Installation

Installing CKEditor is easy. Choose the appropriate procedure (fresh install or upgrade) and follow the steps described below.

Contents

Fresh Installation

To install CKEditor for the first time, proceed in the following way:

1. **Download** the latest version from the CKEditor website: http://ckeditor.com/download
2. **Extract** (decompress) the downloaded archive to a directory called `ckeditor` in the root of your website.

You can place the files in any path of your website. The `ckeditor` directory is the default one.

Upgrade

To upgrade an existing CKEditor installation, proceed in the following way:

1. **Rename** your old editor folder to a backup folder, for example `ckeditor_old`.
2. **Download** the latest version from the CKEditor website: http://ckeditor.com/download
3. **Extract** (decompress) the downloaded archive to the original editor directory, for example `ckeditor`.
4. **Copy** all configuration files that you have changed from the backup directory to their corresponding position in the new directory. These could include (but not limited to) the following files:
    1. `config.js`
    2. `contents.css`
    3. `styles.js`

Verification of the Installation

CKEditor comes with sample pages that can be used to verify that the installation proceeded properly. In order to see whether the editor is working, take a look at the `samples` directory.

To test your installation, call the following page at your website:
`http://<your site>/<CKEditor installation path>/samples/index.html`

For example:
`http://www.example.com/ckeditor/samples/index.html`

Loading CKEditor

CKEditor is a JavaScript application. To load it, you need to include a single file reference in your page. If you have installed CKEditor in the `ckeditor` directory in root of your website, you need to insert the following code fragment into the `<head>` section of your page:

```
<head>
    ...
    <script src="/ckeditor/ckeditor.js"></script>
</head>
```

When the above file is loaded, the CKEditor JavaScript API is ready to be used.

When adding CKEditor to your web pages, use the original file name (`ckeditor.js`). If you want to use a different file name, or even merge the CKEditor script into another JavaScript file, refer to the Specifying the Editor Path section of the Developer's Guide first.

Creating Editors

Now that the CKEditor JavaScript API is available in the page, you can use it create editors. For that, there are two different options available:

- Framed Editing: the most common way to use the editor, usually represented by a toolbar and a editing area placed on a specific position in the page.
- Inline Editing: to be used on pages that look like the final page. Editing is enabled directly on HTML elements through the HTML5 `contenteditable` attribute. The editor toolbar automatically appears, floating in the page.

Just click on your preferred option to have more information.

Framed Editing

Framed Editing is the most common way to use CKEditor. It is usually represented by a toolbar and a editing area placed on a specific position in the page.

After loading the CKEditor script, you'll be ready to create your editors.

Contents

Creating a Framed Editor

On Framed Editing, CKEditor works just like a `textarea` element in your page. The editor offers a user interface to easily write, format, and work with rich text, but the same thing could be achieved (though not that easily) with a `<textarea>` element, requiring the user to type HTML code inside.

As a matter of fact, CKEditor uses the `textarea` to transfer its data to the server. The `textarea` element is invisible to the end user. In order to create an editor instance, you must first add a `<textarea>` element to the source code of your HTML page:

```
<textarea name="editor1">&lt;p&gt;Initial value.&lt;/p&gt;</textarea>
```

Note that if you want to load data into the editor, for example from a database, you need to put that data inside the `<textarea>` element, just like the HTML-encoded <p> element in the example above. In this case the textarea element was named `editor1`. This name can be used in the server-side code later, when receiving the posted data.

After the textarea element is inserted, you can use the CKEditor JavaScript API to replace this HTML element with an editor instance. A simple CKEDITOR.replace method call is needed for that:

```
<script>
    CKEDITOR.replace( 'editor1' );
</script>
```

This script block must be included at any point after the `<textarea>` tag in the source code of the page. You can also call the replace function inside the <head> section, but in this case you will need to listen for the `window.onload` event:

```
<script>
    window.onload = function() {
        CKEDITOR.replace( 'editor1' );
    };
</script>
```

Saving the Editor Data

As stated above, the editor works just like a `<textarea>` field. This means that when submitting a form containing an editor instance, its data will be simply posted, using the `<textarea>` element name as the key to retrieve it.

For example, following the above example, we could create the following PHP code:

```php
<?php
    $editor_data = $_POST[ 'editor1' ];
?>
```

Client-Side Data Handling

Some applications (like those based on Ajax) need to handle all data on the client side, sending it to the server using their specific methods. If this is the case, it is enough to use the CKEditor JavaScript API to easily retrieve the editor instance data. In order to do this, you can use the getData method:

```
<script>
    var editor_data = CKEDITOR.instances.editor1.getData();
</script>
```

Complete Sample

To insert a CKEditor instance, you can use the following sample that creates a basic page containing a form with a `textarea` element that is replaced with CKEditor.

```
<html>
<head>
    <title>CKEditor Sample</title>
    <script src="/ckeditor/ckeditor.js"></script>
</head>
<body>
    <form method="post">
        <p>
            My Editor:<br>
            <textarea name="editor1">&lt;p&gt;Initial value.&lt;/p&gt;</textarea>
            <script>
                CKEDITOR.replace( 'editor1' );
            </script>
        </p>
        <p>
            <input type="submit">
        </p>
    </form>
</body>
</html>
```

Inline Editing

Inline Editing is a new technology designed to make it possible edit pages that look just like the final page. It is a total WYSIWYG experience, because not only the edited content will look like the final, but also the page and the context where the content is placed.

Contents

Enabling Inline Editing

Inline Editing is enabled directly on HTML elements through the HTML5 `contenteditable` attribute.

For example, supposed that you want to make a `<div>` element editable. It's enough to do so:

```
<div id="editable" contenteditable="true">
    <h1>Inline Editing in Action!</h1>
    <p>The div element that holds this text is now editable.
</div>
```

It is also possible to enable editing by code, by calling CKEDITOR.inline:

```
<div id="editable" contenteditable="true">
    <h1>Inline Editing in Action!</h1>
    <p>The div element that holds this text is now editable.
</div>
<script>
    // Turn off automatic editor creation first.
    CKEDITOR.disableAutoInline = true;
    CKEDITOR.inline( 'editable' );
</script>
```

When clicking inside the above `<div>` contents, the CKEditor toolbar will appear.

Retrieving the Editor Data

Unlike Framed Editing, the data edited with CKEditor is not placed inside a `<textarea>` when using Inline Editing. It is instead present directly in the page DOM. Because of this, it is your application job to retrieve the data and manipulate it for saving.

To retrieve the editor data, simply call the CKEDITOR.editor.getData method of the editor instance. For the above examples, tho would look like the following:

```
<script>
    var data = CKEDITOR.instances.editable.getData();

    // Your code to save "data", usually though Ajax.
</script>
```

Note that the original `<div>` id has been passed to the CKEDITOR.instances object, to retrieve the editor instance.

Setting CKEditor Configuration

CKEditor comes with a rich set of configuration options that make it possible to customize its appearance, features, and behavior. The main configuration file is named `config.js`. This file can be found in the root of the CKEditor installation folder.

Contents

Available Configuration Options

All available configuration options can be found in the API documentation. Refer to the CKEDITOR.config object definition.

Defining Configuration In-Page

The best way to set the CKEditor configuration is in-page, when creating editor instances. This method lets you avoid modifying the original distribution files in the CKEditor installation folder, making the upgrade task easier.

In-page settings can be passed to any of the editor instance creation functions, namely CKEDITOR.replace and CKEDITOR.appendTo. For example:

```
CKEDITOR.replace( 'editor1', {
    toolbar: 'Basic',
    uiColor: '#9AB8F3'
});
```

Note that the configuration options are passed through a literal object definition (starting with a "`{`" symbol and ending with a "`}`" symbol). Because of this the proper syntax for each option is `('configuration name') : ('configuration value')`. Be sure to not use the "equal" character (=) in place of the colon character (`:`).

Using the config.js File

CKEditor settings can also be configured by using the `config.js` file. By default this file is mostly empty. To change CKEditor configuration, add the settings that you want to modify to the `config.js` file. For example:

```
CKEDITOR.editorConfig = function( config ) {
    config.language = 'fr';
    config.uiColor = '#AADC6E';
};
```

In order to apply the configuration settings, the CKEDITOR.editorConfig function must always be defined. The `config.js` file will be executed in the scope of your page, so you can also make references to variables defined in-page or even in other JavaScript files.

Using a Custom Configuration File

Using a custom configuration file is another recommended method of setting CKEditor configuration. Instead of using the default `config.js` file, you can create a copy of that file anywhere in your website and simply point the

editor instances to load it. The advantage of this approach is that in this way you can avoid changing the original file, which makes it easier to upgrade CKEditor later by simply overwriting all files.

Suppose you copied the `config.js` file to a folder named `custom` in the root of your website. You also renamed the file to `ckeditor_config.js`. At that point it is enough to only set the customConfig configuration option when creating the editor instances to use the customized settings defined in the file. For example:

```
CKEDITOR.replace( 'editor1', {
    customConfig: '/custom/ckeditor_config.js'
});
```

The custom configuration file must look just like the default `config.js` file.

Configuration Loading Order

You are not required to only use one of the above configuration options. The methods described above can be mixed and the configuration will be loaded properly. The following list presents the configuration loading order used when creating an editor instance:

- An editor instance is created. At this point all its default configuration options are set.
- If the `customConfig` setting has been set "in-page", that file is loaded, otherwise the default `config.js` file is loaded. All settings in the custom configuration file override current instance settings.
- If the settings loaded in step 2 also define a new `customConfig` value, another custom configuration file is loaded and its settings override current instance settings. This happens recursively for all files until no `customConfig` is defined.
- Finally the settings defined "in-page" override current instance settings (except `customConfig`, which has been used in step 1).

Avoiding Loading External Settings Files

It is also possible to completely avoid loading an external configuration file, reducing the number of files loaded. To do that, you need to set the CKEDITOR.config.customConfig setting to an empty string. For example:

```
CKEDITOR.replace( 'editor1', {
    customConfig: ''
});
```

This setting is definitely recommended, if you are not setting the configuration in the `config.js` file nor a custom configuration file.

Toolbar Customization

While CKEditor is a full-featured WYSIWYG editor, not all of its options may be needed in all cases. Because of this, toolbar customization is one of the most common requirements.

There are two ways to configure the toolbar to match your needs:

- Toolbar Groups Configuration
- "Item by Item" Configuration

Contents

Toolbar Groups Configuration

CKEditor 4 introduced a new concept for toolbar organization which is based on "grouping" instead of the traditional "item by item positioning" way.

Grouping configuration is defined by the toolbarGroups setting. The following the the configuration used by the "standard" distribution of CKEditor:

```
config.toolbarGroups = [
    { name: 'clipboard',   groups: [ 'clipboard', 'undo' ] },
    { name: 'editing',     groups: [ 'find', 'selection', 'spellchecker' ] },
    { name: 'links' },
    { name: 'insert' },
    { name: 'forms' },
    { name: 'tools' },
    { name: 'document',    groups: [ 'mode', 'document', 'doctools' ] },
    { name: 'others' },
    '/',
    { name: 'basicstyles', groups: [ 'basicstyles', 'cleanup' ] },
    { name: 'paragraph',   groups: [ 'list', 'indent', 'blocks', 'align' ] },
    { name: 'styles' },
    { name: 'colors' },
    { name: 'about' }
];
```

It is a list (Array) of objects, each one with a "name" (e.g "clipboard" or "links") and a optional "sub-groups" list.

Changing the Groups Order

You can easily customize the groups ordering and position by simply changing the above configuration.

You can force row-breaks in the toolbar by adding `'/'` into the list, just like you could see above.

Note that there are unused groups in the above configuration. This is "by design" (see "The Benefits of Group Configuration").

The Benefits of Group Configuration

The most important benefit of toolbar grouping configuration over the "item by item" configuration is: **automation**.

It is now possible for plugin developers to define into which group their plugins should add buttons in the toolbar. For example, the "image" plugin, includes its button into the "insert" group, while the undo and redo buttons go into

the "undo" sub-group.

While not mandatory, having all groups and sub-groups configured (including not used ones) is recommended because at any moment in the future, when a new plugin gets installed, its button will automatically appear in the toolbar without further configuration requirements.

The Drawbacks of Group Configuration

The most evident problem with grouping configuration its that it is not possible to control precisely where each item is placed in the toolbar. It is the plugin itself to decide it.

"Item by Item" Configuration

Other than the grouping configuration, it is also possible to have more control over every single element in the toolbar by defining their precise position. That is done by configuring a "toolbar definition".

