

# Building XML Editing Applications with Xonomy

Michal Boleslav Měchura

This document is version 1.0 and was released 7 August 2014.  
It applies to Xonomy version 1.0 which was also released 7 August 2014.

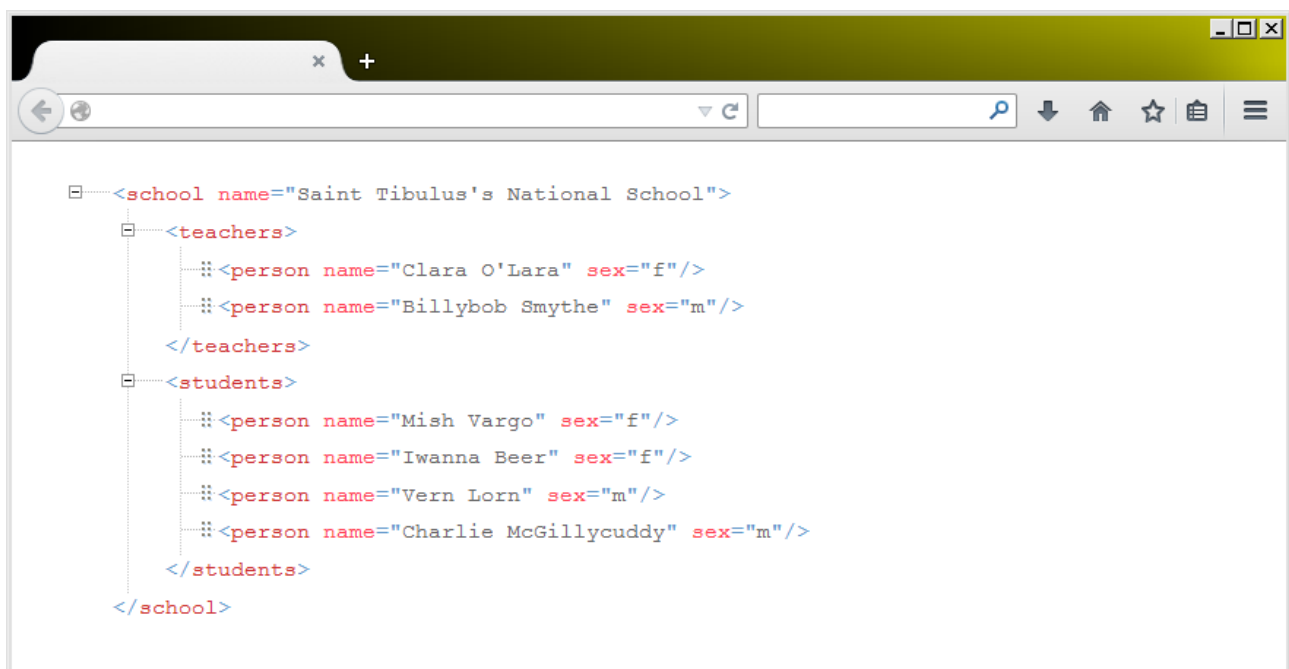
0.Introducing Xonomy.....	2
1.Quick-start guide.....	4
2.Controlling the structure of documents.....	6
3.Editing attribute values.....	11
4.Controlling the contents of menus.....	14
5.Controlling the order of elements and attributes.....	18
6.Dragging and dropping.....	19
7.Controlling the display of elements and their children.....	20
8.Working with mixed content.....	22
9.Validating the document.....	26
10.Working with surrogate objects.....	29
11.Working with namespaces.....	32
12.Document specification reference.....	33

## 0. Introducing Xonomy

Xonomy is a web-based, schema-driven XML editor, written entirely in JavaScript and embeddable in any web page. Xonomy emulates the look and feel of a text editor with syntax highlighting, code folding and autocompletion. It is, however, not a text editor: you edit your XML documents by clicking on nodes, selecting options from context menus, selecting attribute values from picklists, dragging and dropping elements around, and so on.

You can customize Xonomy by giving it a **document specification**. A document specification is similar to a schema: it determines what actions the user can perform on each element, which attributes or child elements the user can create, where attribute values come from, and so on. This gives you a mechanism for constraining the structure of your XML documents which is roughly equivalent to a Document Type Definition (DTD). You can constrain your document further by attaching your own **validation functions** in JavaScript.

Xonomy can handle both **data-centric** XML, which it displays in a tree structure, and **document-centric** XML, which is displayed as running text with inline XML markup. Xonomy is able to handle **mixed content** (elements which contain a mixture of text and elements) and has features for annotating text with inline XML. Xonomy can work with documents that use **namespaces**.



### 0.1. Who Xonomy is for

Xonomy isn't a shrink-wrapped product for end-users. It is a UI component which software developers can use as a building block in their own applications. If you are building a web-based application where your users will need to edit XML documents (or any structured data that can be rendered as XML), then Xonomy will make your job a lot easier. You will no longer need to painstakingly handcraft a user interface for each document type. Instead, you can simply embed Xonomy in your HTML pages and write a few lines of JavaScript to provide a document specification. Xonomy will take care of all the remaining on-screen magic.

## 0.2. Xonomy's vital statistics

### Compatibility

Xonomy should work on all reasonably recent versions of all reasonably modern browsers. It is positively known to work on (= has been tested on) Mozilla Firefox version 30 and Microsoft Internet Explorer version 11, both on Windows 7.

### Dependencies

Xonomy uses the jQuery library, which it expects to find in the `$` variable. If you don't know what jQuery is, find out here: <http://jquery.com/>

### Known limitations

Xonomy can only handle one XML editor and one XML document per page.

### Licence

Xonomy is available to you under the Creative Commons 'Attribution 4.0 International' licence: <http://creativecommons.org/licenses/by/4.0/>

This is a very permissive licence allowing you to extend or reuse Xonomy without payment and for any purpose, including commercial ones. The only restriction is that you acknowledge your use of it. A good place to do that is on your website's 'about' page or in the webpage footer. Say something along the lines of "this website uses Xonomy" and provide a link to Xonomy's homepage: <http://lexiconista.com/xonomy/>

The Creative Commons licence applies to the Xonomy software but not this document, which is copyrighted by the author.

### Contacting the author

You can e-mail me at [valselob@gmail.com](mailto:valselob@gmail.com). Bug reports and feature requests are welcome. Also, I'd really love hear how people are using Xonomy: send me URLs and screenshots!

### Acknowledgments

Xonomy uses icons from <http://famfamfam.com>.

## 0.3. How this document is structured

Chapters 1 and 2 together form the minimum you need to know in order to start using Xonomy in your applications – so these are the chapters you should absolutely not skip.

Chapters 3 through to 11 deal with specialized topics and you can dip into them as and when you need. Although, if you have the time and headspace available, it's probably a good idea to read them all in sequence because they will expose you to Xonomy's design principles: you will re-emerge at the end of chapter 11 with a good understanding of how Xonomy 'ticks'.

Finally, chapter 12 is a reference to the properties of a document specification, so you can find them all in one place instead of having to fumble for them in the other chapters.

# I. Quick-start guide

## I.1. Installing

To start using Xonomy on your website, download Xonomy from <http://lexiconista.com/xonomy/>, uncompress it into a folder somewhere on your website, and include the files `xonomy.js` and `xonomy.css` in your HTML page. You must also include jQuery.

## I.2. Rendering

To open an XML document, you must tell Xonomy to **render** it into a `div` (or some other HTML element). You do this by calling the function `Xonomy.render`. Once you've done all this, your user can start interacting with it.

The function `Xonomy.render` takes three arguments:

- the XML document, which can be XML-as-string or an actual `XMLDocument` object,
- a reference to the `div` where you want to render it,
- and a **document specification**. A document specification is a JavaScript object which tells Xonomy everything it needs to know about the structure of the document and the actions the user is allowed to perform on it. It can be `null`, in which case your user will be able to look at the document but will not be able to do anything to it. More about document specifications in chapter 2.

