

You can save text from Redactor using one of four different approaches.

Sending form data using POST method

```
<script type="text/javascript">
$(function()
{
    $('#content').redactor();
});
</script>

<form action="" method="post">
    <textarea id="content" name="content"></textarea>
    <p>
        <input type="submit" value="Submit" name="send">
    </p>
</form>
```

In this example, text from Redactor will be sent as a form when user clicks on Submit button. On a server side you'll receive a variable POST with a name content.

Form serialize and AJAX

```
<script type="text/javascript">
$(function()
{
    $('#content').redactor();
});
```

```
function sendForm()
{
    $.ajax({
        url: 'savedata.php',
        type: 'post',
        data: $('#formID').serialize()
    });
}
</script>

<form action="" id="formID" method="post">
    <textarea id="content" name="content"></textarea>
    <p>
        <button onclick="sendForm();">Send</button>
    </p>
</form>
```

Via Redactor API method

```
<script type="text/javascript">
$(function()
{
    $('#content').redactor();
});

function sendForm()
{
    $.ajax({
        url: 'savedata.php',
        type: 'post',
        data: 'content=' + $('#content').redactor('code.get')
    });
}
</script>

<form action="" method="post">
    <textarea id="content" name="content"></textarea>
    <p>
        <button onclick="sendForm();">Send</button>
    </p>
</form>
```

Autosave

If you wish the Redactor to automatically save the text every two minutes, just set the path to a script that will handle the incoming data.

```
<script type="text/javascript">
$(function()
{
    $('#content').redactor({
        autosave: '/save.php',
        autosaveInterval: 120, // seconds
```

```
        autosaveCallback: function(json)
        {
            console.log(json);
        }
    });
});
</script>
```

Or autosave data on change:

```
<script type="text/javascript">
$(function()
{
    $('#content').redactor({
        autosave: '/save.php',
        autosaveOnChange: true
    });
});
</script>
```

In this examples, the Redactor will transmit data to the save.php every two minutes (in background mode). Autosave will transmit the $_POST['textareaname'] variable.
You can use autosaveErrorCallback for a triggering when request for an autosave has been error.

```
$('#redactor').redactor({
    autosave: '/save.php',
    autosaveInterval: 120, // seconds
    autosaveErrorCallback: function(json)
    {
        console.log(json);
    }
});
```

Your server-side script must pass error in case of an error, for example:

```
<?php
    $array = array(
        'error' => true,
        'message' => 'Something wrong'
    );

    echo stripslashes(json_encode($array));
?>
```

Upload images

Javascript doesn't allow the implementation of images and files uploads. Therefore, this feature utilizes server-side languages, such as PHP (see Python + Django implementation from Patrick Altman).

First of all, make sure that you set a proper path to the upload file. You can do it by setting Redactor's imageUpload option on load, for example:

```
<script type="text/javascript">
$(function()
{
    $('#redactor').redactor({
        imageUpload: '/modules/upload.php'
    });
});
</script>
```

Let's assume that **upload.php** file will handle images. Its code may look like this:

```
<?php

// This is a simplified example, which doesn't cover security of uploaded
images.
// This example just demonstrate the logic behind the process.

// files storage folder
$dir = '/sitecom/images/';

$_FILES['file']['type'] = strtolower($_FILES['file']['type']);

if ($_FILES['file']['type'] == 'image/png'
|| $_FILES['file']['type'] == 'image/jpg'
|| $_FILES['file']['type'] == 'image/gif'
|| $_FILES['file']['type'] == 'image/jpeg'
|| $_FILES['file']['type'] == 'image/pjpeg')
{
    // setting file's mysterious name
    $filename = md5(date('YmdHis')).'.jpg';
    $file = $dir.$filename;

    // copying
    move_uploaded_file($_FILES['file']['tmp_name'], $file);

    // displaying file
    $array = array(
        'filelink' => 'images/'.$filename
    );

    echo stripslashes(json_encode($array));
```

```
}

?>
```

JSON example:

```
{ "filelink": "/images/img.jpg" }
```

This is a simplified example, which doesn't cover the security of any uploaded files. This example just demonstrates the logic behind the process, because for any given Redactor's integration instance file uploads may widely vary.

Upload files

Javascript doesn't allow the implementation of images and files uploads. Therefore, this feature utilizes server-side languages, such as PHP (see Python + Django implementation from Patrick Altman).

First of all, make sure that you have properly set a path to the upload file. You can do this by modifying Redactor's fileUpload option on load, like this:

```
<script type="text/javascript">
$(function()
{
    $('#redactor').redactor({
        fileUpload: '/file-upload.php'
    });
});
</script>
```

file-upload.php will process files, it may look like this:

```
<?php

// This is a simplified example, which doesn't cover security of uploaded
files.
// This example just demonstrate the logic behind the process.

move_uploaded_file($_FILES['file']['tmp_name'],
'/files/'.$_FILES['file']['name']);

$array = array(
    'filelink' => '/files/'.$_FILES['file']['name'],
    'filename' => $_FILES['file']['name']
);

echo stripslashes(json_encode($array));
?>
```

JSON example:

```
{ "filelink": "/files/file.doc", "filename": "Filename" }
```

This is not the exact example, it is intended to give you an overall idea of the process and doesn't take the upload security into account. Also it doesn't reflect all of the nuances of every possible integration type.

Keyboard shortcuts

	OS X	Windows
Undo	`Cmd + z`	`Ctrl + z`
Redo	`Cmd + Shift + z`	`Ctrl + Shift + z`
Increase indent	`Tab`	`Tab`
Decrease indent	`Shift + Tab`	`Shift + Tab`
Clear formatting	`Cmd + Shift + m`	`Ctrl + Shift + m`
Bold	`Cmd + b`	`Ctrl + b`
Italic	`Cmd + i`	`Ctrl + i`
Numbered list	`Ctrl + Shift + 7`	`Ctrl + Shift + 7`
Bulleted list	`Ctrl + Shift + 8`	`Ctrl + Shift + 8`
Insert link	`Cmd + k`	`Ctrl + k`
Superscript	`Cmd + h`	`Ctrl + h`
Subscript	`Cmd + l`	`Ctrl + l`

Toolbar

By default, Redactor has the following set of buttons:

```
['html', 'formatting', 'bold', 'italic', 'deleted',
'unorderedlist', 'orderedlist', 'outdent', 'indent',
'image', 'file', 'link', 'alignment', 'horizontalrule']

// additional buttons
// 'underline'
```

If you need to setup your own set of buttons you can do so using this option:

```
$('#redactor').redactor({
    buttons: ['formatting', 'bold', 'italic']
});
```

Add a button

How to create an image button

Create a button's CSS

```
<style type="text/css">
.redactor-toolbar li a.re-advanced {
    background-image: url(/img/advanced.png);
}
.redactor-toolbar li a.re-advanced:hover {
    background-image: url(/img/advanced-hover.png);
}
</style>
```

The button size is 14x14px (and 28x28px for retina).
Create the plugin:

```
$.Redactor.prototype.advanced = function()
{
    return {
        init: function ()
        {
            var button = this.button.add('advanced', 'Advanced');
            this.button.addCallback(button, this.advanced.testButton);
        },
        testButton: function(buttonName)
        {
            alert(buttonName);
        }
```

```
    };
};
```

Call Redactor with the plugin:

```
$(function()
{
    $('#redactor').redactor({
        plugins: ['advanced']
    });
});
```

How to create a Font Awesome button

First you need to connect Font Awesome on your page:

```
<link rel="stylesheet"
href="//cdnjs.cloudflare.com/ajax/libs/font-awesome/4.0.3/css/font-awesome.
min.css">
```

Create the plugin:

```
$.Redactor.prototype.advanced = function()
{
    return {
        init: function()
        {
            var button = this.button.add('advanced', 'Advanced');

            // make your added button as Font Awesome's icon
            this.button.setAwesome('advanced', 'fa-tasks');

            this.button.addCallback(button, this.advanced.testButton);
        },
        testButton: function(buttonName)
        {
            alert(buttonName);
        }
    };
};
```

Call Redactor with the plugin:

```
$(function()
{
    $('#redactor').redactor({
        plugins: ['advanced']
    });
});
```

Add a dropdown

Connect Font Awesome on your page or create image button:

```
<link rel="stylesheet"
href="//cdnjs.cloudflare.com/ajax/libs/font-awesome/4.0.3/css/font-awesome.
min.css">
```

Create the plugin:

```
$.Redactor.prototype.advanced = function()
{
    return {
        init: function()
        {

            var dropdown = {};

            dropdown.point1 = { title: 'Point 1', func:
this.advanced.pointFirstCallback };
            dropdown.point2 = { title: 'Point 2', func:
this.advanced.pointSecondCallback };

            var button = this.button.add('advanced', 'Advanced');
            this.button.setAwesome('advanced', 'fa-tasks');

            this.button.addDropdown(button, dropdown);
        },
        pointFirstCallback: function(buttonName)
        {
            alert(buttonName);
        },
        pointSecondCallback: function(buttonName)
        {
            alert(buttonName);
        }
    };
};
```

Call Redactor with the plugin:

```
$(function()
{
    $('#redactor').redactor({
        plugins: ['advanced']
    });
});
```

Languages

Usage:

```
<script src="redactor/langs/es.js"></script>
<script type="text/javascript">
$(function()
{
    $('#content').redactor({
        lang: 'es'
    });
});
</script>
```

Creating plugins

Plugins are here to extend Redactor's features and options. It's very easy to build a plugin. We've developed a powerful API and straightforward development process to allow you to create a broad variety of different plugins with limitless possible functionality.
First, create plugin file and name it, let's say, myplugin.js. myplugin will be your plugin's name.
Now let's build a base for your plugin. Put the following code inside this file:

```
$.Redactor.prototype.myplugin = function()
{
    return {
        myMethod: function()
        {
            // your code
        }
    };
};
```

This code allows you to link your plugin with Redactor and use plugin's methods in Redactor.

