WYMeditor Documentation

Release 1.0.0-beta.8

Jean-Francois Hovinne

September 07, 2014

Contents

WYMeditor is an open source, web-based WYSIWYM editor that allows non-technical users to create clean, semantic, standards-compliant HTML. The WYM-part stands for "What You Mean" compared to the more common "What You See Is What You Get".

CHAPTER 1

Browser Compatibility

WYMeditor is compatible with:

- IE: 7, 8, 9 ,10 and 11
- Firefox: LTS and latest two major versions
- Opera: Latest Version
- Safari: Latest Version
- Google Chrome: Latest two major versions

CHAPTER 2

Examples

WYMeditor has a lot of flexibility. Check out our Examples.

CHAPTER 3

Contents

3.1 Getting Started With WYMeditor

3.1.1 Setting up WYMeditor

Quick Start Instructions

Include jQuery

WYMeditor requires a version of jQuery between 1.4.4 and 2.1.x.

Make sure that your page includes it.

Note: With jQuery 2.x and newer, there is no support for IE8 and older.

Download a Pre-built Release of WYMeditor

Download a pre-built WYMeditor release from the WYMeditor Releases Github Page. Extract the contents of the archive in to a folder in your project (eg. `media/js/wymeditor`).

Or install via Bower

WYMeditor is available via Bower.

jQuery 2.x does not support IE8 and older.

WYMeditor does support IE7 and IE8.

WYMeditor's Bower manifest defines a range of jQuery versions as a dependency.

The latest jQuery version in this range is the newest jQuery version that still supports these browsers.

WYMeditor does support jQuery 2.x, with the acknowledgement that it will not function in IE7 and IE8.

If you decide to use jQuery 2.x, please feel free to override the top limit, while making sure you supply a jQuery version that WYMeditor supports.

See *Include jQuery* for WYMeditor's range of supported jQuery versions.

Source the WYMeditor Javascript

Include the `wymeditor/jquery.wymeditor.min.js` file on your page. This file will pull in anything else that's required.

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title>WYMeditor Quickstart</title>
        <script type="text/javascript" src="jquery/jquery.js"></script>
        <script type="text/javascript" src="wymeditor/jquery.wymeditor.min.js"></script>
    </head>
    <body>
        ... SNIP
    </body>
</html>
```

Create a `textarea` for the Editor

WYMeditor instances are tied to `textarea` inputs.

The `textarea` provides a few things:

- Its text is used as the initial HTML inside the editor.
- Whenever `xhtml()` is called on the editor, the resulting parsed html is placed inside the (hidden) `textarea`.
- The editor UI appears in the same location previously occupied by the `textarea`.

Let's create a `textarea` and give it a `submit` button with the `wymupdate` class. Let's also start with some existing content.

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title>WYMeditor Quickstart</title>
        <script type="text/javascript" src="jquery/jquery.js"></script>
        <script type="text/javascript" src="wymeditor/jquery.wymeditor.min.js"></script>
    </head>
    <body>
        <form method="post" action="">
            <textarea id="my-wymeditor"><p>I'm initial content!</p></textarea>
            <input type="submit" class="wymupdate" />
        </form>
    </body>
</html>
```

Note: The `wymupdate` class is just a convenience so that your `textarea` automatically receives the updated HTML on form submit. Otherwise, you'll need to call `xhtml()` yourself.

Use `wymeditor()` to Create an Editor

Creating a WYMeditor editor instance happens via a jQuery plugin, aptly named `wymeditor`, that you call on a `textarea` element.

Let's use the `wymeditor()` function to select the `my-wymeditor` `textarea` element and turn it in to a WYMeditor instance.

```
$(document).ready(function() {
    $('#my-wymeditor').wymeditor();
});
```

Note: We use the `$(document).ready` to wait until the DOM is loaded. Most users will want to do this, but it's not strictly necessary.

See *Anatomy of an Editor Initialization* for more details.

All Together Now

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title>WYMeditor Quickstart</title>
        <script type="text/javascript" src="jquery/jquery.js"></script>
        <script type="text/javascript" src="wymeditor/jquery.wymeditor.min.js"></script>
    </head>
    <body>
        <form method="post" action="">
            <textarea id="my-wymeditor"><p>I'm initial content!</p></textarea>
            <input type="submit" class="wymupdate" />
        </form>
        <script type="text/javascript">
            $(document).ready(function() {
                $('#my-wymeditor').wymeditor();
            });
        </script>
    </body>
</html>
```

Troubleshooting

If things aren't behaving as you'd expect, the first step is to open your browser's development tools. Chrome, Firefox and recent IE all have acceptable versions. Look for error messages and 404s retrieving files.

It's also a good idea to compare your code to some of the *Contributed Examples and Cookbooks*.

Security Errors Because WYMeditor is based on an iframe, there are restrictions about loading files across domains. That means that you need to serve the WYMeditor media from your current domain.

404s Loading Files WYMeditor automagically detects the paths of required CSS and JS files. You'll need to initialize `basePath`, `cssPath` and `jQueryPath` if you don't use default file names. Those are `jquery.wymeditor.js`, `wymeditor/skins/{skin name}/screen.css`, and `jquery.js`, respectively.

For details, see *Customizing WYMeditor*.

Example

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title>WYMeditor Quickstart</title>
        <script type="text/javascript" src="jquery/jquery.js"></script>
        <script type="text/javascript" src="wymeditor/jquery.wymeditor.min.js"></script>
    </head>
    <body>
        <form method="post" action="">
            <textarea id="my-wymeditor"><p>I'm initial content!</p></textarea>
            <input type="submit" class="wymupdate" />
        </form>
        <script type="text/javascript">
            $(document).ready(function() {
                $('#my-wymeditor').wymeditor();
            });
        </script>
    </body>
</html>
```

3.1.2 WYMeditor Integrations

Most developers who encounter WYMeditor will do so via another tool, whether that's a CMS, a web framework or another project. In an effort to encourage code re-use, we try to keep tabs on the projects that help you integrate WYMeditor.

Content Management Systems

Django CMS

django CMS

Refinery CMS

Refinery CMS is an awesome Rails-based CMS.

Drupal

Drupal

- drupal-wymeditor via the Wysiwyg module

Frog CMS

Frog CMS <http://www.madebyfrog.com/>

- frog_wymeditor

MediaWiki

MediaWiki

- MeanEditor

Midgard CMS Framework

Bundled with the Midgard CMS Framework

Web Frameworks

Jelix

Jelix

- One of the included html editor options

Open Source Projects

Adminer

Adminer: Database management in a single PHP file.

- Available as an included plugin

Commercial Projects

Kickstarter

Kickstarter is a funding platform for creative projects. As of August 2013, they had raised over $700 million in pledges for more than 45000 creative projects.

They use WYMeditor for their project "Story" editor.

PolicyStat

PolicyStat is a policy and procedure management system, primarily targeted towards healthcare.

They use WYMeditor exclusively as their document editor, logging more than 1.25 million editor sessions to date.

3.2 Using WYMeditor

3.2.1 WYMeditor Skins

Loading a Skin

The default WYMeditor distribution now includes all skin javascript and CSS as part of the bundle. If you're using an existing distribution of WYMeditor, and wish to use another one of the bundled skins, you simply need to set the

`skin` option.

If you're migrating from a version of WYMeditor before 1.0, the differences are explained in the migration documentation on *No More Skin Auto-loading*.

Third-party Skins and Skins in Development

When doing WYMeditor development or using a non-bundled skin, your desired skin won't be included in the single bundles of WYMeditor javascript and CSS. To use a skin then, simply include the appropriate CSS and Javascript files on the page before you initialize the editor.

Optimizations

For enhanced optimization, you can create your own WYMeditor bundle only containing the skin that you will load, but that will be a very low-impact optimization for most users, as the amount of CSS/Javascript in a skin is very small relative to the impact of WYMeditor itself.

Included Skins

Seamless

Known Issues

Window doesn't shrink for IE8+ We haven't found a reliable way to detect that the editor window needs to shrink after content has been removed for IE8 and higher.

Usability problems with very-wide tables/images If there are wide tables/images, things get a little goofy. The user must use the arrow keys to navigate within the table.

To fix this, we need to set widths for headers/paragraphs/etc and a min-width for the whole iframe, but allow the width to increase if needed. That will provide a whole-page horizontal scrollbar in the case of wide tables, which is probably better than current behavior.

Frame centering problems in IE IE has some vertical centering issues:

- IE7: Too much space at the top and not enough at the bottom
- IE8: Too much space at the top and not enough at the bottom
- IE9: Too much space at the bottom and not enough at the top

IE8 has some horizontal centering issues:

With full-line content on initialization, IE8 has too much space on the left and not enough on the right.

View HTML box not affixed When the box is open to view HTML, it should float under the toolbar instead of only being visible at the top. Otherwise, it's kind of annoying to edit HTML for large documents.

Toolbar UI Tweaks Needed

- Give the toolbar a gray background so that it contrasts with the page and with the content blocks. That will require changing the background of the containers/classes dropdowns.
- Round the top corners of the toolbar.
- Round the corners for the container/classes dropdowns.
- Round the bottom corners of the iframe body to match the toolbar top.

Window doesn't grow for inserted images If you insert an image, the editor height doesn't automatically update.

Third-Party Skins

wymeditor-refine

wymeditor-refine is a skin for WYMEditor extracted and tweaked from Refinery CMS

3.2.2 Using WYMeditor Plugins

3.2.3 Common WYMeditor Options

3.2.4 Getting Help

- **Wiki/Docs:** https://github.com/wymeditor/wymeditor/wiki
- **Forum:** http://community.wymeditor.org
- **Issue tracking:** https://github.com/wymeditor/wymeditor/issues
- **Official branch:** https://github.com/wymeditor/wymeditor

To Read more on contributing, see *Contributing*.

3.2.5 Anatomy of an Editor Initialization

So what actually happens when you call `$('#my-textarea').wymeditor()`?

1. Your existing `textarea` is hidden.
2. An iframe is created in that location.
3. The attribute `data-wym-initialized` is added to the `textarea`.
4. The WYMeditor toolbars and interface load.
5. The iframe is initialized with the content from your `textarea`.

3.3 Customizing WYMeditor

Note: If you're more of a "Learning by Doing" person, there are tons of hosted examples to try out if you're looking for ideas. You can use `View Source` from there or look in the `examples/` directory of your WYMeditor repository or distribution.

3.3.1 Basics of Customization

There are four primary methods of customizing WYMeditor. All of them come down to passing in options on editor initialization and possibly including extra javascript and CSS on your page. The methods are:

1. Using WYMeditor Initialization Options

Using WYMeditor Options is the easiest method of customization and the most common. By changing the proper *WYMeditor Options*, you can change the user interface's skin, choose from a wide range of plugins, add or remove items from the toolbar, include custom CSS classes for your users to choose and lots more.

2. Using a WYMeditor "Skin"

We use *WYMeditor Skins* to package up user interface tweaks and customizations. There are many included options from which to choose along with the ability to write your own.

3. Using WYMeditor Plugins

By *Using WYMeditor Plugins* you can enable a broad range of extra behaviors. If none of the *Included Plugins with Download* meet your needs, there's a range of *Third Party Plugins* available.

Note: Still can't find what you need? No problem. We have documentation for `writing_plugins/index` too.

---

4. Using a Custom Iframe

If you'd like to make tweaks to the way that your content looks inside the editor (the part with the blue background), then you can also choose a custom iframe. Check out the `wymeditor/iframes` directory for some existing options, or roll your own.

3.3.2 Using WYMeditor Options

All *WYMeditor Options* are set by passing an options object as the only parameter to the WYMeditor initialization function.

```
$(".my-wymeditor-class").wymeditor(options)
```

Example: Start with Existing HTML and Alert on Load

Let's say you want to edit some existing HTML and then annoy your users with an alert box. You've got a mean streak, but we can absolutely do that.

The *postInit* option lets us define a callback after initialization and the editor is automatically initialized with the contents of its `textarea` element. Our options object should look like this:

```
var options = {
    postInit: function() {
        alert('OK');
    }
};
```

Full HTML

Our stripped-down full example then looks like this:

```
<html>
<head>
    <link rel="stylesheet" type="text/css" href="wymeditor/wymeditor.css" />
</head>

<body>
    <form method="post" action="">
        <textarea class="my-wymeditor-class">
            &lt;p&gt;Hello, World!&lt;/p&gt;
        </textarea>
        <input type="submit" class="wymupdate" />
    </form>

    <script type="text/javascript" src="jquery/jquery.js"></script>
    <script type="text/javascript" src="wymeditor/jquery.wymeditor.js"></script>
    <script type="text/javascript">
        $(function() {
            var options = {
                postInit: function() {
                    alert('OK');
                }
            };
            jQuery(".wymeditor").wymeditor(options);
    </script>
</body>
</html>
```

3.3.3 WYMeditor Options

Note: All options and their defaults are located in `wymeditor/core.js` in the `jQuery.fn.wymeditor` method. Though if they're not documented here, they're subject to change between releases without following our deprecation policy.