A toolbar definition is a JavaScript array that contains the elements to be displayed in all **toolbar rows** available in the editor. The following is an example:

```
config.toolbar = [
    { name: 'document', items: [ 'Source', '-', 'NewPage', 'Preview', '-',
'Templates' ] },
    { name: 'clipboard', items: [ 'Cut', 'Copy', 'Paste', 'PasteText',
'PasteFromWord', '-', 'Undo', 'Redo' ] },
    '/',
    { name: 'basicstyles', items: [ 'Bold', 'Italic' ] }
];
```

Here every toolbar group is given a name and their precise list of items is defined.

The above can also be achieved with a simpler syntax (see "Accessibility Concerns" later on):

```
config.toolbar = [
    [ 'Source', '-', 'NewPage', 'Preview', '-', 'Templates' ],
    [ 'Cut', 'Copy', 'Paste', 'PasteText', 'PasteFromWord', '-', 'Undo', 'Redo' ],
    '/',
    [ 'Bold', 'Italic' ]
];
```

Items separator can be included by adding `'-'` (dash) to the list of items, as seen above.

You can force row-breaks in the toolbar by adding `'/'` between groups. They can be used to force a break at the point where they were placed, rendering the next toolbar group in a new row.

The Benefits of "Item by Item" configuration

The most evident benefit of this kind of configuration is that the position of every single item in the toolbar is under control.

The drawbacks of "Item by Item" configuration

The biggest problem it that there will be no automation when new plugins get installed. This means that, if any new plugin get into your editor, you'll have to manually change your configurations, to include the plugin buttons at any desired position.

Accessibility Concerns

The "name" used on every toolbar group will be used by assistive technology such as screen readers. That name will be used by CKEditor too lookup for the "readable" name of each toolbar group in the editor language files (the `toolbarGroups` entries).

Screen readers will announce each of the toolbar groups by using either their readable name, if available, or their defined `name` attribute.

Styles

The Styles Combo plugin adds the a combo to the CKEditor toolbar, containing a list of styles. This list makes it easy to apply customized styles and semantic values to content created in the editor.

The entries available in the combo drop-down list can be easily customized to suit your needs.

Contents

Defining Styles

The styles definition is a JavaScript array which is registered by calling the CKEDITOR.stylesSet.add function. A unique name must be assigned to your style definition, so you can later configure each editor instance to load it. This method lets you have a single style definition which is shared by several CKEditor instances present on the page.

The following code shows how to register a sample style definition.

```
CKEDITOR.stylesSet.add( 'my_styles', [
    // Block-level styles
    { name: 'Blue Title', element: 'h2', styles: { 'color': 'Blue' } },
    { name: 'Red Title' , element: 'h3', styles: { 'color': 'Red' } },

    // Inline styles
    { name: 'CSS Style', element: 'span', attributes: { 'class': 'my_style' } },
    { name: 'Marker: Yellow', element: 'span', styles: { 'background-color':
'Yellow' } }
]);
```

The definition registration like the one above can be placed inline in the page source, or can live in an external file which is loaded "on demand", when needed only (see below).

When the definitions are ready, you must instruct the editor to apply the newly registered styles by using the stylesSet setting. This may be set in the `config.js` file, for example:

```
config.stylesSet = 'my_styles';
```

Using an External Styles Definition File

The style definition registration call can be included in an external JavaScript file. By default, CKEditor load the style definition from `styles.js` file included in its installation folder.

Your style definition file can be saved in any place of your website (or somewhere in the Internet). You must, however, know the URL required to reach it. For example, you can save the file at the root of your website, and then call it as `/styles.js`, or place it anywhere else, and refer to it using its full URL, like `http://www.example.com/styles.js`.

At that point, change the `stylesSet` setting to point the editor to your file:

```
config.stylesSet = 'my_styles:/styles.js';
```

```
OR

config.stylesSet = 'my_styles:http://www.example.com/styles.js';
```

The syntax for the style definition setting is always: `style definition name : file URL`.

Note that you must use the unique name you have used to register the style definition in the file.

Style Rules

The entries inside a style definition are called "style rules". Each rule defines the display name for a single style as well as the element, attributes, and CSS styles to be used for it. The following is the generic representation for it:

```
{
    name: 'Name displayed in the Styles drop-down list',
    element: 'HTML element name (for example "span")',
    styles: {
        'css-style1': 'desired value',
        'css-style2': 'desired value',
        ...
    }
    attributes: {
        'attribute-name1': 'desired value',
        'attribute-name2': 'desired value',
        ...
    }
}
```

The `name` and `element` values are required, while other values are optional.

Style Types

There are three kinds of style types, each one related to the element used in the style rule:

- **Block-level styles** – applied to the text blocks (paragraphs) as a whole, not limited to the text selections. These apply to the following elements: `address`, `div`, `h1`, `h2`, `h3`, `h4`, `h5`, `h6`, `p`, and `pre`.
- **Object styles** – applied to special selectable objects (not textual), whenever such selection is supported by the browser. These apply to the following elements: `a`, `embed`, `hr`, `img`, `li`, `object`, `ol`, `table`, `td`, `tr` and `ul`.
- **Inline styles** – applied to text selections for style rules using elements not defined in other style types.

Stylesheet Parser Plugin

Another simplified method exists of customizing the styles for the document created in CKEditor and populating the drop-down list with style definitions added in an external CSS stylesheet file. The Stylesheet Parser plugin lets you use your existing CSS styles without the need to define the styles specifically for CKEditor in the format presented above.

Having the Stylesheet Parser installed, you then need to supply the location of the CSS file that contains your style definitions by using the contentsCss configuration setting:

```
config.contentsCss = 'sample_CSS_file.css';
```

Finally, if you want to skip loading the styles that are used in CKEditor by default, you may set `stylesSet` to an

empty value:

```
config.stylesSet = [];
```

This solution lets you configure the editor to use existing CSS stylesheet rules without the need to create separate style definitions for CKEditor. On the other hand, the previously used approach offers more control over which styles are available for the users, so both solutions can be employed interchangeably, according to your needs.

Choosing the CSS Selectors

The plugin can be fine-tuned to only take into account the CSS selectors that match the stylesheetParser_validSelectors configuration value. The default regular expression accepts all CSS rules in a form of `element.class`, but you can modify it to refer to a limited set of elements, like in the example below.

```
// Only add rules for <p> and <span> elements.
config.stylesheetParser_validSelectors = /\^(p|span)\.\w+/;
```

Limiting the CSS Selectors

You can also customize by setting the stylesheetParser_skipSelectors configuration value. The plugin will then ignore the CSS rules that match the regular expression and will not display them in the drop-down list nor use them to output the document content. The default value excludes all rules for the `<body>` element as well as classes defined for no specific element, but you can modify it to ignore a wider set of elements, like in the example below.

```
// Ignore rules for <body> and <caption> elements, classes starting with "high",
and any class defined for no specific element.
config.stylesheetParser_skipSelectors = /(^body\.|^caption\.|\.high|^\.)/i;
```

Introduction

Note: Advanced Content Filter was introduced in **CKEditor 4.1**.

Contents

What is Advanced Content Filter (ACF)?

ACF is a highly configurable CKEditor core feature **available since CKEditor 4.1**. It limits and adapts input data (HTML code added in source mode or by the editor.setData method, pasted HTML code, etc.) so it matches the editor configuration in the best possible way. It may also deactivate features which generate HTML code that is not allowed by the configuration.

Advanced Content Filter works in two modes:

- **automatic** – the filter is configured by editor features (like plugins, buttons, and commands) that are enabled with configuration options such as CKEDITOR.config.plugins, CKEDITOR.config.extraPlugins, and CKEDITOR.config.toolbar,
- **custom** – the filter is configured by the CKEDITOR.config.allowedContent option and only features that match this setting are activated.

In both modes it is possible to extend the filter configuration by using the CKEDITOR.config.extraAllowedContent setting.

If you want to disable Advanced Content Filter, set CKEDITOR.config.allowedContent to `true`. All available editor features will be activated and input data will not be filtered.

Automatic Mode

Advanced Content Filter works in automatic mode when the CKEDITOR.config.allowedContent setting is not provided. During editor initialization, editor features add their rules to the filter. As a result, only the content that may be edited using currently loaded features is allowed, and all the rest is filtered out.

The following example might make it easier to understand the automatic ACF mode.

1. Open the `datafiltering.html` sample from the Full or Standard CKEditor package (the set of features offered by the Basic package is too limited).
2. Check *editor 1*. It uses the default configuration, so all buttons, keystrokes, or styles available in your package are activated and editor contents are identical to what was originally loaded (except a small detail in the Standard package — since it does not contain the Justify plugin, the footer is not aligned to the right).
3. Now check *editor 4*. You can see that many plugins and buttons were removed by the CKEDITOR.config.removePlugins and CKEDITOR.config.removeButtons settings; the CKEDITOR.config.format_tags list was trimmed down, too. Configuration changes are automatically reflected in editor contents — there is no Image toolbar button, so there is no image in the contents; there is no Table plugin, so the table added in the original contents was removed, too. You can see how the editor cleans up pasted content or HTML code set in the source mode.

If you want to configure the editor to work in automatic mode, but need to enable additional HTML tags, attributes, styles, or classes, use the CKEDITOR.config.extraAllowedContent configuration option.

Custom Mode

Advanced Content Filter works in custom mode when the CKEDITOR.config.allowedContent setting is defined.

This configuration option tells the filter which HTML elements, attributes, styles, and classes are allowed. Based on defined rules (called *Allowed Content Rules*) the editor filters input data and decides which features can be activated.

Allowed Content Rules may be set in two different formats: the compact **string format** and the more powerful **object format**. Read about Allowed Content Rules in the Allowed Content Rules article.

The following example might make it easier to understand the custom ACF mode.

1. Open the `datafiltering.html` sample from the Full or Standard CKEditor package (the set of features offered by the Basic package is too limited).
2. Check *editor 1*. It uses the default configuration, so all buttons, keystrokes, or styles available in your package are activated and editor contents are identical to what was originally loaded (except a small detail in the Standard package — since it does not contain the Justify plugin, the footer is not aligned to the right).
3. Now check *editor 2*. The CKEDITOR.config.allowedContent option defines Allowed Content Rules using the string format.

   Note that since the rules do not allow the `'s'` element, the Strike Through button was removed from the toolbar and the strike-through formatting was removed from the text. The same happened for example with the Horizontal Line, Subscript, or Superscript features.

   See also that the Styles and Format drop-down lists only contain the items which are defined in the Allowed Content Rules.

   What is more, options available in some dialog windows are limited, too. For example the Image dialog window contains only the URL, Alternative Text, Width, and Height values, because only these attributes were defined in CKEDITOR.config.allowedContent.

4. Additionally, *editor 3* is configured by using a different set of rules defined in the object format.

Content Transformations

Advanced Content Filter not only removes disallowed HTML elements, their classes, styles, and attributes, but it also tries to unify input data by transforming one form of an element to another, preferred form.

Consider the Bold feature. In HTML code it may be represented by `<strong>`, `<b>`, or even a `<span style="font-weight:700|800|900|bold">` element. Suppose that the CKEDITOR.config.allowedContent setting contains only a rule for the `<b>` element. Does this mean that when pasting the `<strong>` or `<span>` element they will be removed?

No. The editor will transform all of them to `<b>` elements. As a result the editor will contain only `<b>` elements and the the visual form of pasted content will be preserved. Exactly the same will happen if you leave the default CKEDITOR.config.allowedContent value (in automatic mode) — all Bold feature forms will be unified to the preferred `<strong>` form.

Suppose that the `'img[!src,alt,width,height]'` setting (`<img>` tag with required `src` and three optional attributes) was added to Allowed Content Rules. Image size should be set with attributes, so for example a pasted image whose size was set with styles will be transformed to an image with attributes (note that it will not be possible in all scenarios — only pixel-based size can be transformed).

The content transformation feature is fully automatic and there is no need to configure it. The only thing you have to do is set the CKEDITOR.config.allowedContent option or use the default value (automatic mode).

Currently, we have defined content transformations for only a handful of editor features, but their number will increase in future releases.

Allowed Content Rules

Note: Advanced Content Filter was introduced in **CKEditor 4.1**.

Contents

Introduction

Allowed Content Rules define which HTML elements, attributes, styles, and classes are allowed. When configuring CKEditor you will be mostly interested in setting the CKEDITOR.config.allowedContent option. Plugin developers will also need to set CKEDITOR.feature.allowedContent properties which tell the editor what kind of content a feature allows in automatic mode.

Allowed Content Rule usually consists of four main parts:

- the **elements** that it allows,
- the **attributes** that these elements may have,
- the **styles** that these elements may have,
- the **classes** that these elements may have.

Note: Instead of writing "attributes, styles, and classes", "**properties**" will be used as a shorthand.