## I.3. Example

```
<!DOCTYPE HTML>
<html>
  <head>
    <script type="text/javascript" src="jquery-1.10.2.min.js"></script>
    <script type="text/javascript" src="/xonomy/xonomy.js"></script>
    <link type="text/css" rel="stylesheet" href="/xonomy/xonomy.css"/>
    <script type="text/javascript">
      function start() {
        var xml="<list><item label='one'/><item label='two'/></list>";
        var editor=document.getElementById("editor");
        Xonomy.render(xml, editor, null);
      }
    </script>
  </head>
  <body onload="start()">
    <div id="editor"></div>
  </body>
</html>
```

```
└─> <list>
    ├── <item label="one"/>
    └── <item label="two"/>
  </list>
```

## I.4. Harvesting

Once your user has finished interacting with the document, you need to **harvest** it by calling the function `Xonomy.harvest`. This function takes no arguments and returns XML-as-string.

```
function submit() {  
  var xml=Xonomy.harvest(); // "<list><item label='one'/><item label='two'/></list>"  
}
```

***“How come `Xonomy.harvest` takes no arguments? How does it know where to find the editor?”***

*The reason why no argument is necessary is that Xonomy can only handle one editor per page. So, `Xonomy.harvest` simply grabs the first editor it can find, believing it to be the only one. On that topic, don't ever try to include more than one Xonomy editor in a single HTML page: unexpected mayhem would ensue!*

## 2. Controlling the structure of documents

### 2.1. A gentle introduction to document specifications

To tell Xonomy what it needs to know about the document your user is editing, you must give Xonomy a **document specification**. This is a JavaScript object which you supply to the function `Xonomy.render` as its third argument. (If that argument is `null`, Xonomy will construct a default document specification which will allow your users to look at the document but not edit it.) The basic skeleton of a document specification looks somewhat like this:

```
var docSpec={
  elements: {
    "myElement": {
      attributes: {
        "myAttribute": {...},
        "myOtherAttribute": {...}
      }, ...
    },
    "anotherElement": {
      attributes: {
        "anotherAttribute": {...},
        "yetAnotherAttribute": {...}
      }, ...
    }, ...
  },
  onchange: function(){...},
  validate: function(obj){...}
};
```

A document specification contains, among other things, a list of the element names that can occur in the document. Each element name comes with an **element specification**: a JavaScript object which tells Xonomy various facts about elements of that name: what actions the user can perform on them, how their content should be rendered, and so on.

In turn, an element specification contains a list of attribute names the element can have. Each attribute name comes with an **attribute specification**: a JavaScript object which tells Xonomy what it needs to know about attributes of that name: what actions the user can perform on them, how the user is supposed to edit their values, and so on.

***“What happens if the document contains an element or attribute that isn’t in the document specification?”***

*Nothing terrible will happen in that situation. In Xonomy, a document specification does not have to be ‘complete’: it does not need to contain the names of all and any elements and attributes that occur in the document. If Xonomy comes across an element or attribute that cannot be found in the document specification, it will construct a default specification for it which will allow the user to look at the element/attribute but not do anything to it.*

Last but not least, a document specification can optionally contain two functions:

- An **onchange function**. Xonomy will call this function every time the user changes the document. You can use this to keep track of whether the document is ‘dirty’ (has been changed by the user since opening).

- A **validate function**. Xonomy will call this function every time the user has changed the document. You can use this to validate the document and to give feedback to your user in the form of warning messages. More about validation in chapter 9.

A complete reference to all the properties of a document specification can be found in chapter 12. Also, the other chapters in this document take a detailed look at how particular topics are handled in document specifications.

## 2.2. Example

Let's now have a look at a working example that makes use of a document specification. We will use Xonomy to let our users edit a simple XML document. The document starts with the `<list>` root element, which may contain one or more `<item>` child elements, which may optionally have a `label` attribute whose value is an arbitrary string of text. The document specification looks like this:

```
var docSpec={
  onchange: function(){
    console.log("I been changed now!")
  },
  validate: function(obj){
    console.log("I be validatin' now!")
  },
  elements: {
    "list": {
      menu: [{
        caption: "Append an <item>",
        action: Xonomy.newElementChild,
        actionParameter: "<item/>"
      }]
    },
    "item": {
      menu: [{
        caption: "Add @label=\"something\"",
        action: Xonomy.newAttribute,
        actionParameter: {name: "label", value: "something"},
        hideIf: function(jsElement){
          return jsElement.hasAttribute("label");
        }
      }, {
        caption: "Delete this <item>",
        action: Xonomy.deleteElement
      }, {
        caption: "New <item> before this",
        action: Xonomy.newElementBefore,
        actionParameter: "<item/>"
      }, {
        caption: "New <item> after this",
        action: Xonomy.newElementAfter,
        actionParameter: "<item/>"
      }
    ]},
    canDropTo: ["list"],
    attributes: {
      "label": {
        asker: Xonomy.askString,
        menu: [{
          caption: "Delete this @label",
          action: Xonomy.deleteAttribute
        }]
      }
    }
  }
}
```

```

    }
  }
}
};

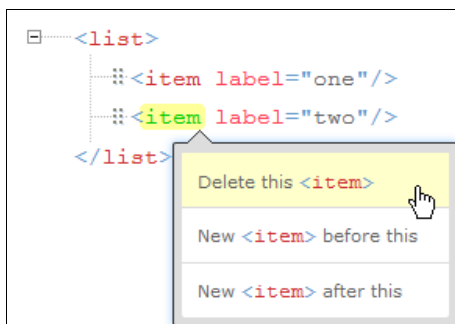
```

And the HTML page can look like this:

```

<!DOCTYPE HTML>
<html>
  <head>
    <script type="text/javascript" src="jquery-1.10.2.min.js"></script>
    <script type="text/javascript" src="/xonomy/xonomy.js"></script>
    <link type="text/css" rel="stylesheet" href="/xonomy/xonomy.css"/>
    <script type="text/javascript">
      function start() {
        var docSpec={...}; //insert docSpec here
        var xml="<list><item label='one'/><item label='two'/></list>";
        var editor=document.getElementById("editor");
        Xonomy.render(xml, editor, docSpec);
      }
      function submit() {
        var xml=Xonomy.harvest();
        //do something with xml...
      }
    </script>
  </head>
  <body onload="start()">
    <div id="editor"></div>
    <button onclick="submit()">Submit!</button>
  </body>
</html>

```



Now, let's read the document specification from top to bottom.

You see that the document specification is a hash table with three key: `onchange`, `validate` and `elements`. You already know what the first two do. The most interesting is the third one, `elements`. It too is a hash table where the keys are element names and the values are **element specifications**. You see that the document has two kinds of elements: `<list>` and `<item>`.

The `<list>` element specification is, again, a hash table. This one contains only one key, `menu`, which specifies the menu that appears when the user clicks on the element. The value is an array of **menu item specifications**. A menu item specification is again a hash table (this is getting boring!) with properties that specify what the menu item does and what its caption is. You will learn more about menus and menu item specifications in chapter 4. The menu item in this example appends a new `<item>` child element to the `<list>` element when the user clicks it. The function `Xonomy.newElementChild` takes



care of that, which is one of Xonomy's predefined **menu action functions**. Menu action functions are explained in chapter 4, and you can also write your own.

The `<item>` element specification is a little richer. Its menu has four items: one to add the `label` attribute (along with an initial value), two to add sibling `<item>` elements before and after, and one to delete the element. Notice that the first menu item – the one that inserts the attribute – has a property called `hideIf`. This is a function which causes the menu item to be hidden if it returns `true`. Here, it causes the menu item to be hidden if the element already has an attribute called `label`. The object passed to the function is an **element surrogate object**, a JavaScript object which encapsulates the current state of the element when the user clicked on it. Understanding surrogate objects is important if you want to write moderately complex document specifications, and they are explained in chapter 10. You will also see surrogate objects mentioned quite a lot before we even get to chapter 10.

The `<item>` element specification also has a `canDropTo` property. This tells Xonomy that elements of this name can be dragged-and-dropped around. Its value is an array of element names that can accept this element as a child. In this case the specification says that the `<item>` element can be dropped inside the `<list>` element. Try adding a few `<item>` elements to your document and then use the drag handle to drag them around. You will see that, while dragging, Xonomy shows you 'drop anchors' everywhere the element can be dropped.

