DOMBuilder Documentation

Release 2.0.0

Jonathan Buchanan

Sep 27, 2017

Contents

DOMBuilder takes *some* of the pain out of dynamically creating HTML content in JavaScript and supports generating multiple types of output from the same inputs.

CHAPTER 1

DOMBuilder Core

This page documents the `DOMBuilder` object implemented in the core DOMBuilder.js script.

Note: For brevity, some further examples will assume that element functions are available in the global scope.

Element Functions

Element functions accept flexible combinations of input arguments, creating a declarative layer on top of *`DOMBuilder.createElement()`*.

`DOMBuilder.elements` is an `Object` containing a function for each valid tag name declared in the HTML 4.01 Strict DTD, Frameset DTD and HTML5 differences from HTML4 W3C Working Draft, referenced by the corresponding tag name in uppercase.

`DOMBuilder.`**`elements`**
: Element functions which create contents based on the current value of *`DOMBuilder.mode`*

An exhaustive list is available below in *Element Function Names*.

When called, these functions will create an element with the corresponding tag name, giving it any attributes which are specified as properties of an optional `Object` argument and appending any child content passed in.

Element functions accept the following variations of arguments:

Element Creation Function Arguments	
`(attributes, child1, ...)`	an attributes `Object` followed by an arbitrary number of children.
`(attributes, [child1, ...])`	an attributes `Object` and an `Array` of children.
`(child1, ...)`	an arbitrary number of children.
`([child1, ...])`	an `Array` of children.

Example:

The following function creates a <table> representation of a list of objects, taking advantage of the flexible combinations of arguments accepted by element functions:

```
/**
 * @param headers a list of column headings.
 * @param objects the objects to be displayed.
 * @param properties names of object properties which map to the
 *                   corresponding columns.
 */
function createTable(headers, objects, properties) {
  return TABLE({cellSpacing: 1, 'class': 'data sortable'}
  , THEAD(TR(TH.map(headers)))
  , TBODY(
      TR.map(objects, function(obj) {
        return TD.map(properties, function(prop) {
          if (typeof obj[prop] == 'boolean') {
            return obj[prop] ? 'Yes' : 'No'
          }
          return obj[prop]
        })
      })
    )
  )
}
```

Given this function, the following code...

```
createTable(
  ['Name', 'Table #', 'Vegetarian'],
  [{name: 'Steve McMeat',   table: 3, veggie: false},
   {name: 'Omar Omni',      table: 5, veggie: false},
   {name: 'Ivana Huggacow', table: 1, veggie: True}],
  ['name', 'table', 'veggie']
)
```

...would produce output corresponding to the following HTML:

```
<table class="data sortable" cellspacing="1">
  <thead>
    <tr>
      <th>Name</th>
      <th>Table #</th>
      <th>Vegetarian</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Steve McMeat</td>
      <td>3</td>
      <td>No</td>
    </tr>
    <tr>
      <td>Omar Omni</td>
      <td>5</td>
      <td>No</td>
    </tr>
    <tr>
      <td>Ivana Huggacow</td>
      <td>1</td>
```

```
      <td>Yes</td>
    </tr>
  </tbody>
</table>
```

Map Functions

New in version 1.3.

Map functions provide a shorthand for:

- creating elements for each item in a list, via *DOMBuilder.map()*
- wrapping elements created for each item in a list with a fragment, via *DOMBuilder.fragment.map()*

DOMBuilder.**map**(*tagName*, *defaultAttributes*, *items*[, *mappingFunction*[, *mode*]])

Creates an element for (potentially) every item in a list.

Arguments

- **tagName** (*String*) – the name of the element to create for each item in the list.
- **defaultAttributes** (*Object*) – default attributes for the element.
- **items** (*Array*) – the list of items to use as the basis for creating elements.
- **mappingFunction** (*Function*) – a function to be called with each item in the list, to provide contents for the element which will be created for that item.
- **mode** (*String*) – the DOMBuilder mode to be used when creating elements.

If provided, the mapping function will be called with the following arguments:

```
mappingFunction(item, attributes, loopStatus)
```

Contents returned by the mapping function can consist of a single value or a mixed `Array`.

Attributes for the created element can be altered per-item by modifying the `attributes` argument, which will initially contain the contents of `defaultAttributes`, if it was provided.

The `loopStatus` argument is an `Object` with the following properties:

index 0-based index of the current item in the list.

first `true` if the current item is the first in the list.

last `true` if the current item is the last in the list.

The mapping function can prevent an element from being created for a given item altogether by returning `null`.

If a mapping function is not provided, a new element will be created for each item in the list and the item itself will be used as the contents.

Changed in version 2.0: `defaultAttributes` is now required - flexible arguments are now handled by the `map` functions exposed on element creation functions; the `mode` argument was added; a loop status object is now passed when calling the mapping function.

