Widget Interface

W3C Recommendation 31 October 2013
obsoleted 11 October 2018

This Version:
https://www.w3.org/TR/2018/OBSL-widgets-apis-20181011/
Latest Version:
http://www.w3.org/TR/widgets-apis/
Previous Version:
http://www.w3.org/TR/2013/REC-widgets-apis-20131031/
Latest Editor's Draft:
http://w3c.github.io/packaged-webapps/api/Overview.html
Test suite:
http://dev.w3.org/2006/waf/widgets-api/test-suite/
Implementation report:
http://dev.w3.org/2006/waf/widgets-api/imp-report/
Editor:
Marcos Cáceres

Please refer to the **errata** for this document, which may include some normative corrections.

See also **translations**.

Abstract

This specification defines an application programming interface (API) for widgets that provides, amongst other things, functionality for accessing a widget's metadata and persistently storing data.

Status of this Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at https://www.w3.org/TR/.

This specification is obsolete and should no longer be used as a basis for implementation.

The Widget specifications became W3C Recommendations in 2012-2013. They were designed to enable interactive single purpose application for displaying and/or updating local data or data on the Web, packaged in a way to allow a single download and installation on a user's machine or mobile device.

Since 2013, Widgets has had limited deployment and its usage has been reduced since then. Service Workers and Web App Manifest are considered to provide better solutions nowadays.

For purposes of the W3C Patent Policy this Obsolete Recommendation has the same status as an active Recommendation; it retains licensing commitments and remains available as a reference for old implementations but is no longer recommended for future implementation.

Table of Contents

1 Introduction

This section is non-normative.

This specification defines an application programming interface that enables the ability to:

- Access some of the metadata declared in a widget's configuration document.
- Persistently store data relating to a widget instance.
- Retrieve the name and value of preferences, which may have been declared in a widget's configuration document or at runtime.

1.1 The Widget Family of Specifications

This section is non-normative.

This specification is part of the **Widgets family of specifications**, which together standardize widgets as a whole. The list of specifications that make up the Widgets family of specifications can be found on the Web Applications Working Group's wiki.

1.2 Design Goals and Requirements

This section is non-normative.

The design goals for this specification are documented in the [Widget Requirements] document. This document addresses some of the requirements relating to Application Programming Interfaces of the [Widget Requirements] document:

- Instantiated Widget API: addressed by `widget` object.
- IDL Definitions: to meet this requirement, this specification makes use of [WebIDL].
- Manipulation of Author-defined start-up values: addressed by using a widget storage area and `preferences` attribute's extension of the `Storage` interface defined in [Web Storage].
- Configuration Document Data: this is addressed by the `widget` object's attributes.

2 Conformance

All examples and notes in this specification are non-normative, as are all sections explicitly marked as non-normative. Everything else in this specification is normative.

The key words must, must not, should, recommended, may and optional in the normative parts of this specification are to be interpreted as described in [RFC2119].

The IDL blocks in this specification are conforming IDL fragments as defined by the WebIDL specification.

Only user agents can claim conformance to this specification. Conformance requirements phrased as algorithms or specific steps can be implemented in any manner, so long as the end result is equivalent to what would be achieved when following the specification.

> ***Note: Implementations can partially check their level of conformance to this specification by successfully passing the test cases of the [Interface-Test-Suite]. Passing all the tests in the test suite does not imply conformance to this specification; It only implies that the implementation conforms to aspects tested by the test suite.***

3 Definitions

The following definitions are used throughout this specification. Please note that the following list is not exhaustive; other terms are defined throughout this specification.

Author script

Some code running within a widget instance (e.g., some ECMAScript).

Configuration document

A configuration document is reserved file called "`config.xml`" at the root of the widget package as specified in the [Widgets-Packaging] specification.

Getting

A DOM attribute is said to be getting when its value is being retrieved (e.g. by an author script).

Initialization

The act of user agent processing a widget package through the Steps for Processing a Widget Package, as specified in the [Widgets-Packaging] specification.