Multiple rules may exist for one element and one element may be included in numerous element lists. For example each of the rules may allow another set of element properties.

Rules are applied one by one. Initially the element being filtered is invalid and all its properties are rejected. The first rule applied to the element validates it (it will not be removed) and that rule may accept some element properties. Another rule may cause the editor to accept further element properties. Therefore:

- If there are no rules for an element it is removed.
- It is possible to accept an element, but reject all its properties which will then be removed.
- Once validated, an element or its property cannot be invalidated by another rule.

String Format

The string format is a compact notation for Allowed Content Rules, but it does not offer all features available in the object format. However, in most cases it should be sufficient.

Rule format:

```
elements [attributes]{styles}(classes)

Regexp pattern:
 <   elements   ><                    styles, attributes, and classes
>< separator >
/^([a-z0-9*\s]+)((?:\s*\{[!\w\-,\s\*]+\}\s*|\s*\[[!\w\-,\s\*]+\]\s*|\s*\([!\w\-
,\s\*]+\)\s*){0,3})(?:;\s*|$)/i,
```

Where:

- `elements` – a list of space-separated element names or an asterisk (`*`) character,
- `attributes` – a comma-separated list of attribute names or an asterisk (`*`) character,
- `styles` – a comma-separated list of style names or an asterisk (`*`) character,
- `classes` – a comma-separated list of classes or an asterisk (`*`) character.

Special characters:

- Asterisk used in the element list means: "This rule accepts the following properties for all elements, but not the elements themselves; there has to be another rule accepting these elements explicitly".
- Asterisk used in the property list means: "Accept all properties".
- Exclamation mark (`!`) used before an item name (e.g.: `[!href]`) in the property list means: "This property is required. If an element does not have it, this rule should not be applied to the element (so the element will not be validated by it)".

Examples:

```
// A rule accepting <p> and <h1> elements, but without any property.
p h1

// A rule accepting <p> and <h1> elements with optional "left" and "right"
classes.
// Note: Both elements may contain these classes, not only <h1>.
p h1(left,right)

// A rule accepting <p> and <h1> elements with all their attributes.
p h1[*]

// A rule accepting <a> only if it contains the "href" attribute.
a[!href]

// A rule accepting <img> with a required "src" attribute and an optional "alt"
attribute plus optional "width" and "height" styles.
img[alt,!src]{width,height}

// The same as above, because the order of properties and their lists is
irrelevant and white-spaces are ignored.
img { height, width } [ !src, alt ]
```

The Allowed Content Rules set may consist of many rules separated by semicolon (`;`) characters. Examples:

```
// Rules allowing:
// * <p> and <h1> elements with an optional "text-align" style,
// * <a> with a required "href" attribute,
// * <strong> and <em> elements,
// * <p> with an optional "tip" class (so <p> element may contain
//  a "text-align" style and a "tip" class at the same time).
p h1{text-align}; a[!href]; strong em; p(tip)

// Rules allowing:
// * <p> and <h1> elements with an optional "id" attribute,
// * <a> with a required "href" attribute and an optional "id" attribute.
p h1; a[!href]; *[id]
```

Debugging

In order to verify if Allowed Content Rules were parsed correctly, you can check the CKEDITOR.filter.allowedContent property of the CKEDITOR.editor.filter object.

```
var editor = CKEDITOR.replace( 'textarea_id', {
    allowedContent: 'a[!href]; ul; li{text-align}(someclass)'
} );
```

```
editor.on( 'instanceReady', function() {
    console.log( editor.filter.allowedContent );
} );

// This will log the following array:
// { elements: 'p br', ... } (default editor rules)
// { elements: 'a', attributes: '!href' }
// { elements: 'ul' }
// { elements: 'li', styles: 'text-align', classes: 'someclass' }
```

Object Format

For the convenience of the users Allowed Content Rules can be defined as a standard object literal so the following:

```
allowedContent: 'p h1{text-align}; a[!href]; strong em; p(tip)'
```

correspond with:

```
allowedContent: {
    'p h1': {
        styles: 'text-align'
    },
    a: {
        attributes: '!href'
    },
    'strong em': true,
    p: {
        classes: 'tip'
    }
}
```

With this approach, Allowed Content Rules can be dynamically generated by JavaScript or stored for any purposes in JSON data format. Note that keys of object literals **must be unique**, so:

```
allowedContent: {
    p: {
        styles: 'text-align'
    },
    p: {
        classes: 'tip'
    }
}
```

is an equivalent of:

```
allowedContent: 'p(tip)'
```

but never:

```
allowedContent: 'p{text-align}(tip)'
```

Output Formatting

CKEditor offers a powerful and flexible output formatting system. It gives developers full control over what the HTML code produced by the editor will look like. The system makes it possible to control all HTML tags and can give a different result for each one of them.

Contents

The HTML Writer

The HTML Writer plugin makes it possible to generate advanced formatted output with CKEditor.

The "writer" is used by the CKEDITOR.htmlDataProcessor class to write the output data. Therefore, the current writer for a specific editor instance can be retrieved with the editor.dataProcessor.writer property.

It is possible to configure several output formatting options by setting the writer properties. The following example summarizes the most used of them, giving their default values:

```
var writer = editor.dataProcessor.writer;

// The character sequence to use for every indentation step.
writer.indentationChars = '\t';

// The way to close self closing tags, like <br />.
writer.selfClosingEnd = ' />';

// The character sequence to be used for line breaks.
writer.lineBreakChars = '\n';

// The writing rules for the <p> tag.
writer.setRules( 'p', {
    // Indicates that this tag causes indentation on line breaks inside of it.
    indent: true,

    // Inserts a line break before the <p> opening tag.
    breakBeforeOpen: true,

    // Inserts a line break after the <p> opening tag.
    breakAfterOpen: true,

    // Inserts a line break before the </p> closing tag.
    breakBeforeClose: false,

    // Inserts a line break after the </p> closing tag.
    breakAfterClose: true
});
```

Setting Writer Rules

Because the writer is a property of each editor instance, and also due to its dependency on the writer plugin to be loaded, the best way to modify it is by listening to the CKEDITOR.instanceReady event, so it is safe to assume that

the CKEDITOR.editor.dataProcessor property will be loaded and ready for changes. The following code shows an example of this approach used when creating an editor instance:

```
CKEDITOR.replace( 'editor1', {
    on: {
        instanceReady: function( ev ) {
            // Output paragraphs as <p>Text</p>.
            this.dataProcessor.writer.setRules( 'p', {
                indent: false,
                breakBeforeOpen: true,
                breakAfterOpen: false,
                breakBeforeClose: false,
                breakAfterClose: true
            });
        }
    }
});
```

Another method is to use the CKEDITOR object, so all editor instances will be changed:

```
CKEDITOR.on( 'instanceReady', function( ev ) {
    // Ends self closing tags the HTML4 way, like <br>.
    ev.editor.dataProcessor.writer.selfClosingEnd = '>';
});
```

Spell Checking

CKEditor can be configured to use either native spell checking capabilities provided by the browser or to use an external spell checking web service.

Contents

The Native Spell Checker

Native spell check functionality is by default disabled in the editor, use disableNativeSpellChecker to enable it:

```
config.disableNativeSpellChecker = false;
```

You should be able to see the spelling underline in content after reloading the editor.

Note: If the context menu plugin is enabled, its necessary to hold the CTRL key when right-clicking misspelled words to see their suggestions.

Note: The spell check is not generically available for all browsers.

The SpellCheckAsYouType Plugin

The SpellCheckAsYouType (SCAYT) plugin provides inline spell checking, much like the native spell checker, well integrated with the CKEditor context menu.

It is provided by WebSpellChecker.net. It users their web-services, transferring the text to their servers and performing spell checking. This is a cross browser solution.

The WebSpellChecker Plugin

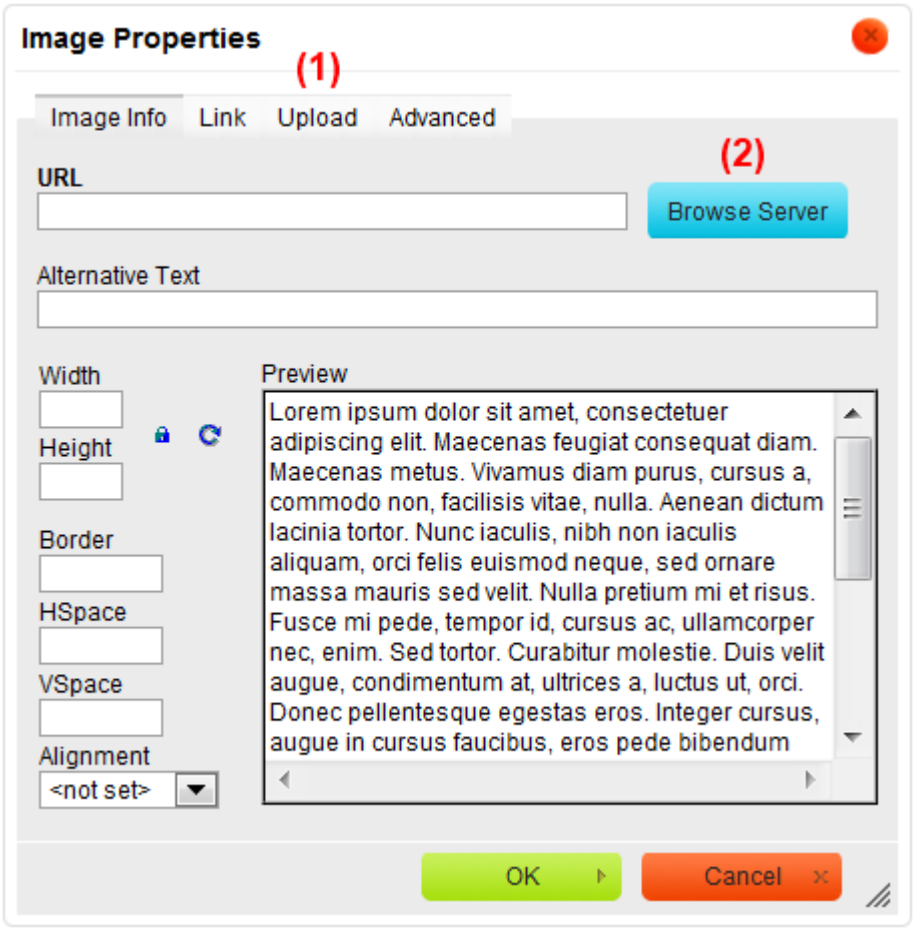
The WebSpellChecker plugin is another spell checking solution provided by WebSpellChecker.net that instead runs the check through a dialog windows, instead of marking misspelled words inline.

Integrate with File Browser

CKEditor can be easily integrated with an external file browser/uploader.

Once properly set up, all file browser features will automatically become available. This includes the **Upload** tab `(1)` in the **Link**, **Image**, and **Flash Properties** dialog windows as well as the **Browse Server** button `(2)`.

Contents



Basic Configuration

- The filebrowserBrowseUrl setting contains the location of an external file browser that should be launched when the **Browse Server** button is pressed.
- The filebrowserUploadUrl setting contains the location of a script that handles file uploads. If set, the **Upload** tab will appear in some dialog windows — the ones where such functionality is available, i.e. **Link**, **Image** and **Flash Properties**.

Example 1

The sample below shows basic configuration code that can be used to insert a CKEditor instance with the file browser configured.

```
CKEDITOR.replace( 'editor1', {
```

```
    filebrowserBrowseUrl: '/browser/browse.php',
    filebrowserUploadUrl: '/uploader/upload.php'
});
```

Example 2

It is also possible to set a separate URL for a selected dialog window by using the dialog window name in file browser settings: `filebrowserBrowseUrl` and `filebrowserUploadUrl`.

For example to set a special upload URL for the image dialog window, use the filebrowserImageUploadUrl property.

```
CKEDITOR.replace( 'editor1', {
    filebrowserBrowseUrl: '/browser/browse.php',
    filebrowserImageBrowseUrl: '/browser/browse.php?type=Images',
    filebrowserUploadUrl: '/uploader/upload.php',
    filebrowserImageUploadUrl: '/uploader/upload.php?type=Images'
});
```

In the example above, the `filebrowserBrowseUrl` and `filebrowserUploadUrl` settings will be used by default. In the **Image Properties** dialog window CKEditor will use the `filebrowserImageBrowseUrl` and `filebrowserImageUploadUrl` configuration settings instead.

File Browser Window Size

The default width of the file browser window in CKEditor is set to 80% of the screen width, while the default height is set to 70% of the screen height.

If for any reasons the default values are not suitable for you, you can adjust them to your needs by using the filebrowserWindowWidth to change the width and filebrowserWindowHeight to change the height of the window.