Common Options

`html`

Initializes the editor's HTML content.

Example

```
html: "<p>Hello, World!<\/p>"
```

As an alternative, you can just fill the textarea WYMeditor is replacing with the initial HTML.

`lang`

The language to use with WYMeditor. Default is English (en). Codes are in ISO-639-1 format.

Language packs are stored in the `wymeditor/lang` directory.

Example: Choosing the Polish Translation To use the Polish translation instead, use the value `pl`.

```
$('.wymeditor').wymeditor({
    lang: 'pl'
});
```

toolsItems

An array of tools buttons, inserted in the tools panel, in the form of:

```
[
    {
        'name': 'value',
        'title': 'value',
        'css': 'value'
    }
]
```

Example

```
toolsItems: [
  {'name': 'Bold', 'title': 'Strong', 'css': 'wym_tools_strong'},
  {'name': 'Italic', 'title': 'Emphasis', 'css': 'wym_tools_emphasis'}
]
```

Default value

```
toolsItems: [
  {'name': 'Bold', 'title': 'Strong', 'css': 'wym_tools_strong'},
  {'name': 'Italic', 'title': 'Emphasis', 'css': 'wym_tools_emphasis'},
  {'name': 'Superscript', 'title': 'Superscript', 'css': 'wym_tools_superscript'},
  {'name': 'Subscript', 'title': 'Subscript', 'css': 'wym_tools_subscript'},
  {'name': 'InsertOrderedList', 'title': 'Ordered_List', 'css': 'wym_tools_ordered_list'},
  {'name': 'InsertUnorderedList', 'title': 'Unordered_List', 'css': 'wym_tools_unordered_list'},
  {'name': 'Indent', 'title': 'Indent', 'css': 'wym_tools_indent'},
  {'name': 'Outdent', 'title': 'Outdent', 'css': 'wym_tools_outdent'},
  {'name': 'Undo', 'title': 'Undo', 'css': 'wym_tools_undo'},
  {'name': 'Redo', 'title': 'Redo', 'css': 'wym_tools_redo'},
  {'name': 'CreateLink', 'title': 'Link', 'css': 'wym_tools_link'},
  {'name': 'Unlink', 'title': 'Unlink', 'css': 'wym_tools_unlink'},
  {'name': 'InsertImage', 'title': 'Image', 'css': 'wym_tools_image'},
  {'name': 'InsertTable', 'title': 'Table', 'css': 'wym_tools_table'},
  {'name': 'Paste', 'title': 'Paste_From_Word', 'css': 'wym_tools_paste'},
  {'name': 'ToggleHtml', 'title': 'HTML', 'css': 'wym_tools_html'},
  {'name': 'Preview', 'title': 'Preview', 'css': 'wym_tools_preview'}
]
```

containersItems

An array of containers buttons, inserted in the containers panel in the form of:

```
[
    {
        'name': 'value',
        'title': 'value',
        'css': 'value'
    }
]
```

Example

```
containersItems: [
  {'name': 'P', 'title': 'Paragraph', 'css': 'wym_containers_p'},
  {'name': 'H1', 'title': 'Heading_1', 'css': 'wym_containers_h1'}
]
```

classesItems

An array of classes buttons, inserted in the classes panel in the form of:

```
[
    {
        'name': 'value',
        'title': 'value',
        'expr': 'value'
    }
]
```

`expr` is a jQuery selector that allows you to control to which elements the class defined by `name` can be applied. Only elements matching the selector `expr` will be given the class on a user's click.

Example Let's support adding the class `date` to paragraphs, and the class `hidden-note` to paragraphs that don't already have the class `important`.

```
classesItems: [
  {'name': 'date', 'title': 'PARA: Date', 'expr': 'p'},
  {'name': 'hidden-note', 'title': 'PARA: Hidden note', 'expr': 'p[@class!="important"]'}
]
```

preInit

A custom function which will be executed directly before WYMeditor's initialization.

Parameters

- wym: the WYMeditor instance

preBind

A custom function which will be executed directly before handlers are bound on events (e.g. buttons click).

Parameters

- wym: the WYMeditor instance

postInit

A custom function which will be executed once, when WYMeditor is ready.

Parameters

- wym: the WYMeditor instance

Example

```
postInit: function(wym) {
  //activate the 'tidy' plugin, which cleans up the HTML
  //'wym' is the WYMeditor instance
  var wymtidy = wym.tidy();
  wymtidy.init();
}
```

Uncommon Options

Most of these options are only required in abnormal deployments, or useful for deep customization.

Note: Any options not documented are considered private and are subject to change between releases without following the deprecation policy.

---

basePath

WYMeditor's relative/absolute base path (including the trailing slash). Until we remove our pop-up window dialogs, this is used to load necessary javascript in those dialog windows.

This value is automatically guessed by `computeWymPath`, which looks for the `script` element loading `jquery.wymeditor.js`.

Example

```
basePath: "/admin/edit/wymeditor/"
```

wymPath

WYMeditor's main JS file path.

Similarly to *basePath*, this value is automatically guessed by `computeBasePath`.

iframeBasePath

The path to the directory containing the iframe that will be initialized inside the editor body itself.

This value is automatically guessed, based on the `basePath` value.

jQueryPath

This contains the full path or URL to a copy of the jQuery library. Like *options-basePath*, we only need to know this path in order to load it inside pop-up dialogs.

Also like *basePath*, this value is automatically guessed, but in this case by `computeJqueryPath()`.

Example

```
jQueryPath: "/js/jquery.js"
```

updateSelector and updateEvent

Note: Will be removed in 1.0. Instead, users should do their own event handling/registration and make a call to `wym.update()`.

Allows you to update the value of the element replaced by WYMeditor (typically a `textarea`) with the editor's content on .while e.g. clicking on a button in your page.

preInitDialog

A custom function which will be executed directly before a dialog's initialization.

Parameters

1. wym: the WYMeditor instance
2. wdw: the dialog's window object

postInitDialog

A custom function which will be executed directly after a dialog is ready.

Parameters

1. wym: the WYMeditor instance
2. wdw: the dialog's window object

UI Customization Options

These options allow deep customization of the structure of WYMeditor's user interface by changing the HTML templates used to generate various UI components. Rather than using these, most user's prefer *WYMeditor Skins*.

List of HTML Template Options

- boxHtml
- logoHtml
- iframeHtml

- toolsHtml
- toolsItemHtml
- containersHtml
- containersItemHtml
- classesHtml
- classesItemHtml
- statusHtml
- htmlHtml
- dialogHtml
- dialogLinkHtml
- dialogFeatures The dialogs' features. e.g.

```
dialogFeatures: "menubar=no,titlebar=no,toolbar=no,resizable=no,width=560,height=300,top=0,l
```

- dialogImageHtml
- dialogTableHtml
- dialogPasteHtml
- dialogPreviewHtml

UI Selectors Options

These selectors are used internally by WYMeditor to bind events and control the user interface. You'll probably only want to modify them if you've already changed one of the *UI Customization Options*.

List of Selectors

- boxSelector
- toolsSelector
- toolsListSelector
- containersSelector
- classesSelector
- htmlSelector
- iframeSelector
- statusSelector
- toolSelector
- containerSelector
- classSelector
- htmlValSelector
- hrefSelector
- srcSelector

- titleSelector
- altSelector
- textSelector
- rowsSelector
- colsSelector
- captionSelector
- submitSelector
- cancelSelector
- previewSelector
- dialogLinkSelector
- dialogImageSelector
- dialogTableSelector
- dialogPasteSelector
- dialogPreviewSelector
- updateSelector

Example Selector Customization

```
classesSelector: ".wym_classes"
```

3.3.4 Upgrading to Version 1

Using the Legacy Skin and Editor Styles

WYMeditor version 1.0 made some significant changes to the default skin and editor styles. For folks who need the old UI, we've included a skin and iframe to meet their needs.

These are both labeled as `legacy`. To use them, first load the legacy skins CSS and javascript, then choose those options on editor initialization:

```
$(function() {
    $('.wymeditor').wymeditor({
        iframeBasePath: "../wymeditor/iframe/legacy/",
        skin: "legacy"
    });
});
```

See Example 21-legacy for a demonstration of the Pre-1.0 user interface.

Deprecation

It's easy to provide your own skin and iframe, though, so these will be removed according to WYMeditor's deprecation policy.

No More Skin Auto-loading

Versions of WYMeditor prior to 1.0 would use javascript to automatically load your chosen skin's javascript and CSS. While this was a small first-usage usability improvement, it created some "magic" that quickly became confusing when it came time for an optimized, production-ready deployment. In production, you should be looking to reduce the number of HTTP requests as much as possible, which means including the skin's assets along with your other combined/minified/compressed assets.

The default WYMeditor distribution now includes all skin javascript and CSS as part of the bundle. They're only activated based on your choice of `skin` option, though. For enhanced optimization, you can create your own WYMeditor bundle only containing the skin that you will load, but that will be a very low-impact optimization for most users, as the amount of CSS/Javascript in a skin is very small relative to the impact of WYMeditor itself.

For more details, see the documentation on *Loading a Skin*.

Options Removed

- `skinPath`

No More CSS Configuration Options

Versions prior to 1.0 had various options that supported the ability to set CSS rules on various editor and dialog components. In the spirit of *No More Skin Auto-loading* and moving WYMeditor closer to just an idiomatic collection of javascript/css/html, we're no longer supporting those options. All of the things that were previously accomplished with them can be better-accomplished by actually including those rules in stylesheets.

If you're having difficulty determining the best strategy for migration please open an Issue on Github and we'll be happy to document your use case and help you with a plan.

Options Removed

- `styles`
- `stylesheet`
- `editorStyles`
- `dialogStyles`

Methods Removed

- `addCssRules()`
- `addCssRule()`

No More Language Automatic Loading

Instead of doing an additional HTTP request to load a language file, the default WYMeditor distribution comes bundled with all of the translation files. If you're creating your own bundle, you'll need to include those files on the page before the editor is initialized.

Options Removed

- `langPath`

Contributed Examples and Cookbooks

Running the Examples Locally Each release of WYMeditor comes bundled with a set of examples (available online here). To run these locally, they need to be served on a proper web server. If not, WYMeditor might not work properly as most modern browsers block certain JavaScript functionality for local files.

If you've finished *Configuring Your Development Environment*, then serving the examples is easy:

```
vagrant:~/wym$ grunt server:dist
$ google-chrome "http://localhost:9000/examples/"
```

Examples Table of Contents

Integrate WYMeditor in Django Administration I have managed to easily integrate WYMeditor into Django's administrative interface.

Here is how I did it...

First I copied the wymeditor to my project's static-served files directory, which in my case had an URL prefix of `/site/media/`

There I put the `wymeditor`, `jquery` and a special file `admin_textarea.js`, that I have written myself consisting of:

```
$(document).ready(function() {
    $('head', document).append('<link rel="stylesheet" type="text/css" media="screen" href="/site/me
    $("textarea").wymeditor({
        updateSelector: "input:submit",
        updateEvent:    "click"
    });
});
```

This file instructs the browser to load an additional WYMeditor's CSS and to convert each `<textarea>` HTML tag into a WYMeditor.

In each of the Django models that I whished to set WYMeditor for, I have added the following `js` setting to the Admin section:

```
class ExampleModel(models.Model):
    text = models.TextField()      # each TextField will have WYM editing enabled
    class Admin:
         js = ('/site/media/jquery.js',
               '/site/media/wymeditor/jquery.wymeditor.js',
               '/site/media/admin_textarea.js')
```

That is it. If you wish to use WYM in your own Django app, just follow the steps and replace the `/site/media/...` with whatever your static media prefix is.

Using the filebrowser Application To integrate Wymeditor with the Django filebrowser, put the code below in a file, set the `fb_url` variable to point to your filebrowser instance and add the file to your Javascript headers:

```
<script type="text/javascript" src="/media/wymeditor/plugins/jquery.wymeditor.filebrowser.js"></scrip
```

or in your admin.py:

```
class Media:
    js = ('/media/wymeditor/plugins/jquery.wymeditor.filebrowser.js',)
```

Add the `postInitDialog` parameter to the Wymeditor initialization:

```
$('textarea').wymeditor({
  postInitDialog: wymeditor_filebrowser
});
```

If you already have a `postInitDialog` function, you need to put a call to `wymeditor_filebrowser` inside that function:

```
$('textarea').wymeditor({
  postInitDialog: function (wym, wdw) {
      // Your code here...

      // Filebrowser callback
      wymeditor_filebrowser(wym, wdw);
  }
});
```

Then you should be able to click on the Filebrowser link to select an image.

