How to customize Ephox Textbox.io

To use Textbox.io on Internet Explorer 11 on a site in the *local* and *intranet* Internet Explorer security zones, your page **must** place the browser into *edge* mode using the `X-UA-Compatible` header (see the code fragment below). Otherwise, in these zones Internet Explorer defaults to using *compatibility mode* (i.e. it reverts to using the Internet Explorer 9 rendering and JavaScript engines). If your application/site is in Internet Explorer's *Internet* zone then it is not necessary to adjust Internet Explorer's settings.

```
<meta http-equiv="X-UA-Compatible" content="IE=edge" />
```

Textbox.io DOES NOT support IE compatibility mode

To check systems requirements check the next link: http://docs.ephox.com/display/tbio/System+Requirements

All the information was copy & pasted from: https://developer.ibm.com/recipes/tutorials/how-to-customize-ephox-textbox-io-for-ibm-web-content-manager-8-5/#r_overview

Base22 doesn't keep authoring from the content of this article, was reproduced in case the original source is lost, since the content is very useful. All content below to its original author: Luca Fiori

Overview

The requisites are:

1. Basic knowledge about the WPS administration.
2. Basic knowledge about using the WAS console.
3. Basic knowlege about the WCM authoring interface (WCM authoring portlet).
4. Basic knowledge about the javascript language.
5. Knowledge about JAVA language and WCM API (optional).

The steps described in this guide was tested in a WPS/WCM version 8.5 CF13 installation, in Windows 7 Professional environment Service Pack 1. This guide does not pretend to cover all the editor's customization aspects, but only to deepen some and to provide general guidelines on the customization. Please let me know about any error, imprecision, inadequacy or inaccuracy of this document.

Step-by-step

Introduction

Ephox Textbox.io JavaScript HTML Rich Text Editor is a good replacement of the *Ephox EditLive* rich text editor which is Java based. It is based on HTML5 JavaScript and CSS3 standards and It supports a wide variety of Internet browser. For a full list of them and system requirements see the Ephox documentation [2].

The Textbox.io editor can be embedded in any HTML page and for a running example see the Ephox page [1]:

Since the *Web Content Manager* version 8.5, Cumulative Fix 11 (CF11), Textbox.io is installed by default as the “Advanced Editor” under the Web Content authoring portlet “Editor Options” configuration (see Ephox page [3]).

However, it can be installed manually in IBM Web Content Manager earlier versions, starting from IBM WCM 8.5.0.0 Cumulative Fix 06 (CF6).

The configuration file

To customize the Textbox.io editor for WCM is necessary to modify a copy of the **tbio_config.jsp** configuration file, located at the directory:

[PortalServer root]/wcm/prereq.wcm/wcm/config/templates/shared/app/config/textboxio

The possible customization will be performed by adding, changing and removing Java and JavaScript code. The customization of the

configuration file allows to change the configuration object, by inserting, removing or changing properties and their values.

The configuration object

The configuration object is a JavaScript object that gather all the editor's properties. Its original Ephox default is:

```
var cnfDefault =
{
    autosubmit: true,
    css:
    {
        stylesheets: [''],
        styles:
        [
            {rule: 'p', text: 'block.p'},
            {rule: 'h1', text: 'block.h1'},
            {rule: 'h2', text: 'block.h2'},
            {rule: 'h3', text: 'block.h3'},
            {rule: 'h4', text: 'block.h4'},
            {rule: 'div', text: 'block.div'},
            {rule: 'pre', text: 'block.pre'}
        ]
    },
    codeview:
    {
        enabled: true,
        showButton: true
    },
    images:
    {
        allowLocal: true
    },
    languages: ['en', 'es', 'fr', 'de', 'pt', 'zh']
    ui:
    {
        toolbar:
        {
            items: ['undo', 'insert', 'style', 'emphasis', 'align',
'listindent', 'format', 'tools'],
            contextual: ['image-tools', 'table-tools']
        }
    }
};
```

The configuration consists of several property/value pairs, separated by a colon (eg. **autosubmit: true**), which can also be nested. Some values are scalar, other are mono-dimensional arrays. All the configuration object properties are optional. We actually need to put in the configuration object only the properties we need to change.