This function is also exposed via element creation functions. Each element creation function has its own `map` function, which allows more flexible arguments to be passed in.

Element Creation Function `.map()` Arguments	
`(defaultAttributes, [item1, ...], mappingFunction)`	a default attributes attributes object, a list of items and a mapping Function.
`([item1, ...], mappingFunction)`	a list of items and a mapping Function.
`([item1, ...])`	a list of items, to be used as element content as-is.

This example shows how you could make use of the `attributes` and `itemIndex` arguments to the mapping function to implement table striping:

```
TR.map(rows, function(row, attributes, loop) {
  attributes['class'] = (loop.index % 2 == 0 ? 'stripe1' : 'stripe2')
  return TD.map(row)
})
```

Core API

These are the core functions whose output can be controlled using *Output Modes*.

`DOMBuilder.`**`createElement`**`(`*tagName[, attributes], children], mode]*`)`

Creates an HTML element with the given tag name, attributes and children, optionally with a forced output mode.

Arguments

- **`tagName`** (*`String`*) – the name of the element to be created.
- **`attributes`** (*`Object`*) – attributes to be applied to the new element.
- **`children`** (*`Array`*) – childen to be appended to the new element.
- **`mode`** (*`String`*) – the mode to be used to create the element.

If children are provided, they will be appended to the new element. Any children which are not elements or fragments will be coerced to `String` and appended as text nodes.

Changed in version 2.0: Now delegates to the configured mode to do all the real work.

`DOMBuilder.`**`fragment`**`()`

Creates a container grouping any given elements together without the need to wrap them in a redundant element. This functionality was for *DOM Mode* - see *Document Fragments* - but is supported by all output modes for the same grouping purposes.

Supported argument formats are:

Fragment Creation Arguments	
`(child1, ...)`	an arbitrary number of children.
`([child1, ...])`	an `Array` of children.

Output Modes

Changed in version 2.0: Output modes now sit independent of DOMBuilder core and are pluggable.

DOMBuilder provides the ability to register new modes, which make use of the arguments given when elements and fragments are created.

DOMBuilder.**addMode**(*mode*)

Adds a new mode and exposes an API for it in the DOMBuilder object under a property corresponding to the mode's name.

The first mode to be added will have its name stored in *DOMBuilder.mode*, making it the default output mode.

Arguments

- **mode** (*Object*) – Modes are defined as an `Object` with the following properties.

  name the mode's name.

  createElement(tagName, attributes, children) a Function which takes a tag name, attributes object and list of children and returns a content object.

  fragment(children) a Function which takes a list of children and returns a content fragment.

  isModeObject(object) (optional) a Function which can be used to eliminate false positives when DOMBuilder is trying to determine whether or not an attributes object was given - it should return `true` if given a mode-created content object.

  api (optional) an Object defining a public API for the mode's implementation, exposing variables, functions and constructors used in implementation which may be of interest to anyone who wants to make use of the mode's internals.

  apply (optional) an Object defining additional properties to be added to the object DOMBuilder creates for easy access to mode-specific element functions (see below). Just as element functions are a convenience layer over *DOMBuilder.createElement()*, the purpose of the `apply` property is to allow modes to provde a convenient way to access mode-specific functionality.

  Any properties specified with `apply` will also be added to objects passed into *DOMBuilder.apply()* when a mode is specified.

When a mode is added, a `DOMBuilder.<mode name>` Object is also created, containing element functions which will always create content using the given mode and any additional properties which were defined via the mode's `apply` properties.

New in version 2.0.

Example: a mode which prints out the arguments it was given:

```
DOMBuilder.addMode({
  name: 'log'
, createElement: function(tagName, attributes, children) {
    console.log(tagName, attributes, children)
    return tagName
  }
})
```

```
>>> DOMBuilder.build(article, 'log')
h2 Object {} ["Article title"]
p Object {} ["Paragraph one"]
p Object {} ["Paragraph two"]
div Object { class="article"} ["h2", "p", "p"]
```

Setting a mode's name as *DOMBuilder.mode* makes it the default output format.

`DOMBuilder.`**`mode`**
: Determines which mode *`DOMBuilder.createElement()`* and *`DOMBuilder.fragment()`* will use by default.

Provided Modes

Implementations of the following default modes are provided for use:

Output modes:

Name	Outputs	Documentation
`'dom'`	DOM Elements	*DOM Mode*
`'html'`	*`MockElement()`* objects which `toString()` to HTML4	*HTML Mode*

Temporarily Switching Mode

If you're going to be working with mixed output types, forgetting to reset *`DOMBuilder.mode`* would be catastrophic, so DOMBuilder provides *`DOMBuilder.withMode()`* to manage it for you.

`DOMBuilder.`**`withMode`**(*mode*, *func*[, *args...*])
: Calls a function, with *`DOMBuilder.mode`* set to the given value for the duration of the function call, and returns its output.