Code:

```
wymeditor_filebrowser = function(wym, wdw) {
  // the URL to the Django filebrowser, depends on your URLconf
  var fb_url = '/admin/filebrowser/';

  var dlg = jQuery(wdw.document.body);
  if (dlg.hasClass('wym_dialog_image')) {
    // this is an image dialog
    dlg.find('.wym_src').css('width', '200px').attr('id', 'filebrowser')
      .after('<a id="fb_link" title="Filebrowser" href="#">Filebrowser</a>');
    dlg.find('fieldset')
      .append('<a id="link_filebrowser"><img id="image_filebrowser" /></a>' +
              '<br /><span id="help_filebrowser"></span>');
    dlg.find('#fb_link')
      .click(function() {
        fb_window = wdw.open(fb_url + '?pop=1', 'filebrowser', 'height=600,width=840,resizable=yes,sc
        fb_window.focus();
        return false;
      });
  }
}
```

Image Gallery Implementation Example The purpose of this page is to explain how an image gallery built with jQuery's jCarousel plugin and based on data loaded via AJAX can be integrated into WYMEditor. The example is built using CodeIgniter (CI) for structure and queries. It uses a series of JS includes in the image dialog, a CI controller function called via AJAX, and a CI model function containing an active record query on an images database. This example presupposes that you have a functioning install of WYMEditor v0.4 and you are relatively familiar with hacking it. It also assumes that you have the source files for and know how to implement an image gallery based on jCarousel. More information and source downloads for jCarousel can be found here: http://sorgalla.com/jcarousel/

1. Including Plugin Code You will need to include 3 js files and 2 css files into the dialogs' HTML. This is done by using the `dialogHtml` option, which makes up the outer shell of the dialog boxes that are opened by the top buttons in WYMeditor. You will need to change the paths here to match up with whatever your setup is:

```
$('.wymeditor').wymeditor({

  //options

  dialogHtml: "<!DOCTYPE html PUBLIC '-//W3C//DTD XHTML 1.0 Strict//EN'"
    + " 'http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd'>"
    + "<html><head>"
    + "<link rel='stylesheet' type='text/css' media='screen'"
    + " href='"
    + WYM_CSS_PATH
    + "' />"
    + "<title>"
    + WYM_DIALOG_TITLE
    + "</title>"
    + "<script type='text/javascript'"
    + " src='"
    + WYM_JQUERY_PATH
    + "'></script>"
    + "<script type='text/javascript'"
    + " src='"
    + WYM_BASE_PATH
    + "jquery.wymeditor.js'></script>"
    + "<script type='text/javascript' src='/scripts/jquery.imager.js'></script>"
    + "<script type='text/javascript' src='/scripts/easing.js'></script>"
    + "<script type='text/javascript' src='/scripts/jquery.jcarousel.js'></script>"
    + "<link rel='stylesheet' type='text/css' href='/css/jquery.jcarousel.css' />"
    + "<link rel='stylesheet' type='text/css' href='/css/skins/tango/skin.css' />"
    + "<style type='text/css'>"
    + ".jcarousel-skin-tango.jcarousel-container-horizontal { width: 85%; }"
    + ".jcarousel-skin-tango .jcarousel-clip-horizontal { width: 100%; }"
    + "</style>"
    + "</head>"
    + WYM_DIALOG_BODY
    + "</html>",

  dialogImageHtml:  "<body class='wym_dialog wym_dialog_image'"
    + " onload='WYM_INIT_DIALOG(" + WYM_INDEX + ")'"
    + ">"
    + "<form>"
    + "<fieldset>"
    + "<legend>{Image}</legend>"
    + "<div class='row'>"
    + "<label>{URL}</label>"
    + "<input type='text' class='wym_src' value='' size='40' />"
    + "</div>"
    + "<div class='row'>"
    + "<label>{Alternative_Text}</label>"
    + "<input type='text' class='wym_alt' value='' size='40' />"
    + "</div>"
    + "<div class='row'>"
    + "<label>{Title}</label>"
    + "<input type='text' class='wym_title' value='' size='40' />"
    + "</div>"
    + "<div class='row row-indent'>"
    + "<input class='wym_submit' type='button'"
```

```
    + " value='{Submit}' />"
    + "<input class='wym_cancel' type='button'"
    + "value='{Cancel}' />"
    + "</div>"
    + "</fieldset>"
    + "</form>"
    + "<div id='gallery'><h2>Loading images, please wait...</h2></div>"
    + "</body>"

  //other options

});
```

Note: Options specified when calling a new WYMEditor instance must be separated by commas. If you add in some other options to the example shown above, you will need to add a comma to the end of the dialogImageHtml assignment as well as all the rest of the new options except the last one.

jquery.js, easing.js, and jquery.jcarousel.js are all pre-built components that can be downloaded from various sites - see the jCarousel link above. jquery.imager.js is a custom script that will be built below. jquery.jcarousel.css and skins/tango/skin.css are parts of jCarousel, please refer to the jCarousel link above for more information. The header styles are used to make jCarousel expand / contract based on the width of the dialog window.

2. Adding Injection Target to Dialog Body OK, now all of our scripts should be in place. Upload the file, refresh your WYMEditor install and open the image dialog; you should see all that stuff in the head of the document. It's usually a good idea to copy the paths out of the source here, paste them into another browser window, and make sure they open; this is just to make sure you have the paths correct and all the files are in the right place.

The next step is to add a target `div` to the image dialog body HTML. This gives us a place to inject our dynamic image list a bit later once we have built it with AJAX and PHP. In the `dialogImageHtml` string (around line 619) add the following line:

```
+ "<div id='gallery'><h2>Loading images, please wait...</h2></div>"
```

This line should go after the form but before the close body tag, like so:

```
+ "</div>"
+ "</fieldset>"
+ "</form>"
+ "<div id='gallery'><h2>Loading images, please wait...</h2></div>"
+ "</body>",
```

Once this is in place, upload and refresh WYMEditor again, then re-open the image dialog. You should now see a big fat "Loading images, please wait..." message underneath the form. OK now its time for the slick stuff.

3. AJAX Call from Javascript Make a new javascript file and save it as jquery.imager.js or whatever else you want to call it; just make sure it's included in the dialog HTML. Paste the following code into the js file:

```
// JavaScript Document

var $j = jQuery.noConflict();

$j(function() {
    //set up your AJAX call
    $.ajax({
      type: "POST",
      url: "http://www.your-domain.com/controller/ajaxer", //path to your PHP function
```

```
      data: "img=test&stamp=now",                                    //not required for this example,
      success: function(msg){                                              //trigger this code if t
        /*
            The PHP function needs to return an image UL with the following prototype:
            <ul id="mycarousel" class="jcarousel-skin-tango">
                <li><img src='http://www.test.com/upload/titan.jpg' width='68' height='60' alt='Tenn
                <li><img src='http://www.test.com/upload/canyon.jpg' width='68' height='60' alt='Gra
                <li><img src='http://www.test.com/upload/img_2_big.jpg' width='68' height='60' alt='
            </ul>

            The returning HTML is contained in the msg variable.
        */

        //inject the image list into the target div with ID of "gallery"
        $("div#gallery").html(msg);

         //Once the list is in place we can create a new instance of jCarousel and point it at the i
         //which has an ID of 'mycarousel'. For more information on these options see http://sorgall
        jQuery('#mycarousel').jcarousel({
            easing: 'backinout',
            visible: 5,
            animation: 500

        });

        //assign behaviors to the jCarousel thumbnails, triggered when they are clicked upon.
        $(".jcarousel-skin-tango img").click(function() {
            //$(this) is a reference to the thumbnail that got clicked
            $("input.wym_src").val($(this).attr('src'));            //inject the thumb's src attribu
            $("input.wym_alt").val($(this).attr('alt'));            //inject the thumb's alt attribu
            $("input.wym_title").val($(this).attr('title'));        //inject the thumb's title attri

            //loop through all the images and remove their "on" states if it exists
            $(".jcarousel-skin-tango img").each(function(i){
              $(this).removeClass("on");
            });
            //add "on" state to the selected image
            $(this).addClass("on").fadeIn('slow');
        })
      }
    });
});
```

Now you have a structure and behaviors for inserting your image code into the dialog. Now all you need is some images!

4. Database Structure I have an images mySQL table with the following structure:

```
img_id          int(11)
img_upload_date int(11)
img_upload_by   int(11)
img_name        varchar(64)
img_file_name   varchar(64)
img_size        float
img_width       int(11)
img_height      int(11)
img_string      varchar(64)
img_alt         varchar(255)
```

5. PHP / CodeIgniter Functions Basically you can make an AJAX call to any PHP page that will return a list of the images you want to display in the gallery. It can be connect to a database or not; that's up to you. For my particular setup, I have a CI function in my Content model called `get_images()` that returns either a list of all images in the DB or a specific image if you send a valid ID. The model function goes like this:

```
//ARGS: image ID (int) or -1 to get all images
function get_images($img_id)
{
    if($img_id != -1){ $this->db->where('img_id', $img_id); }
    $this->db->orderby('img_upload_date desc');
    $query = $this->db->get('tq_images');

    if($query->num_rows() > 0)
    {
        return $query->result_array();
    } else {
        return NULL;
    }
}
```

This will return either one or many rows of the database, ordered by the date the image was added, descending. If you send -1 as `$img_id` it will return all images; if you send it a number it will return a specific row if it's a valid ID. If the function can't find any rows based on what you sent it, it will return NULL. If you need more information about codeigniter or model functions see http://codeigniter.com/user_guide/.

6. Put it all Together Now you will make a controller function that is actually accessible via a browser or AJAX call. Open a CI controller and insert the following function:

```
function ajaxer()
{
    //pull in data from javascript AJAX call (not used for now)
    $img = $_POST['img'];
    $stamp = $_POST['stamp'];

    //call your model function
    $img = $this->Content->get_images(-1);
    //create a the return string. This is the structure for your jCarousel list.
    $lst = "<ul id='mycarousel' class='jcarousel-skin-tango'>\n";
    //loop through the images record set. This should be a list of all the images you want to displa
    foreach($img as $i)
    {
        //call a custom function in another model to format the date...you probly don't need this
        $date = $this->Page->get_date($i['img_upload_date']);
        //Build a list item for each image in the database. Insert values as needed.
        //This will produce an unordered list with the prototype specified in jquery.imager.js
        $lst .= "<li><img src='". base_url() . "upload/" . $i['img_file_name'] ."' width='68' height=

    }
    //close the list
    $lst .= "</ul>\n";
    //return the list to the dialog
    echo $lst;
}
```

And that's pretty much it. When you are setting up your AJAX call, its URL attribute should be pointed at this controller function. `ajaxer()` will call the model function outlined above then process the returned recordset into an HTML list. Echoing out the list will return it to your Javascript code as the `msg` variable I mentioned above. You should already have code in place in jquery.imager.js to handle the incoming data and inject it where it needs to go. In that same script you have already specified click behaviors for each thumbnail in the list. If you need more information about codeigniter or controller functions see http://codeigniter.com/user_guide/.

7. Using it Now upload everything and refresh WYMEditor. If you set everything up correctly, you should see your jCarousel load in underneath the dialog form. If you click on an image, you should see its values pop into the input boxes above the carousel. Once you have the correct information in the boxes, hit "submit" on the dialog as usual and WYMEditor should plop it into your piece. Repeat as needed.

If you have any questions about specific technologies used above, see their respective websites. If you have questions about my code, hit me up at rhinocerous at gmail dot com.

Integrate WYMeditor into sNews CMS This page explains step by step how to integrate WYMeditor version 0.4 into sNews version 1.6. sNews is a light and open source Content Management System. All information about sNews is available on its official site.

Since sNews already includes an editor, some hacks are mandatory in order to replace the default editor with WYMeditor. These hacks are all listed below. Fortunately, in order to avoid you these tedious code modifications, a ready-to-run archive file has been prepared. Unarchive this file and install sNews according to readme.html before to run it. Hacks made inside sNews have all been enclosed between tags [WYMeditor Hack] for allowing you to easily retreive them.

Install sNews **Download** archive file sNews16.zip

Unarchive sNews16.zip under your Web space and follow the install instructions contained into readme.html.

Install WYMeditor **Download** archive file wymeditor-0.4.tar.gz

Unarchive wymeditor-0.4.tar.gz into a temporary directory and move directories 'jquery' and 'wymeditor' into 'sNews' root directory

Integrate WYMeditor All the following code modifications must be made in **snews.php**.

Replace line 325

```
if ($_SESSION[db('website').'Logged_In'] == token()) {js();}
```

with

```
    if ($_SESSION[db('website').'Logged_In'] == token()) {js();
        echo
'   <link rel="stylesheet" type="text/css" media="screen" href="wymeditor/skins/default/screen.css" /
    <script type="text/javascript" src="jquery/jquery.js"></script>
    <script type="text/javascript" src="wymeditor/jquery.wymeditor.js"></script>
    <script type="text/javascript">
        jQuery(function() {
            jQuery(".wymeditor").wymeditor({
                cssPath: "'.db('website').'wymeditor/skins/default/screen.css",
                jQueryPath: "'.db('website').'jquery/jquery.js",
                wymPath: "'.db('website').'wymeditor/jquery.wymeditor.js"
            });
        });
```

```
    </script>';
    }
```

This code loads WYMeditor with its default skin and initializes WYMeditor with correct paths.

