

1. Textbox.io Documentation	3
1.1 Developer Guide	4
1.1.1 System Requirements	5
1.1.2 Getting Started	7
1.1.3 API Basics	9
1.1.3.1 Creating and Removing Editors	10
1.1.3.2 Getting Editor Instances	11
1.1.3.3 Setting and Getting Content	12
1.1.3.4 HTML Code View	13
1.1.4 Customizing the Editor	15
1.1.4.1 Configuration Object	17
1.1.4.2 Changing the Toolbar	18
1.1.4.2.1 Adding custom commands	20
1.1.4.3 Customizing the User Interface Colors	22
1.1.4.4 Using Your Own Document Styles	23
1.1.4.5 Filtering Content	27
1.1.4.6 Macros: Writing Content-Aware Code	29
1.1.4.7 Creating Dynamic Content	30
1.1.5 Editor types - Classic vs Inline	31
1.1.6 Working with Images	33
1.1.6.1 Handling Local Images	34
1.1.6.1.1 Handling Asynchronous Image Uploads	37
1.1.6.1.2 Node.js Upload Handler	38
1.1.6.1.3 PHP Upload Handler	41
1.1.7 Checking Spelling	43
1.2 API reference	44
1.2.1 configuration	45
1.2.1.1 autosubmit	47
1.2.1.2 basePath	48
1.2.1.3 codeview	49
1.2.1.4 css	50
1.2.1.4.1 documentStyles	51
1.2.1.4.2 styles	52
1.2.1.4.3 stylesheets	55
1.2.1.5 images	56
1.2.1.5.1 allowLocal	57
1.2.1.5.2 editing	58
1.2.1.5.3 upload	60
1.2.1.6 links	62
1.2.1.6.1 embed	63
1.2.1.6.2 validation	64
1.2.1.7 macros	65
1.2.1.8 paste	67
1.2.1.9 spelling	69
1.2.1.10 ui	70
1.2.1.10.1 aria-label	71
1.2.1.10.2 autoresize	72
1.2.1.10.3 colors	73
1.2.1.10.4 fonts	75
1.2.1.10.5 languages	77
1.2.1.10.6 locale	80
1.2.1.10.7 shortcuts	82
1.2.1.10.8 toolbar	83
1.2.2 editor	94
1.2.2.1 editor.content	95
1.2.2.1.1 editor.content.set()	96
1.2.2.1.2 editor.content.get()	97
1.2.2.1.3 editor.content.insertHtmlAtCursor()	98
1.2.2.1.4 editor.content.documentElement()	99
1.2.2.1.5 editor.content.uploadImages()	100
1.2.2.1.6 editor.content.getSelectedText()	101
1.2.2.1.7 editor.content.isDirty()	102
1.2.2.1.8 editor.content.setDirty()	103
1.2.2.1.9 editor.content.selection	104
1.2.2.2 editor.element	106
1.2.2.3 editor.events	107
1.2.2.3.1 editor.events.loaded	108
1.2.2.3.2 editor.events.focus	111
1.2.2.3.3 editor.events.dirty	114
1.2.2.3.4 editor.events.change	117
1.2.2.4 editor.filters	120
1.2.2.4.1 selector	121
1.2.2.4.2 predicate	124
1.2.2.5 editor.focus	127
1.2.2.6 editor.macros	128
1.2.2.6.1 editor.macros.addSimpleMacro()	129
1.2.2.6.2 editor.macros.removeMacro()	130
1.2.2.7 editor.message	131
1.2.2.8 editor.mode	132

1.2.2.8.1 editor.mode.get()	133
1.2.2.8.2 editor.mode.set(mode)	134
1.2.2.9 editor.restore	135
1.2.3 textboxio	136
1.2.3.1 get	137
1.2.3.2 getActiveEditor	138
1.2.3.3 inline	139
1.2.3.4 inlineAll	140
1.2.3.5 isSupported	141
1.2.3.6 replace	142
1.2.3.7 replaceAll	143
1.2.3.8 version	144
1.3 Server-Side Components	145
1.3.1 Installation and Setup	146
1.3.2 Logging	155
1.3.3 Embed Rich Media	156
1.3.3.1 Configure Enhanced Media Embed Server	157
1.3.3.2 Integrate Enhanced Media Embed Server	162
1.3.4 Adding Custom Dictionaries	168
1.4 Accessibility	170
1.4.1 Textbox.io Accessibility Compliance	171
1.4.2 Creating Accessible Content	172
1.5 Help & Support	173
1.5.1 Web Services Troubleshooting	174
1.5.1.1 Browser Specific Issues	175
1.5.1.2 Error Messages About the "Origins" or "Spelling Service Missing"	176
1.5.1.2.1 Using Browser Tooling to Investigate Services Issues	179
1.5.1.3 General Troubleshooting	181
1.5.1.4 Out of Memory Errors	182
1.5.1.5 Troubleshooting Tools - curl	185

Textbox.io Documentation

Welcome to Textbox.io

Textbox.io is a tool for creating great looking content in online apps. Whether it's in social communities, blogs, emails, or anything in between, Textbox.io lets people express themselves on the web.

We believe that writing isn't just a task, it's how we express who we are in a new digital world. The sum of what you write is often far more than the characters on the page. Writing matters, and the tools we use to write should matter too.

Getting Started

Textbox.io consists of a JavaScript-based rich text editor and a set of server components that provide functionality to enrich the editing experience.

Getting started with Textbox.io is easy. See the [Getting Started](#) guide on how to install the editor client onto a web server and be up and running with the [Textbox.io SDK](#) in minutes.

More Information

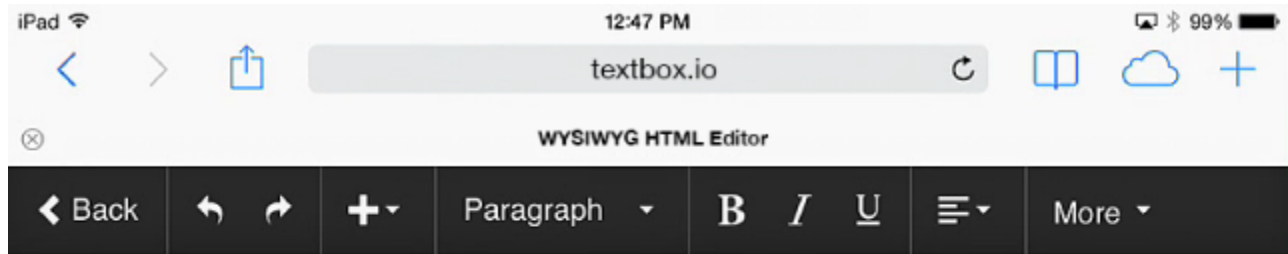
Features, [examples](#) and pricing information can be found at <http://textbox.io>

Getting Help

Please open a support case or post on the online forums at the [Tiny Support](#) center. Before contacting support, review the [information our support team needs](#) to assist you.

Developer Guide

The WYSIWYG HTML Editor SDK. Designed for Mobile and Desktop.



Write for the Web

Learning the craft of how to write for the Web is a definite departure from doing so for a newspaper (remember those?), magazine or brochure. The writing styles are distinctly different for each medium.

On the Web, today's readers are often distracted: they might very well be reading your content, blogs or articles on small-screen mobile devices while they're on the go doing a hundred different things. That makes it tougher to grab--and hold--their attention. They want to see visuals and great, provocative headlines (or they won't click on your content). And, if they do click on it, they want to scan for meaning instead of parsing every word.

Writers have to remember that one of the fundamental differences between learning how to write for the Web vs. writing for print involves the difference between narrative writing and actionable writing.



Textbox.io is a tool for creating great looking content in online apps. Whether it's in social communities, blogs, emails, or anything in between, Textbox.io lets people express themselves on the web.

Textbox.io uses the latest HTML5, JavaScript and CSS3 technologies to deliver the best possible editing experience. Key design goals of Textbox.io include:

- Provide a minimal, functional and responsive user interface designed to deliver great usability in any application
- Support mobile devices with first class editing and responsive design
- Deliver unbeatable copy and paste from Microsoft® Office™
- Create clean, well formatted HTML in all circumstances
- Provide a simple yet powerful API to enable easy integration
- Leverage the power of modern web standards

The Basics

Working with Textbox.io is simple. In the most basic terms, all you need to do to integrate Textbox.io is:

1. Load the `textboxio.js` library
2. Call the `replace` method to replace `<div>` or `<textarea>` elements with a Textbox.io instance.

Textbox.io can be used as either a standalone editor client or combined with the [Server-Side Components](#). These Server-Side Components offer a set of supporting services that can be used to enrich the editing environment.

Getting Started

See the [Getting Started](#) guide to start using Textbox.io in your applications in minutes.

System Requirements

- [Browsers](#)
- [Web Services](#) (for optional server-side components)

Supported Browsers

Textbox.io is based on HTML5 JavaScript and CSS3 standards and will operate on most web browsers that support these standards.

If your application needs to support older browsers, use the [textboxio.isSupported\(\)](#) API to verify whether the editor is available at runtime.


Supported Browsers by Platform

The following client platforms are fully supported for Textbox.io.

Platform	Browsers
Windows	Microsoft Edge* Internet Explorer 11 Chrome* Firefox* Firefox ESR*
OS X	Safari 9+ Chrome* Firefox* Firefox ESR*
Linux	Chrome* Firefox* Firefox ESR*
iOS 9	Safari
Android 5+	Chrome*

* Current stable channel version.

Internet Explorer must be in Edge mode


 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

[More information \(external link\)](#)

Importing Images from Microsoft Office

 Importing images from Microsoft Word content requires the use of the Adobe Flash plugin on the following browsers:


- Safari
- Microsoft Edge

If the Adobe Flash plugin is not available or is blocked in the client browser then textual content will be imported but no image content will be imported.

Copy and paste on Internet Explorer 11, Firefox (version 45+) and Chrome (version 54+) does not require any additional plugins.

Copy and paste from Microsoft Office is not supported on mobile device platforms.

Embedded Web View Support for Mobile Platforms

 For information on how to use Textbox.io within the web view of an iOS or Android application please contact us.

Tiny is currently working with selected app developers to bring this functionality to Textbox.io in future. If you're interested in working with Textbox.io inside an iOS or Android application (other than the browser) then please get in touch with Tiny via [support.tiny.cloud](#).

Web Services

Server-side Components require a Java Web Application Server that supports Servlet Implementation API 3.0.

Java Development Kit

JDK 7 update 55+

Java (J2EE) Application Servers

Tomcat 7+

Jetty 8+

Operating Systems

Windows Server 2008 SP2

Red Hat Enterprise Linux v6

Red Hat Enterprise Linux v5

Minimum Hardware Requirements

CPU: Dual Core Processor ~ 2Ghz. For higher loads, a quad core or higher is recommended.

RAM: 4 Gigabytes of RAM available for services

Platform Support

 Tiny is committed to broadening support for various application servers / platforms. If you have a requirement to support an application server that is not listed here, please contact support@tiny.cloud.

Getting Started

Getting started with Textbox.io is super easy. In this guide you will invoke Textbox.io on a `<textarea>` as part of an HTML `<form>`. When the `<form>` is submitted, the contents of the Textbox.io editor will be submitted as part of the `<form>` POST.

To complete this guide, you will need access to a web server. This guide assumes a web server is running on port 80 on localhost.

Step 1: Download a copy of Textbox.io and put it on a web server

- [Download the Textbox.io SDK](#) from Tiny.
- Unzip the package and open the `textboxio-all` or `textboxio-client` directory.
- Move the `textboxio` subdirectory into a web accessible location on your web server (for example, `localhost`).

Step 2: Add Textbox.io to a page

With Textbox.io accessible via your web server, you can now include the Textbox.io script anywhere you would like to use the Textbox.io editor.

To add the script, add the following inside your page's `<head>` tag.

```
<script src="http://YOUR-DOMAIN/YOUR-DIRECTORY/textboxio/textboxio.js"></script>
```

Step 3: Invoke Textbox.io as Part of a Web Form

With the script included, you may then invoke Textbox.io on any element (or elements) in your webpage.

Textbox.io lets you identify elements to *replace* via a [CSS3 selector](#). To add Textbox.io to a page you pass a selector to `textboxio.replace()`.

In this example, you will *replace* `<textarea id="mytextarea"></textarea>` with a Textbox.io editor by passing the selector `'#mytextarea'` to `textboxio.replace()`.

```
<!DOCTYPE html>
<html>
<head>
  <script src="http://localhost/textboxio/textboxio.js"></script>
</head>
<body>
  <h1>Textbox.io Getting Started Guide</h1>
  <form>
    <textarea id="mytextarea"></textarea>
    <button type="submit">Submit</button>
  </form>
  <script type="text/javascript">
    var editor = textboxio.replace('#mytextarea');
  </script>
</body>
</html>
```

For the best user experience, it is recommended that the HTML5 doc type is set.

 `<!DOCTYPE html>`

Try it:

You've added Textbox.io to the page - that's all there is to it!

Next, we'll look at retrieving content as part of a `<form>` POST.

Step 4: Saving Content with a `<form>` POST

When the form is submitted, the Textbox.io editor mimics the behavior of a normal HTML `<textarea>` during a form POST. No additional configuration is required.

What's Next

At this point, you've seen how to create Textbox.io instances on a page using `textboxio.replace()`, and you've seen how to retrieve content from Textbox.io as part of an HTML `<form>` POST. There's way more under the hood if you're feeling adventurous: read on for more Textbox.io goodness:

- [Editor types - Classic vs Inline](#) - Learn about the 2 modes of editing supported by Textbox.io, and decide which works best for you
- [API Basics](#) - Learn the basics of using the editor API, such as getting and setting content in the editor.
- [Customizing the Editor](#) - Learn how to configure and customize Textbox.io for your applications.
- [Customizing the User Interface Colors](#) - How to change the appearance of Textbox.io
- [Working with Images](#) - Learn how to work with images for image editing and local image upload.
- [API reference](#)

API Basics

This section explores several important concepts for using Textbox.io within your application.

Creating and Removing Editors

Learn how to create and remove editor instances from pages using the `replace()` and `restore()` methods.

Getting Editor Instances

How to obtain a reference to a particular editor. Using an editor reference, you can get and set content and manipulate a particular editor in the page.

Setting and Getting Content

Textbox.io supports several mechanisms of setting and getting content. Learn which will be of most use within your application.

HTML Code View

Creating and Removing Editors

The `textboxio` JavaScript global enables you to create, modify and interact with instances of the editor. This is the starting point for any integration. This global is available immediately after the editor's JavaScript file has been loaded on the page.

The `replace` and `editor.restore` methods are the two most important methods in the API. They enable editor instances to be created and removed.


Creating Editors

The `replace` method is used to create a Textbox.io editor. The method takes a [CSS3 selector](#) as an argument, which identifies a target element on the page, and then replaces it with an editor. This target element must be either a `<div>` or `<textarea>` element. The target element's contents are automatically loaded into the new Textbox.io editor.

This method returns an instance of `textboxio.editor`. See [Textbox.io API Reference](#) for more information on using `textboxio.editor`.

Create customized instances of Textbox.io by providing an [editor configuration](#) as a second argument. See the [Customizing the Editor Developer Guide](#) for more information.

Form Handling / Editor Contents

 When used in conjunction with a `<textarea>` in an HTML `<form>` the content contained within Textbox.io will be submitted as part of the form's POST operation.

To retrieve the editor contents using JavaScript see [Setting and Getting Content](#).

Restoring the source element (Removing the editor)

The `editor.restore` method enables you to remove Textbox.io instances from the page, restoring the original target element. When called, `editor.restore` updates the target element's HTML content to match the HTML content of the Textbox.io editor.

Getting Editor Instances

Get the Active Editor Instance

Access the current or last active editor using `textboxio.getActiveEditor()`. The *active* editor is the editor that was most recently focused by a user. If no editor has been focused, then the first editor on the page is returned.

Get Editor Instances

Textbox.io remembers the selector that was used to create each editor. A developer may retrieve editor instances by passing that selector to `textboxio.get()`. Note, that this method returns an array of editors.

Setting and Getting Content

Textbox.io supports several mechanisms of setting and getting content. This article outlines a number of common methods for working with HTML content using Textbox.io.

Note that both programmatic (JavaScript) and automatic (HTML + POST) methods for working with content are available.

Setting Content with Javascript

A developer can explicitly set the content of an `editor`, rather than rely on the automatic behavior of `replace`. To set the HTML contents of an `editor`, simply pass a string of HTML content to `editor.content.set()`.

Setting Content Automatically with `replace()`

When adding a [Textbox.io](#) editor to a page using `replace`, the HTML content for any matched element is passed to the newly created `editor` instance. In this way [Textbox.io](#) can be used to edit the HTML contents of any specified `<textarea>` or `<div>` without the need for a specific *setting* of the editor's content - the editor simply reads in the contents of the matched element.


Getting Content with Javascript

To get the content of an `editor` explicitly, a developer can use `editor.content.get()`. This method returns a string representation of the editor's HTML content.

Getting Content Automatically with HTML `<form>` POST

When [Textbox.io](#) is used to *replace* a `<textarea>` element within a `<form>`, [Textbox.io](#) will mimic the behavior of the original `<textarea>` when the form is submitted. This means that the contents of the editor are simply posted along with the other form elements, using the name attribute of the original `<textarea>` as the key.

Image Content

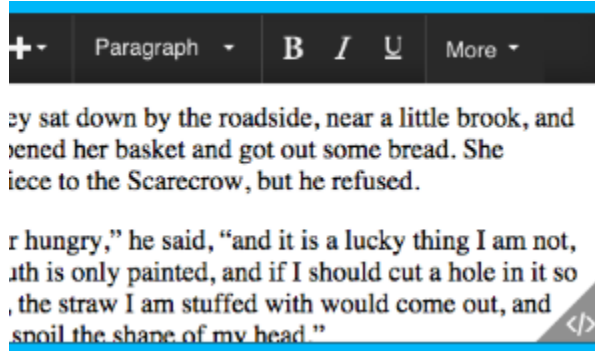
 By default [Textbox.io](#) saves local images as [Base64 data URIs](#). This encodes images directly within the HTML of the content rather than separate files. This enables images to be imported from Microsoft Word and images to be available in the document immediately without uploading.

See [Handling Local Images](#) for information on uploading local image content to your application.

HTML Code View

Textbox.io includes the ability for users to directly edit HTML via code view. This provides power users with more flexibility over the content they create and includes features such as automatic tag completion and matching.

When enabled, the user can toggle between *code view* and *design view* by clicking on the icon in the bottom right corner of the editor container



Editor container with code view enabled. The user can toggle between code and design views by clicking the icon in the bottom right corner.

Code view is only available in [classic mode](#) and is enabled by default.

Switching between code view and design view programmatically

The Textbox.io API provides functionality to toggle between code and design views via `editor.mode.set(mode)`.

```
var editor = textboxio.replace('#id');
editor.mode.set('code'); // switches the editor into HTML markup editing mode
editor.mode.set('design'); // switches the editor into WYSIWYG ('design') mode
var currentMode = editor.mode.get(); //returns the current mode, either 'design' or 'code'.
```

Hiding the code view button

If you wish to switch between code view and design view programmatically, you may wish to disable the code view button. To disable the code view button but leave the code view feature enabled, set the `configuration.codeview.showButton` property to `false`.

```
var config = {
  codeview: {
    showButton : false // Hides the code view button, default is true (shown)
  }
};

var editor = textboxio.replace('#id', config);
```

Disabling code view

In some instances, you may need to disable code view feature via the `configuration.codeview.enabled` property.

When you disable the code view feature, the button is also removed from the Textbox.io UI.

```
var config = {
  codeview: {
    enabled : false // Disables code view feature, default is true (enabled)
  }
};

var editor = textboxio.replace('#id', config);
```

Filtering

Toggleing between code and design views triggers filter logic (both external and potentially via plugins) to be executed on editor content. This is necessary to sanitize content and remove superfluous UI (such as spelling underlines, etc) from content when switching.

Logic is as follows:

- **Switching from design to code view** - code view's content is run through the `output` filter chain
- **Switching from code view to design view** - the design view's content is run through the `input` filter chain

See [Filtering Content](#) for more information on content filtering.

See also:

- [API reference](#)
- [Filtering Content](#)
- [Customizing the Editor](#)

Customizing the Editor

Textbox.io provides a number of API features enabling you to integrate it tightly with your application.

Basic Customizations

Configuration Object

The configuration object serves as the foundation to all customizations you can make in Textbox.io. This guide will show you what it is, and what you can configure.

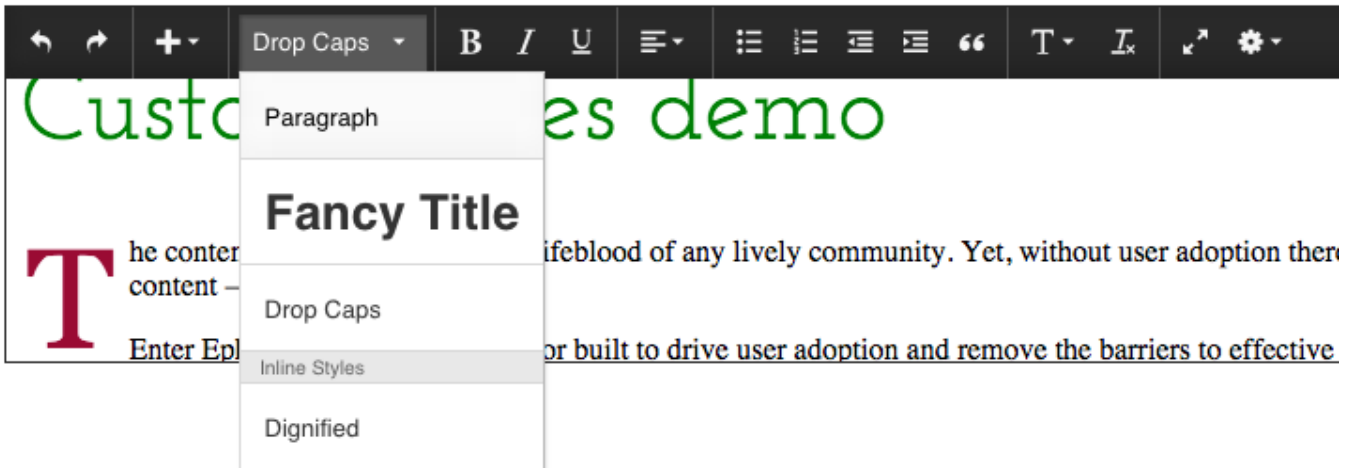
Customizing the User Interface Colors

The colors of the Textbox.io user interface are completely customizable. Textbox.io is packaged with a simple theme builder tool that enables designers to customize the colors of the editor to suit their application.

Changing the Toolbar

You can choose which items appear on the toolbar as well as the order and grouping of toolbar items. Additionally, you can add [your own custom items](#) to the toolbar.

Using your Own Document Styles



Textbox styles drop-down menu

Textbox.io can be configured to use custom stylesheets to more closely reflect the theme and styling of your web application's published content. Additionally, you can specify custom styles to appear in the editor's styles drop-down menu.

Advanced Customizations

Filtering Content

You can create custom code to manipulate pieces of the document as content is **get** and **set** in the editor. This enables you to build a number of unique features, such as showing invisible elements to the editor user, or removing styles from editor content "Just in Time" before they are published

The [Filtering Content](#) guide will show you how to get started on building your own filters.

Macros: Writing Content-Aware Code

Textbox.io's macro engine provides real-time pattern detection and manipulation of content by custom javascript functions, *as the user types!*

The [macro guide](#) will take you through an example macro to replace text as you write.

Creating Dynamic Content

This guide will show you how to create customizations that leverage both macros and filters to allow handling of dynamic content in the editor.

Configuration Object

Textbox.io is configured by providing a [configuration object](#) as a parameter to the [replace](#) method. The configuration object enables you to control:

- the [functionality available on the toolbar](#),
- the [CSS](#) used to render the HTML content,
- the behaviour of [copy paste](#) functionality, and
- the [Textbox.io Server Components](#) available in an editor.

Providing a configuration to Textbox.io is optional. You can choose to override all or part of the default configuration object when supplying an object as a parameter to the [replace](#) method. Textbox.io is provided with a set of configuration defaults for the toolbar and paste functionality (see [Configuration Defaults](#)).

It's strongly recommended you provide a style sheet configuration for rendering the content. Textbox.io's default configuration only specifies the set of toolbar commands and paste operation behavior. It does **not** specify a style sheet for rendering the content or any services for use with the editor.

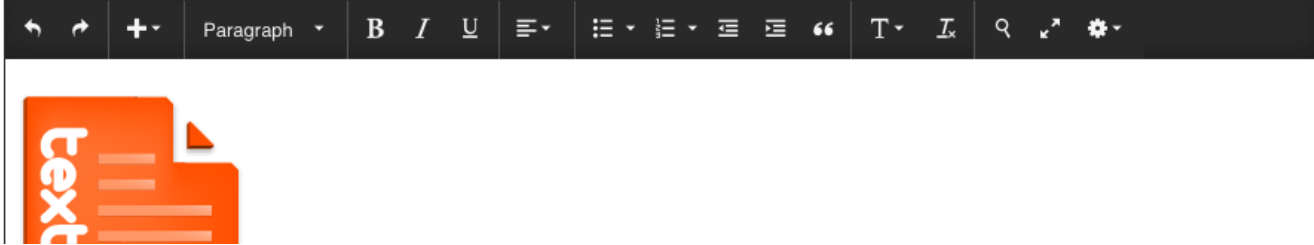
Example

For more information on the configuration options for [Textbox.io](#) see the [configuration](#) documentation in [API reference](#).

Changing the Toolbar

The items that appear in the editor toolbar, as well as the order in which they appear, can be customized by the Textbox.io API. Additionally, you can define your own [custom toolbar commands](#) to extend the functionality of the editor and tailor it to your specific needs.

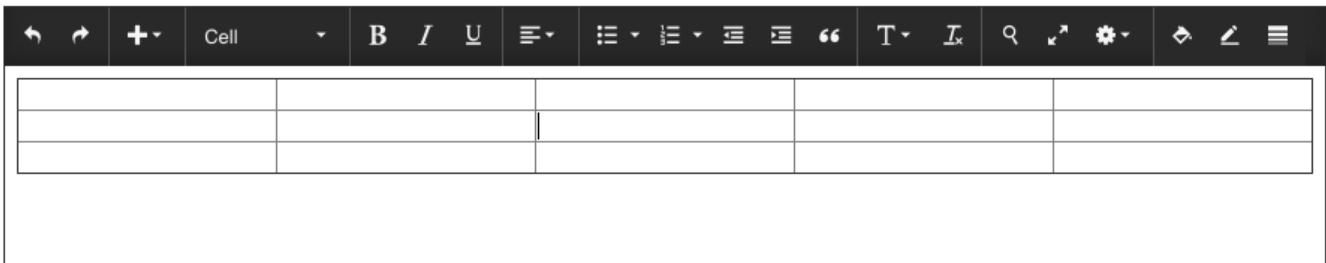
The stock toolbar configuration



Stock editor toolbar

The editor provides a stock toolbar configuration if one is not explicitly defined. This contains a number of items of core functionality.

Contextual toolbar items



Stock editor toolbar with contextual items showing

The editor will show extra toolbar items depending on the selection context. These can be disabled in the [toolbar](#) configuration.

Defining the toolbar configuration

The items in the toolbar are defined in the editor [configuration object](#).

Out of the box, Textbox.io contains a number of available toolbar ("Command") items. Each item is referenced by a [Command Item ID](#), and each button you see in the editor has its own Command Item ID- for example 'bold', 'italic', 'underline'.

Further, there are aliases to groups of functionality in the editor to enable you to quickly add blocks of functionality to the editor. These are outlined in [Command Group Aliases](#).

The following example shows an editor where the toolbar items have been explicitly defined using [Command Group Aliases](#):

Grouping buttons

When customizing the editor, it is often useful to organize toolbar command items into groups of functionality. Groups provide users with a visual separation in the UI and commands within a group move to the More Drawer as group of items when required.

Visually, toolbar groups appear as a set of items between 2 thin vertical lines:



Groups are defined inline to the toolbar item configuration.

An example of an editor with 2 groups is shown below:

Advanced toolbar customization

Textbox.io also provides the ability to create your own toolbar commands. Read more about this in the [Adding custom commands](#) guide.

Adding custom commands

Creating custom toolbar commands

[Textbox.io](#)'s toolbar can be customized to suit the needs of your application. Buttons on the toolbar are referred to as *commands*. *Custom commands* can be created at initialization time as part of the editor configuration.

Important

When using custom commands, you'll need to explicitly define the rest of your toolbar configuration (i.e. stipulate all items to appear). See [Changing the Toolbar](#) for more information.

A typical custom command can be defined appear as the following:

```
var customCommand = {
  id : 'myCommand',
  text : 'my custom command',
  icon: './path/to/my/image.png',
  action : function () {
    alert('hello world');
    // write custom functionality to occur when the button is clicked here.
  }
}
```

Properties:

Property	Type	Description
id	String	A unique, user-defined key to identify the custom command. This enables the command to be uniquely identified at runtime
text	String	The text to display as a tooltip when a user hovers over the command
icon	String	a URL path (relative or absolute) to a 13*13px image for the command's icon.
action	function	A function defining the action to occur when the user clicks on the button.

Including custom commands in your toolbar

Including the custom command in your toolbar requires specifying where you'd like it in your toolbar:

```
var myEditor = textboxio.replace('#someDiv', {
  ui:{
    toolbar: {
      items: ['undo','insert', 'style', 'emphasis', 'align', 'listindent', 'format', 'tools',
// other toolbar items
      { //create a group to house our custom command
        label: 'Custom commands group',
        items: [customCommand] //our newly created command
      }
    ]
  }
});
```

Basic example

This example shows how to create a basic custom item and add it to the toolbar.