Now, add a link to your plugin on a page, where you're using Redactor.

```
<!DOCTYPE html>
<html>
<head>
    <title>Plugins are friends, not food!</title>

    <meta charset="utf-8">
```

```
    <link rel="stylesheet" type="text/css" href="css/your-site-style.css"
/>
    <link rel="stylesheet" href="redactor/redactor.css" />

    <script type="text/javascript" src="lib/jquery-2.0.1.min.js"></script>
    <script src="redactor/redactor.js"></script>
    <script src="redactor/myplugin.js"></script>

    <script type="text/javascript">
    $(function()
    {
        $('#content').redactor({
            plugins: ['myplugin']
        });
    });
    </script>

</head>
<body>
    <div id="page">
        <textarea id="content" name="content"></textarea>
    </div>
</body>
</html>
```

You can link as many plugins as you like:

```
<script type="text/javascript">
$(function()
{
    $('#content').redactor({
        plugins: ['myplugin', 'anotherplugin']
    });
});
</script>
```

Most likely, you wish to launch your plugin with Redactor simultaneously. Create 'init' method in your plugin, and this method will launch at the same time as Redactor:

```
$.Redactor.prototype.myplugin = function()
{
    return {
        init: function()
        {
            console.log('Plugin started');
        }
    };
};
```

It is time to create a method in the plugin and see how it interacts with Redactor API. Let’s create ‘show’ method and call it from 'init'.

```
$.Redactor.prototype.myplugin = function()
{
    return {
        init: function()
        {
            this.myplugin.show();
        },
        show: function()
        {
            console.log('myplugin show');
        }
    };
};
```

Now let's say we are creating a button in the toolbar that will perform some action when pressed by a user.

```
$.Redactor.prototype.myplugin = function()
{
    return {
        init: function()
        {
            var button = this.button.add('my-button', 'My Button');
            this.button.addCallback(button, this.myplugin.show);
        },
        show: function()
        {
            console.log('myplugin show');
        }
    };
};
```

This way, a new button will be added to the toolbar and whenever a user clicks on this button, 'show' method will be called. Please note, that for a button to appear in the toolbar, it must have an image or an icon.

Building modals

Create the plugin with modal template:

```
$.Redactor.prototype.advanced = function()
{
    return {
        getTemplate: function()
        {
            return String()
            + '<section id="redactor-modal-advanced">'
            + '<label>Enter a text</label>'
            + '<textarea id="mymodal-textarea" rows="6"></textarea>'
            + '</section>';
        },
```

```
        init: function ()
        {
            var button = this.button.add('advanced', 'Advanced');
            this.button.addCallback(button, this.advanced.show);

            // make your added button as Font Awesome's icon
            this.button.setAwesome('advanced', 'fa-tasks');
        },
        show: function()
        {
            this.modal.addTemplate('advanced',
this.advanced.getTemplate());

            this.modal.load('advanced', 'Advanced Modal', 400);

            this.modal.createCancelButton();

            var button = this.modal.createActionButton('Insert');
            button.on('click', this.advanced.insert);

            this.selection.save();
            this.modal.show();

            $('#mymodal-textarea').focus();
        },
        insert: function()
        {
            var html = $('#mymodal-textarea').val();

            this.modal.close();
            this.selection.restore();

            this.insert.html(html);

            this.code.sync();

        }
    };
};
```

Call Redactor with the plugin:

```
$(function()
{
    $('#redactor').redactor({
        plugins: ['advanced']
    });
});
```

Security

We are taking Redactor's security very seriously (sometimes even too seriously though). It is a tricky task now for those who want to inject malicious code using Redactor's window. However, JavaScript and its environment does not allow us to cover you against 100% of attacks. That's we strongly recommend you to perform a server-side clean-up of a code that you receive from Redactor.

You can perform such clean-up using any server-side programming language. Here're some basic examples on PHP.

First off, send text from Redactor via POST, using form or with AJAX. Also, check if the form contents came from your site. You can do it by checking REFERER like this:

```
function is_referer()
{
    if (!isset($_SERVER['HTTP_REFERER'])) return false;

    $url = parse_url($_SERVER['HTTP_REFERER']);

    if ($url['host'] == 'yoursite.com') return true;
    else return false;
}
```

Clean the tags that are not allowed in Redactor. Here's how:

```
function clear_tags($str)
{
    return strip_tags($str,
'<code><span><div><label><a><br><p><b><i><del><strike><u><img><video><audio
><iframe><object><embed><param><blockquote><mark><cite><small><ul><ol><li><
hr><dl><dt><dd><sup><sub><big><pre><code><figure><figcaption><strong><em><t
able><tr><td><th><tbody><thead><tfoot><h1><h2><h3><h4><h5><h6>');
}
```

And clean attributes of tags with html-clean libraries (in PHP is to use a library like htmLawed or htmlpurifier).

When a user uploads a picture, perform a check if it actually is a picture:

```
function is_image($image_path)
{
    if (!$f = fopen($image_path, 'rb'))
    {
```

```
        return false;
    }

    $data = fread($f, 8);
    fclose($f);

    // signature checking
    $unpacked = unpack("H12", $data);
    if (array_pop($unpacked) == '474946383961' || array_pop($unpacked) ==
'474946383761') return "gif";
    $unpacked = unpack("H4", $data);
    if (array_pop($unpacked) == 'ffd8') return "jpg";
    $unpacked = unpack("H16", $data);
    if (array_pop($unpacked) == '89504e470d0a1a0a') return "png";

    return false;
}
```

API

Initialization

This jQuery plugin initializes Redactor and provides access to API.

Launching Redactor

```
$('#content').redactor();
```

With class or any selector:

```
$('.content-class').redactor();
```

On launch, plugin returns jQuery object or objects (if plugin has been applied to multiple elements on a page)

Redactor can be launched with certain settings. Settings are passed as JavaScript object:

```
$('#content').redactor({
    focus: true,
    initCallback: function()
    {
        console.log(this.code.get());
    }
});
```

External access to API

Although Redactor provides the ability to access API externally, we strongly recommend you to always create a plugin for API access. This makes code nicer, simpler and logically accurate.

For external API access, you can pass a string with function name to the plugin:

```
$('#content').redactor('module.method');
```

All API functions are part of different modules; first, set module's name, and then provide an API function (separated from module's name by a dot.)

For example, to get HTML code from Redactor, let's set 'code' module and 'get' function:

```
var html = $('#content').redactor('code.get');
```

API calls can return data. If plugin has been applied to multiple elements on a page, data will be returned as an array, for example

```
var arrayHtml = $('.content-class').redactor('code.get');
```

When calling an API function, you can also pass an argument to it:

```
$('#content').redactor('code.set', '<p>My html code</p>');
```

Multiple arguments can be passed, if a function accepts multiple arguments.

Variables

`this.$box`	container element, that contains editable area
`this.$editor`	editable layer
`this.$element`	element to which Redactor has been applied
`this.$textarea`	textarea to with Redactor has been applied, or textarea created by Redactor
`this.$toolbar`	toolbar

Alignment

left

Aligns selected text to the left.

```
this.alignment.left();
```

right

Aligns selected text to the right.

```
this.alignment.right();
```

center

Aligns selected text to the center.

```
this.alignment.center();
```

justify

Aligns selected text to the width of a text area.

```
this.alignment.justify();
```

Buffer

This module sends code into Undo/Redo buffer.

If your plugins are performing any operations that may require undo/redo functionality, you need to add code to the buffer first in order for you users to be able to undo/redo plugin actions.

set

This method adds code to the Undo/Redo buffer.

```
this.buffer.set();
```

add

This method adds code to the Undo/Redo buffer, but unlike buffer.set, it doesn't set any temporary markers to preserve cursor position. It means that this method can be used whenever cursor is not in Redactor and there's no need to save its position.

```
this.buffer.add();
```

Button

get

Accessing toolbar button element by key.

```
var button = this.button.get('bold');
```

setAwesome

Sets Font Awesome icon to a button.

```
this.button.setAwesome('my-button', 'fa-cog');
```

add

Adds button at the end of toolbar. Returns button element.

```
var button = this.button.add('my-button', 'My Button');
```

addFirst

Adds button at the beginning of toolbar. Returns button element.

```
var button = this.button.addFirst('my-button', 'My Button');
```

addAfter

Adds button after specified button. Returns button element.

```
var button = this.button.addAfter('bold', 'my-button', 'My Button');
```

addBefore

Adds button before specified button. Return button element.

```
var button = this.button.addBefore('bold', 'my-button', 'My Button');
```

addCallback

Applies callback to added button.

```
var button = this.button.add('my-button', 'My Button');
this.button.addCallback(button, function()
{
    // your code
});
```

addDropdown

Adds dropdown to a button.

```
var button = this.button.add('my-button', 'My Button');

var dropdown = {
    item1: { title: 'Item 1', func: function() { // your code } },
    item2: { title: 'Item 2', func: function() { // your code } }
};

this.button.addDropdown(button, dropdown);
```

remove

Removes specified button from toolbar.

```
this.button.remove('my-button');
```

Block

format

Applies block tag to a selected text.

```
this.block.format('p');
```

CSS class, attribute or data can be set at the same time:

```
this.block.format('p', 'class', 'red');
this.block.format('p', 'attr', { name: 'title', value: 'Hello!' });
this.block.format('p', 'data', { name: 'data-name', value: 'red-block' });
```

toggleData

Sets or removes specified data attribute from block elements within selected text.

```
this.block.toggleData('data-name', 'red-block');
```

setData

Sets specified data attribute to block elements within text selection.

```
this.block.setData('data-name', 'red-block');
```

removeData

Removes specified data attribute to block elements within text selection.

```
this.block.removeData('data-name');
```

toggleAttr

Sets or removes specified attribute from block elements within selected text.

```
this.block.toggleAttr('name', 'red-block');
```

setAttr

Sets specified attribute to block elements within text selection.

```
this.block.setAttr('name', 'red-block');
```

removeAttr

Removes specified attribute to block elements within text selection.

```
this.block.removeAttr('name');
```

toggleClass

Sets or removes specified CSS class from block elements within selected text.

```
this.block.toggleClass('red');
```

setClass

Sets specified CSS class to block elements within text selection.

```
this.block.setClass('red');
```

removeClass

Removes specified CSS class from block elements within text selection.

```
this.block.removeClass('red');
```