Replace line 1048

```
case 'textarea': $output = '<p>'.$lbl.':<br /><textarea name="'.$name.'" rows="'.$rows.'" cols="'.$co
```

with

```
case 'textarea':
    if ($_SESSION[db('website').'Logged_In'] == token()) {
        $output = '<p>'.$lbl.':<br /><textarea class="wymeditor" name="'.$name.'" rows="'.$rows.'" co
    } else {
        $output = '<p>'.$lbl.':<br /><textarea name="'.$name.'" rows="'.$rows.'" cols="'.$cols.'"'.$a
    }
    break;
```

This code allows WYMeditor to replace the 'textarea' block with a WYMeditor instance.

Comment from line 1312

```
/*  echo '<p>';
```

to line 1322

```
echo '</p>';*/[/code]
```

This code disables the default editor toolbar.

Replace line 1327

```
echo html_input('fieldset', '', '', '', '', '', '', '', '', '', '', '', '', '', '<a title="'.l('custo
```

with

```
if ($_SESSION[db('website').'Logged_In'] == token()) {
    echo html_input('fieldset', '', '', '', '', '', '', '', '', '', '', '', '', '', '<a title="'.l('c
} else {
    echo html_input('fieldset', '', '', '', '', '', '', '', '', '', '', '', '', '', '<a title="'.l('c
}
```

This code allows the edited article to be correctly previewed by sNews each time the sNews preview button is clicked.

Replace line 1393

```
echo html_input('submit', $frm_task, $frm_task, $frm_submit, '', 'button', '', '', '', '', '', '', ''
```

with

```
echo html_input('submit', $frm_task, $frm_task, $frm_submit, '', 'wymupdate', '', '', '', '', '', '',
```

This code allows the edited article to be correctly saved when the sNews save button is clicked.

That's all, sNews should now run WYMeditor instead of its default editor. If you experiment problems in running or using WYMeditor inside sNews, you are invited to read this topic on the WYMeditor forum or this one on the sNews forum. If you don't find the answer to your problem, feel free to post a new message.

Integrate WYMeditor into Rails WYM Editor Helper is a plugin that makes it dead easy to incorporate the WYM editor into your Rails views.

Follow these steps to use:

1. From your project's root, run:

```
$ ruby script/plugin install svn://zuurstof.openminds.be/home/kaizer/svn/rails_stuff/plugins
$ rake wym:install
```

2. Put `<%= wym_editor_initialize %>` in the view that will host the text editing form. Prefereably this goes into your html's HEAD, to keep our html W3C valid. Use `<% content_for :head do %> <%= wym_editor_initialize %> <% end %>` in the view that needs the editor, and `<%= yield :head %>` in the layout. This means the editor will only load when it is truly called for.
3. In your form, instead of i.e. `<%= text_area :article, :content %>`, use `<%= wym_editor :article, :content %>` OR add a `wymeditor` class to the `textarea`.
4. Add a `wymupdate` class to the submit button.

Extra Info This plugin uses an `svn:external` to automatically get the latest version of WYMeditor. If for some reason the checked out version is not working, you can install a different version like so:

```
$ svn export svn://svn.wymeditor.org/wymeditor/tags/0.4 vendor/plugins/wym_editor_helper/assets/wymed
```

To see what versions are available, check the repository browser at http://files.wymeditor.org/wymeditor/. Keep in mind that if you check out a newer version of WYM, you need to re-run the `wym:install` rake command to actually copy the wym files to the public dir. If you do so, be sure to first back up your configuration file (/javascripts/boot_wym.js) if you made any changes to it.

Updates on this plugin will appear on http://www.gorilla-webdesign.be/artikel/42-WYM+on+Rails

3.4 WYMeditor Plugins

3.4.1 Included Plugins with Download

Embed

Add description.

Fullscreen

Add description.

Hovertools

This plugin improves visual feedback by:

- displaying a tool's title in the status bar while the mouse hovers over it.
- changing background color of elements which match a condition, e.g. on which a class can be applied.

List

Add description.

RDFA

Add description.

Resizable

Add description.

Table

Add description.

Tidy

Add description.

Structured Headings

This plugin modifies the styling and function of headings in the editor to make them easier to use to structure documents. Currently in development.

Features

- Simplifies the process of adding headings to a document. Instead of having to choose between a heading 1-6 in the containers panel, those options are replaced with one single "Heading" option that inserts a heading at the same level as the preceding heading in the document.
- Makes it easier to adjust heading levels properly. In order to change the level of a heading, the indent and outdent tools can be used to lower and raise the level of the heading respectively. The indent tool will also prevent a user from indenting a heading more than one level below its preceding heading.
- Automatically numbers headings to better outline a document.
- Provides an optional tool that can be used to automatically fix improper heading structure in a document (although it's still in development).
- More features to come on further development.

Applying Structured Headings Styling outside of the Editor

The numbering added to headings in the editor is not parsed with the content of the document, but rather, it is added as styling using CSS in IE8+, Chrome, and Firefox, or it is added as separate content using a stand-alone JavaScript function with additional CSS styling in IE7.

To apply the heading numbering to a document outside of the editor, follow these steps:

1. Get the CSS for the structured headings. If you are using IE8+, Chrome, or Firefox, enter the `WYMeditor.printStructuredHeadingsCSS()` command in the browser console on the page of the editor using the structured headings plugin to print the CSS to the console. If you are using IE7, it uses different CSS than the other browsers, and you can get this CSS from its stylesheet `structured_headings_ie7_user.css` available in the plugin's directory.
2. Either copy this CSS to an existing stylesheet or make a new stylesheet with this CSS.

3. Apply the stylesheet with the CSS to all of the pages that contain documents that had heading numbering added to them in the editor.
4. If you are using IE7, in addition to the previous steps, add the script `jquery.wymeditor.structured_headings.js` available in the plugin's directory to all of the pages that contain documents that had heading numbering added to them in the editor. This script provides the function `numberHeadingsIE7()` which detects which headings on the page were assigned numbering in WYMeditor and adds the numbering for these headings into the page. This function should be called in a script on document ready after the `jquery.wymeditor.structured_headings.js` script is included on the page. Here is an example of what that script should look like:

```
jQuery(document).ready(function() { numberHeadingsIE7(); });
```

Browser Support

The plugin supports IE7+, Chrome, and Firefox, but it works less conveniently in IE7 as you can see by the additional steps it requires for implementation in that browser.

3.4.2 Third Party Plugins

Modal Dialog (by samuelcole)

https://github.com/samuelcole/modal_dialog

Replaces the default dialog behavior (new window) with a modal dialog. Known bug in IE, more information here.

Alignment (by Patabugen)

https://bitbucket.org/Patabugen/wymeditor-plugins/src

Set Text Alignment with classes.

Site Links (by Patabugen)

https://bitbucket.org/Patabugen/wymeditor-plugins/src

A plugin to add a dropdown of links to the Links dialog, especially for making it easier to link to your own site (or any other predefined set).

Can also add a File Upload form to let you upload files right from the Link dialog.

Image Float (by Patabugen)

https://bitbucket.org/Patabugen/wymeditor-plugins/src

Float images with classes.

Image Upload (by Patabugen)

https://bitbucket.org/Patabugen/wymeditor-plugins/src

Adds an Image Upload form to the Insert Image dialog.

Catch Paste

Force automatic "Paste From Word" usage so that all pasted content is properly cleaned.

http://forum.wymeditor.org/forum/viewtopic.php?f=2&t=676

3.4.3 Plugins

The WYMeditor includes a simple plugin which demonstrates how easy it is to write plugins for WYMeditor.

Note: Next versions will use a more advanced events handling architecture which will allow a better interaction between plugins and the editor.

Using a Plugin

To use a plugin you need to include its script file using a `<script>` tag and then initialize it (passing an instance of wym) in the `postInit` function, passed as an option when you call `$().wymeditor()`.

```
postInit: function(wym) {
    //activate the 'tidy' plugin, which cleans up the HTML
    //'wym' is the WYMeditor instance
    var wymtidy = wym.tidy();
    // You may also need to run some init functions on the plugin, however this depends on the plugi
    wymtidy.init(wym);
}
```

Writing Plugins

Writing a plugin for WYMeditor is quite easy, if you have a basic knowledge of jQuery and Object Oriented Programming in Javascript.

Once you decide the name for your plugin you should create a folder of that name in the plugins folder and then a file called jquery.wymeditor.__plugin_name__.js. You need to include this file in your HTML using a `<script>` tag.

For details on interacting with the editor, including the selection, see *API*.

Example Plugin

```
// wymeditor/plugins/hovertools/jquery.wymeditor.hovertools.js
// Extend WYM
Wymeditor.prototype.hovertools = function() {
    var wym = this;

    //bind events on buttons
    $j(this._box).find(this._options.toolSelector).hover(
        function() {
        wym.status($j(this).html());
        },
        function() {
        wym.status(' ');
        }
    );
};
```

This example extends WYMeditor with a new method `hovertools`, and uses jQuery to execute a function while the mouse hovers over WYMeditor tools.

`this._box` is the WYMeditor container, `this._options.toolSelector` is the jQuery selector, `wym.status()` displays a message in the status bar.

Adding a Button to the Tool Bar

```
// Find the editor box
var wym = this,
    $box = jQuery(this._box);

//construct the button's html
var html = '' +
    "<li class='wym_tools_fullscreen'>" +
        "<a name='Fullscreen' href='#' " +
            "style='background-image: url(" +
                wym._options.basePath +
                "plugins/fullscreen/icon_fullscreen.gif)'>" +
            "Fullscreen" +
        "</a>" +
    "</li>";
//add the button to the tools box
$box.find(wym._options.toolsSelector + wym._options.toolsListSelector)
    .append(html);
```

(work in progress)

Available Plugins

See *WYMeditor Plugins* for a listing and descriptions of the plugins included in the download and available third party plugins.

3.5 Writing WYMeditor Plugins

3.5.1 API

Note: In the code examples below, `wym` is a variable which refers to the WYMeditor instance, and which must be initialized.

Core

`html(sHtml)`

Get or set the editor's HTML value.

Example:

```
wym.html("<p>Hello, World.</p>");
```

xhtml()

Get the cleaned up editor's HTML value.

update()

Update the value of the element replaced by WYMeditor and the value of the HTML source textarea.

iframeInitialized

A boolean. After an editor's `iframe` initialization, this is set to `true`.

During the execution of *postInit*, for example, this can be expected to be `true`, if the editor initialized succesfully.

vanish()

Removes the WYMeditor instance from existence and replaces the 'data-wym-initialized' attribute of its textarea with 'data-wym-vanished'.

```
wym.vanish();
```

jQuery.getWymeditorByTextarea()

Get the WYMeditor instance of a textarea element. If an editor is not initialized for the textarea, returns false.

```
var myWym,
    myDocument;

myWym = jQuery.getWymeditorByTextarea(jQuery('textarea#myDocument'));

if (myWym) {
  myDocument = myWym.xhtml();
}
```

body()

Returns the document's `body` element.

Example; get the root-level nodes in the document:

```
var rootNodes = wym.body().childNodes;
```

$body()

Returns a jQuery object of the document's body element.

Example; find first paragraph in the document:

```
var $firstP = wym.$body().children('p').first();
```

Selection Setting and Getting

Note: For selection setting and selection getting, WYMeditor uses the Rangy library internally.

The Rangy library doesn't seem to provide a consistent interface for selection getting. Instead, the selection could be in many cases described differently in different browsers.

Additionally, erroneous selections are performed by some browsers under certain conditions.

In light of this, an effort has been made to provide reliable methods in WYMeditor for selection setting an getting.

Core contributors, as well as plugin authors, are encouraged to use these methods and to avoid using the Rangy API directly.

If you find these methods lack a feature that you require, then please file an issue describing your requirement so that we could look into answering it in a consistent and reliable way.

Pull requests regarding this or any other issue are warmly welcomed. For detailed pull request recommendations, please see our documentation on *Contributing*.

---

nodeAfterSel()

Get the node that is immediately after the selection, whether it is collapsed or not.

selectedContainer()

Get the selected container.

This is currently supposed to be used with a collapsed selection only.

mainContainer(sType)

Get or set the main container in which the selection is entirely in.

A main container is a root element in the document. For example, a paragraph or a 'div'. It is only allowed inside the root of the document and inside a blockquote element.

Example: switch the main container to Heading 1.

```
wym.mainContainer('H1');
```

Example: get the selected main container.

```
wym.status(wym.mainContainer().tagName);
```

canSetCaretBefore(node)

Check whether it is possible to set a collapsed selection immediately before provided node.

For an example see the test named 'selection: Set and get collapsed selection'.

Returns true if yes and false if no.

setCaretBefore(node)

This sets a collapsed selection before the specified node.

Note: Due to browser and/or Rangy bugs it has been decided that `node` could be either a text node or a `br` element and if it is a `br` element it must either have no `previousSibling` or its `previousSibling` must be a text node, a `br` element or any block element.