Preference

A persistently stored name-value pair that is associated with the widget the first time the widget is initiated.

Start file

A file in the widget package to be loaded by the user agent when it instantiates the widget, as specified in the [Widgets-Packaging] specification.

Setting

A DOM attribute is said to be setting when its value is being set to some value (e.g. by an author script).

Supports

A user agent implements a mentioned specification or conformance clause.

Viewport

A *CSS viewport*. For a start file rendered on *continuous media*, as defined in the [CSS] specification, a viewport is the area on which the `Document` of the start file is rendered by the user agent. The dimensions of a viewport excludes scrollbars, toolbars, and other user interface "chrome".

Widget Instance

A browsing context that comes into existence after initialization. The concept of a **browsing context** is defined in [HTML]. Multiple widget instances can be instantiated from a single widget package. A widget instance is unique and does not share any DOM attribute values, widget storage area, or [Web Storage] storage areas with any other widget instance.

4 User Agent

A **user agent** is a software implementation of this specification that also supports the [Widgets-Packaging] specification.

> ***Note: The user agent described in this specification does not denote a "widget user agent" at large. That is, a user agent that implements all the specifications, and dependencies, defined in the***

***widgets family of specifications**. The user agent described in this specification is only concerned with the behavior of programming interfaces. A user agent needs to impose implementation-specific limits on otherwise unconstrained inputs, e.g. to prevent denial of service attacks, to guard against running out of memory, or to work around platform-specific limitations.*

5 The `WindowWidget` Interface

For a widget instance, a user agent must expose a unique object that implements the `Widget` interface to author scripts that are same origin as the instance of the widget (e.g., a `Document` loaded in a [HTML] `iframe` element with content from within a widget package). User agent implementing [HTML]'s `Window` interface must implement the `Widget` interface as the `widget` attribute of the `window` object in the manner defined by the `WindowWidget` interface.

```
[NoInterfaceObject]
interface WindowWidget {
  readonly attribute Widget widget;
};

Window implements WindowWidget;
```

6 The `Widget` Interface

An object that implements the `Widget` interface exposes the following attributes:

```
interface Widget {
    readonly attribute DOMString     author;
    readonly attribute DOMString     description;
    readonly attribute DOMString     name;
    readonly attribute DOMString     shortName;
    readonly attribute DOMString     version;
    readonly attribute DOMString     id;
    readonly attribute DOMString     authorEmail;
    readonly attribute DOMString     authorHref;
    readonly attribute WidgetStorage preferences;
    readonly attribute unsigned long height;
    readonly attribute unsigned long width;
};
```

Note: A user agent can support the `Storage` interface on DOM attributes other than the `preferences` attribute (e.g., a user agent can support the [Web Storage] specification's `localStorage` attribute of the `window` object in conjunction to the `preferences` attribute). For the sake of interoperability across widget user agents, and where it makes sense, authors can use the `preferences` attribute in conjunction to other APIs that expose an object that implements the `Storage` interface.

If a user agent has previously associated a widget storage area with a widget instance, the user agent must not re-create the `preferences` attribute unless explicitly requested to do so by the end-user or for security or privacy reasons (e.g., the end-user wants to purge personal data). Instead, the previously associated widget storage area (or an equivalent clone) can be accessed using the `Storage` interface. When an object implementing the `Widget` interface is instantiated, if a user agent has not previously associated a widget storage area with the instance of a widget, then the user agent must create the `preferences` attribute.

6.1 Usage Example

This section is non-normative.

This example shows how a widget's metadata can be accessed by through the widget interface.