Caret

setStart

Setting cursor position at the beginning of an element. Arguments are jQuery element or DOM node.

```
this.caret.setStart(node);
```

setEnd

Setting cursor position at the end of an element. Arguments are jQuery element or DOM node.

```
this.caret.setEnd(node);
```

setBefore

Setting cursor position before an element. Arguments are jQuery element or DOM node.

```
this.caret.setBefore(node);
```

setAfter

Setting cursor position after an element. Arguments are jQuery element or DOM node.

```
this.caret.setAfter(node);
```

getOffsetOfElement

Returns cursor position within an element. Arguments are jQuery element or DOM node.

```
var offset = this.caret.getOffsetOfElement(node);
```

getOffset

Returns an object with x position of the cursor in Redactor.

```
var offset = this.caret.getOffset();
```

setOffset

Setting cursor at specific position.

```
this.caret.setOffset(10);
```

Clean

stripTags

Removes tags form HTML code.

```
var html = this.clean.stripTags(html);
```

You can set which tags should be preserved, for example:

```
var html = this.clean.stripTags(html, '<a><p><br><span>');
```

getPlainText

Converts HTML to plain text.

```
var text = this.clean.getPlainText(html);
```

getOnlyImages

Removes all tags from HTML, except for image tags.

```
var code = this.clean.getOnlyImages(html);
```

getOnlyLinksAndImages

Removes all tag for HTML except for image tags and links.

```
var code = this.clean.getOnlyLinksAndImages(html);
```

encodeEntities

Converts symbols to HTML entities.

```
var code = this.clean.encodeEntities(html);
```

replaceDivsToBr

Replaces tag with line breaks
.

```
var html = this.clean.replaceDivsToBr(html);
```

replaceParagraphsToBr

Replaces <p> tag with line breaks
.

```
var html = this.clean.replaceParagraphsToBr(html);
```

Code

set

Places specified code inside Redactor.

```
1. this.code.set('<p>My html code</p>');
```

get

Returns code from Redactor.

Redactor always writes the final code to textarea. Editable layer, however, may contain temporary tags for proper visual representation of the text and for formatting purposes. Textarea code doesn't contain any of these temporary tags.

'code.get' is the only proper way to get HTML code from Redactor.

```
var html = this.code.get();
```

sync

This function synchronizes code between visual layer and textarea. This function runs every time there has been a change in content inside Redactor.

```
this.code.sync();
```

toggle

Toggles Redactor from visual mode to HTML code mode and vice versa.

```
this.code.toggle();
```

showCode

Switches Redactor from visual mode to HTML code mode.

```
this.code.showCode();
```

showVisual

Switches Redactor from HTML code mode to visual mode.

```
this.code.showVisual();
```

Core

getObject

Getting Redactor object.

```
var redactor = $('#content').redactor('core.getObject');
```

When using in plugins, Redactor object can be accessed using 'this'

getEditor

Accessing editable layer.

```
var editor = $('#content').redactor('core.getEditor');
```

When using in plugins, editable layer can be accessed using this.$editor variable.

getBox

Accessing container element, that contains editable layer and textarea.

```
var box = $('#content').redactor('core.getBox');
```

When using in plugins, editable layer can be accessed using this.$box variable.

getElement

Accessing element to which Redactor has been applied.

```
var element = $('#content').redactor('core.getElement');
```

When using in plugins, editable layer can be accessed using this.$element variable.

getTextarea

Accessing textarea. In case Redactor has been applied to textarea element, getTextarea will return this element. Otherwise Redactor will create its own textarea and return it.

Redactor always writes the final code to textarea. Editable layer, however, may contain temporary tags for proper visual representation of the text and for formatting purposes. Textarea code doesn't contain any of these temporary tags.

```
var textarea = $('#content').redactor('core.getTextarea');
```

When using in plugins, editable layer can be accessed using this.$textarea variable

getToolbar

Accessing toolbar element.

```
var toolbar = $('#content').redactor('core.getToolbar');
```

When using in plugins, editable layer can be accessed using this.$toolbar variable.

setCallback

Setting callback for functions. This method only works with plugins, and doesn't properly respond to external calls.

In plugins it can be called this way:

```
this.core.setCallback('my', e, data);
```

Arguments:

- **'my'** — callback name. Callback prefix will be added automatically to the name, resulting in name like this: 'myCallback'
- **e** — event object (optional)
- **data** — any data, object or function (optional)

If there's no need to pass an event object, it can be dropped from the arguments:

```
this.core.setCallback('my', data);
```

Now callback can be called on launch of Redactor:

```
$('#content').redactor({
    myCallback: function(e, data)
    {
        // your code
    }
});
```

Or, if you didn't pass event object in arguments:

```
$('#content').redactor({
    myCallback: function(data)
    {
        // your code
    }
});
```

Or, if you didn't pass any arguments:

```
$('#content').redactor({
    myCallback: function()
    {
        // your code
    }
});
```

Redactor object is always available inside of a callback via 'this'. It allows to call any Redactor function or variable, for example:

```
$('#content').redactor({
    myCallback: function(e, data)
    {
        consle.log(this.code.get());
    }
});
```

destroy

Turn Redactor off.

```
$('#content').redactor('core.destroy');
```

To call from a plugin:

```
this.core.destroy();
```

Focus

setStart

Setting focus at the beginning of text.

```
this.focus.setStart();
```

setEnd

Setting focus at the end of text.

```
this.focus.setEnd();
```

isFocused

Checking if focus is in Redactor and returns true or false.

```
var focused = this.focus.isFocused();
```

Indent

increase

Increases indent of selected text.

```
this.indent.increase();
```

decrease

Decreases indent of selected text.

```
this.indent.decrease();
```

Inline

format

Applies inline formatting to selected text.

```
this.inline.format('code');
```

CSS class or style can be set at the same time:

```
this.inline.format('span', 'class', 'red');
this.inline.format('span', 'style', 'color: red; font-weight: bold;');
```

removeStyle

Removes style attribute from all inline elements within selection.

```
this.inline.removeStyle();
```

removeStyleRule

Removes style attribute from all inline elements within selection.

```
this.inline.removeStyleRule('color');
```

removeFormat

Removes all inline formatting in selected text.

```
this.inline.removeFormat();
```

toggleClass

Sets or removes CSS class of a span tag in selected text.

```
this.inline.toggleClass('red');
```

toggleStyle

Sets or removes CSS style of a span tag in selected text.

```
this.inline.toggleStyle('color: red;');
```

Insert

set

Replaces previous code with new code.

```
this.insert.set('<p>Hello world!</p>');
```

text

Inserts plain text at the cursor.

```
this.insert.text('Hello world!');
```

html

Inserts HTML at the cursor.

```
this.insert.html('<p>Hello world!</p>');
```

Inserts HTML without cleaning html.

```
this.insert.html('<p>Hello world!</p>', false);
```

node

Inserts element at the cursor.

Returns inserted element.

```
var node = $('<span />').html('My element');
this.insert.node(node);
```

nodeToPoint

Inserts element at specified coordinates.

Coordinates can be previously obtained using caret.getOffset.

```
var node = $('<span />').html('My element');
this.insert.nodeToPoint(node, x, y);
```

nodeToCaretPositionFromPoint

Inserts element at the last coordinates of the cursor. This function is used for drag and drop.

```
var node = $('<span />').html('My element');
this.insert.nodeToCaretPositionFromPoint(e, node);
```

htmlWithoutClean

Inserts HTML without cleaning html.

```
this.insert.htmlWithoutClean('<p>Hello world!</p>');
```

Lang

get

This method returns a value by key for current language.

```
this.lang.get('html');
```

Line

insert

This method adds a horizontal rule <hr> to the text.

```
this.line.insert();
```

Link

toggleClass

If there are links inside of selected text, this method allows to toggle their class.

```
this.link.toggleClass('my-class');
```

List

toggle

This method detects if whether there's a list in selected text, and then toggle list formatting. If a list has been detected in the selected text, it will be removed; if a list hasn't been detected in the selected text, it will be added.

Arguments are list types: orderedlist or unorderedlist.

```
this.list.toggle('orderedlist');
```

insert

This method applies list formatting to selected text.

Arguments are list types: orderedlist or unorderedlist.

```
this.list.insert('orderedlist');
```

remove

This method removes list formatting from selected text.

Arguments are list types: orderedlist or unorderedlist.

```
this.list.remove('orderedlist');
```

Modal

load

Creates modal window and prepares it for launch.

Arguments:

- **name** – modal window template name
- **title** – modal window title
- **width** – modal window width in pixels

```
this.modal.load('image', 'Insert Image', 600);
```

show

Shows modal window.

```
this.modal.load('image', 'Insert Image', 600);
this.modal.show();
```

close

Closes modal window.

```
this.modal.close();
```

setTitle

Sets title for current modal window.

```
this.modal.setTitle('My Modal Header');
```

addTemplate

Adds modal window template.

```
var template = '<section id="redactor-modal-my-template"></section>';

this.modal.addTemplate('my-template', template);
```

addCallback

Adds function that will be launched on modal window open.

```
this.modal.addCallback('my-template', function()
{
    // your code
});
```

createCancelButton

Creates close button for modal window.

```
this.modal.load('my-template', 'My Template', 600);
this.modal.createCancelButton();
this.modal.show();
```

createDeleteButton

Creates red button inside of modal window. This button can be used for delete action.

```
this.modal.load('my-template', 'My Template', 600);

var buttonDelete = this.modal.createDeleteButton();
buttonDelete.on('click', function()
{
    // your code
});

this.modal.show();
```

createActionButton

Creates blue button inside of modal window. This button can be used for any actions.

```
this.modal.load('my-template', 'My Template', 600);

var button = this.modal.createActionButton();
button.on('click', function()
{
    // your code
});

this.modal.show();
```

getModal

Accessing 'body' of current modal window.

```
var $modal = this.modal.getModal();
```

createTabber

Creates tabs navigation in modal window.

```
var $modal = this.modal.getModal();

this.modal.createTabber($modal);
this.modal.addTab(1, 'Tab 1', 'active');
this.modal.addTab(2, 'Tab 2');

var $tabBox1 = $('<div class="redactor-tab redactor-tab1">');
var $tabBox2 = $('<div class="redactor-tab redactor-tab2">').hide();

$modal.append($tabBox1);
$modal.append($tabBox2);
```

addTab

Adds a tab in tab navigation in modal window.

```
var $modal = this.modal.getModal();

this.modal.createTabber($modal);
this.modal.addTab(1, 'Tab 1', 'active');
this.modal.addTab(2, 'Tab 2');

var $tabBox1 = $('<div class="redactor-tab redactor-tab1">');
var $tabBox2 = $('<div class="redactor-tab redactor-tab2">').hide();

$modal.append($tabBox1);
$modal.append($tabBox2);
```

Observe

images

Enables on click editing for images. This method is required whenever image code has been added by any means other then Redactor API.

```
this.observe.images();
```

links

Enables on click tooltip for all links in Redactor (if tooltip setting is on). This method is required whenever image code has been added by any means other than Redactor API.

```
this.observe.links();
```

addButton

Adds a tag which will activate a button if cursor is placed inside of this tag.