It checks whether this is possible, before doing so, using `canSetCaretBefore`.

canSetCaretIn(node)

Check whether it is possible to set a collapsed selection at the start inside a provided node. This is useful for the same reason as `canSetCaretBefore`.

setCaretIn(element)

Sets a collapsed selection at the start inside a provided element.

Note: Due to what seems like browser bugs, setting the caret inside an inline element results in a selection across the contents of that element.

For this reason it might not be useful for implementation of features.

It can, however, be useful in tests.

It checks whether this is possible, before doing so, using `canSetCaretIn`.

Content Manipulation

exec(cmd)

Execute a command.

Supported command identifiers

- Bold: set/unset `strong` on the selection
- Italic: set/unset `em` on the selection
- Superscript: set/unset `sup` on the selection
- Subscript: set/unset `sub` on the selection
- InsertOrderedList: create/remove an ordered list, based on the selection
- InsertUnorderedList: create/remove an unordered list, based on the selection
- Indent: *indent* the list element
- Outdent: *outdent* the list element
- Undo: undo an action
- Redo: redo an action
- CreateLink: open the link dialog and create/update a link on the selection

- Unlink: remove a link, based on the selection
- InsertImage: open the image dialog and insert/update an image
- InsertTable: open the table dialog and insert a table
- Paste: opens the paste dialog and paste raw paragraphs from an external application, e.g. Word
- ToggleHtml: show/hide the HTML value
- Preview: open the preview dialog

paste(data)

Parameters

- data: string

Description

Paste raw text, inserting new paragraphs.

insert(data)

Parameters

- data: XHTML string

Description

Insert XHTML string at the cursor position. If there's a selection, it is replaced by `data`.

Example:

```
wym.insert('<strong>Hello, World.</strong>');
```

wrap(left, right)

Parameters

- left: XHTML string
- right: XHTML string

Description

Wrap the inline selection with XHTML.

Example:

```
wym.wrap('<span class="city">', '</span>');
```

unwrap()

Unwrap the selection, by removing inline elements but keeping the selected text.

switchTo(node, sType, stripAttrs)

Switch the type of the given `node` to type `sType`.

If `stripAttrs` is true, the attributes of node will not be included in the new type. If `stripAttrs` is false (or undefined), the attributes of node will be preserved through the switch.

toggleClass(sClass, jqexpr)

Set or remove the class `sClass` on the selected container/parent matching the jQuery expression `jqexpr`.

Example: set the class `my-class` on the selected paragraph with the class `my-other-class`.

```
wym.toggleClass('.my-class', 'P.my-other-class')
```

User Interface

status(sMessage)

Update the HTML value of WYMeditor' status bar.

Example:

```
wym.status("This is the status bar.");
```

dialog(sType)

Open a dialog of type `sType`.

Supported values: Link, Image, Table, Paste_From_Word.

Example:

```
wym.dialog('Link');
```

toggleHtml()

Show/hide the HTML source.

focusOnDocument()

Set the browser's focus on the document.

This may be useful for returning focus to the document, for a smooth user experience, after some UI interaction.

For example, you may want to bind it as a handler for a dialog's window `beforeunload` event. For example:

```
jQuery(window).bind('beforeunload', function () {
    wym.focusOnDocument();
});
```

getButtons()

Returns a jQuery object, containing all the UI buttons.

Example:

```
var $buttons = wym.getButtons();
```

Internationalization

replaceStrings(sVal)

Localize the strings included in `sVal`.

encloseString(sVal)

Enclose a string in string delimiters.

Utilities

box

The WYMeditor container.

jQuery.wymeditors(i)

Returns the WYMeditor instance with index i (zero-based).

Example:

```
jQuery.wymeditors(0).toggleHtml();
```

jQuery.copyPropsFromObjectToObject(origin, target, props)

General helper function that copies specified list of properties from a specified origin object to a specified target object.

Example:

```
var foo = {A: 'a', B: 'b', C: 'c'},
    bar = {Y: 'y'};
jQuery.copyPropsFromObjectToObject(foo, bar, ['A', 'B']);
```

`bar` will then be `{A: 'a', B: 'b', Y: 'y'}`.

isInlineNode(node)

Returns true if the provided node is an inline type node. False, otherwise.

WYMeditor.isInternetExplorer*()

`WYMeditor.isInternetExplorerPre11()` and `WYMeditor.isInternetExplorer11OrNewer()`.

Internet Explorer's engine, Trident, had changed considerably in version 7, which is the version that IE11 has, and now behaves very similarly to Mozilla.

These two functions help detect whether the running browser is IE before 11 or IE11-or-newer, by returning a boolean.

3.5.2 Custom Events

WYMeditor allows plugin authors to hook in to various editor internals through a system of custom events. By registering a handler on a specific event, your plugin will be notified when something occurs so that you can modify the default behavior.

All events are triggered against the `textarea` element to which the wymeditor instance is bound. On an editor object `wym`, this element is available as `wym._element`.

Available Events

All events are present as a member of `WYMeditor.EVENTS`.

postBlockMaybeCreated

This event is fired when some user input occurred that might have created a new block-level element. Things that qualify include pressing the Enter key, pasting content, inserting tables, etc.

This event fires **after** WYMeditor's default handling occurs.

postIframeInitialization

This event is fired directly after the appropriate browser-specific editor subclass's `initIframe` method finishes. Hook in to this event if you need the iframe to exist, which usually occurs **after** the editor's `postInit` fires.

3.5.3 Plugin-Writing Mini Guide

1. Create an example in `examples/`

Make sure and provide a description on the example page and ideally, initialize the editor with content that makes playing with your plugin easy.

Note: Please avoid adding a file in `test` other than for unit tests. The `.html` files in `test` rot quickly (other than unit tests) because users don't see them.

Serving Examples

You can load your example via:

```
 vagrant$ grunt server
$ google-chrome http://localhost:9000/examples/
```

Serving Examples from `dist/`

To make sure your examples also works from the built distribution, we can tell `grunt` to build first:

```
 vagrant$ grunt server:dist
$ google-chrome http://localhost:9000/examples/
```

2. Create your plugin folder

- Your folder will live at `src/wymeditor/plugins/<pluginName>`.
- Your main javascript file should be `jquery.wymeditor.<pluginName>.js`.
- Any images or CSS should live in that folder.

3. Build your awesome thing

Now get cracking!

See the *Example Contribution Process* guide for general contribution advice.

If you get stuck, join us in the `#wymeditor` IRC channel on freenode.

4. Document your awesome plugin

Create a `docs/wymeditor_plugins/included_plugins/<your_plugin>.rst` file and tell your future users:

1. What your plugin does.
2. How to enable it (a link to the example is good).
3. How to customize it, if you have customization options.
4. Anything else they need to know about browser compatibility, etc.

3.5.4 Plugin Do's and Don'ts

Don't enable your plugin on file load

Your plugin should not activate itself just by loading your plugin's javascript. To use your plugin, the user must add some kind of initialization in the wymeditor `postInit`. For example:

```
jQuery('.wymeditor').wymeditor({
    postInit: function(wym) {
        wym.yourPlugin = new YourPlugin({optionFoo: 'bar'}, wym);
    }
});
```

Note: The `embed` plugin currently violates this, which is a bug.

Mark your dialog-opening buttons

If your plugin includes buttons that open dialog windows, mark the list item with the class `wym_opens_dialog`. This will prevent your dialog window from opening in the background.

Handle focus

When UI interactions steal focus from the document, consider using `editor.focusOnDocument`.

For example, right before a dialog window closes.

3.6 WYMeditor Development

3.6.1 Contributing

We <3 Contributions

We love your contributions. Anything, whitespace cleanup, spelling corrections, translations, jslint cleanup, etc is very welcome.

The general idea is that you fork WYMeditor, make your changes in a branch, add appropriate unit tests, hack until you're done, make sure the tests still pass, and then send a pull request. If you have questions on how to do any of this, please stop by #wymeditor on freenode IRC and ask. We're happy to help!

Example Contribution Process

1. Fork wymeditor to your personal GitHub account.
2. Clone it (`git clone <your personal repo url>`) and add the official repo as a remote so that you easily can keep up new changes (`git remote add upstream https://github.com/wymeditor/wymeditor.git`).
3. Create a new branch and check it out (`git checkout -b my-cool-new-feature`).
4. Make your changes, making sure to follow the *Coding Standard*. If possible, also include a unit test in `src/test/unit/test.js`.
5. Add the changed files to your staging area (`$ git add <modified files>`) and commit your changes with a meaningful message (`$ git commit -m "Describe your changes"`).
6. Repeat steps 4-5 until you're done.
7. Add yourself to the AUTHORS file!
8. Make sure unit tests pass in as many browsers as you can. If you don't have access to some of the supported browsers, be sure and note that in your pull request message so we can test them.
9. Make sure your code is up to date (see below) and if everything is fine push your changes to GitHub (`git push origin <your branch>`) and send a *Pull Request*.

Staying up to Date

If your fork or local branch falls behind the official upstream repository please do a `git fetch` and then `merge` or `rebase` to make sure your changes will apply cleanly – otherwise your pull request will not be accepted.

See the GitHub help section for further details.

Configuring Your Development Environment

WYMeditor uses the standard modern javascript development toolchain, centered on `Grunt` as our build tool and `node.js` via `NPM` for installing requirements. If you don't have your machine configured for node.js development, we've provided a `Vagrantfile` for easy setup using Vagrant.

If you want a custom, non-Vagrant environment, the basic requirements are:

- `git` and whatever tools you need to build from source. eg. `sudo apt-get install build-essential`
- Node.JS and NPM.
- *grunt-cli*

Then you just need to

```
$ npm install
```

Note: For the example setup of an Ubuntu Precise machine, check out our vagrant_provision.sh script, which we use for configuring the Vagrant machine.

Front-end dependencies with Bower

Our front-end dependencies are pulled in by Bower.

Grunt orchestrates this automatically so you don't have to think about it.

If you changed `bower.json` and want those changes to take affect, just restart the server or run `grunt bower`.

Environment Setup with Vagrant

1. Install Virtualbox First, you need a working installation of VirtualBox.

On Ubuntu, that's as easy as:

```
$ sudo apt-get install virtualbox
```

2. Install Vagrant Vagrant builds and provisions our Virtualbox. See their documentation for *Vagrant Installation Instructions <http://docs.vagrantup.com/v2/installation/>*.

3. Install Vagrant Plugins We use a couple of Vagrant plugins to make managing things easier.

```
$ vagrant plugin install vagrant-omnibus
$ vagrant plugin install vagrant-librarian-chef
```

4. Build Your Box

```
$ vagrant up
```

Vagrant Troubleshooting

Encrypted Home Directory: Problems with the NFS mount If you use an FUSE-based encrypted home directory, as is the default for Ubuntu, you might see an error like:

```
mount.nfs: access denied by server while mounting 10.10.10.1:/home/you/your-wym-repo
```

Unfortunately, NFS can't share encrypted directories, which is how Virtualbox and Vagrant keep your files synchronized. To work around this we recommend putting your git clone in a directory like `/opt`.

```
$ mkdir -p /opt/wym
$ cd /opt/wym
$ git clone https://github.com/wymeditor/wymeditor.git
$ cd wymeditor
$ vagrant up
```

Enabling Automatic Livereload for Development

The `grant`, `server`, and `server:dist` tasks both support "Live Reload" functionality. That means that if you have a proper browser extension installed, changing a file will automatically trigger a reload event in your browser.

If this sounds nifty, simply install the proper extension.

Note: If you're using the Vagrant development route, the performance hit from using the NFS share means that live reload won't be instantaneous.

3.6.2 Building WYMeditor

We use `grunt` to build WYMeditor. Assuming you've already followed the instructions for *Configuring Your Development Environment*, it's pretty straight forward.

```
$ vagrant ssh
vagrant:~$ cd wym
vagrant:~/wym$ grunt
```

That command runs the tests before the build. For just a plain build:

```
vagrant:~/wym$ grunt build
```

The resulting compressed distribution will appear in your `dist` directory.

3.6.3 Testing WYMeditor

WYMeditor includes a full unit test suite to help us ensure that the editor works great across a variety of browsers. The test suite should pass in any of our supported browsers and if it doesn't, please file a bug so we can fix it!

Note: All of the following assumes you've completed the process of *Configuring Your Development Environment*

Running the Tests in a Browser

1. Use `grunt` to start a server

```
$ grunt server
```

2. Load the test suite

By default, the server serves on port `9000`. Open up a browser to http://localhost:9000/test/unit/.

All green means you're good to go.

Note: In Internet Explorer 7, after the first test run, it appears that subsequent test run considerably slower. It is possible to work around this issue by restarting the browser.

Note: If testing in a virtual machine, for better performance, disabling virtual memory may help.

Running the Tests via `grunt`

In addition to the browser test suite, you can also run the unit tests from the command line in a headless Phantom.js browser using Grunt. This is how the `travis-ci` test suite runs.

```
$ grunt test
```

If the task runs with no errors or failures, you're good to go.