Here the list of the main level properties of the configuration object:

Property	Type	Brief description

autosubmit	Boolean	Specifies whether Textbox.io should handle form submission
basePath	String	Specifies the path to the Textbox.io resources folder
css	Object	Editor rendering CSS and styling application
codeview	Object	Code view feature
images	Object	Editor image handling & upload
links	Object	Editor link validation
paste	Object	Editor paste behavior
spelling	Object	Editor spell checking service
ui	Object	Editor UI including toolbars, menus, etc.

For further details, see Ephox page [11].

Follows the description of some of these properties and how to customize them.

1. ## The css propery

   In the default configuration, Textbox.io have the following styles pulldown menu:

   The corresponding code, in the **tbio_config.jsp** file, is the following one:

```
var cnfDefault =
{
    …
    css:
    {
        stylesheets: [''],
        styles:
        [
            {rule: 'p', text: 'block.p'},
            {rule: 'h1', text: 'block.h1'},
            {rule: 'h2', text: 'block.h2'},
            {rule: 'h3', text: 'block.h3'},
            {rule: 'h4', text: 'block.h4'},
            {rule: 'div', text: 'block.div'},
            {rule: 'pre', text: 'block.pre'}
        ]
    },
    …
};
```

For further details, see Ephox page [13].

2. ### The codeview property

The editor instance contains, as default, in the bottom right corner, the button

that permits to switch between the WYSIWYG and HTML source code. In the configuration object, as default, we have the codeview property:

```
var cnfDefault =
{
    …
    codeview:
    {
        enabled: true,
        showButton: true
    },
    …
};
```

For further details, see Ephox page [12].

3. ### The ui property

The user interface property (**ui**) defines options related to the editor user interface, including the editor's toolbar.

Here the list of the sub-properties of the **ui** property:

Property	Type	Brief description
aria-label	Boolean	Text to use for the editor's ARIA label instead of the default
languages	String	Specifies available language codes that can be applied to content for internationalization
locale	Object	Specifies editor UI language
fonts	Object	Specifies the list of fonts available to the editor
shortcuts	Object	Specifies whether to enable content keyboard shortcuts
toolbar	Object	An object representing the desired toolbar functionality for the editor

For further details, see Ephox page [14].

4. ## The languages sub-property

The languages array lets a developer specify one or more language codes that can be applied to HTML content for internationalization. Adding the languages configuration won't add automatically the language button to the toolbar which needs to be specified as explained in the next paragraph.

This array directly configures the languages available for application in the languages menu. Selecting a language from the languages menu sets the **HTML lang attribute** for text selected in the editor (see w3.org page [18]).

Setting the languages configuration array overrides the languages array defaults.

Note that only some languages have their names translated into all Textbox.io UI languages (see Translated Language Codes below). A developer may choose to apply languages codes from this list, or specify any 2 or 4 letter language code. When specifying a language code that is outside of the translated language codes list (like 'x-klingon'), that language code will appear in the language menu.

The default languages are:

Language	Language code String
English	en
Spanish	es
French	fr
German	de
Portuguese	pt
Chinese	zh

For further details, see Ephox page [17].

5. ## The toolbar sub-property

The user interface toolbar property defines options related to editor toolbar commands, groups and child menus. The contextual toolbar items currently appear at the end of the toolbar (they are relating to images and tables only).

Its main sub-properties are:

Sub-property	Type	Default	Description
items	Array	['undo', 'insert', 'style', 'emphasis', 'align', 'listindent', 'format', 'tools']	An array representing the structure of the Textbox.io toolbar and menu system. Each item represents a toolbar group
contextual	Array	['table-tools','image-tools']	An array listing the items that can appear depending on the selection context

hence, the fragment of the default configuration object, related to the toolbar, will be:

```
var cnfDefault =
{
    …
    ui:
    {
        toolbar:
        {
            items: ['undo', 'insert', 'style', 'emphasis', 'align',
'listindent', 'format', 'tools'],
            contextual: ['image-tools', ‘table-tools’]
        }
    }
};
```