Given the following configuration document:

```
<widget xmlns      = "http://www.w3.org/ns/widgets"
        id         = "http://example.org/exampleWidget"
        version    = "2.0 Beta"
        height     = "200"
        width      = "200"
        viewmodes = "floating">

<name short="Example 2.0">The example Widget!</name>

<description>A sample widget to demonstrate some of the possibilities.</description>

<author href  = "http://foo-bar.example.org/"
        email = "foo-bar@example.org">Foo Bar Corp</author>

<preference name     = "apikey"
```

```
            value    = "ea31ad3a23fd2f"
            readonly = "true"/>

</widget>
```

And given the following start file:

```
<!doctype html>
<title>About this Widget</title>
<style>
html {
  padding: 20px;
}

#aboutBox{
  padding: 20px;
  box-shadow: 2px 2px 10px #444;
  border-radius: 15px;
  background-color: #ECEDCF;
  text-align:center;
}
</style>

<body onload="makeAboutBox()">
<div id="aboutBox">
<h1><a id="storeLink"><img src="icon.png" id="icon"></a></h1>
<h1 id="name">Name</h1>
<p id="version">Version: </p>
<hr>
<p id="description">...</p>
<hr>
<p id="author">© </p>
</div>
<script>
  // example that generates an about box
  // using metadata from a widget's configuration document.
  function makeAboutBox(){
  var storeLink = document.getElementById("storeLink");
  storeLink     = storeLink.setAttribute("href", widget.id);

  var icon     = document.getElementById("icon");
  icon.setAttribute("alt", widget.shortName);
  var title    = document.getElementById("name");
  title.innerHTML = widget.name;

  var version       = document.getElementById("version");
  var prodKey       = widget.preferences["productKey"];
  version.innerHTML += widget.version +
" (" + prodKey + ")";
  var description   = document.getElementById("description");
  description.innerHTML = widget.description;

  var author = document.getElementById("author");
  author.innerHTML += widget.author;
  }
</script>
```

The widget would render as:

6.2 Metadata Attribute Values

Most of the attributes of the `widget` interface correspond to the metadata derived from the initialization process.

When an object implementing the `Widget` interface is instantiated, a user agent sets the attributes identified in the left column of the configuration attributes table to the values that correspond to values in *table of configuration defaults* as defined in [Widgets-Packaging] (identified by the values in the right hand column).

Configuration Attributes Table

Attributes	Values in Table of Configuration Defaults	Is localizable string
`author`	*author name*	yes
`version`	*widget version*	yes
`shortName`	*widget short name*	yes
`name`	*widget name*	yes
`description`	*widget description*	yes
`authorEmail`	*author email*	no
`authorHref`	*author href*	no
`id`	*widget id*	no

Upon getting any of the attributes from the attributes column of the configuration attributes table, a user agent must return the corresponding value from the 'Values in Table of Configuration Defaults' column.

Attributes that contain a localizable string are identified by having word yes in the "Is localizable string" column in the Configuration Attributes Table above.

6.2.1 Localizable strings

Some attributes in the Configuration Attributes Table come in the form of a **localizable string**, which is defined by the [Widgets-Packaging] specification as a...

> "data structure containing a sequence of one or more strings, each having some associated directional information and language information (if any). The purpose of a localizable string is to assist user agent in correctly applying the Unicode [BIDI] algorithm when displaying text."

When getting an attribute that is identified as a localizable string, the user agent must apply the rule for getting localizable strings and return the result.

6.2.2 Examples

This example shows how a user agent is expected to behave when an empty configuration document is given:

```
<widget xmlns = "http://www.w3.org/ns/widgets"/>
```

Would result in the following being reflected in the through the `widget` object:

```
<!doctype html>
<script>
  alert(widget.version     === "") //true
  alert(widget.name        === "") //true
  alert(widget.author      === "") //true
  alert(widget.authorEmail === "") //true
  alert(widget.authorHref  === "") //true
  alert(widget.description === "") //true
  alert(widget.id          === "") //true
  alert(widget.shortName   === "") //true
</script>
```

6.3 The `width` Attribute

Upon getting the `width` attribute, a user agent must return a number that represents the width of the widget instance's viewport in [CSS] pixels.

6.4 The `height` Attribute

Upon getting the `height` attribute, a user agent must return a number that represents the height of the widget instance's viewport in [CSS] pixels.