Accessing the editor from action functions

Often when writing action functions for custom commands, you'll want to do something meaningful with the editor content. To do this, use `textboxio.getActiveEditor()` to obtain a reference to the editor where the cursor is currently located.

See the [API reference](#) for more information.

Example

Customizing the User Interface Colors

The user interface colors of Textbox.io can be customized to match with the theme of your application.

All the tooling that you require to theme Textbox.io can be found within the *theme* folder of the Textbox.io distribution. Several example themes are also packaged in this directory.

Creating a Theme

The theme building tool requires knowledge of CSS syntax and use of command line tools. It also requires a node.js installation (link provided in *README.html*).

The *README.html* file in the *theme* folder contains instructions on how to setup and use the theme building tool packaged with Textbox.io.

The tool enables designers to specify a small set of colors that are then compiled into a complete theme for use with Textbox.io.

Themes are compiled into a set of CSS files that are placed in the *textboxio/resources/css* folder ready for deployment.

Textbox.io Theme Tooling

The *theme* folder of the Textbox.io distribution contains a theme building tool that can be used to easily create Textbox.io themes. This tool consists of 3 main parts:


- *README.html* - Instructions on how to use the theme builder and live preview the compiled theme
- *theme.css* - A file containing the variables representing the colors that can be set for Textbox.io's interface
- *Gruntfile.js* and *package.json* - A grunt build script and supporting NPM package that automatically compiles the editor theme when you change *theme.css*

Deploying a Theme

Once a theme has been built the resulting CSS files are placed into the *textbox.io/resources/css* folder ready to be deployed with your instance of Textbox.io. This overwrites the default Textbox.io theme and no further action is required before deploying Textbox.io.

It's recommended that the theme be deployed as part of a complete Textbox.io deployment to ensure that the compiled theme matches the deployed version of Textbox.io.

Deploying and Maintaining Themes

 Compiled themes should only be used with the same version of Textbox.io as the theme was developed and compiled against.

When updating to a new build of Textbox.io (i.e. a new version or a patch release) you will need to recompile your theme and deploy it as part of your Textbox.io update.

Developers should retain a backup of their *theme.css* file so that they can easily recompile their theme.

Using Your Own Document Styles

As a developer, Textbox.io enables you to take fine-grained control over the way content is presented, as well as the styles that users are able to select from the styles dropdown menu.

Injecting your own CSS files

In [Classic Editing Mode](#), the textbox.io editor document is contained within an `iframe` element. This means that any styles or stylesheets declared on your host page **are not** inherited by the editor. These need to be explicitly declared as references in the editor configuration.

CSS and styles are configured per-instance as part of the Textbox.io [configuration object](#)

You can declare your own stylesheets using the [stylesheets](#) property or inject CSS directly using the [documentStyles](#) property:

```
var configCSS = {
  css : {
    stylesheets : ['http://www.example.com/mycss.css', 'anotherfile.css'], // an array of CSS file URLs
    documentStyles : 'body { background:red; }' // a string of CSS
  }
};
```

Special cases: Table cells & List Items

The list of items shown on the styles drop-down menu are dependent on the current editor selection. Custom block styles declared for table cells or list items will only become visible when the editor selection exists on those elements as these styles are only valid CSS within a table or list structure. In this way [Textbox.io](#) prevents users from creating invalid HTML.



Custom styles for table cells will only appear when the selection is on a table

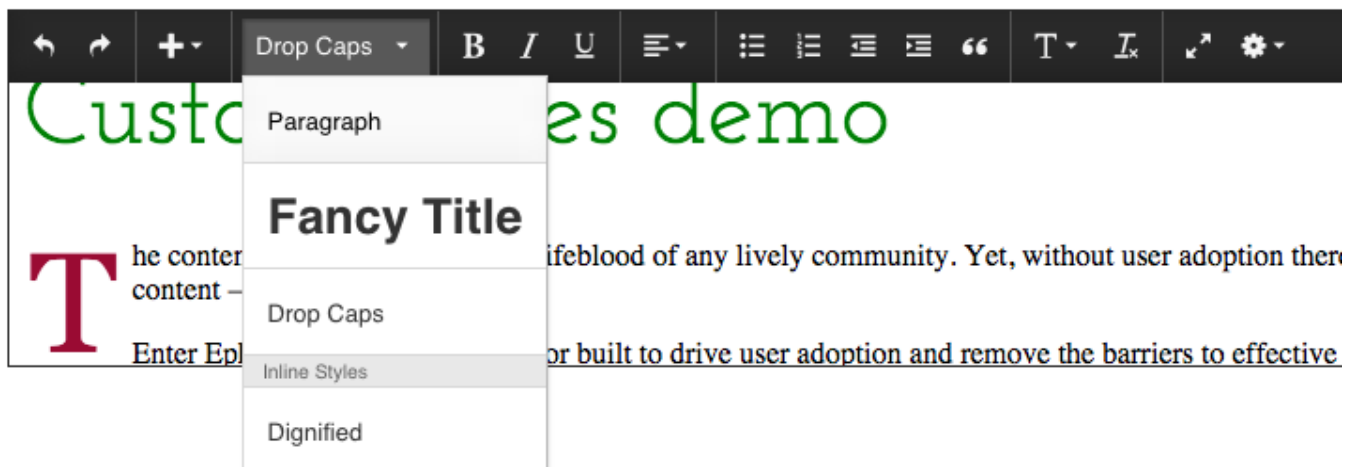
Example: custom CSS

The example below shows the editor using a highly customized CSS file:

Customizing the styles drop-down menu

The items shown in Textbox.io's styles drop-down menu can be customized to suit the requirements of your application. There configuration approaches available are:

- A fixed list of custom styles defined per-editor using the [the editor CSS configuration object](#).
- The editor inspects configured content CSS to optionally show additional styles on the menu (*new in version 2.4.1*)



The Textbox.io styles drop-down menu

Overview: Block vs. Inline styles

Before attempting to create your own custom styles, it is worthwhile understanding the difference between *block style transformations*, and *inline transformations*.

- When a **block transformation** is applied to a selection, the entire block encasing the selection (such as a paragraph or heading) is transformed:

The quick brown fox jumped over the lazy dog.



The quick brown fox jumped over the lazy dog.

- When an **inline transformation** is applied, to a selection, only the selection itself (and no surrounding content) is transformed:

The quick brown fox jumped over the lazy dog.



The quick brown fox jumped over the lazy dog.

In its stock configuration, the drop-down styles menu contains **only block style transformations** - inline style changes can be made via the font drop-down menu.

A [detailed explanation](#) of transformation behaviour is included with the [styles configuration object documentation](#).

Defining Styles in the configuration object

The styles drop-down can be specified in the editor styles configuration object.

The following is an example of custom styles:

- "fancy title" is an example of a *block style*. Similarly, if the rules had defined `h2.title`, `h3.title`, or `p.title`, this transformation would take place at a block level.
- "dignified" is an example of an *inline style*, where the prefixing element has been removed from the definition.

Note the differences in the presentation of these styles in the drop-down menu declaration compared to the [CSS declaration file](#).

Defining styles in the content CSS

This feature was introduced in Textbox.io release 2.4.1

The editor inspects all content styles and can be configured to add style entries automatically. This is useful in scenarios where the editor configuration is fixed across an entire CMS platform but the content stylesheet is set by a template. With this configuration, the styles dropdown can change with the template rather than requiring configuration changes.

Showing or hiding styles

When specifying CSS for use in an instance of the editor, you may not necessarily want every style to appear in the styles drop-down combo box. Using specific CSS attributes you can hide or display specified CSS elements in the styles drop-down.

This process is controlled by the `showDocumentStyles` configuration option, disabled by default, which allows for detailed control over the process:

- When disabled, only styles explicitly set to visible are shown. This can be useful when a complete stylesheet is used in the editor content and showing them all would result in a confusing drop-down.
- When enabled, all styles are shown unless explicitly set to hidden. This is useful when a dedicated stylesheet is created for the editor content.

Custom styles are subject to the same restrictions as rules defined in the [configuration object](#). Any rules that are not valid under those restrictions are ignored regardless of their visibility attribute.

CSS definition attributes

In order to leverage the document stylesheet to achieve this feature, the attributes used must be completely valid CSS. To this end, the editor looks for rules that are targeted at an element attribute (`ephox-data`) which the editor will never set and therefore the rule should never activate in normal use. Using the `visibility` and `content` properties, both whether a style is shown and the text shown in the menu can be controlled.

```
/* never shown in the drop-down, regardless of configuration */
h1.blue {
  color: blue
}
h1.blue[ephox-data] {
  visibility: hidden;
}

/* always shown in the drop-down, regardless of configuration */
h1.red {
  color: red
}
h1.red[ephox-data] {
  visibility: visible;
  content: "red heading";
}

/* Only shown in the drop-down when showDocumentStyles is true */
h1.green {
  color: green;
}
h1.green[ephox-data] {
  content: "green heading";
}
```

Content style entries defined in this way are added *below* any rules defined in the configuration object, which allows either a mixture of fixed and dynamic styles or (if configured with an empty list) a completely dynamic style drop-down.

The ability of the editor to inspect the content CSS is dependent on CORS. If an external content stylesheet is on a different domain to the editor page, the [CSS](#) request must return CORS headers otherwise the browser blocks scripts from inspecting the stylesheet.

Custom style examples

showDocumentStyles disabled

This example replicates the configuration object example above, but adds rules to the style drop-down using the content CSS rather than using the configuration object. In this case `showDocumentStyles` is false, so only the style explicitly configured is added.

showDocumentStyles enabled

This example inverts the `showDocumentStyles` config from above. In this mode, all styles are shown on the menu except the one that is hidden through a CSS property. The configuration object is an empty array indicating only document styles should be shown.

Filtering Content

Filter overview

Filters are used by the editor to manipulate content as it is entered (via `editor.content.set()` or a paste event), or when content is requested from the editor (via `editor.content.get()`).

Such filters are used internally by the editor to perform meaningful functionality such as:

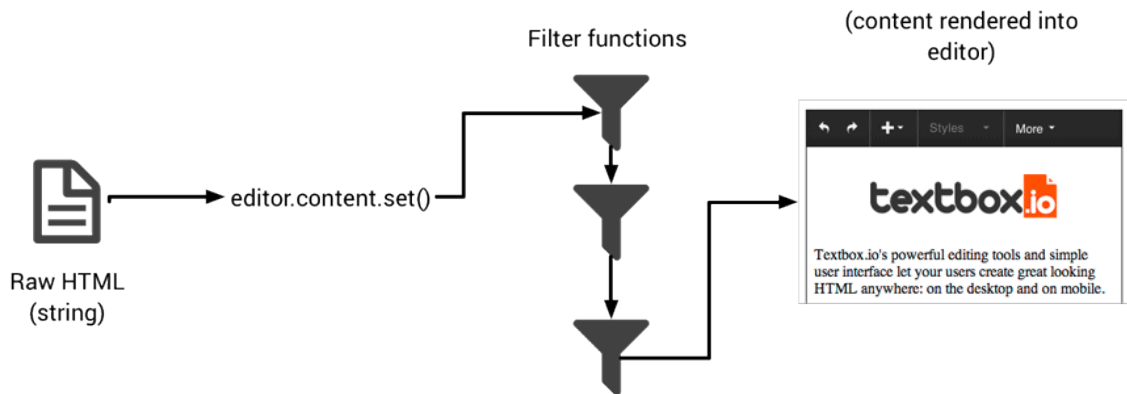
- Cleaning dirty HTML from Microsoft® Word when it is pasted
- Stripping unnecessary or undesirable CSS from input HTML

Filters come in 2 types:

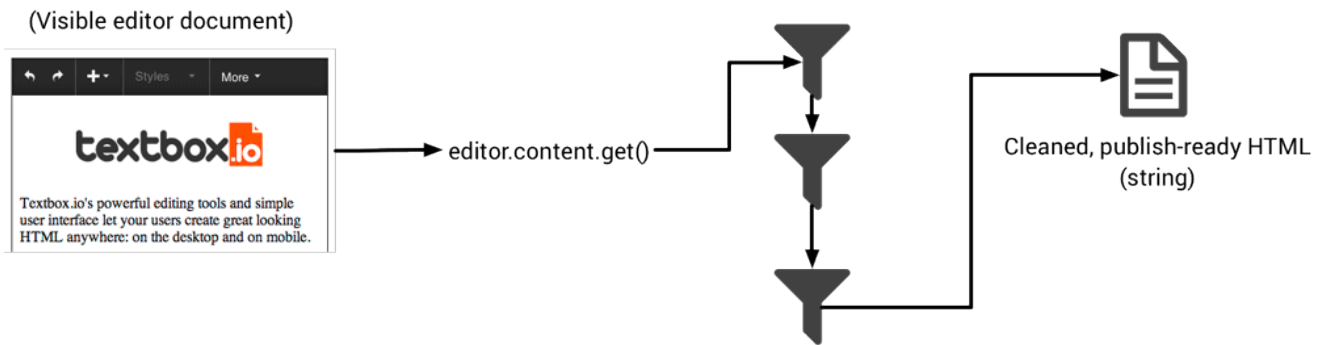
1. **Input filters** - manipulate raw content before it becomes visible in the editor. Such uses for an input filter may be to change or strip styling from input, or convert invisible elements into elements that are visible
2. **Output filters** - executed before the editor returns content to the API (i.e. before `editor.content.get()` returns a string). Output filters can be used to strip-out internal content created during the editing process (such as invisible marker elements, or other elements used for editing and annotation, but that are not necessarily desired in the final output)

Filters are added to editor instances at runtime.

Input filters:



Output filters:



Input filters vs. Output filters

Implementing filters: Predicate vs. Selector filters

The basic premise behind implementing a filter is to *find* elements in the source document that you wish to manipulate in some way. You can do this 2 ways in the Textbox.io API:

- **Using a selector** - Where your filter function is executed on all elements on the document matching the selector pattern you provide.
- **Using a predicate function**- Where you provide a *match predicate function* to determine which elements your filter function should be executed on. Your *predicate function* is passed every element in the source document. It should return `true` if you require the element to be processed by the filter function.

Example: Creating a selector input filter

In this example, we'll create an input filter that adds some custom CSS to every `H1` element in the source document. Note how we use a *selector string* as the first argument.

Example: Creating a predicate input filter

Here's an example the same input filter, but rather than using a selector, a predicate function is to find the `H1` elements:

Example: Output filter

In the example below, a filter has been implemented to strip any inline styling from `p` tags before `editor.content.get()` returns a value:

Macros: Writing Content-Aware Code

What is a macro?

Textbox.io provides a [Macro API](#) enabling you to create functions that detect particular content in the editor as a user types, and perform meaningful actions to manipulate the content. Specifically macros are evaluated when the user presses the space or the enter keys.

Currently, Textbox provides the ability to add a "Simple" macro, using:

```
editor.macros.addSimpleMacro('startString', 'endString', callback)
```

where any text entered before the spacebar is pressed that begins with `startString` and ends with `endString` forms a pattern. When a match is found, `callback` is applied with the match as an argument.

`callback` can return a replacement (if one is necessary), or the original that was matched.

Built-in macros

A collection of macros are enabled by default. The list of macros, and how to control their availability, is documented in the [macros](#) configuration object.

Example: Red text

In the example provided below, we match on any text that begins with `[red]` and ends with `[/red]`, and wrap the text in a `span` where the color is set to red. For example, any of the following would be matches:

- `[red]hello world[/red]`
- `[red]testing[/red]`
- `[red] She sells sea shells by the sea shore[/red]`

See Also

[macros](#) configuration object

[editor.macros](#) runtime API

Creating Dynamic Content

The combination of filters and macros in the editor can create a powerful environment for creating dynamic page content in the editor.

Dynamic content is where a *script*, *templating engine*, or *process* evaluates a variable or declared function contained within a HTML document.

An example of such a dynamic page fragment might be:

```
<html>
<body>
  <p> The date today is: [[date]] </p>
</body>
</html>
```

Where `[[date]]` is evaluated at render-time by some outside process, for example.

Often, it is difficult to portray such dynamic content in a WYSIWYG editing environment, without taking away the dynamic behavior itself.

Using macros, you can replace content in real-time, whilst preserving original state by wrapping original values in `data-` attributes.

The demo below shows an example where we can show true WYSIWYG output of dynamic functions, such as `[[date]]`, but use the original function when `editor.content.get()` is called:

Editor types - Classic vs Inline

Textbox.io is able to edit content in 2 distinct ways - "Classic" mode, where the editor appears in a self-contained box, and "Inline mode" (also known as *in-context mode*) where the editor literally takes an element and makes it editable. In inline mode, you edit content exactly as it appears within the context of the page it is being hosted upon.

Inline vs classic mode - a quick comparison

Inline	Classic
<ul style="list-style-type: none">• Inherits stylesheets from the <i>host page</i> the editor is invoked upon• Provides a 'true' WYSIWYG editing experience within the context of the host page• Doesn't work on mobile (...yet...)• Does not provide a fullscreen editing mode or source code view• Container grows as the content inside grows• Toolbar grows and can be dragged around the page	<ul style="list-style-type: none">• Is self contained (within an iframe)• Requires injection of outside CSS styles using the configuration API• Works on mobile devices• Editor toolbar is always visible (regardless of whether the editor is focused)• Allows editing in fullscreen mode• Editor container remains a fixed size

Inline editing does not currently work on Mobile devices. See: [Developing editor experiences optimized for mixed platforms](#)

Demo - Classic Editing

Demo - Inline Editing

Which type of Editor should I choose?

It really depends on the kinds of problems you're looking to solve through the use of an editor.

Classic Mode is useful for generating content that requires isolation- such as form fields, or email compose fields.

Inline Mode is useful for editing content such as real-time WYSIWYG editing of complete web pages

Inline-specific configuration options

Toolbar visibility

The toolbar can be made invisible at initialization via the configuration object. Such functionality is useful when developing visually lightweight "minimal" applications.

Toolbar draggability

By default, the user can drag the toolbar away from it's container object as if it were a floating window. This functionality may be disabled.

Toolbar offset

An offset may be set on the toolbar to provide visual separation between the toolbar and the content being edited.

Note: the editor assumes a "pinned" behavior when it is attached to the content (i.e. when it has no offset). This gives it special properties (such as attaching it to the top of the viewport) when large blocks of content are scrolled. This functionality is no-longer executed when the toolbar has an offset from the container window.

Features disabled for inline editing

The following features are disabled when inline editing is used (regardless of whether they are declared in configuration):

- Source code view
- Fullscreen editing

See also

- [API Basics](#) - Learn the basics of using the editor API, such as Getting and setting content in the editor.
- [Toolbar configuration](#) object
- [Customizing the Editor](#) - Learn how to configure and customize Textbox.io [for your applications](#).
- [Spell Checking](#) - Find out more about how to enhance the editing experience with spell checking, image upload and other web services from Tiny.
- [API reference](#)

Working with Images

Textbox.io offers rich and comprehensive functionality to work with images in content.

- Image editing - crop, resize/resample, rotate and flip
- Automatic image resizing
- Insert local images
- Insert images from the web
- Broad copy and paste support
 - From the file system
 - From Microsoft Word
 - From image editing applications
- Drag and drop insertion

Image Editing

Image editing in Textbox.io enables users to crop, resize/resample, rotate and flip images within the editor.

Editing an image creates a new copy of the image which can be retained in the content as a data URL, or uploaded when the content is submitted or by calling the [uploadImages](#) function.

Editing Web Images

Editing images from the same domain as the editing page is supported without any changes.

Editing images from a server other than the current domain requires the deployment of the [Textbox.io Server-side Components](#), specifically the Extended Image Editing proxy. This is due to the cross origin resource sharing (CORS) requirements of modern browsers. The Extended Image Editing proxy service ensures that images from a remote server can be edited within Textbox.io.

A local copy of the image is created for editing and the edited image is then retained as a data URI in the content or uploaded to a server based on your configuration.

Limiting Image Size

The size of inserted local images can be restricted when image editing is turned on. This enables developers to ensure image file sizes are more appropriate for their site.

Setting the `preferredHeight` and `preferredWidth` configuration options in the [image configuration](#) ensures that local images will be resized and resampled to ensure that they do not exceed the dimensions specified.

Working with Local Images and Edited Images

To enable end users and developers to work with content that contains local images (i.e. images from the end-user's machine) Textbox.io includes several options on how to store and manage images.

Developers can choose to turn off all functionality that relies on local images, store the images at data URIs within the content or upload images to a server using the uploading capabilities of Textbox.io.

For more information see the article on [Handling Local Images](#).

Uploading Local Images

Textbox.io's image handling functionality includes methods to upload asynchronously via HTTP.

When [image editing is turned on](#) images are **not** uploaded while the document is being edited. This ensures that extraneous, partially edited images are not uploaded while editing is taking place.

When [image editing is turned off](#) local images are uploaded in the background as soon as they have been added to the document.

The article on [Handling Local Images](#) explains more about this process and how to handle asynchronous image uploads when saving content.


Handling Local Images

Textbox.io gives you the ability to handle local images in one of several ways within your application. You may either upload local images from the client to your application, store images directly in the editor generated HTML itself (using base64 data URIs), or have the Textbox.io editor prevent local images from being inserted.

With all but the last option (prevent local images) the user experience is the same: users can add images to [Textbox.io](#) instances via the image upload dialog, by dragging and dropping images from their computer, or via copy-paste.

Image Handling Option	Description
Keep image data in HTML content	[Default] Local images are stored within the editor's HTML content as base64 encoded data URIs.
Upload images	Local images are uploaded to a remote server when added to the editor via HTTP POST. Textbox.io automatically updates the <code><image> src</code> attribute with the new path to the uploaded image once the upload has been completed. See the information on uploading local images below to learn how to configure Textbox.io to do this.
Prevent local image insertion	Local image functionality is turned off - users can no longer use the local image upload dialog tab, drag-drop, or copy paste to add images to editor content. See the information on Preventing Local Image Insertion below to learn how to configure Textbox.io to do this.

Local vs. Remote Images

 Local images are defined as those residing on the client filesystem. They may also be part of a word processor document or otherwise present on the clipboard. Textbox.io can be configured to upload local images to your application or embed them in editor HTML.

Remote images are those which exist on a remote host and are accessible via a URL.

Storing Local Images in Content (base64 data URIs)

[Textbox.io](#) will by default store local images added to an editor as embedded base64 [data URIs](#).

If this is the desired editor behavior in your application, no further action is necessary. When a user adds an image to a [Textbox.io](#) editor within your application [Textbox.io](#) will automatically embed that image into the HTML content.

Uploading Local Images

Configuring your application and Textbox.io for local image uploads involves first creating a server-side handler and then configuring your Textbox.io instance to use that handler.

Image Upload on Form Submission

 If you have enabled the image upload functionality of Textbox.io it is **strongly** recommended that you review the information on [Handling Asynchronous Image Uploads](#).

Textbox.io uploads images asynchronously to ensure the author's flow isn't interrupted by multiple image upload dialogs/prompts. However, developers need to be mindful of this when integrating Textbox.io so as to ensure all images are uploaded prior to content being submitted to a server.

Server-side Upload Handler

In order to upload local files to the remote server via HTTP POST, you will need a server-side upload handler script that accepts the images and objects on the server and stores them in the correct directory or database. This script is the same script that would be used for uploading any file to the server via the HTTP POST method.

For example, when you use a file input element (`<INPUT type="file">`), the script specified in the `action` attribute of the parent `<form>` element is used to upload the file to the server.

The server-side upload handler script should return a JSON object similar to the one below. The returned JSON should include a single `location` attribute with the path to the stored image as its value.

```
{ "location" : "/uploaded/image/path/image.png" }
```

Note, that the '/' here in the `location` field is used to suggest a `root-relative` path. If you don't provide the leading '/', then the path will be `relative`. If you provide a protocol (e.g. `http`), then the path will be `absolute`. The table below shows how the full image path will be resolved against the various types of image locations:

Base Path	Image Location	Path Type	Full Image Path
<code>http://server-name/base/</code>	<code>/uploaded/image.png</code>	root-relative	<code>http://server-name/uploaded/image.png</code>
<code>http://server-name/base/</code>	<code>uploaded/image.png</code>	relative	<code>http://server-name/base/uploaded/image.png</code>
<code>http://server-name/base/</code>	<code>http://elsewhere/image.png</code>	absolute	<code>http://elsewhere/image.png</code>

Example Upload Handler Scripts

The following scripts are reference implementations for handling server-side uploads with Textbox.io. Please note that these scripts are provided only for reference - they are not intended for production use.

- [Node.js Upload Handler](#)
- [PHP Upload Handler](#)

Your upload handler script should:

- Store the image in a location appropriate for your application
- Success: Return JSON with the path to the uploaded image
- Failure: Return HTTP 500 if an error occurs

Configuring Textbox.io to Use a Server-side Upload Handler

Once you've set up a server side upload handler, all that's left is to make Textbox.io aware of the handler's location via a `configuration` object and set up the path used to construct the `<image>` `src` attribute.

In the example below, any local image added to a the Textbox.io editor is uploaded to the handler script located at <http://example.com/postAcceptor.php>. Textbox.io then constructs the path to the newly uploaded image by combining the (optional) `basePath` value and the image filename. The resulting editor html is then updated with the new path to the image, generating HTML like: ``.

```
var config = {
  images : {
    upload : {
      url : '/postAcceptor.php',           // Handler URL
      basePath: '/my/application/images/', // Remote image storage path
      credentials: false                  //
    }
  }
  Optional: sends cookies with the request when true
};

var editor = textboxio.replace('#targetId', config);
```

For more detail on see the [images configuration property](#).

CORS Considerations

You may choose for your web application to upload image data to a separate domain. If so, you will need to configure [Cross-origin resource sharing \(CORS\)](#) for your application to comply with JavaScript "same origin" restrictions.

CORS has very strict rules about what constitutes a cross-origin request. The browser can require CORS headers when uploading to the same server the editor is hosted on, for example:

- A different port on the same domain name
- Using the host IP address instead of the domain name
- Swapping between HTTP and HTTPS for the page and the upload script

The upload script URL origin must *exactly* match the origin of the URL in the address bar, or the browser will require CORS headers to access it. A good way to guarantee this is to use a relative URL to specify the script address, instead of an absolute one.

All supported browsers will print a message to the JavaScript console if there is a CORS error, and Textbox.io will display an error banner.

The [Reference Upload Handler Scripts](#) provided here configure CORS on a per script basis. You may also choose to configure CORS at the [web application layer](#) or the [HTTP server layer](#).

Further Reading on CORS

- [W3C Wiki - CORS Enabled](#)

- [MDN - HTTP access control \(CORS\)](#)
- [W3C - Cross-Origin Resource Sharing Specification](#)

Preventing Local Image Insertion

If you wish to prevent the insertion of local images by users you may do so by setting the `images.allowLocal` to `false`.

When `images.allowLocal` is set to `false`, users will be unable to add local images to editor content from the local image upload dialog tab, by drag-drop, or by copy paste. If a user takes an action that would normally result in the insertion of an image, a notification will be displayed that insertion of images is not allowed.

The example below creates an editor where local images have been prevented.

```
var config = {
  images : {
    allowLocal : false      // Prevent users from adding local images
  }
};

var editor = textbio.replace('#targetId', config);
```

Handling Asynchronous Image Uploads

Textbox.io automatic image uploads are asynchronous.

It is therefore possible for users to save their content before all images have completed uploading. In this situation, no server path to the image resource is available and those images will be stored in the HTML as Base64 data URIs.

When replacing a textarea inside a form element the editor will automatically enable `autosubmit`. This delays the standard form submit event to wait for images to upload. In all other scenarios, or if `autosubmit` is disabled, this delay must be introduced manually.

To ensure that all images have been uploaded developers can use the `editor.content.uploadImages()` method to ensure that all image uploads have completed and that the HTML content of the editor has been updated appropriately. `uploadImages()` accepts a callback function, triggered when images have finished uploading and the content is ready.

The example below demonstrates how to use the `uploadImages()` method in conjunction with the `content.get()` method to ensure that all images have been uploaded **before** the content is retrieved from the editor. It could also be used with a form's `onsubmit` method in a similar way.

uploadImages and Content Retrieval Callback


```
editor.content.uploadImages(function() {  
    var content = editor.content.get();  
    console.log(content);  
});
```

`uploadImages()` scans the editor content for all Base64 encoded images and queues them for upload. If no images are uploading, the callback is executed immediately.

Node.js Upload Handler

The following script is a simple Node.js application that creates an HTTP server, and handles image uploads (multipart form posts) suitable for use with Textbox.io. Please note that this script is provided for your reference only.