  Any additional arguments passed after the `func` argument will be passed to the function when it is called.

```
>>> function createParagraph() { return P('Bed and', BR(), 'BReakfast') }
>>> DOMBuilder.mode = 'dom'
>>> createParagraph().toString() // DOM mode by default
"[object HTMLParagraphElement]"
>>> DOMBuilder.withMode('HTML', createParagraph).toString()
"<p>Bed and<br>BReakfast</p>"
```

Referencing Element Functions

Some options for convenient ways to reference element functions.

Create a local variable referencing the element functions you want to use:

```
var el = DOMBuilder.dom
el.DIV('Hello')
```

Use the with statement to put the element functions of your choice in the scope chain for variable resolution:

```
with (DOMBuilder.dom) {
  DIV('Hello')
}
```

You could consider the with statement misunderstood; some consider with Statement Considered Harmful the final word on using the `with` statement *at all*, but to quote The Dude - yeah, well, y'know, that's just, like, your opinion, man. It's actually a pretty nice fit for builder and templating code in which properties are only ever *read* from the scoped object and it accounts for a significant proportion of property lookups.

Just be aware that the `with` statement will be considered a syntax error if you wish to *opt-in* to ECMAScript 5's strict mode in the future, but there are ways are ways to mix strict and non-stict code, as it can be toggled at the function level.

Add element functions to the global scope using *DOMBuilder.apply()*:

```
DOMBuilder.apply(window, 'dom')
DIV('Hello')
```

Filling the global scope full of properties isn't something which should be done lightly, but you might be ok with it for quick scripts or for utilities which you'll be using often and which are named in ways which are unlikely to conflict with your other code, such as DOMBuilder's upper-cased element functions.

This particular piece of documentation won't judge you - it's your call.

DOMBuilder.**apply**(*context*[, *mode*])

Adds element functions to the given object, optionally for a specific mode.

Arguments

- **context** (*Object*) – An object which element functions will be added to.
- **mode** (*String*) – The name of a mode for which mode-specific element functions and convenience API should be added.

  If not given, element functions from *DOMBuilder.elements* will be used.

Changed in version 2.0: The `context` argument is now required; added the `mode` argument.

Element Function Names

An exhaustive list of the available element function names.

Element Function Names									
A	ABBR	ACRONYM	ADDRESS	AREA	ARTICLE	ASIDE	AUDIO	B	BDI
BDO	BIG	BLOCKQUOTE	BODY	BR	BUTTON	CANVAS	CAPTION	CITE	CODE
COL	COLGROUP	COMMAND	DATALIST	DD	DEL	DETAILS	DFN	DIV	DL
DT	EM	EMBED	FIELDSET	FIGCAPTION	FIGURE	FOOTER	FORM	FRAME	FRAMESET
H1	H2	H3	H4	H5	H6	HR	HEAD	HEADER	HGROUP
HTML	I	IFRAME	IMG	INPUT	INS	KBD	KEYGEN	LABEL	LEGEND
LI	LINK	MAP	MARK	META	METER	NAV	NOSCRIPT	OBJECT	OL
OPTGROUP	OPTION	OUTPUT	P	PARAM	PRE	PROGRESS	Q	RP	RT
RUBY	SAMP	SCRIPT	SECTION	SELECT	SMALL	SOURCE	SPAN	STRONG	STYLE
SUB	SUMMARY	SUP	TABLE	TBODY	TD	TEXTAREA	TFOOT	TH	THEAD
TIME	TITLE	TR	TRACK	TT	UL	VAR	VIDEO	WBR	

Building from Arrays

New in version 2.0.

To make use of DOMBuilder's *Output Modes* without using the rest of its API, you can define HTML elements as nested Arrays, where each array represents an element and each element can consist of a tag name, an optional `Object` defining element attributes and an arbitrary number of content items.

For example:

Input	Sample HTML Output
`['div']`	`<div></div>`
`['div', {id: 'test'}]`	`<div id="test"></div>`
`['div', 'content']`	`<div>content</div>`
`['div', {id: 'test'}, 'content']`	`<div id="test">content</div>`
`['div', 'oh, ', ['span', 'hi!']]`	`<div>oh, <span>hi!</span></div>`

To create content from a nested Array in this format, use:

`DOMBuilder.`**`build`**(*contents*[, *mode*])

Builds the specified type of output from a nested Array representation of HTML elements.

Arguments

- **`contents`** (*`Array`*) – Content defined as a nested Array
- **`mode`** (*`String`*) – Name of the output mode to use. If not given, defaults to *`DOMBuilder.mode`*

```
var article =
  ['div', {'class': 'article'}
  , ['h2', 'Article title']
  , ['p', 'Paragraph one']
  , ['p', 'Paragraph two']
  ]
```

```
>>> DOMBuilder.build(article, 'html').toString()
<div class="article"><h2>Article title</h2><p>Paragraph one</p><p>Paragraph two</p></
↪div>
```

You can also use the element function and core API to create array representations of HTML elements, by setting *`DOMBuilder.mode`* to `null` and using *`DOMBuilder.elements`*, or directly by using the element functions defined in *`DOMBuilder.array`*:

`DOMBuilder.`**`array`**

Element functions which will always create nested element Array output.