6.5 The `preferences` Attribute

The `preferences` attribute allows authors to manipulate a widget storage area that is unique for the instance of a widget. It does this by implementing the `Storage` interface specified in [Web Storage].

Upon invocation of the `setItem()`, `removeItem()` and `clear()` methods, if the invocation did something, a user agent must dispatch a storage event akin to what is specified in "the storage event" section of [Web Storage] (i.e., the `preferences` attribute behaves the same as `localStorage` with regards to dispatching events).

Upon invocation of the `setItem()` or `removeItem()` method by an author script on a read-only item, a user agent must throw a `NO_MODIFICATION_ALLOWED_ERR` exception and must not fire a storage event. The `NO_MODIFICATION_ALLOWED_ERR` is defined in the [DOM3Core] specification.

Upon invocation of the `preferences` attribute's `clear()` method, a user agent must not remove read-only items and corresponding values from a widget storage area. A user agent must, however, remove other items from the widget storage area in the manner described in the [Web Storage] specification without throwing a `NO_MODIFICATION_ALLOWED_ERR` exception for items that the user agent cannot remove.

When getting or setting the `preferences` attribute, if the origin of a widget instance is mutable (e.g., if the user agent allows `document.domain` to be dynamically changed), then the user agent must perform the preference-origin security check. The concept of **origin** is defined in [HTML].

6.5.1 Preference Origin Security Check

The steps to perform the **preference-origin security check** are given by the following algorithm:

1. The user agent may throw a `SECURITY_ERR` exception instead of returning a `Storage` object if the request violates a policy decision (e.g. if the user agent is configured to not allow the page to persist data).
2. If the `Document`'s origin is not a scheme/host/port tuple, then throw a `SECURITY_ERR` exception and abort these steps.
3. Otherwise, return the `Storage` object associated with that widget instance's `preferences` attribute.

6.5.2 Creating the `preferences` Attribute

The steps to **create the `preferences` attribute** are given by the following algorithm:

1. Create a widget storage area that is unique for the origin of this instance of a widget.
2. If the *widget preferences* variable of the table of configuration defaults contains any *preferences*, then for each *preference* held by *widget preferences*:
    a. Let *pref-key* be the *name* of the *preference*.
    b. If the *pref-key* already exists in the *storage area*, stop processing this *preference:* go back to step 2 in this algorithm, and process the next *preference* (if any).
    c. Let *pref-value* be the *value* of the *preference*.
    d. Add *pref-key* and *pref-value* to the widget storage area:
        1. If the user agent cannot write to the widget storage area (e.g., because it ran out of disk space, or the space quota was exceeded, etc.), terminate all processing of this widget. It is recommended that the user agent inform the end-user of the error.
        2. If this *preference*'s associated *readonly* value is `true`, then flag this key as a read-only item in the widget storage area.
3. Implement the Storage interface on the widget storage area, and make the `preferences` attribute a pointer to that storage area.

6.5.3 Usage Example 1

This section is non-normative.

The following example shows the different means by which an author can interface with the `widget` object's `preferences` attribute in ECMAScript. The possibilities include using either the `getItem()` and `setItem()` methods, or bracket access (or a combination of both).

```
<!doctype html>
...
```

```
<fieldset id="prefs-form">
<legend>Game Options</legend>
  <label>Volume:  <input type="range" min="0" max="100" name="volume"/> </label>
  <label>Difficulty:  <input type="range" min="0" max="100" name="difficulty"/> </label>
  <input type="button" value="Save" onclick="savePrefs()"/>
  <input type="button" value="load" onclick="loadPrefs()"/>
</fieldset>

...

<script>
var form   = document.getElementById("prefs-form");
var fields = form.querySelectorAll("input[type='range']");
function loadPrefs () {
   for(var i = 0; i < fields.length; i++){
      var field = fields[i];
      if (typeof widget.preferences[field.name] !== "undefined") {
        field.value = widget.preferences.getItem(field.name);
      }
   }
}

function savePrefs () {
  for(var i = 0; i < fields.length; i++){
      var field = fields[i];
      widget.preferences.setItem(field.name,field.value);
  }
}
</script>
```

6.5.4 Usage Example 2

This section is non-normative.