File	Modified
JavaScript File postAcceptor.js	Dec 10, 2015 by Michael Fromin

Drag and drop to upload or [browse for files](#) 

Node.js Post Acceptor

```

var http = require('http');
var fs = require('fs');
var multipart = require('multipart'); // https://www.npmjs.com/package/multipart

// #####
// Setup the Textbox.io Example Post Acceptor
// #####
// Set the URL and port for the post acceptor
var acceptorPath = '/upload';
var acceptorPort = 8080;
// Only these origins will be allowed to upload images
var allowedOrigins = ['http://localhost', 'http://192.168.1.1'];
// Set the upload directory
// Notw: This string is prepended to the filename to construct
// the image url returned to the Textbox.io editor, like: uploadDirectory/filename.png
var uploadDirectory = 'images/';
// Set the resource URL for from which stored images will be served.
var resourceUrl = 'http://localhost/';
// #####
// #####

http.createServer(function(req, res) {
  var origin = req.headers.origin;
  if(allowedOrigins.indexOf(origin) == -1) {
    // Deny invalid origins
    res.writeHead(403, { 'HTTP/1.0 403 Origin Denied' : origin });
    res.end();
  } else {
    // For valid origins check path and method
    if (req.url === acceptorPath && req.method === 'POST') {
      var form = new multipart.Form();
      form.parse(req);
      form.on('file', function(name, file) {
        var saveFilePath = uploadDirectory + file.originalFilename;
        fs.rename(file.path, saveFilePath, function(err) {
          if (err) {
            // Handle problems with file saving
            res.writeHead(500);
            res.end();
          } else {
            // Respond to the successful upload with JSON.
            // Use a location key to specify the path to the saved image resource.
            // { location : '/your/uploaded/image/file' }
            var textboxResponse = JSON.stringify({
              location : saveFilePath
            });

            // If your script needs to receive cookies, set images.upload.credentials:true in
            // the Textbox.io configuration and enable the following two headers.
            // res.setHeader('Access-Control-Allow-Credentials', 'true');
            // res.setHeader('P3P', 'CP="There is no P3P policy."');
            res.statusCode = 200;
            res.setHeader('Access-Control-Allow-Origin', origin);
            res.end(textboxResponse);
          }
        });
      });
      form.on('error', function(err) {
        res.writeHead(500);
        res.end();
      });
      return;
    } else {
      // Return 404 for requests for other paths/methods
      res.writeHead(404);
      res.end();
    }
  }
}).listen(acceptorPort);


```


PHP Upload Handler

The following script creates a server-side upload handler in PHP suitable for use with Textbox.io.

Please note that this script is provided for your reference - you'll need to update this as necessary for your application.

File	Modified
File postAcceptor.php	Dec 10, 2015 by Michael Fromin

Drag and drop to upload or [browse for files](#) 

postAcceptor.php

```
<?php
/*****
 * Only these origins will be allowed to upload images *
 *****/
$accepted_origins = array("http://localhost", "http://192.168.1.1", "http://example.com");

/*****
 * Change this line to set the upload folder *
 *****/
$imageFolder = "images/";

reset ($_FILES);
$temp = current($_FILES);
if (is_uploaded_file($temp['tmp_name'])){
    if (isset($_SERVER['HTTP_ORIGIN'])) {
        // same-origin requests won't set an origin. If the origin is set, it must be valid.
        if (in_array($_SERVER['HTTP_ORIGIN'], $accepted_origins)) {
            header('Access-Control-Allow-Origin: ' . $_SERVER['HTTP_ORIGIN']);
        } else {
            header("HTTP/1.0 403 Origin Denied");
            return;
        }
    }

    /*
     * If your script needs to receive cookies, set images.upload.credentials:true in
     * the Textbox.io configuration and enable the following two headers.
     */
    // header('Access-Control-Allow-Credentials: true');
    // header('P3P: CP="There is no P3P policy."');
    // Sanitize input
    if (preg_match("/([^\w\s\d\-\~,\;:\[\]\(\)\.])|([\.]{2,})/", $temp['name'])) {
        header("HTTP/1.0 500 Invalid file name.");
        return;
    }

    // Verify extension
    if (!in_array(strtolower(pathinfo($temp['name'], PATHINFO_EXTENSION)), array("gif", "jpg", "png"))) {
        header("HTTP/1.0 500 Invalid extension.");
        return;
    }

    // Accept upload if there was no origin, or if it is an accepted origin
    $filetowrite = $imageFolder . $temp['name'];
    move_uploaded_file($temp['tmp_name'], $filetowrite);

    // Respond to the successful upload with JSON.
    // Use a location key to specify the path to the saved image resource.
    // { location : '/your/uploaded/image/file' }
    echo json_encode(array('location' => $filetowrite));
} else {
    // Notify Textbox.io editor that the upload failed
    header("HTTP/1.0 500 Server Error");
}
?>
```


Checking Spelling

Spell Checking in Textbox.io

Textbox.io includes optional server-side spell checking and autocorrect as you type for a number of common languages. Textbox.io detects the language of the editor document (or the surrounding page when using [inline editing](#)), and checks the spelling of words against the remote dictionary.

Spell Checking Languages Supported

Languages	Document Language Code
English	en, en_US
English (UK)	en_UK, en_GB, en_BR
Danish	da
Dutch	nl
Finnish	fi
French	fr
German	de
Italian	it
Norwegian	nb
Portuguese (Brazil)	pt
Portuguese (Europe)	pt_PT
Spanish	es
Swedish	sv

Getting Started Using Spell Checking

Follow the [installing the server-side spelling component](#) article for more detail and to get started.

Note: The [Textbox.io Spelling & server-side components](#) are compatible with Java Application Servers.

Note



If you have already installed the [Textbox.io](#) server-side components you'll just need to use the [spelling Textbox.io](#) Client Editor API to reference the URL locations of your installed services.

API reference

The Textbox.io Editor JavaScript API starts at the `textboxio` JavaScript global, all Textbox.io code runs via this object. The `textboxio` global is available immediately after `textboxio.js` has finished loading.

See the [Getting Started](#) guide for details on loading `textboxio.js`.

- [configuration](#)
 - [autosubmit](#)
 - [basePath](#)
 - [codeview](#)
 - [css](#)
 - [images](#)
 - [links](#)
 - [macros](#)
 - [paste](#)
 - [spelling](#)
 - [ui](#)
- [editor](#)
 - [editor.content](#)
 - [editor.element](#)
 - [editor.events](#)
 - [editor.filters](#)
 - [editor.focus](#)
 - [editor.macros](#)
 - [editor.message](#)
 - [editor.mode](#)
 - [editor.restore](#)
- [textboxio](#)
 - [get](#)
 - [getActiveEditor](#)
 - [inline](#)
 - [inlineAll](#)
 - [isSupported](#)
 - [replace](#)
 - [replaceAll](#)
 - [version](#)

configuration

Overview

Textbox.io `editor` instances are configured through a configuration object, passed as a second argument when creating an editor via `replace`, `replaceAll`, `inline`, or `inlineAll`. Defining properties of a `configuration` object will override the configuration defaults supplied for any generated instances of `editor`.

Configuration Object Properties

Note that all properties are optional. Defining a property overrides the editor default behavior.

<code>autosubmit</code>	Boolean	Specifies whether textboxio should handle form submission
<code>basePath</code>	String	Specifies the path to the textboxio resources folder
<code>css</code>	Object	Content CSS and styling application
<code>codeview</code>	Object	Code view feature
<code>images</code>	Object	Image handling & upload
<code>links</code>	Object	Content link validation
<code>macros</code>	Object	Built-in macro configuration
<code>paste</code>	Object	Content paste behavior
<code>spelling</code>	Object	Content spell checking service
<code>ui</code>	Object	Editor UI including toolbars, menus, etc.


Configuration Defaults

This following object will replicate the default configuration for all Textbox.io instances.

Implicit Configuration Defaults

```
var defaultConfig = {
  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']
  macros : {
    allowed : [ 'headings', 'lists', 'semantics', 'entities', 'hr', 'link' ]
  }
  ui : {
    toolbar : {
      items : [ 'undo', 'insert', 'style', 'emphasis', 'align', 'listindent', 'format',
'tools' ]
    }
  }
};
```

Internationalization

 Note that `text` properties are pre-configured with values like `block.p`. These string keys refer to the internationalized string label for that item. Using a pre-configured value means that the text value will be internationalized by `locale`.

autosubmit

```
configuration
  autosubmit
```

This option configures whether `textboxio` will automatically handle form submission for text areas. Its value must be either `true` or `false`.

Some frameworks will replace normal form submissions with AJAX requests. These frameworks may be incompatible with `textboxio`'s `autosubmit` functionality. In this scenario it is recommended that you disable `autosubmit`.

By default `autosubmit` is set to `true`. When set to `true` `textboxio` intercepts form submission to place the contents of the editor back into its original `textarea` before the form submits. This process is asynchronous as `textboxio` will wait for images to upload if required.

When `textboxio` is **not** handling form submission (i.e. `autosubmit` is set to `false`), the page's form submission will be unaffected. It is then up to the integrator to ensure that images in the content are uploaded. This process is explained in [Handling Asynchronous Image Uploads](#).

Example Configuration

In this example, a `configuration` object turns off the automatic form submission handling of `textboxio`.

```
var config = {
  autosubmit: false
};

var editor = textboxio.replace('#targetId', config);
```

Properties

Property	Type	Default	Description
<code>autosubmit</code>	Boolean	<code>True</code>	<code>True</code> turns on <code>textboxio</code> 's automatic form handling, and <code>false</code> turns it off.

See also

- [Getting Started](#)
- [editor.content.uploadImages\(\)](#)
- [editor.content.get\(\)](#)
- [Handling Asynchronous Image Uploads](#)

basePath

```
configuration
  basePath
```

This option configures the path to the `textboxio` folder in your application. This path on the server must contain the `resources` folder from the unpacked `textboxio.zip`. The editor will attempt to detect this path automatically by searching for a script tag reference to itself, assuming that the `textboxio.zip` file has been unpacked at that location.

The editor will throw a JavaScript error and not load if a `basePath` is not provided and auto detection fails.



Successful auto detection requires a script tag referencing `textboxio.js`. The referenced filename may be different if it is added to the page dynamically or processed by another script / application. When this occurs it is necessary to specify a `basePath` in the editor configuration.

Example Configuration

In this example, a `configuration` object specifies a custom path to the `textboxio` folder as unpacked from `textboxio.zip`.

```
Server contents:
server
  path
    to
      textboxio
        resources
          textboxio.js

var config = {
  basePath : '/path/to/textboxio'
};

var editor = textboxio.replace('#targetId', config);
```

Properties

Property	Type	Properties
<code>basePath</code>	String	A string that defines the file path to the <code>textboxio</code> folder.

See also

- [Getting Started](#)

codeview

```
configuration
  codeview
```

This option configures the behavior of the code view feature and button. Note that it is possible to disable the code view button, but still trigger the code view feature with `editor.mode.set(mode)`.

This configuration attribute is **ignored** on mobile devices and inline editing modes. For these scenarios, the code view has been forcibly disabled.

Example Configuration

In this example, a `configuration` object disables the editor code view button while leaving the feature enabled.

```
var config = {
  codeview : {
    enabled: true,
    showButton: false
  }
};

var editor = textboxio.replace('#targetId', config);
```

Properties

Property	Type	Properties
enabled	Boolean	true to enable codeview for the editor instance. false to disable.
showButton	Boolean	true to show the codeview button for the editor instance. false to disable.

Codeview Default Configuration



The Codeview feature is enabled by default, the default value for the `enabled` and `showButton` properties are `true`.

For further information see [Configuration Defaults](#).

See also

- [HTML Code View](#)

CSS

```
configuration
  css
```

The `configuration.css` property defines options related to editor rendering CSS and CSS classes/elements available in the styles menu.



Example Configuration

In this example, a simple `configuration` object is created that adds a rendering stylesheet to the editor, inline rendering CSS, and configures the styles available to apply.

```
var config = {
  css : {
    documentStyles : 'body { background: red; }',
    styles : [
      { rule : 'p' },
      { rule : 'h1.blue', text: 'Blue Heading 1' },
      { rule : '.green', text: 'Green Inline Style' }
    ],
    stylesheets : ['test.css'],
    showDocumentStyles : true
  }
};

var editor = textboxio.replace('#targetId', config);
```

Properties

Property	Type	Value
<code>documentStyles</code>	String	String of CSS for rendering editor content. Note  This parameter is ignored when using inline editing (i.e. invoking the editor with <code>textboxio.inline()</code>)
<code>showDocumentStyles</code>	Boolean	Whether to show content style rules in the styles drop-down menu by default. Defaults to <code>false</code> . Full details on defining styles in content CSS is covered in the Using Your Own Document Styles article. Note  This parameter is ignored when using inline editing (i.e. invoking the editor with <code>textboxio.inline()</code>) This API was introduced in Textbox.io release 2.4.1
<code>styles</code>	Array	An array of available styles definitions .
<code>stylesheets</code>	Array	An array of string paths to editor rendering stylesheets. Note: this parameter is ignored when using inline editing (i.e. invoking the editor with <code>textboxio.inline()</code>)

documentStyles

```
configuration
  css
    documentStyles
```

Developers can directly use CSS to render content in [classic editing mode](#) with `documentStyles`. This configuration property accepts a string of CSS, which is added to a [classic editing mode](#) editor document immediately following any stylesheets added with `stylesheets`.

The value for this attribute is **ignored** in inline editing mode. In this mode editor content is part of the host page DOM and is thus rendered using CSS from the surrounding page.

Example

In the example below a configuration is created that specifies inline document CSS along with 2 custom stylesheets. Note that CSS added with `documentStyles` is loaded after any stylesheets added with `stylesheets`.

```
var configCSS = {
  css : {
    documentStyles : 'body { background:red; }',
    stylesheets : [
      'http://example.com/path/to/styles.css',
      '../relative/path/to/sheet.css'
    ]
  }
};
```

The following is a representation of the editor document when the configuration above configuration is used.

```
<html>
<head>
  <link rel="stylesheet" type="text/css" href="http://example.com/path/to/styles.css">
  <link rel="stylesheet" type="text/css" href="../relative/path/to/sheet.css">
  <style type="text/css">
    body {
      background: red;
    }
  </style>
</head>
<body>
  [...]
</body>
</html>
```

styles

```
configuration
  css
    styles
      [ style ]
```

Textbox.io enables fine-grained configuration of the styles shown on the styles drop-down menu.

`styles` is an array of `style` definitions. A `style` is an item that appears on the styles drop-down. When clicked, the editor selection is transformed to the rule defined in the style definition.

style definitions

Property	Type	Value
<code>rule</code>	String	The transformation to be made to the selection
<code>text</code>	String	(Optional) The text to appear in the dropdown list item. eg: "My Custom Style"

Example

In the example below a configuration is created that specifies a custom `css styles` array with 4 style definitions.

```
var config = {
  css : {
    styles : [
      { rule : 'p' }, // Change selected elements to paragraph elements
      { rule : 'h1.blue', text: 'Blue heading' }, // Change selected elements to heading1
      { rule : 'a.red' }, // Apply a red class to selected anchor elements
      { rule : '.green' } // Apply green class to selected elements
    ]
  }
};
```

This configuration would result in 4 options added to the block-styles drop-down:

- Paragraph
- Blue Heading
- a.red
- .green

Styling Transformation Rules

The following intended styling actions are available, based on the syntax used when defining `css styles` array items.

Rule Syntax	Transformation
<code>.clazz</code>	Wrap selected text in <code></code>
<code>e1</code>	Convert the block containing the selection to type <code>e1</code>
<code>e1.clazz</code>	Convert the block containing the selection to type <code>e1</code> with css class <code>clazz</code>

Transformations are subject to some limitation, based on styling transformation rules.

Invalid Style Transformation Rules

Styling transformations that are dependent on the CSS cascade, attempt to apply multiple classes, or affect element IDs are not supported.

Unsupported selectors

Rule Syntax	Description
.clazz p	Complex selectors are not supported.
p.clazz.clazz	Cannot add multiple classes to selected elements.
#id	Cannot add IDs to elements.

Unsupported elements

a, embed, hr, img, object, table, tr, ul, span are unsupported blocks for the styles dropdown menu

For more detail on setting available classes/transformations (configuring the block-styles drop-down) see: [Using Your Own Document Styles](#).

Example Transformation Behaviors

Element transformations follow the following behaviors. Note that elements are separated into the following types: block, object, and inline.

Block type elements can be freely transformed into any other type of block element, while object and inline type elements may never be transformed.

When a user selects one of these options, selected elements are then evaluated against the styling transformation rules, if applicable. The results of applying a block-style choice to selected HTML content are as follows.

Selected Content	Block-Style Rule Chosen	Resulting HTML	Description
<p>...</p>	{ rule : 'p' }	<p>...</p>	No change.
<h1>...</h1>	{ rule : 'p' }	<p>...</p>	Heading1 transformed to paragraph.
<td><a>...</td>	{ rule : 'p' }	<td><a>...</td>	Can't transform a table cell or link (object type). No change.
<p>...</p>	{ rule : 'p' }	<p>...</p>	Can't transform span (inline type). No change.
<p><a>...</p>	{ rule : 'p' }	<p><a>...</p>	Can't transform anchor (object type). No change.
<p>...</p>	{ rule : 'h1. blue' }	<h1 class="blue">...</h1>	Paragraph transformed to heading1, blue class applied.
<h1>...</h1>	{ rule : 'h1. blue' }	<h1 class="blue">...</h1>	Blue class applied to existing heading1.
<td><a>...</td>	{ rule : 'h1. blue' }	<td><a>...</td>	Can't transform a table cell or link (object type). No change.
<p>...</p>	{ rule : 'h1. blue' }	<h1 class="blue">...</h1>	Paragraph transformed to heading1, blue class applied. Can't transform span (inline type).
<p><a>...</p>	{ rule : 'h1. blue' }	<h1 class="blue"><a>...</h1>	Paragraph transformed to heading1, blue class applied. Can't transform anchor (object type).
<p>...</p>	{ rule : 'a. red' }	<p>...</p>	Can't transform paragraph (block type). No change.
<h1>...</h1>	{ rule : 'a. red' }	<h1>...</h1>	Can't transform heading1 (block type). No change.
<td><a>...</td>	{ rule : 'a. red' }	<td>...</td>	Can't transform table cell (object type). Red class applied to existing anchor.
<p>...</p>	{ rule : 'a. red' }	<p>...</p>	Can't transform paragraph (block type). Can't transform span (inline type). No change.
<p><a>...</p>	{ rule : 'a. red' }	<p>...</p>	Can't transform paragraph (block type). Red class applied to existing anchor.
<p>...</p>	{ rule : '. green' }	<p>...</p>	Can't transform paragraph (block type). New span created. Green class applied to new span.

<code><h1>...</h1></code>	<code>{ rule : '.green' }</code>	<code><h1>...</h1></code>	Can't transform heading1 (block type). New span created. Green class applied to new span.
<code><td><a>...</td></code>	<code>{ rule : '.green' }</code>	<code><td><a>...</td></code>	Can't transform table cell and anchor (object type). New span created. Green class applied to new span.
<code><p>...</p></code>	<code>{ rule : '.green' }</code>	<code><p>...</p></code>	Can't transform paragraph (block type). Green class applied to existing span.
<code><p><a>...</p></code>	<code>{ rule : '.green' }</code>	<code><p><a>...</p></code>	Can't transform paragraph and anchor. New span created. Green class applied to new span.

See Also:

- [stylesheets](#)
- [documentStyles](#)

stylesheets

```
configuration
  css
    stylesheets
```

Developers can control the CSS stylesheets used to render content in [classic editing mode](#) with the `stylesheets` array. This array should be made up of paths to CSS stylesheets, which are loaded in order from first to last.

The value for this attribute is **ignored** in inline editing mode. In this mode editor content is part of the host page DOM and is thus rendered using CSS from the surrounding page.

Example

In the example below a configuration is created that specifies 2 custom stylesheets. These sheets are then used to render editor content in [classic editing mode](#).

```
var configCSS = {
  css : {
    stylesheets : [
      'http://example.com/path/to/styles.css',
      '../relative/path/to/sheet.css'
    ]
  }
};
```

The following is a representation of the editor document when the configuration above configuration is used.

```
<html>
<head>
  <link rel="stylesheet" type="text/css" href="http://example.com/path/to/styles.css">
  <link rel="stylesheet" type="text/css" href="../relative/path/to/sheet.css">
</head>
<body>
  [...]
</body>
</html>
```

images

```
configuration
  images
```

The `images` property controls how images are handled within a Textbox.io editor instance.

This property controls multiple behaviors:

- Uploading Images via Textbox.io's Built-in Upload Mechanism
- Uploading Images via a Custom JavaScript Upload Handler Function
- Disabling image editing, which turns on automatic image upload
- Preventing the insertion of Local Images

Properties

Property	Type	Default	Properties
<code>allowLocal</code>	Boolean	<code>true</code>	Whether to allow local images to be inserted into the editor.
<code>editing</code>	Object		Configures image editing.
<code>upload</code>	Object		Configures the uploading of local images to your application using Textbox.io's built in upload mechanism.

allowLocal

```
configuration
  images
    allowLocal
```

A developer may choose to prevent local images from being added to an editor. When local images have been blocked, the insert image dialog does not display the Local Files tab. If the user attempts to paste an image into the editor or use drag and drop to insert an image, an error message is displayed.

Example Configuration

In this example, a simple `configuration` object is created that prevents local images from being inserted.

```
var config = {
  images : {
    allowLocal : false
  }
};

var editor = textboxio.replace('#targetId', config);
```

editing

```
configuration
  images
    editing
```

Configures image editing in Textbox.io.

This option was introduced in Textbox.io release 2.0.0.

Property	Type	Default	Properties
enabled	Boolean	true	Enables or disables the image editing feature. When image editing is enabled inserted images are not uploaded until the <code>editor.content.uploadImages()</code> method is called. This is to ensure that editing images does not cause multiple uploads. Disable image editing to have inserted images upload immediately.
proxy	String		Allows editing images from a remote server using an Image Proxy. Specify the URL to your proxy here. The Textbox.io SDK comes with an Image Proxy service (for J2EE environments). For information on installing and configuring this service, please see the Server-Side Components article.
preferredWidth	Number		When specified, any large local images that are inserted will be scaled down to fit within this width.
preferredHeight	Number		When specified, any large local images that are inserted will be scaled down to fit within this height.

Examples

In this example, a simple `configuration` object is created that turns off image editing.

```
var config = {
  images : {
    editing : {
      enabled : false
    }
  }
};

var editor = textboxio.replace('#targetId', config);
```

Proxy

In this example, a proxy is set to allow editing images from remote servers.

```
var config = {
  images : {
    editing : {
      proxy : "http://YOUR-DOMAIN/ephox-image-proxy/image"
    }
  }
};

var editor = textboxio.replace('#targetId', config);
```

Restricting image size

When image editing is enabled local images above a certain resolution can be automatically scaled down to a preferred size. Aspect ratio is maintained; in the following example the editor is configured with both preferred width and height at 1000, which means:

- A wide image at 2000x1000 will be resized to 1000x500

- A tall image at 1000x2000 will be resized to 500x1000

```
var config = {
  images : {
    editing : {
      preferredWidth : 1000,
      preferredHeight : 1000
    }
  }
};

var editor = textboxio.replace('#targetId', config);
```

upload

```
configuration
  images
    upload
```

Configures the uploading of local images to your application using Textbox.io's built in upload mechanism.

This property can be configured to either:

- Upload Images using a form POST
- Upload Images via a Custom JavaScript Upload Handler Function

Properties

To use Textbox.io's built-in upload mechanism, use `upload.url`, `upload.basePath`, and `upload.credentials`.

Property	Type	Default	Properties
<code>url</code>	String		Defines the location of your POST acceptor script. This is the URL to which images will be uploaded.
<code>basePath</code>	String		[Optional] Defines the base path of images uploaded in your application. This path is combined with the path returned from your POST acceptor script.
<code>credentials</code>	Boolean	<code>false</code>	[Optional] <code>true</code> : sends cookies with the upload POST (sets the XHR credentials flag). <code>false</code> : does not send cookies.

To use a custom JavaScript handler function, use `upload.handler`. Note that defining `upload.handler` disables Textbox.io's built in upload mechanism.

Property	Type	Default	Properties
<code>handler</code>	Function		Configures the uploading of images to your application using a custom Javascript upload handler.

Examples

Using Textbox.io's Built-in Upload Mechanism

The simplest way to handle images is use automatic background uploads. A simple `configuration` object is used to define the upload location for images. `basePath` is used in conjunction with the value returned by the POST handler to link to the uploaded image within the editor's content. More information is available in the [Handling Local Images](#) guide.

```
var config = {
  images : {
    upload : {
      url : 'http://example.com/path/to/POSTAcceptor.php',
      basePath : 'http://example.com/path/to/images'
    }
  }
};

var editor = textboxio.replace('#targetId', config);
```

Using a Custom JavaScript Upload Handling Function

If images must be uploaded using more than a simple HTTP POST, the entire upload process can be replaced. For example, you may wish to perform additional client side validation or manipulation on the images prior to their upload. The handler function is passed three arguments; Data (object), Success (function) and Failure (function).

Data Object Properties

The first argument is an object providing information about the image that has been inserted. All properties are functions.

Property Function	Type	Description
<code>data.blob()</code>	Blob	JavaScript Blob instance representing the image binary data
<code>data.base64()</code>	String	A copy of the image binary data pre-converted to base64 for your convenience
<code>data.filename()</code>	String	The actual filename, where possible, otherwise a generated filename based on the MIME type
<code>data.id()</code>	String	A unique identifier for this image

Success Function Parameters

The second argument is a function that indicates the upload has successfully completed

<code>url</code>	String	The URL to use as the image src. No post-processing is performed on this value (the <code>basePath</code> configuration property is ignored).
------------------	--------	---

Failure Function Parameters

The third argument is a function that indicates the upload failed.

<code>message</code>	String (optional)	A message to display to the user in an error banner. This value is optional - if you do not wish to translate your error a generic message is available which has been translated into all supported languages.
----------------------	-------------------	--

```

var config = {
  images : {
    upload : {
      handler : function (data, success, failure) {
        // For example, if myuploader.upload() returns a promise, e.g. jQuery ajax
        myuploader.upload(data.blob(), data.filename()).then(function (<upload
response>) {
          success(<response image url>);
        }, function () {
          failure("my failure message");
        });
      }
    }
  }
};

var editor = textboxio.replace('#targetId', config);

```

links

```
configuration
  links
```

The `configuration links` property defines options related to `editor` hyperlink behavior.

Properties

Property	Type	Default	Properties
<code>validation</code>	Object		Configuration values for hyperlink URL validation

embed

```
configuration
  links
    embed
```

The `configuration` embed property defines options related to `editor` link embed behavior. When configured, the editor will embed links inserted into the editor on pressing the enter key or paste.

This option was introduced in Textbox.io release 2.2.0.

Example Configuration

In this example, a simple `configuration` object is created to define a link validation service.


```
var configLinks = {
  links : {
    embed : {
      url : 'http://yourlinks.server.com/ephox-hyperlinking/'
    }
  }
};

var editor = textboxio.replace('#targetId', configLinks);
```

Properties

Property	Type	Value
<code>url</code>	String	URL to the link embed service. For more information on the validation service, please see the services Installation and Setup section.

Services Disabled by Default

 [Textbox.io](#) services are not enabled by default. Defining service urls enables those services and their respective client components. For more details, see: [Server-Side Components](#).

validation

```
configuration
  links
    validation
```

The `configuration` validation property defines options related to `editor` link validation behavior. When configured, the editor will validate links inserted into the editor. Invalid links will be marked with a red dotted border.

Example Configuration

In this example, a simple `configuration` object is created to define a link validation service.


```
var configLinks = {
  links : {
    validation : {
      url : 'http://yourlinks.server.com/ephox-hyperlinking/'
    }
  }
};

var editor = textboxio.replace('#targetId', configLinks);
```

Properties

Property	Type	Value
<code>url</code>	String	URL to the link validation service. For more information on the validation service, please see the services Installation and Setup section.

Services Disabled by Default

 `Textbox.io` services are not enabled by default. Defining service urls enables those services and their respective client components. For more details, see: [Server-Side Components](#).

macros

Textbox.io ships with built-in macros that aim to improve the user editing experience. They are all enabled by default; this configuration setting can be used to control which macros are enabled. Additional macros can be added using the [editor.macros](#) runtime API.

While these macros are in all versions of the editor, this API to control them was introduced in Textbox.io release 2.4.1

Available macros

The built-in macros are grouped into sets based on the type of tags they produce. Markdown macros are also listed in the Textbox.io help dialog for user reference; disabling a macro set does not however adjust the contents of the dialog.

Headings

This macro set converts Markdown heading syntax into H1-H6 tags.

Syntax	HTML result
# Largest Heading	<h1>Largest Heading</h1>
## Larger Heading	<h2>Larger Heading</h2>
### Large Heading	<h3>Large Heading</h3>
#### Heading	<h4>Heading</h4>
##### Small Heading	<h5>Small Heading</h5>
##### Smallest Heading	<h6>Smallest Heading</h6>

Lists

This macro set converts Markdown list syntax into UL and OL tags.

Syntax	HTML result
* Unordered list	<ul style="list-style-type: disc">Unordered List
1. Ordered list	<ol style="list-style-type: decimal">Ordered List
a. Ordered list	<ol style="list-style-type: lower-alpha">Ordered List
i. Ordered list	<ol style="list-style-type: lower-roman">Ordered List
1) Ordered list	<ol style="list-style-type: decimal">Ordered List
a) Ordered list	<ol style="list-style-type: lower-alpha">Ordered List
i) Ordered List	<ol style="list-style-type: lower-alpha">Ordered List

Semantics

This macro set converts variations of the markdown Italic and Bold syntax into EM and STRONG tags.

Syntax	HTML result
Italic	Italic
Italic	Italic
bold	Bold
__bold__	Bold

HR

This macro converts a triple dash into a HR tag.

Syntax	HTML result
---	<hr />

Entities

This macro set offers a convenient shorthand for a few HTML entities

Syntax	HTML result	Character
(c)	©	©
--	—	—
-	—	—

Link

This macro implements the editor autolink feature.

Syntax	HTML result
http://ephox.com	http://ephox.com
www.ephox.com	www.ephox.com

Properties

Property	Type	Value
allowed	Array	An array of built-in macros that are allowed (any macro not in this list is disabled).