Arguments:

- **tag** – tag that will activate a button
- **btnName** – name of a button that will be activated

```
this.observe.addButton('code', 'my-button');
```

Paragraphize

load

Marks up text with paragraphs, if it hasn't been marked up with paragraphs yet.

```
var html = this.paragraphize.load(html);
```

Placeholder

toggle

Turns placeholder on and off depending on whether there is any content in Redactor or not.

```
this.placeholder.toggle();
```

remove

Removes placeholder.

```
this.placeholder.remove();
```

Progress

show

Shows progress bar.

```
this.progress.show();
```

hide

Hides progress bar.

```
this.progress.hide();
```

Selection

get

Accessing Selection and Range of selected text objects. These objects will be placed in this.sel and this.range variables.

```
this.selection.get();

console.log(this.sel);
console.log(this.range);
```

addRange

Adding modified Range object.

```
this.selection.get();
this.range.setStart(node, 0);
this.range.setEnd(node, 0);
this.selection.addRange();
```

getCurrent

Accessing element that contains cursor. If cursor is located in an empty element, getCurrent returns this element, otherwise it returns text node. If current element is not found, returns 'false'

```
var current = this.selection.getCurrent();
```

getParent

Accessing parent element (in regards to cursor position). If parent element is not found, returns 'false'

```
var parent = this.selection.getParent();
```

getBlock

Accessing parent block element (in regards to cursor position). If parent block element is not found, returns 'false'

```
var block = this.selection.getBlock();
```

getNodes

Returns an array of all elements in a selected text. If selected text doesn't have any elements, returns an array with first element value set to 'false'.

```
var nodes = this.selection.getNodes();
```

getBlocks

Returns an array of all block elements in a selected text. If selected text doesn't have any block elements, returns an array with first element value set to 'false'.

```
var blocks = this.selection.getBlocks();
```

getInlines

Returns an array of all inline elements in a selected text.

If selected text doesn't have any inline elements, returns an array with first element value set to 'false'

```
var inlines = this.selection.getInlines();
```

wrap

Wraps selected text with specified tag. Returns an element created by this tag.

```
var wrapper = this.selection.wrap('h2');
```

selectElement

Selects specified element.

```
this.selection.selectElement(node);
```

save

Saves cursor position in Redactor.

This is useful in case, for example, you want to open a modal window and perform some actions in it. In this case, Redactor will lose focus and cursor position. To restore focus cursor position, you should save it first using selection.save and restore afterward using selection.restore.

```
this.selection.save();
```

getMarker

Returns marker element.

If you wish to set cursor position to a specific location after pasting some text, you can use markers.

For example:

```
var node1 = $('<p>Paragraph 1</p>');
var node2 = $('<p>Paragraph 2</p>');

var marker = this.selection.getMarker();

node1.append(marker);

var container = $('<div />').append(node1).append(node2);

this.insert.htmlWithoutClean(container.html());
this.selection.restore();
```

To select some text, you can add two markers:

```
var node1 = $('<p>Paragraph 1</p>');
var node2 = $('<p>Paragraph 2</p>');

var marker1 = this.selection.getMarker(1);
var marker2 = this.selection.getMarker(2);

node1.prepend(marker1);
node2.append(marker2);

var container = $('<div />').append(node1).append(node2);

this.insert.htmlWithoutClean(container.html());
this.selection.restore();
```

getMarkerAsHtml

Returns marker as HTML.

If you wish to set cursor position to a specific location after pasting some text, you can use markers.

For example:

```
var marker = this.selection.getMarkerAsHtml();
var html = '<p>Paragraph 1' + marker + '</p><p>Paragraph 2</p>';
this.insert.htmlWithoutClean(html);
this.selection.restore();
```

To select some text, you can add two markers:

```
var marker1 = this.selection.getMarkerAsHtml(1);
var marker2 = this.selection.getMarkerAsHtml(2);
var html = '<p>' + marker1 + 'Paragraph 1' + marker2 + '</p><p>Paragraph
2</p>';

this.insert.htmlWithoutClean(html);
```

```
this.selection.restore();
```

restore

Restores cursor position (previously saved by selection.save)

```
this.selection.restore();
```

removeMarkers

Removes markers that were created to save cursor position using selection.save.

If selection.restore was called after selection.save, markers will be removed automatically.

```
this.selection.removeMarkers();
```

getText

Returns selected text in text format.

```
var text = this.selection.getText();
```

getHtml

Returns selected text in HTML format.

```
var html = this.selection.getHtml();
```

remove

Deselects text.

```
this.selection.remove();
```

selectAll

Select all text inside Redactor.

```
this.selection.selectAll();
```

replaceSelection

This method allows to replace selected text with other text or HTML code.

```
this.selection.replaceSelection(html);
```

Utils

isMobile

Detects whether Redactor is launched in on a mobile device (excluding iPad) or not. Returns true or false.

```
var isMobile = this.utils.isMobile();
```

isDesktop

Detects whether Redactor is launched on a desktop device. Returns true or false.

```
var isDesktop = this.utils.isDesktop();
```

isString

Detects if a variable is a string. Returns true or false.

```
var isString = this.utils.isString(variable);
```

isEmpty

Detects is a string is empty. Returns true or false.

```
var isEmpty = this.utils.isEmpty(html);
```

By default, removes empty tags from a string; if you have to preserve empty tags, second argument should be 'false':

```
var isEmpty = this.utils.isEmpty(html, false);
```

normalize

Removes 'px' form variable value.

```
var str = '10px';
str = this.utils.normalize(str);
// returns 10
```

hexToRgb

Converts CSS colors from HEX to RGB.

```
var color = '#000';
color = this.utils.hexToRgb(color);
// returns rgb(0, 0, 0);
```

getOuterHtml

Returns HTML code of an element, including element itself.

```
var $div = $('<div>').html('My code');
var html = this.utils.getOuterHtml($div);
// returns <div>My code</div>
```

removeEmptyAttr

If an element has an empty attribute, this function will delete this attribute.

```
this.utils.removeEmptyAttr(el, 'style');
```

removeInlineTags

This function removes inline tags inside of an element while preserving their content.

```
this.utils.removeInlineTags(el);
```

replaceToTag

Replaces element tag with a new tag while preserving all attributes of an old tag.

```
this.utils.replaceToTag(el, 'h3');
```

isStartOfElement

Detects if cursor is located at the beginning of the current element. Returns true or false.

```
var isStart = this.utils.isStartOfElement();
```

isEndOfElement

Detects if cursor is located at the beginning of the current element. Returns true or false.

```
var isEnd = this.utils.isEndOfElement();
```

isBlock

Detects if an element is a block element. Returns true or false.

```
var isBlock = this.utils.isBlock(el);
```

isBlockTag

Detects if a tag is a block element tag. Returns true or false.

```
1. var isBlockTag = this.utils.isBlockTag('p');
```

isSelectAll

Detects if all text is selected. Returns true or false.

```
var isSelectAll = this.utils.isSelectAll();
```

enableSelectAll

Turns on a flag that all text is selected.

```
this.utils.enableSelectAll();
```

disableSelectAll

Turns off a flag that all text is selected.

```
this.utils.disableSelectAll();
```

isRedactorParent

Detects if an element is located inside of Redactor. Returns false if it is not, and returns element itself if it is located inside of Redactor.

```
var isRedactorParent = this.utils.isRedactorParent(el);
```

isCurrentOrParent

Detects if a tag equals to the current or parent element's tag (in regards to cursor position). Returns false if tags are different and element itself if tags are equal.

```
var isCurrentOrParent = this.utils.isCurrentOrParent('BLOCKQUOTE');
```

isOldIe

Detects if Redactor is launched in IE9 or older. Returns true or false.

```
var isOldIe = this.utils.isOldIe();
```

isIe11

Detects if Redactor is launched in IE11. Returns true or false.

```
var isIe11 = this.utils.isIe11();
```

browser

Detects user's browser.

Possible values:

- webkit (chrome, safari or opera webkit)
- chrome
- msie
- mozilla
- opera (not webkit)

Returns true or false.

```
var isBrowser = this.utils.browser('msie');
```

Callbacks

Overview

Each callback receives Redactor object on init, so that you can always access any API methods from callback function using ‘this’, for example:

```
$('#redactor').redactor({
    initCallback: function()
    {
        var html = this.code.get();
    }
});
```

Initialization

startCallback

Called before Redactor launch.

This callback allows to initialize your custom functions, attributes and settings prior to Redactor initialization.

```
$('#redactor').redactor({
    startCallback: function()
    {
        // your code
    }
});
```

initCallback

Called after Redactor launch.

