

OpenAjax Widgets Proposal (DRAFT v0.1)

Abstract

There are a number of widget definitions and providers with a great amount of overlap in approaches and concepts. Each widget definition has its own set of runtime and presentation characteristics. Some consolidation among widget providers may occur but many widget definitions will continue to exist and thrive. The goal of this document is to define a lightweight metadata format and a thin runtime environment that enables application developers and mashup assemblers to create composite assemblies of loosely coupled pre-packaged application components from a variety of widget providers.

Widgets come in many shapes and sizes this document will use the definition of 'web widgets' from [Wikipedia](#)

"A web widget is a portable chunk of code that can be installed and executed within any separate HTML-based web page by an end user without requiring additional compilation. They are derived from the idea of reusable code that has existed for years."

Table of Contents

<i>OpenAjax Widgets Proposal</i>	
<i>Abstract</i>	
<i>Table of Contents</i>	
<hr/>	
<i>Introduction</i>	
<i>Use Cases</i>	
<i>Model</i>	
<i>Widget Specification Format</i>	
Widget	
Widget Dependencies	

- Widget Instance Properties**
- Example of widget properties:
- Widget Content**
- Content Examples:
- Widget Lifecycle Management**
- Widget Wrapper**
- Widget Creation**
- Widget Destruction**
- Widget Property Management**.....
- (ED: Additional Sections/Topics in progress)*
- OpenAjax Hub Integration**
- CSS**
- Translations**
- Widget Event Handling**
- Predefined Events**

DRAFT

Introduction

There are several efforts within the OpenAjax organization that are key building blocks for the achieving interoperability among widgets definitions:

- Interoperability Work Group – provides the ability to combine JavaScript components from multiple Ajax toolkits within the same Web application and have those components interact with each other.
- Communications Hub Task Force - defines a publish/subscribe event manager, which enables loose assembly and integration of Ajax components
- IDE Work Group - industry standards for describing Visual and non-visual Controls and Ajax library APIs for use in design-time and potentially run-time scenarios
- SMash (Secure Mashup) - a set of technique and open source JavaScript that runs in today’s browsers (without extensions or plugins) and enables secure handling of 3rd party mashup components.

Use Cases

A key umbrella use case for the OpenAjax Widgets is the set of scenarios where an assembler needs to create a single composition that uses multiple Ajax libraries simultaneously. Key use cases for Widgets is portals and mashups, where the assembler creates a page of loosely assembled pre-packaged application components. Portals and mashups typically pull in Ajax-powered components that are built using multiple different Ajax toolkits.

The following are some of the scenarios where developers will find a need for using multiple Ajax libraries simultaneously:

- Ajax developer finds that a single Ajax library does not meet all of his requirements - In some cases, an Ajax developer might choose to build the majority of his application using a particular Ajax library, but discover that a particular component from that library does not have the features he needs, and therefore would like to incorporate a component from a different Ajax library, leaving the rest of the application unchanged. In other cases, most of the application might be (non-Ajax) DHTML, but the developer would like to incorporate two components of Ajax technology into the application, where the two components come from different Ajax libraries.
- Mashups - In order to build a composite application from pre-existing application components, a developer (or end-user) is likely to have to combine components based on different Ajax libraries.
- Portals - Portals support Ajax technologies and represent a variation on composite applications. It is common that different portlets use different Ajax libraries.

Model

The proposed approach does not include the creation of a new component model, but rather a simple metadata layer plus a minimal mapping or adapter layer to the native widget environment. Developers will continue to create widgets in their choice of component models, the goal as stated in the Use Cases is to enable the widget assembler to easily use widgets from a variety of native formats to create composite applications or mashups. The metadata describe below is intended to convey enough information so that assembly tools can understand the properties and possible event wiring capabilities of a given widgets. The mapping layer will propagate lifecycle events (e.g. onAdd, onDelete) related to the assembly process as well as events for properties changes. The metadata also contains *require* elements which declare conforming toolkit references and configuration information. These are placeholders for a more direct mapping to other efforts within the OpenAjax.org, for example the Registry.

Widget Specification Format

Widget

A widget contains a set of attributes that describes characteristics of the widget within a global context. For example, the widget attributes can be used by a composition canvas to place the widget on a palette for use within a composition.

The widget attributes are directly available from the widget wrapper instance managed by the runtime library.