Finally, the `<item>` element specification has an `attributes` property. This is a hash table of the names of attributes the element can have. Here it says that the element can have one attribute called `label`. The value is an **attribute specification**. The attribute specification says that the attribute has a menu with one item that allows the user to delete the attribute. Also, the attribute specification specifies how the user is expected to edit the attribute's value. The `asker` property takes care of that. It refers to a function which 'asks' the user for a value. Here it refers to `Xonomy.askString`, a predefined **asker function** in Xonomy which affords editing the value as a short one-line string. Xonomy comes with several predefined asker functions. More details about asker functions, including instructions for writing your own, are explained in chapter 3.

Well, congratulations, we've gotten to the end of this example! You now have a good idea of how Xonomy works. The rest of this document will delve deeper into some of the topics we touched above.

***"This document specification looks nothing like a DTD or an XML Schema! It doesn't even say what the root element is or what children it can have! What's going on?"***

Xonomy's document specification isn't meant to be a schema against which you could validate a document. Instead, it is intended as a specification of the actions and operations the user is allowed to perform on each element and attribute. In other words, a document specification in Xonomy is less abstract and more down-to-earth than a schema. It is not a schema, but a schema is implicit in it.

Take for example the question 'how does Xonomy know that the root element is `<list>`'. This isn't stated explicitly anywhere in the document specification, but it is implied by the fact that the initial XML document you are passing to `Xonomy.render` has `<list>` as root element and that the document specification doesn't allow the user to change or rename it.

Similarly, take the question 'how does Xonomy know what children `<list>` can have'. Although the document specification doesn't say anywhere that a `<list>` can have zero or more `<item>` elements as its children, this is implicit in the fact that the document specification allows the user to add and delete `<item>` elements to/from the `<list>` element.

So it turns out that a Xonomy document specification allows you to do pretty much everything a DTD or an XML Schema does, only in a more verbose and less abstract way. The verbosity is compensated by extra flexibility: you have direct control over the menus that appear when the user clicks on something, you have direct control over how the user is allowed to edit values, and so on.

**Note**

*In theory, it is possible to write a script that generates a Xonomy document specification from a DTD or from an XML Schema, but that would be a separate project.*

### 3. Editing attribute values

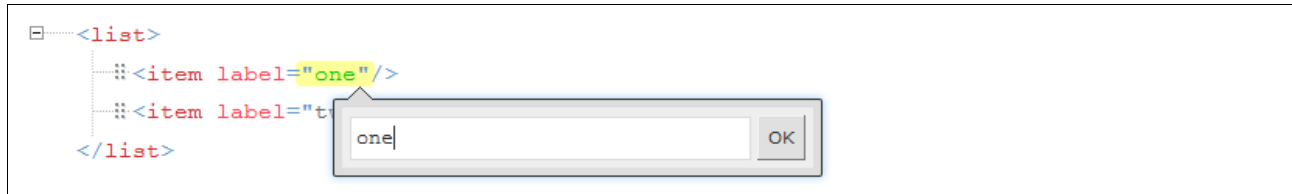
When a user clicks an attribute value, a pop-up box appears for the user to edit the value in. You have control over the contents of that box. Each attribute specification has a property called `asker`; this refers to a function which ‘asks’ the user for a value. An asker function returns HTML-as-string which Xonomy inserts into the pop-up box. Some asker functions take an additional argument, which you can assign to the attribute specification’s `askerParameter` property.

If you fail to assign anything to the `asker` property, Xonomy will treat the attribute value as read-only, preventing the user from editing it.

Xonomy comes with three predefined asker functions. We will have a look at them in the next three subchapters. and then we’ll explain how you can write your own asker functions.

#### 3.1. Xonomy.askString

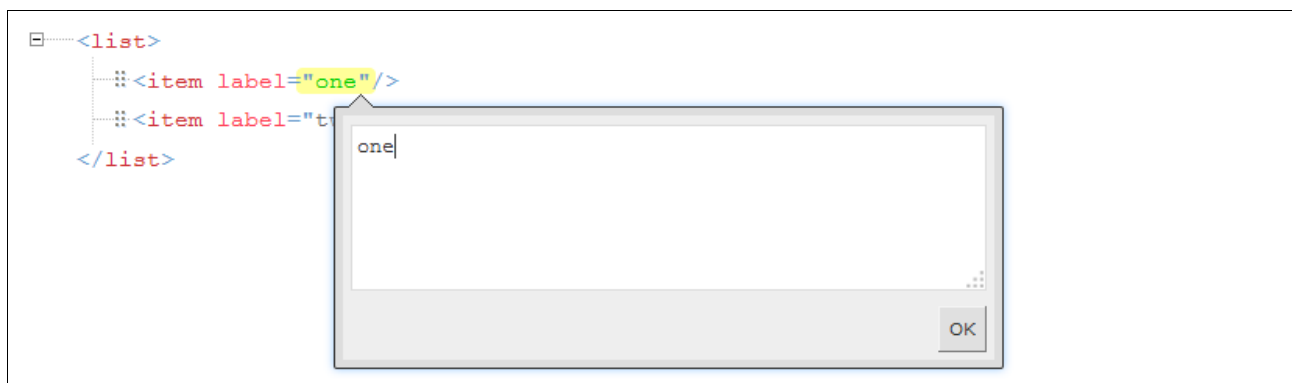
```
attributes: {  
  "label": {  
    asker: Xonomy.askString  
  }  
}
```



This function allows the user to edit the value as a single-line string.

#### 3.2. Xonomy.askLongString

```
attributes: {  
  "label": {  
    asker: Xonomy.askLongString  
  }  
}
```



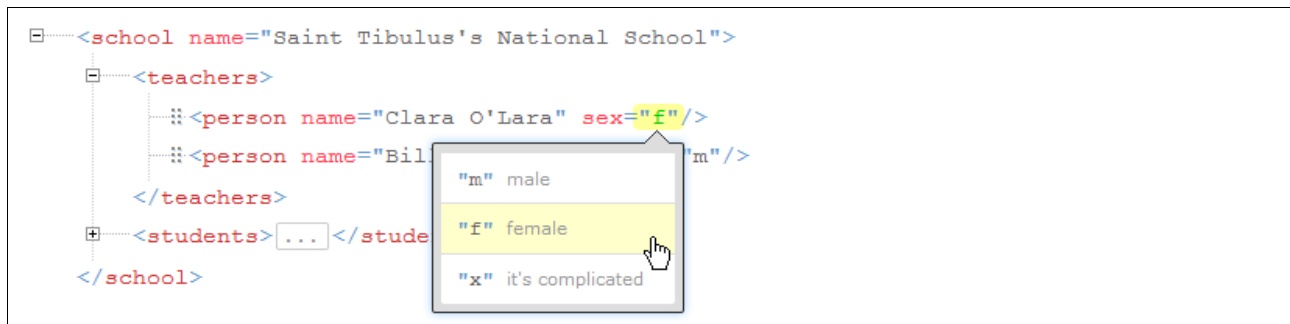
This function allows the user to edit the value as a multi-line string which may contain line breaks.

#### **Note**

*This function is also used by Xonomy for editing text nodes.*

### 3.3. Xonomy.askPicklist

```
attributes: {  
  "label": {  
    asker: Xonomy.askPicklist,  
    askerParameter: [  
      {value: "m", caption: "male"},  
      {value: "f", caption: "female"}  
    ]  
  }  
}
```



This function allows the user to pick the value from a list. You must assign the list to the attribute specification's `askerParameter` property. The list can be:

- an array of strings,  
 `["m", "f"]`
- or an array of objects where each has a value and an explanatory caption,  
 `[{value: "m", caption: "male"}, {value: "f", caption: "female"}]`
- or a combination of both.  
 `[{value: "m", caption: "male"}, "f"]`

### 3.4. Writing your own asker functions

If the three predefined asker functions are not enough for you, you can create your own function and assign it to the `asker` property of an attribute specification. The function must take at least one and at most two arguments:

1. As a first argument, Xonomy passes the current value of the attribute to your function. This is a string.
2. Whatever you assign to the attribute specification's `askerParameter` property will be passed to the asker function as a second argument. You have seen that the `Xonomy.askPicklist` function uses this argument to know the contents of the picklist.