This callback allows to initialize your custom functions, attributes and settings.

```
$('#redactor').redactor({
```

```
    initCallback: function()
    {
        console.log(this.code.get());
    }
});
```

Destroy

destroyCallback

Called after Redactor is destroyed using core.destroy.

```
$('#redactor').redactor({
    destroyCallback: function()
    {
        // your code
    }
});
```

Paste

pasteBeforeCallback

Called on text paste before this text is stripped of garbage tags and styles.

Callback must return HTML code.

This callback allows to perform custom code cleanup or make any necessary changes to the code before Redactor performs its own cleanup operations.

The callback passes following argument:

- **html** — inserted text (HTML code) prior to Redactor cleanup

```
$('#redactor').redactor({
    pasteBeforeCallback: function(html)
    {
        console.log(html);
        return html;
    }
});
```

pasteCallback

This callback triggers after pasted text has been cleaned up from garbage tags and styles.

Callback must return HTML code.

This callback allows to make any necessary changes to the code after Redactor performs its cleanup operations.

The callback passes following argument:

- **html** — inserted text (HTML code) after Redactor cleanup

```
$('#redactor').redactor({
    pasteCallback: function(html)
    {
        console.log(html);
        return html;
    }
});
```

Change

changeCallback

This callback is triggered every time there's a change in text, but only if users is editing text in visual mode. changeCallback will not trigger if a user edits code in HTML code mode, to track chsnges in HTML code mode use codeKeydownCallback.

```
$('#redactor').redactor({
    changeCallback: function()
    {
        console.log(this.code.get());
    }
});
```

Sync

syncBeforeCallback

This callback is triggered upon synchronization of code between visual layer and textarea. syncBeforeCallback is triggered before text from visual layer is pasted to textarea.

This callback must return HTML code.

Visual layer contains multiple temporary tags that allow visual editing. Synchronization with textarea inserts clean code to the textarea, removing all temporary tags, styles and attributes. This callback allows to intercept this code before it is cleaned and inserted into the textarea to make necessary changes to the code prior to synchronization.

Argument

- **html** — code form Redactor prior to inserting into textarea

```
$('#redactor').redactor({
    syncBeforeCallback: function(html)
    {
        console.log(html);
        return html;
    }
});
```

syncCallback

This callback is triggered after synchronization between visual layer and textarea. Resulting code will be in textarea

This callback is identical to changeCallback and makes callback functions more consistent.

```
$('#redactor').redactor({
    syncCallback: function()
    {
        console.log(this.code.get());
    }
});
```

Keypress

keydownCallback

This callback is triggered on keydown in Redactor. keydownCallback is called before Redactor functions, that handle keydown.

Argument:

- **e** — event object

```
$('#redactor').redactor({
    keydownCallback: function(e)
    {
        console.log(this.code.get());
    }
});
```

Next example allows to avoid handling keydown functions by Redactor and create custom functions for handling key inputs:

```
$('#redactor').redactor({
    keydownCallback: function(e)
    {
        e.preventDefault();

        // your keydown methods
```

```
        return false;
    }
});
```

keyupCallback

This callback is triggered on keyup in Redactor. keyupCallback is called before Redactor functions, that handle keyup.

Argument:

- **e** — event object

```
$('#redactor').redactor({
    keyupCallback: function(e)
    {
        console.log(this.code.get());
    }
});
```

Next example allows to avoid handling keyup functions by Redactor and create custom functions for handling key inputs:

```
$('#redactor').redactor({
    keyupCallback: function(e)
    {
        e.preventDefault();

        // your keydown methods

        return false;
    }
});
```

enterCallback

This callback is triggered after Enter/Return key is pressed.

Arguments:

- **e** — event object

```
$('#redactor').redactor({
    enterCallback: function(e)
    {
        console.log(this.code.get());
    }
});
```

You can return false in this callback to prevent Redactor from handling Enter/Return key inputs. It allows you to develop custom handlers for this event.

```
$('#redactor').redactor({
    enterCallback: function(e)
    {
        console.log(this.code.get());
        return false;
    }
});
```

Autosave

autosaveCallback

This callback is triggered on successful autosave. Returns null or JSON, depending on what you decide your autosave script will return.

Arguments:

- **name** – POST variable name (textarea name)
- **json** — JSON object with data from server script

For example, after autosave, you wish to pass the saved text back to the callback. Make your autosave script to return a JSON string. The callback will get this string as an object and you'll be able to get data from this object.

```
$('#redactor').redactor({
    autosaveCallback: function(name, json)
    {
        console.log(name);
        console.log(json.content);
    }
});
```

autosaveErrorCallback

This callback is triggered whenever there's an error during autosave.

Arguments:

- **name** – POST variable name (textarea name)
- **json** — JSON object with data from autosave script

For this callback to trigger, your autosave script should return a JSON string with error parameter, for example (in PHP):

```
<?php
    $array = array(
        'error' => true,
```

```
        'message' => 'Something went wrong...'
    );

    echo stripslashes(json_encode($array));
?>
```

As a result, autosaveErrorCallback will be triggered and you'll be able to display appropriate message to the user:

```
$('#redactor').redactor({
    autosaveErrorCallback: function(name, json)
    {
        console.log(name);
        alert(json.message);
    }
});
```

Events

focusCallback

This callback is trigered every time when Redactor gets focus.

Argument:

- **e** — event object

```
$('#redactor').redactor({
    focusCallback: function(e)
    {
        console.log(this.code.get());
    }
});
```

blurCallback

This callback is triggered every time a blur event occurs, for example, on click outside of Redactor.

Argument:

- **e** — event object

```
$('#redactor').redactor({
    blurCallback: function(e)
    {
        console.log(this.code.get());
    }
});
```

dropCallback

This callback is triggered on drop event (including image and file uploads via drag and drop).

Argument:

- **e** — event object

```
$('#redactor').redactor({
    dropCallback: function(e)
    {
        console.log(e);
    }
});
```

clickCallback

This callback is triggered on click inside of editable area.

Argument:

- **e** — event object

```
$('#redactor').redactor({
    clickCallback: function(e)
    {
        console.log(e);
    }
});
```

Source

sourceCallback

This callback is triggered every time when a user switches from visual mode to HTML code view mode.

```
$('#redactor').redactor({
    sourceCallback: function(html)
    {
        console.log(html);
    }
});
```

codeKeydownCallback

This callback is triggered in HTML code mode on keydown event.

Argument:

- **e** — event object

```
$('#redactor').redactor({
    codeKeydownCallback: function(e)
    {
        console.log(this.code.get());
    }
});
```

codeKeyupCallback

This callback is triggered in HTML code mode on keyup event.

Argument:

- **e** — event object

```
$('#redactor').redactor({
    codeKeyupCallback: function(e)
    {
        console.log(this.code.get());
    }
});
```

visualCallback

This callback is triggered every time when a user switches from HTML code view mode back to visual mode.

```
$('#redactor').redactor({
    visualCallback: function(html)
    {
        console.log(html);
    }
});
```

Image

imageUploadCallback

This callback is triggered on successful image upload (including via drag and drop)

Arguments:

- **image** – DOM element of successfully uploaded and inserted image
- **json** – JSON object with data from upload script

Example:

```
$('#redactor').redactor({
    imageUploadCallback: function(image, json)
    {
        $(image).attr('id', json.id);
    }
});
```

Upload script forms a JSON string:

```
{
    "filelink": "/images/img.jpg",
    "id": 1
}
```

JSON may contain any data, however, filelink is required.

imageUploadErrorCallback

This callback is triggered whenever there is an error in image upload.

Argument

- **json** — JSON object with data from upload script

For this callback to trigger, your autosave script should return a JSON string with error parameter, for example (in PHP):

```
<?php
    $array = array(
        'error' => true,
        'message' => 'Something went wrong...'
    );

    echo stripslashes(json_encode($array));
?>
```

As a result, imageUploadErrorCallback will be triggered and you'll be able to display appropriate message to the user:

```
$('#redactor').redactor({
    imageUploadErrorCallback: function(json)
    {
        alert(json.message);
    }
});
```

imageDeleteCallback

This callback is triggered when an image is deleted.

Arguments:

- **url** - link to an image
- **image** - DOM element of an image

In this callback, you can get more from the DOM of the image and send the data to the server for the complete identification of the image.

```
$('#redactor').redactor({
    imageDeleteCallback: function(url, image)
    {
        var id = $(image).attr('id');

        $.ajax({
            url: '/myadmin/images/delete/',
            type: 'post',
            data: url=' + url + '&id=' + id
        });
    }
});
```

File

fileUploadCallback

This callback is triggered on successful file upload (including via drag and drop).

Arguments:

- **link** – DOM of an element with a link to uploaded file (required)
- **json** – JSON object with data from upload script.

Example:

```
$('#redactor').redactor({
    fileUploadCallback: function(link, json)
    {
        $(link).attr('id', json.id);
    }
});
```

Upload script forms a JSON string:

```
{
```

```
    "filelink": "/files/myfile.pdf",
    "filename": "My PDF file",
    "id": 1
}
```

JSON may contain any data, however, filelink and filename are required.

fileUploadErrorCallback

This callback is triggered whenever there is an error in file upload.

Argument

- **json** — JSON object with data from upload script

For this callback to trigger, your autosave script should return a JSON string with error parameter, for example (in PHP):

```
<?php
    $array = array(
        'error' => true,
        'message' => 'Something went wrong...'
    );

    echo stripslashes(json_encode($array));
?>
```

As a result, fileUploadErrorCallback will be triggered and you'll be able to display appropriate message to the user:

```
$('#redactor').redactor({
    fileUploadErrorCallback: function(json)
    {
        alert(json.message);
    }
});
```

Modal

modalOpenedCallback

This callback is triggered after a modal window opens.

Arguments:

- **name** - unique modal window name (usually a name of a modal window template)
- **modal** - DOM element of modal window

```
$('#redactor').redactor({
    modalOpenedCallback: function(name, modal)
    {
        console.log(name);
    }
});
```

modalClosedCallback

This callback is triggered after modal window closes.