This is the default output format if *`DOMBuilder.mode`* is `null`, effectively making it a `null` mode.

CHAPTER 2

DOM Mode

DOM mode provides an output mode which generates DOM Elements from *DOMBuilder.createElement()* calls and DOM DocumentFragments from *DOMBuilder.fragment()* calls.

The DOM mode API is exposed through `DOMBuilder.modes.dom.api`.

Mode-specific element functions are exposed through *DOMBuilder.dom*.

`DOMBuilder.`**`dom`**

Element functions which will always create DOM Element output.

New in version 2.0.

Attributes

Some attributes are given special treatment based on their name.

Event Handlers

Event handlers can be specified by supplying an event name as one of the element's attributes and an event handling function as the corresponding value. Any of the following events can be registered in this manner:

Event Names					
blur	focus	focusin	focusout	load	resize
scroll	unload	click	dblclick	mousedown	mouseup
mousemove	mouseover	mouseout	mouseenter	mouseleave	change
select	submit	keydown	keypress	keyup	error

These correspond to events which have jQuery shortcut methods, which will be used for event handler registration if jQuery is available, otherwise legacy event registration will be used.

For example, the following will create a text input which displays a default value, clearing it when the input is focused and restoring the default if the input is left blank:

```
var defaultInput =
    el.INPUT({
        type: 'text', name: 'email'
      , value: 'email@host.com', defaultValue: 'email@host.com'
      , focus: function() {
          if (this.value == this.defaultValue) {
            this.value = ''
          }
        }
      , blur: function() {
          if (this.value == '') {
            this.value = this.defaultValue
          }
        }
      }
    )
```

Other 'Special' Attributes

Other attributes which trigger special handling or explicit compatibility handling between DOM and HTML modes.

innerHTML If you specify an `innerHTML` attribute, the given String will be the sole source used to provide the element's contents, even if you pass more contents in as arguments.

- In DOM mode, the element's `innerHTML` property will be set and no further children will be appended, even if given.
- In HTML mode, the given HTML will be used, unescaped, as the element's contents.

Document Fragments

A DOM DocumentFragment is a lightweight container for elements which allows you to append its entire contents with a single call to the destination element's `appendChild()` method.

If you're thinking of adding a wrapper `<div>` solely to be able to insert a number of sibling elements at the same time, a DocumentFragment will do the same job without the need for the redundant element. This single append functionality also makes it a handy container for content which needs to be inserted repeatedly, calling `cloneNode(true)` for each insertion.

DOMBuilder provides a *`DOMBuilder.fragment()`* wrapper function, which allows you to pass all the contents you want into a DocumentFragment in one call, and also allows you make use of this functionality in HTML mode by creating equivalent *Mock DOM Objects* as appropriate. This will allow you to, for example, unit test functionality you've written which makes use of DocumentFragment objects by using HTML mode to verify output against HTML strings, rather than against DOM trees.

See http://ejohn.org/blog/dom-documentfragments/ for more information about DocumentFragment objects.

Mapping Fragments

`DOMBuilder.fragment.`**`map`**(*items*, *mappingFunction*)

Creates a fragment wrapping content created for (potentially) every item in a list.

Arguments

- **`items`** (*`Array`*) – the list of items to use as the basis for creating fragment contents.

- **mappingFunction** (*Function*) – a function to be called with each item in the list, to provide contents for the fragment.

The mapping function will be called with the following arguments:

```
mappingFunction(item, itemIndex)
```

The function can indicate that the given item shouldn't generate any content for the fragment by returning `null`.

Contents created by the function can consist of a single value or a mixed `Array`.

This function is useful if you want to generate sibling content from a list of items without introducing redundant wrapper elements.

For example, with a newforms `FormSet` object, which contains multiple `Form` objects. If you wanted to generate a heading and a table for each form object and have the whole lot sitting side-by-side in the document:

```
var formFragment = DOMBuilder.fragment.map(formset.forms, function(form, loop) {
  return [
    H2('Widget ' + (loop.index + 1)),
    TABLE(TBODY(
      TR.map(form.boundFields(), function(field) {
        return [TH(field.labelTag()), TD(field.asWidget())]
      })
    ))
  ]
})
```

Appending `formFragment` would result in the equivalent of the following HTML:

```
<h2>Widget 1</h2>
<table> ... </table>
<h2>Widget 2</h2>
<table> ... </table>
<h2>Widget 3</h2>
<table> ... </table>
...
```

CHAPTER 3

HTML Mode

HTML mode provides an output mode which generates *MockElement()* objects from *DOMBuilder.createElement()* calls and *MockFragment()* objects from *DOMBuilder.fragment()* calls.