Testing Different jQuery Versions

The unit tests can be run with the different versions of jQuery hosted on Google's CDN.

In the Browser

To do this when running tests in a browser, append the URL parameter `?jquery=<version>` to the test suite URL.

For example, to use jquery version `1.7.0`:

```
http://localhost:9000/test/unit/jquery=1.7.0
```

Via grunt

To do this when running tests from the command line with Grunt, include the parameter `--jquery=<version>` when running the `test` task.

To use jquery 1.6.0 .. code-block:: shell-session

    $ grunt test –jquery=1.8.0

Travis CI

WYMeditor is set up on Travis CI so that the unit tests are run automatically using the `test` Grunt task with several different versions of jQuery whenever commits are submitted to the Git repositiory for the project. Any submitted pull requests should pass these tests.

3.6.4 Coding Standard

The goal of this document is to define a set of general rules regarding the formatting and structure of the WYMeditor code as well as defining some best practices. It should also serve as a good starting point for developers that want to contribute code to the WYMeditor project.

jshint

All Javascript source should pass the version of jshint that's defined via `grunt-contrib-jshint` in our `packages.json` with the options defined in our `.jshintrc`.

Ideally, you are running this in your text editor on every file save. At the very least, you can perform this check via Grunt:

```
vagrant@precise32:~/wym$ grunt jshint
```

Making Files Pass

`jshint` against master should always pass, all the time. That means that before sending a pull request, a feature branch should pass. Generally, there are a few techniques to make this happen.

Fix Your Code Most of the time, the errors are useful and changing the code to pass results in cleaner, easier to read, more-consistent code. If you're confused about an error, a quick google usually results in a Stack Overflow question with a good solution.

Define `global` and `exported` The combination of the unused and undef options is very useful for eliminating global leakage and dead code. It does have a cost in that we must tell jshint about which files rely on other files and which files are used as libraries by other files.

We do this using Inline configuration.

Do **not** use single-line exceptions to tell jshint to ignore global/exported problems.

`global` If you get a `W117` warning about a variable not being defined because it comes from another file, you need to define a global. For example, the `ok` method comes from QUnit so test files need the following at the top:

```
/* global ok */
```

`exported` For variables and functions defined in one file but used in another, you'll get a `W098` warning about the variable being defined but never used. You should fix this with an `exported` Inline configuration at the top of the file.

For example, if the file defines a `usefulUtiityFunction`, you would add the following at the top of the file:

```
/* exported usefulUtilityFunction */
```

Make single-section jshint Exceptions This should be used as little as possible. There's almost always a way to fix the code to match. When WYMeditor started enforcing a passing jshint, we added exceptions liberally just to get to a stable starting point. New code should almost never come with new exceptions.

This JSHint options documentation explains how to suppress a warning and then re-enable the warning. **Always** re-enable the warning after you disable it, even if there's currently no code after the block. Otherwise, new code added might accidentally not be subject to the check.

For example, if you're doing a `for in` loop that you know to be safe, you could do:

```
var y = Object.create(null);

/* jshint -W089 */
for (var prop in y) {
    // ...
}
/* jshint +W089 */
```

Make file-wide jshint Exceptions Any file-wide jshint exception is considered a bug, but in the interests of getting a passing jshint, some were used initially. Do **not** add these, despite the fact that some files currently use them. Pull requests that remove a file-wide exception and fix the resulting lint problems are greatly appreciated.

Current vs Ideal

Some of our current jshint options are a result of legacy code and not an indication of where we'd like to be. In the spirit of getting a 100% passing jshint run, we made some initial sacrifices. The goal is to get all of the code on the same standard.

Additionally, there are several files listed in `.jshintignore` because they don't yet conform to our standards. We would really all of our code to conform, which means removing the ignores for everything but the 3rd-party code.

Crockford Javascript Code Conventions

Please refer to the Crockford Javascript Code Conventions for our default code conventions. The *Formatting and Style* section describes some areas where we are more-strict.

Changes to Crockford Conventions

We also have some choices that contradict Crockford's conventions.

1. We use one `var` statement per scope, versus another `var` for each variable.
2. We still use `eval()` in a couple of places. It is evil, though, and it's considered an implementation bug.

Formatting and Style

Naming Conventions

Variables and Functions

- Give variables and function **meaningful names**. Naming is very important!
- Use mixedCase (lower CamelCase) for names spanning several words.
- *Constants* should be in all CAPITAL_LETTERS with underscores to separate words.
- Avoid the use of Hungarian Notation, instead make sure to *type* your variables by assigning default values and/or using comments.
- Use one `var` statement per scope, declaring all of your variables there on separate lines.

Example:

```
var elements = [],
    somethingElse = '',
    VERSION = 0.6;
function parseHtml () {};
```

Constructors Constructors should be named using PascalCase (upper CamelCase) for easier differentiation.

Example:

```
function MyObject () {}

MyObject.prototype = {
    function myMethod () {}
}
```

Namespacing

All code should be placed under the WYMeditor namespace to avoid creating any unnecessary global variables. If you're extending and/or modifying WYM, place you code where you see fit (most likely WYMeditor.plugins).

WYMeditor.core contains the Editor object and the SAPI as well as HTML, CSS and DOM parsers which make out the core parts of WYMeditor.

WYMeditor.ui contains the UI parts of WYM (i.e. the default Toolbar and Dialogue objects).

WYMeditor.util contains any utility methods or objects, see *Leave the Natives Alone*.

WYMeditor.plugins – place your plug-ins here.

Multi-Line Strings

Choosing among syntaxes for multi-line strings is rough, because they mostly all suck. We've settled on this as the least-bad:

```
var bigString = [""
    , wym._options.containersSelector
    , wym._options.classesSelector
].join('');
```

Advantages:

- Passes `jshint`
- Leading commas allows re-ordering without comma juggling
- A one-line addition is a one-line diff

- Can use other join characters like `, '' or ''\n` for flexibility
- Can indent lines in source to avoid >79 character lines
- Can indent lines in source to display HTML nesting for readability

HTML Strings Building HTML strings also kind of sucks. Eventually, we hope to using something like JSX. For now, just build a multi-line string with proper HTML indentation and using `'` as the quote character (so that it's easy to use proper `"` to quote HTML attributes).

```
var iframeHtml = [""
    , '<div class="wym_iframe wym_section">'
        , '<iframe src="' + WYMeditor.IFRAME_BASE_PATH + 'wymiframe.html" '
            , 'frameborder="0" '
            , 'scrolling="no" '
            , 'onload="this.contentWindow.parent.WYMeditor.INSTANCES['
            , WYMeditor.INDEX + '].initIframe(this)"'
            , '>'
        , '</iframe>'
    , '</div>'
].join(""),
```

Inheritance and "Classes"

There's a lot of different ways of doing inheritance in JavaScript. There have been attempts to emulate Classes and several patterns trying enhance, hide or modify the prototypal nature of JavaScript – some more successful than others. But in order to keep things familiar for as many JavaScript developers as possible we're sticking with the "Pseudo Classical" model (constructors and prototypes).

It's not that the different variations of the "Pseudo Classical" model out there are all bad, but there is no other "standard" way of doing inheritance.

Other Rules and Best Practices

Leave the Natives Alone WYMeditor is used by a lot of people in a lot of different environments thus modifying the prototypes for native objects (such as Array or String) can result in unwanted and complicated conflicts.

The solution is simple – simply leave them alone. Place any kind of general helper methods under WYMeditor.util.

Use Literals This is a basic one – but there's still a lot of developers that use the Array and Object constructors.

http://yuiblog.com/blog/2006/11/13/javascript-we-hardly-new-ya/

Use the `which` Property of jQuery Event Objects When watching for keyboard key input, use the `event.which` property to find the inputted key instead of `event.keyCode` or `event.charCode`. This should be done for consistency across the project because the `event.which` property normalizes `event.keyCode` and `event.charCode` in jQuery. Using `event.which` is also the recommended method by jQuery for watching keyboard key input.

Comments should read as "why?" sentences Wherever possible, comments should read like a sentence. Sentences evolved because they're good at conveying information. Fragments are often ambiguous to those who need the comment most. They should also mostly answer the question "why?" instead of what/how.

When tempted to write a comment that describes what a block of code does, instead, write a function with a good name. The exception is one-liners that are conceptually dense, although those are usually the sign of a need for a refactor or utility function.

"What" comment example

```
function MyPlugin(options, wym) {
    var defaults = {
        'optionFoo1': 'bar'
    };
    this._options = jQuery.extend(defaults, options);
    this._wym = wym;

    this.init();
}

MyPlugin.prototype.init = function() {
    var wym = this._wym,
        buttonFoo1,
        buttonFoo2,
        buttonsHtml,
        box = jQuery(wym._box);

    //construct the buttons' html
    buttonFoo1 = [""
        , "<li class='wym_tools_foo1'>"
        ,     "<a name='foo1' title='Foo 1' href='#'"
        ,         "{foo1}"
        ,     "</a>"
        , "</li>"
    ].join('');
    buttonFoo2 = [""
        , "<li class='wym_tools_foo2'>"
        ,     "<a name='foo2' title='Foo 2' href='#'"
        ,         "{foo2}"
        ,     "</a>"
        , "</li>"
    ].join('');

    buttonsHtml = buttonFoo1 + buttonFoo2;

    //add the button to the tools box
    box.find(wym._options.toolsSelector + wym._options.toolsListSelector)
        .append(buttonsHtml);

    //bind listeners
    box.find('li.wym_tools_foo1 a').click(function () {
        // Do foo1 things
    });
    box.find('li.wym_tools_foo2 a').click(function () {
        // Do foo2 things
    });
};
```

Improved

```
function MyPlugin(options, wym) {
    var defaults = {
```

```
        'optionFoo1': 'bar'
    };
    this._options = jQuery.extend(defaults, options);
    this._wym = wym;

    this.init();
}

MyPlugin.prototype.init = function() {
    var wym = this._wym,
        buttonsHtml,
        box = jQuery(wym._box);

    buttonsHtml = this._buildButtonsHtml();

    // Add the button to the tools box.
    // TODO: There should probably be a WYMeditor utility function for
    // doing this.
    box.find(wym._options.toolsSelector + wym._options.toolsListSelector)
        .append(buttonsHtml);

    this._bindEventListeners(box);
};

MyPlugin.prototype._buildButtonsHtml = function () {
    var buttonFoo1 = '',
        buttonFoo2 = '';

    buttonFoo1 = [""
        , "<li class='wym_tools_foo1'>"
        ,     "<a name='foo1' title='Foo 1' href='#'"
        ,         "{foo1}"
        ,     "</a>"
        , "</li>"
    ].join('');
    buttonFoo2 = [""
        , "<li class='wym_tools_foo2'>"
        ,     "<a name='foo2' title='Foo 2' href='#'"
        ,         "{foo2}"
        ,     "</a>"
        , "</li>"
    ].join('');

    return buttonFoo1 + buttonFoo2;
};

MyPlugin.prototype._bindEventListeners = function (box) {
    var myPlugin = this;

    box.find('li.wym_tools_foo1 a').click(function () {
        myPlugin._doFoo1Things();
    });
    box.find('li.wym_tools_foo2 a').click(function () {
        myPlugin._doFoo2Things();
    });
};

MyPlugin.prototype._doFoo1Things = function () {
```

```
    // Do foo1 things
};

MyPlugin.prototype._doFoo2Things = function () {
    // Do foo2 things
};
```

Further Reading Got any other links that you think can be of help for new WYM developers? Share them here!

- http://dev.opera.com/articles/view/javascript-best-practices/

3.6.5 WYMeditor Architecture

At a high level, WYMeditor is a jQuery plugin that replaces a textarea with a designMode iframe surrounded by an editor skin and editor controls/buttons. When you call `$('.myEditorClass').wymeditor()`, several things happen:

1. WYMeditor uses your configuration/settings to build out HTML templates for all of the various components.
2. Your textarea is hidden and WYMeditor uses the generated HTML to insert the editor buttons and interface in the same spot.
3. An iframe with `designMode=true` is created and inserted as the area you actually type in.
4. WYMeditor detects your browser and uses a form of prototype inheritance to instantiate the browser-specific version of the editor. The editor is the collection of event listeners, browser-specific DOM manipulations and interface hacks needed to provide the same behavior across IE, Firefox, Chrome, Safari and Opera.

Code Structure

WYMeditor's source is organized in to several modules corresponding to large pieces of functionality and all of the actual code lives in `src/wymeditor/`. The `build/` directory contains build tools and is the default location where bundled/packed/minified/archived versions of the project can be built (to `build/build/`) using the `Makefile`.

Inside `src/` you also have:

- `jquery/` – Includes the bundled version of jQuery and jQuery UI.
- `test/` – Includes both manual and unit tests. The manual tests in the .html files have various editor configurations and combinations of plugins and options.
  - `unit/` – Here lives the automated test suite which makes it possible to make a change without breaking 90 other things. The unit test suite is based on QUnit.

core.js

This is the core of the project including the definition of the WYMeditor namespace, project constants, some generic helpers and the definition of the jQuery plugin function.