Toolbars are made up of item objects. Items represent either editor commands, toolbar groups or menus. Items infer their user interface from their position in the items array. An item placed inside a menu will be rendered as a menu item, while an item placed inside a toolbar group will be rendered as a button. Similarly, a menu item placed within a menu will result in a sub-menu. Items have 3 distinct types, representing user interface constructs in a Textbox.io editor. **Command** items represent discrete editor functionality. **Menu** items represent a nested group of commands invoked from a root user interface element. **Group** items represent logical groupings of commands either inside the toolbar or within menus. Follows a table explaining the properties for each of these item types:

Item type	Properties			Description
Command	**id**	String	The ID string for the command	Command type items represent discrete editor commands, such as: apply bold, insert link, etc. Note that built-in command items are referenced by their string ID rather than specified as objects
	text	String	The friendly name of the command, shown in tooltips (optional)	
	icon	String	The path to the icon used to represent the command	
	action	Function	A function to be executed when the command is invoked via user action	
Menu	**id**	String	The ID string for the menu	Menu type items represent groupings of commands in a menu. When rendered, menus appear on the host toolbar as an icon, or on a host menu via an icon followed by the menu’s name
	label	String	The friendly name of the menu, visible to assistive devices (optional)	
	icon	String	The path to the icon used to represent the menu	
	items	Array	An array of command or menu items	
Group	**label**	String	The friendly name of the menu, visible to assistive devices (optional)	Group type items represent logical groupings of commands on a toolbar or within a menu. When an editor is rendered groups are designated by visual separators
	tems	Array	An array of commands or menu items	

Built-in editor commands are represented by a predefined string id in toolbar configurations. For a list of built-in editor command ids see Ephox page [16]. The toolbar items array is the primary way to configure toolbars, menus and buttons for a Textbox.io editor instance. The items array can be set to one or more toolbar group objects. These group objects can themselves be populated with further items to create toolbar buttons and menus in a rendered editor. Follows a schema of toolbar items array hierarchical structure:

- Toolbar items
  - Toolbar group(s)
    - Command item(s)
    - Menu item(s)
      - Command item(s)
      - Menu item(s)

Group items are objects that consist of a string name and an items array. The items array may contain command item IDs or command item objects, for example:

```
var aItems =
[
    // Simple Toolbar group object with 2 function IDs
    {
        label: 'Toolbar Group 1',
        items: ['undo', 'redo']
    }
];
```

Command items are objects that consist of a string id, a string name, a string path to an icon resource, and an action function. When command items are placed in a toolbar group, their functionality will be represented on an editor toolbar with a button. The button will contain the specified icon image. When command items are placed within a menu item object, their functionality will be represented with a menu item. The menu item will contain the specified icon image and the name string. When a user clicks on the button (toolbar) or menu item (menu), the function specified in action will execute. Example:

```
// Command item object
var itmCustom =
{
    id: 'custom1',
    text: 'Custom Button 1',
    icon: '/path/to/icon1.png',
    action: function () {alert('Custom button 1 Clicked');}
};

var aItems =
[
    {
        // Toolbar group object with custom command
        label: 'Toolbar Group 2',
        items: ['undo', 'redo', itmCustom]
    }
];
```

Toolbar **menu items** are objects that consist of a string id, a string name, a string path to an icon resource, and an items array. The items array may contain command item IDs or command item objects, for example:

```
// Menu item object with 2 function IDs
var itmCustom =
{
    id: 'custom1',
    label: 'Custom Menu',
    icon: '/path/to/icon1.png',
    items: ['bold', 'italics']};

// Items array with one group object containing 2 function IDs and a
// custom menu item object
var aItems =
[
    {
        // Toolbar group object with custom menu item
        label: 'Toolbar Group 2',
        items: ['undo', 'redo', itmCustom]
    }
];
```

Nested menus are only supported to the second level. For further details, see Ephox page [15].