Arguments:

- **name** - unique modal window name (usually a name of a modal window template)

```
$('#redactor').redactor({
    modalClosedCallback: function(name)
    {
        console.log(name);
    }
});
```

Dropdown

dropdownShowCallback

This callback is triggered before dropdown shows up.

Arguments:

- **dropdown** — DOM element of dropdown
- **key** - a name of a button that triggered dropdown
- **button** - DOM element of a button

```
$('#redactor').redactor({
    dropdownShowCallback: function(dropdown, key, button)
    {
        console.log(dropdown);
    }
});
```

dropdownShownCallback

This callback is triggered after dropdown shows.

Arguments:

- **dropdown** — DOM element of dropdown
- **key** - a name of a button that triggered dropdown

- **button** - DOM element of a button

```
$('#redactor').redactor({
    dropdownShownCallback: function(dropdown, key, button)
    {
        console.log(dropdown);
    }
});
```

dropdownHideCallback

This callback is triggered after dropdown collapses.

Argument:

- **dropdown** — DOM element of dropdown

```
$('#redactor').redactor({
    dropdownHideCallback: function(dropdown)
    {
        console.log(dropdown);
    }
});
```

Table

insertedTableCallback

This callback requires Table plugin.
This callback is triggered after a table been inserted to Redactor using table plugin.

Argument:

- **table** - DOM element of inserted table

```
$('#redactor').redactor({
    insertedTableCallback: function(table)
    {
        console.log(table);
    }
});
```

Link

insertedLinkCallback

This callback is triggered after a link inserted into Redactor.

Arguments:

- **link** - DOM element of inserted link

```
$('#redactor').redactor({
    insertedLinkCallback: function(link)
    {
        console.log(link);
    }
});
```

deletedLinkCallback

This callback is triggered whenever a link or links are deleted.

Arguments:

- **links** - an array of DOM elements of removed links

```
$('#redactor').redactor({
    deletedLinkCallback: function(links)
    {
        $.each(links, function(i,s)
        {
            console.log(s);
        });
    }
});
```

Linkify

linkifyCallback

This callback is triggered every time Redactor performs automatic URL conversion. At this time, Redactor automatically converts URLs into links ('convertUrlLinks' and 'convertUrl' settings), YouTube and Vimeo URLs into video embeds ('convertVideoLinks' setting) and image URLs into images ('convertImageLinks' setting). Whenever such conversion is performed, this callback will return a complete set of DOM elements of all converted objects — `<a>` for converted links, `<iframe>` for video embeds and `<img>` for converted images.

```
$('#content').redactor({
     linkifyCallback: function(elements)
     {
         $.each(elements, function(i,s)
         {
              console.log(s);
         });
     }
});
```

Upload

uploadStartCallback

This callback triggers before image or file upload begins. Callback arguments are:

- e - upload event
- formData - form data object

```
$('#redactor').redactor({
    imageUpload: '/image-upload/',
    uploadStartCallback: function(e, formData)
    {
        console.log('My upload started!');
    }
});
```

Toolbar

toolbarFixedTopOffset

This setting allows to set how far from the top of the page the fixed toolbar will be placed. By default, `toolbarFixedTopOffset` is set to 0.

```
$('#redactor').redactor({
    toolbarFixedTopOffset: 100 // pixels
});
```

toolbar

This setting allows to turn off the toolbar. Keep in mind, that keyboard shortcuts will still function.

```
$('#redactor').redactor({
    toolbar: false
});
```

toolbarExternal

This setting allows you to display Redactor toolbar separately from the text area. The external toolbar can be placed inside of any element on the page:

```
$('#redactor').redactor({
    toolbarExternal: '#your-toolbar-layer-id'
});
```

toolbarFixed

This setting affixes external toolbar to a specific position on a page. When the is being scrolled down, fixed toolbar will remain in place and preserve its initial width.

```
$('#redactor').redactor({
    toolbarFixed: true
});
```

toolbarFixedTarget

This setting allows to set a specific element on a page in relation to which fixed toolbar will be displayed. By default, toolbarFixedTarget is set to 'document'

```
$('#redactor').redactor({
    toolbarFixed: true,
    toolbarFixedTarget: '#my-parent-layer'
});
```

toolbarOverflow

When set to 'true', this setting will place all toolbar buttons on mobile devices in a single line regardless of how many buttons are there. If there's more buttons then can fit on a screen, horizontal scroll will appear.

Buttons

buttons

By default set to:

```
['html', 'formatting', 'bold', 'italic', 'deleted',
'unorderedlist', 'orderedlist', 'outdent', 'indent',
'image', 'link', 'alignment', 'horizontalrule']
// + 'underline'
```

Sets an array of buttons that will be displayed in the toolbar.

Can be customized like this:

```
$('#redactor').redactor({
    buttons: ['formatting', 'bold', 'italic', 'deleted']
});
```

source

Set to 'true' by default.

This setting adds HTML code view button to the toolbar.

```
$('#redactor').redactor({
    source: false
});
```

buttonsHide

This setting allows to hide certain buttons on launch.

```
$('#redactor').redactor({
    buttonsHide: ['image', 'link']
});
```

buttonsHideOnMobile

This setting allows to hide certain buttons on mobile devices:

```
$('#redactor').redactor({
    buttonsHideOnMobile: ['image', 'video']
});
```

In this example, video and image buttons will be hidden on smartphones, but still visible on desktops, laptops and tablets.

activeButtons

By default set to:

```
['deleted', 'italic', 'bold', 'underline', 'unorderedlist', 'orderedlist',
'alignleft', 'aligncenter', 'alignright', 'justify']
```

This setting allows to set which buttons will become active if the cursor is positioned inside of a respectively formatted text.

activeButtons should always be used with activeButtonsStates, for example:

```
$('#redactor').redactor({
    activeButtons: ['italic', 'bold', 'my-button'],
    activeButtonsStates: {
        b: 'bold',
        strong: 'bold',
        i: 'italic',
        em: 'italic',
        code: 'my-button'
    }
});
```

activeButtonsStates

By default set to:

```
{
    b: 'bold',
    strong: 'bold',
    i: 'italic',
```

```
    em: 'italic',
    del: 'deleted',
    strike: 'deleted',
    ul: 'unorderedlist',
    ol: 'orderedlist',
    u: 'underline'
}
```

This setting allows to set which tags make buttons active.

activeButtonsStates should always be used with activeButtons, for example:

```
$('#redactor').redactor({
    activeButtons: ['italic', 'bold', 'my-button'],
    activeButtonsStates: {
        b: 'bold',
        strong: 'bold',
        i: 'italic',
        em: 'italic',
        code: 'my-button'
    }
});
```

Special

visual

This setting allows to launch Redactor in code view mode by default:

```
$('#redactor').redactor({
    visual: false
});
```

linebreaks

This setting turns on markup with line breaks instead of paragraphs when user presses Return key.

```
$('#redactor').redactor({
   linebreaks: true
});
```

s3

Set to 'false' by default.

This setting allows to setup image and file uploads to Amazon S3 servers:

```
$('#redactor').redactor({
    s3: '/your-server-side-script.php'
```

```
    // you can use any server-side language, not only PHP
});
```

Server script forms all necessary parameters and a URL for Amazon S3 upload, for example:

```
<?php

$S3_KEY = 'Your Amazon access key';
$S3_SECRET = 'Your Amazon secret key';
$S3_BUCKET = '/your-bucket-name'; // bucket needs / on the front


$S3_URL = 'http://s3.amazonaws.com';

// expiration date of query
$EXPIRE_TIME = (60 * 5); // 5 minutes

$objectName = '/' . $_GET['name'];

$mimeType = $_GET['type'];
$expires = time() + $EXPIRE_TIME;
$amzHeaders = "x-amz-acl:public-read";
$stringToSign =
"PUTnn$mimeTypen$expiresn$amzHeadersn$S3_BUCKET$objectName";

$sig = urlencode(base64_encode(hash_hmac('sha1', $stringToSign, $S3_SECRET,
true)));
$url =
urlencode("$S3_URL$S3_BUCKET$objectName?AWSAccessKeyId=$S3_KEY&Expires=$exp
ires&Signature=$sig");

echo $url;

?>
```

scrollTarget

Set to false by default.

This setting allows to set a parent layer when Redactor is placed inside of layer with scrolling. When this is set, scroll will return to correct position when a user pastes some text.

```
$('#redactor').redactor({
   scrollTarget: '#container'
});
```

tabIndex

This setting sets tab index form value for Redactor.

```
$('#redactor').redactor({
    tabIndex: 1
```

```
});
```

tabifier

Set to true by default.

Sets indent for code when using code.toggle or code.get.

```
$('#redactor').redactor({
    tabifier: false
});
```

Lang

lang

By default, Redactor is set to English language, but you can download (or create) and install your own language file. All you need is to set a language file URL in page header:

```
<!doctype html>
<html>
    <head>
        <title>Redactor</title>
        <meta charset="utf-8">

        <script src="/js/jquery-2.0.1.min.js"></script>

        <link rel="stylesheet" href="/js/redactor/redactor.css" />
        <script src="/js/redactor/redactor.js"></script>

        <script src="/js/redactor/es.js"></script>

        <script type="text/javascript">
        $(function()
        {
            $('#content').redactor({
                lang: 'es'
            });
        });
        </script>
    </head>
    <body>
        <textarea id="content" name="content"></textarea>
    </body>
</html>
```

direction

Redactor supports both right-to-left and left-to-right text directions. By default, Redactor is set to work with left-to-right, to set it to right-to-left, use 'lang' setting:

```
$('#redactor').redactor({
    direction: 'rtl'
});
```

Placeholder

placeholder

Placeholder can be set in two different ways:

1. You can set 'placeholder' setting with text argument:

```
$('#redactor').redactor({
    placeholder: 'Enter a comment...'
});
```

2. You can set placeholder as an attribute of an HTML element that uses Redactor:

```
1. <textarea id="redactor" placeholder="Enter a comment..."></textarea>
```

Height

minHeight

This setting allows to set minimum height for Redactor.

```
$('#redactor').redactor({
    minHeight: 300 // pixels
});
```

maxHeight

This setting allows to set maximum height for Redactor.

```
$('#redactor').redactor({
    maxHeight: 800 // pixels
});
```

Autosave

autosave

This setting turns on autosave feature either at a time interval or on change.