The HTML mode API is exposed through `DOMBuilder.modes.html.api`.

Mode-specific element and convenience functions are exposed through *DOMBuilder.html*.

`DOMBuilder.`**`html`**
: Contains element functions which always create *Mock DOM Objects* (which `toString()` to HTML) and convenience functions related to *HTML Escaping*.

  New in version 2.0.

Mock DOM Objects

In HTML mode, DOMBuilder will create mock DOM objects which implement a small subset of the Node interface operations available on their real counterparts. Calling `toString()` on these objects will produce the appropriate type of HTML based on the mode at the time they and their contents were created.

With foreknowledge of the available operations (and requests for additional operations which would be useful), it's possible to write complex content creation code which works seamlessly in both DOM and HTML modes.

Mock Elements

class **`MockElement`**(*tagName*[, *attributes*[, *childNodes*]])
: A representation of a DOM Element, its attributes and child nodes.

  Arguments are as per *DOMBuilder.createElement()*.

  Changed in version 2.0: Renamed from "HTMLElement" to "MockElement"

  `MockElement.`**`appendChild`**(*node*)
  : Adds to the list of child nodes, for cases where the desired structure cannot be built up at creation time.

Changed in version 1.3: Appending a *MockFragment()* will append its child nodes instead and clear them from the fragment.

MockElement.**cloneNode**(*deep*)
Clones the element and its attributes - if deep is `true`, its child nodes will also be cloned.

New in version 1.3: Added to support cloning by an *MockFragment()*.

MockElement.**toString**([*trackEvents*])
Creates a `String` containing the HTML representation of the element and its children. By default, any `String` children will be escaped to prevent the use of sensitive HTML characters - see the *HTML Escaping* section for details on controlling escaping.

If `true` is passed as an argument and any event handlers are found in this object's attributes during HTML generation, this method will ensure the element has an `id` attribute so the handlers can be registered after the element has been inserted into the document via `innerHTML`.

If necessary, a unique id will be generated.

Changed in version 1.4: Added the optional `trackEvents` argument to support registration of event handlers post-insertion.

MockElement.**addEvents**()
If event attributes were found when `toString(true)` was called, this method will attempt to retrieve a DOM Element with this element's `id` attribute, attach event handlers to it and call `addEvents()` on any MockElement children.

New in version 1.4.

MockElement.**insertWithEvents**(*element*)
Convenience method for generating and inserting HTML into the given DOM Element and registering event handlers.

New in version 1.4.

Mock Fragments

New in version 1.3.

In HTML mode, *DOMBuilder.fragment()* will create *MockFragment()* objects which mimic the behaviour of DOM DocumentFragments when appended to another fragment or a *MockElement()*.

class MockFragment([*childNodes*])
A representation of a DOM DocumentFragment and its child nodes.

Changed in version 2.0: Renamed from "HTMLFragment" to "MockFragment"

Arguments

- **childNodes** (*Array*) – initial child nodes

MockFragment.**appendChild**(*node*)
Adds to the list of child nodes - appending another fragment will append its child nodes and clear them from the fragment.

MockFragment.**cloneNode**(*deep*)
Clones the fragment - there's no point calling this *without* passing in `true`, as you'll just get an empty fragment back, but that's the API.

MockFragment.**toString**([*trackEvents*])
Creates a `String` containing the HTML representation of the fragment's children.

Changed in version 1.4: If the `trackEvents` argument is provided, it will be passed on to any child MockElements when their *`MockElement.toString()`* method is called.

`MockFragment.`**`addEvents`**`()`
Calls *`MockElement.addEvents()`* on any MockElement children.

New in version 1.4.

`MockFragment.`**`insertWithEvents`**`(`*element*`)`
Convenience method for generating and inserting HTML into the given DOM Element and registering event handlers.

New in version 1.4.

Event Handlers and `innerHTML`

New in version 1.4.

In DOM mode, *Event Handlers* specified for an element are registered when it's being created - these are skipped when generating HTML, as we would just be inserting the result of calling `toString()` on the event handling functions, which wouldn't make any sense.

To allow you to use the same code to define event handlers regardless of which mode you're in, the mock DOM objects support passing in a flag to their `toString()` methods indicating that you'd like to register event handlers which have been specified at a later time, after you've inserted the generated HTML into the document using `innerHTML`:

```
var article = html.DIV({"class":"article"},
   html.P({id: "para1", click: function() { alert(this.id) }}, "Paragraph 1"),
   html.P({click: function() { alert(this.id) }}, "Paragraph 2")
)
document.getElementById("articles").innerHTML = article.toString(true)
```

When you pass `true` into the `toString()` call as above, DOMBuilder does two things:

1. Looks at the attributes of each element while generating HTML and determines if they contain any event handlers, storing a flag in the element if this is the case.
2. Ensures the element has an `id` attribute if event handlers were found. If an `id` attribute was not provided, a unique id is generated and stored in the element for later use.