The function must return HTML-as-string, which Xonomy will insert into the pop-up box. The HTML may contain an element with the class name `focusme`; Xonomy will automatically move input focus to the first such element it finds in the pop-up box.

Your user will interact with the contents of the pop-up box and, at some stage, he or she will presumably click on something or press a key to indicate that he or she wishes to set the attribute's value. At that stage, your HTML must call the function `Xonomy.answer` whose one and only argument is the new value. This must again be a string.

A good way to understand how an asker function is supposed to work is to look at the source code of one of Xonomy's predefined asker functions – such as `Xonomy.askString`, which is reproduced for you here.

```
Xonomy.askString=function(defaultString) {  
    var html="";  
    html+="    html+="    html+=" <input type='submit' value='OK'>";  
    html+="</form>";  
    return html;  
};
```

## 4. Controlling the contents of menus

When a user clicks the name of an element or attribute, a menu appears, presetting the user with actions he or she can perform on the element or attribute. You have control over what appears in those menus. Element and attribute specifications have a property called `menu`; this is an array of objects called **menu item specifications**. Each menu item specification is a hash table with several properties that describe the menu item.

```
menu: [
  {
    caption: "...",
    action: ...,
    actionParameter: ...,
    hideIf: ...
  },
  ...
]
```

### Note

*In addition to element and attribute menus, Xonomy has inline menus which appear when the user has selected a stretch of text in a text node. These menus are found in an element specification's `inlineMenu` property. We will deal with this in a separate chapter devoted to inline markup and mixed content (chapter 8).*

### 4.1. Captions

```
caption: "Delete this element"
```

Each menu item has a `caption` containing a human-readable string that appears on the menu. If you quote XML tags and attributes in the caption, Xonomy will format them in a mono-space font; just make sure that tags are enclosed in angle brackets and attributes start with the `@` character:

```
"Add a new <item>"
"Add a new <item @label=\"something\">"
"Add a new @label"
"Add a new @label=\"something\""
```

### 4.2. Actions and action parameters

```
action: Xonomy.newAttribute,
actionParameter: {name: "label", value: "something"}
```

The `action` property is a reference to an **action function**: a function that does something to the element or attribute on which it has been invoked. Most action functions need some additional data to do their work, and that data is to be found in the `actionParameter` property. Xonomy comes with several predefined action functions and you can also write your own. The following is a list of the predefined action functions. Instructions for writing your own functions can be found at the end of this chapter.

#### **Xonomy.deleteAttribute**

Can be used in: attribute menus.

Deletes the attribute.

#### **Xonomy.newAttribute**

Can be used in: element menus.

Adds a new attribute to the element.

For `actionParameter`, assign `{name: "", value: ""}` where `name` is the name of the attribute and `value` is its initial value (which can be an empty string, of course).

#### **Xonomy.newElementChild**

Can be used in: element menus.

Adds a new element as a child of the current element. If the current element already has children, the new child is added to the end (while respecting constraints mandated by `mustBeBefore` and `mustBeAfter`, if any – for that, see chapter 5).

For `actionParameter`, assign XML-as-string: `"<element>...</element>"`. The XML can contain attributes, child nodes etc. Make sure the XML is well-formed!

#### **Xonomy.newElementBefore**

Can be used in: element menus.

Adds a new sibling element before the current element.

For `actionParameter`, assign XML-as-string: `"<element>...</element>"`. The XML can contain attributes, child nodes etc. Make sure the XML is well-formed!

#### **Xonomy.newElementAfter**

Can be used in: element menus.

Adds a new sibling element after the current element.

For `actionParameter`, assign XML-as-string: `"<element>...</element>"`. The XML can contain attributes, child nodes etc. Make sure the XML is well-formed!

#### **Xonomy.deleteElement**

Can be used in: element menus.

Deletes the element.

#### **Note**

*Additionally, Xonomy has two predefined action functions for working with inline markup: `Xonomy.wrap` and `Xonomy.unwrap`. These are explained in the chapter that deals with inline markup, which is chapter 8.*

### **4.3. Hiding menu items**

The `hideIf` property is a function which causes the menu item to be hidden if it returns `true`. A typical use for this is to hide an attribute-insertion menu item if the element already has such an attribute:

```
{
  caption: "Add a @label",
  action: Xonomy.newAttribute,
  actionParameter: {name: "label", value: ""},
  hideIf: function(jsElement){
    return jsElement.hasAttribute("label");
  }
}
```

```
}
```

Another popular option is to hide a child-insertion menu item if the parent already has such a child (this is how you can make sure, for example, that a `<list>` always only has up to one `<item>`, if that is what your schema dictates):

```
{
  caption: "Add an <item>",
  action: Xonomy.newElementChild,
  actionParameter: "<item/>",
  hideIf: function(jsElement) {
    return jsElement.hasChildElement("item");
  }
}
```

The argument that Xonomy passes to the `hideIf` function (called `jsElement` in the examples above) is an **element surrogate object** (if we are in an element menu) or an **attribute surrogate object** (if we are in an attribute menu). Surrogate objects are objects which encapsulate the state of the element or attribute at the time the user has clicked the menu item. They have various properties which you can test against. More about surrogate objects in chapter 10.

## 4.4. Writing your own action functions

If Xonomy's predefined action functions are not enough for you, you can write your own. An action function is a function that takes two arguments and returns nothing.

- The first argument is a unique ID of the element or attribute in which the action is being invoked. Xonomy assigns a unique ID (usually called `htmlID` in Xonomy's source code; it is a string like `"xonomy1"`, `"xonomy2"` etc.) to every element, attribute and text node in your XML document. This is how your action function knows which object the action is being applied to.
- The second argument is whatever you have assigned to the `actionParameter` property – which can be nothing, of course.

### Example I

The easiest way to write your own action functions is to reuse Xonomy's predefined action functions. Let's demonstrate that on an example. Let's say you want to have a menu item on one of your elements that inserts two attributes at once. You can write an action function that simply calls `Xonomy.newAttribute` twice. This is what the menu item would like:

```
{
  caption: "Insert @firstname and @surname",
  action: insertTwoAttributes,
  actionParameter: {name1: "firstname", name2: "surname"},
  hideIf: function(jsElement){
    return jsElement.hasAttribute("firstname")
      || jsElement.hasAttribute("surname");
  }
}
```

And this what the action function would look like:

```
function insertTwoAttributes(htmlID, param) {
  Xonomy.newAttribute(htmlID, {name: param.name1, value: ""});
  Xonomy.newAttribute(htmlID, {name: param.name2, value: ""});
}
```

Let's describe step-by-step what's happening here. You have a menu item with the caption 'Insert firstname and surname'. When the user clicks this menu item, Xonomy calls the function



`insertTwoAttributes` and passes two arguments to it: `htmlID` is the ID of the current element and `param` is what you have in the menu item's `actionParameter`. Your function then calls `Xonomy.newAttribute` twice, which actually creates the attributes.

Notice that, when calling Xonomy's predefined action functions, they each take two arguments: an `htmlID`, and a parameter object as specified in the reference section above (chapter 4.2).

## Example 2

If the action function you have in mind cannot be implemented by reusing Xonomy's predefined action functions, you can take a different approach:

1. First, 'harvest' the element or attribute into a surrogate object.
2. Then, manipulate the surrogate object to make whatever changes you want.
3. Finally, tell Xonomy to replace the original element or attribute with the surrogate object.