6. ### Example 1: css property customization

To change the style menu by inserting our own styles, proceed as follows, by using two sub-properties, **documentStyles** (to apply our styles to the editor content) and **styles** (to show the style list in the drop-down menu):

```
…
// Customize the Style menu
var aStyles = flatten
(
    [
        {rule: 'p.Red', text: 'Red text'},
        {rule: 'h1', text: 'Header 1'},
        // Inline styles (will become span.SuperScript and
span.SubScript)
        {rule: '.SuperScript', text: 'Superscript'},
        {rule: '.SubScript', text: 'Subscript'}
    ]);

var sStyleClasses = 'p.Red{color: #FF0000;}
span.SuperScript{vertical-align: super; font-size: 80%;} \
                    span.SubScript{vertical-align: sub; font-size:
80%;}';
…
var cnf =
{
    …
    css:
    {
        // To see the styles applied in the editor content
        documentStyles: sStyleClasses,
        // To have the styles list in the drop down menu
        styles: aStyles
    },
    …
};

return cnf;
…
```

The result will be:

The content source will be:

```
<h1>Header 1</h1>
<p class="Red">Red text</p>
<p>This is <span class="SuperScript">Superscript</span></p>
<p>This other is <span class="SubScript">Subscript</span></p>
```

7. **Example 2: customizing the code view and import/export HTML markup visibility**

In certain circumstances, it's necessary to hide such capabilities, except certain groups of users. To perform this, is necessary to discover if the current user belongs to these groups. The code we can put into the **tbio_config.jsp** file to perform this, is the following one:

```
…
<%@page import="com.ibm.wps.puma.*"%>
<%@page import="java.util.*"%>
…
<%!

/**
 * This function returns the WCM workspace for the current user.
 *
 * @param req The HttpServerRequest object.
 * @return The WCM workspace (Workspace type) or null if error occurs.
 */
public Workspace getWorkspace(HttpServletRequest req)
{
    Principal princ;
    Workspace ws;

    ws = null;

    try
```

```
    {
        princ = req.getUserPrincipal();

        if (princ == null)
        {
            // Get the WCM workspace for the anonymous user.
            ws = WCM_API.getRepository().getAnonymousWorkspace();
        }
        else
        {
            // Get the WCM workspace for the current logged-in user.
            ws = WCM_API.getRepository().getWorkspace(princ);
        }

        ws.useUserAccess(true);
    }
    catch (Exception e)
    {
        ws = null;
    }

    return ws;
}

/**
 * This function verify if the current user belongs to at least one of
the
 * passed groups.
 *
 * @param ws The WCM workspace.
 * @param vectGroups A Vector object containing the names of the
groups to
 * scan.
 * @return true if current user is at least member of one of the
groups,
 * otherwise false.
 */
public boolean isCurrentUserBelongingTo(Workspace ws, Vector
vectGroups)
{
    boolean f;
    int i;
    String sGroup;

    f = false;

    for (i = 0; i < vectGroups.size(); ++i)
    {
        sGroup = (String)vectGroups.elementAt(i);

        if (ws != null && ws.isMemberOfGroup(sGroup))
        {
            f = true;
```

```
            break;
        }
    }

    return f;
}

%>
…
<%

/*
 * The scope of this piece of code is to verify if the current user
 * belongs at least to one of the groups passed through a Vector
object.
 * It's possible here to modify this list by
changing/inserting/deleting
 * the rows of code:
 * vectGroups.addElement("myGroupName");
 *
 * The returned fMember flag will be used to set the visibility of the

 * HTML source switch.
 */
Workspace ws;
boolean fMember;
Vector vectGroups;

vectGroups = new Vector();
vectGroups.addElement("wpsadmins");
vectGroups.addElement("wcmadmins");
vectGroups.addElement("anothergroup");

ws = getWorkspace(request);
fMember = isCurrentUserBelongingTo(ws, vectGroups);

%>
…
// Javascript code
var fCodeViewEnabled = <%=fMember%>;
…
var grpTools =
{
    label: 'category.tools',
    items:
    (
       fCodeViewEnabled ?
       [
           'find',
           'accessibility',
           Button
           (
               'WcmExport',
```

```
                'EXPORT_BUTTON',
                'upload.svg',
                exportRTFHTML
            ),
            Button
            (
                'WcmImport',
                'IMPORT_BUTTON',
                'download.svg',
                importRTFHTML
            ),
            'fullscreen',
            'usersettings'
        ] :
        [
            'find',
            'accessibility',
            'fullscreen',
            'usersettings'
        ]
    )
};
…
// Compute the whole items array
var aItems = flatten
(
     [
         [
             'undo', 'insert', 'style', 'emphasis',
             'align', 'listindent', 'format'
         ],
         [grpTools]
    ]);
…
var cnf =
{
    …
    codeview:
    {
        enabled: fCodeViewEnabled
    },
    …
    ui:
    {
        …
        toolbar:
        {
            items: aItems,
            …
        }
```

```
    },
    …
};
```