This is the HTML which resulted from the above code, where you can see the generated `id` attribute in place:

```
<div class="article">
  <p id="para1">Paragraph 1</p>
  <p id="__DB1__">Paragraph 2</p>
</div>
```

Since we know which elements have event handlers and what their ids are, we can use that information to fetch the corresponding DOM Elements and register the event handlers - you can do just that using *`MockElement.addEvents()`*:

```
article.addEvents()
```

Now, clicking on either paragraph will result in its id being alerted.

DOMBuilder also provides a bit of sugar for performing these two steps in a single call, *`MockElement.insertWithEvents()`*:

```
article.insertWithEvents(document.getElementById("articles"))
```

HTML Escaping

HTML mode was initially introduced with backend use in mind - specifically, for generating forms and working with user input. As such, autoescaping was implemented to protect the developer from malicious user input. The same can still apply on the frontend, so *MockElement.toString()* automatically escapes the following characters in any `String` contents it finds, replacing them with their equivalent HTML entities:

```
< > & ' "
```

If you have a `String` which is known to be safe for inclusion without escaping, pass it through *DOMBuilder.html.markSafe()* before adding it to a *MockElement()*.

`DOMBuilder.html.`**`markSafe`**`(`*value*`)`

Arguments

- **value** (*String*) – A known-safe string.

Returns A `SafeString` object.

There is also a corresponding method to determine if a `String` is already marked as safe.

`DOMBuilder.html.`**`isSafe`**`(`*value*`)`

Returns `true` if the given `String` is marked as safe, `false` otherwise.

Assuming we're in HTML mode, this example shows how autoescaping deals with malicious input:

```
>>> var input = "<span style=\"font-size: 99999px;\" onhover=\"location.href=
↪'whereveriwant'\">Free money!</span>"
>>> P("Steve the dog says: ", input).toString()
"<p>Steve the dog says: &lt;span style=&quot;font-size: 99999px;&quot; onhover=&quot;
↪location.href=&#39;whereveriwant&#39;&quot;&gt;Free money!&lt;/span&gt;</p>"
```

But say you have a `String` containing HTML which you trust and do want to render, like a status message you've just created, or an `XMLHTTPRequest` response:

```
>>> var html = DOMBuilder.html
>>> var response = 'You have <strong>won the internet!</strong>'
>>> html.P('According to our experts: ', response).toString()
'<p>According to our experts: You have &lt;strong&gt;won the internet!&lt;/strong&gt;
↪</p>'
>>> html.P('According to our experts: ', html.markSafe(response)).toString()
'<p>According to our experts: You have <strong>won the internet!</strong></p>'
```

Warning: `String` operations performed on a `String` which was marked safe will produce a `String` which is no longer marked as safe.

To avoid accidentally removing safe status from a `String`, try not to mark it safe until it's ready for use:

```
>>> var response = '<span style="font-family: Comic Sans MS">Your money is safe with␣
↪us!</span>'
>>> function tasteFilter(s) { return s.replace(/Comic Sans MS/gi, 'Verdana') }
```

```
>>> var safeResponse = html.markSafe(response)
>>> html.P('Valued customer: ', safeResponse).toString()
'<p>Valued customer: <span style="font-family: Comic Sans MS">Your money is safe with␣
↪us!</span></p>'
>>> html.P('Valued customer: ', tasteFilter(safeResponse)).toString()
'<p>Valued customer: &lt;span style=&quot;font-family: Verdana&quot;&gt;Your money is␣
↪safe with us!&lt;/span&gt;</p>'
```

CHAPTER 4

News for DOMBuilder

Version 2.0.0

July 17th, 2011

- Output modes are now pluggable, using `DOMBuilder.addMode`.
- Output mode specific element functions are now available under `DOMBuilder.dom` and `DOMBuilder.html`.
- HTML Mode no longer has any dependency on DOM Mode.
- Updated attribute-setting code based on jQuery 1.6.2.
- Nested Array representations of HTML can now be used to generate output with an output mode, using `DOMBuilder.build`.
- Nested Array structures can be built using element functions under `DOMBuilder.array`.
- Added support for new tags defined in HTML 5.
- You can now specify a mode for `DOMBuilder.apply`, which will also apply any additional API for the specified mode, if available.

Backwards-incompatible changes:

- When calling `DOMBuilder.map`, the default attributes argument is now required - flexible arguments are now handled by the `map` functions exposed on element creation functions.
- `DOMBuilder.map` now passes a loop status object to the given mapping function instead of an index.
- The context argument object to `DOMBuilder.apply` is now required.
- `DOMBuilder.apply` no longer adds an `NBSP` property.
- HTML mode mock DOM objects were renamed to `MockElement` and `MockFragment`.
- HTML mode no longer supports XHTML-style closing slashes for empty elements.