1. title – OPTIONAL string that is the title of the widget that may be used by the composition canvas as decoration around the widget.
2. website – OPTIONAL string what is a URI for the widget author's website
3. description - OPTIONAL string that describes the widget. NOTE: The description SHOULD NOT contain XHTML markup
4. height - OPTIONAL positive integer that specifies the height of the area in which the gadget runs. The default height is 200.
5. width - OPTIONAL positive integer that specifies the width of the area in which the gadget runs. The default height is 200.
6. author – OPTIONAL comma separated string containing the name(s) of the author(s) of the widget
7. email – OPTIONAL comma separated string containing the email(s) of the authors(s). NOTE: This value must be in the same occurrence order as the author attribute.
8. thumbnail – OPTIONAL string that gives the URL of a widget thumbnail. This must be an image on a public web site that is not blocked by robots.txt. PNG is the preferred format, though GIF and JPG are also acceptable. Widget thumbnails should be (ED: need default size) X by Y pixels. The URL can be absolute or relative to the assets contain within the widget package.
9. category – OPTIONAL string describing a default categorization of the widget (e.g finance, sports).
10. scaling - OPTIONAL boolean that specifies whether the aspect ratio (height-to-width ratio) of the widget is maintained when the browser is resized. Widgets that can automatically scale vertically should set this to true, but widgets which have a fixed height should set this to false. The default is true.
11. scrolling - OPTIONAL boolean that provides vertical and/or horizontal scrollbars if the content exceeds the space provided. If false, then the content is clipped to the height and width provided (not that width is not configurable). The default is false.
12. scope – OPTIONAL string which specifies the name of the a JavaScript constructor function to use when creating the widget wrapper for an instance of the widget (See Widget Wrapper below), the default value is `OpenAjax.widgets.WidgetWrapper`.

Sample:

```
<widget xmlns ="http://openajax.org/widget"
  name="GoogleMap"
  scope="GoogleMap"
  description="This is a simple example of a googlemap widget...">
...
</widget>
```

Widget Dependencies

A widget can declare the toolkit dependencies, configuration and other widgets on which it depends upon. The *require* tag within the widget specification contains the following attributes:

- library – OPTIONAL string specifying the Ajax Toolkit that is used to implement the widget. Conforming OpenAjax libraries names are the allowable values for this attribute. (ED: How will this be managed? OpenAjax Registry?)
- version – OPTIONAL string specifying the version number of the Ajax Toolkit required. (ED: Can this be combined with the library name from above when interacting with the registry)
- src – OPTIONAL string representing a URL to a JavaScript file that will be included with the widget.
- config –OPTION attribute or child describing the configuration for a given library.

(ED: For conforming libraries this information can be composition wide, for example, what happens when you have competing configuration requests? IFRAME? Need to integrate with the Registry if appropriate)

Sample:

```
<widget xmlns ="http://openajax.org/widget"
  name="GoogleMap"
  scope="GoogleMap"
  description="This is a simple example of a googlemap widget...">

  <require library='dojo' version="0.9">
    <config>
      "parseOnLoad: false, isDebug: true, extraLocale: ['en-us',
'ar-sy', 'es-es', 'zh-cn']"
    </config>
  </require>
...
</widget>
```

Widget Instance Properties

Widgets contain properties whose values affect the behavior and display characteristics of the widgets. The properties of a widget can be set by the user using a general purpose property editing tool or a widget specified property editing.

Each property of the widget will contain the following attributes:

1. name – REQUIRED string that is unique within the context of the widget’s properties that specifies the name of the property.
2. display_name – OPTIONAL string that represents property name that will be displayed to the end-user while editing the property.
3. description – OPTIONAL string that contains a longer description of the property that MAY contain XHTML.
4. short_description – OPTIONAL string that contains an abbreviated description of the property, appropriate for tooltip style of usage.
5. scope – OPTIONAL string specifying the scope of the property regarding the widget instance, valid values are ‘instance’ or ‘static’, the default value is instance. Instanced scope specifies that each instance of the widget contains its own copy of the property. Statically scoped properties are global all instance of the widget. (ED: What about semantics of static as it relates to ‘site’ widget instances)
6. type – OPTIONAL string that specifies the data type of the property, default value is string. Allowable values are (string, bool, hidden, integer, float, currency, enum, array, object, URL, location, color, font). (ED: How is this list extensible? QNAMES?)
7. default_value – OPTIONAL string that specifies the default value of the property. If the type is an enumeration this value represents the selected value of the enumeration. If the type of property repeats the default_value is ignored.
8. room – OPTIONAL string that specifies a message room in which the property participates as an event source or target. “A "Room" (shorthand for Message Room) is a common area to which multiple agents can connect. Within a Room, agents send and receive messages to/from each other via publish/subscribe APIs.” (see OpenAjax Hub 1.1 for more detail http://www.openajax.org/member/wiki/OpenAjax_Hub_1.1_Roadmap).
9. listen – OPTIONAL Boolean specifying this property is an target for events delivered to the room property above. Default value is false.
10. publish – OPTIONAL Boolean specifying this property is an event source. The event is delivered to the OpenAjax Hub using the room specified above.
11. onChange – OPTIONAL string specifying a JavaScript function or method to invoke when the property changes, if listen is true. The implementation will first check for a function within the scope of the widget and then check for a method on the widget wrapper of the specified name