Let's say, for example, that you want to have an item on an element's menu to rename the element. This is what the menu item would look like:

```
{
  caption: "Rename to <listing>",
  action: renameElement,
  actionParameter: "listing"
}
```

And this is what the action function would look like:

```
function renameElement(htmlID, param) {
  var div=document.getElementById(htmlID);
  var jsElement=Xonomy.harvestElement(div);
  jsElement.name=param;
  Xonomy.replace(htmlID, jsElement);
}
```

Let's describe step-by-step what's happening here. You have a menu item with the caption 'Rename to <listing>'. When the user clicks this menu item, Xonomy calls the function `renameElement` and passes two arguments to it: `htmlID` is the ID of the current element and `param` is what you have in the menu item's `actionParameter`. Your function then:

1. obtains the `div` in your webpage that the element has been rendered to,
2. asks Xonomy to harvest this `div` into a surrogate element object (you must use `Xonomy.harvestElement`, `Xonomy.harvestAttribute` or `Xonomy.harvestText` depending on whether the node you want to harvest is an element, an attribute or a text node),
3. changes the surrogate object's `name` property, effectively renaming the surrogate element object
4. and, finally, asks Xonomy to replace the existing element with the surrogate object.

The function `Xonomy.replace` is a function you can call from anywhere to replace an existing object in your XML document (element, attribute or text node) with another one. It takes the `htmlID` of the existing object and a surrogate object representing the new one.

As usual, this action function makes use of surrogate objects. We have not looked at surrogate objects in any systematic way yet in this document, but we will get to them in chapter 10.

## 5. Controlling the order of elements and attributes

### 5.1. Attribute order

Xonomy automatically makes sure the attributes of an element are listed in the order in which they are given in the element specification. If they are not, Xonomy reorders them. This happens everytime the element is rendered (including when you initially render the document by calling `Xonomy.render`) and everytime an attribute is added.

### 5.2. Child element order

Unlike attributes, elements are not required by Xonomy to appear in any particular order by default. However, you can specify in a element's specification that the element must always appear **before** or **after** another element. You do this by setting the element specification's `mustBeBefore` and `mustBeAfter` properties. Both are arrays of element names. Example:

```
var docSpec={
  elements: {
    "introduction": {
      mustBeBefore: ["exposition", "conclusion"],
      ...
    },
    "exposition": {
      ...
    },
    "conclusion": {
      mustBeAfter: ["introduction", "exposition"],
      ...
    },
    ...
  }
}
```

A `mustBeBefore` array means that, if the current element has a sibling whose name is listed in `mustBeBefore`, then the current element must be its preceding sibling. Conversely, a `mustBeAfter` array means that, if the current element has a sibling whose name is listed in `mustBeAfter`, then the current element must be its following sibling. Note that the terms 'preceding' and 'following' do not necessarily mean 'immediately preceding' and 'immediately following'; there may be intervening elements.

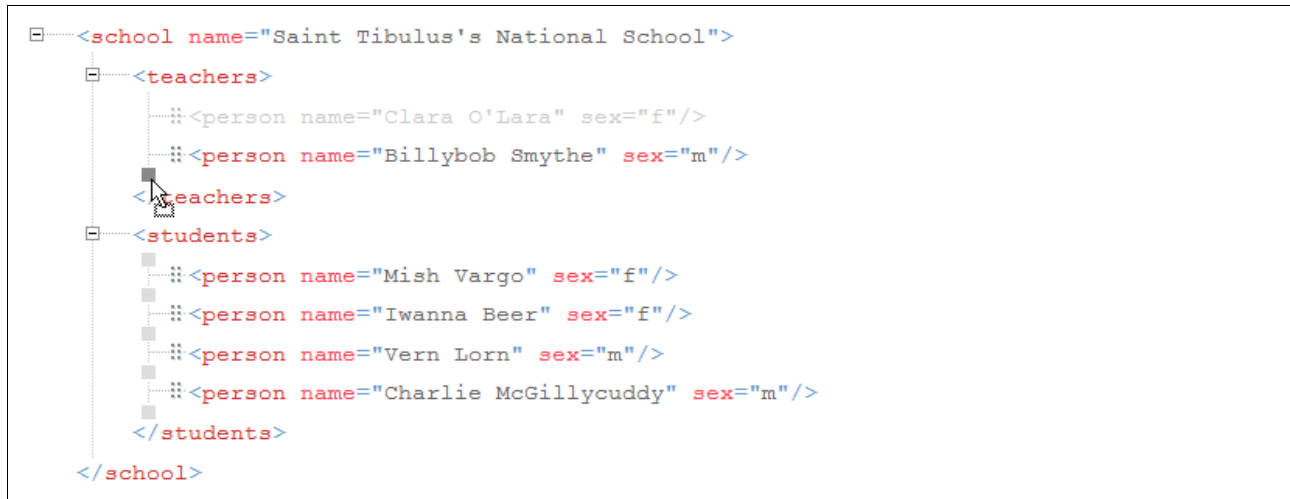
The `mustBeBefore` and `mustBeAfter` constraints are imposed:

- When the parent element is rendered (including when you initially render the document by calling `Xonomy.render`) and when a new child element is added to a parent element. The elements are reordered if necessary to satisfy the constraints. (If the constraints are mutually inconsistent, nothing terrible will happen: Xonomy will first move the element to satisfy its `mustBeBefore` constraint, then move it again to satisfy its `mustBeAfter` constraint, and then go on the next element.)
- When dragging and dropping: Xonomy displays drop anchors in places where the dragged element can be dropped, and these anchors respect `mustBeBefore` and `mustBeAfter` constraints. See chapter 6 for more about dragging and dropping.

If the `mustBeBefore` and `mustBeAfter` formalism is not flexible enough to impose ordering restrictions in your application, you can always write your own validation functions and notify the user when he or she has violated a constraint. More about validation functions in chapter 9.

## 6. Dragging and dropping

Xonomy allows you to specify that certain elements can be dragged and dropped with the mouse. Draggable elements are displayed with a 'drag handle' which the user can grab and drag to somewhere else in the document. While the element is being dragged, Xonomy shows 'drop anchors' in places where the element can be dropped.



Whether an element can be dragged and where it can be dropped is something you control by setting the `canDropTo` property of an element's specification. The property is an array of element names, referring to elements that can take the current element as a (direct) child. If the array is empty or non-existent, the element cannot be dragged (and does not have a drag handle).

```
var docSpec={
  elements: {
    "teachers": {
      ...
    },
    "students": {
      ...
    },
    "person": {
      canDropTo: ["teachers", "students"],
      ...
    },
    ...
  }
}
```

Allowing an element to be dragged and dropped is how you can let the user reorder child elements within a parent element. Just add the parent element's name to the child element specification's `canDropTo` array.

Allowing an element to be dragged and dropped is also how you let the user move an element from one parent to another. In the example above, the elements `<teachers>` and `<students>` can both have `<person>` children, and the user is allowed to move a `<person>` from `<teachers>` to `<students>` and back as he or she pleases.

When dragging and dropping, Xonomy respects the dragged element's `mustBeBefore` and `mustBeAfter` constraints: it does not allow the user to drop the element (= does not display a drop anchor) where its position would violate those constraints. For more on `mustBeBefore` and `mustBeAfter` constraints see chapter 5.2.

## 7. Controlling the display of elements and their children

### 7.1. Collapsing elements

You will no doubt have noticed that elements that have children come with little plus/minus symbols which you can click to collapse and expand the element's content. You can control whether an element will be collapsed by default when it is first rendered. You do this by setting the `collapsed` property in the element's specification. It is a function which returns either `true` (= the element should be collapsed initially) or `false`. The default value for this property is a function that returns `false`.

```
var docSpec={
  elements: {
    "item": {
      collapsed: function(jsElement){return true;}
    },
    ...
  }
}
```

The reason why the `collapsed` property is a function rather than a straight `true/false` value is that you may want to decide its value dynamically. For example, if your XML document contains sections in different languages, you may want all English sections to be expanded by default and the others collapsed for an English-speaking user, and so on. To achieve this, make sure the `collapsed` function of the `<section>` element's specification returns `false` if the element has `lang="en"` and `true` in all other cases.

```
<section lang="en">
  ...
</section>
<section lang="de">
  ...
</section>
<section lang="fr">
  ...
</section>
```

The `collapsed` function takes an argument which is a **surrogate element object** (see chapter 10).