8. ### Example 3: language property customization

As example of configuration, let's suppose to want English (US) and Italian, then the configuration object will be:

```
var cnf =
{
    …
    ui:
    {
        languages: ['en_us', 'it'],
        …
    },
    …
};
```

9. ### Example 4: contextual menu customization

Let's suppose to want to hide the table's contextual menu and leave the image's contextual menu:

```
// Compute the whole items array
var aItems = flatten([…]);
…
var cfg =
{
    …
    ui:
    {
        …
        toolbar:
        {
            items: aItems,
            contextual: ['image-tools']
        }
    },
    …
};
```

10. ### Example 5: custom menu

The add menu in the WCM version of the Textbox.io editor is a custom menu coming out with a number of items:

- Insert a link to a WCM content button
- Insert an image button
- WCM tag helper button
- Insert an external content manager link
- Insert a media
- Insert a table
- Insert a special character

- Insert a horizontal line

The source is:

```
…
var grpInsert =
{
    label: 'category.insert',
    items:
    [
        {
            id: 'insert',         // This is required to get the +
icon
            label: 'insert.menu', // This is required to get the
                                  // translated "Insert" hover text
            items:
            [
                [
                    Button
                    (
                        'WCMInsertLink',
                        'INSERT_LINK_TO_WCM_CONTENT',
                        'link.svg',
                        performInsertLinkIntoRTF
                    ),
                    Button
                    (
                        'WCMInsertImage',
                        'INSERT_IMAGE_FROM_WCM_LIBRARY',
                        'image.svg',
                        ephox_performInsertImgIntoRTF
                    ),
                    Button
                    (
                        'WCMTagHelper',
                        'TAG_HELPER_BUTTON',
                        'tag.svg',
                        performInsertTagIntoField
                    ),
                    {
                        id: 'WCMInsertECMLink',
                        text: buttonText('ECM_LINK_BUTTON'),
                        icon: buttonImage('globe.svg'),
                        action: function ()
                        {
                            var pickerUnavailable =
buttonText("FED_DOCS_PICKER_UNAVAILABLE");
                            var pickerDialogTitle =
buttonText("FED_DOCS_PICKER_DIALOG_TITLE");
                            insertEcmLink
                            (
                                wcmId,
                                function()
                                {
```

```
                                    elEcmLaunched = false;
                                },
                                pickerUnavailable,
                                pickerDialogTitle
                            );
                        }
                    },
                    'media',
                    'table'
                ],
                [
                    'specialchar',
                    'hr'
                ]
            ]
        }
    ]};
…
var aItems = flatten
(
    [
        ['undo', grpInsert, 'style', 'emphasis', 'align',
'listindent'],
        …
    ]);
…
var cfg =
{
    …
    ui:
    {
        …
        toolbar:
        {
            items: aItems
        }
    },
    …
};
```

```
return cfg;
…
```

This is a good example on how to build a completely custom menu. If we want to remove some items from this custom menu it is enough to drop some line of code. Let suppose to want to remove some items from this menu, in particular:

- WCM tag helper button
- Insert an external content manager link
- Insert a special character

It's enough to change the **grpInsert** object in this way:

```
var grpInsert =
{
    label: 'category.insert',
    items:
    [
        {
            id: 'insert',         // This is required to get the +
icon
            label: 'insert.menu', // This is required to get the
                                  // translated "Insert" hover text
            items:
            [
                [
                    Button
                    (
                        'WCMInsertLink',
                        'INSERT_LINK_TO_WCM_CONTENT',
                        'link.svg',
                        performInsertLinkIntoRTF
                    ),
                    Button
                    (
                        'WCMInsertImage',
                        'INSERT_IMAGE_FROM_WCM_LIBRARY',
                        'image.svg',
                        ephox_performInsertImgIntoRTF
                    ),
                    'media',
                   'table'
                ],
                ['hr']
            ]
        }
    ]};
```