Setting requires server script URL as an argument:

```
$('#redactor').redactor({
    autosave: '/myadmin/content/save/'
```

```
});
```

Data will be passed to the server script using POST method, 'textarea' variable will contain all the content. Here's a PHP example:

```
echo $_POST['textareaname'];
```

The name of 'textarea' variable can be changed via autosaveName setting.

autosaveName

Sets POST variable name for the variable to be passed to the autosave script:

```
$('#redactor').redactor({
    autosave: '/myadmin/content/save/',
    autosaveName: 'my-content-name'
});
```

autosaveInterval

By default, this autosaveInterval is set to 60 seconds. You can change it:

```
$('#redactor').redactor({
    autosave: '/myadmin/content/save/',
    autosaveInterval: 30 // seconds
});
```

If the text hasn't been changed since the last autosave event, Redactor will not send data to the server again.

autosaveOnChange

Data can be saved automatically on change regardless of time interval between changes (time interval autosaving will be turned off automatically with this setting set to 'true'). Here is a setting for on change autosave:

```
$('#redactor').redactor({
    autosave: '/myadmin/content/save/',
    autosaveOnChange: true
});
```

autosaveFields

This setting allows to pass extra parameters with autosave requests. These parameters are passed using POST method.

```
$('#redactor').redactor({
    autosave: '/my-autosave-server-script/'
    autosaveFields: {
      'id': '#page-id'
```

```
    }
});
```

In this example autosave will pass input value of a page with ID #page-id:

```
<input type="hidden" id="page-id" name="page-id" value="10" />
```

In your server script you can access this value in POST array. Here's a PHP example:

```
echo $_POST['page-id'];
```

You could also set value directly:

```
$('#redactor').redactor({
    autosave: '/my-autosave-server-script/'
    autosaveFields: {
      'id': 10
    }
});
```

Focus

focus

By default, Redactor doesn't receive focus on load, because there may be other input fields on a page. However, to set focus to Redactor, you can use this setting:

```
$('#redactor').redactor({
    focus: true
});
```

focusEnd

This setting allows to set focus after the last character in Redactor:

```
$('#redactor').redactor({
    focusEnd: true
});
```

Keypress

enterKey

This setting allows to prevent use of Return key.

```
$('#redactor').redactor({
    enterKey: false
});
```

With this set to 'false' and pastePlainText is set to true, line breaks will be replaced by spaces.

tabKey

This setting turns on Tab key handling. If tabKey is set to false, Tab key will set focus to the next input field on a page.

```
$('#redactor').redactor({
    tabKey: false
});
```

If there's no text in Redactor, then regardless of tabKey setting focus will be set to the next input field on a page.

tabAsSpaces

This setting allows to apply spaces instead of tabulation on Tab key. To turn this setting on, set number of spaces:

```
$('#redactor').redactor({
    tabAsSpaces: 4
});
```

preSpaces

Set to '4' by default.

This setting allows to set the number of spaces that will be applied when a user presses Tab key inside of preformatted blocks.

If set to 'false', Tab key will apply tabulation instead of spaces inside of preformatted blocks.

```
$('#redactor').redactor({
   preSpaces: false
});
```

Clean

pastePlainText

Set to 'false' by default.

This setting turns on pasting as plain text. The pasted text will be stripped of any tags, line breaks will be marked with
 tag. With this set to 'true' and 'enterKey' set to 'false', line breaks will be replaced by spaces.

```
$('#redactor').redactor({
    pastePlainText: true
```

```
});
```

cleanOnPaste

Set to 'true' by default.

This setting allows clean up all unnecessary tags and extra styles from the pasted text. When set to 'false' , the text will not be stripped of any MS Word tags or styles.

```
$('#redactor').redactor({
   cleanOnPaste: false
});
```

cleanSpaces

Set to 'true' by default.

This setting allows to replace two spaces with a single space when pasting from MS Word.

```
$('#redactor').redactor({
   cleanSpaces: false
});
```

cleanStyleOnEnter

Set to 'false' by default.

If set to 'true', this setting will prevent new paragraph from inheriting styles, classes and attributes form a previous paragraph

```
$('#redactor').redactor({
   cleanStyleOnEnter: true
});
```

paragraphize

Set to 'true' by default.

When linebreaks setting is not set, new and pasted text will be processed by a paragraph markup function. All pasted text and all newly entered text will be marked up with paragraphs to preserve proper text formatting.

This setting can be turned off:

```
1. $('#redactor').redactor({
2.    paragraphize: false
3. });
```

replaceDivs

Set to 'true' by default.

This setting makes Redactor to convert all divs in a text into paragraphs. With 'linebreaks' set to 'true', all div tags will be removed, and text will be marked up with
tag.

To turn this setting off:

```
$('#redactor').redactor({
   replaceDivs: false
});
```

allowedTags

Set to empty by default.

This setting allows to set tags that are allowed inside of Redactor. All other tags will be automatically removed upon initialization, on paste or when using this.insert.html.

```
$('#redactor').redactor({
    allowedTags: ['p', 'h1', 'h2', 'pre']
});
```

If 'linebreaks' is set to 'true' and there's <p> in allowedTags, it will be removed.

If 'linebreaks' is set to 'false' and there's no <p> tag in allowedTags, it will be added automatically to preserve proper formatting.

allowedTages and deniedTags cannot be set simultaneously.

Top-level HTML tags ('html', 'head', 'link', 'body', 'meta', 'applet') are never allowed inside of Redactor and will always be removed regardless of this setting.

deniedTags

By default set to:

```
['style', 'script']
```

This setting allows to set extra tags that are not allowed inside of Redactor. These tags will be automatically removed upon initialization, on paste or when using this.insert.html.

```
$('#redactor').redactor({
    deniedTags: ['p', 'h1', 'h2', 'pre']
});
```

If 'linebreaks' is set to 'false' and there's <p> tag in deniedTags, it will be added automatically to preserve proper formatting.

'allowedTages' and 'deniedTags' cannot be set simultaneously.

Top-level HTML tags ('html', 'head', 'link', 'body', 'meta', 'applet') will always be removed regardless of this setting, unless wrapped in 'pre' tag (formatting option 'Code').

removeComments

Set to 'false' by default.

If set to 'true', all HTML comment will be removed from code.

```
$('#redactor').redactor({
    removeComments: true
});
```

replaceTags

By default, <strike> tag will always be replaced with <del>. You can set your own array of tags to be replaced:

```
$('#redactor').redactor({
    replaceTags: [
        ['strike', 'del'],
        ['i', 'em'],
        ['b', 'strong'],
        ['big', 'strong'],
        ['strike', 'del']
    ]
});
```

replaceStyles

This setting allows to set which span styles will be replaced by tags.

By default

```
"font-weight: bold" will be replaced by <strong>

"font-style: italic" will be replaced by <em>

and
```

"text-decoration: underline" will be replaced by <u>

You can set your own array of styles that will be replaced:

```
$('#redactor').redactor({
    replaceStyles: [
        ['font-weight:\s?bold', "strong"],
        ['font-style:\s?italic', "em"],
        ['text-decoration:\s?underline', "u"],
        ['color:\s?red', "mark"]
    ]
});
```

First parameter in the array is a string for regular expression.

removeDataAttr

Set to 'false' by default.

If set to 'true', Redactor will remove all data attributes in the code.

```
$('#redactor').redactor({
    removeDataAttr: true
});
```

removeAttr

Set to 'false' by default.

This setting allows to set attributes that will be removed from the code.

```
$('#redactor').redactor({
    removeAttr:  [
        ['p', 'class'],
        ['span', 'style']
    ]
});
```

If you need to set more than one attribute:

```
$('#redactor').redactor({
    removeAttr:  [
        ['p', ['class', 'style']],
        ['span', 'style']
    ]
});
```

'removeAttr' and 'allowedAttr' cannot be used simultaneously.

allowedAttr

Set to 'false' by default.

This setting allows to set attributes that will not be removed from the code.

```
$('#redactor').redactor({
    allowedAttr:  [
        ['p', 'class'],
        ['span', 'style']
    ]
});
```

If you need to set more than one attribute:

```
$('#redactor').redactor({
    allowedAttr:  [
        ['p', 'class'],
        ['img', ['src', 'alt']]
    ]
});
```

'allowedAttr' and 'removeAttr' cannot be used simultaneously.

removeEmpty

By default set to remove empty <p> tags.

You can set your custom array of tags that will be removed if empty.

```
$('#redactor').redactor({
    removeEmpty: ['strong', 'em', 'span', 'p']
});
```

All empty tags will always be removed on paste regardless of this setting.

removeWithoutAttr

Use this setting to set an array of tags, that you'd like to have removed if they have no attributes.

```
$('#redactor').redactor({
    removeWithoutAttr: ['strong', 'em', 'span']
});
```

By default, set to `['span']` — all span tags without attributes to be removed.
Set to an empty array to prevent any tags without attributes from being removed: `removeWithoutAttr: []`

Links

linkNofollow

When set to 'true', all links inside of Redactor will get a 'nofollow' attribute. This attribute restricts search engines indexing for these links.

```
$('#redactor').redactor({
   linkNofollow: true
});
```

linkSize

Set to '50' characters by default.

This setting allows to automatically truncate link text.

```
$('#redactor').redactor({
   linkSize: 100
});
```

linkProtocol

Set to 'http' by default.

Can be set to 'https' Redactor is required to use this protocol for newly created links.

```
$('#redactor').redactor({
   linkProtocol: 'https'
});
```

linkTooltip

Set to 'true' by default.

Shows link tooltip with Edit and Unlink buttons on click.

```
$('#redactor').redactor({
   linkTooltip: false
});
```

convertLinks

Set to 'true' by default.