Example Configuration

This example shows the default macro configuration where all built-in macros are enabled.

```
var config = {
  macros: {
    allowed : [ 'headings', 'lists', 'semantics', 'entities', 'hr', 'link' ]
  }
};

var editor = textboxio.replace('#targetId', config);
```

See Also

[Macros: Writing Content-Aware Code](#)

paste

```
configuration  
  paste
```

The `configuration` `paste` property defines options related to `editor` paste behavior.

Example Configuration

In this example, a simple `configuration` object is created that overrides the default paste behavior. The simple `configuration` object is then passed to `replace` to create an `editor`.

```
var configPaste = {  
  paste : {  
    style : 'clean', // Overrides default: 'prompt' for MS Office content  
    enableFlashImport: true // Note, true is the default  
  }  
};  
  
var editor = textboxio.replace('#targetId', configPaste);
```

Properties

Property	Type	Default	Values
<code>style</code>	String	'prompt'	'clean', 'retain', 'plain' or 'prompt'
<code>enableFlashImport</code>	Boolean	true	false or true

Style Values

Value	Description
prompt	(Default for importing MS Office content) A prompt is displayed to the user on paste. The prompt asks the user to either remove or retain inline CSS styling.
clean	Inline CSS styling is stripped from pasted content.
plain	Only the plain text clipboard content is pasted, if any is available
retain	(Default for all other content) Inline CSS styling is retained in pasted content.

Enable Flash Import


This option was introduced in Textbox.io release 1.3.1

On some browsers, a flash plugin is used to extract images when pasting from MS Office documents. On these browsers the use of flash can be disabled.

This option has no impact on browsers that do not require flash; on those browsers images will still be included. To disable local image pasting completely use the `allowLocal` configuration option.

Value	Description
true	Allow flash to be used to extract images when pasting from MS Office documents
false	On browsers that require flash, no images will be pasted when copying and pasting from MS Office documents. If there are images in the document, the user will see a warning that images were blocked. Other content will still be pasted as normal.

Inline CSS styling is CSS that is applied directly to HTML elements via the HTML style attribute ([W3C](#)).
Protected View

 *Note: When using the Windows operating system, copying and pasting content from Word 2013 or later in Protected View will result in plain, unformatted text. This is due to how Protected View interacts with the clipboard.*

spelling

```
configuration
  spelling
```

The `spelling` property defines the location of [Textbox.io server-side spelling service](#).

Example Configuration

In this example, a simple `configuration` object is created that defines the location for the Textbox.io Server Components spelling service for an editor instance. The configuration is used to generate an editor via [replace](#).


```
var config = {
  spelling : {
    url : 'http://yourspelling.server.com/ephox-spelling/',
    autocorrect : true
  }
};

var editor = textboxio.replace('#targetId', config);
```

Properties

Property	Type	Properties
<code>url</code>	String	A URL string that defines the endpoint for the Server-Side Components spelling service .
<code>autocorrect</code>	Boolean	A boolean value to enable spelling autocorrect by default

Services Disabled by Default

 Textbox.io services are not enabled by default. Defining service urls enables those services and their respective client components. For more details, see: [Server-Side Components](#).

ui

```
configuration
  ui
```

The `configuration ui` property defines options related to the editor UI, and allows for the configuration of the `editor` toolbar.

Properties

<code>aria-label</code>	String	Text to use for the editor's ARIA label instead of the default
<code>autoresize</code>	Boolean	Automatically scale the vertical height of the editor to fit the content
<code>colors</code>	Object	Specifies the list of colors available to the editor
<code>fonts</code>	Array	Specifies the list of fonts available to the editor
<code>languages</code>	Array	Specifies available language codes that can be applied to content for internationalization (HTML lang attribute)
<code>locale</code>	String	Specifies editor UI language
<code>shortcuts</code>	Boolean	Whether to enable content keyboard shortcuts
<code>toolbar</code>	Object	An object representing the desired toolbar functionality for an <code>editor</code> .

aria-label

```
configuration
  ui
    aria-label
```

The `ui` `aria-label` property defines the editor's [ARIA label](#). The editor's ARIA label identifies the editor to screen readers and other assistive devices.

Default label

If no label is configured, the default ARIA label is "Textbox.io Rich Text Editor - `<id>`" where `<id>` is the ID attribute of the element used to load the editor. When the editor fullscreen feature is used, this changes to "Textbox.io Fullscreen Rich Text Editor - `<id>`".

If no ID is available on the target element and the default label is in use, a random 6 digit number is used instead.

The default ARIA label is translated into all supported languages, and will match the [locale](#) of the editor.

Configured label

If a label is configured none of the above translation or referencing the target element ID is performed. The string is used exactly as configured.

Example Configuration

In this example using default configuration, the ARIA label will be "Textbox.io Rich Text Editor - blog"

```
<div id="blog"></div>

<script type="text/javascript">
var editor = textboxio.replace('#blog');
</script>
```

In this example, a simple [configuration](#) object is created that specifies a custom ARIA Label.

The ARIA label will be "Post Body Text", as none of the default label is included in this scenario.

```
var configUi = {
  ui: {
    'aria-label': 'Post Body Text'
  }
};

var editor = textboxio.replace('#targetId', configUi);
```

autoresize

```
configuration
  ui
    autoresize
```

The `autoresize` attribute, when set to `true`, causes the editor's height to automatically scale to fit the content inside. The editor's height will match that of the content, with height increasing and decreasing as the content changes. Vertical scrollbars will never appear.

This option was introduced in Textbox.io release 2.4.1

Sticky



Enabling `autoresize` will automatically enable `ui.toolbar.sticky`, unless overridden, to ensure the toolbar is still usable if the editor grows taller than the window.

Example Configuration

In this example, a simple `configuration` object is created that enables `autoresize` for the `editor` created by `replace`.

```
var config = {
  ui : {
    autoresize : true
  }
};

var editor = textboxio.replace('#targetId', config);
```


colors

The textbox.io color widget offers customisation to adjust the available colour buttons. These changes apply to all features where the widget is used:

- Font and Highlight color
- Table Cell Border
- Table Cell Background

The widget will grow or shrink vertically as required to accommodate the list of colors. The column count is not configurable.

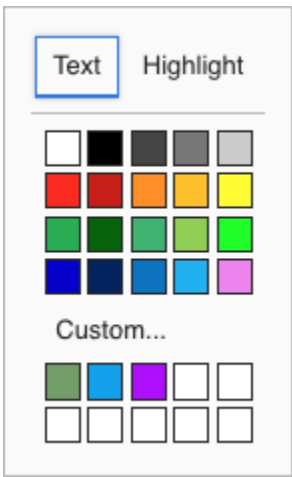
This API was introduced in Textbox.io release 2.4.1



The custom colour picker was added in Textbox.io release 2.4.2

Default colors

The default set of colors includes a color name used for the tooltip and ARIA label. These names are translated to all supported languages, and the API allows use of these translations. To use a translated name, set the text of a custom color to one of the `Translation Key` entries from this table; translation even works if the text is not provided but the color matches a translation key.



The default colors, in order, are:

HTML color value	Translation Key	English Color Name
#FFF	white	White
#000	black	Black
#444	gray	Gray
#777	metal	Metal
#CCC	smoke	Smoke
#FC1D00	red	Red
#C81500	darkred	Dark Red
#FF8C00	darkorange	Dark Orange
#FEBE00	orange	Orange
#FFFC00	yellow	Yellow
#22AE50	green	Green
#006400	darkgreen	Dark Green
#3CB371	mediumseagreen	Medium Sea Green
#8FCD4E	lightgreen	Light Green
#00FF00	lime	Lime
#0000CD	mediumblue	Medium Blue

#002360	navy	Navy
#0173C1	blue	Blue
#14B2F2	lightblue	Light Blue
#EE82EE	violet	Violet

Properties

Property	Type	Default	Value
buttons	Array	All colors listed above, in order	An array of color button objects to show on the widget
custom	Boolean	true	Whether to show a custom color picker option below the buttons

Color Button objects

Property	Type	Value
value	String	The HTML color representation. This string will be set as the color in the document without additional processing. While the default colors all use hex representation, any valid HTML color value can be used.
text	String (optional)	Text representation used for the tooltip and ARIA label of the color button. See note above about translations. If no text is provided, the <code>color</code> string is used as the <code>text</code> string.

Example Configuration

This example demonstrates a color configuration where a single row of 5 buttons are shown.

The first three take advantage of the fact that translation keys are also valid HTML color names. The fourth supplies a custom string, and the fifth replaces one of the built-in colors by using a custom color value and a translation key for the text.

fonts

```
configuration
  ui
    fonts
```

The `ui` fonts property defines the list of available fonts from the fonts dropdown menu.

The fonts property accepts an array of strings that define font-face values. These font-face values are then seen in the editor font menu. Note that font-face values may contain font fallbacks if you so choose.

For more advanced use cases, the fonts property can also accept an array of objects. Each object should contain a value and text property. The value property defines the font-face value, while the text property defines the visible label given to the font-face value in the editor font menu.

Simple Example

If strings are provided to the fonts array, the editor will use these values to construct available font-face values in the font menu. Note that these strings will be displayed in the font menu UI unchanged.

Easy configuration

```
// Create a simple font menu configuration with some single font choices and a font fallback
var configUi = {
  ui : {
    fonts : [ 'Helvetica', 'Arial', 'Times New Roman', '"Comic Sans MS", cursive, sans-serif' ]
  }
};

var editor = textboxio.replace('#targetId', configUi);
```

Advanced Example

If objects are provided to the fonts array that define a value and text, the editor will use the value property to define the CSS font-face to apply while displaying the font menu item in the UI with the text provided. This allows you to customize the display names of fonts available in the font menu.

Full configuration

```
var configUi = {
  ui : {
    fonts : [
      {
        value: '"Comic Sans MS", cursive, sans-serif',
        text: 'A silly font'
      },
      {
        value: 'Tahoma' // equivalent to providing just a string
      },
      'Arial', // you can use a mixture of objects and strings
      {
        value: 'Helvetica',
        text: 'A nicer font'
      }
    ]
  }
};

var editor = textboxio.replace('#targetId', configUi);
```

Advanced Object Properties

value	String	The font name
text	String	Optional display name for the font. When not provided, the value is used as the text.

languages

```
configuration
  ui
    languages
```

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 [command and id's section here](#).

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.

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-klington'), that language code will appear in the language menu.

Example Configuration

In this example, a simple `configuration` object is created that specifies which languages to use in the language menu. Note, as mentioned before, not all of these language codes will be translated by Textbox.io.

```
var config = {
  ui: {
    languages : [          // Languages array, sets the available languages to apply
      'fr',                // French
      'fr_ca',             // French (Canadian)
      'en_gb',             // English (United Kingdom)
      'x-klington' // x-klington : klington
    ]
  }
};

var editor = textboxio.replace('#targetId', config);
```

Properties

Property	Type	Properties
languages	Array	array of string language codes

Defaults

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

Translated Language Codes

Language	Language Code String
----------	----------------------

Arabic	ar
Catalan	ca
Chinese	zh
Chinese (Simplified)	zh_cn
Chinese (Traditional)	zh_tw
Croatian	hr
Czech	cs
Danish	da
Dutch	nl
English	en
English (Australia)	en_au
English (Canada)	en_ca
English (United Kingdom)	en_gb
English (United States)	en_us
Farsi	fa
Finnish	fi
French	fr
French (Canada)	fr_ca
German	de
Greek	el
Hebrew	he
Hungarian	hu
Italian	it
Japanese	ja
Kazakh	kk
Korean	ko
Norwegian	no
Polish	pl
Portugese	pt
Portuguese (Brazil)	pt_br
Portuguese (Portugal)	pt_pt
Romanian	ro
Russian	ru
Slovak	sk
Slovenian	sl
Spanish	es
Spanish (Latin America)	es_419
Spanish (Spain)	es_es
Swedish	sv
Thai	th
Tartar	tt

Turkish	tr
Ukrainian	uk

locale

```
configuration
  ui
    locale
```

The locale attribute lets a developer choose a language UI translation from one of the [33 available languages](#) by specifying its locale string.

By default, the editor language is automatically set according to the client browser's language. The locale property allows the developer to specify an editor UI translation directly, overriding the browser's default language.

Multiple Locales



Using more than one unique value for editor locale on a page is not currently supported. This includes configuring one editor to use the browser default and another to use a specific value via this API.

Example Configuration

In this example, a simple `configuration` object is created that defines the locale for the editor UI for the `editor` created by `replace`.

```
var config = {
  ui : {
    locale : 'fr' //sets the editor language to french
  }
};

var editor = textboxio.replace('#targetId', config);
```

Properties

Property	Type	Properties
locale	String	locale short code: e.g en for English

Supported Locales

Language	Locale String
Arabic	ar
Catalan	ca
Chinese (Simplified)	zh
Chinese (Traditional)	zh_tw
Croatian	hr
Czech	cs
Danish	da
Dutch	nl
English	en
Farsi	fa
Finnish	fi
French (Europe)	fr
German	de
Greek	el

Hebrew	he
Hungarian	hu
Italian	it
Japanese	ja
Kazakh	kk
Korean	ko
Norwegian	no
Polish	pl
Portuguese (Brazil)	pt_br
Portuguese (Europe)	pt_pt
Romanian	ro
Russian	ru
Slovak	sk
Slovenian	sl
Spanish	es
Swedish	sv
Turkish	tr
Thai	th
Ukrainian	uk

shortcuts

```
configuration
  ui
    shortcuts
```

The `ui shortcuts` property turns editor content keyboard shortcuts on or off. Content keyboard shortcuts include any keyboard shortcut that directly affects the contents of the editor (eg: indenting, marking bold text, etc).

This property is true by default. When set to false, most of the editor keyboard shortcuts will be disabled. The following shortcuts are not affected by this:

- undo
- redo
- focus toolbar
- open context menu

Use of this property does not alter the toolbar, it just deactivates the keyboard shortcuts.

Example Configuration

In this example, a simple `configuration` object is created that disables editor content keyboard shortcuts.

```
var configUi = {
  shortcuts : false
};

var editor = textboxio.replace('#targetId', configUi);
```

toolbar

```
configuration
  ui
    toolbar
```

The `ui toolbar` property defines options related to editor toolbar commands, groups and child menus.

Context toolbar items

These items currently appear at the end of the toolbar. Specific configuration is not available as they may move to a different part of the UI in a future release. They can however be disabled:

```
var customToolbar = {
  contextual: [ ]
};

var config = {
  ui : { toolbar : customToolbar }
};

var editor = textboxio.replace('#targetId', config);
```

Example Configuration

In this example, a custom toolbar object with two custom toolbar buttons is created and added to a `configuration` object via the `ui toolbar` property. This config is then used to create an `editor` by `replace`.

```

var customToolbar = {
  items : [
    {
      label: 'Undo and Redo group',
      items: [ 'undo', 'redo' ]
    },
    {
      label: 'Insert group',
      items: [
        {
          id   : 'insert',
          label : 'Insert Menu',
          items : [ 'link', 'fileupload', 'table' ]
        }
      ]
    },
    {
      label: 'Custom Toolbar Group',
      items: [
        {
          id   : 'custom1',
          text : 'Custom Button 1',
          icon : '/path/to/icon1.png',
          action : function () { alert('Custom Button 1 Clicked'); }
        },
        {
          id   : 'custom2',
          text : 'Custom Button 2',
          icon : '/path/to/icon2.png',
          action : function () { alert('Custom Button 2 Clicked'); }
        }
      ]
    }
  ]
};

var config = {
  ui : { toolbar : customToolbar }
};

var editor = textboxio.replace('#targetId', config);

```

Properties

Item	Properties	Default	Description
items	Array	(Default Toolbar)	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
visible (demo)	Boolean	true	Inline editing only. Whether the toolbar should be visible in the UI at all.
draggable (demo)	Boolean	true	Inline editing only. Whether the editor should be draggable from its offset position.
offset top left (demo)	Position	top: 0, left: 0	Inline editing only. The offset coordinates of the toolbar from the top-left vertex of the container being edited.

sticky	Boolean	the same value as ui.autosize	iframe editing only. Whether the toolbar should stick to the top of the window as it scrolls, similar to Inline editing behaviour. This option was introduced in Textbox.io release 2.4.1
--------	---------	---	---

Items

Toolbars are made up of item objects. Items represent either editor *commands* (apply bold), toolbar *groups*, or *menus*.


Items infer their UI from their position in an 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.

Item Types

Items have 3 distinct types, representing ui constructs in a Textbox.io editor. *command* items represent discrete editor functionality. *menu* items represent a nested group of commands invoked from a root UI element. *group* items represent logical groupings of *commands* either inside the toolbar or within menus.

Item	Properties	Description												
command	<table border="1"> <tr> <td>id</td> <td>String</td> <td>Id string for the command.</td> </tr> <tr> <td>text</td> <td>String</td> <td>(Optional) Friendly name of the command, shown in tooltips.</td> </tr> <tr> <td>icon</td> <td>String</td> <td>Path to the icon used to represent the command.</td> </tr> <tr> <td>action</td> <td>Function</td> <td>A function to be executed when the command is invoked via user action.</td> </tr> </table>	id	String	Id string for the command.	text	String	(Optional) Friendly name of the command, shown in tooltips.	icon	String	Path to the icon used to represent the command.	action	Function	A function to be executed when the command is invoked via user action.	<p>Command type items represent discrete editor commands, such as: apply bold, insert link, etc.</p> <p>Note that built-in command items are referenced by their string id rather than specified as objects.</p>
id	String	Id string for the command.												
text	String	(Optional) Friendly name of the command, shown in tooltips.												
icon	String	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	<table border="1"> <tr> <td>id</td> <td>String</td> <td>Id string for the menu.</td> </tr> <tr> <td>label</td> <td>String</td> <td>(Optional) Friendly name of the menu, visible to assistive devices per WAI-ARIA.</td> </tr> <tr> <td>icon</td> <td>String</td> <td>Path to the icon used to represent the command.</td> </tr> <tr> <td>items</td> <td>Array</td> <td>Array of command or menu items.</td> </tr> </table>	id	String	Id string for the menu.	label	String	(Optional) Friendly name of the menu, visible to assistive devices per WAI-ARIA .	icon	String	Path to the icon used to represent the command.	items	Array	Array of command or menu items.	<p>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.</p> <p>For users of assistive devices, the name of the menu is applied to menu's aria-label per WAI-ARIA.</p>
id	String	Id string for the menu.												
label	String	(Optional) Friendly name of the menu, visible to assistive devices per WAI-ARIA .												
icon	String	Path to the icon used to represent the command.												
items	Array	Array of command or menu items.												
group	<table border="1"> <tr> <td>label</td> <td>String</td> <td>(Optional) Friendly name of the menu, visible to assistive devices per WAI-ARIA.</td> </tr> <tr> <td>items</td> <td>Array</td> <td>An array of command or menu items.</td> </tr> </table>	label	String	(Optional) Friendly name of the menu, visible to assistive devices per WAI-ARIA .	items	Array	An array of command or menu items.	<p>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.</p> <p>For users of assistive devices, the name of the group is applied to group's aria-label per WAI-ARIA.</p>						
label	String	(Optional) Friendly name of the menu, visible to assistive devices per WAI-ARIA .												
items	Array	An array of command or menu items.												

Built-In Command Item IDs

 Built-in editor commands are represented by a predefined string id in toolbar configurations. For a list of built-in editor command ids see: [Command Item IDs](#).

Items Array Structure

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.

Group Items

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.

Command Items

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.

Menu Items

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.

Nested menus are only supported to the [second level](#).

Toolbar Items Array Structure

```
Toolbar Items
  Toolbar Group(s)
    Command Item(s)
      Menu Item(s)
        Command Item(s)
          Menu Items(s)
```

Group Item Example

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

Command Item Example

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

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

Menu Item Example

```
// Menu Item Object with 2 function IDs
var customMenuItem = {
  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 items = [
  {
    // Toolbar group object with custom menu item
    label : 'Toolbar Group 2',
    items : ['undo', 'redo', customMenuItem]
  }
];
```

See Also

- [Editor types - Classic vs Inline](#)

Command Item IDs

Built-in editor commands are represented by a string ID in Textbox.io toolbar configurations. The following built-in command item IDs are available for use in any Textbox.io instance. Adding a command via its ID to a menu structure adds that feature to the toolbar or menu.

New in 2.4



In release 2.4 special labels were added to aid in creating groups and menus very similar to the default, for example to recreate a default group without one of the items. These labels can be used to leverage the Textbox.io translations for both tooltips and ARIA labels.

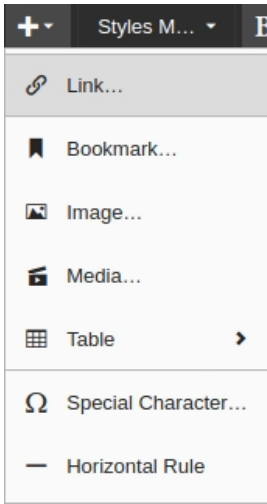







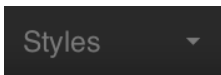






These labels are [documented below](#).







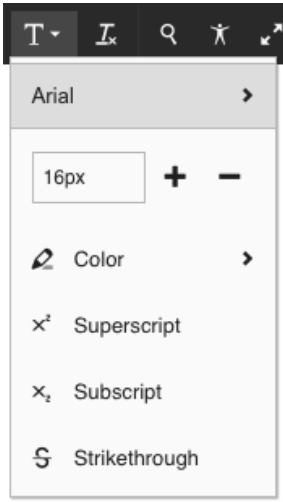


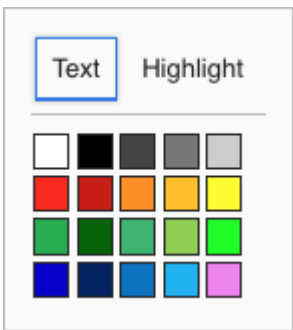
Group IDs







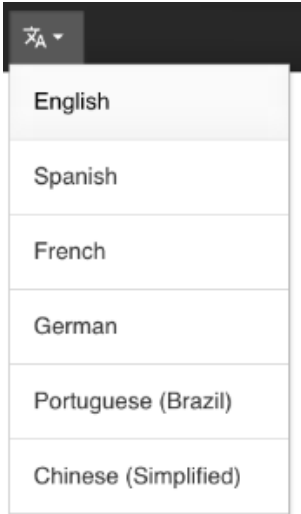


ID	Commands included	Appearance
undo	undo, redo	
insert	insert	
style	style	
emphasis	bold, italic, underline	
align	alignment	
listindent	ul, ol, indent, outdent, blockquote	
format	font-menu, removeformat	
tools	find, accessibility, fullscreen, usersettings	
language	language, ltrdir, rtlidir	

Command IDs

ID	Function Description	Appearance
undo	Trigger undo.	
redo	Trigger re-do.	

insert	<p>Default insert menu.</p> <p>Included commands:</p> <pre>link, bookmark, fileupload, table, media, hr, specialchar</pre>	
link	Open link insertion dialog.	
bookmark	Insert bookmark.	
fileupload	Open file/image upload dialog.	
table	Open table insertion menu.	
specialchar	Open special character chooser dialog.	
media	Open media embed insertion dialog.	
hr	Insert horizontal rule.	
styles	<p>Open style menu.</p> <p>The list can be configured with the styles configuration option.</p>	
bold	Apply bold to selection.	
italic	Apply italic to selection.	
underline	Apply underline to selection.	
strikethrough	Apply strikethrough to selection.	
superscript	Apply superscript to selection.	
subscript	Apply subscript to selection.	

alignment	Open modify element alignment menu.	
ul	Change selection to un-ordered list.	
ol	Change selection to ordered list.	
indent	Add indent to selection.	
outdent	Remove indent from selection.	
blockquote	Apply blockquote to selection.	
font-menu	Default font transformation menu. Included commands: <code>font-face</code> , <code>font-size</code> , <code>font-color</code> , <code>superscript</code> , <code>subscript</code> , <code>strikethrough</code>	
font-face	Select the font-face value from a list. The list can be configured with the fonts configuration option.	
font-size	Shows the font-size widget.	
font-color	Shows the font-color widget.	

removeformat	Remove formatting from selection.	
find	Toggle inline find and replace dialog	
accessibility	Accessibility checker	
fullscreen	Toggle full-screen mode.	
wordcount	Open word count dialog.	
usersettings	Open user settings menu. Included commands: <ul style="list-style-type: none"> • wordcount • feature toggles such as spelling and capitalisation • help 	
language	Open the language application menu. The list can be changed with the languages configuration option.	
ltrdir	Toggle left to right text direction.	
rtlDir	Toggle right to left text direction.	

Context Features


 These items are currently appended to the end of the toolbar. These features are not directly configurable but they can be disabled in the [toolbar](#) configuration.

Table Toolbar

The features for table cell styling appear when one or more table cells are selected:




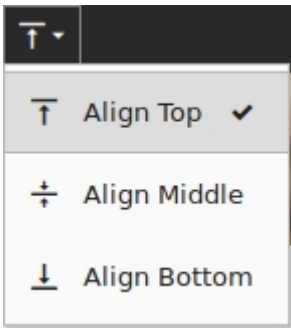





Cell Background Color	
Cell Border Color	
Cell Border Width	
Cell Vertical Alignment	

Image Editing Toolbar

The features for image editing appear when an image (and only an image) is selected:

Crop Image	
Rotate Left 90°	
Rotate Right 90°	
Horizontal Flip	
Vertical Flip	

Translated labels

To aid with the creation of custom menus and toolbar groups, the editor translations can be leveraged through special labels. English is shown here as an example but the appropriate language for the editor UI will be used.

Group labels

For each [Group ID](#), "group." is prepended to the name.

Group ID	special label value	English ARIA label
----------	---------------------	--------------------

undo	group.undo	Undo and Redo group
insert	group.insert	Insert group
style	group.style	Styles group
emphasis	group.emphasis	Formatting group
align	group.align	Alignment group
listindent	group.listindent	List and Indent group
format	group.format	Font group
tools	group.tools	Tools group
language	group.language	Language group

Menu labels

The two menus on the default toolbar that contain configurable items also have translations available.

Command ID	special label value	English ARIA label
insert	menu.insert	Insert Menu
font-menu	menu.font	Font Menu

Example Configuration

In this example, a simple [configuration](#) object is created that specifies a custom toolbar using built-in command ids for an [editor](#) created by [repl ace](#). It uses translated labels to create an experience matching the default undo group and insert menu but with a shorter menu.

```

var configBuiltIn = {
  ui : {
    toolbar : {
      items : [
        {
          label: 'group.undo',
          // Built-in Command ids: 'undo', 'redo'
          items: [ 'undo', 'redo' ]
        },
        {
          label: 'group.insert',
          items: [
            {
              // Built-in menu id: 'insert'
              id   : 'insert',
              label : 'menu.insert',
              // Built-in command ids: 'link', 'fileupload', 'table'
              items : [ 'link', 'fileupload', 'table' ]
            }
          ]
        }
      ]
    }
  }
};

var editor = textboxio.replace('#targetId', configBuiltIn);

```

editor

Textbox.io Editor instances are represented by `editor`. Instances are created by `replace`, `replaceAll`, `inline`, and `inlineAll`. You can also get editor instances by calling the `get` or `getActiveEditor()` methods.

The editor instance allows you to interact with a specific editor's `editor.content` and `editor.events`.

Properties

<code>editor.content</code>	Groups methods relating to an editor instance's content.
<code>editor.events</code>	Groups methods relating to an editor instance's event model.
<code>editor.filters</code>	Groups properties relating to an editor instance's content filters.
<code>editor.macros</code>	Groups properties relating to an editor instance's macros.
<code>mode</code>	Provides methods to set/get the editor's mode to either <code>code view</code> or <code>design view</code> . See HTML Code View for more information.

Methods

<code>element</code>	Returns the editor container DOM element.
<code>focus</code>	Focus a Textbox.io editor instance.
<code>message</code>	Display a message banner in the editor UI.
<code>restore</code>	Removes this Textbox.io Editor instance from its DOM element.

Note

`get` returns arrays of editor instances. The examples in this section assume a single editor - the first editor instance in the returned array.

```
// Retrieve the first editor instance in the returned array
var editors = textboxio.get('textarea');
var editor = editors[0];
```

editor.content

eGroups together functions related to the content of a Textbox.io Editor instance.

Properties

<code>editor.content.selection</code>	Methods for working with selected content in the editor
---------------------------------------	---

Methods

<code>editor.content.get()</code>	Retrieves the HTML contents of an instance's content <code><body></code> element.
<code>editor.content.set()</code>	Sets the HTML contents of an instance's content <code><body></code> element.
<code>editor.content.documentElement()</code>	Retrieves the editors HTML Document Object.
<code>editor.content.insertHtmlAtCursor()</code>	Inserts HTML at the last known cursor position in an instance.
<code>editor.content.isDirty()</code>	Check if the content has been updated or changed since it was set.
<code>editor.content.setDirty()</code>	Set the content dirty state.
<code>editor.content.uploadImages()</code>	Adds Base64 images to the Textbox.io Services upload queue.

Deprecated methods

`editor.content.getSelectedText` has been deprecated in favour of the selection `getText` method

editor.content.set()

Set the HTML content of an instance's <body> element with `editor.content.set()`.

Example

editor.content.set(html)

```
// Set the HTML content of an instance
var contentToSet = "<p>Any HTML Content</p>";
editor.content.set(contentToSet);
```


Parameters

html	String	Specify the HTML to which the editor's contents will be set as a string.
------	--------	--

Returns

No return value.