To specify the size of the file browser window in pixels, set it to a number (e.g. `"800"`). If you prefer to set the height and width of the window as a percentage value of the screen, do not forget to add the percent sign after the number (e.g. `"60%"`).

Example 3

The sample below shows basic configuration code that can be used to insert a CKEditor instance with the file browser paths and window size configured.

```
CKEDITOR.replace( 'editor1', {
    filebrowserBrowseUrl: '/browser/browse.php',
    filebrowserUploadUrl: '/uploader/upload.php',
    filebrowserWindowWidth: '640',
    filebrowserWindowHeight: '480'
});
```

To set the window size of the file browser for a specific dialog window, use the `filebrowserWindowWidth` and `filebrowserWindowHeight` settings.

For example, to change the file browser window size only in "Image" dialog box, change set the `filebrowserImageWindowWidth` and `filebrowserImageWindowHeight` settings.

Example 4

The sample below shows basic configuration code that can be used to insert a CKEditor instance with the file browser paths configured. It also changes the default dimensions of the file browser window, but only when opened

from the **Image Properties** dialog window.

```
CKEDITOR.replace( 'editor1', {
    filebrowserBrowseUrl: '/browser/browse.php',
    filebrowserUploadUrl: '/uploader/upload.php',
    filebrowserImageWindowWidth: '640',
    filebrowserImageWindowHeight: '480'
});
```

Using CKFinder

CKEditor may easily be integrated with CKFinder, an advanced Ajax file browser. For a live demonstration, see here.

The integration may be conducted in two ways: by setting CKEditor configuration options (example below) or by using the CKFinder.SetupCKEditor() method available in the CKFinder API.

Example 5

The sample below shows the configuration code that can be used to insert a CKEditor instance with CKFinder integrated. The browse and upload paths for images and Flash objects are configured separately from CKFinder default paths.

```
CKEDITOR.replace( 'editor1', {
    filebrowserBrowseUrl: '/ckfinder/ckfinder.html',
    filebrowserImageBrowseUrl: '/ckfinder/ckfinder.html?Type=Images',
    filebrowserFlashBrowseUrl: '/ckfinder/ckfinder.html?Type=Flash',
    filebrowserUploadUrl: '/ckfinder/core/connector/php/connector.php?
command=QuickUpload&type=Files',
    filebrowserImageUploadUrl: '/ckfinder/core/connector/php/connector.php?
command=QuickUpload&type=Images',
    filebrowserFlashUploadUrl: '/ckfinder/core/connector/php/connector.php?
command=QuickUpload&type=Flash'
});
```

The example above is valid for PHP environment. Note that `/ckfinder/` is a base path to the CKFinder installation directory.

If you are using CKFinder for ASP, ASP.NET, or ColdFusion, remember to change `php` above to the right extension:

- asp – CKFinder for ASP
- aspx – CKFinder for ASP.NET
- cfm – CKFinder for ColdFusion
- php – CKFinder for PHP

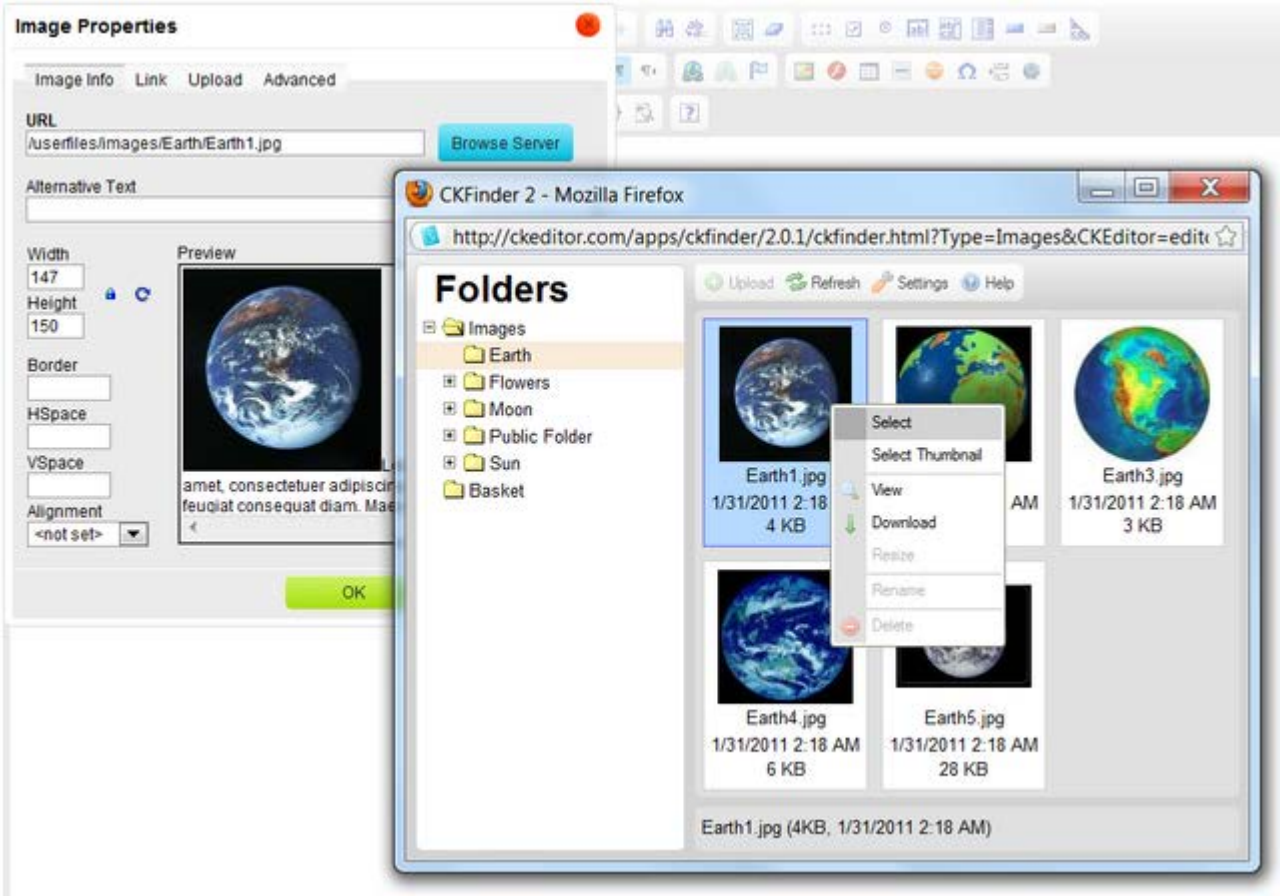
Example 6

The sample below shows the use of the `CKFinder.SetupCKEditor()` to insert a CKEditor instance with CKFinder integrated.

```
var editor = CKEDITOR.replace( 'editor1' );
CKFinder.SetupCKEditor( editor, '/ckfinder/' );
```

The second parameter of the `SetupCKEditor()` method is the path to the CKFinder installation.

Please check the `_samples/js/ckeditor.html` sample distributed with CKFinder to see the full working example of this integration method.



PHP API

As of CKFinder 1.4.2 and CKEditor 3.1 it is possible to integrate CKFinder with CKEditor using the PHP API.

See CKFinder for PHP documentation for more details.

Other Resources

For more advanced information on integrating CKEditor with a file browser refer to the following articles:

- Creating a Custom File Browser
- Adding the File Browser to Dialog Windows

File Browser API

CKEditor can be easily integrated with your own file browser.

To connect a file browser that is already compatible with CKEditor (like CKFinder), follow the File Browser (Uploader) documentation.

Contents

Interaction Between CKEditor and File Browser

CKEditor automatically sends some additional arguments to the file browser:

- CKEditor – name of the CKEditor instance,
- langCode – CKEditor language (en for English),
- `CKEditorFuncNum` – anonymous function reference number used to pass the URL of a file to CKEditor (a random number).

For example:

```
CKEditor=editor1&CKEditorFuncNum=1&langCode=en
```

Example 1

Suppose that CKEditor was created using the following JavaScript call:

```
CKEDITOR.replace( 'editor2', {
    filebrowserBrowseUrl: '/browser/browse.php?type=Images',
    filebrowserUploadUrl: '/uploader/upload.php?type=Files'
});
```

In order to browse files, CKEditor will call:

```
/browser/browse.php?type=Images&CKEditor=editor2&CKEditorFuncNum=2&langCode=de
```

The call includes the following elements:

- `/browser/browse.php?type=Images` – the value of the `filebrowserBrowseUrl` parameter,
- `&CKEditor=editor2&CKEditorFuncNum=2&langCode=de` – the information added by CKEditor:
  - `CKEditor=editor2` – the name of a CKEditor instance (`editor2`),
  - `CKEditorFuncNum=2` – the reference number of an anonymous function that should be used in the callFunction,
  - `langCode=de` – language code (in this case: German). This parameter can be used to send localized error messages.

Passing the URL of the Selected File

To send back the file URL from an external file browser, call CKEDITOR.tools.callFunction and pass `CKEditorFuncNum` as the first argument:

```
window.opener.CKEDITOR.tools.callFunction( funcNum, fileUrl [, data] );
```

If `data` (the third argument) is a string, it will be displayed by CKEditor. This parameter is usually used to display an error message if a problem occurs during the file upload.

Example 2

The following example shows how to send the URL from a file browser using JavaScript code:

```
// Helper function to get parameters from the query string.
function getUrlParam( paramName ) {
    var reParam = new RegExp( '(?:[\?&]|&)' + paramName + '=([^&]+)', 'i' ) ;
    var match = window.location.search.match(reParam) ;

    return ( match && match.length > 1 ) ? match[ 1 ] : null ;
}
var funcNum = getUrlParam( 'CKEditorFuncNum' );
var fileUrl = '/path/to/file.txt';
window.opener.CKEDITOR.tools.callFunction( funcNum, fileUrl );
```

Example 3

The following code shows how to send back the URL of an uploaded file from the PHP connector:

```
<?php
    // Required: anonymous function reference number as explained above.
    $funcNum = $_GET['CKEditorFuncNum'] ;
    // Optional: instance name (might be used to load a specific configuration file
or anything else).
    $CKEditor = $_GET['CKEditor'] ;
    // Optional: might be used to provide localized messages.
    $langCode = $_GET['langCode'] ;

    // Check the $_FILES array and save the file. Assign the correct path to a
variable ($url).
    $url = '/path/to/uploaded/file.ext';
    // Usually you will only assign something here if the file could not be
uploaded.
    $message = ;

    echo "<script
type='text/javascript'>window.parent.CKEDITOR.tools.callFunction($funcNum, '$url',
'$message');</script>";
?>
```

Example 4

If `data` is a function, it will be executed in the scope of the button that called the file browser. It means that the server connector can have direct access CKEditor and the dialog window to which the button belongs.

Suppose that apart from passing the `fileUrl` value that is assigned to an appropriate field automatically based on the dialog window definition you also want to set the `alt` attribute, if the file browser was opened in the **Image Properties** dialog window. In order to do this, pass an anonymous function as a third argument:

```
window.opener.CKEDITOR.tools.callFunction( funcNum, fileUrl, function() {
    // Get the reference to a dialog window.
```

```
    var element,
        dialog = this.getDialog();
    // Check if this is the Image dialog window.
    if ( dialog.getName() == 'image' ) {
        // Get the reference to a text field that holds the "alt" attribute.
        element = dialog.getContentElement( 'info', 'txtAlt' );
        // Assign the new value.
        if ( element )
            element.setValue( 'alt text' );
    }
    ...
    // Return false to stop further execution - in such case CKEditor will ignore
the second argument (fileUrl)
    // and the onSelect function assigned to a button that called the file browser
(if defined).
    [return false;]
});
```

Adding File Browser to Dialog

CKEditor can be easily integrated with your own file browser.

To connect a file browser that is already compatible with CKEditor (like CKFinder), follow the File Browser (Uploader) documentation.

Contents

Dialogs

Please refer to the Dialog definition API for general help on how to create a dialog box.

Filebrowser Plugin

The `filebrowser` plugin is built-in into CKEditor. It's only purpose is to provide an API inside of CKEditor to easily integrate any external file browser with it and to add file browser features to various CKEditor components (usually to dialogs).

Adding "Browse Server" button

To assign the filebrowser plugin to an element inside of a dialog box, set the "filebrowser" property. For example in the `image` plugin source there is:

```
{
    type: 'button',
    hidden: true,
    id: 'browse',
    filebrowser: 'Link:txtUrl',
    label: editor.lang.common.browseServer,
    style: 'float:right',
},
```

This button will be hidden by default (`hidden:true`). The filebrowser plugin looks for all elements with the filebrowser attribute and unhides them if appropriate configuration setting is available (filebrowserBrowseUrl/filebrowserUploadUrl).