The most important object in core is `WYMeditor.editor()` which is the object/function that `jquery.fn.wymeditor()` farms out to for all of the work of actually building the editor. `WYMeditor.editor()` does the basic job of building the `options` object (which for most cases is 99% the defaults), locating your JavaScript files and calling `init()` (which is defined in `editor/base.js` for the heavy lifting.)

parser.js

Here be dragons. `parser.js` is fundamentally responsible for taking in whatever HTML and CSS the browser or user throws at it, parsing it, and then spitting out 100% compliant, semantic xHTML and CSS. This also includes some work to correct invalid HTML and guess what the user/browser probably actually meant (unclosed li tags, improperly nested lists, etc.) It uses several components to do its job.

1. A lexer powered by parallel regular expression object
2. A base parser and an xHTML-specific parser
3. An XhtmlValidator to drop nonsense tags and attributes
4. A SAX-style listener to clean up the xHTML as it's parsed (this does most of the magic for guessing how to fix broken HTML)
5. A CSS-specific lexer and parser. This is mostly just used to take in CSS configuration options.

editor/

This folder is where most of the magic happens. It includes `base.js` for doing most of the heavy lifting and anything that can be done in a cross-browser manner. It takes the options you passed to the jQuery plugin (defined in `core.js`) and actually creates all of the UI and event listeners that drive the editor.

In the `init()` it also performs browser detection and loads the appropriate browser-specific editor extensions that handle all of the fun browser-specific quirks. The extensions are also located in this folder.

iframe/

This folder contains the HTML and CSS that's used to create the actual editor iframe (which is created by the `init()` method in the `WYMeditor.editor` object defined primarily in `editor/base.js`.) By switching the iframe source configuration, you choose which iframe to use (default is of course, `default`.)

Of special interest is the `wymiframe.css` file inside your chosen iframe (eg. *src/wymeditor/iframe/default/wymiframe.css*.) This file defines the signature blue background with white boxes around block-level elements and with the little "P, H2, CODE" images in the upper left.

skins/

The `skins/` folder is structured like the `iframe/` folder, with folders corresponding to different available skins and a default of `default`. An editor skin allows customization of the editor controls and UI, separate from the editable area that where a user actually types. The skin generally contains CSS, JS and icons and hooks in to the bare HTML that's produced by `WYMeditor.editor` using defined class names that correspond to different controls. The best way to understand and create/edit a skin is to look at the HTML constants defined in `WYMeditor` (inside `core.js`) and compare them to the CSS/JS defined in an existing skin (e.g. `src/wymeditor/skins/default`).

lang/

This is where translations to other languages live. Each file defines a specific `WYMeditor.STRINGS.<language code>` object with mappings from the English constant to the translated word.

plugins/

This is where all plugins bundled with WYMeditor live. There's currently quite a lot of variance between the ways plugins are created an organized, but they're at least organized with a top-level folder in the plugins directory with the plugin's name. In general, the name of the main plugin file has the format `jquery.wymeditor.<plugin_name>.js`.

A set of plugin system hooks is on the roadmap, but for now most plugins modify things in different ways and are relying on APIs that are not guaranteed. A future release will provide those guaranteed APIs. See *Plugin System Architecture*.

3.6.6 Expected Cross-browser behavior of the Cursor in Reaction to Keystrokes

For usability, it's very important that the common navigation keys (up/down/enter/backspace) should behave as a the user expects. This means that whenever possible, any cross-browser behavioral differences should be corrected.

General Guidelines

- The Enter and Backspace keys should be compliments with regards to navigation. Doing one and then the other should return to the same state.
- The Up and Down keys should be compliments.
- The Up and Down keys should *not* create new blocks. Only move the cursor between them.
- When navigating to "blue space" (areas without blocks) via the nav keys or the mouse, a paragraph block should be created on first content keystroke.
- Backspace at the start of a block joins the contents of the current block with the contents of the previous.
- Enter when in a block creates a new block after the current, with the content after the cursor in the new block.
- Tables, Images, Blockquotes and Pre areas are "special".

Guide

The | character represents the cursor.

Paragraph at Start

```
<p>|text</p>
```

Up

No Change

Down

No Change

Enter

```
<p></p>
<p>|text</p>
```

backspace

No Change

3.6.7 Planning For Future Enhancements

Roadmap for WYMeditor 2.0 (Draft)

Version 2.0 of WYMeditor will be a complete rewrite fixing a lot of the issues with the current stable version.

Version 2.0 can be grouped in to four major components: the Selection API or SAPI, the Editor core, the HTML, DOM and CSS parsers and the UI. These four components also make up the four steps we'll need to complete before we release version 2.0. Of course these steps overlap somewhat, and it's possible to work on several of these things in parallel.

General Goals

- Separating the different areas of WYMeditor from each other, making it more modular
- Implementing a solid event system
- Documenting the WYMeditor source more thoroughly
- A more consistent source code (through the *Coding Standard*)

The Roadmap

Step 1: The HTML, DOM and CSS Parsers In the current stable version there's a HTML parser and a CSS parser. In 2.0, we'll also be introducing a DOM parser, as we can not rely on the innerHtml property due to a couple of issues and lack of flexibility.

Goals

- Document the HTML and CSS parsers
- Get rid of the innerHtml dependency

New Features

- Inline controls
- Place holders for forms/flash/dynamic content

Issues to Fix

- innerHtml: Problems with links and urls (#69)
- innerHtml: Insertion of script elements doesn't work

Resources

- http://ejohn.org/blog/pure-javascript-html-parser/
- http://www.stevetucker.co.uk/page-innerxhtml.php

Step 2: the Selection API The SAPI in the current stable version is rather incomplete. To make things easier, the excellent IERange library will be used for Internet Explorer compatibility.

The SAPI could also be released as a standalone jQuery plug-in, as it could be useful to a lot of other people and projects.

Goals

- Define a general jQuery plugin for selections and ranges, and integrate the plugin with WYMeditor
- Creating a consistent API that can be used for manipulating content

New Features

- Save and restore selections, allowing support for modal dialogues and the like
- A more solid API

Resources

- http://code.google.com/p/ierange/
- https://developer.mozilla.org/en/DOM/Selection
- https://developer.mozilla.org/en/DOM:range

Step 3: The Editor Core In 2.0 we'll be moving away from designMode in favour of contentEditable. This will not only reduce the complexity a lot, it will also open up a lot of new interesting possibilities. Through easier interaction with external scripts things such as drag an drop, placeholders and the like are a lot easier to achieve.

Goals

- Move away from designMode in favour of contentEditable
- Solid event handling
- New features
- onChange/isDirty functionality (#138)

Issues to Fix

- Support for different block level elements (#152, #178)

Step 4: The UI The goal is to create a new clean looking, extendable and skinnable UI that's completely separated the WYMeditor core. This would allow the developer to completely replace the default UI with another one, opening up a lot of different integration possibilities.

Goals

- Create a new clean looking and consistent UI
- Separate the UI from the editor core
- Make it extendable and skinnable
- Make it more user friendly through on-screen help and well thought out layout, features and (visual) feedback

New features

- One toolbar for several editor instances (#97)
- Modal dialogues (#63)

Issues to Fix

- Link editing (#69)

- Block level elements inside list items (#135)

Proposed Build Stack

100% Javascript

For WYMeditor 1.0, we're moving to an all-Javascript build stack from the Node.js community. The rationale behind this is simple – use tools that make sense to JS developers. Any person with `node.js` and `npm` installed should be able to run a few `npm` commands and then do development or run a build.

The goal will be to combine asset-management best practices like concatenation and minification for CDNs with automatic file generation for easy development. Editors like TinyMCE are often criticized for poor HTTP performance and there are posts strewn across the interwebs from people attempting to hack together solutions to this problem. We should solve it for everyone by default and even make it easy for folks to make custom builds with just the plugins that they require.

Components

General build tool There are always miscellaneous tasks and helpers that developers need, from building documentation to bumping version numbers and tagging a new release. It's always nice for new users if there is a consistent way of performing these actions.

Jake

A port of Rake. We should be able to lint, run tests, build documentation, create a full build or create customized builds all from jake.

Asset Management We ultimately want the ability to produce a trio of one minified javascript file, one minified CSS file and one image file with all requirements. This will include the editor, its skin and any plugins the developer wants to enable (including 3rd-party plugins) along with all of their assets.

Either node-ams or local-cdn. Both provide a file-watching development server via node.js to allow easy development. Both let you statically define and then generate files on demand for releases.

Documentation What we really need is Sphinx ported to Javascript, but until something like that emerges in the node community, the standard solution is inline documentation plus stand-alone statically-generated HTML documentation.

Inline Documentation Docco

Allows writing inline docs in pure Markdown. Not as restricting as most other solutions since it isn't modeled after programming paradigms foreign to JavaScript. Also good for consistency across different platforms (GitHub wiki, issues and comments, the new forum, etc.)

Full Documentation Inline documentation doesn't cover tutorials, API documentation and reference docs. The Django documentation is a good example of what we're going for. For now, it seems most projects are rolling their own combination of statically-generated HTML sites powered by MarkDown. To start, rip off the documentation from a project like express and get started.

Testing

- QUnit for the test framework.
- phantomJS and qunit-tap to run the tests from the command line for quick webkit tests.
- Jenkins for CI to automatically package builds, run static analysis, run phantomJS tests and then use Sauce On Demand to spin up supported browsers for cross-browser testing.
- TestSwarm for coordinating cross-browser tests in all supported browsers.
- node-jshint for static analysis and linting.

Plugin System Architecture

A proper plugin system should provide one obvious way to perform all of the common things that plugins do. It should still be possible, however, to use plain Javascript to do uncommon things.

Goals

- Plugins should read such that someone who understands Javascript should be able to follow along without knowing all of the details of the plugin API. This means that a little more "boilerplate" is better than magic and explicit is better than implicit. An example of explicit:

```
function TableEditor(name, wymeditor, options) {
    this.name = name;
    this.wymeditor = wymeditor;
    this.options = options;
}

TableEditor.prototype.init = function() {
    this.wymeditor.buttons.addButton({'name': 'AddRow', 'title': 'Add Row', 'cssClass': 'wym
}

TableEditor.prototype.bindEvents = function() {
    var tableEditor = this;
    var wym = this.wymeditor;

    wym.buttons.getButton('AddRow').find().click(function(evt) {
        return tableEditor.handleAddRowClick(evt);
    });
}
```

  An implicit way to do things would be to set up some magically-named class attribute that is automatically used by WYMeditor at some point during initialization to create these things.

- Very little method replacing and attribute mangling of the wymeditor object itself should be necessary.
- Most/all of a plugins UI actions (creating dialogs, adding buttons etc) should be done through API calls, allowing editor-wide standardization and theming.
- Plugins should have strong hooks into actions so that they're able to clean up the DOM to fix cross-browser problems.
- All currently-core WYMeditor actions should be migrated to being stand-alone plugins.
- The plugin API shouldn't be responsible for automatically locating Javascript files. The only way to efficiently handle that is on the server side and in the HTML file itself.

Enabling a Plugin

All well-behaving plugins should be explicitly enabled through the `plugins` configuration option on editor initialization. Plugins that alter WYMeditor behavior without being explicitly enabled (like the 0.5.x embed plugin) are considered misbehaving.

The `WYMeditor` `plugins` configuration option is where plugin configuration occurs. This object is an array of objects with the plugin's `name`, `func`, and `options`. For example:

```
jQuery('.wymeditor').wymeditor({
    plugins: [
        {
            name: 'table',
            func: TableEditor,
            options: {enableCellTabbing: false}
        }
    ]
});
```

API

WYMeditor.plugins.addPlugin(<pluginName>, <pluginFunction>, <configurationObject>);

WYMeditor.plugins.getPlugin(<pluginName>);

WYMeditor.buttons.addButton(<options>);

WYMeditor.buttons.getButton(<buttonName>);

WYMeditor.buttons.removeButton(<buttonName>);

WYMeditor.dialogs.createDialog(<dialogName>, <options>, <callback>);

WYMeditor.dialogs.destroyDialog(<dialogName>, <options>, <callback>);

WYMeditor.addXhtmlCleanup(<cleanupName>, <cleanupFunction>);

WYMeditor.removeXhtmlCleanup(<cleanupName>);

WYMeditor.addDomCleanup(<cleanupName>, <cleanupFunction>);

WYMeditor.removeDomCleanup(<cleanupName>);

Note: The following methods already exist:

WYMeditor.editor.findUp(); WYMeditor.editor.container(); WYMeditor.editor.update(); WYMeditor.editor.html(); WYMeditor.editor.xhtml(); WYMeditor.editor.switchTo(); WYMeditor.editor.wrap(); WYMeditor.editor.unwrap(); WYMeditor.editor.setFocusToNode(); WYMeditor.editor.exec(<commandName>);

Selection API

The Selection API will be used to ease the interaction with selected text and the current cursor position. If you want to handle an event yourself (and stop the default one), it's important to know where the cursor currently is. But often you only need to know the position relative to a container element and the container itself. There some problems that arise, such as nested tags. In this document, `|` will be used to indicate the current cursor position:

```
<p>Some <em>nice|</em> text</p>
```

Of course you don't want returned as container element, but the <p>.