Injecting Content into JavaScript

 If you are using a server side programming language to inject your content you need to make sure that the result of your code creates valid JavaScript. For example, if you use this in a JSP:

```
var content = "${myObject.content}";
```

you will create invalid JavaScript if `${myObject.content}` contains any double quotes and/or carriage returns and/or line feeds. If `${myObject.content}` had a double quote you would end up with this:

```
var content = "<p>This is content with "quoted content" in the paragraph</p>";
```

As you can see the double quotes in the content leads to a malformed string. CR/LF characters will create similar issues as the string would stretch over multiple lines. To avoid these issues you should **remove** CR and LF characters and **escape** any other characters that could cause issues for a string. These include:


- Single quote
- Double quote

How you escape these characters is specific to the server side language you are using but all popular server side languages provide the ability to escape characters in strings.

editor.content.get()

Retrieve the HTML contents of an instance's <body> element with `editor.content.get()`.

Image Uploads

 Local images are not guaranteed to be uploaded when this API is used. If they have not been uploaded, image data will be returned in base64.

For more information, see the [Handling Local Images](#) and [Handling Asynchronous Image Uploads](#) articles.

Example

editor.content.get()

```
// Retrieve the HTML content of the instance as a string
var content = editor.content.get();
```

Returns

BODY HTML	String	The HTML content of the instance's <body> element as a string.
-----------	--------	--

editor.content.insertHtmlAtCursor()

Insert HTML content at the cursor position with `editor.content.insertHtmlAtCursor()`.

Content is inserted at the last known cursor position, regardless of whether the editor has focus.

Example

`editor.content.insertHtmlAtCursor(html)`

```
// Insert a string of HTML at the cursor position
var newContent = "<p>Any HTML Content</p>";
editor.content.insertHtmlAtCursor(newContent);
```

Parameters

html	String	A string of HTML that will be inserted at the last known cursor position.
------	--------	---

Returns

No return value.

editor.content.documentElement()

Access an instance's HTML document object by using `editor.content.documentElement()`.

To obtain the body element, simply use the `document.body` property.

Example

editor.content.documentElement()

```
// Retrieve an editor's document, then identify the body element
var edDocument = editor.content.documentElement();
var edBody = edDocument.body;
```

Returns

HTML Document	Object	The HTML document element for the target editor. Null if the editor has not finished loading.
---------------	--------	---

editor.content.uploadImages()

Add all Base64 images from an `editor`'s content to the Textbox.io Services upload queue with `editor.content.uploadImages()`.

If `editor.content.uploadImages()` is called before an existing image upload has completed, identified Base64 images will be appended to the upload queue. Images that are not Base64 encoded will be ignored.

Upon completion of the queue, `editor.content.uploadImages()` triggers a callback function. This callback function is passed a result set containing a list of all images uploaded during since `editor.content.uploadImages()` was called. Each item in the result set contains a DOM reference to the ``, and a status (success/failure) for that image upload.

- A *success* status means that the `` should now contain a valid `src` attribute to uploaded image.
- A *false* status implies that the image has failed to upload. After a failed upload, the image will still be Base64 encoded in the `editor` content. If `editor.content.uploadImages()` is called again, the editor will pick up the failed image and try again.

Example

editor.content.uploadImages()

```
var callback = function (results) {
    console.log('Images have finished uploading.');
```

```
    results.forEach(function (result) {
        console.log('upload successful: ' + result.success);
        console.log('the image element ', result.element);
    });
};
```

```
// Add all Base64 images from editor content to the upload queue
editor.content.uploadImages(callback);
```

Parameters

callback	Function	Receives an array of objects with the upload status and DOM reference of each image that was uploaded. Objects are in the form of: <pre>var result = { success: true, element: reference }</pre>
----------	----------	--

Returns

No return value.

editor.content.getSelectedText()

Example

```
editor.content.getSelectedText()
```

```
// Retrieve the current editor text selection  
var selectedText = editor.content.getSelectedText();
```

Returns

String	The text content of the selection. If no text selection exists, an empty string ("") is returned.
--------	---

editor.content.isDirty()

Check if an instance's content has been updated or changed since it was set.

Content state is clean after any of these methods are invoked:

- `textboxio.replace()`
- `textboxio.replaceAll()`
- `textboxio.inline()`
- `textboxio.inlineAll()`
- `editor.content.set()`
- `editor.content.setDirty(false)`

Content state is considered dirty if:

- the content is modified
- `editor.content.setDirty(true)` is used

Returns

Boolean	Returns True if the content has changed since it was last set.
---------	--

editor.content.setDirty()

Explicitly set the content dirty state.

Parameters

Bool	Set the content dirty state with a boolean value
------	--

Returns

No return value.

Example:

setDirty example

```
// Set isDirty to false
editor.content.setDirty(false);
```

editor.content.selection

The aim of this API is to provide detailed access to the editor selection while adding a layer of safety for developers. As a result it may be somewhat more limited than is convenient for developers accustomed to raw DOM access.

This API was introduced in Textbox.io release 2.2.1

Methods

Name	Parameters	Returns	Details
getText	none	String	Returns the text content of the selection, removing all HTML tags. If the selection is collapsed an empty string (" ") is returned.
getHtml	none	String	Similar to <code>getText</code> but returns the complete HTML representation. Partially selected tags are automatically closed at the edge of the selection range.
findTagAtCursor	CSS3 selector (String)	Element or null	See below for Element API details. <code>Null</code> is returned if the cursor is not within a tag matching the selector.

API status



This API is fully supported but has been released with bare minimum functionality to gauge developer interest and gather feedback.

Please [contact us](#) if there is an addition to the selection API you would like to see in a future release.

Examples

editor.content.selection.getText()

```
// Retrieve the current editor selection as text
var selectedText = editor.content.selection.getText();
```

editor.content.selection.getHtml()

```
// Retrieve the current editor selection as HTML
var selectedHtml = editor.content.selection.getHtml();
```

editor.content.selection.findTagAtCursor()

```
// Retrieve a reference to a hyperlink surrounding the cursor
var selectedLink = editor.content.selection.findTagAtCursor('a[href]');
```

Element object

The Element API is a wrapper around DOM operations designed to smooth over differences between browsers and clean up internal editor attributes from returned data.


Do not retain references



The Element object retains a reference to a DOM node within the editor, which is easily replaced by user action (for example undo and redo). Retaining a reference to Element objects in your application will cause a memory leak.

Methods

Name	Parameters	Returns	Details
getAttributes	none	Object	Returns a frozen object of <code>key:value</code> pairs matching the element attribute <code>name:value</code> pairs (see below). Tag name is not included.
setAttributes	Object	no return value	Given an object similar to the one returned by <code>getAttributes</code> , adds (or overwrites) attributes on the element. This method cannot be used to remove attributes.
getText	none	String	Identical to <code>selection.getText()</code> but returns content for this element only

getHtml	none	String	Identical to <code>selection.getHtml()</code> but returns content for this element only
setText	String	no return value	Updates the element text contents (using <code>.textContent</code>)
setHtml	String	no return value	Updates the element HTML contents (using <code>.innerHTML</code>)
replaceElement	String	new Element object	<p>Completely replaces the element in the document and handles cursor placement as it will be impacted by this change. This is equivalent to setting <code>outerHTML</code> on the DOM node, except the old element reference is not retained.</p> <p>If parsing the string argument produces more than one HTML element, a JavaScript error is thrown.</p> <p>Replaced Element references</p> <p> Once an element has been replaced, any reference to the original element should be discarded. Continuing to use it will either have no effect or fail with a DOM exception.</p>

Examples

attribute objects

```
// for a HTML element <a class="mylink" href="http://ephox.com" alt="alternative text for link" target="_blank"
>...</a>
var attributes = {
  href: 'http://ephox.com',
  alt: 'alternative text for link',
  target: '_blank',
  'class': 'mylink'
}
```

replaceElement

```
var newElement = element.replaceElement('<a href="http://google.com">new text content</a>');
```

Adjusting link attributes

```
// Retrieve a reference to a hyperlink surrounding the cursor - assuming not null for this example
var selectedLink = editor.content.selection.findTagAtCursor('a[href]');

// Retrieve the attributes of the link
var linkAttributes = selectedLink.getAttributes();

// Create new attributes. Mutating the linkAttributes variable will have no effect, it's a frozen object
var newAttributes = {
  href: linkAttributes.href + '?replaced=true',
  'data-link-details': 'replaced'
};

// Update the link attributes
selectedLink.setAttributes(newAttributes);
```

Updating a complete link

```
// Retrieve a reference to a hyperlink surrounding the cursor - assuming not null for this example
var selectedLink = editor.content.selection.findTagAtCursor('a[href]');

// Retrieve the HTML contents of the link
var linkHtml = selectedLink.getHtml();

// Replace the link completely without changing the contents
selectedLink = selectedLink.replaceElement('<a href="http://google.com">' + linkHtml + '</a>');
```

editor.element

Returns the editor container DOM element, this HTML element is unique to each editor instance and contains all Textbox.io UI components.

Example

editor.restore()

```
// Replace the element with id 'replaceMe'  
var editor = textboxio.replace( '#replaceMe' );  
// Get the editor container element  
var element = editor.element();
```

Returns

element	HTML element	The element that contains all the Textbox.io UI components for the editor instance.
---------	--------------	---

editor.events

`editor.events` is a grouping of properties related to the events of a Textbox.io Editor instance.

Editor events allow you to detect when the editor is shown and focused.

Properties

<code>editor.events.loaded</code>	Triggered when an editor instance has finished loading.
<code>editor.events.focus</code>	Triggered when an editor receives focus.
<code>editor.events.dirty</code>	Triggered when content has been modified or considered dirty.
<code>editor.events.change</code>	Triggered when content changes.

editor.events.loaded

`editor.events.loaded` is a grouping of methods related to the loaded event for a Textbox.io Editor instance.

The loaded event is triggered when an editor has finished loading.

Methods

<code>editor.events.loaded.addListener()</code>	Binds a function to an editor instance's loaded event.
<code>editor.events.loaded.removeListener()</code>	Removes a function from an editor instance's loaded event.

Example

```
editor.events.loaded.addListener(function () {  
    // do something  
    console.log('The editor instance has loaded')  
});
```

editor.events.loaded.addListener()

Add a custom event listener to a Textbox.io editor loaded event with `editor.events.loaded.addListener()`.

You can detect that an editor has finished loading by attaching a callback function to the editor's loaded event.

Example

`editor.events.loaded.addListener(callback)`

```
// Add a custom event listener to the editor loaded event
editor.events.loaded.addListener(function() {
  alert('Textbox.io has loaded.');
```

Parameters

callback	Function	Function executed when the editor has loaded.
----------	----------	---

Returns

No return value.

editor.events.loaded.removeListener()

Remove a custom event listener on a Textbox.io editor loaded event with `editor.events.loaded.removeListener()`.

You may remove any custom event listeners by passing the callback function used to specify the event with `.addListener()` to `.removeListener()`.

Example

editor.events.loaded.removeListener(callback)

```
var notify = function () {
    alert('Textbox.io has loaded.');
```



```
};

// Add a custom event listener to the editor loaded event
editor.events.loaded.addListener(notify);

// Remove a custom event listener
editor.events.loaded.removeListener(notify);
```

Parameters

callback	Function	Function used to specify the custom event listener that should be removed.
----------	----------	--

Returns

No return value.

editor.events.focus

`editor.events.focus` is a grouping of methods related to the focus event for a [Textbox.io](#) Editor instance.

The focus event is triggered when an editor receives focus.

Methods

<code>editor.events.focus.addListener()</code>	Binds a function to an editor instance's focus event.
<code>editor.events.focus.removeListener()</code>	Removes a function from an editor instance's focus event.

Example

```
editor.events.focus.addListener(function () {  
  // do something  
  console.log('this editor has focus', editor.element());  
});
```

editor.events.focus.addListener()

Add a custom event listener to a [Textbox.io](#) editor focus event with `editor.events.focus.addListener()`.

You can detect that an editor has received focus by attaching a callback function to the editor's focus event.

Example

editor.events.loaded.addListener(callback)

```
// Add a custom event listener to the editor focus event
editor.events.focus.addListener(function() {
    alert('Editor got focus.');
```

Parameters

callback	Function	Function executed when the editor receives focus.
----------	----------	---

Returns

No return value.

editor.events.focus.removeListener()

Remove a custom event listener on a [Textbox.io](#) editor focus event with `editor.events.focus.removeListener()`.

You may remove any custom event listeners by passing the callback function used to specify the event with `.addListener()` to `.removeListener()`.

Example

editor.events.loaded.addListener(callback)

```
var notify = function () {
    alert('Textbox.io got focus.');
```



```
};

// Add a custom event listener to the editor focus event
editor.events.focus.addListener(notify);

// Remove a custom event listener
editor.events.focus.removeListener(notify);
```

Parameters

callback	Function	Function used to specify the custom event listener that should be removed.
----------	----------	--

Returns

No return value.

editor.events.dirty

The dirty event is triggered when an editors content is modified or has been explicitly set by `editor.content.setDirty(true)`.

Methods

<code>editor.events.dirty.addListener()</code>	Binds a function to an editor instance's dirty event.
<code>editor.events.dirty.removeListener()</code>	Removes a function from an editor instance's dirty event.

Example

```
editor.events.dirty.addListener(function () {  
    // do something  
    console.log('editor content is now dirty', editor.element())  
});
```

editor.events.dirty.addListener()

Add a custom event listener to a [Textbox.io](#) editor dirty event with `editor.events.dirty.addListener()`.

You can detect that content has changed on an editor instance by attaching a callback function to the dirty event.

The dirty event only occurs when an instance's content has been updated or changed since it was set. See [editor.content.isDirty\(\)](#) for more detail on the editor 'clean' and 'dirty' states.

Example

editor.events.loaded.addListener(callback)

```
// Add a custom event listener to the content dirty event
editor.events.dirty.addListener(function() {
  alert('The editor content is now dirty.');
```

Parameters

callback	Function	Function executed when the editor's content state is considered dirty.
----------	----------	--

Returns

No return value.

editor.events.dirty.removeListener()

Remove a custom event listener on a [Textbox.io](#) editor's content dirty event with `editor.events.dirty.removeListener()`.

You may remove any custom event listeners by passing the callback function used to specify the event with `.addListener()` to `.removeListener()`.

Example

`editor.events.loaded.addListener(callback)`

```
var notify = function () {
    alert('The editor content is now dirty.');
```



```
};

// Add a custom event listener to the editor's content dirty event
editor.events.dirty.addListener(notify);

// Remove a custom event listener
editor.events.dirty.removeListener(notify);
```

Parameters

callback	Function	Function used to specify the custom event listener that should be removed.
----------	----------	--

Returns

No return value.

editor.events.change

The change event is triggered when the editor content changes as defined by when an undo save point is created. These points are throttled to happen a few seconds after the user has stopped typing, or when the enter key is pressed.

This API was introduced in Textbox.io release 2.3.0

Methods

<code>editor.events.change.addListener()</code>	Binds a function to an editor instance's change event.
<code>editor.events.change.removeListener()</code>	Removes a function from an editor instance's change event.

Example

```
editor.events.change.addListener(function () {  
    // do something  
    console.log('The editor content has changed')  
});
```

editor.events.change.addListener()

Add a custom event listener to a Textbox.io editor change event with `editor.events.change.addListener()`.

This API was introduced in Textbox.io release 2.3.0

You can detect when editor content changes by attaching a callback function to the editor's change event.

Example

`editor.events.change.addListener(callback)`

```
// Add a custom event listener to the editor change event
editor.events.change.addListener(function() {
  alert('Editor content has changed.');
```

Parameters

callback	Function	Function executed when the editor content changes.
----------	----------	--

Returns

No return value.

editor.events.change.removeListener()

Remove a custom event listener on a Textbox.io editor change event with `editor.events.change.removeListener()`.

This API was introduced in Textbox.io release 2.3.0

You may remove any custom event listeners by passing the callback function used to specify the event with `.addListener()` to `.removeListener()`.

Example

`editor.events.change.removeListener(callback)`

```
var notify = function () {
    alert('Textbox.io editor content has changed.');
```



```
};

// Add a custom event listener to the editor change event
editor.events.change.addListener(notify);

// Remove a custom event listener
editor.events.change.removeListener(notify);
```

Parameters

callback	Function	Function used to specify the custom event listener that should be removed.
----------	----------	--

Returns

No return value.

editor.filters

`editor.filters` is a grouping of properties related to content filtering for a Textbox.io editor instance.

You may add filters to manipulate the HTML content on the way into (or on the way out of) the editor. Filtering occurs when triggering `editor.content` methods.

All Textbox.io filters work by passing DOM elements to a callback function. This function can then operate on the elements as necessary - removing them, transforming them, or even storing that content elsewhere within your application.

Properties

<code>selector</code>	Groups methods relating to an selector based filters.
<code>predicate</code>	Groups methods relating to predicate based filters.

selector

[selector](#) is a grouping of methods related to selector based content filtering for a Textbox.io editor instance.

Selector based filters let you identify the elements to be filtered by specifying a [CSS3 selector](#). Elements matching the selector are added to an array which is then passed to a callback function. The callback function then operates on the element array.

Methods

<code>editor.filters.selector.addInput()</code>	Creates a filter when content is added to the editor with: <code>textboxio.replace()</code> , <code>editor.content.set()</code> , <code>editor.content.insertHtmlAtCursor()</code> .
<code>editor.filters.selector.addOutput()</code>	Creates a filter when content is requested from the editor with: <code>editor.content.get()</code> .

See Also

[Filtering Content](#)

editor.filters.selector.addInput()

Create a selector based input filter for an editor instance with `editor.filters.selector.addInput()`.

The selector based input filter modifies content added to an editor with: `textboxio.replace()`, `editor.content.set()`, `editor.content.insertHtmlAtCursor()`.

Example

`editor.filters.selector.addInput(selector, callback)`

```
// Create an input filter for elements with class 'blue' that strips classes from identified elements
editor.filters.selector.addInput('.blue', function(elements) {
  elements.forEach(function(element) {
    element.className = "";
  });
});
```

Parameters

selector	String	Specify a CSS3 selector representing the elements you wish to pass to the filter.
callback	Function	Specify a function to process the array of matched elements.

Returns

No return value.

See Also

[Filtering Content](#)

editor.filters.selector.addOutput()

Create a selector based output filter for an editor instance with `editor.filters.selector.addOutput()`.

The selector based output filter modifies content requested from an editor with: `editor.content.get()`.

Example

`editor.filters.selector.addOutput(selector, callback)`

```
// Create an output filter for elements with class 'green' that strips classes from identified elements
editor.filters.selector.addOutput('.green', function(elements) {
  elements.forEach(function(element) {
    element.className = "";
  });
});
```

Parameters

<code>selector</code>	String	Specify a CSS3 selector representing the elements you wish to pass to the filter.
<code>callback</code>	Function	Specify a function to process the array of matched elements.

Returns

No return value.

See Also

[Filtering Content](#)

predicate

[predicate](#) is a grouping of methods related to predicate based content filtering for a Textbox.io editor instance.

Predicate based filters let you specify a matching function into which all elements passing into (or out of) the editor are passed. This matching function evaluates each element and returns elements that you identify for filtering.

Elements identified and returned by the matching function are passed as an array to a callback function, which operates on the array of identified elements. These operations can include, but are not limited to, filtering, element transformation, etc.

Methods

<code>editor.filters.predicate.addInput()</code>	Creates a filter when content is added to the editor with: <code>textboxio.replace()</code> , <code>editor.content.set()</code> , <code>editor.content.insertHtmlAtCursor()</code> .
<code>editor.filters.predicate.addOutput()</code>	Creates a filter when content is requested from the editor with: <code>editor.content.get()</code> .

See Also

[Filtering Content](#)

editor.filters.predicate.addInput()

Create a predicate based input filter for an editor instance with `editor.filters.predicate.addInput()`.

The predicate based input filter modifies content added to the editor with: `textboxio.replace()`, `editor.content.set()`, `editor.content.insertHtmlAtCursor()`.

Example

`editor.filters.predicate.addInput(matchingFn, callback)`

```
// A matching function identifying elements that are HTML comments and then returning those elements
var comments = function(element) {
  return element.nodeType == 8 || element.nodeName == '#comment';
};

// Create an in filter identifying comments, supplying the matching function highlights the comment text in a
// red span and then removes the original comment
editor.filters.predicate.addInput(comments, function(elements) {
  $(elements).each(function(index, element){
    var textContent = $(element)[0].textContent;
    $(element).after('<span style="background-color:red;">' + textContent + '</span>');
    $(element).remove();
  });
});
```

Parameters

<code>matchingFn</code>	Function	Specify a named function that returns the elements you wish to pass to the callback function.
<code>callback</code>	Function	A function to process the array of matched elements.

Returns

No return value.

See Also

[Filtering Content](#)

editor.filters.predicate.addOutput()

Create a predicate based output filter for an editor instance with `editor.filters.predicate.addOutput()`.

The predicate based output filter modifies content when requested from the editor with: `editor.content.get()`.

Example

`editor.filters.predicate.addOutput(matchingFn, callback)`

```
// A matching function identifying elements that are not allowed in the output
var disallowed = function(element) {
    var name = element.nodeName.toLowerCase()
    return name === 'iframe' || name === 'embed';
};
// Create an out filter identifying tags that are not allowed and removing them
editor.filters.predicate.addInput(disallowed, function(elements) {
    $(elements).remove();
});
```

Parameters

<code>matchingFn</code>	Function	Specify a named function that returns the elements you wish to pass to the callback function.
<code>callback</code>	Function	A function to process the array of matched elements.

Returns

No return value.

See Also

[Filtering Content](#)

editor.focus

Focus an editor instance with `editor.focus()`.

Example

`editor.focus()`

```
// Focus an editor  
editor.focus();
```

Returns

No return value.

editor.macros

`editor.macros` is a grouping of methods related to content macros for a Textbox.io editor instance.

Macros allow your application to be aware of content entered into Textbox.io. Macros listen for patterns in user entered text, and then take action when constraints are met that result in a match. Potential matches are evaluated each time the user presses the space or return keys.

Methods

<code>editor.macros.addSimpleMacro()</code>	Adds a simple content macro to an editor instance.
<code>editor.macros.removeMacro()</code>	Removes a macro from an editor instance.

See Also

[Macros: Writing Content-Aware Code](#)

editor.macros.addSimpleMacro()

Create a simple content macro for an editor instance with `editor.macros.addSimpleMacro()`.

The simple replacement macro allows you to replace entered text when that text is surrounded by a `startString` and `endString`. The callback function can then operate on the matched string, including the `startString` and `endString`.

Example

`editor.macros.addSimpleMacro(startString, endString, callback)`

```
// Create a macro that replaces text surrounded in double-brackets with a string
var key = editor.macros.addSimpleMacro('[[' , ']]', function(str) {
    var newValue = "Macro Triggered!";
    return newValue;
});
```

Parameters

<code>startString</code>	String	A string pattern to search for signifying the beginning of the macro replacement.
<code>endString</code>	String	A string pattern to search for signifying the end of the macro replacement.
<code>callback</code>	Function	A function that receives the complete matching string (including start and end) and returns the string to insert instead.

Returns

<code>key</code>	String	A key that identifies this macro in this editor instance.
------------------	--------	---

See Also

[Macros: Writing Content-Aware Code](#)

editor.macros.removeMacro()

Remove a previously added macro by passing its key to `editor.macros.removeMacro()`.

Example

```
// Remove a previously added macro by key
editor.macros.removeMacro('macro_1234567');
```

Parameters

key	String	Key string returned when a macro is created via <code>editor.macros.addSimpleMacro()</code> . No error is thrown if the key does not match a previously added macro.
-----	--------	---

Returns

No return value.

editor.message

Display a message banner in the editor UI with `editor.message()`.

A message banner may be of one of three types: info (blue), warning (orange), or error (red). If you so choose, you may display more than one message at a time.



Example

`editor.message(type, timeout, message)`

```
// Replace the element with id 'replaceMe'
var editor = textboxio.replace( '#replaceMe' );

// Display an informational message in the editor UI for 5 seconds
editor.message('info', 5000, 'This editor is ready to use.');
```

Parameters

type	String	Specify the message type: info, warning, or error.
timeout	Integer	Specify time in msec before the editor message is automatically dismissed. <i>Note: Specifying 0 will cause the message to be displayed indefinitely.</i>
message	String	The text to be displayed in the message.

Returns

The message function returns an object with a `hide` function. This can be used to create a message that is hidden by some other action, instead of by a timeout.

Example

`editor.message(type, timeout, message)`

```
// Replace the element with id 'replaceMe'
var editor = textboxio.replace( '#replaceMe' );

// Display a warning message that doesn't hide automatically
var msg = editor.message('warning', 0, 'Connection lost');

... later ...

// Hide the message
msg.hide();
```

editor.mode

The editor's *mode* object provides functions to switch the editor into code and design views respectively

Methods

<code>editor.mode.get()</code>	Retrieves the HTML contents of an instance's content <code><body></code> element.
<code>editor.mode.set(String mode)</code>	Sets the editor mode type to either <code>code</code> or <code>design view</code>

editor.mode.get()

Gets the current editor mode

Example

```
editor.content.set(html)
```

```
// Get the editor mode  
var mode = editor.mode.get()
```

Parameters

none.

Returns

String - `code` if the editor is in `code mode`, `design` if the editor is `design mode`.

editor.mode.set(mode)

Programmatically sets the editor's mode to either `code` or `design` mode.

Example

```
// put the editor instance explicitly into code mode
editor.mode.set('code');

//put the editor instance explicitly into design mode
editor.mode.set('design');
```

Parameters

mode	String	Either <code>code</code> or <code>design</code> for code or design views respectively. If any other values are used, an error is thrown.
------	--------	--

Returns

No return value.

editor.restore

Remove an existing Textbox.io editor from the page, restoring the original element container. Content modified using Textbox.io is preserved within the restored container.

Example

editor.restore()

```
// Replace the element with id 'replaceMe'  
var editor = textboxio.replace( '#replaceMe' );  
// Remove the Textbox.io editor  
var element = editor.restore();
```

Returns

element	HTML element	The element that was previously replaced by Textbox.io. Returned element contents are updated with changes made during an editing session.
---------	--------------	--

textboxio

The Textbox.io Editor API starts with the `textboxio` JavaScript global. The `textboxio` global is available after the editor JavaScript file is loaded on a page.

The `textboxio` global lets you create, modify, and interact with instances of the Textbox.io Rich Text Editor.

Methods

<code>replace</code>	Replaces a DOM element with a Textbox.io Editor instance.
<code>replaceAll</code>	Replaces DOM elements with Textbox.io Editor instances.
<code>inline</code>	Creates an inline editable instance from a DOM element
<code>inlineAll</code>	Creates a list of inline editable instances from a JQuery-style selector string or list of DOM elements
<code>get</code>	Retrieves editor instances.
<code>getActiveEditor</code>	Retrieves the active (last focused) editor instance.
<code>isSupported</code>	Identifies whether Textbox.io supports the current browser.
<code>version</code>	Identifies the version of the Textbox.io Editor client.

Options

<code>config</code>	Configuration item for Textbox.io Editor instances created with <code>replace</code> or <code>replaceAll</code> .
---------------------	---

get

Retrieve editor instances using `get()`. Note that you must pass a [CSS3 selector](#) identifying original elements that have been replaced by Textbox.io, in the same manner as [replace](#) and [replaceAll\(\)](#). The elements returned by the selector are compared against the active editors, and where an element has been replaced by an editor that editor is returned.

Example

`textboxio.get(selector)`

```
// Retrieve all editors whose original elements have the css class 'alpha'
var editors = textboxio.get( '.alpha' );

// Identify the first editor in the returned array.
var editor = editors[0];
```

Parameters

<code>selector</code>	String	Specify a CSS3 selector representing the element or elements that contain Textbox.io editors.
-----------------------	--------	---

Returns

textboxio.editor	Array	An array of Textbox.io editor instances
----------------------------------	-------	---

getActiveEditor

Retrieve the last active editor (the editor that was last given focus) using `getActiveEditor()`.

Example

`textboxio.getActiveEditor()`

```
// Retrieve the last editor used  
var activeEditor = textboxio.getActiveEditor();
```

Returns

<code>textboxio.editor</code>	Object	<p>The last active editor instance.</p> <ul style="list-style-type: none">• If no editor is active: returns the editor last given focus.• If no editors have been focused: returns the first editor created.• If no editor has been created: returns null.• If the active editor is removed: returns null until another editor is given focus.
-------------------------------	--------	---

inline

Creates an inline editor instance from the provided DOM element or jQuery selector string. The initial content of the editor is set to the content/value of the element it makes editable.

Example

Parameters

<code>element</code>	String or Element	Specify a CSS3 selector representing the element you wish Textbox.io to replace for editing. Note that <code>textarea</code> is not supported in this mode. or The element you wish Textbox.io to make inline-editable.
<code>configuration</code>	Object (optional)	An optional group of key-value pairs that specify options/settings for the Textbox.io instances you are invoking.

Note about selectors



If the selector matches multiple elements, only the first is replaced as per the rules for [querySelector](#).

If it does not match any elements, a JavaScript error will be thrown. If the number of elements that will match is not known, use [replaceAll](#) which does not have this restriction.

Returns

<code>textboxio.editor</code>	Object	A single instance of the Textbox.io editor.
-------------------------------	--------	---

See also:

- [Editor types - Classic vs Inline](#)

inlineAll

Creates editable instances from a list or CSS3 query string of elements. The initial content of the editor is set to the content/value of the replaced element.

Example

```
<div class="editor first">First Content</textarea>
<div class="editor first">Second Content</textarea>
...
// Create a Textbox.io editor by searching for items with the "editor" class
var editors = textboxio.inlineAll( '.editor' ); // returns an array of 2 editor instances
```

Parameters

<code>elements</code>	String or Array	Specify a CSS3 selector representing the elements to be made editable. Note that <code>textarea</code> is not supported in this mode. or Specify an array of DOM elements you wish Textbox.io to make editable.
<code>configuration</code>	Object (optional)	An optional group of key-value pairs that specify options/settings for the Textbox.io instances you are invoking.