The result will be:

The horizontal line appearing in the menu that separates the first four items from the last one is due to the presence of two distinct sub-arrays as values for the **items** property of the **grpInsert** object:

```
var grpInsert =
{
    label: 'category.insert',
    items:
    [
        {
            id: 'insert',         // This is required to get the +
icon
            label: 'insert.menu', // This is required to get the
                                  // translated "Insert" hover text
            items:
            [
[…], ['hr']
            ]
        }
    ]};
```

11. ## Example 6: remove other top-level items

Let suppose to want to remove the following items:

a. The underline action
b. The indent and outdent actions
c. All format items except removeformat

The code will then be:

```
…
// Removed the 'underline" item
var grpEmphasis =
{
    label: 'category.emphasis',
    items: ['bold', 'italic']};

// Removed the "indent" and "outdent" items
var girpIdent =
{
    label: 'category.indent',
    items: ['ul', 'ol']};

// Removed all format items except the 'removeformat' one
var grpFormat =
{
    label: 'category.format',
    items: ['removeformat']};
…
// Compute the whole items array
var aItems = flatten
(
    [
        ['undo', 'insert', 'style', grpEmphasis', 'align', grpIndent],
        styleGroup(grpFormat),
        ['tools']
    ]);
…
var cfg =
{
    …
    ui:
    {
        toolbar:
        {
            items : items
        }
    },
    …
};

return cfg;
…
```

12. **Final example: put all together**

    By performing all the above examples of customization we will obtain the following result (a very reduced toolbar):

Note that the **More** action

used to expand the toolbar in a second line, is not present yet, because now there is enough space for all the toolbar items on a single line.

13. ## Apply the customization

Changing content of our copy of the configuration file tbio_config.jsp isn't enough to see our customizations on the Textbox.io editor; whenever we need to apply a new customization to WCM, it is necessary to go through the following steps:

a. Copy our customized version of the tbio_config.jsp file to the directory

   [wp_profile root]/PortalServer/wcm/shared/app/config/textboxio

   (if the directory doesn't exist, create it before).

b. From the directory **[wp_profile root]/ConfigEngine**, run the command

   ConfigEngine.sh configure-wcm-ephox-editor-custom-configuration -DWasPassword=password -DPortalAdminId=wpsadmin -DPortalAdminPwd=password

   This may take several minutes to complete. At the end, we should verify that everything went properly (we must see the text **BUILD SUCCESSFUL** and not **BUILD FAILED**):

   In the case of BUILD FAILED, verify the command's syntax, the command's parameters and the presence of the configuration file in the correct directory, then relaunch the command, even if you haven't found any error.

c. Restart WebSphere Portal.

14. Textbox.io services

The Texbox.io additional services are provided through a WebSphere enterprise application, **EphoxTbioServices**, in the form of **tbioServices_wcm.ear** file. If the application is not installed, we must do so through the path *Applications -> Application Types -> WebSphere Enterprise Applications* of the WebSphere console. We can find the file to install in the

[WebSphere Portal root]/installableApps

directory. It's is also important to check that the application is started:

Textbox.io services include the following three modules:

a. **ephox-allowed-origins.war** to allow all server-side components to communicate with specified domains, now deprecated.
b. **ephox-spelling.war** to manage spell checking and autocorrect.
c. **ephox-image-proxy.war** to allow editing images from the web.

as shown by the WAS console screen accessible through the path *Enterprise Applications -> EphoxTbioServices -> Manage Modules*:

Further details are available at [7].

Spell checking

The spell checking is enabled if Textbox.io services are started. It includes optional server-side spell checking and autocorrect as you type for a number of common languages.

Spell checking and web image insertion require the deployment of several server-side components onto WebSphere Application server. It don't need further configuration/customization activities unless **EphoxTbioServices** were installed on a different server.

Further details are available at [8] and at [9].