The action performed by the plugin depends on the element type, for fileButton it is **QuickUpload**, for other elements the default action is **Browse**. In the example above, the file browser will be launched (in a popup) when button is clicked.

The `'Link:txtUrl'` value instructs the plugin to update an element with id `txtUrl` inside of the `Link` tab when `CKEDITOR.tools.callFunction( funcNum )` is called (see Custom File Browser).

Adding "Quick Upload" support

Again, to see how we can handle file uploads in our dialog box, we'll use working example from CKEditor. In the `image` plugin there is a definition of the `Upload` tab:

```
{
    id: 'Upload',
    hidden: true,
    filebrowser: 'uploadButton',
```

```
    label: editor.lang.image.upload,
    elements: [
        {
            type: 'file',
            id: 'upload',
            label: editor.lang.image.btnUpload,
            style: 'height:40px',
            size: 38
        },
        {
            type: 'fileButton',
            id: 'uploadButton',
            filebrowser: 'info:txtUrl',
            label: editor.lang.image.btnUpload,
            'for': [ 'Upload', 'upload' ]
        }
    ]
},
```

This example is a little bit more complicated than the previous one, because 1) we have here a definition of the whole tab 2) we need two elements: file and fileButton to upload a file.

In the example above, the id of a tab is `'Upload'`. It is hidden by default (`hidden:true`). As already mentioned, the filebrowser plugin looks for all elements with the filebrowser attribute and unhides them if appropriate configuration setting is available. In this case, the tab will be unhidden automatically if a filebrowser setting for `'uploadButton'` (because of `filebrowser:'uploadButton'`) will be available (`filebrowserUploadUrl`).

The `file` element is simple and doesn't need to be explained, it is just an input element that will store the name of a file that will be uploaded.

The `fileButton` element is more interesting. The `'info:txtUrl'` value instructs the filebrowser plugin to update an element with id `txtUrl` inside of the `info` tab when `CKEDITOR.tools.callFunction( funcNum )` is called (see Custom File Browser). The `'for': [ 'Upload', 'upload' ]` line is used to connect fileButton with file element. It is an instruction for CKEditor to upload a file using the `'file'` element with id `'upload'` (second value) inside of the `'Upload'` tab (first value).

Advanced configuration (Browsing)

It is possible to define your own function that will be called when file is selected/uploaded.

```
{
    type: 'button',
    hidden: true,
    id: 'id0',
    label: editor.lang.common.browseServer,
    filebrowser: {
        action: 'Browse',
        // target: 'tab1:id1',
        onSelect: function( fileUrl, data ) {
            alert( 'The selected file URL is "' + fileUrl + '"' );

            for ( var _info in data )
                alert( 'data[ "' + _info + '" ]' + ' = ' + data[ _info ] );

            var dialog = this.getDialog();
            dialog.getContentElement( 'tab1', 'id1' ).setValue( data[ 'fileUrl' ]
);
```

```
            // Do not call the built-in onSelect command
            return false;
        }
    }
}
```

In this example we're setting the action to 'Browse' to call the file browser when button is clicked. `'target'` is not required, because we'll update the target element in the custom `onSelect` function.

As explained in the documentation, we have called `CKEDITOR.tools.callFunction( funcNum, fileUrl, data );` when user selected a file. The fileUrl and data arguments are passed to our custom `onSelect` function and we can use it to update the target element.

Advanced configuration (Quick Uploads)

In a similar way like we configured the button to open the file browser, we can configure the fileButton.

```
{
    type: 'file',
    label: editor.lang.common.upload,
    labelLayout: 'vertical',
    id: 'id2'
},
{
    type: 'fileButton',
    label: editor.lang.common.uploadSubmit,
    id: 'id3',
    filebrowser: {
        action: 'QuickUpload',
        params: { type: 'Files', currentFolder: '/folder/' },
        target: 'tab1:id1',
        onSelect: function( fileUrl, errorMessage ) {
            alert( 'The url of uploaded file is: ' + fileUrl + '\nerrorMessage: ' +
errorMessage );
            // Do not call the built-in onSelect command
            // return false;
        }
    },
    'for': [ 'tab1', 'id2' ]
}
```

In the `filebrowser.params` attribute we can add additional arguments to be passed in the query string to the external file browser. `filebrowser.target` is the target element to be updated when file is returned by the server connector (uploader) - we don't need it if we define our own `onSelect` function (`filebrowser.onSelect`) and update the target element in this function, just like we did in previous example.

Plugins

CKEditor is totally based on plugins. In fact, the editor core is an empty box, which is them filled with features provided by plugins. Even the editor interface, like toolbars, buttons and the editing area are plugins.

The default installation of CKEditor, that you probably are using now, comes with a set of plugins present on it. You can add plugins into your editor, bring nice and useful features for your users.

Contents

Where to look for plugins?

The CKEditor Add-ons Repository is an online service designed to find and share plugins. It makes it easy to understand the plugins features and to socialize with the CKEditor community. It's also the best place to showcase your skills and reach a large user base, if you are a plugin developer.

Creating a Custom Editor with CKBuilder

CKBuilder is the sister service of the CKEditor Add-ons Repository, making it possible to create a customized editor, by selecting plugins the plugins that best fit your needs.

Through the navigation of the add-ons repository, you'll be able to use the "Add to my editor button" to send your preferred plugins to your custom editor. Once done, you can simply download it an enjoy an editing experience that is perfect for your needs.

Installing Plugins Manually

If you prefer not to use CKBuilder, if you have plugins developer by yourself of by third parties or if you just want to test plugins before going through the CKBuilder process, you can also add plugins to your local installation, by following a few steps:

1. **Extracting the zip file:** Plugins are usually available as zip files. So, to start, be sure to have the zip extracted into a folder.
2. **Copying the files into CKEditor:** The easiest way to install the files is by simply copying them into the `plugins` folder of your CKEditor installation. They must be placed into a sub-folder that matches the "technical" name of the plugin. For example, the Magic Line plugin would be installed into this folder: `<CKEditor folder>/plugins/magicline`.
3. **Enabling the plugin:** Now it is time to tell CKEditor that you have a new plugin for it. For that, you simply use the extraPlugins configuration option:

```
config.extraPlugins = 'magicline';
```

That's all. Your plugin will be now enabled in CKEditor.

Skins

The CKEditor user interface look and feel can be totally customized through skins. Elements like the toolbar, dialogs, buttons and even their icons, can be changed to match your preferred style.

The default installation of CKEditor comes with the Moono skin.

Contents

Where to look for skins?

The CKEditor Add-ons Repository is an online service designed to find and share skins. It's also the best place to showcase your skills and reach a large user base, if you are a skin developer.

Downloading CKEditor with your preferred skin

CKBuilder is the sister service of the CKEditor Add-ons Repository, making it possible to create a customized editor with any skin you want.

Installing Skins Manually

If you prefer not to use CKBuilder, if you have skins developed by yourself of by third parties or if you just want to test skins before going through the CKBuilder process, you can also add skins to your local installation, by following a few steps:

1. **Extracting the zip file:** Skins are usually available as zip files. So, to start, be sure to have the zip extracted into a folder.
2. **Copying the files into CKEditor:** The easiest way to install the files is by simply copying them into the `skins` folder of your CKEditor installation. They must be placed into a sub-folder that matches the "technical" name of the skin. For example, the Kama skin would be installed into this folder: `<CKEditor folder>/skins/kama`.
3. **Enabling the plugin:** Now you must just setup CKEditor, by using the skin configuration option:

```
config.skin = 'kama';
```

That's all. The new skin will be now enabled in CKEditor.

Getting the Source Code

Working with the source code of CKEditor may be useful. These are some possible situations that you may face:

- You're developing plugins or skins, so you can build your own distributions.
- You're assembling and editor "by hand", by adding plugins and skins to it manually.
- You want to understand better the CKEditor API by simply reading the code.
- You want to fix an issue. (Yes, do it!)

Contents

Cloning from GitHub

The CKEditor source code is available in the ckeditor-dev git repository, hosted at GitHub.

Having git installed in your system, it's enough to call the following at the command line to have your local copy:

```
git clone https://github.com/ckeditor/ckeditor-dev.git
```

It'll download the full CKEditor development code into the `ckeditor-dev` folder.

Performance

Note that the source code version of CKEditor is not optimized for production websites. It works flawlessly on a local computer or network, but a simple sample file can download more than two hundred files and more than one megabyte.

Because of this **do not use the source code version of CKEditor in production websites**.

Once your local development is completed, be sure to build CKEditor, making it perfect for distribution.

Build from Source Code

If you're working with the source code of CKEditor in your computer or local network, at some stage you'll have to distribute it into test or production websites.

Never distribute the source version of CKEditor into production websites. There are serious performance implications on doing this.

Instead, you must create a CKEditor "build" or "release version" (in contrast to "source version"). It is an optimized production ready CKEditor distribution.

Contents

The dev/builder Folder

The source code of CKEditor contains a pre-configured environment so you can easily create CKEditor builds.

The following are the files that are most relevant:

- `build.sh`: the build runner bash script.
- `build-config.js`: the build configuration file.

Step 1: Build Setup

You should edit the `build-config.js` file, which contains the build configuration. It has the following sections:

```
var CKBUILDER_CONFIG = {
    // Skin name.
    skin: '...',

    // Files to be ignored.
    ignore: [ ... ],

    // Plugins to be included.
    plugins: { ... }
};
```

The most important parts of the file is the `skin` name and the list of `plugins`. Be sure to have them properly set, with the things you really want in your build.

You don't need to include all plugins into that list. CKBuilder will discover their dependencies and load them as well.

Step 2: Running the Builder

There is little to add now. Simply go to the command line and call the build script:

```
sh build.sh
```

The builder will be executed and the resulting build will be created in the `dev/builder/build` folder.

About CKBuilder (Command Line)

The building process is handled by the command line version of **CKBuilder**. It is a powerful application that makes several enhancement to the source code. It loads the configuration file, discover plugin dependencies, merge and minify files, create icon strips and perform many other tasks.

For the first run, `build.sh` will download CKBuilder, if not available, and copy it into the `dev/builder/ckbuilder/<ckbuilder version>` folder. Therefore, Internet connection is required. Once the file is available, no more downloads are required (but still the script tries to update it, if possible).

The only requirement to run CKBuilder is Java.

Basic Configuration and Customization

Contents

How Do I Change the Default CKEditor Configuration?

CKEditor is a highly flexible tool that you can easily customize to your needs. If you want to change the editor configuration, refer to the Setting Configuration page from the Developer's Guide. Your custom configuration will help you adjust the CKEditor look and feel to the requirements of your project.

How Do I Find the CKEditor Configuration Settings to Change?

A full listing of configuration settings that you can change in order to customize the editor to your needs can be found in the CKEditor JavaScript API. Use the methods described in the Setting Configuration article from the Developer's Guide.

How Do I Remove Unneeded CKEditor Functionality?

CKEditor is a truly flexible tool with a modular structure — most editor functionality is based on plugins. Some core plugins are necessary for the editor to work or depend on one another, however, there are lots of optional plugins that you can skip when you do not need the functionality that they provide.

If you want to disable some functionality that comes from a CKEditor plugin, you can use the CKEDITOR.config.removePlugins setting to prevent the plugin from loading.

```
// Remove one plugin.
config.removePlugins = 'elementspath';

// Remove multiple plugins.
config.removePlugins = 'elementspath,save,font';
```

You can also use the CKBuilder Online service to download a truly customized version of CKEditor.