Returns

<code>editor</code> []	Array	An array of Textbox.io editor instances. Each element matched with <code>textboxio.replaceAll()</code> creates a new instance of Textbox.io , which is then added to the returned array in order.
----------------------------	-------	---

See also

- [inline\(\)](#)
- [Editor types - Classic vs Inline](#)

isSupported

This function was introduced in Textbox.io release 1.3.1.

Identify whether Textbox.io supports the current browser.

If [this](#) API returns false, all other APIs except [version](#) will throw an exception.

Example

```
textboxio.getActiveEditor()
```

```
// Identify whether Textbox.io is supported on this browser  
var supported = textboxio.isSupported();
```

Returns

textboxio.isSupported	Boolean	Whether or not the browser is supported
-----------------------	---------	---

replace

Create editor instances by replacing `<textarea>` or `<div>` elements with `textboxio.replace()`. The initial content of the editor is set to the content /value of the replaced element.

When Textbox.io replaces a `<textarea>` element within a `<form>`, it will update the original `<textarea>` element when the form is submitted, supplying updated content as part of the `<form>` POST.

When Textbox.io replaces `<div>` elements, content must be requested from Textbox.io using a JavaScript API, see: [Setting and Getting Content](#).

Example

textboxio.replace(selector, [configuration])

```
<div id="replaceMe">Content</div>
...
// Create a Textbox.io editor by searching for a DOM element with id 'replaceMe'
var simpleEditor = textboxio.replace( '#replaceMe' );
```

textboxio.replace(element, [configuration])

```
<div id="replaceMe">Content</div>
...
// Create a Textbox.io editor by replacing a specific DOM element
var div = document.getElementById( 'replaceMe' );
var simpleEditor = textboxio.replace( div );
```

Parameters

selector or element	String or Element	Specify a CSS3 selector representing the <code><div></code> or <code><textarea></code> element you wish Textbox.io to replace for editing. or The <code><div></code> or <code><textarea></code> element you wish Textbox.io to replace for editing.
configuration	Object (optional)	An optional group of key-value pairs that specify options/settings for the Textbox.io instances you are invoking.

Note about selectors



If the selector matches multiple elements, only the first is replaced as per the rules for [querySelector](#).

If it does not match any elements, a JavaScript error will be thrown. If the number of elements that will match is not known, use [replaceAll](#) which does not have this restriction.

Returns

textboxio.editor	Object	A single instance of the Textbox.io editor.
----------------------------------	--------	---

replaceAll

Create editor instances by replacing `<textarea>` or `<div>` elements with `textboxio.replaceAll()`. The initial content of the editor is set to the content/value of the replaced element.

When Textbox.io replaces a `<textarea>` element within a `<form>`, it will update the original `<textarea>` element when the form is submitted, supplying updated content as part of the `<form>` POST.

When Textbox.io replaces `<div>` elements, content must be requested from Textbox.io using a JavaScript API, see: [Get Editor Content](#).

Example

`textboxio.replaceAll(selector, [options])`

```
<textarea id="editor1">First Content</textarea>
<textarea id="editor2">Second Content</textarea>
...
// Create a Textbox.io editor by searching for textareas
var editors = textboxio.replaceAll( 'textarea' );
```

`textboxio.replaceAll(elements, [options])`

```
<div id="editor1">First Content</div>
<textarea id="editor2">Second Content</textarea>
...
// Create a Textbox.io editor by replacing the specific DOM elements
var div = document.getElementById( 'editor1' );
var textarea = document.getElementById( 'editor2' );
var editors = textboxio.replaceAll( [div, textarea] );
```

Parameters

<code>selector</code>	String	Specify a CSS3 selector representing the <code><div></code> or <code><textarea></code> element or elements you wish Textbox.io to replace for editing.
or	or	or
<code>elements</code>	Array	Specify an array of the <code><div></code> or <code><textarea></code> elements you wish Textbox.io to replace for editing.
<code>configuration</code>	Object (optional)	An optional group of key-value pairs that specify options/settings for the Textbox.io instances you are invoking.

Returns

<code>textboxio.editor</code>	Array	An array of Textbox.io editor instances. Each element matched with <code>textboxio.replaceAll()</code> creates a new instance of Textbox.io, which is then added to the returned array in order.
-------------------------------	-------	--

version

Identify the version of the Textbox.io Editor client using `textboxio.version()`.

Example

`textboxio.getActiveEditor()`

```
// Identify the version of Textbox.io in use
var version = textboxio.version();
```

Returns

<code>textboxio.version</code>	Object	A Textbox.io version object containing keys for values of <code>major</code> , <code>minor</code> , and <code>full</code> version number strings.
--------------------------------	--------	---

Server-Side Components

Textbox.io comes packaged with *optional* server-side components that enhance the functionality of the editor.

These server-side components enable:

- **Enhanced Spellchecking** for improved error detection and suggestions as well as greater UI integration.
- **Extended Image Editing** by allowing editing of images from the web through a proxy.
- **Hyperlink Validation** to ensure that links in the content will resolve.
- **Enhanced Media Embed** for transforming a pasted link into a rich representation of the target location.

The Textbox.io server-side components are compatible with Java Application Servers.

See the [installing the server-side components](#) article for more detail and to get started.

Once you have installed the Textbox.io server-side components you'll need to use the Editor APIs to reference the URL locations of your installed services.

Component	Editor API	Description
Spell Checking	spelling	Spell checking service for JavaScript editors.
Extended Image Editing (Image Proxy)	image.editing	Image proxy to allow editing images from the web.
Hyperlink Checking	links.validation	Hyperlink validity checking service.
Enhanced Media Embed	links.embed	Transform hyperlinks into rich representations of the target location.


Installation and Setup

Some features require the deployment of server-side components onto a Java Servlet 3.0 compatible application server. We currently support Jetty, Apache Tomcat, and WebSphere Application Server.

The following server-side components are packaged with the Textbox.io SDK:

Component	File	Description
Spell Checking	ephox-spelling.war	Spell checking service for JavaScript editors.
Extended Image Editing (Image Proxy)	ephox-image-proxy.war	Image proxy to allow editing images from the web.
Hyperlink Checking	ephox-hyperlinking.war	Hyperlink validity checking service.
Enhanced Media Embed	ephox-hyperlinking.war	Transform hyperlinks into rich representations of the target location.

Allowed Origins Service Depreciated

 **Note:** The "Allowed Origins" service (ephox-allowed-origins.war) has been deprecated. Trusted domains should now be specified directly in the configuration file.


This guide will help you get these server-side components up and running.

1. Install a Java Application Server

If you've already got a Java application server like Jetty or Tomcat installed, skip to Step 2.

If you don't, pick either [Tomcat](#) or [Jetty](#) and install one of these with their default settings using the instructions on their website.

Memory Requirement

 Please ensure that you configure your Java Server (Tomcat/Jetty etc) with a minimum of 4GB. Please refer to the [Out of Memory Errors](#) troubleshooting page if you require instructions on how to explicitly define how much RAM will be allocated to your Java server.

2. Deploy Server-side Components

Deploy all the WAR files that came packaged with the Textbox.io SDK to your newly installed Java application server:

- ephox-spelling.war
- ephox-image-proxy.war
- ephox-hyperlinking.war


The easiest way to deploy these files is to copy them into the `webapps` directory of your Tomcat/Jetty installation and then restart the application server.

More information can be found in the documentation of your chosen application server:

[Deploying applications with Tomcat 9.0](#)
[Deploying applications with Jetty](#)

3. Create a configuration file

Choice of editor

 Use a plain text editor (such as gedit, vim, emacs or notepad) when creating or editing the `application.conf` file. Do not use word processors like Microsoft Word or Evernote as these can insert extra characters which make the file unreadable to the server-side components.

The Textbox.io server-side components require a configuration file to function correctly. By convention, this file is named `application.conf`.

The SDK comes packaged with an example configuration file (`examples/sample_application.conf`) that can be used as a reference guide. You can use this example file (after modifying it with your settings). We recommend that you make a backup of the file before editing it.

This configuration file will require you to enter **at least** the following information:

- `allowed-origins` - the domains allowed to communicate with the server-side editor features. This is required by all server-side components.

Some server-side components require additional configuration which can be found in their individual documentation:

- [Enhanced Media Embed](#)

allowed-origins (required)

Textbox.io editor instances make use of the server-side components by performing a cross-origin HTTP request. These requests are subject to a form of HTTP access control called Cross-Origin Resource Sharing (CORS). CORS is built into web browsers and is not a feature of Textbox.io's server side components. A detailed explanation of CORS can be found on the [Mozilla Developer Network](#).

The `allowed-origins` element configures a list of **all** values that can be expected by the server-side components in a HTTP Origin header from your Textbox.io instances (see the [Mozilla Developer Network](#) for more information on the HTTP Origin header). In short, you'll need to supply a list of all the URLs that your Textbox.io instances will be served from without the path information.

This is best illustrated with some examples:

If users load Textbox.io from the following URLs:

- `http://server.example.com/editor.php`
- `http://server.example.com/subpage/editor.php`

Add `http://server.example.com` to the `allowed-origins` list.

If users load Textbox.io from the following URLs:

- `https://server.example.com/editor.php`
- `http://server.example.com/subpage/editor.php`

Add `http://server.example.com` and `https://server.example.com` to the `allowed-origins` list.

If users load Textbox.io from the following URLs:

- `https://server.example.com/editor.php`
- `https://server.example.com/`

Add `https://server.example.com` to the `allowed-origins` list.

If users load Textbox.io from the following URLs:

- `http://oneserver.example.com/editor.php`
- `http://twoserver.example.com/subpage/editor.php`

Add `http://oneserver.example.com` and `http://twoserver.example.com` to the `allowed-origins` list.

Note



If some of your URLs include a port then add an entry with and without the port. The value of the `Origin` header may be different across browsers. Add both to be safe.

If users load Textbox.io from the following URLs:

- `http://server.example.com:8080/editor.php`

Add `http://server.example.com:8080` and `http://server.example.com` to the `allowed-origins` list.

If users load Textbox.io from the following URLs:

- `https://server.example.com:9000/editor.php`

Add `https://server.example.com:9000` and `https://server.example.com` to the `allowed-origins` list.

element	<code>allowed-origins</code>	Stores CORS setup information
attribute	<code>origins</code>	An array of strings containing all possible values of the HTTP Origin header the server-side components can expect.

Example:

allowed-origins example

```
ephox {
  allowed-origins {
    origins = [ "http://myserver", "http://myserver.example.com", "http://myserver:8080",
"http://myotherserver", "http://myotherserver:9090", "https://mysecureserver" ]
  }
}
```

Wildcard support

The * wildcard character matches any value. Wildcards are supported in the following parts of entries in the `allowed-origin` list:

1. The scheme (e.g. `*://mydomain.com`). Omitting the scheme entirely is equivalent (e.g. `mydomain.com`).
2. The port (e.g. `http://mydomain.com:*`).
3. As a prefix of the domain (e.g. `http://*.mydomain.com`).
4. Any combination of scheme, port, and domain prefix (e.g. `*://*.mydomain.com:*`).
5. As the only character (e.g. `*`). This will allow any Origin to access the server-side components.
6. As the only character after the scheme (e.g. `https://*`). This will allow **any** Origin serving Textbox.io from a HTTPS page to access the server-side components.

Wildcards

Options 5 and 6 allow a broad set of origins access to the server-side components and are **not** recommended for production deployments.

allowed-origins with wildcards example

```
ephox {
  allowed-origins {
    origins = [ "http://myserver:*", "*/myotherserver.example.com", "*//*.mydomain.example.com:
*" ]
  }
}
```

Troubleshooting Origins

If you missed an Origin or specified an Origin incorrectly, Textbox.io features that rely on the server-side components will not work from that Origin. If you observe that requests to the server-side components are failing or features are unavailable and you're not sure why, refer to the troubleshooting information about [Investigating Using the Browser's Network Tools](#).

link-checking.enabled (optional)

This element enables or disables the hyperlinking feature. Valid values are `true` and `false`. If not set, the service is enabled by default.

link-checking.cache (optional)

This element configures the hyperlinking service's built-in cache. When a hyperlink is checked and confirmed valid, the result is cached to save unnecessary network traffic in the future. Default settings are automatically configured, meaning these settings are optional.

The `capacity` attribute sets the capacity of the cache. The default setting is 500.

The `timeToLiveInSeconds` attribute sets the time-to-live of elements of the cache, measured in seconds. The default setting is 86400 seconds, which is one day.

The `timeToIdleInSeconds` attribute sets the time-to-idle of elements of the cache, measured in seconds. The default setting is 3600 seconds, which is one hour.

element	link-checking.cache	Stores cache settings for the hyperlink checker.
---------	---------------------	--

attribute	capacity	An integer defining the maximum number of elements stored in the cache at any one time.
attribute	timeToLiveInSeconds	An integer defining the time-to-live of the cache, measured in seconds. This is the maximum total amount of time that an element is allowed to remain in the cache.
attribute	timeToIdleInSeconds	An integer defining the time-to-idle of the cache, measured in seconds. This is the maximum amount of time that an element will remain in the cache if it is not being accessed.

Example

link-checking Example
<pre>ephox { link-checking { enabled = true cache { capacity = 500 timeToLiveInSeconds = 86400 timeToIdleInSeconds = 3600 } } }</pre>

proxy (optional)

This element configures use of an HTTP proxy for outgoing HTTP/HTTPS requests made by the server-side components.

Default proxy settings are picked up from JVM system properties, usually provided on the command line, as defined in "[Networking Properties for Java](#)". The system properties `http.proxyHost`, `http.proxyPort`, `http.nonProxyHosts`, `https.proxyHost`, `https.proxyPort` are recognized as well as `http.proxyUser` and `http.proxyPassword` to support authenticating proxies.

This optional proxy element provides an alternative to providing proxy settings as JVM system properties, or to override system properties.

element	proxy	Stores HTTP outgoing proxy settings for the server-side components.
attribute	http.proxyHost	A string defining the host name of the proxy for plain HTTP (not HTTPS) connections. (Mandatory)
attribute	http.proxyPort	An integer defining the port number of the proxy for plain HTTP (not HTTPS) connections. (Mandatory)
attribute	http.nonProxyHosts	A list of strings separated by vertical lines (" ") listing hosts and domains to be excluded from proxying, for both plain HTTP and HTTPS connections. The strings can contain asterisks ("*") as wildcards. (Optional, defaults to "localhost 127.*[::1]" if not set.)
attribute	https.proxyHost	A string defining the host name of the proxy for HTTPS connections. (Optional)
attribute	https.proxyPort	An integer defining the port number of the proxy for HTTPS connections. (Optional)
attribute	http.proxyUser	Username for authenticating to both the HTTP and HTTPS proxy. (Optional)
attribute	http.proxyPassword	Password for authenticating to both the HTTP and HTTPS proxy. (Optional)

Example

proxy Example

```
ephox {
  proxy {
    http.proxyHost = someproxy.example.com
    http.proxyPort = 8080
    https.proxyHost = someproxy.example.com
    https.proxyPort = 8080
    http.nonProxyHosts = localhost|*.example.com
  }
}
```

http (optional)

element	http	HTTP Configuration for linkchecking and media embedding.
attribute	max-redirects	The maximum number of redirects that will be followed to check a hyperlink or retrieve open graph details from that resource. The default value is 10 redirects before giving up on the resource.
attribute	request-timeout-seconds	An integer defining the number of seconds to allow HTTP requests to take. Default: 10
attribute	trust-all-cert	A boolean indicating whether to bypass SSL security and indiscriminately trusts all SSL certificates. Default: false

Some server-side components make outbound HTTP and HTTPS connections. These include Link Checker, Enhanced Media Embed and Image Tools Proxy. In an evaluation or pre-production environment, you might want to test these features against resources with untrusted SSL certificates such as in-house servers with self-signed SSL certificates. In these circumstances, it is possible to bypass all SSL security.

This is not recommended for production environments.

trust-all-cert-example

```
ephox {
  http {
    trust-all-cert = true
  }
}
```

There are some additional http settings that apply to the hyperlinking service when it is following HTTP redirects (for link checks and open graph style embeds) as well as the image proxy service.

redirects example

```
ephox {
  http {
    max-redirects = 20
    request-timeout-seconds = 10
  }
}
```

image-proxy (optional)

The image proxy service has some optional configuration to set a maximum size for images proxied. Images beyond this size it will not be proxied. Please note that the `http.request-timeout-seconds` above also applies to requests made by the image proxy service.


element	image-proxy	Configures image proxy behaviour.
attribute	size-limit	An integer defining the maximum allowed image size in bytes. Default: 10000000

redirects example

```
ephox {
  image-proxy {
    size-limit = 10000000
  }
}
```

4. Pass the configuration file to the Java application server

HTTP Proxy

 If you are relying on an HTTP proxy for outgoing HTTP/HTTPS connections to the Internet, consider configuring use of the proxy by the application server by setting JVM system properties at this point. These can be set in the same manner as `config.file` using the instructions below (using the `-D` option to the `java` command). Please refer to "[Networking Properties for Java](#)" for details. The system properties `http.proxyHost`, `http.proxyPort`, `http.nonProxyHosts`, `https.proxyHost`, `https.proxyPort` are recognized as well as `http.proxyUser` and `http.proxyPassword` to support authenticating proxies.

Alternatively, use of a proxy for server-side components can be set directly in their configuration file as discussed [above](#).

Tell the services about the configuration file by setting the `config.file` JVM system property to the absolute path of the configuration file. The exact method for doing this varies depending on your operating system, application server and whether the application server is being run as a system service. The authoritative reference for configuring any application server is the vendor documentation, but we'll do our best to get you started below.

Windows

All Windows examples will assume the name of your configuration file is `application.conf` and it is located in the directory `C:\config\file\location\`. You'll need to set the JVM system property `-Dconfig.file=C:\config\file\location\application.conf`.

Tomcat

From the command line

The following assumes you've downloaded the Tomcat 9.0 zip archive from the Tomcat website, unpacked it and you're working from the unpacked Tomcat directory.

Create or edit the script `.\bin\setenv.bat` to contain the following line:

Example setenv.bat

```
set "CATALINA_OPTS= -Dconfig.file=C:\config\file\location\application.conf"
```

There should only be a single line in this file defining the `CATALINA_OPTS` environment variable.

You may also need to add another line with the path to your Java Runtime Environment installation (replace with the actual path on your system) such as:

Example setenv.bat

```
set "JRE_HOME=C:\Program Files\Java\jre1.8.0_131"
```

After editing `setenv.bat`, run the following command to start Tomcat:

Starting Tomcat

```
.\bin\startup.bat
```

For further information see the documentation on [running Tomcat 9.0](#).

As a Windows service

If you download the Windows installer, Tomcat 9.0 will always be installed as a Windows system service. See the notes on [Windows setup](#) for Tomcat 9.0 and the instructions for setting JVM system properties in the [Tomcat 9.0 Windows Service HOW-TO](#).

As a minimal example, if the installer installed Tomcat to `C:\Program Files\Apache Software Foundation\Tomcat 9.0\` (default option):

- Run `C:\Program Files\Apache Software Foundation\Tomcat 9.0\bin\Tomcat9w` which opens the **Apache Tomcat 9.0 Tomcat9 Properties** dialog box
- Select the `Java` tab
- Add the following line to `Java Options`:

Java Options

```
-Dconfig.file=C:\config\file\location\application.conf
```

For other versions of Tomcat on Windows, check the Tomcat documentation for that version.

Jetty

From the command line

If you're following the instructions for [Starting Jetty](#) for Jetty 9.4.5, the path to the configuration file can simply be supplied as a command option:

Starting Jetty

```
java -D"config.file=C:\config\file\location\application.conf" -jar C:\jetty\install\directory\start.jar
```

For other versions of Jetty on Windows, check the Jetty documentation for that version.

As a Windows service

Follow the instructions in [Startup via Windows Service](#) for Jetty 9.4.5. Remember to append the following snippet to the line beginning with `set PR_JVMOPTIONS` in your `install-jetty-service.bat` script:

Append "set PR_JVMOPTIONS" line in install-jetty-service.bat

```
; -Dconfig.file="C:\config\file\location\application.conf"
```

Note


 Check the `install-jetty-service.bat` has the correct paths to your Java installation. The service will fail to start with some rather unhelpful errors if the paths are incorrect.

For other versions of Jetty on Windows, check the Jetty documentation for that version.

Linux

All Linux examples will assume the name of your configuration file is `application.conf` and it is located in the directory `/config/file/location/`. You'll need to set the JVM system property `-Dconfig.file=/config/file/location/application.conf`.

Note

 If the path to your `application.conf` file has spaces in it, you must ensure you prefix each white space with an escape character (`\`).

Example: `-Dconfig.file=/config/file/location/with/white\ space/application.conf`

Tomcat and/or Jetty can be obtained via the package manager for many Linux distributions. The commands for starting the service and the location of the configuration files will vary across distributions. If you installed an application server via the package manager, follow your distribution's documentation for configuring it.

Tomcat

The following assumes you've downloaded Tomcat 9.0 from the Tomcat website and unpacked the archive to `/opt/tomcat`.

For other versions of Tomcat on Linux, check the Tomcat documentation for that version.

If you've obtained Tomcat from your distribution's package manager, refer to your distribution's documentation for Tomcat.

From the command line

Create or edit the script `/opt/tomcat/bin/setenv.sh` to contain the following line:

Example `setenv.sh`

```
CATALINA_OPTS=" -Dconfig.file=/config/file/location/application.conf "
```

There should only be a single line in this file defining the `CATALINA_OPTS` environment variable.

After editing `setenv.sh`, run the following command to start Tomcat:

Starting Tomcat

```
/opt/tomcat/bin/startup.sh
```

For further information see the documentation on [running Tomcat 9.0](#).

Jetty

The following assumes you've downloaded Jetty 9.4.5 from the Jetty website and unpacked the archive to `/opt/jetty`.

For other versions of Jetty on Linux, check the Jetty documentation for that version.

If you've obtained Jetty from your distribution's package manager, refer to your distribution's documentation for Jetty.

From the command line

The path to the configuration file can simply be supplied as a command option:

Starting Jetty

```
java -Dconfig.file="/config/file/location/application.conf" -jar /opt/jetty/start.jar
```

As a Linux service

Assuming you've followed the instructions to [Startup a Unix Service using jetty.sh](#) for Jetty 9.4.5, edit `/etc/default/jetty` and add the line:

Example `/etc/default/jetty`

```
JETTY_ARGS=" -Dconfig.file=/config/file/location/application.conf "
```

There should only be a single line in this file defining the `JETTY_ARGS` variable.

5. Restart the Java application server

After you've completed the steps on this page to [Deploy server-side components](#), [Create a configuration file](#) and [Pass the configuration file to the Java application server](#), the application server may need to be restarted to pick up all your changes. Turn it off and on again now, just to be safe.

6. Set up editor client instances to use the server-side functionality

Now that the server-side components deployed and running, you'll need to tell your Textbox.io instances where to find them:

- Set the [spelling.url](#) configuration property to the URL of the deployed server-side spelling component.
- Set the [images.editing.proxy](#) configuration property to the URL of the deployed server-side image proxy component. Note that this URL must include the full path as shown below, including the `/image path`.
- Set the [links.validation.url](#) and [links.embed.url](#) configuration properties to the URL of the deployed server-side link-checking and enriched media embed component.

This example assume your Java application server is running on port 80 ([http](#)) on `yourserver.example.com` and that all the server-side components are deployed to the same Java application server. Replace `yourserver.example.com` with the actual domain name or IP address of your server.

Example

Textbox.io Client Config

```
var config = {
  spelling : {
    url: 'http://yourserver.example.com/ephox-spelling/' // Spelling service base URL
  },
  images : {
    editing : {
      proxy: 'http://yourserver.example.com/ephox-image-proxy/image' // Image proxy service URL
    }
  },
  links : {
    validation : {
      url : 'http://yourserver.example.com/ephox-hyperlinking/' // Link-Checking service URL
    }
  },
  links : {
    embed : {
      url : 'http://yourserver.example.com/ephox-hyperlinking/' // Enriched Media Embed service URL
    }
  }
};

var editor = textboxio.replace('#id', config);
```

[links.embed.url](#)

Logging

It may be useful to make the Textbox.io server-side components write to their own log file. This can assist in troubleshooting and make it easier to provide logs as part of a support ticket.

To write the logs to a specific file, you'll need to perform the following steps:

Step 1. Create a logging configuration XML file

The Textbox.io services use the [Logback](#) logging format.

Save the snippet below as `logback.xml` after replacing `{ $LOG_LOCATION }` with the full path to the destination log file (e.g. `/var/log/textboxio_server_components.log`).

```
<configuration>

  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>
    </encoder>
  </appender>

  <appender name="FILE" class="ch.qos.logback.core.FileAppender">
    <file>{ $LOG_LOCATION}</file>
    <encoder>
      <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>
    </encoder>
  </appender>

  <!-- The name "com.ephox" refers to all the Textbox.io server-side components. -->
  <logger name="com.ephox" level="INFO"/>

  <root level="INFO">
    <appender-ref ref="FILE" />
    <!-- If you want logging to go to the container as well uncomment
    the following line -->
    <!-- <appender-ref ref="STDOUT" /> -->
  </root>

</configuration>
```

Step 2. Pass the configuration file to the Java application server

Assuming you've saved your `logback.xml` file in `/etc/opt/textbox`, follow step 4 and step 5 on the [Installation and Setup](#) page to set the following JVM system property on your Java application server:

Logging configuration file JVM property

```
-Dlogback.configurationFile=/etc/opt/textbox/logback.xml
```

Embed Rich Media

The [Enhanced Media Embed](#) plugin makes it easy to add an enhanced content creation experience in your website or app with enriched media previews from the most popular web sources. Facebook, YouTube, Flickr, NY Times, Vimeo, Hulu, Tumblr, CodePen, SlideShare, TechCrunch, WordPress, Twitch, Spotify ... and many more!

By easy, we mean easy. The service automatically “looks behind the link” to see whether a URL in the Textbox.io editor points to a rich media source. That URL will resolve into an enhanced media thumbnail whenever possible.


It is now as simple as adding a plugin to deliver a modern, content creation experience that everyone takes for granted.

Enhanced Media Embed Quick Setup

Once you've got the server-side component installed, additional configuration to your `application.conf` file is required. (Don't forget to restart the Java application server after updating the configuration.)

You also need to [integrate](#) and [configure](#) the Enhanced Media Embed server.

WebSphere Application Support

 The Enhanced Media Embed server currently does not support integration with IBM WebSphere Application Server.

Configure Enhanced Media Embed Server

Once you've got the [server-side component](#) installed, additional configuration to your `application.conf` file is required. (Don't forget to restart the Java application server after updating the configuration.)

The Enhanced Media Embed service allows you to choose between using your own [Iframely](#) account, configuring custom [oEmbed](#) endpoints or using a combination of both.

When you insert media into your content, the service will do the following (in order):

1. Check if the URL matches any custom oEmbed configuration. If that fails,
2. If Iframely is configured, query the Iframely API. If Iframely is not configured,
3. Create a summary card.

`embed.enabled` (optional)

This element enables or disables the Enhanced Media Embed feature. Valid values are `true` and `false`. If not set, the service is enabled by default.

```
ephox {
  embed {
    enabled = true
  }
}
```

Use your own Iframely account

To use your own Iframely account, provide the following configuration items:

- `enabled` - set to `true` to use Iframely. If set to `false`, you don't need to provide the `base-url` or `api-key` configuration items.
- `base-url` - the base URL of the Iframely API. Check their [docs](#) if you're unsure.
- `api-key` - your Iframely API key. This is provided by Iframely after setting up an account with them.

Example with Iframely enabled (replace `xxx` with your Iframely API key):

```
ephox {
  embed {
    iframely {
      enabled = true
      base-url = "https://iframe.ly/api/iframe"
      api-key = "xxx" // change this to your own Iframely API key
    }
  }
}
```

Example with Iframely disabled:

```
ephox {
  embed {
    iframely {
      enabled = false
    }
  }
}
```

Configure a custom endpoint

The service can be configured to hit a specific oEmbed endpoint when media from a URL matching a provided pattern is inserted into your content.

- `endpoint` - the URL of the oEmbed endpoint that should be consulted when inserting media with a URL that matches an entry in `schemes`.
- `schemes` - a list of schemes as defined in [Section 2.1. Configuration](#) of the oEmbed specification. Note that HTTP and HTTPS are two separate schemes.

Example note: This configuration is provided as an example only. The Enhanced Media Embed service converts an oEmbed response into an embeddable snippet of code. The content and quality of the snippet is dependent on the oEmbed response.