This example demonstrates the expected behavior of a user agent that is interacting with preferences that were declared in a configuration document. The user is able to modify and save various preferences. However, if the user attempts to modify and save the license key, which is set to read-only, the widget will throw an error and display an alert message.

```
<!-- Configuration Document -->

<widget xmlns="http://www.w3.org/ns/widgets">
  <name>The 80's: Greatest Hits!</name>
  <preference name="licenseKey"
              value="f199bb20-1499-11df"
              readonly="true"/>
  <preference name="favtrack"
              value="billy"/>
  <preference name="playorder"
              value="1" />
</widget>
```

```
<!doctype html>

...

<script>
var fields;
function init(){
  fields = document.forms[0].elements;
  loadPrefs()
}


function loadPrefs () {
  for(var i = 0; i < fields.length; i++){
     var field = fields[i];
     if (typeof widget.preferences[field.name] !== "un
        field.value = widget.preferences[field.name];
     }
  }
}

function savePrefs () {
  for(var i = 0; i < fields.length; i++){
    var field = fields[i];
    try{
      widget.preferences.setItem(field.name,field.valu
    }catch(e){
       if(e.code === DOMException.NO_MODIFICATION_ALLO
          alert(e);
       }
    }
   }
  }
</script>

<body onload="init()">
  <fieldset id="prefs-form">
  <legend>Album Playback Settings</legend>
  <form>
  <label>Volume:
  <input type="range" min="0" max="100"
         step="10.0" value="50" name="volume">
  </label>
  <label>Playback:
  <select name="playorder">
     <option value=0>Loop
     <option value=1>Random
     <option value=2>Normal
  </select>
  </label>

  <label>Favorite Song

  <select name="favtrack">
     <option value=kate>Kate Bush: Babooshka
     <option value=billy>Billy Idol: White Wedding
     <option value=culture>Culture Club: Karma Chamele
  </select>
  </label>

  <label>License Key: <input name="licenseKey"></label
</form>
<button onclick="savePrefs()">Save</button>
<button onclick="loadPrefs()">Load</button>
</fieldset>

...
```

7 The `WidgetStorage` Interface

The `WidgetStorage` Interface extends [Web Storage]'s `Storage` interface so that it can provide the necessary functionality provided by this specification. It does not add any new methods or attributes; it just provides a wrapper that makes it easier to implement on some platforms.

```
[NoInterfaceObject]
interface WidgetStorage : Storage {
};
```

7.1 Example - comparing storage areas

This example shows how to work out if a storage event came from the widget or from the Web Storage's `localStorage`.

```
<!doctype html>
<script>
//note that this code is only really useful inside an iframe!
window.addEventListener("storage", function handleStorage(event){
   if (event.storageArea === widget.preferences) {

      //the event was fired by the widget

   } else if (event.storageArea === window.localStorage ||
              event.storageArea === window.sessionStorage){

      //the event was fired by the Web Storage

   } else {

      //the event was fired by some other object.

   }
});
</script>
```

8 Widget Storage Areas

A **widget storage area** is a data-store that is unique for the widget instance that uses [Web Storage]'s `Storage` interface but modifies the behavior of [Web Storage] by allowing some items to be read-only. A user agent uses a widget storage area to store key-value pairs that pertain to the `preferences` attribute. An author script interfaces with the widget storage area via the [Web Storage] specification's `Storage` interface.

A user agent must preserve the values stored in a widget storage area when a widget is re-instantiated (i.e., when the device is rebooted and the widget is reopened, the previously set data is made available to the widget storage area).

8.1 Read-only Items

As an extension to the [Web Storage] specification, a widget storage area allows certain key-value pairs (items) to be read-only: a **read-only item** is an item in a widget storage area that cannot be modified or deleted by an author script. A read-only item always represents a preference that author explicitly flagged as "read-only" in the widget's configuration document (denoted by a `preference` element having a `readonly` attribute value set to `true`).