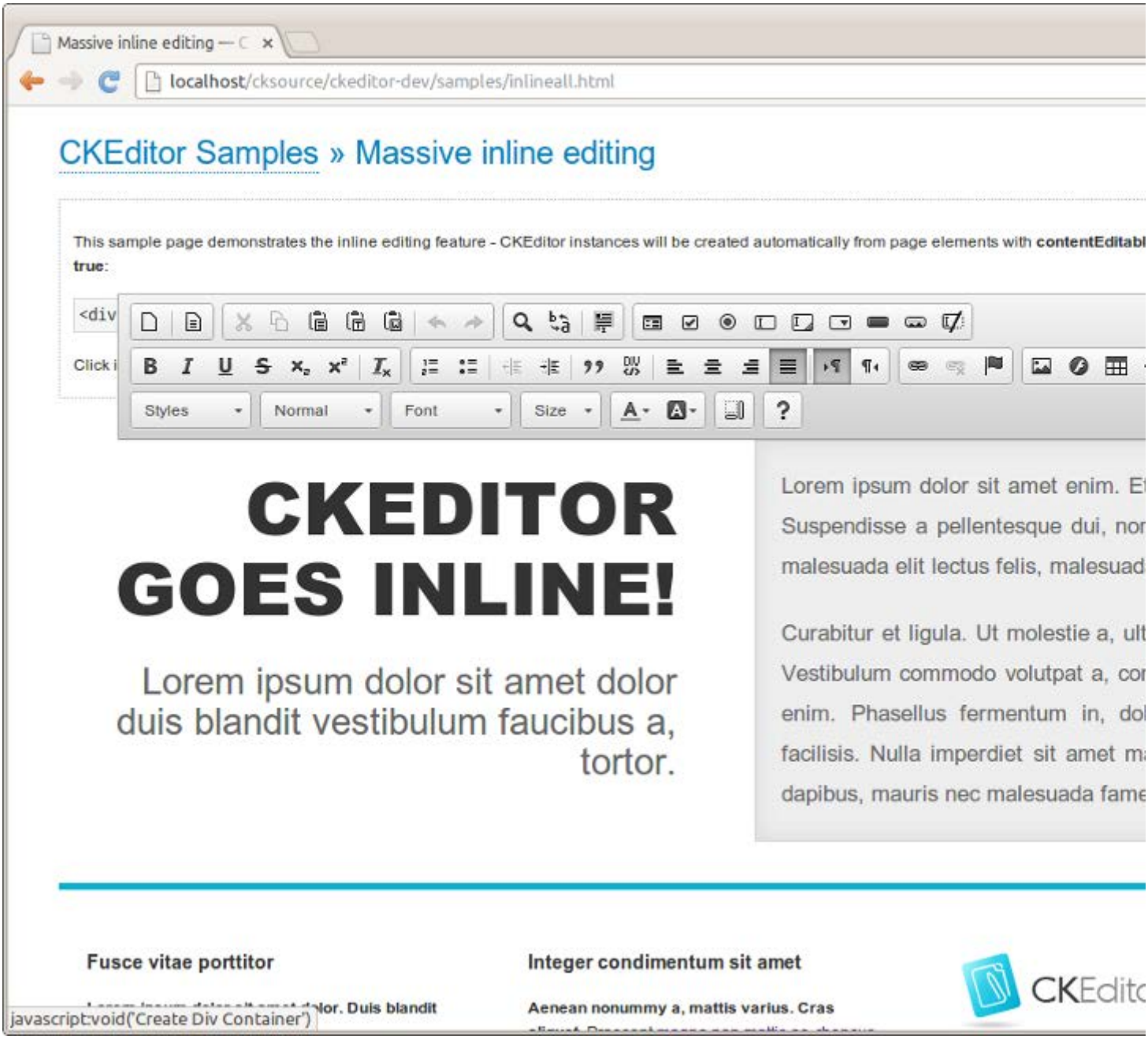
How Do I Find Code Examples Showing CKEditor Customization?

Each CKEditor installation package available on the official download site contains a `samples/` folder.

Once you download CKEditor, open the `samples/index.html` file in your browser to see a list of samples presenting a broad range of usage scenarios and customization options for CKEditor. Each sample contains a short description along with a code snippet as well as one or more CKEditor instances to play with.

If you are interested in learning how to create your own code to embed, configure, and customize CKEditor, have a look at the source code of the sample pages.

The figure below presents one of the CKEditor samples, Massive inline editing (`inlineall.html`), opened in a browser.


Massive inline editing
localhost/cksource/ckeditor-dev/samples/inlineall.html
CKEditor Samples » Massive inline editing
This sample page demonstrates the inline editing feature - CKEditor instances will be created automatically from page elements with contentEditabl
true:
Styles
Normal
Font
Size
CKEDITOR
GOES INLINE!
Lorem ipsum dolor sit amet dolor
duis blandit vestibulum faucibus a,
tortor.
Lorem ipsum dolor sit amet enim. Et
Suspendisse a pellentesque dui, nor
malesuada elit lectus felis, malesuad
Curabitur et ligula. Ut molestie a, ult
Vestibulum commodo volutpat a, cor
enim. Phasellus fermentum in, dol
facilisis. Nulla imperdiet sit amet ma
dapibus, mauris nec malesuada fame
Fusce vitae porttitor
Integer condimentum sit amet
Aenean nonummy a, mattis varius. Cras
CKEdito
javascript:void('Create Div Container')

Licensing and Support

Contents

How Do I Get Support?

If you are having trouble installing, configuring, or integrating CKEditor to your application, there are a few solutions that you can try.

Documentation

First of all, CKEditor comes with really extensive documentation that you should read.

Community Forum

The CKEditor Community Forums the place where CKEditor developers and integrators can share their problems and solutions. CKEditor has grown as an Open Source product, thanks to the amazing community of developers and users. The community forum works in Open Source way, so you are encouraged to not only ask questions, but also answer them and help fellow developers. It will be much appreciated!

Professional Support Channel

There are times and situations, however, when you are unable to get going by yourself. If this is a case, you will be happy to know that CKEditor Premium includes professional assistance directly from CKEditor core developers. The dedicated support channel is available for you and responses to your inquiries come in one business day, or even in the same day.

How Do I Support the Development of CKEditor?

CKEditor is and has always been an Open Source product. We are proud to be a part of the Open Source movement and are happy that we can offer the result of long hours of work and dedicated contribution **completely for free** to everyone who wants to use the editor.

If you also want to support the development of CKEditor, it will be most welcome. Here are a couple of things that you can do:

- Report bugs or feature requests and submit patches on our Development site.
- Help localize CKEditor into your native language and update existing localizations by joining us at the CKEditor UI Translation Center.
- Create CKEditor plugins with new functionality and publish them on the CKEditor Add-Ons Repository.
- Join the Community Forum and share your knowledge with your fellow CKEditor users and developers.
- Buy CKEditor Premium. This will not only let you support the development of new editor features, but it will also give you access to a dedicated support channel where CKEditor developers are available to help you solve any issues you might have.

Thank you for your support and for helping us make CKEditor better every day!

How Does CKEditor Premium Works?

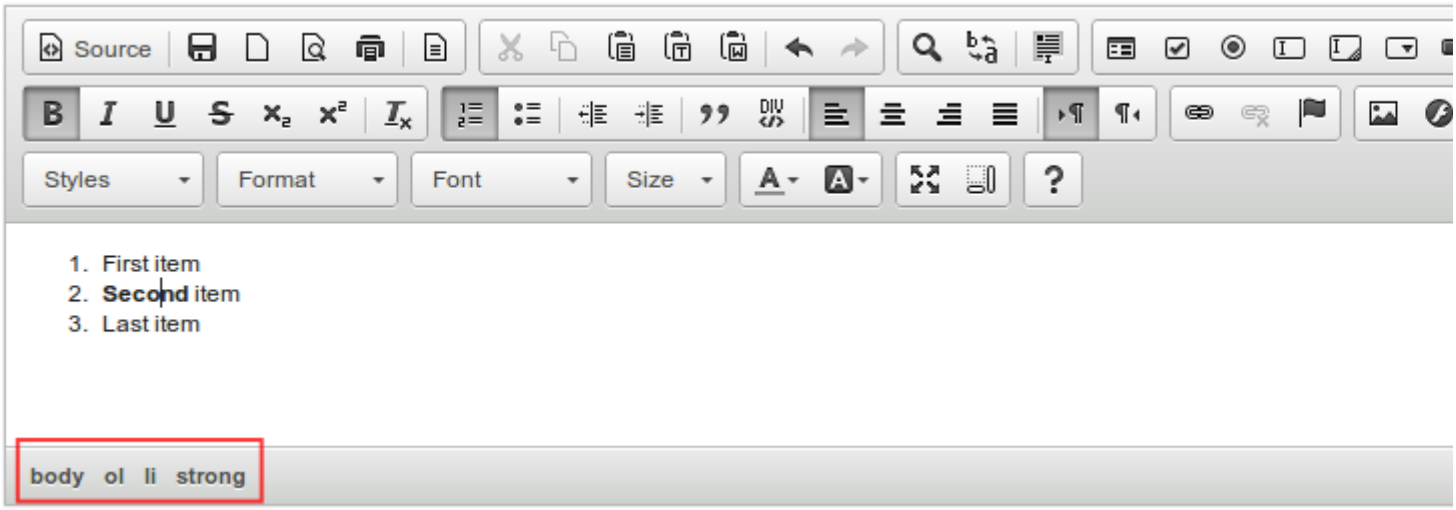
For full details about it, visit the CKEditor Premium website.

Interface

Contents

How Do I Remove the Elements Path?

The **elements path** displays information about the HTML elements of the document for the position of the cursor.



If you want to get rid of it, use the CKEDITOR.config.removePlugins setting to remove the `elementspath` plugin.

```
config.removePlugins = 'elementspath';
```

How Do I Change the Size of the Editor?

To define the default size of the editor, use the width and height configuration settings.

Note that the `width` value can be given as a number representing the value in pixels or as a percent representing the size relative to the parent element containing the editor.

```
config.width = 850;
config.width = '75%';
```

The `height` value defines the height of CKEditor editing area and can be given in pixels or em. Percent values are not supported.

```
config.height = 500;
config.height = '25em';
config.height = '300px';
```

How Do I Change the Size of the Editor on the Fly?

Besides defining a default size of the editor window you can also change the size of a CKEditor instance on the fly.

To achieve this, use the resize function to define the dimensions of the editor interface, assigning the window a width and height value in pixels or CSS-accepted units.

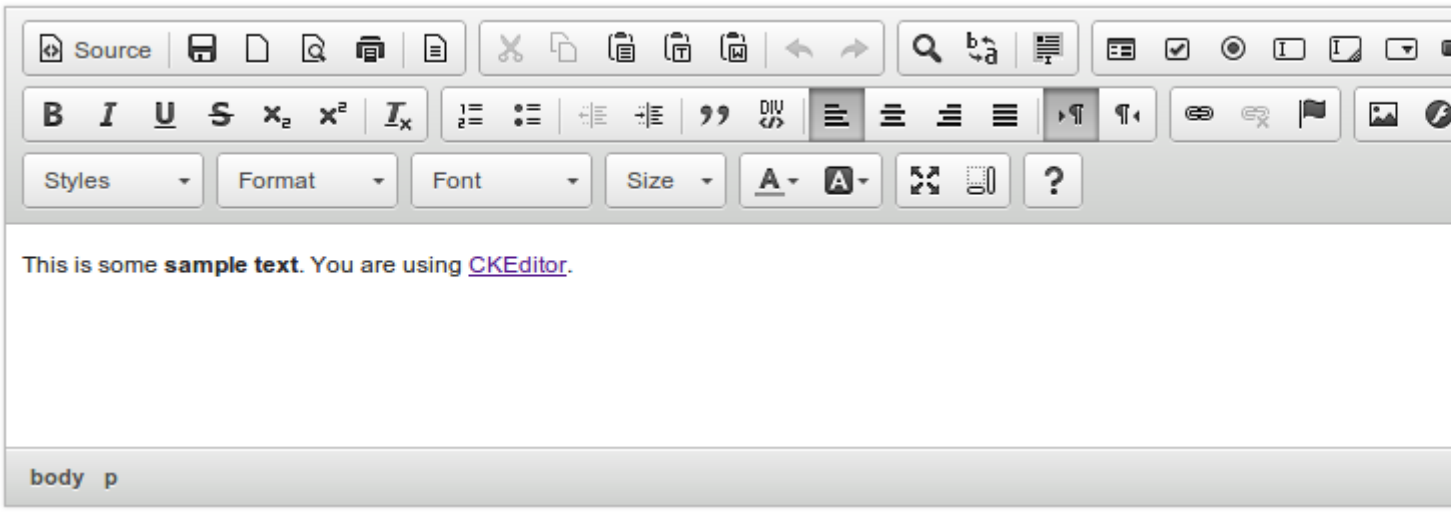
```
// Set editor width to 100% and height to 350px.
editor.resize( '100%', '350' );
```

While setting the height value, use the `isContentHeight` parameter to decide whether the value applies to the whole editor interface or just the editing area.

```
// The height value now applies to the editing area.
editor.resize( '100%', '350', true );
```

How Do I Remove the Ability to Resize CKEditor?

The editor window can be resized by using the resizing grip located in the bottom right-hand corner of CKEditor interface (for RTL languages — in the bottom left-hand corner).



To prevent the editor from being resized you can use the removePlugins setting to remove the `resize` plugin.

```
config.removePlugins = 'resize';
```

You can also disable this feature by setting the resize_enabled parameter to `false`.

```
config.resize_enabled = false;
```

How Do I Limit the Width and Height for CKEditor Resizing?