Example:

```

ephox {
  embed {
    custom = [
      # youtube
      {
        endpoint = "http://www.youtube.com/oembed"
        schemes = [
          "http://youtu.be/*",
          "https://youtu.be/*",
          "http://www.youtu.be/*",
          "https://www.youtu.be/*",
          "http://youtube.com/*",
          "https://youtube.com/*",
          "http://www.youtube.com/*",
          "https://www.youtube.com/*",
          "http://m.youtube.com/*",
          "https://m.youtube.com/*"
        ]
      },
      # NY Times
      {
        endpoint = "https://www.nytimes.com/svc/oembed/json/"
        schemes = [
          "http://www.nytimes.com/*",
          "https://www.nytimes.com/*"
        ]
      },
      # Daily Motion
      {
        endpoint = "http://www.dailymotion.com/services/oembed"
        schemes = [
          "http://www.dailymotion.com/video/*"
          "https://www.dailymotion.com/video/*"
          "http://www.dailymotion.com/embed/video/*"
          "https://www.dailymotion.com/embed/video/*"
        ]
      },
      # Soundcloud
      {
        endpoint = "http://soundcloud.com/oembed"
        schemes = [
          "http://soundcloud.com/*",
          "https://soundcloud.com/*",
          "http://api.soundcloud.com/tracks/*",
          "https://api.soundcloud.com/tracks/*"
        ]
      },
      # Facebook post
      {
        endpoint = "https://www.facebook.com/plugins/post/oembed.json/"
        schemes = [
          "http://*.facebook.com/permalink.php*",
          "https://*.facebook.com/permalink.php*",
          "http://*.facebook.com/photo.php*",
          "https://*.facebook.com/photo.php*",
          "http://*.facebook.com/*/photos/*",
          "https://*.facebook.com/*/photos/*",
          "http://*.facebook.com/*/posts/*",
          "https://*.facebook.com/*/posts/*",
          "http://*.facebook.com/*/activity/*",
          "https://*.facebook.com/*/activity/*",
          "http://*.facebook.com/notes/*",
          "https://*.facebook.com/media/set*"
        ]
      },
      # Facebook Video
      {
        endpoint = "https://www.facebook.com/plugins/video/oembed.json/"
        schemes = [
          "http://www.facebook.com/video*",

```

```

        "https://www.facebook.com/video*",
        "http://www.facebook.com/*/videos/*",
        "https://www.facebook.com/*/videos/*",
        "http://business.facebook.com/video*",
        "https://business.facebook.com/video*",
        "http://business.facebook.com/*/videos/*",
        "https://business.facebook.com/*/videos/*"
    ]
},
# Facebook Page
{
    endpoint = "https://www.facebook.com/plugins/page/oembed.json/"
    schemes = [
        "http://www.facebook.com/*",
        "https://www.facebook.com/*",
        "http://m.facebook.com/*",
        "https://m.facebook.com/*"
    ]
},
# Spotify
{
    endpoint = "https://embed.spotify.com/oembed/"
    schemes = [
        "http://spotify.com/*",
        "https://spotify.com/*",
        "http://open.spotify.com/*",
        "https://open.spotify.com/*",
        "http://embed.spotify.com/*",
        "https://embed.spotify.com/*",
        "http://play.spotify.com/*",
        "https://play.spotify.com/*"
    ]
},
# Hulu
{
    endpoint = "http://www.hulu.com/api/oembed.json",
    schemes = [
        "http://www.hulu.com/watch/*",
        "https://www.hulu.com/watch/*"
    ]
},
# Vimeo
{
    endpoint = "http://vimeo.com/api/oembed.json",
    schemes = [
        "http://vimeo.com/*",
        "https://vimeo.com/*",
        "http://www.vimeo.com/*",
        "https://www.vimeo.com/*"
    ]
},
# SmugMug
{
    endpoint = "http://api.smugmug.com/services/oembed/"
    schemes = [
        "http://*.smugmug.com/*",
        "https://*.smugmug.com/*"
    ]
},
# Slideshare
{
    endpoint = "http://www.slideshare.net/api/oembed/2"
    schemes = [
        "http://*.slideshare.net/*"
    ]
},
# Wordpress
{
    endpoint = "https://public-api.wordpress.com/oembed/1.0/?for=ephox"
    schemes = [
        "http://*.wordpress.com/*",

```

```

    "https://*.wordpress.com/*"
  ]
},
# Meetup
{
  endpoint = "https://api.meetup.com/oembed"
  schemes = [
    "http://www.meetup.com/*",
    "https://www.meetup.com/*",
    "http://meetup.com/*",
    "https://meetup.com/*",
    "http://meetu.ps/*",
    "https://meetu.ps/*"
  ]
},
# Spotify
{
  endpoint = "https://embed.spotify.com/oembed/"
  schemes = [
    "http://open.spotify.com/*",
    "https://open.spotify.com/*",
    "http://play.spotify.com/*",
    "https://play.spotify.com/*"
  ]
},
# Tech crunch
{
  endpoint = "http://public-api.wordpress.com/oembed/1.0/?for=ephox"
  schemes = [
    "http://techcrunch.com/*",
    "https://techcrunch.com/*"
  ]
},
# Dotsub
{
  endpoint = "https://dotsub.com/services/oembed"
  schemes = [
    "http://dotsub.com/view/*",
    "https://dotsub.com/view/*"
  ]
},
# Speaker deck
{
  endpoint = "https://speakerdeck.com/oembed.json"
  schemes = [
    "http://speakerdeck.com/*/*",
    "https://speakerdeck.com/*/*"
  ]
},
# Tumblr
{
  endpoint = "https://www.tumblr.com/oembed/1.0"
  schemes = [
    "http://*.tumblr.com/post/*",
    "https://*.tumblr.com/post/*"
  ]
},
# Adobe Stock
{
  endpoint = "https://stock.adobe.com/oembed"
  schemes = [
    "http://stock.adobe.com/*",
    "https://stock.adobe.com/*"
  ]
},
# Code pen
{
  endpoint = "https://codepen.io/api/oembed"
  schemes = [
    "http://codepen.io/*pen/*",
    "https://codepen.io/*pen/*"
  ]
}

```



```

    ]
  },
  # 500px
  {
    endpoint = "https://500px.com/oembed"
    schemes = [
      "http://500px.com/photo/*",
      "https://500px.com/photo/*"
    ]
  }
]
}
}
}

```

Combining Iframely and custom endpoints

It is also possible to configure Iframely with custom oEmbed endpoints. For example, you may want to use Iframely to embed media from the Internet and an internal oEmbed server to embed media from an Intranet.

Example (replace xxx with your Iframely API key):

```

ephox {
  embed {
    iframely {
      enabled = true
      base-url = "https://iframe.ly/api/iframe.ly"
      api-key = "xxx" // change this to your own Iframely API key
    },

    custom = [
      {
        endpoint = "http://localhost:3000/oembed"
        schemes = [
          "http://intranet.example.com/*"
        ]
      }
    ]
  }
}
}
}

```

Summary cards

If neither Iframely or an oEmbed endpoint is configured for a given URL, a summary card will be created.

A summary card is an embeddable snippet of code which is generated based on what the Enhanced Media Embed service can work out about the content at the URL. See the integration docs for [Enhanced Media Embed Server](#) for further details.

Integrate Enhanced Media Embed Server

Websites like [Facebook](#) and [Instagram](#) expose an [oEmbed](#) endpoint for developers to utilise. The [Iframe.ly](#) service creates embeds from websites on the public Internet. For content on private networks, such as a corporate CMS, this document outlines how to enrich the content or build an API that the Enhanced Media Embed server can utilise to create rich hyperlinks. We'll also provide some information about the Enhanced Media Embed server's summary cards.

There are three options for enhancing the embeds created for private content by the Enhanced Media Embed server:

- Annotate content with [Open Graph](#) or other meta tags
- Develop your own custom endpoint that returns JSON in the oEmbed format
- Develop your own custom endpoint that returns JSON in the Tiny Enhanced Media Embed format

A note on cookies & authentication

If the content or endpoint is on the same domain as the Enhanced Media Embed server, cookies will be forwarded to that server. This should reuse the existing login of the user in a CMS environment.

As an example:

- The Enhanced Media Embed server is accessible at `http://mydomain.example.com/media`
- Your custom oEmbed endpoint is accessible at `http://mydomain.example.com/endpoint`
- You have already signed into your CMS at `http://mydomain.example.com/cms`
- You embed content from `http://mydomain.example.com/cms/secretcontent` that normally requires authentication to access
- Because your content, oEmbed endpoint and Enhanced Media Embed server are all on the same domain, `http://mydomain.example.com/cms/secretcontent` is embeddable using cookie passthrough

Annotate content with Open Graph or other meta tags

Opengraph

If your content is marked up with Open Graph tags and is accessible with a HTTP GET request from the Enhanced Media Embed server, then business card style embeds will be created based on your content.

If [Iframe.ly](#) is enabled in the configuration, then the Open Graph look up will be performed by [Iframe.ly](#). [Iframe.ly](#) requires that the content be publicly accessible on the Internet.

At a minimum, you'll need to define these Open Graph tags:

- `og:title`
- `og:image`

It's a good idea to also define:

- `og:url`
- `og:description`
- `og:site_name`

Additionally, you can specify a video or audio resource to include in the embed by defining:

- `og:video` & `og:video:type` (only MP4 is supported across all browsers)
- `og:audio` & `og:audio:type` (only MP3 is supported across all browsers)

SEO tags

As an alternative to Open Graph tags, you can include meta tags using the older style recommended by search engines such as Google.

- `<title>...</title>`
- `<meta name="description">...</meta>`
- `<link rel=image_src href="..." />`

Pros

- This is the easiest method for creating embeds without an existing embed API
- No server configuration required
- There are existing CMS plugins that will add these tags to content
- The Enhanced Media Embed server will handle creating the embed HTML and styling

Cons

- The page must be accessible to the Enhanced Media Embed server (it must not require authentication)
- The embed HTML and styling created by the Enhanced Media Embed server is not configurable
- Only works for HTML URLs
- [Iframe.ly](#) can only be enabled or disabled as a global option. If [Iframe.ly](#) is enabled, then all content must be world accessible and cookie passthrough will not happen.

Custom API

As an alternative to including meta tags in your content, you can write a custom API that returns JSON in either the oEmbed or Tiny Enhanced Media Embed formats.

See the docs on [configuring a custom endpoint](#) for details on getting the Enhanced Media Embed server to utilise your custom API.

OEmbed endpoint

This is a popular choice and many CMSs have existing plugins that support oEmbed. While you can create custom HTML embeds this way, they cannot contain scripts. If the HTML contains a script, then the Enhanced Media Embed server will embed a [summary card](#).

Pros

- There are existing CMS plugins that support oEmbed
- You can write your own custom HTML

Cons

- No room in this spec for multiple embed representations of the same URL
- Must be a separate API rather than just metadata embedded in the content
- Error messages aren't defined as part of the spec

Tiny Enhanced Media Embed endpoint

The other option for developing a custom API endpoint is to return JSON in the [Tiny Enhanced Media Embed format](#).

Pros

- You can write your own custom HTML
- The format has the ability to house multiple embed representations of the same URL
- Better defined ability to communicate errors to the media server

Cons

- Must be a separate API rather than just metadata embedded in the content
- No support from existing plugins
- The Textbox.io editor does not fully take advantage of this format yet

Tiny Enhanced Media Embed format

HTTP response status codes

- HTTP 200 (OK): `EphoxEmbedObj`
- HTTP 400 (User Error): `ErrorObj`
- HTTP 503 (Upstream Error): `ErrorObj`
- HTTP 500 (Unexpected Error): `ErrorObj`

JSON response objects

`EphoxEmbedObj`

`rel`, `media` and `html` combine to form the default representation of the embeddable resource that your server has chosen. Clients of the Enhanced Media Embed server (such as the Textbox.io editor) can look for alternative representations in `links`.

- `title` (optional)
 - String containing the document title.
- `author_name` (optional)
 - String containing the author's name.
- `author_iri` (optional)
 - String containing an [IRI](#) for the author.
- `provider_iri` (optional)
 - String containing an IRI for the resource provider.
- `provider_name` (optional)
 - String containing the name of the resource provider.
- `short_iri` (optional)
 - String containing a shortened IRI for the resource.
- `canonical_iri` (required)
 - String containing the IRI of the resource.
- `description` (optional)
 - String containing a description of the document.
- `cache_age` (optional)
 - Integer containing the suggested cache lifetime for this resource, in seconds.
- `date` (optional)

- String containing the date of the document in the format **YYYY-MM-DD**.
- **links** (required)
 - [LinksObj](#)
- **rel** (optional)
 - [RelObj](#)
- **media** (optional)
 - [MediaObj](#)
- **html** (optional)
 - String containing the HTML snippet to be embedded by Textbox.io.

RelObj

An array of tags describing the primary type of an embed, where it came from and whether there are any technical attributes that you may want to know about (autoplay, ssl, file format (flash, html5, etc)).

- **primary** (required)
 - Array of [PrimaryRels](#)
- **technical** (required)
 - Array of [TechnicalRels](#)
- **source** (required)
 - Array of [SourceRel](#)

PrimaryRel

A string describing the primary type of an embed containing one of the following values:

- **player**: A video or audio player
- **thumbnail**: A thumbnail representation of the resource
- **image**: A full sized image for the resource
- **reader**
- **file**: No HTML provided. Should just be a hyperlink to a downloadable file.
- **survey**
- **app**: An embed that will switch over to a mobile app if played on a mobile (e.g. soundcloud)
- **summary**: Summary card (scriptless embed)
- **icon**
- **logo**
- **promo**

TechnicalRel

A string describing technical attributes of an embed containing one of the following values:

- **flash**
- **html5**
- **gifv**
- **inline**
- **ssl**
- **autoplay**

SourceRel

A string describing the source of an embed containing one of the following values:

- **iframe**: From Iframely
- **opengraph**: Generated from Open Graph tags in a resource
- **twitter**: Retrieved from a [Twitter Card](#)
- **oembed**: Retrieved from an oEmbed API
- **sm4**
- **fallback**: Tiny fallback embeds that look at SEO tags and Open Graph tags.
- **script_censor**: The original embed (from Iframely or oEmbed) had a script in it and has been converted to a summary card.
- **smartframe_censor**: The original embed had an Iframely smart frame and has been censored into a summary card to avoid a content dependency on Iframely.

LinksObj

Represents all of the possible representations of this resource.

- **players** (required)
 - Array of [LinkObjs](#)
- **thumbnails** (required)
 - Array of [LinkObjs](#)
- **apps** (required)
 - Array of [LinkObjs](#)
- **readers** (required)
 - Array of [LinkObjs](#)
- **surveys** (required)
 - Array of [LinkObjs](#)

- `summary_cards` (required)
 - Array of [LinkObjs](#)
- `icons` (required)
 - Array of [LinkObjs](#)
- `logos` (required)
 - Array of [LinkObjs](#)
- `promos` (required)
 - Array of [LinkObjs](#)
- `images` (required)
 - Array of [LinkObjs](#)
- `files` (required)
 - Array of [LinkObjs](#)

LinkObj

This represents a representation that you could link to / embed.

- `media` (optional)
 - [MediaObj](#)
- `rels` (required)
 - [RelObj](#)
- `href` (optional)
 - String containing the URL of the resource.
- `mime_type` (required)
 - String containing the mime-type of the resource.
- `html` (required)
 - String containing the embeddable HTML snippet.

MediaObj

The media object describes the bounds of the embed. It can either be responsive or fixed.

- `type` (required)
 - String with the value `fixed` or `responsive`

Fields when `type` is `fixed`:

- `width` (required)
 - Integer containing width in pixels.
- `height` (required)
 - Integer containing height in pixels.
- `paddingBottom` (optional)
 - Integer

Fields when `type` is `responsive`:

- `aspectRatio` (optional)
 - Double
- `paddingBottom` (optional)
 - Integer
- `width` (required)
 - [DimensionBoundObj](#)
- `height` (required)
 - [DimensionBoundObj](#)

DimensionBoundObj

The dimension bounds define the height or width of a responsive embed.

- `type` (required)
 - String with the value of `fixed`, `constrained` or `unbounded`

Fields when `type` is `fixed`:

- `pixels` (required)
 - Integer

Fields when `type` is `constrained`:

- `min_pixels` (optional)
 - Integer
- `max_pixels` (optional)
 - Integer

No additional fields when `type` is `unbounded`.

ErrorObj

- `code` (required)
 - Integer with the value of 400 (User Input Error) or 503 (Upstream Failure)
- `subcode` (required)
 - Integer with one of the following values:
 - When `code` is **503**:
 - 1 - Upstream connection issue
 - 2 - Upstream returned not OK
 - 3 - Upstream returned a response that didn't make sense to the server
 - When `code` is **501**:
 - 1 - Support for URI not implemented
 - When `code` is **400**:
 - 1 - URI Failed to parse
 - 2 - URI was relative
 - 3 - URI was empty
 - 4 - URI was not http or https
 - 5 - Max width was not a positive integer
- `msg` (required)
 - A string message for developers / support people.

Summary cards

When the Enhanced Media Embed server generates a summary card of a URL (using the title, thumbnail, description and website), it returns a HTML snippet like the following. You should apply styles to the document style to suit these to your preference.

```
<div class="ephox-summary-card">
  <a class="ephox-summary-card-url-thumbnail" href="http://www.imdb.com/title/tt0117500/">
    
  </a>
  <a class="ephox-summary-card-link" href="http://www.imdb.com/title/tt0117500/">
    <span class="ephox-summary-card-title">The Rock (1996)</span><br><br>

    <span class="ephox-summary-card-description">Directed by Michael
    Bay. With Sean Connery, Nicolas Cage, Ed Harris, John Spencer. A
    mild-mannered chemist and an ex-con must lead the counterstrike when a
    rogue group of military men, led by a renegade general, threaten a nerve
    gas attack from Alcatraz against San
    Francisco.</span><br><br>
    <span class="ephox-summary-card-website">IMDb</span>
  </a>
</div>
```

Recommended CSS

```
.ephox-summary-card {
  border: 1px solid #AAA;
  box-shadow: 0 2px 2px 0 rgba(0,0,0,.14), 0 3px 1px -2px rgba(0,0,0,.2), 0 1px 5px 0 rgba(0,0,0,.12);
  padding: 10px;
  overflow: hidden;
  margin-bottom: 1em;
}

.ephox-summary-card a {
  text-decoration: none;
  color: inherit;
}

.ephox-summary-card a:visited {
  color: inherit;
}

.ephox-summary-card-title {
  font-size: 1.2em;
  display: block;
}

.ephox-summary-card-author {
  color: #999;
  display: block;
  margin-top: 0.5em;
}

.ephox-summary-card-website {
  color: #999;
  display: block;
  margin-top: 0.5em;
}

.ephox-summary-card-thumbnail {
  max-width: 180px;
  max-height: 180px;
  margin-left: 2em;
  float: right;
}

.ephox-summary-card-description {
  margin-top: 0.5em;
  display: block;
}
```

Adding Custom Dictionaries

Custom dictionaries can be added to Spell Checker Pro.

Configuring the Custom Dictionary Feature

Additional configuration to your `application.conf` file is required. (Don't forget to restart the Java application server after updating the configuration.)

Adding the `ephox.spelling.custom-dictionaries-path` element activates the custom dictionary feature. It points to a directory on the server's file system that will contain custom dictionary files and should not contain anything else. It is a good idea to store these files where the `application.conf` file lives, i.e. if `application.conf` is in a directory called `/opt/ephox`, the dictionary files could live in a sub-directory `/opt/ephox/dictionaries`.

Example

```
ephox {
  spelling {
    custom-dictionaries-path = "/opt/ephox/dictionaries"
  }
}
```

Creating Custom Dictionary Files

One custom dictionary can be created for each language supported by the spell checker (see [supported languages](#)), as well as an additional "global" dictionary that contains words that are valid across all languages, such as trademarks.

A dictionary file for a particular language must be named with the language code of the language (see [supported languages](#) for language codes), plus the suffix `.txt`: E.g. `en.txt`, `en_gb.txt`, `fr.txt`, `de.txt` etc.

The "global" dictionary file for language-independent words must be called `"global.txt"`.

The server will scan the dictionary directory as per configuration above and pick up `"txt"`-files for each language and the global file as present.

Custom Dictionary File Format

A dictionary file must be a simple text file with:

- one word on each line,
- either Windows-style or Linux-style line endings (CR or CR+LF)
- no comments or blank lines, and
- saved in UTF-8 encoding, with or without BOM (byte-order mark).

The last point is important for files created or edited on non-Linux (Windows or Mac) systems, as these will usually encode text files differently. However, Windows or Mac editors such as Windows Notepad can optionally save files in UTF-8 if asked to do so. Please check your editor of choice for this functionality. Failure to choose the correct encoding will result in problems with non-English letters such as umlauts and accents.

NOTE for German and Finnish languages: Spell checking in German and Finnish will employ compound word spell checking. Compound words such as "Fußballtennis" will be assumed correct as long as the root words "Fußball" and "Tennis" are individually present in the dictionary. It is not necessary to add "Fußballtennis" separately.

Verifying Custom Dictionary Functionality

If successfully configured, the custom dictionary feature will report dictionaries found in the application server's log at service startup.

Example

```
2017-06-12 17:46:00 [main] INFO com.ephox.ironbark.IronbarkBoot - Starting task (booting Ironbark)
2017-06-12 17:46:00 [main] INFO com.ephox.ironbark.IronbarkBoot - using custom dictionary: [global] = 1 words
2017-06-12 17:46:00 [main] INFO com.ephox.ironbark.IronbarkBoot - using custom dictionary: "en" = 3 words
2017-06-12 17:46:00 [main] INFO com.ephox.ironbark.IronbarkBoot - using custom dictionary: "fr" = 2 words
2017-06-12 17:46:01 [main] INFO com.ephox.ironbark.IronbarkBoot - Finished task (booting Ironbark)
```

The above log shows that 3 custom dictionaries were found, one "global", language-independent one and one each for English and French. They were found to contain 1, 3 and 2 words, respectively. Please check that this report matches your expectations.

Ongoing Dictionary Maintenance

Future additions/changes to dictionaries after the initial deployment will require a restart of the spell check service each time.

Accessibility

Tiny is committed to making the web accessible for content creators and readers of all abilities.

Creating Accessible Content

[Textbox.io](#) includes a built in web content accessibility checker that content authors can use to ensure that they're creating the most accessible content possible.

See the article on [Creating Accessible Content](#) for more information.

Accessibility Compliance

[Textbox.io](#) complies with the W3C's Web Content Accessibility Guidelines 2.0, Authoring Tool Accessibility Guidelines 2.0, Accessibility for Rich Internet Applications, and the US Government Section 508 recommendations.

See the article on [Textbox.io Accessibility Compliance](#) for more information.

Textbox.io Accessibility Compliance

This document provides an overview of the accessibility of Textbox.io as a web component (i.e. the accessibility of the editor's interface). See the document on [Creating Accessible Content](#) for more information on the accessibility tooling provided in Textbox.io to assist authors.

Textbox.io is fully accessible and usable for all users regardless of ability. The editor has been developed as an accessible component from the ground up, following the relevant best practices set forth by the W3C's [ARIA](#), [ATAG 2.0](#) and [WCAG 2.0](#) guidelines and the US Government Section 508 standards.

Screen Reader Support

Platform	Browsers	Screen Reader
Windows	Firefox*	JAWS

* Current stable channel version.

ATAG 2.0 Conformance

On June 13, 2016, the Textbox.io editor (<http://textbox.io>) conforms to Authoring Tool Accessibility Guidelines 2.0 (<http://www.w3.org/TR/ATAG20/>). Level AA conformance.

[Textbox.io 2.0 ATAG 2.0 Conformance Claim](#)

WCAG 2.0 Conformance

On June 13, 2016, the Textbox.io editor (<http://textbox.io>) conforms to Web Content Accessibility Guidelines 2.0 at <http://www.w3.org/TR/2008/REC-WCAG20-20081211/>. Level A & AA conformance.

Section 508 Compliance

Textbox.io adheres to the recommendations set forth in Section 508 of the Rehabilitation Act of 1973.

[Textbox.io 2.x VPAT Statement](#)

Creating Accessible Content

Textbox.io is compliant with the [W3C's Authoring Tool Accessibility Guidelines version 2.0](#) (ATAG 2.0). The editor assists users in creating accessible content in several ways:

- Where possible the editor creates accessible content by default - for example, table header and data cells are automatically mapped when the user specifies heading rows and/or columns.
- Authors can detect and easily rectify accessibility issues using the editor's accessibility checker
- The editor provides access to all the properties (e.g. alternative text for images, captions for tables) that an author requires to create accessible content.

Accessibility Checking and Repair

Textbox.io's accessibility checker provide authors with the ability to both check their content for accessibility issues and, in most cases, take simple, direct action to repair the issue. The issue will be explained in simple, user friendly terms that authors with little or no understanding of accessibility guidelines can take action on.

For example, if an image without alternative text is detected the accessibility check will prompt the user to provide alternative text. The user is able to take this action within the accessibility checker directly and is not required to open the image properties dialog or take action outside of the accessibility checker.

A complete list of checks performed by the editor and the corresponding guidelines can be found below.

Accessibility Checks and Guidelines

Document Element	Check Description	WCAG 2.0 Guideline	Section 508 Guideline
Images	Check if an image has alternative text	1.1.1	1194.22(a)
Images	Ensure that the alternative text is not the same as the image's filename	1.1.1	1194.22(a)
Tables	Check if a table has a caption specified	1.3.1	
Tables	Check if a table has summary specified	1.3.1	
Tables	Check if the table has headers mapped to data cells	1.3.1	1194.22(g,h)
Tables	Ensure that table markup is used to correctly structure a table	1.3.1	1194.22(g,h)
Tables	Ensure that a summary is provided for complex tables (i.e. tables that contained spanned rows or columns).	1.3.1	
Hyperlinks	Check that the URLs for adjacent links are not the same	2.4.4	
Headings	Check for use of paragraphs as headings i.e. applying styling to paragraph text to emulate a heading instead of using heading markup	1.3.1	1194.22(d)
Headings	Ensure that heading markup is used sequentially i.e. H2s only appear after H1s and H3s only after H2s	1.3.1	
Lists	Detect the use of paragraphs as lists. I.e. using consecutive paragraphs as ordered or unordered lists	1.3.1	
Text	Check that the contrast ratio of text (i.e. ratio of foreground color to background color) meets accessibility guidelines	1.4.3	

Help & Support

Overview

Support for Textbox.io, is provided online via the [Tiny support site](#). You can access the support site at: <https://support.ephox.com>.

Prior to opening a support ticket, we require all users to [register with a valid business email address](#). You can register on the support site at: <https://support.ephox.com/registration>. Upon registration you will receive an email requesting that you validate your registration. If you don't receive such a request within 5 minutes please check your spam folder to see if the email ended up there in error. If you never receive the validation email please [contact the Tiny Client Services team via email](#).

Creating a support ticket

Once you are registered on the site you will be able to interact with the Tiny support team via our support site. To create a new ticket simply go to the following URL: <http://ephox.com/support/ticket>.

Note that Tiny support requires that you provide the following information along with your ticket:

Textbox.io Editor Client Issues

- Operating System and Browser version
- JavaScript Console Log
- View source of the editor page (or some other means of obtaining the JS used to instantiate the editor, as we used to do in the early ELJ days)

Textbox.io Server Components Issues

- Server type and version (eg. Jetty, Tomcat)
- Server log - This will be available either on the system.out log for the server or in a separate file if you've configured the [optional logging parameters](#) for Textbox.io on you server(s).

Please see [Troubleshooting](#) for troubleshooting advice with the server side components.

Web Services Troubleshooting

Browser Specific Issues

Internet Explorer Specific Troubleshooting Tips

If the editor is reporting that the service cannot be found and tracing the network traffic reveals that no request is made at all, check that the server is not listed in the "Trusted Sites" section of Internet Explorer's security options. If it is, remove it and try again.

Chrome Specific Tips

If the server is not running on a standard http or https port (80 or 443) then Chrome will include the port in the origin header that is sent to the server. Other browsers do not do this. This is why when specifying the "allowed-origins" config, you should use both the hostname by itself and the hostname and port in the configuration. See [Entering Origins](#) for more details

Error Messages About the "Origins" or "Spelling Service Missing"

If you see these errors occurring, this is generally caused by the following reasons. This guide will walk you through the debugging process to identify the specific problem and how to remedy the issue.

1. The application.conf file is incorrect. Please go back and follow the steps listed in the [installation guide](#). This is the most common problem - often the origins are specified without the port numbers and this can cause things to fail, eg: use 'http://localhost:8080' instead of 'http://localhost'. After making changes to the application.conf file, please restart Tomcat .
2. The application.conf file is correct, but something is wrong with one of the services. See the section below to debug the services
3. The application.conf file is correct, and the services are working, but the URL's that editor uses are not quite right. Refer to this [config](#) page for help
4. All of the above are correct, but the browser sends back a different origin. See step 6. of '[Using Browser Tooling to Investigate Services Issues](#)' below and add the origin value to the list of origins. Restart Tomcat and then refresh the editor page in a browser and things should work.

Checking the Server Configuration

To test the services, we will start with the following:

We will use Tomcat 7 installed at /opt/tomcat/ and running on port 8080, and this is our 'application.conf' file. We have made a folder called 'images' in the webapps directory of the tomcat install i.e. /opt/tomcat/webapps/images/. Some of the commands below that require a 'terminal' window to be used.

If you are on a Linux or Mac environment use a shell of your choice and make sure the 'curl' package is installed.

If you are on a windows environment follow the page [Installing curl in Windows](#) and then open a 'cmd' prompt and run the commands from there.