- `markSafe` and `isSafe` moved to `DOMBuilder.html.markSafe` and `DOMBuilder.html.isSafe`, respectively.

Version 1.4.4

May 19th, 2011

- Additional arguments can now be passed in to `withMode` to be passed into the function which will be called.

Version 1.4.3

April 26th, 2011

- Fixed defect doing child checks on `null` and `undefined` children.

Version 1.4.2

April 12th, 2011

- Added support for using the `innerHTML` attribute to specify an element's entire contents consistently in DOM and HTML modes.

Version 1.4.1

March 4th, 2011

- Fixed HTML mode bug: event registration now works for nested elements.
- DOMBuilder can now be used as a Node.js module, defaulting to HTML mode.
- Fixed bug: SafeString is no longer used as an attributes object if passed as the first argument to an element creation function.

Version 1.4

February 13th, 2011

- Fixed HTML escaping bugs: attribute names and unknown tag names are now escaped.
- A new `insertWithEvents` method on DOMBuilder.HTMLElement attempts to use `innerHTML` in a cross-browser friendly fashion. It's safe to use this method on elements for which innerHTML is readonly, as it drops back to creating DOM Elements in a new element and moving them. If jQuery is available, its more comprehensive `html` function is used.
- Fixed issue #1 - HTML mode now supports registering event listeners, specified in the same way as DOM mode, after HTML has been inserted with `innerHTML`. If necessary, `id` attributes will be generated in order to target elements which need event listeners.
- Fixed issue #3 - jQuery is now optional, but will be made use of if present.

Version 1.3

February 4th, 2011

- Tag names passed into `DOMBuilder.HTMLElement` are now lower-cased.
- Added `DOMBuilder.elementFunctions` to hold element creation functions instead of creating them every time `DOMBuilder.apply()` is called. This also allows for the possibility of using a `with` statement for convenience (not that you should!) instead of adding element creation functions to the global scope.
- Added `DOMBuilder.fragment.map()` to create contents from a list of items using a mapping function and wrap them in a single fragment as siblings, negating the need for redundant wrapper elements.
- Fixed (Google Code) issue #5 - added `HTMLFragment.toString()`.
- Fixed (Google Code) issue #3 - we now append "nodey" contents (anything with a truthy `nodeType`) directly and coerce everything else to `String` when appending child nodes, rather than checking for types which should be coerced to `String` and appending everything else directly.
- Bit the bullet and switched to using jQuery for element creation and more. DOMBuilder now depends on jQuery >= 1.4.
- Fixed (Google Code) issue #2 - nested `Array` objects in child arguments to `DOMBuilder.createElement()` and `DOMBuilder.fragment()` are now flattened.
- Extracted `HTMLNode` base class to contain common logic from `HTMLElement` and `HTMLFragment`.
- Renamed `Tag` to `HTMLElement`.
- `DOMBuilder.fragment` now works in HTML mode - `DOMBuilder.HTMLFragment` objects lightly mimic the DOM DocumentFragment API.
- Added `DOMBuilder.map()` to create elements based on a list, with an optional mapping function to control if and how resulting elements are created.
- Added `DOMBuilder.fragment()`, a utility method for creating and populating DocumentFragment objects.

Version 1.2

January 21st, 2011

- Created Sphinx docs.
- `Tag` objects created when in HTML mode now remember which mode was active when they were created, as they may not be coerced until a later time, when the mode may have changed.
- Added `DOMBuilder.withMode()` to switch to HTML mode for the scope of a function call.
- Fixed short circuiting in element creation functions and decreased the number of checks required to determine which of the 4 supported argument combinations the user passed in.
- Attributes are now lowercased when generating HTML.
- `DOMBuilder.isSafe()` and `DOMBuilder.markSafe()` added as the public API for managing escaping of strings when generating HTML.
- Added support for using the DOMBuilder API to generate HTML/XHTML output instead of DOM elements. This is an experimental change for using the same codebase to generate HTML on the backend and DOM elements on the frontend, as is currently being implemented in https://github.com/insin/newforms

Version 1.1

October 10th, 2008

- An `NBSP` property is now also added to the context object by `DOMBuilder.apply()`, for convenience.
- `Boolean` attributes are now only set if they're `true`. Added items to the demo page to demonstrate that you can now create an explicitly unchecked checkbox and an explicitly non-multiple select.
- Added more IE workarounds for:
  - Creating multiple selects
  - Creating pre-selected radio and checkbox inputs

Version 1.0

June 1st, 2008

- Added support for passing children to element creation function as an `Array`.
- Added more robust support for registering event handlers, including cross-browser event handling utility methods and context correction for IE when the event handler is fired.
- IE detection is now performed once and once only, using conditional compilation rather than user-agent `String` inspection.