CKEditor window can be resized if the `resize` plugin is enabled. You can however define the minimum and maximum dimensions to prevent the editor window from becoming too small or too big to handle.

To define the minimum editor dimensions after resizing, specify the resize_minWidth and resize_minHeight values in pixels.

```
config.resize_minWidth = 300;
config.resize_minHeight = 300;
```

To define the maximum editor dimensions after resizing, specify the resize_maxWidth and resize_maxHeight values in pixels.

```
config.resize_maxWidth = 800;
config.resize_maxHeight = 600;
```

How Do I Limit the Directions for CKEditor Resizing to Horizontal or Vertical Only?

CKEditor window can be resized if the `resize` plugin is enabled. You can however define the resizing directions in order to have more control over the resulting editor appearance.

By default CKEditor resizing is allowed in both directions — vertical and horizontal. This is achieved thanks to setting the resize_dir configuration value to `'both'` (this is the default setting).

```
config.resize_dir = 'both';
```

If you want to allow vertical resizing only, you need to set the resize_dir configuration value to `'vertical'`.

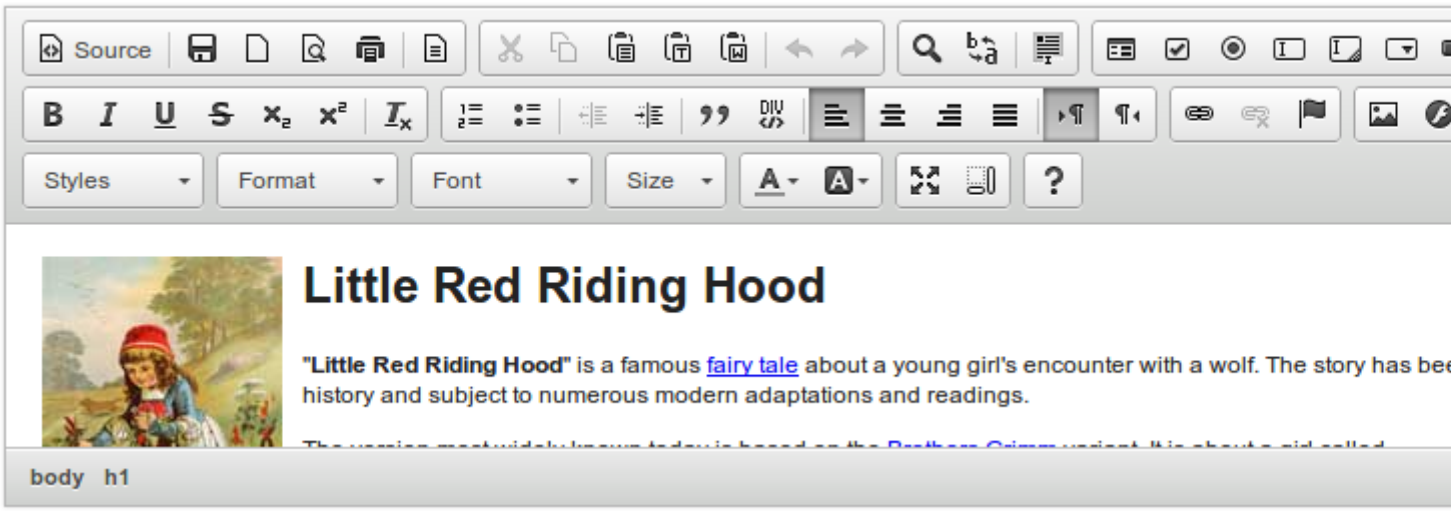
```
config.resize_dir = 'vertical';
```

If you set the resize_dir configuration value to `'horizontal'`, CKEditor window will only be resizable in horizontal dimension.

```
config.resize_dir = 'horizontal';
```

How Do I Add the Toolbar Collapse Button?

CKEditor toolbar can be collapsed and restored by using the **Collapse Toolbar** button located in the bottom right-hand corner of the toolbar (for RTL languages — in the bottom left-hand corner).

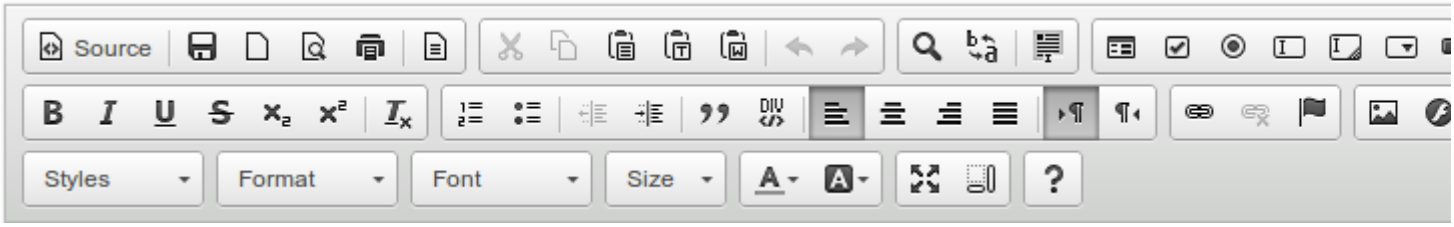


If you want to enable this feature, you need to set the toolbarCanCollapse parameter to `true`.

```
config.toolbarCanCollapse = true;
```

How Do I Add or Remove Toolbar Buttons?

CKEditor toolbar is an array of button elements that you can freely add or remove.



Check the Toolbar Customization section of this guide for more information on how to customize it.

How Do I Navigate CKEditor Using the Keyboard?

CKEditor Accessibility Guide contains lots of useful information on using the CKEditor interface with your keyboard or with assistive devices such as screen readers.

Many functions in CKEditor have their equivalent keyboard shortcuts. This is one of the reasons why working with the editor is simple and efficient.

The Keyboard Shortcuts article describes available keyboard shortcuts grouped by problem areas.

How Do I Configure CKEditor to Use the Arrow Keys to Navigate Between All Toolbar Buttons?

In CKEditor 3.6 the concept of **toolbar button groups** was introduced to enable faster and more efficient navigation using the keyboard or assistive devices. In all previous versions of the editor, the `Tab` and `Shift+Tab` keys had the same effect as using the `Right` and `Left Arrow` keys and were used to cycle between consecutive toolbar buttons.

Since CKEditor 3.6, `Tab` and `Shift+Tab` navigate between toolbar button groups, while the `Arrow` keys are used to cycle between the buttons within a group.

In order to change the new default toolbar navigation mode and use the `Arrow` keys as an equivalent to `Tab` and `Shift+Tab`, use the following toolbarGroupCycling configuration setting:

```
config.toolbarGroupCycling = false;
```

Styles

Contents

How Do I Customize the Styles Drop-Down List?

You need to pass your own style definitions to the CKEDITOR.stylesSet.add function, giving them a unique name, and then configure the editor to use them by setting the stylesSet value.

```
CKEDITOR.stylesSet.add( 'my_styles', [
    // Block-level styles.
    { name: 'Blue Title', element: 'h2', styles: { color: 'Blue' } },
    { name: 'Red Title',  element: 'h3', styles: { color: 'Red' } },

    // Inline styles.
    { name: 'CSS Style', element: 'span', attributes: { 'class': 'my_style' } },
    { name: 'Marker: Yellow', element: 'span', styles: { 'background-color':
'Yellow' } }
]);
```

Depending on whether your definitions were placed inline or in an external file, you need to set the stylesSet configuration setting accordingly.

```
// For inline style definition.
config.stylesSet = 'my_styles';

// For a definition in an external file.
config.stylesSet = 'my_styles:http://www.example.com/styles.js';
```

For more details on the definition format and best practices on how to customize the styles please refer to the Styles article from the Developer's Guide.

Stylesheet Parser Plugin

Note that since CKEditor 3.6 you can also populate the **Styles** drop-down list with style definitions added in an external CSS stylesheet file. Check the How Do I Add Existing CSS Styles from an External File to the Styles Drop-Down List? article for more information about using the new (and optional) **Stylesheet Parser** plugin.

How Do I Add Existing CSS Styles from an External File to Editor Output and the Styles Drop-Down List?

CKEditor 3.6 and later includes the **Stylesheet Parser** (`stylesheetparser`) plugin that can be used to point to an external CSS stylesheet containing style definitions. It will help you use existing CSS styles and display them in the **Styles** drop-down list without a need to define them specifically for CKEditor as described here.

For more information on using the plugin refer to the Stylesheet Parser Plugin section of the Developer's Guide and check the "Stylesheet Parser plugin" (`stylesheetparser.html`) sample from the `samples/` folder of your CKEditor installation package.

How Do I Add Custom Styles Based on CSS Classes?

Add a style definition as described in the How do I customize the Styles drop-down list? article and pass the class name in the `attributes parameter.

Note: do remember that since some old browsers recognize `class` as a resereved word in JavaScript, you need to place it in quotes.

The following example adds a myClass class to an img element. The image element will now be styled as defined in this CSS class.

```
{
    name: 'Custom Image',
    element: 'img',
    attributes: { 'class': 'myClass' }
}
```

For more details on the definition format and best practices on how to customize the styles please refer to the Styles article from the Developer's Guide.

How Do I Use the Styles on Images, Tables or Other Elements?

If you added some custom style definitions for objects such as tables or images, you need to select these objects first before you will be able to apply the style. Object styles are only shown in the **Styles** drop-down list and can be applied `after` the element was selected in the editor.

File Upload

Contents

How Do I Upload Files or Images Using CKEditor?

By default CKEditor does not include a file browser or uploader. You can, however, create a custom file browser or use an existing one, like CKFinder.

How Do I Paste a Local Image from my Clipboard to CKEditor?

It is not possible to paste a local file (like an image) directly to a website located on the server. This issue has nothing to do with CKEditor, but is related to the security model of the Internet browsers.

If you want to add images to your document, you need to upload them to a server first. You can either upload them to a server manually and then insert them using the Insert Image feature, giving the URL of the image, or integrate CKEditor with a file browser and uploader like CKFinder and use it to upload the image from your local computer to the server inside the CKEditor interface.

Output

How Do I Output HTML Instead of XHTML Code Using CKEditor?

Contents

If you want CKEditor to output valid HTML4 code instead of XHTML, you should configure the behavior of the dataProcessor.

For some tips on how to achieve this, check the Output Formatting section of Developer's Guide as well as the Output HTML (`plugins/htmlwriter/samples/outputhtml.html`) and Output XHTML (`samples/xhtmlstyle.html`) samples that can be found in CKEditor installation package.

If, for example, you want CKEditor to output the self-closing tags in the HTML4 way, creating `<br>` elements instead of `<br />`, configure the `selfClosingEnd` setting in the following way.

```
CKEDITOR.on( 'instanceReady', function( ev ) {
    ev.editor.dataProcessor.writer.selfClosingEnd = '>';
});
```

How Do I Output BBCode Instead of HTML Code Using CKEditor?

You should try the BBCode plugin.

Pasting

How Do I Preserve Font Styles and Backgrounds When Pasting from Word?

The **Paste from Word** feature lets you copy the contents of Microsoft Word or Excel documents and paste them into the editor, preserving the structure and styles that were present in the original text.

Note, however, that by default some font styles are not preserved to avoid conflicting with the styles of the document created in CKEditor. If however, you want to use Word font styles, including font size, font family, and font foreground/background color, set the pasteFromWordRemoveFontStyles configuration value to `false`.

```
config.pasteFromWordRemoveFontStyles = false;
```

Spellchecker and Spell Check As You Type (SCAYT)

How Do I Set SCAYT to Turn On Automatically?

Contents

If you want to turn on the Spell Check As You Type (SCAYT) feature in CKEditor by default, set the `scayt_autoStartup` configuration setting to `true`.

```
config.scayt_autoStartup = true;
```

How Do I Disable SCAYT in CKEditor?

If you want to completely disable the Spell Check As You Type (SCAYT) feature in CKEditor, remove the `scayt` plugin using the removePlugins configuration setting.

```
config.removePlugins = 'scayt';
```

If you want to leave SCAYT available, but prevent the feature from being turned on automatically on loading the editor, set the `scayt_autoStartup` configuration setting to `false`. This is the default value for CKEditor configuration.

```
config.scayt_autoStartup = false;
```

How Do I Change the Default Language for Spell Check As You Type (SCAYT)?

By default SCAYT treats the text written in the editor as American English (`en_US`). If you want to change the default SCAYT language, set the scayt_sLang configuration value to one of the 16 possible language codes that are currently accepted.

```
// Sets SCAYT to French.
config.scayt_sLang = 'fr_FR';
```

These are the language codes of languages currently supported by SCAYT: `en_US`, `en_GB`, `pt_BR`, `da_DK`, `nl_NL`, `en_CA`, `fi_FI`, `fr_FR`, `fr_CA`, `de_DE`, `el_GR`, `it_IT`, `nb_NO`, `pt_PT`, `es_ES`, and `sv_SE`. If you enter a language code that is not supported, SCAYT will fall back to default American English.