### 7.2. Oneliner display

By default, if an element has children (child elements or text nodes or both), Xonomy displays the children as a block-level list of items between the opening and closing tag. If you prefer, you can tell Xonomy to display the element in a more compact layout in which the entire element (the opening tag, the children and the closing tag) are displayed as a single line of items. You do this by setting the `oneliner` property of the element's specification.

```
var docSpec={
  elements: {
    "title": {
      oneliner: true
    },
    ...
  }
}
```

```
└─<article id="gcc" language="en">
  └─<title>Gleann Cholm Cille</title>
  └─<body>Gleann Cholm Cille (angli...</body>
</article>
```

Notice that the `oneliner` property is inherited by child elements. If, in the example above, the `<title>` element had any child elements, the child elements would also be treated as if they had `oneliner` set to `true`, regardless of what their element specifications said.

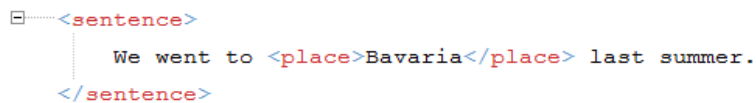
## 8. Working with mixed content

The term ‘mixed content’ refers to a situation when a parent element contains a mixture of child elements and text nodes. When a situation like that occurs, by default Xonomy displays each text node and each child element as a separate (block-level) item. This probably isn’t very convenient for editing. Luckily, you can override this behaviour by setting a few properties in the parent element’s specification to make editing mixed content more user-friendly. The rest of this chapter explains how.

### 8.1. Allowing mixed content

To tell Xonomy that the contents of a particular element is meant to be not a list of block-level elements but **text** (potentially decorated with inline markup), set the element specification’s `hasText` property to `true`. This changes a few things about how Xonomy treats the children of that element. One important change is that it displays the children not as a block-level list but as a single inline sequence.

```
var docSpec={
  elements: {
    "sentence": {
      hasText: true
    },
    "place": {
      hasText: true
    }
  }
};
```

A screenshot of the Xonomy editor interface. It shows a document with a single block-level element, `<sentence>`. Inside this element, there is a text node "We went to " followed by a block-level element `<place>` containing the text "Bavaria", followed by another text node " last summer." and the closing tag `</sentence>`. The `<place>` element is displayed as a separate block-level item within the `<sentence>` container.

Notice that you need to set `hasText` to `true` for all inline markup elements as well. If you have a `<sentence>` element which can contain a mixture of text nodes and `<place>` elements, and if `<place>` elements can also contain text, then you need to set `hasText` to `true` in the element specifications of both `<sentence>` and `<place>`. If you only set it to `true` on `<sentence>` but not on `<place>`, then the contents of `<sentence>` will be displayed inline but the contents of `<place>` will still be a block-level list. This is not a bug: it may actually be desired behaviour if the `<place>` element contains complex markup which you don’t want to display inline.

```
var docSpec={
  elements: {
    "sentence": {
      hasText: true
    },
    "place": {
      hasText: false
    },
    "name": {
      oneliner: true
    }
  }
};
```

```

<sentence>
  We went to <place>
    <name lang="en">Bavaria</name>
    <name lang="de">Bayern</name>
  </place> last summer.
</sentence>

```

### “What is the difference between `hasText` and `oneLiner`?”

We met the `oneLiner` property when we were looking at controlling the display of elements in the previous chapter. The difference is as follows:

- The `oneLiner` property affects the entire element. When it is `true` for an element, the entire element is displayed as an inline stretch of text, including its opening tag, its children collection, and its closing tag.
- The `hasText` property affects only the children of the element. When it is `true` for an element, only its children collection is displayed as an inline stretch of text, not its opening tag or closing tag.

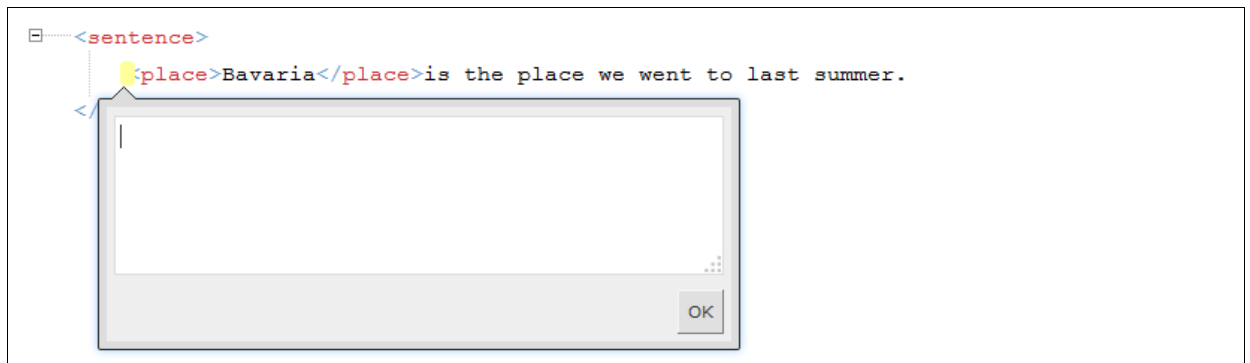
Moreover, `oneLiner` is inherited by child elements while `hasText` isn't.

You can of course combine the two (set both `oneLiner` and `hasText` to `true`), which will result in the most compact (= least space-consuming) display for your mixed-content elements.

## 8.2. Entering text at tag boundaries

Another thing that changes when you set an element specification's `hasText` property to `true` is that Xonomy will now allow your user to create new text nodes at tag boundaries. Here is what it means in practice:

- If the first child is an element (as opposed to a text node), Xonomy will allow your user to create a text node before it by hovering the mouse there and clicking.



- If the last child is an element (as opposed to a text node), Xonomy will allow your user to create a text node after it by hovering the mouse there and clicking.
- If two child elements meet without an intervening text node, Xonomy will allow your user to create a text node between them by hovering the mouse there and clicking.
- If the element has no content at all (no text node and no child elements), Xonomy will allow your user to create a text node inside it by hovering the mouse between the opening and closing tag and clicking.

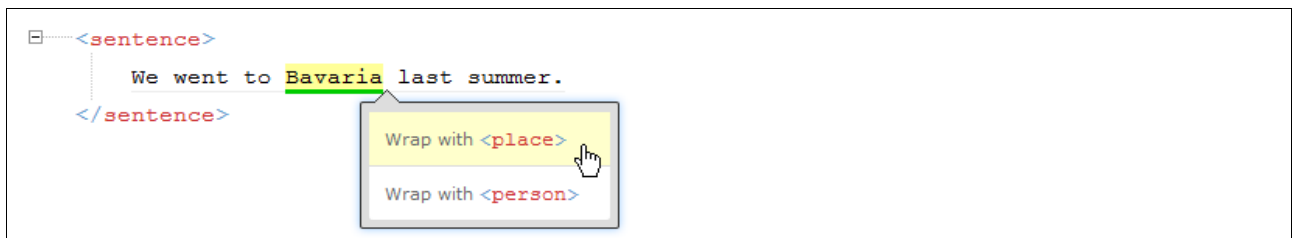
### 8.3. Inserting inline markup

Xonomy makes it possible for your users to select a stretch of text inside a text node and surround it with XML markup. Let's first have a look how it works from a user's point of view. Then we'll explain how to make this feature available through a document specification.

When a text node can be annotated with inline markup, you will notice a thin grey line underlining the text. To select a stretch of text, click on this line underneath the first character and then again underneath the last character. (If you want to select just one character, click on the same character twice.) You will see that the stretch has been highlighted and a menu has appeared. You can now select something from the menu and Xonomy will surround the selected text with XML markup.

The menu that appears beside a selected stretch of text comes from the `inlineMenu` property of the element's specification. Like all other menu properties, it is an array of menu specifications (see chapter 4).

```
var docSpec={
  elements: {
    "sentence": {
      hasText: true,
      inlineMenu: [{
        caption: "Wrap with <place>",
        action: Xonomy.wrap,
        actionParameter: {template: "<place>$/</place>", placeholder: "$"}
      }, {
        caption: "Wrap with <person>",
        action: Xonomy.wrap,
        actionParameter: {template: "<person>$/</person>", placeholder: "$"}
      }
    ],
    ...
  }
}
```



Xonomy provides a prefabricated menu action function which you will probably be using very often in inline menus: `Xonomy.wrap`. This function works by splitting the text node in three and replacing the middle one with the XML markup specified in the menu item's `actionParameter`. Notice that the XML markup contains a placeholder: the contents of the middle text node will be inserted in its place.