Here's my proposal of some basic functionality. That was all I needed for my prototype. The following code is Javascript pseudocode:

```
selection = {
    // properties

    // the original DOM Selection object
    //(http://developer.mozilla.org/en/docs/DOM:Selection)
    original:

    // the node the selection starts. "start" is the most left position of the
    // selection, not where the user started to select (the user couls also
    // select from right to left)
    startNode:

    // the node the selection ends
    endNode:

    // the offset the cursor/beginning of the selection has to it's parent nodenode
    startOffset:

    // the offset the cursor/end of the selection has to it's parent nodenode
    endOffset:

    // whether the selection is collapsed ot not
    isCollapsed:


    // That one don't need to be implemented at the moment. It's diffcult as
    // one need to define what length is in this context. Is it the number of
    // selected characters? Or the number of nodes? I'd say it's the number of
    // selected characters of the normalized (no whitespaces at end or beginning,
    // multiple inner whitepsaces collapsed to a single one) selected text.
    length:

    // methods

    // Returns true if selection starts at the beginning of a (nested) tag
    // @param String jqexpr The container the check should be performed in
    // @example element = <p><b>|here</b></p>, element.isAtStart("p") would
    // return true
    isAtStart: function(jqexpr)

    // Returns true if selection ends at the end of a (nested) tag
    // @param String jqexpr The container the check should be performed in
    // @example element = <p><b>here|</b></p>, element.isAtEnd("p") would
    // return true
    isAtEnd: function(jqexpr)

    // set the cursor to the first character (it can be nested),
    // @param String jqexpr The cursor will be set to the first character
    //     of this element
    // @example elements = <p><b>test</b></p>, element.cursorToStart("p")
    //     will set the cursor in front of test: <p><b>|test</b></p>
    cursorToStart: function(jqexpr)

    // set the cursor to the last character (it can be nested),
```

```
    // @param String jqexpr The cursor will be set to the last character
    //     of this element
    // @example elements = <p><b>test</b></p>, element.cursorToEnd("p")
    //     will set the cursor behind test: <p><b>test|</b></p>
    cursorToEnd: function(jqexpr)

    // removes the current selection from document tree if the cursor isn't
    // collapsed.
    // NOTE Use the native function from DOM Selection API:
    // http://developer.mozilla.org/en/docs/DOM:Selection:deleteFromDocument
    // NOTE First I had also "deleteFromDocument()" in the Selection API, but
    // it isn't needed as a "sel.deleteIfExpanded()" would do exactly the same
    // as "sel.deleteFromDocument()" with the only difference that this one
    // has a return value
    // @returns true:  if selection was expanded and therefore deleted
    //          false: if selection was already collapsed
    // @example // do what delete key normally does
    //          if (!sel.isCollapsed)
    //          {
    //              sel.original.deleteFromDocument();
    //              return true;
    //          }
    //          will be now:
    //          if (sel.deleteIfExpanded())
    //              return true;
    deleteIfExpanded: function()
};
```

Todo

- save selection (and current cursor position)

Plan for Insertion Functions

Current plan for a clean API around inserting inline and block elements and text.

Deprecate `insert` and alias it to `insertInline`.

insertInline

Basically the same as the current `insert` function with a few exceptions.

- Inserts the contents where the current selector is inside the current block element.
- If the selector is in the body, it creates a paragraph to wrap the contents inside, if needed.
- If `insertInline` is called with HTML representing a non-nestable block level element, an exception is raised instead of attempting to guess the proper behavior.

insertBlockAfter

(This is @samuelcole's insert_next, effectively)

Used to insert block-level elements after the currently-selected block-level element.

- If given a string representing a text node, wraps the text in the appropriate container (`p` or `li` tag, depending).

- No matter the current selection, does *not* split the current block.

`insertBlock`

Used to insert a block-level element and split the current block-level element on the selection boundary.

- If given a string representing a text node, wraps the text in the appropriate container (`p` or `li` tag, depending)

Examples Inserting `<p>inserted</p>` with `|` representing current cursor.

Middle of Node

```
<p>before| after</p>
<p>before</p><p>inserted</p><p>| after</p>
```

Start of Node

```
<p>|before after</p>
<p>|before after</p><p>inserted</p>
```

End of Node

```
<p>before after|</p>
<p>before after|</p><p>inserted</p>
```

Dev Process Improvements

This page is where we brain-dump ideas for improving the development/testing/packaging/documentation/etc process because making Issues for speculation feels dirty.

Build Process

1. Minify/Concat all the things HTTP requests are silly.

Put all of the plugins in the bundled version, changing any of them so that just including the code doesn't activate them.

2. Support Custom Builds Use an `--exclude` option to grunt similar to this. Then folks can make their own slimmed-down builds.

Documentation

Better Structure Current planned ideal:

- index
- getting_started/
  - index
  - setup (mention existing_integrations)
  - philosophy (or should this be on the website?)
  - getting_help (link to contributing)

- customizing_wymeditor/
  - index (general overview of architecture and explanation of customization methods)
  - configuration_options
  - using_plugins (content from using_wymeditor/using_plugins and plugins/index)
  - using_skins
  - using_content_layouts (better name than iframe)
  - howto/ (instead of customizing_wymeditor/examples)
    - toolbar_items ?
    - etc
- plugins/
  - index
  - core_plugins/
    - bidi
    - list
    - table
    - etc
  - third_party_plugins
- skins/
  - index
  - core_skins/
    - silver
    - legacy
    - minimal
  - third_party_skins
- content_layouts
  - index
  - core_content_layouts/
    - pretty
    - legacy
  - third_party_content_layouts
- resources/
  - index
  - wymeditor_end_user_guide
- writing_plugins/
  - index
- writing_skins/

    - index
- writing_content_layouts/
    - index
- existing_integrations
- upgrade_guide
- changelog
- wymeditor_development/ (as it already exists)

Automate Taking Demo Screenshots for the Docs Screenshots are worth a thousand words. Use grunt-autoshoot to take screenshots of the examples and update the docs to embed them.

Code Cleanup

JSHint the remaining first-party files Only things we don't control should be in `.jshintignore`.

Don't use `eval()` It's not necessary.

Vendor Rangy We should include Rangy via bower.

Testing

Eliminate the Selenium tests There are some bugs that can only be tested with native events, so we wrote Selenium tests for them. This kind of sucks, though, since it's a very separate concern.

Instead, use one of the Java Applet-based real user event simulators.

Candidates

- Dojo's DOHRobot.
- Ephox JSRobot.

Of the two, `DOHRobot` seems to have more of an active community. `JSRobot` is used to test TinyMCE, though.

Run the Unit Tests in Every Browser on Every Build Testling-CI seems like the way to go for running our unit tests across our supported browsers. It won't work for our Selenium tests, but it will at least make it easy to catch regressions and the like when lazy developers *cough*me*cough* don't test in all of the IE's.

Load all examples/tests on travis Have travis load all of the examples and tests using phantomjs and verify that WYMeditor is at least finishing initialization.

3.6.8 Troubleshooting

Error Running `$ grunt server`

On some linux systems, (eg. Ubuntu 10.04), you see something like:

```
$ grunt server
Running "express:all" (express) task

Running "open:all" (open) task
Web server started on port:9000, hostname: 0.0.0.0 [pid: 29903]

Running "watch" task
Waiting...Fatal error: watch ENOSPC
```

That `ENOSPC` thing is related to your `inotify` watchers. Basically, you're trying to watch more files than are allowed.

Just Fix It

To just up the number of `inotify` watchers allowed, run:

```
$ echo fs.inotify.max_user_watches=524288 | sudo tee -a /etc/sysctl.conf
$ sudo sysctl -p
```

More Details

For a more detailed explanation, see the guard/listen wiki.

3.6.9 WYMeditor Website

The website at http://wymeditor.github.io/wymeditor/ is served via Github pages and uses Jekyll. Instead of dealing with different content between a `gh-pages` and master branch, `master` also contains the jekyll content. This also allows us to server the demos/examples via github pages, while also adding custom content.

Website vs README vs Docs

Currently, there's a lot of overlap between the docs, the website and the README.

The focus of these should be:

- Website = Marketing/Examples/Getting-started
- README = Funnels to Website but contains project-wide info
- docs = detailed user and development documentation

Website Files

Currently, we have a single-page website controlled by an `index.html` file, which uses a layout defined in `_layouts/home.html`. This file uses several custom variables defined inside `_config.yml`.

Website Theme

Our theme is a ported version of a github pages layout. Its media and styles are located in `website-media/`.

Configuring Jekyll

```
$ rvm use 1.9.3
$ bundle install
```

Previewing the Website Locally

```
$ jekyll build
$ jekyll serve
$ google-chrome "http://localhost:4000"
```

Updating the Hosted Version

```
$ git checkout gh-pages
$ git merge origin/master
$ git push origin gh-pages
```

3.7 Resources

3.7.1 Using WYMeditor

Introduction

These guidelines will teach you why to use certain techniques when creating content and how to work with them in WYMeditor. Following these will keep the structure and meaning of your content correct - making it more flexible to technologies like SEO1 (Search Engine Optimization), screen readers2, mobile phone technologies, printers etc. Following these guidelines will also make your content accessible to a wider range of people, with or without disabilities

Guidelines

1. Use elements as intended

HTML is a so called markup language. It's purpose is to give the content meaning and structure, not visual effects. Therefore, it's important to use elements as intended to not break meaning, structure and compatibility to other technologies. If you want to style your content you've created with WYMeditor, you shall learn to use CSS (Cascading Style Sheets).

1.1. Headings

Using a logical and proper heading structure is among one of the most important SEO techniques. Always use a correct hierarchy. Start with Heading 1, followed by Heading 2, Heading 3... etc. A proper heading structure will also help users to find what they are looking for, faster.

To create a heading you simply place your marker where you want the heading to go and then select "Containers > Heading 1" for example.

1.2. Tables

Creating lean and accessible tables is a craft but WYMeditor makes it easy. Only use the Table element to arrange different kinds of data - time tables, statistics, member listings etc. Use the Table Header container to mark up headers properly. Doing this wrong will confuse users with screen readers when accessing tables, as headers aren't part of the data.

To create a Table Header you simply place your marker in a table cell and select "Containers > Table Header".

It's also good practice to give users the ability to do a quick overlook of the table's content by filling in a short description in the Caption field. You can do that in the dialog appearing when creating the table.

1.3. Lists

It's important to mark up lists of items the right way, using Ordered list or Unordered list. Doing this wrong makes it harder for users to overlook the content and will also confuse users with screen readers. Creating a list using Shift+Enter to separate items is therefore highly depreciated as both structure and meaning will break in the document.

To create a list you simply use one of the two list buttons in WYMeditor's toolbar. Hit enter twice to "jump out of" a list.

1.4. Indent and outdent

The Indent and Outdent icons are only intended to create nested lists of list items (as described in 1.3). Place the marker on a list item and hit the Indent icon to make it go sub level. To undo this move or to make it go back one level, simply hit the Outdent icon.

1.5. Blockquotes

Mark up your quoted text by using the Blockquote container. Doing this the right way will help search engines and users with screen readers to understand your text and give it relevant meaning.

2. Provide alternatives

2.1. Alternative text

An alternative text string shall be accurate and equivalent in presenting the same content and function as the image. This also helps to identify the relevance and meaning of your image which improves SEO and accessibility for user with screen readers.

You can enter the Alternative text in the image dialog, appearing when adding or editing an image. The only exception to leave the field empty is when your image is for decoration purpose only.

2.2. Title attribute

To provide additional information of a link or and image you can provide a Title text. This helps several other technologies to identify the relevance and meaning of your elements which improves SEO and accessibility for user with screen readers.

You can enter the Title text in the image/link dialog, appearing when adding or editing an element.

3. Write understandable

Writing understandable is among one of the most important checkpoints in creating accessible content. Unfortunately computers can't do anything about this, yet. But with WYMeditor you have got the tools to make it down the right way.

3.1. Descriptive headings

A heading element shall briefly describe the topic of the section it introduces making it easier to overlook both for all users. This will also improve the SEO aspects of your content.

3.2. Descriptive link texts

A link text shall be completely descriptive of its location, which makes it a lot easier to overlook and skim-read the content for all users. This will also improve the SEO aspects and the "link indexing" procedure for users with screen readers. "Click here" links are therefor highly depreciated.

Word References

SEO (Search Engine Optimization) "Search engine optimization" is a conception for everything that makes your content rank higher when using search engines like Google.

Screen reader A screen reader is a text-only web browser that dictates webpages' content to visually disabled peoples. In this text "screen readers" refers to all sorts of assistive and adoptive web technologies.