9 Getting Localizable Strings

The **rule for getting localizable strings** is as follows:

1. Let *IString* be a copy of the localizable string to be processed.
2. If *IString* has no directional information associated with it (i.e., no `dir` attribute was used anywhere in the ancestor chain of the element or attribute in question), and the localized string does not contain any sub-strings with directional information within the string itself, return the *IString* and terminate this algorithm.

   For example, the consider the following configuration document :

   ```
   <widget xmlns = "http://www.w3.org/ns/widgets"
       version   = "1.0">
      <name>Hello</name>
   </widget>
   ```

   Would result in the following in the API:

   ```
   <!doctype html>
   <script>
     alert(widget.version === "1.0")   //returns true
     alert(widget.name    === "Hello") //returns true
   </script>
   ```

3. If the *IString* contains directional information and/or contains any sub-strings with directional information, then recursively do the following from the outermost string to the inner most sub-string of *IString*:

   A. Let *direction* be the direction of the sub-string.

   B. Prepend one of the following Unicode characters to the sub-string based on matching the following directions:

      `lro`

U+202D 'LEFT-TO-RIGHT OVERRIDE'.

`ltr`
U+202A 'LEFT-TO-RIGHT EMBEDDING'.

`rlo`
U+202E 'RIGHT-TO-LEFT OVERRIDE'.

`rtl`
U+202B 'RIGHT-TO-LEFT EMBEDDING'.

C. Append the sub-string with a U+202C 'POP DIRECTIONAL FORMATTING' character.

D. If the sub-string contains any further sub-strings with directional information repeat the steps A-D in this algorithm.

4. Return *IString*.

9.1 Example 1

This section is non-normative.

The following configuration document demonstrates how having an `dir` attribute is handled by the Widget API:

```
<widget xmlns = "http://www.w3.org/ns/widgets"
    version   = "1.0"
    dir       = "ltr">
   <name>Hello</name>
</widget>
```

Would result in the following in the API:

```
<!doctype html>
<title>Example 1</title>
<body style="background-color: #ECEDCF">
<p id ="name"></p>
<p id = "version"> </p>

<script>
var nameElem    = document.getElementById("name");
var versionElem = document.getElementById("version");

nameElem.innerHTML    = 'The widget's name is "' + widget.name +
                        '". <br>Escaped, the value of name is "' +
                        escape(widget.name) + '".';

versionElem.innerHTML = 'The widget\'s version is "' + widget.version +
                        '". <br> Escaped, the value of version is "' +
                        escape(widget.version) +'".';
</script>
```

Would render as:

The widget's name is 'Hello'.
Escaped, the value of name is [%u202AHello%u202C].

The widget's version is '1.0'.
Escaped, the value of version is [%u202A1.0%u202C].

9.2 Example 2

This section is non-normative.

Given this configuration document, where the widget element has `dir` set to `ltr` and name has a `span` element with a `dir` attribute set to `lro`:

```
<widget xmlns = "http://www.w3.org/ns/widgets"
    version   = "1.0"
    dir       = "ltr">
   <name><span dir="rlo">olleH</span></name>
</widget>
```

The following would result in the start file of the widget:

```
<!doctype html>
<title>Example 2</title>
<body style="background-color: #ECEDCF">
<p id ="name"></p>
<p id = "version"> </p>

<script>
var nameElem    = document.getElementById("name");
var versionElem = document.getElementById("version");
nameElem.innerHTML    = 'The widget\'s name is "' + widget.name +
                        '". <br>Escaped, the value of name is "' +
                        escape(widget.name) + '".';

versionElem.innerHTML = 'The widget\'s version is "' + widget.version +
                        '". <br> Escaped, the value of version is "' +
                        escape(widget.version) +'".';
</script>
```

Would render as:

The widget's name is 'Hello'.
Escaped, the value of name is "%u202A%u202EolleH%u202C%u202C".