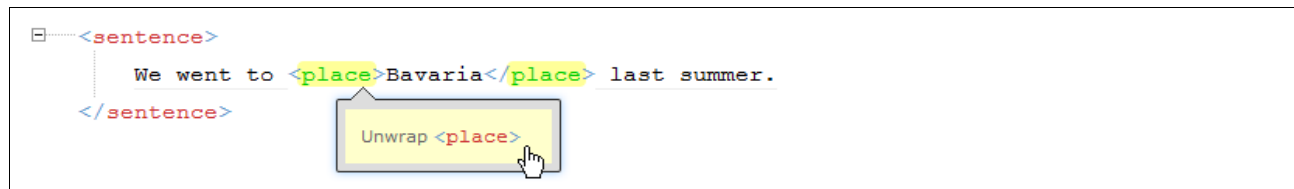
### 8.4. Removing inline markup

To make it possible for your users to remove an element from around a stretch of content without also deleting the content, add an item to the element's menu and use the function `Xonomy.unwrap`. Unlike `Xonomy.deleteElement`, this function doesn't delete the whole element but replaces it with its content.

```
var docSpec={
  elements: {
    ...
    "place": {
      hasText: true,
      menu: [{
```



```
caption: "Unwrap <place>",  
action: Xonomy.unwrap  
  }]  
},  
...  
}
```



## 9. Validating the document

For the (hopefully rare) occasions when Xonomy's document specification is not expressive enough to impose structure on the document your user is editing, Xonomy lets you write your own validation functions in JavaScript. A Xonomy validation function is a function that checks the document and issues human-readable warning messages to the user.

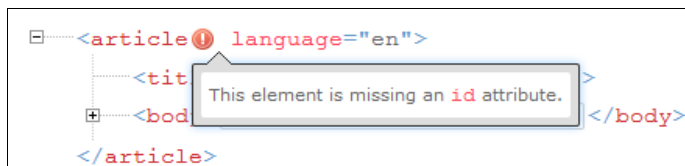
You must assign your validation function to the document specification's `validate` property. The function is called by Xonomy every time the document changes, and also immediately after the document has been rendered (after you have called `Xonomy.render`).

```
var docSpec={
  validate: function(jsElement){
    console.log("Let's validate this document!");
  },
  ...
}
```

The validation function takes a single argument, a **surrogate element object** representing the document's top element. From there you can access all other elements, attributes and text nodes present in the document, test them in any way you want, and issue warning messages.

To issue a warning message, push an object to the array `Xonomy.warnings`. The object you push there must have two properties, `htmlID` and `text`. The `htmlID` property is the ID of the element or attribute to which the warning messages is to be attached; you can get this ID from the surrogate object. The `text` property is a human-readable warning message (any element or attribute name mentioned in the message will be pretty-formatted as explained in chapter 4.1)

```
var docSpec={
  validate: function(jsElement){
    if(...) { //if something is true
      Xonomy.warnings.push({
        htmlID: jsElement.htmlID,
        text: "This element is missing an @id attribute."
      });
    }
  },
  ...
}
```



When a warning message is pushed to the `Xonomy.warnings` array, Xonomy displays an exclamation mark icon in the element or attribute and, upon clicking on it, shows the warning message in a pop-up box. The user is now expected to correct the error. As soon as he or she has made a change to the document, the `validate` function is called again and the error message will either disappear or appear again, depending on whether the error has been corrected or not. (The `Xonomy.warnings` array is emptied automatically before each `validate` call.)

***"I'm confused. The `validate` function doesn't seem to return any value. So how does it tell me whether the document is valid or not?"***

*It doesn't tell you that. The purpose of the `validate` function is to issue human-readable warnings to the user, and that is all. Its purpose is not to produce a single `true/false`*

*judgment. Neither is its purpose to stop the user from saving or submitting the document if it is invalid.*

*If you require this kind of functionality, you should implement it externally from Xonomy, such as by doing server-side validation.*

Performing validation requires good knowledge of surrogate objects, which you can learn more about in the next chapter (10).

## 9.1. Validating with recursion

When designing a `validate` function, a popular choice is to cycle through the document recursively, performing checks on each element individually.

```
var docSpec={
  validate: function(jsElement){
    //Cycle through the element's attributes:
    for(var i=0; i<jsElement.attributes.length; i++) {
      var jsAttribute=jsElement.attributes[i];
      //Make sure item/@label is not an empty string:
      if(jsElement.name=="item" && jsAttribute.name=="label") {
        if(jsAttribute.value=="") {
          Xonomy.warnings.push({
            htmlID: jsAttribute.htmlID,
            text: "This attribute must not be empty."}
        );
      }
    }
  };
  //Cycle through the element's children:
  for(var i=0; i<jsElement.children.length; i++) {
    var jsChild=jsElement.children[i];
    if(jsChild.type=="element") { //if element
      docSpec.validate(jsChild); //recursion
    }
  }
},
...
}
```

## 9.2. Validating with delegation

Another approach to validation is to give each element and attribute specification its own validation function, and to call these functions from the document specification's `validate` function. In other words, instead of doing all validation in a single function, you 'delegate' the validation of each element and attribute to its specification.

```
var docSpec={
  validate: function(jsElement){
    //Validate the element:
    var elementSpec=docSpec.elements[jsElement.name];
    if(elementSpec.validate) elementSpec.validate(jsElement);
    //Cycle through the element's attributes:
    for(var i=0; i<jsElement.attributes.length; i++) {
      var jsAttribute=jsElement.attributes[i];
      var attributeSpec=elementSpec.attributes[jsAttribute.name];
      if(attributeSpec.validate) attributeSpec.validate(jsAttribute);
    }
  }
}
```

```

    };
    //Cycle through the element's children:
    for(var i=0; i<jsElement.children.length; i++) {
        var jsChild=jsElement.children[i];
        if(jsChild.type=="element") { //if element
            docSpec.validate(jsChild); //recursion
        }
    }
},
elements: {
    "list": {
        ...
    },
    "item": {
        attributes: {
            "label": {
                validate: function(jsAttribute){
                    //Make sure item/@label is not an empty string:
                    if(jsAttribute.value=="") {
                        Xonomy.warnings.push({
                            htmlID: jsAttribute.htmlID,
                            text: "This attribute must not be empty."}
                    );
                }
            },
            ...
        }
    },
    ...
}
}
}
}
}

```

## 10. Working with surrogate objects

**Surrogate objects** are JavaScript objects that Xonomy passes as arguments to certain functions. You have seen them mentioned several times in this document, last but not least in the chapter on validation (chapter 9). A surrogate object encapsulates the state of an element, attribute or text node at a moment in the lifetime of a document being edited in Xonomy. Surrogate objects are like objects in the XML document object model (DOM) but simpler and more geared towards the requirements of Xonomy. For example, the element `<item label='one'/>` will be represented by a surrogate object that looks like this:

```
{
  type: "element",
  name: "item",
  htmlID: "xonomy2",
  parent: ...,
  attributes: [{
    type: "attribute",
    name: "label",
    value: "one",
    htmlID: "xonomy3",
    parent: ...
  }],
  children: []
}
```

### 10.1. The properties of a surrogate object

#### type

Who has it: elements, attributes, text nodes.

The string `"element"`, `"attribute"` or `"text"` to tell you whether the surrogate object represents an element, an attribute or a text node.

#### name

Who has it: elements, attributes.

The name of the element or attribute (a string).

#### value

Who has it: attributes, text nodes.

The value of the attribute, or the text of the text node (a string).

#### htmlID

Who has it: elements, attributes, text nodes.

The unique ID of the node (a string).

#### parent

Who has it: elements, attributes, text nodes.

A reference to the surrogate object that represents the current object's parent. For elements and text nodes, this is the parent element. For attributes, this is the element that contains the attribute. For the document's top-level element, this is `null`.

### attributes

Who has it: elements.