Please modify your configuration according to how you have it set up based on your environment.

example application.conf

```
ephox{
  allowed-origins {
    origins = [ "http://localhost", "http://localhost:8080", "http://myserver" ]
    url = "http://localhost:8080/ephox-allowed-origins/cors"
  }

  image-proxy {
    storageURL = "http://localhost:8080/ephox-local-storage/api/1/storeImage"
  }

  local-storage {
    imagesDirectory = "/opt/tomcat7/webapps/images/"
    returnUrl = "http://localhost:8080/images/"
  }
}
```

Lets start with the 'ephox-allowed-origins'

Once you've deployed the service and started Tomcat/Jetty open a browser window and enter the following:

origins request

```
http://localhost:8080/ephox-allowed-origins/cors
```

This should return the **entire** list of origins that you specified in the application.conf file. So you should see the following returned, if everything is working correctly. If you see **only one** or do not see this altogether, then Tomcat hasn't been configured correctly to use the application.conf file. Refer to step 3 of the [Installing Services](#) Section to fix this and make sure you can see the response below before proceeding

origins response

```
{"value":["http://localhost", "http://localhost:8080", "http://myserver"]}
```

Debugging 'ephox-spelling'

Once you've deployed the service and started Tomcat/Jetty open a browser window and enter the following:

spelling request

```
http://localhost:8080/ephox-spelling/version
```

This should return the version number "1.1.0"

Now open a terminal window and enter the following to test that the dictionaries are loaded correctly and the service is responding as intended. Note: the "Origin" value specifies the origin that the request is originating from - in a browser with the editor loaded this happens automatically

spelling suggestion req unix

```
curl -v --request POST -H "Content-Type: application/json" -H "Origin: http://localhost:8080" http://localhost:8080/ephox-spelling/1/suggestion -d '{"words":["hello","world"], "language": "en_US" }'
```

If you are using Windows and curl, please make sure you add escape characters ("\") to the quotes and use double quotes (") around the data parameter, instead of single quotes (') as shown below

WINDOWS ONLY : spelling suggestion

```
curl -v --request POST -H "Content-Type: application/json" -H "Origin: http://localhost:8080" http://localhost:8080/ephox-spelling/1/suggestion -d "{ \"words\":[\"hello\",\"world\"], \"language\": \"en_US\" }"
```

The response below is what you should see. The example below is good and returns a status code 200. If you see a error code 403, that means the value of the "Origin " supplied is incorrect. Please edit the list of origins in the 'application.conf' file and restart Tomcat7.

origins response

```
* Hostname was NOT found in DNS cache
*   Trying w.x.y.z...
* Connected to localhost (w.x.y.z) port 8080 (#0)
> POST /ephox-spelling/1/suggestion HTTP/1.1
> User-Agent: curl/7.35.0
> Host: localhost:8080
> Accept: */*
> Content-Type: application/json
> Origin: http://localhost:8080
> Content-Length: 64
>
* upload completely sent off: 64 out of 64 bytes
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Access-Control-Allow-Credentials: true
< Access-Control-Allow-Methods: GET,PUT,POST,DELETE,OPTIONS
< Access-Control-Allow-Origin: http://localhost:8080
< Access-Control-Max-Age: 3600
< Content-Type: application/json;charset=UTF-8
< Date: Tue, 25 Nov 2014 23:34:09 GMT
< Content-Length: 302
< Connection: close
<
* Closing connection 0
{"suggest":{"hello":["hello","hellos","hell","helot","hellion","helloed","helloes","jello","halo","shell","he'll","he'lll","helots","hallow","hollow","halloo"],"world":["world","worlds","word","wold","worldly","whorled","sword","whorl","wordy","words","old","would","worked","ward","swords","whorls"]}}
```

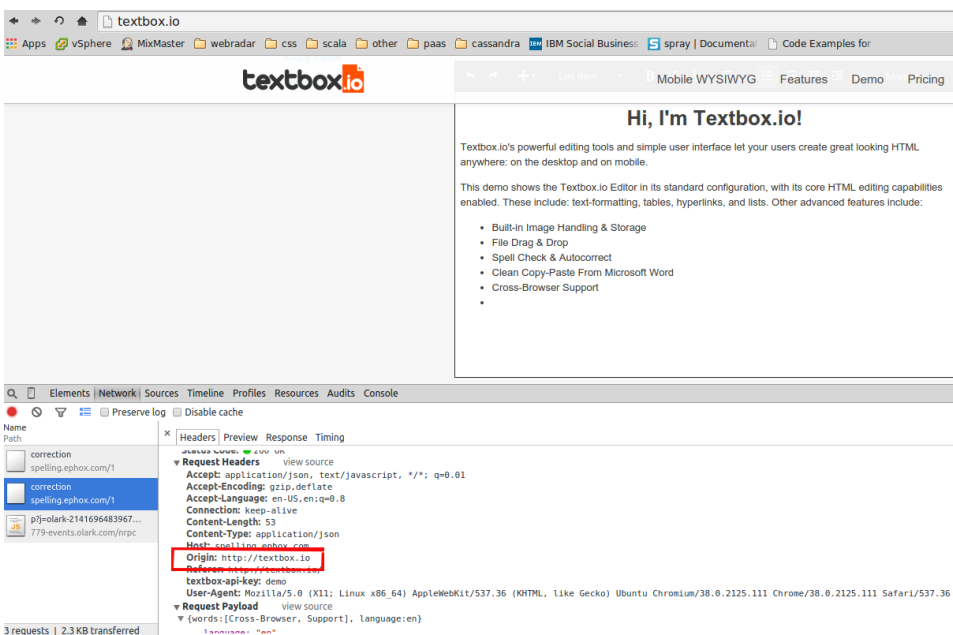
Once you see that the service is responding with the correct response, the next steps are to check the data that the [browser is sending to the service](#).

Related articles

- [Using Browser tooling to investigate services](#)
- [Error messages about the "Origins" or "Spelling Service Missing"](#)
- [Out of Memory Errors](#)

Using Browser Tooling to Investigate Services Issues

1. Open your browser's Console/Network tools:
Chrome: View menu -> Developer -> JavaScript Console. Click the *Network* tab (located between *Elements* and *Sources*).
Firefox: Tools menu -> Web Developer -> Network.
Safari: Develop -> Show Error Console. Click the *Timelines* link between *Resources* and *Debugger*.
Internet Explorer: Click the cog icon on the top-right side of the browser. Select *F12 Developer Tools*. Click the *Network* link (next to the 'Profiler' link).
2. Refresh the webpage featuring an Tiny rich-text editor configured with the spelling service. Enter a misspelled word into the editor.
3. Locate the network results that match the following URLs:
- http://YOUR_SERVER:YOUR_PORT/ephox-spelling/1/autocorrect
- http://YOUR_SERVER:YOUR_PORT/ephox-spelling/1/correction
4. If the network response for these services is *404*, try the following:
- Take the service URL displayed as erroneous (example: http://YOUR_SERVER:YOUR_PORT/ephox-spelling/1/autocorrect).
- Remove everything from the '1' onwards (including the '1') and replace it with */version*. (example: Change http://YOUR_SERVER:YOUR_PORT/ephox-spelling/1/correction to http://YOUR_SERVER:YOUT_PORT/ephox-spelling/version). The response code should be 200 and the body should display the version number.
5. If the response for the version URL is still 404, it means the service has not been started or installed correctly.
6. To check the "Origin" value that the browser uses, open the network tools (chrome in this screenshot) and refresh the page. Then enter a couple of words in the editor and select one of the requests on the bottom left ('correction' in the screen shot) and select the 'Headers' section. Look for the 'Origin' header value.



The value of the origin header sent by the must match the origin specified in the application.conf server configuration. If it does not match, you must make the server configuration match the browser

Windows Server Specific Issues:

Sometimes the 'Origin' header is never sent to the services, which results in the editor and services not working as intended. Follow step 6 from above and determine what the 'Origin' is - if you do not see an 'Origin' header at all, please do the following:

1. Try accessing the editor web page using your machine's fully qualified domain name (FQDN) rather than 'localhost'; and keep the network tools open so you can see if the 'Origin' header is sent back to the services.
So open a browser window and try (replace the path to match your setup):

```
fqdn request

http://myhost:myport/textboxio/index.html
```

2. If you now see an 'Origin' header being sent across, please alter your application.conf and replace all instances of 'localhost' with the domain name of your machine

3. Restart the Tomcat / Jetty service
4. Reload the browser page and all should work well
5. If you are still experiencing problems, please contact [Tiny Support](#).

General Troubleshooting

General Troubleshooting Advice

Step 1

Verify that the [spelling configuration](#) is correct in your editor client JavaScript configuration.

Step 2

Ensure that your firewall has the appropriate ports and rules configured correctly. Be sure that the server the service is running on is accessible from the browser via the port specified in the server configuration

Step 3

Check the logs of the appropriate Java server for information. When making changes to the configuration you will need to restart the application server each time a change is made for that change to take effect. Refresh your browser window and then try the service again.

Out of Memory Errors

The Java Application Server Throws "Out of Memory" Errors

Even though you may have a large amount of RAM available, the Java Virtual Machine doesn't get to see all of that - by default it is limited to only 256Mb.

For example, if you are using Tomcat, you can view how much memory is being consumed by apps. To do this you need to have the management console enabled.

On a vanilla install this is done by editing the file `/tomcat/install/directory/conf/tomcat-users.xml` adding these lines in:

```
<role rolename="manager-gui" />
<user username="tomcat" password="password" roles="manager-gui" />
```

Then, restart the server and go to a browser and open the default tomcat page <http://localhost:8080>. On the top right hand side are three buttons, the first of which should be "Server Status". Click that link, login with the details you set above and you should be able to see the memory consumption (see the figure below for an example).

The screenshot shows the Apache Tomcat Manager interface. At the top, there's the Apache Software Foundation logo and a cat icon. The main heading is "Server Status". Below it, there are navigation links: "List Applications", "HTML Manager Help", "Manager Help", and "Complete Server Status".

The "Server Information" section contains a table with the following data:

Tomcat Version	JVM Version	JVM Vendor	OS Name	OS Version	OS Architecture	Hostname	IP Address
Apache Tomcat/7.0.54	1.7.0_55-b14	Oracle Corporation	Linux	3.13.0-24-generic	amd64	test	127.0.1.1

The "JVM" section shows memory usage statistics:

Memory Pool	Type	Initial	Total	Maximum	Used
PS Eden Space	Heap memory	513.00 MB	513.00 MB	513.00 MB	266.78 MB (52%)
PS Old Gen	Heap memory	1365.50 MB	1365.50 MB	1365.50 MB	0.00 MB (0%)
PS Survivor Space	Heap memory	85.00 MB	85.00 MB	85.00 MB	0.00 MB (0%)
Code Cache	Non-heap memory	2.43 MB	2.43 MB	48.00 MB	1.39 MB (2%)
PS Perm Gen	Non-heap memory	64.00 MB	64.00 MB	512.00 MB	16.65 MB (3%)

Two request logs are shown. The first is for "ajp-bio-8009" with a table of request details. The second is for "http-bio-8080" with a table showing a single request:

Stage	Time	B Sent	B Recv	Client (Forwarded)	Client (Actual)	VHost	Request
S	4 ms	0 KB	0 KB	127.0.0.1	127.0.0.1	localhost	GET /manager/status HTTP/1.1
P	?	?	?	?	?	?	
P	?	?	?	?	?	?	
P	?	?	?	?	?	?	

At the bottom, there is a copyright notice: "Copyright © 1999-2014, Apache Software Foundation".

To increase the amount of memory:

Tomcat :

Edit the setenv.sh (Unix) or setenv.bat (Windows) to read as follows:

windows config

On Windows, please prefix each line with 'set' and remove the quotes . So the configuration would look like:

```
set CATALINA_OPTS= -Dconfig.file=/config/file/location/application.conf
set JAVA_OPTS= -Xms2048m -Xmx2048m -XX:PermSize=64m -XX:MaxPermSize=512m -Dfile.encoding=utf-8 -Djava.awt.
headless=true -XX:+UseParallelGC -XX:MaxGCPauseMillis=100
```

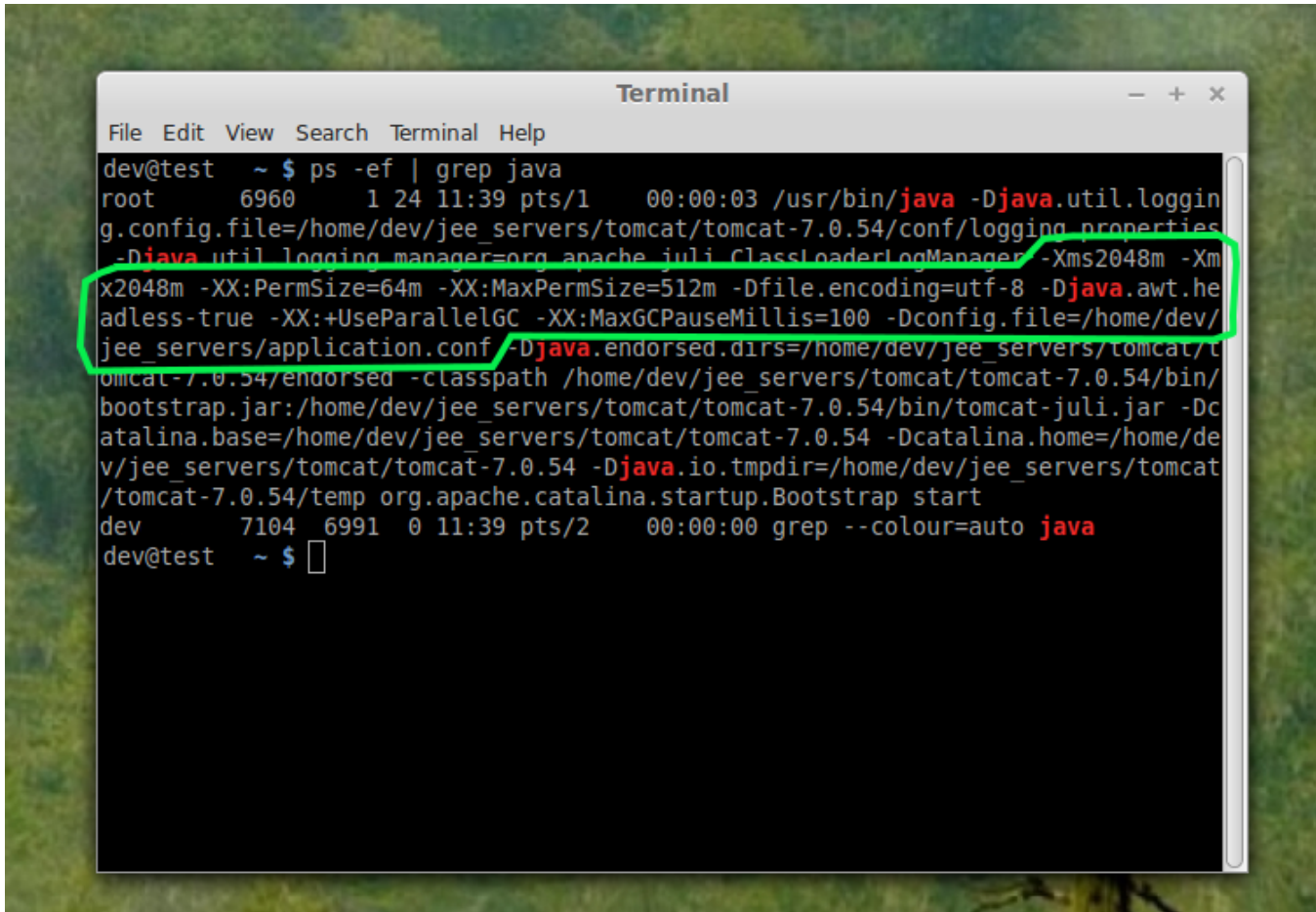
```
CATALINA_OPTS=" -Dconfig.file=/config/file/location/application.conf
JAVA_OPTS=" -Xms2048m -Xmx2048m -XX:PermSize=64m -XX:MaxPermSize=512m -Dfile.encoding=utf-8 -Djava.awt.headless-true -XX:+UseParallelGC -XX:MaxGCPauseMillis=100"
```

Jetty :

Edit the start.ini file to read as follows:

```
#####
# Jetty start.jar arguments
# Each line of this file is prepended to the command line
# arguments # of a call to:
# java -jar start.jar [arg...]
#####
-Xms2048m -Xmx2048m -XX:PermSize=64m -XX:MaxPermSize=512m -Dconfig.file=/config/file/location/application.conf
```

Restart the service and confirm the settings have been applied like so:



```
Terminal
File Edit View Search Terminal Help
dev@test ~ $ ps -ef | grep java
root    6960    1 24 11:39 pts/1    00:00:03 /usr/bin/java -Djava.util.logging.config.file=/home/dev/jee_servers/tomcat/tomcat-7.0.54/conf/logging.properties -Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager -Xms2048m -Xmx2048m -XX:PermSize=64m -XX:MaxPermSize=512m -Dfile.encoding=utf-8 -Djava.awt.headless=true -XX:+UseParallelGC -XX:MaxGCPauseMillis=100 -Dconfig.file=/home/dev/jee_servers/application.conf -Djava.endorsed.dirs=/home/dev/jee_servers/tomcat/tomcat-7.0.54/endorsed -classpath /home/dev/jee_servers/tomcat/tomcat-7.0.54/bin/bootstrap.jar:/home/dev/jee_servers/tomcat/tomcat-7.0.54/bin/tomcat-juli.jar -Dcatalina.base=/home/dev/jee_servers/tomcat/tomcat-7.0.54 -Dcatalina.home=/home/dev/jee_servers/tomcat/tomcat-7.0.54 -Djava.io.tmpdir=/home/dev/jee_servers/tomcat/tomcat-7.0.54/temp org.apache.catalina.startup.Bootstrap start
dev     7104    6991    0 11:39 pts/2    00:00:00 grep --colour=auto java
dev@test ~ $
```


Troubleshooting Tools - curl

Installing curl on Mac

curl is installed by default on all MacOS X installations. Open the "terminal" application to use it

Installing curl on Linux

Use your distribution package manager to install curl. See your distribution documentation for details.

Installing curl (or equivalent package) on Windows

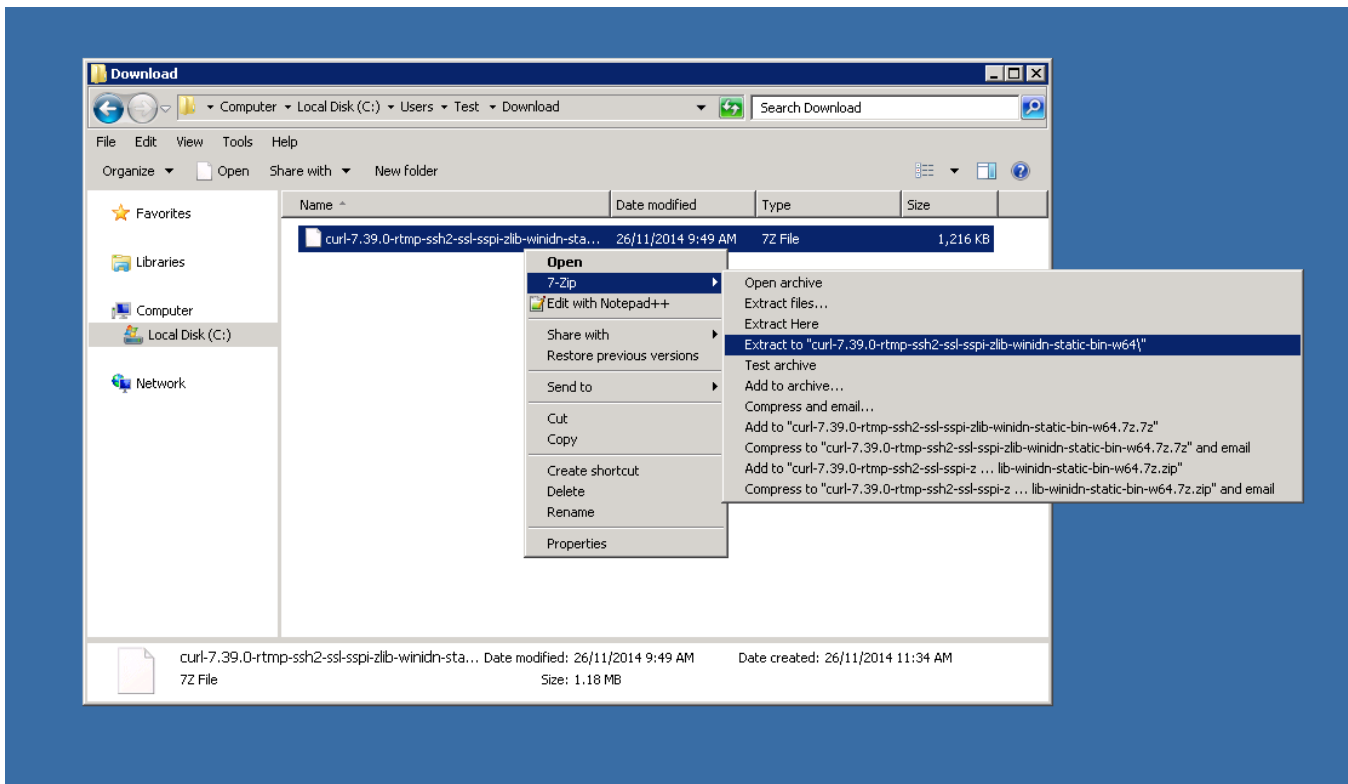
Download and install the curl package based on your environment:

x64: <http://curl.haxx.se/dlwiz/?type=bin&os=Win64&flav=MinGW64>

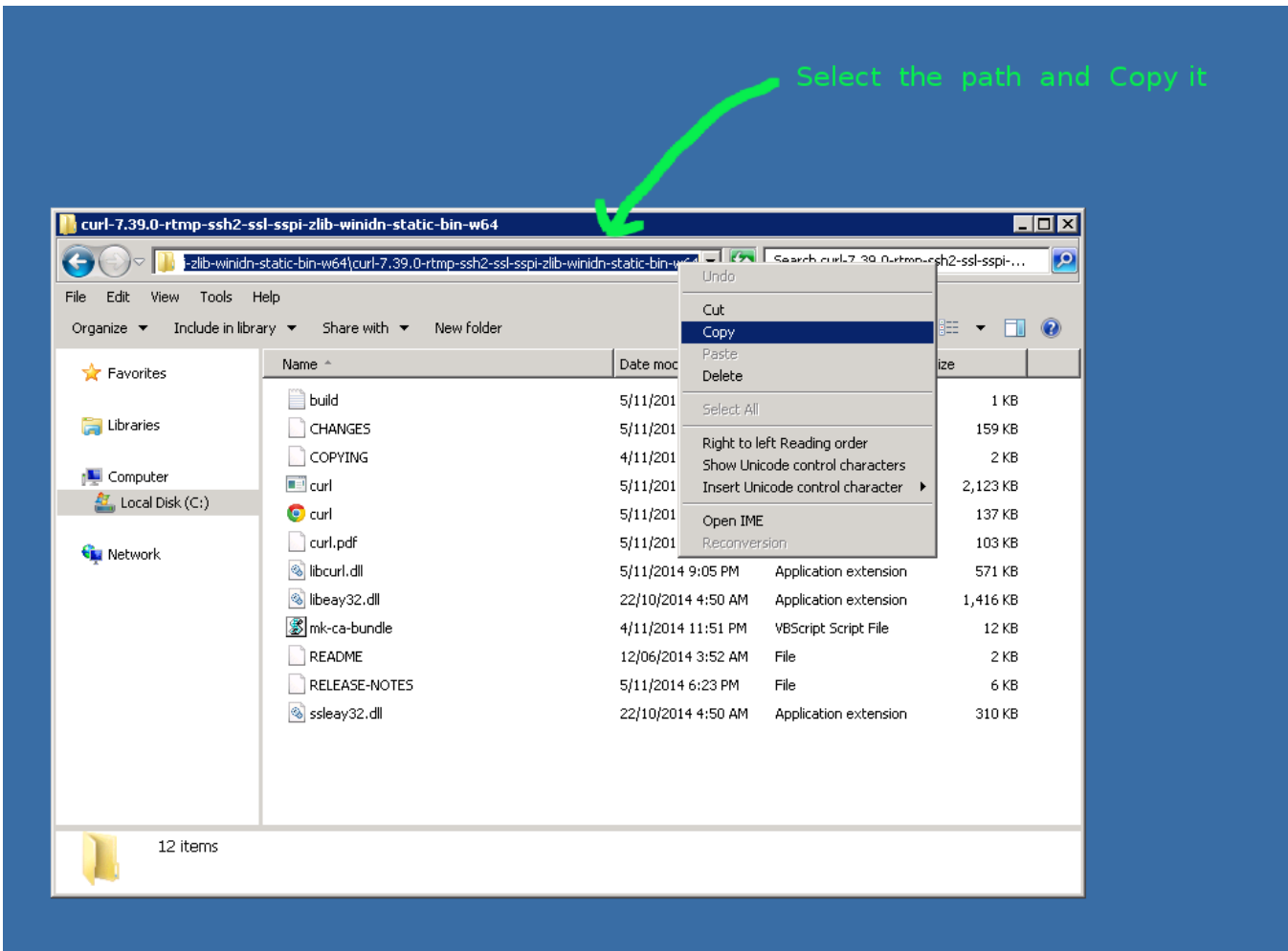
x86: <http://curl.haxx.se/dlwiz/?type=bin&os=Win32&flav=-&ver=2000%2FXP> and select either of the **curl version: 7.39.0 - SSL enabled SSH enabled packages**

Once downloaded:

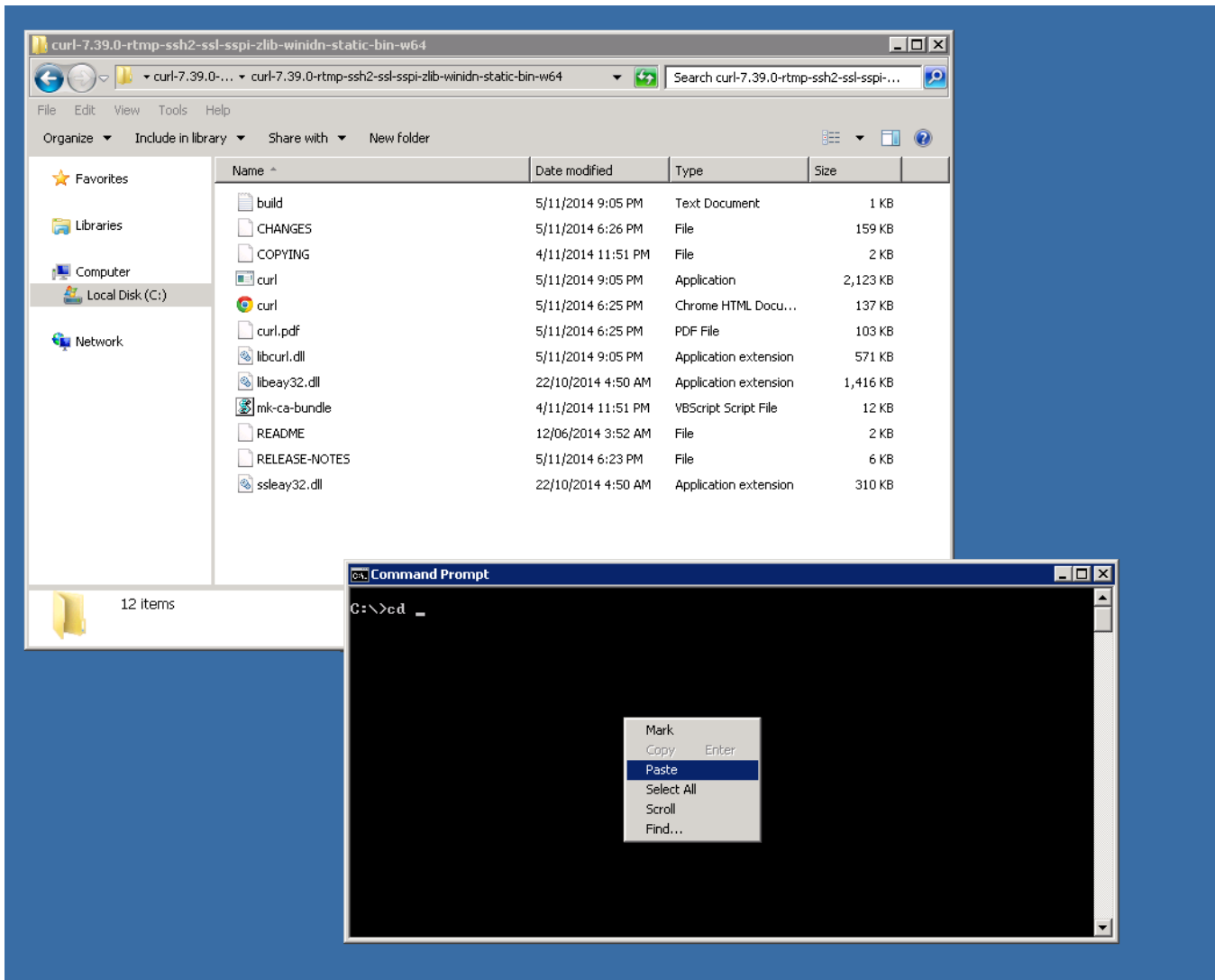
1. Unzip the package like so:



2. Copy the path of the folder to where the 'curl.exe' is in:



3. Open a cmd prompt. Start -> Programs -> Accessories -> cmd (or command prompt). Then change to that directory to the folder where the 'curl.exe' is found. Enter 'cd' (without quotes) and then paste in the path from step 2.



4. Once in the folder enter 'curl --version' (without quotes) and ensure you get a valid version

```
Command Prompt
C:\Users\test\Download\curl-7.39.0-rtmp-ssh2-ssl-sspi-zlib-winidn-static-bin-w64\curl-7.39.0-rtmp-ssh2-ssl-sspi-zlib-winidn-static-bin-w64>curl --version
curl 7.39.0 (x86_64-pc-win32) libcurl/7.39.0 OpenSSL/1.0.0o zlib/1.2.8 WinIDN libssh2/1.4.3 librtmp/2.3
Protocols: dict file ftp ftps gopher http https imap imaps ldap pop3 pop3s rtmp
rtsp scp sftp smtp smtps telnet tftp
Features: AsynchDNS IDN Largefile SSPI SPNEGO NTLM SSL libz
```