TBD:

- | Grouping or batching event changes into a single event.
- | Additionally, the listen/publish/onChange properties only focus on binding NOT on the format of the event payloads. How do we represent more

complex properties, for example, a list of locations each of which consists of an address, icon URL and display text?

How is Point-to-point wiring accomplished?

Example of widget properties:

Simple:

```
<properties name="userProperties" description="General properties
available to the user">
  <property name="firstName" display_name="First Name"
short_description="Your first name">
    <description>You must specify your <b>first name</b> in
this field</description>
  </property>
  <property name="lastName" description="Attribute style of
description"/>
</properties>
```

Example of a statically scoped property:

```
<properties>
  <property name="bgColor" scope="static" display_name="Share
background color" type="color"
    default_value="white"/>
</properties>
```

Example of an enumeration property type:

```
<properties>
  <property name="bgColor" type="enum" default_value="white">
    <properties>
      <property default_value="red" type="color"
display_name="Red"/>
      <property default_value="green" type="color"
display_name="Green"/>
      <property default_value="blue" type="color"
display_name="Blue"/>
    </properties>
  </property>
</properties>
```

Example of a list of lat/long and icon URL

```
<properties>
  <property name="locations" type="array">
    <properties>
      <property name="lat" type="string"/>
      <property name="long" type="string"/>
      <property name="icon" type="URL"/>
    </properties>
  </property>
</properties>
```

```
        </properties>
    </property>
</properties>
```

ED: We need a simple mechanism for representing anonymous JavaScript objects, in the example above the properties clause is used as an anonymous collector. We do NOT want to define yet another mechanism for specifying complex types. Can new properties reference to existing properties as new types?

Example of an Object property:

```
<properties>
  <property name='location' type='object'>
    <properties>
      <property name="lat" type="string"/>
      <property name="long" type="string"/>
      <property name="icon" type="URL"/>
    </properties>
  </property>
</properties>
```

Widget Content

The widget content consists of an XHTML fragment that is spliced into the composite assembly. The implementation canvas is responsible for performing the page or desktop composition of widgets as defined by the user. The *content* element contains the widget content or references the content via a reference. A widget may contain several *content* sections, the content modes supported are:

- view – the default, which specifies the main view to the widget
- edit – OPTIONAL section that specifies the Content used to edit the properties of the widget
- help – OPTIONAL section that specifies the help content of the widget

Below are the attributes of the content section:

- mode – OPTIONAL string specifying the mode of the content section (see Content section above), default value is “view”
- href – OPTIONAL string specifying a URL that returns the content of the widget.
- type – OPTIONAL string specifying how to treat the content contained in the section. Type supported include:
 - html – content is to be spliced inline with the page or assembly canvas
 - uri – the href points to the content of the widget, in which case an IFRAME is spliced into the assembly canvas pointing to the widget
 - frame – the content of the section is contained within a IFRAME for security and other runtime related requirements, this is the default

Content Examples:

Simple Content

```
<content mode="view" type="html">
<![CDATA[
    Hello World, widget content
]]>
</content>
```

HREF Content

```
<content mode="view" type="uri"
href="http://www.acme.com/widgetcontent.xml" />
```

Widget Lifecycle Management

A composition canvas will need the ability to:

- Create a widget and add it to the composition surface
- Query the properties of a widget along with their current values
- Update the properties of a widget
- Persist the properties of a widget
- Remove a widget from the composition canvas
- Wire event producers with event consumers

Communication between the widget and the host environment is accomplished through the Widget Wrapper instance.

Widget Wrapper

Each instance of a widget is represented using a JavaScript object which provides the communication between the host environment and the native widget executing in a given widget toolkit. The scope attribute of the widget specifies an optional JavaScript constructor function to use when instantiating a given widget, the default value is `OpenAjax.widgets.WidgetWrapper`.