An array of surrogate objects representing the attributes the element has. Can be an empty array.

### children

Who has it: elements.

An array of surrogate objects representing the children the element has. A child is either an element or a text node. Can be an empty array.

### hasAttribute(name)

Who has it: elements.

A function that returns `true` if the element has an attribute of that name.

### getAttribute(name, ifNull)

Who has it: elements.

A function that returns the value of the element's attribute (as a string) . If no such attribute exists in the element, returns whatever `ifNull` is.

### hasChildElement(name)

Who has it: elements.

A function that returns `true` if the element has at least one child element of that name.

## 10.2. Quick reference to surrogate objects

### Element

```
{
  type: "element",
  name: "",
  htmlID: "",
  attributes: [{...}, {...}, ...],
  children: [{...}, {...}, ...],
  parent: {...},
  hasAttribute(name),
  getAttributeValue(name, ifNull),
  hasChildElement(name)
}
```

### Attribute

```
{
  type: "attribute",
  name: "",
  value: "",
  htmlID: "",
  parent: {...}
}
```

## Text node

```
{  
  type: "text",  
  value: "",  
  htmlID: "",  
  parent: {...}  
}
```

## 11. Working with namespaces

Xonomy is able to work with XML documents that refer to namespaces. If an element or attribute name contains a namespace prefix, such as `<mbm:list>` the whole 'local' name is treated as an element name. In this example, the name of the element is `"mbm:list"` as far as Xonomy is concerned. Xonomy makes no effort to understand or analyze what the prefix `"mbm:"` means.

### 11.1. Namespaces in document specifications

This means that you must use the whole 'local' name in your document specification.

```
var docSpec={
  elements: {
    "mbm:list": {
      //...
    }
  }
}
```

### 11.2. Rendering with namespaces

Even though Xonomy pays no attention to what namespace prefixes mean, the XML parser that Xonomy uses internally does, and so you still need to declare namespaces properly in the XML-as-string you pass to the `Xonomy.render` function. If you fail to declare what a namespace prefix means, the XML will fail to parse:

```
var xml="<mbm:list xmlns:mbm='http://lexiconista.com'><mbm:item label='one'/></mbm:list>";
var editor=document.getElementById("editor");
Xonomy.render(xml, editor, docSpec);
```

The same holds for the XML-as-string you pass to menu action functions such as `Xonomy.newElementChild`. Xonomy uses an XML parser internally to parse the XML as if it were a brand new document. Therefore, you need to declare all namespaces, even if they have been declared in your document before. This is what your menu items should look like:

```
{
  caption: "New <mbm:item>",
  action: Xonomy.newElementChild,
  actionParameter: "<mbm:item xmlns:mbm='http://lexiconista.com'/>"
}
```

Xonomy doesn't show namespace declarations in the rendered document. The only reason you need to include them in your XML-as-string is to make sure the internal parser doesn't fail.

### 11.3. Harvesting with namespaces

Xonomy remembers all the namespace declarations you've used in your XML-as-strings throughout the lifetime of the document, and adds them back to the document's top-level element when you call `Xonomy.harvest`.

```
var xml=Xonomy.harvest();
//xml=="<mbm:list xmlns:mbm='http://lexiconista.com'><mbm:item label='one'/></mbm:list>";
```



## 12. Document specification reference

### 12.1. Document specification

```
var docSpec={
  elements: {
    "myElement": {...},
    "anotherElement": {...}
  },
  onchange: function(){...},
  validate: function(jsTopElement){...}
}
```

#### **elements: {...}**

An associative array where the keys are the names of elements the document can contain and each value is an **element specification**. For details of what goes in an element specification see chapter 12.2.

Default: an empty object.

#### **onchange: function() {...}**

Xonomy calls this function each time the user has done something to the document, telling you that the document is now 'dirty'.

Default: a function that does nothing.

#### **validate: function(jsTopElement) {...}**

Xonomy calls this function each time it thinks you might want to validate the document.

Default: a function that does nothing.

### 12.2. Element specification

```
"myElement": {
  attributes: {
    "myAttribute": {...},
    "myOtherAttribute": {...}
  },
  menu: [{...}, {...}, ...],
  canDropTo: ["myParentElement", "someOtherElement", ...],
  mustBeAfter: ["aSibling", "anotherSibling", ...],
  mustBeBefore: ["aSibling", "anotherSibling", ...],
  oneliner: false,
  hasText: true,
  inlineMenu: [{...}, {...}, ...],
  collapsed: function(jsElement){...}
}
```

#### **attributes: {...}**

An associative array where the keys are the names of attributes the element can contain and each value is an **attribute specification**. For details of what goes in an attribute specification see chapter 12.3.

Default: an empty object.

**menu:** [{...}, {...}, ...]

An array of **menu items** for the menu that appears when the user clicks this element. For details of what goes in a menu item see chapter 12.4.

Default: an empty array.

**canDropTo:** ["myParentElement", "someOtherElement", ...]

An array of the names of elements that this element can be dragged-and-dropped to as (direct) child.

Default: an empty array.

**mustBeAfter:** ["aSibling", "anotherSibling", ...]

An array of the names of elements who can never be following siblings of this element.

Default: an empty array.

**mustBeBefore:** ["aSibling", "anotherSibling", ...]

An array of the names of elements who can never be preceding siblings of this element.

Default: an empty array.

**onliner:** true|false

Should this element be displayed inline, with the opening tag, the children and the closing tag in a single (possibly wrapped) line of text?

Default: false.

**hasText:** true|false

Is this element allowed to have text nodes?

Default: false.

**inlineMenu:** [{...}, {...}, ...]

An array of **menu items** for the menu when the user selects a stretch of text in a text node which is a direct child of this element. For details of what goes in a menu item see chapter 12.4.

Default: an empty array.

**collapsed:** function(jsElement){...}

Should this element be collapsed when the document is first rendered?

Default: false.

## 12.3. Attribute specification

```
"myAttribute": {  
  asker: function(currentValue, askerParameter),  
  askerParameter: {...},  
  menu: [{...}, {...}, ...]  
}
```

**asker: function(currentValue, askerParameter)**

An asker function which renders the contents of the pop-up box where the user edits the value. You can use one of Xonomy's prefabricated asker functions (see chapter 4.2 for a list of what's available). A popular choice is `Xonomy.askString` which permits editing the value in a simple one-line textbox.

Default: a function that returns nothing, meaning the attribute cannot be edited (is read-only).

You can also assign your own asker function to this. If you are writing your own function, design it in such a way that it returns HTML-as-string and accepts two arguments: `currentValue` is the current value of the attribute (which must be a string and can be empty) and `askerParameter` is an optional arbitrary object which you can specify in the next property described below. For more details on editing attribute values, see chapter 3.

**askerParameter: {...}**

An arbitrary object to be passed to the attribute's asker function.

Default: an empty object.

**menu: [{...}, {...}, ...]**

An array of **menu items** for the menu that appears when the user clicks this element. For details of what goes in a menu item see chapter 12.4.

Default: an empty array.

## 12.4. Menu item specification

```
{
  caption: "",
  action: function(htmlID, actionParameter),
  actionParameter: {...},
  hideIf: function(jsElement|jsAttribute)
}
```

**caption: ""**

A human-readable caption to appear on the menu, for example `"Delete this attribute"`.

Default: the string `"?"`.

**action: function(htmlID, actionParameter)**

A function representing the action that's supposed to happen when the user clicks this menu item. It can be one of Xonomy's prefabricated menu action functions (see chapter Error: Reference source not found for a list of what's available) or you can write your own.

Default: a function that does nothing.

If you are writing your own, design the function in such a way that it takes two arguments: `htmlID` is the ID of the element or attribute the action is being applied to, and `actionParameter` is an optional arbitrary object which you can specify in the next property described below.

**actionParameter: {...}**

An arbitrary object to be passed to the menu action function.

Default: an empty object.

**hideIf: function(jsElement|jsAttribute)**

A function that returns **true** if the menu item is to be hidden, **false** if not. The argument passed to the function is either a surrogate element object or a surrogate attribute object, depending on what is being edited.

Default: a function that returns **false**.