Dialog Windows

How Do I Change the Contents of a CKEditor Dialog Window?

Contents

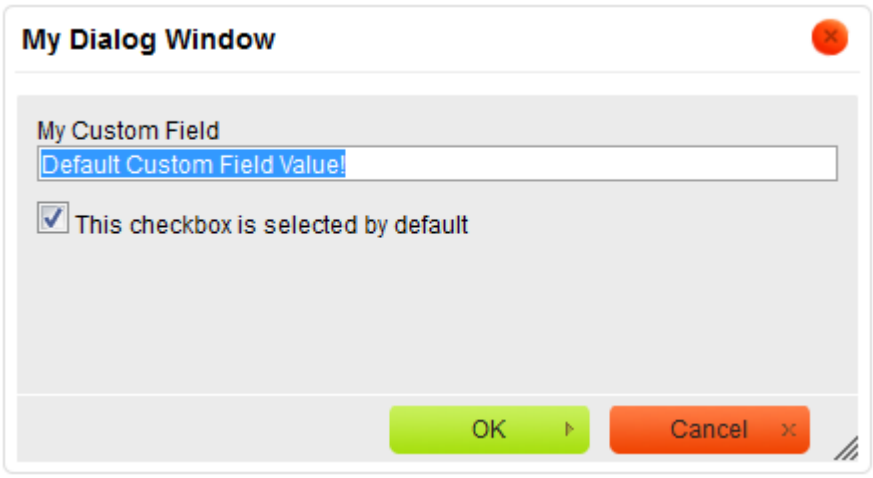
CKEditor allows you to customize dialog windows without changing the original editor code. For an example on how to add or remove dialog window tabs and fields refer to the **Using the JavaScript API to customize dialog windows** sample and its source code from your CKEditor installation.

How Do I Set a Default Value for a CKEditor Dialog Window Field?

In order to assign a default value to a dialog window field, use the 'default' parameter in the dialog window UI element definition.

```
elements: [
    {
        type: 'text',
        id: 'myCustomField',
        label: 'My Custom Field',
        'default': 'Default Custom Field Value!'
    },
    {
        type: 'checkbox',
        id: 'myCheckbox',
        label: 'This checkbox is selected by default',
        'default': true
    }
]
```

The code above creates the following UI elements in a sample dialog window tab.



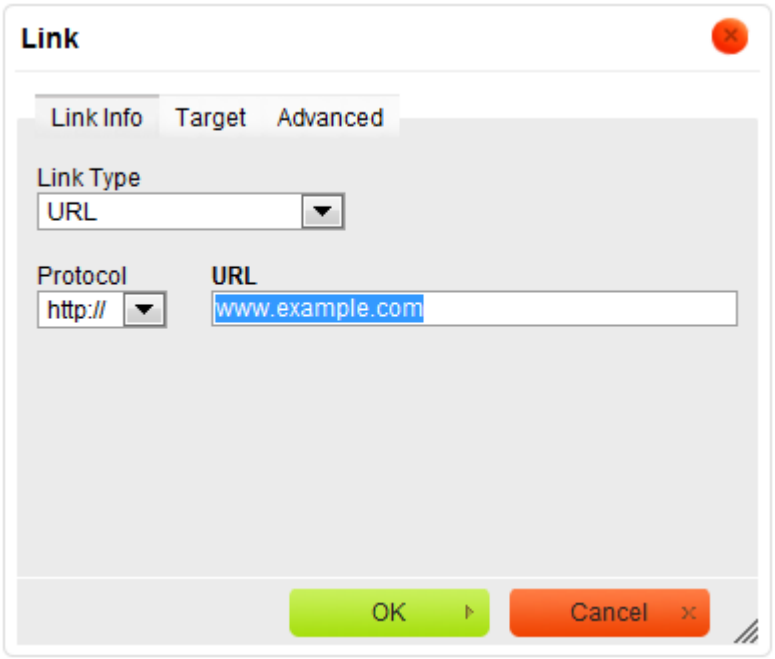
You can also customize existing dialog windows and give them default values. The following code sets the default **URL** field value for the **Link** dialog window.

```
CKEDITOR.on( 'dialogDefinition', function( ev ) {
    // Take the dialog name and its definition from the event data.
    var dialogName = ev.data.name;
    var dialogDefinition = ev.data.definition;

    // Check if the definition is from the dialog window you are interested in
(the "Link" dialog window).
    if ( dialogName == 'link' ) {
        // Get a reference to the "Link Info" tab.
        var infoTab = dialogDefinition.getContents( 'info' );

        // Set the default value for the URL field.
        var urlField = infoTab.get( 'url' );
        urlField[ 'default' ] = 'www.example.com';
    }
});
```

After this customization the **Link** dialog window will contain the `www.example.com` default value in the **URL** field.



For more examples on setting a default field value refer to the **Using the JavaScript API to customize dialog windows** sample and its source code from your CKEditor installation.

Note: Since in some old browsers `default` is a reserved word in JavaScript, remember to always put it in quotes when used in your code (`'default'`).

How Do I Set a Specific Dialog Window Tab to Open by Default?

If you want to change your CKEditor configuration to show a different tab on opening a dialog window, you can hook into the onShow event of the dialog window.

Firstly, you will need to know the names of the dialog window and the tab that you want to set as default, so use the Developer Tools plugin to get these.

Once you have the names you can add the following code into the page that contains your CKEditor instance. The

example below sets the **Image Properties** dialog window to open the **Advanced** tab by default.

```
CKEDITOR.on('dialogDefinition', function( ev ) {
    // Take the dialog window name and its definition from the event data.
    var dialogName = ev.data.name;
    var dialogDefinition = ev.data.definition;

    if ( dialogName == 'image' ) {
        dialogDefinition.onShow = function() {
            // This code will open the Advanced tab.
            this.selectPage( 'advanced' );
        };
    }
});
```

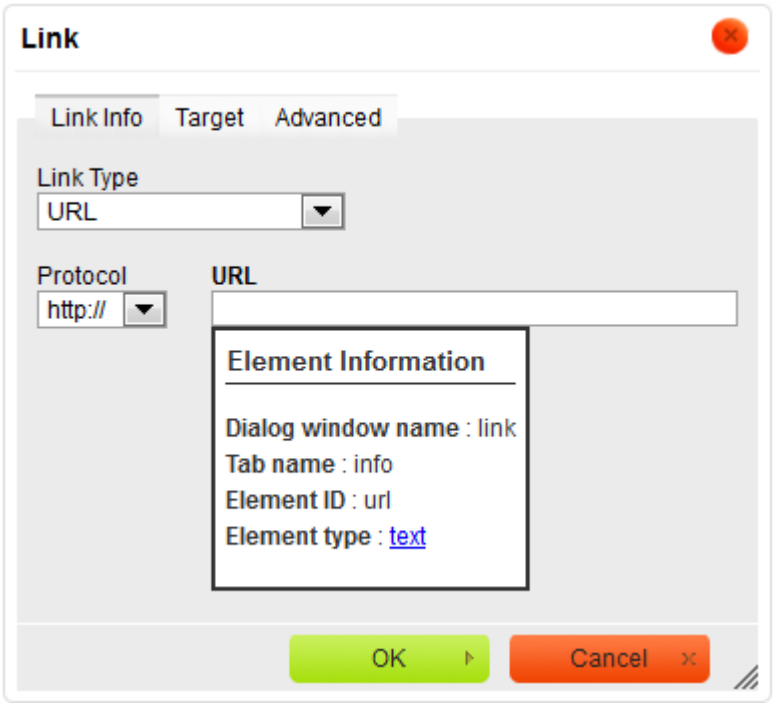
If, for example, you want to open the **Upload** tab first to make it more convenient for your users to use the (existing and previously integrated) file uploader, change the code in the following way:

```
// This code will open the Upload tab.
this.selectPage( 'Upload' );
```

How Do I Learn the Names of CKEditor Dialog Window Fields?

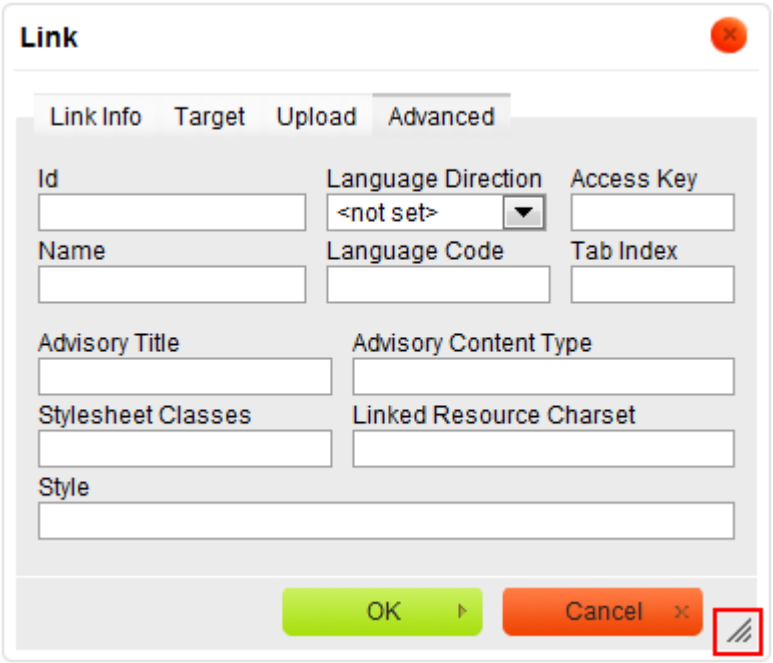
If you want to customize a dialog window, the easiest and most convenient way is to enable the Developer Tools plugin that displays the name and IDs of all dialog window elements when you hover them with your mouse.

The following figure shows the tooltip that describes the **URL** field of the **Link** dialog window that is displayed after the **Developer Tools** plugin was enabled.



How Do I Remove the Ability to Resize All CKEditor Dialog Windows?

Dialog windows of CKEditor can be resized by using the resizing grip located in the bottom right-hand corner of a dialog window (for RTL languages — in the bottom left-hand corner).



You can disable the resizing feature completely by setting the resizable parameter to CKEDITOR.DIALOG_RESIZE_NONE.

```
CKEDITOR.on( 'dialogDefinition', function( ev ) {
    ev.data.definition.resizable = CKEDITOR.DIALOG_RESIZE_NONE;
});
```

Use the CKEDITOR.DIALOG_RESIZE_WIDTH and CKEDITOR.DIALOG_RESIZE_HEIGHT values to enable resizing of a dialog window in one dimension only.

How Do I Remove the Ability to Resize Specific CKEditor Dialog Windows?

If you want to leave the resizing feature for some of the dialog windows and turn it off for others, you may define the value of the resizable parameter for each dialog window separately, like in the example below.

```
CKEDITOR.on( 'dialogDefinition', function( ev ) {
    if ( ev.data.name == 'link' )
        ev.data.definition.resizable = CKEDITOR.DIALOG_RESIZE_NONE;
    else if ( ev.data.name == 'image' )
        ev.data.definition.resizable = CKEDITOR.DIALOG_RESIZE_HEIGHT;
});
```

Use the CKEDITOR.DIALOG_RESIZE_WIDTH and CKEDITOR.DIALOG_RESIZE_HEIGHT values to enable resizing of a dialog window in one dimension only.

Miscellaneous

How Do I Read or Write the Contents of CKEditor from JavaScript?

Contents

If you want to read CKEditor contents, use the getData method.

If you want to write some content into CKEditor, use the setData method.

An example of how to use these functions can be found in the **Basic usage of the API** sample (`samples/api.html`) located in the `samples/` directory of CKEditor installation package.

How Do I Compress CKEditor Source Code After Customization?

Check the Build from Source Code page.

Bugs and New Features

Contents

How Do I Report CKEditor Bugs?

The Reporting Issues session of this guide brings all the necessary information on how to properly report issues.

How Do I Request New CKEditor Features?

If you feel there is an interesting **feature missing in CKEditor**, go to the development site: http://dev.ckeditor.com/ and file a ticket.

Please make sure you read the Bug Reporting Instructions and Ticket Specs articles beforehand in order to make it easier for the developers to examine the issue. It is also recommended to start from a search at the community forum to see whether the proposed functionality has not been implemented yet and offered as a plugin or code sample. You can also start a discussion to see whether the proposed feature could gain some interest from other community members.

When submitting a new ticket, feel free to attach your code suggestions or patches that can serve as a prototype for the requested feature.