CHAPTER 5

License

Originally based on DomBuilder

OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Fallback attribute setting code based on `jQuery.attr`

CHAPTER 6

Quick Guide

DOMBuilder provides a convenient, declarative API for generating HTML elements, via objects which contain functions named for the HTML element they create:

```
with(DOMBuilder.dom) {
  var article =
    DIV({'class': 'article'}
    , H2('Article title')
    , P('Paragraph one')
    , P('Paragraph two')
    )
}
```

Every element function also has a `map` function attached to it which allows you to easily generate content from a list of items:

```
var el = DOMBuilder.html
function shoppingList(items) {
  return el.OL(el.LI.map(items))
}
```

```
>>> shoppingList(['Cheese', 'Bread', 'Butter'])
<ol><li>Cheese</li><li>Bread</li><li>Butter</li></ol>
```

You can control `map` output by passing in a callback function:

```
function opinionatedShoppingList(items) {
  return el.OL(el.LI.map(function(item, attrs, loop) {
    if (item == 'Cheese') attrs['class'] = 'eww'
    if (item == 'Butter') return el.EM(item)
    return item
  })
}
```

```
>>> opinionatedShoppingList(['Cheese', 'Bread', 'Butter'])
<ol><li class="eww">Cheese</li><li>Bread</li><li><em>Butter</em></li></ol>
```

If you want to use this API to go straight to a particular type of output, you can do so using the functions defined in *DOMBuilder.dom* and *DOMBuilder.html*, as demonstrated above.

If you want to be able to switch freely between output modes, or you won't know which kind of output you need until runtime, you can use the same API via *DOMBuilder.elements*, controlling what it outputs by setting the *DOMBuilder.mode* flag to `'dom'` or `'html'`, or calling a function which generates content using *DOMBuilder.withMode()*:

```
var el = DOMBuilder.elements
function shoutThing(thing) {
  return el.STRONG(thing)
}
```

```
>>> DOMBuilder.mode = 'html'
>>> shoutThing('Hello!').toString()
<strong>Hello!</strong>
>>> DOMBuilder.withMode('dom', shoutThing, 'Hey there!')
[object HTMLStrongElement]
```

This is useful for writing libraries which need to support outputting both DOM Elements and HTML Strings, or for unit-testing code which normally generates DOM Elements by flipping the mode in your tests to switch to HTML String output.

DOMBuilder also supports using its output modes with another common means of defining HTML in JavaScript code, using nested lists (representing elements and their contents) and objects (representing attributes), like so:

```
var article =
  ['div', {'class': 'article'}
  , ['h2', 'Article title']
  , ['p', 'Paragraph one']
  , ['p', 'Paragraph two']
  ]
```

You can generate output from one of these structures using *DOMBuilder.build()*, specifying the output mode:

```
>>> DOMBuilder.build(article, 'html').toString()
<div class="article"><h2>Article title</h2><p>Paragraph one</p><p>Paragraph two</p></
↪div>

>>> DOMBuilder.build(article, 'dom').toString()
[object HTMLDivElement]
```

You can also generate these kinds of structures using the element functions defined in *DOMBuilder.array*.

This is just a quick guide to what DOMBuilder can do - dive into the rest of the documentation to find out about the rest of its features, such as:

- Registering *Event Handlers*.
- Making it more convenient to work with *Event Handlers and innerHTML*.
- Populating *Document Fragments* with content in a single call.
- Being able to use fragments in HTML mode via *Mock DOM Objects*.
- *HTML Escaping* in HTML mode.

CHAPTER 7

Installation

Browsers

DOMBuilder is a modular library, which supports adding new output modes and feature modes as plugins.

The available components are:

DOMBuilder.js Core library

DOMBuilder.dom.js DOM output mode - adds `DOMBuilder.dom`

DOMBuilder.html.js HTML output mode - adds `DOMBuilder.html`

Compressed Builds

Multiple preconfigured, compressed builds of DOMBuilder are available to suit various needs:

DOM and HTML For creation of mixed content, with *DOM Mode* as the default output format.

DOM only For creation of DOM Elements, with *DOM Mode* as the default output format.

HTML only For creation of HTML Strings, with *HTML Mode* as the default output format.

Dependencies

There are no *required* dependencies, but if jQuery (>= 1.4) is available, DOMBuilder will make use of it when creating DOM Elements and setting up their attributes and event handlers.

If not, DOMBuilder will fall back to using some less comprehensive workarounds for cross-browser DOM issues and use the traditional event registration model for compatibility.

Changed in version 1.4: jQuery was made optional, with the caveat that cross-browser support will be less robust.

Node.js

New in version 1.4.1.

DOMBuilder can be installed as a Node.js module using Node Package Manager. The Node.js build includes *HTML Mode* and has HTML as the default output format.

Install:

```
npm install DOMBuilder
```

Import:

```
var DOMBuilder = require('DOMBuilder')
```

Index

D

M