The widget's version is '1.0'.
Escaped, the value of version is "%u202A1.0%u202C".

9.3 Example 3

This section is non-normative.

Given this configuration document, where the widget element has no base direction set, the `name` has two `span` element with a `dir` attribute set to `lro`:

```
<widget xmlns = "http://www.w3.org/ns/widgets"
    version   = "1.0">
     <name>
          Hello1
                        <span dir="rlo">2olleH</span>

                        Goodbye1
          <span dir="rlo">2eybdooG</span>
    </name>
</widget>
```

The following would result in the start file of the widget:

```
<!doctype html>
<title>Example 3</title>
<body style="background-color: #ECEDCF">
<p id ="name"></p>
<p id = "version"> </p>

<script>
var nameElem    = document.getElementById("name");
var versionElem = document.getElementById("version");
nameElem.innerHTML    = 'The widget\'s name is "' + widget.name +
                        '". <br>Escaped, the value of name is "' +
                        escape(widget.name) + '".';
versionElem.innerHTML = 'The widget\'s version is "' + widget.version +
                        '". <br> Escaped, the value of version is "' +
                        escape(widget.version) +'".';
</script>
```

Would render as:

The widget's name is 'Hello1 Hello2 Goodbye1 Goodbye2'.
Escaped, the value of name is
"Hello1%20%u202E2olleH%u202C%20Goodbye1%20%u202E2eybdooG%u202C".

The widget's version is '1.0'.
Escaped, the value of version is [1.0].

Revision history

19 April 2012

Clarified "the origin of a widget instance" based on feeback we received that indicated it was unclear.

5 December 2011

Added example of how to compare storage areas.

Added `WidgetStorage` interface.

13 June 2011

Clarified storage event text (hopefully).

Editorial cleanup, found a few conformance requirements that were not being tested.

Removed the definition of valid IRI, as it was not referenced anywhere.

Removed the definition of feature, as it was not referenced anywhere.

Added examples for i18n related material.

7 September, 2010

The 7 September, 2010 version of the specification drops support for the `openURL` method, which was part of previous versions of this specification. The Working Group found a number of privacy and security issues relating to `openURL`, as well as a way to achieve the same intended functionality via other means, and hence decided to drop it from the specification.

The working group recommends that authors use HTML's `a` element to achieve the same functionality, or use the `window.open()` method where appropriate. Some examples of how the a element can be used to achieve the same functionality as `openURL` are given below. Of course, the examples will only work on implementations that actually have scheme handlers to handle each type of URI.

Send email:
Was: `openURL("mailto:jsmith@example.org?subject=A%20Hello&body=hi")`
Now: `<a href="mailto:jsmith@example.org?subject=A%20Hello&body=hi">Email Jane</a>`
Make a phone call:
Was: `openURL("tel:+1234567678")`
Now: `<a href="tel:+1234567678">Call Bill!</a>`
Open a web page:
Was: `openURL("http://example.org")`
Now: `<a href="http://example.org">Example</a>`
Send and sms:
Was: `openURL("sms:+41796431851,+4116321035;?body=hello%20there")`
Now: `<a href="sms:+41796431851,+4116321035;?body=hello%20there">SMS Bob</a>`

Normative References

[CSS]
Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification. W3C.
[DOM3Core]
Document Object Model (DOM) Level 3 Core Specification. W3C.
[DOM2Events]
Document Object Model (DOM) Level 2 Events Specification. W3C.
[RFC2119]
Key words for use in RFCs to Indicate Requirement Levels. IETF.
[Widgets-Packaging]
Widget Packaging and Configuration. W3C.
[WebIDL]
Web IDL (Work in progress). W3C.

[Web Storage]
Web Storage. W3C.

[HTML]
HTML (Work in progress). W3C.

Informative References

[Interface-Test-Suite]
Test Suite for the Widget Interface Specification.

[Widget Requirements]
Widgets Requirements. W3C.