This setting turns on and off automatic converting text to link on Return key or on paste.

```
$('#redactor').redactor({
   convertLinks: false
```

```
});
```

convertUrlLinks

This setting turns off converting text (for instance, www.imperavi.com) to link.

```
$('#redactor').redactor({
   convertUrlLinks: false
});
```

convertVideoLinks

This setting turns off converting YouTube and Vimeo links into embedded video

```
$('#redactor').redactor({
   convertVideoLinks: false
});
```

convertImageLinks

This setting turns off converting text (for instance, http://site.com/image.jpg) to an <img> tag.

```
$('#redactor').redactor({
   convertImageLinks: false
});
```

Shortcuts

shortcuts

Set to default keyboard shortcuts and corresponding Redactor functions by default.

Shortcuts can be turned off:

```
$('#redactor').redactor({
   shortcuts: false
});
```

Keep in mind, that turning 'shortcuts' off will also disable Tab key, however, 'tabAsSpaces' will still function.

shortcutsAdd

This setting allows to add custom keyboard shortcuts or replace default shortcuts:

```
$('#redactor').redactor({
    shortcutsAdd:
    {
        'ctrl+t, meta+t': { func: 'block.format', params: ['blockquote'] },
```

```
        'ctrl+shift+0': { func: 'block.format', params: ['p'] }
    }
});
```

The command in this example is one of Redactor functions and the attribute is an array of parameters. Parameters are optional in case Redactor function doesn't require parameters.

Images

imageUpload

Sets URL path to the image upload script.

```
$('#redactor').redactor({
    imageUpload: '/myadmin/images/upload/'
});
```

Image file will be transmitted in FILES array, for example, in PHP:

```
$_FILES['file']
```

By default, variable name is 'file', but this can be changed using imageUploadParam.

Image script must return a JSON string in following format in order for an image to be inserted into Redactor:

```
{ "filelink": "/images/img.jpg" }
```

In PHP, such string can be created like this:

```
$array = array(
    'filelink' => '/tmp/images/img.jpg'
);

echo stripslashes(json_encode($array));
```

imageUploadParam

This setting allows to change default name ('file') for imageUpload variable.

```
$('#redactor').redactor({
    imageUpload: '/myadmin/images/upload/',
    imageUploadParam: 'my-name'
});
```

In this example, imageUpload variable name will be 'my-name' and it can be retrieved like this (in PHP):

```
1. $_FILES['my-name']
```

dragImageUpload

Set to 'true' by default.

Turns off drag and drop image uploads.

```
$('#redactor').redactor({
    dragImageUpload: false
});
```

imageEditable

Set to 'true' by default.

Turns on image editing (opens in modal window)

```
$('#redactor').redactor({
   imageEditable: false
});
```

When set to true, user can launch image editing by clicking on an image. 'Edit' button will appear, and clicking on this button will launch a modal window with editable image parameters. When set to 'false', clicking on an image will do nothing.

Regardless of imageEditable, users can still resize an image visually by dragging its bottom right corner. This can be turned off by setting 'imageResizable' to 'false'.

imageResizable

Set to 'true' by default.

Turns on visual manual image resizing.

```
$('#redactor').redactor({
   imageResizable: false
});
```

imageLink

Set to 'true' by default.

Turns on the ability to add a link to an image via edit modal window.

```
$('#redactor').redactor({
   imageLink: false
});
```

imagePosition

Set to 'true' by default.

This setting allows to set image position (float alignment) in relation to the text.

```
$('#redactor').redactor({
   imagePosition: false
});
```

imageFloatMargin

Set to '10px' by default.

This setting sets margins for image float alignment.

```
$('#redactor').redactor({
   imageFloatMargin: '20px'
});
```

uploadImageFields

This setting allows to pass additional parameters with image upload. Parameters will be passed using POST method.

```
$('#redactor').redactor({
    uploadImageFields: {
        'field1': '#field1'
    }
});
```

In this example, value of field1 will be taken from an input with #field1 id. This input must be on the same page. Here's how you can retrieve this field value from POST array (in PHP):

```
echo $_POST['field1'];
```

Field value can be set dynamically:

```
$('#redactor').redactor({
    uploadImageFields: {
        'field2': '12345'
    }
});
```

In this example, value is set explicitly. Here's how you can retrieve this field value from POST array (in PHP):

```
echo $_POST['field2']; // print '12345'
```

For more convenient data and parameters management for uploads using plugins, we've introduced following API methods:

- upload.clearImageFields
- upload.addImageFields
- upload.removeImageFields

Files

fileUpload

Sets path URL to the file upload script.

```
$('#redactor').redactor({
    fileUpload: '/myadmin/files/upload/'
});
```

File will be transmitted in FILES array, for example, in PHP:

```
$_FILES['file']
```

By default, variable name is 'file', but this can be changed using fileUploadParam.

File script must return a JSON string in following format in order for a file to be inserted into Redactor as a link:

```
{
    "filelink": "/files/file.pdf",
    “filename”: "My PDF file"
}
```

Both filelink and filename parameters are required.

In PHP, such string can be created like this:

```
$array = array(
    "filelink" => "/tmp/files/file.pdf",
    "filename": "My PDF file"
);

echo stripslashes(json_encode($array));
```

fileUploadParam

This setting allows to change default name ('file') for fileUpload variable.

```
$('#redactor').redactor({
    fileUpload: '/myadmin/files/upload/',
```

```
    fileUploadParam: 'my-name'
});
```

In this example, imageUpload variable name will be 'my-name' and it can be retrieved like this (in PHP):

```
$_FILES['my-name']
```

dragFileUpload

Set to 'true' by default.

Turns off the ability to upload files using drag and drop:

```
$('#redactor').redactor({
    dragFileUpload: false
});
```

uploadFileFields

This setting allows to pass additional parameters with uploaded file. Parameters will be passed using POST method.

For example:

```
$('#redactor').redactor({
    uploadFileFields: {
        'field1': '#field1'
    }
});
```

In this example, value of field1 will be taken from an input with #field1 id. This input must be on the same page. Here's how you can retrieve this field value from POST array (in PHP):

```
echo $_POST['field1'];
```

Field value can be set dynamically:

```
$('#redactor').redactor({
    uploadFileFields: {
        'field2': '12345'
    }
});
```

In this example, value is set explicitly. Here's how you can retrieve this field value from POST array (in PHP):

```
echo $_POST['field2']; // print '12345'
```

For more convenient data and parameters management for uploads using plugins, we've introduced following API methods:

- upload.clearImageFields
- upload.addImageFields
- upload.removeImageFields

Formatting

formatting

By default set to:

```
['p', 'blockquote', 'pre', 'h1', 'h2', 'h3', 'h4', 'h5']
```

This setting allows to adjust a list of formatting tags in the default formatting dropdown. For example, following settings will only show 'Normal', 'Blockquote' and 'Header 2' in the formatting dropdown:

```
$('#redactor').redactor({
    formatting: ['p', 'blockquote', 'h2']
});
```

You can customize this by adding your own tags and CSS classes by setting formattingAdd.

formattingAdd

Set to 'false' by default.

This setting allows to select tags and styles for the formatting dropdown.

`formattingAdd` can only be applied to p, pre, blockquote and header tags.
Each formatting tag gets a CSS class that allows to customize style of each element. The class format is this:

```
.redactor-dropdown .redactor-formatting-p
.redactor-dropdown .redactor-formatting-blockquote
.redactor-dropdown .redactor-formatting-pre
.redactor-dropdown .redactor-formatting-h1
// etc
```

You can add your own tags with custom styles and classes:

```
$('#redactor').redactor({
    formattingAdd: [
    {
        tag: 'p',
        title: 'Red Block',
```

```
        class: 'red'
    },
    {
        tag: 'p',
        title: 'Blue Styled Block',
        class: 'blue-styled'
    },
    {
        tag: 'p',
        title: 'P Attr Title',
        attr: {
            name: 'title',
            value: 'Hello World!'
        },
        class: 'p-attr-title'
    },
    {
        tag: 'p',
        title: 'P Data Set',
        data: {
            name: 'data-name',
            value: 'true'
        },
        class: 'p-data-set'
    },
    {
        tag: 'span',
        title: 'Big Red',
        style: 'font-size: 20px; color: red;',
        class: 'big-red'
    },
    {
        tag: 'span',
        title: 'Font Size 20px',
        style: 'font-size: 20px;',
        class: 'font-size-20'
    },
    {
        tag: 'span',
        title: 'Font Georgia',
        style: 'font-family: Georgia;',
        class: 'font-family-georgia'
    },
    {
        tag: 'code',
        title: 'Code'
    },
    {
        tag: 'mark',
        title: 'Marked'
    },
    {
        tag: 'span',
```

```
        title: 'Marked Span',
        class: 'marked-span'
    },
    {
        tag: 'ins',
        title: 'Inserted',
        class: 'inserted-from-editor'
    }]
});
```

If you need to clear all styles and classes before adding your own class or style, you can set `clear: true`

```
$('#redactor').redactor({
    formattingAdd: [
    {
        tag: 'p',
        title: 'Red Block',
        class: 'red',
        clear: true
    }
});
```

For tags with styles 'class' is required. It serves as an identifier of a style and provides correct formatting.

In this example, the following classes being automatically created for styling of dropdown elements:

```
.redactor-dropdown .redactor-formatting-span-big-red
.redactor-dropdown .redactor-formatting-code
.redactor-dropdown .redactor-formatting-mark
.redactor-dropdown .redactor-formatting-span-marked-span
.redactor-dropdown .redactor-formatting-ins-inserted-from-editor
// etc
```

FormattingAdd can also be used with various API methods. For example, you could add "Clear Format" option to your formatting dropdown by using API method `inline.removeFormat`:

```
$('#redactor').redactor({
    formattingAdd: [{
        title: 'Clear Format',
        func: 'inline.removeFormat'
    }]
});
```

Codemirror

codemirror

Native integration with CodeMirror syntax highlighting. Users can switch to source code and have their code syntax nicely highlighted. There's a whole lot of settings available, see CodeMirror Manual for details.

```
$('#redactor').redactor({
    codemirror: true
});
```