The `WidgetWrapper` class is defined as follows:

NOTE: Implementations of the `WidgetWrapper` may contain a more elaborate interface, however, additional protocol is considered implementation specific and **SHOULD NOT** be used as a general purpose interface.

```
/**
 *Generic wrapper classes for widgets
```

```

* @constructor
*/
OpenAjax.widgets.WidgetWrapper = function() {};

with (OpenAjax.widgets.WidgetWrapper) {
  /**
   * Sets the value of the specified property name to the value
   * @param {String} propertyName This is the name of the
property
   * #param {Object} value The value of the property
   */
  prototype.setPropertyValue = function(propertyName, value) {};

  /**
   * Returns the current value of the requested property name.
   * @param {String} propertyName This is the name of the
property to retrieve
   *
   */
  prototype.getPropertyValue = function(propertyName) {};

  /**
   * This method requests that the wrapper save the current state
of the widget properties.
   * The same method operates asynchronously, if you wish to
determine the success/failure
   * of the method pass a callback parameter.
   * @param {Function} saveCallback This is a function that is
invoked with the result of the
   * save operation
   */
  prototype.save = function(saveCallback) {};
}

```

The saveCallback signature is as follows:

```

/**
 * Save callback exemplar
 * @param {XMLHttpRequest} xhrUsed Queryable for the status codes, etc
 * @param {WidgetWrapper} The widget instance whose values were just
saved
 */
function saveCallback(xhrUsed, widgetWrapper) {}

```

Widget Creation

The composition canvas uses the widget description document to determine the runtime characteristics of the widget. Adding a widget to a composition consists of the following steps:

- o Load the widget description document (
- o Determine the requested content mode of the widget(e.g. view, edit, help)

- Locate the content and splice the content into the composition canvas (ED: TBD locale expansion)
- Register the subscribers with the OpenAjax Hub as defined in the description document

The implementation will transform the widget document expanding the `__WIDGETID__` eye-catcher to the actual ID to the widget being created. IDs are required to be unique within the context of the assembly canvas.

```
<content type='html'>
  <![CDATA[
    <div id="__WIDGETID__map" style="width: 100%; height:
430px"></div>
    </div>
  ]]>
</content>
```

After replacement the content may look like this:

```
<content type='html'>
  <![CDATA[
    <div id="ID_1234567890map" style="width: 100%; height:
430px"></div>
    </div>
  ]]>
</content>
```

The expansion enables the same widget fragment to be used multiple times on a single composition and avoid name collisions.

NOTE: It is not intended that the widget description document be the persistent format for widgets on a particular composition canvas. Implementations are free to persist the property values and wiring relationships in a more appropriate (efficient) scheme.

(ED:

Do widget need to describe when a page refresh is needed or they only run in an IFRAME mode?

Is there a factory for widgets from conforming Toolkits?

)

Widget Destruction

The predefined `onDelete` event is triggered on a widget before it is removed from the composition canvas. This event is synchronous so that the widget has an opportunity to free up resources before the widget's removal. It is the responsibility of the composition canvas to remove any additional adornment associated with managing the widget from the canvas. Additionally, the implementation is responsible for unwiring the widget from the OpenAjax Hub, that is, removing listeners via the `unsubscribe` call.

Widget Property Management

The list of supported properties, types, names, definition and wiring capabilities are contained in the widget description document (WDD). The composition canvas will consult the WDD and use the public API of the WidgetWrapper to query and set the values of the properties exposed by a given widget. The WidgetWrapper implementation is responsible for persisting the widget properties.

(ED: Additional Sections/Topics in progress)

OpenAjax Hub Integration

The *listen* and *publish* property modifiers defined in the Widget Instance Properties section above are used to wire the widget to the OpenAjax Hub V 1.1. The implementation is responsible for opening a connection to the Hub, adding listeners to the Hub and publishing property change events to the Hub.

CSS

(ED: Requirements:

Mixing widgets from a variety of toolkit poses interesting challenges especially if each widget uses different styles. We have the following requirements:

- | Widgets need to be able to load style sheets dynamically. Do we expand the WidgetWrapper with a service factory protocol that enables dynamically loading style sheets?
- | The style sections can be defined in the content section of the widgets.
- | Do we formalize the dependencies on style sheets so that an implementation can optimized/redefine as appropriate?
- | Do we support any type of 'themes' across widgets?

Translations

The translation into difference locales will be down using a simple macro style expansion mechanism that is employed during the widget preparation phase.

(ED:

- | What is the format of message bundles?
 - | What do the eye-catchers look like in the widget description document?)
-)

Widget Event Handling

Widgets can both produce and consume events that are mediated by the OpenAjax Hub.

The scenarios for widget interaction include:

- | Browser Page to Browser Page communication
- | Browser Page to Widget communication
- | Widget to Widget communication

Predefined Events

Implementations will support a set of predefined events that are delivered to the WidgetWrapper during assembly and runtime.

(ED: Examples are: onAdd, onDelete...) TBD...