

Protowidget Reference Manual

Redcode Technologies

June 11, 2006

Contents

1	Introduction	2
1.1	About This Document	2
1.2	Document Conventions	2
2	Concept Reference	3
2.1	Class System	3
2.1.1	BaseClass	3
2.2	Advice	4
2.3	Components	4
2.3.1	Properties	4
2.3.1.1	Custom 'get' logic	5
2.3.1.2	Custom 'set' logic	5
2.3.1.3	Defining properties	5
2.3.2	Scope	5
2.3.3	Attributes	6
2.3.4	Property Listeners	6
2.3.5	Bindings	6
2.4	Widgets	7
2.4.1	HTML Integration	7
2.4.1.1	Custom Attributes	8
2.4.1.2	CSS Integration	8
2.5	Models	9
2.6	Templates	9
2.7	Input Controls	9
2.8	Repeaters	9
2.9	Integration with HTML	9
2.10	Logging	9
3	Core API Reference	10
4	Widget Reference	11

Chapter 1

Introduction

The best tools are born out of pain. With regard to existing Ajax tools, here is a brief list of some of the pain that Protowidget attempts to address:

- ID-madness: Some frameworks (ie. Dojo) attempt to avoid this, but most make the primary coupling point between JavaScript and HTML be the ID attribute. While many have shown this to work, it is somewhat offensive to have only one flat namespace. This becomes an especial problem when trying to create repeating sections or scoped, re-usable components. Since the namespace is flat, it also causes problems when one component needs to logically interact with others.
- Non-standard UI construction: At the end of the day, HTML+CSS is a pretty good tool for laying out web-based UIs. We can certainly come up with better representations, but they are rarely as rich and they are typically not designer friendly. Despite a lot of other great ideas, I consider this to be OpenLaszlo's primary failing. Other frameworks which take the Swing-esque approach of coding the UI programatically by assembling objects are even weaker.
- No client side model: Years ago someone co-opted the long standing MVC design pattern and claimed that having servlets and views somehow conformed to the pattern. Maybe it technically does, but it is certainly not in the spirit of MVC as seen in real client-side apps. Today, we have an interesting inversion going on where most people think that the web-MVC (type II or whatever) pattern is classical-MVC. We need to be applying some classical-MVC principles to our client-side interfaces. Otherwise, we will end up with as much spaghetti as can be found in an old-school VB6 app. OpenLaszlo and Flex with their pervasive property bindings provide the building blocks for this type of development. Few others do.
- Separating form from function: This is the holy grail, and likely no tool will ever get 100% there. The idea is that the visual layout and characteristics should be specified out-of-band from the code that manipulates it. A good framework should have well-defined intersection points where the visual bits mix and interact with the programmatic bits.

Protowidget addresses these concerns by leveraging the simplicity of Prototype, the concept of widgets/templates from Dojo, and the event-based property binding methodology from OpenLaszlo/Flex.

1.1 About This Document

This document is a reference to the core Protowidget libraries. It provides conceptual and API references for all public facilities. Since the library is in JavaScript, the notion of public and private is not as clearly delineated as it is in some other languages. In this context, *public* refers to the parts of the libraries that are intended for consumption by users either through invocation or sub-classing. The intention is that this manual outlines the parts of the library that are stable from an API perspective. Any parts of the library not documented in this manual should be considered unstable and subject to change.

1.2 Document Conventions

TODO

Chapter 2

Concept Reference

2.1 Class System

While JavaScript provides the core structures needed to implement classes and inheritance, it does not provide built-in facilities for maintaining complex class structures. There are a number of external libraries available which fill this gap by providing more advanced functionality atop JavaScript's prototypical inheritance facility. At this time, however, Protowidget does not use these libraries and elects to instead define its own simple, augmented class system. The entire Protowidget class facility is defined by the BaseClass constructor. Applications are free to extend from BaseClass to create custom classes; however Prototype's built-in *Class.create* facility is simpler and sufficient for classes which do not participate in complex inheritance relationships. The overhead of extending BaseClass is higher than using a simpler scheme such as *Class.create*.

2.1.1 BaseClass

The *Protowidget.BaseClass* constructor is an abstract super-class that forms the root of most Protowidget class hierarchies. It provides facilities similar to other OO languages such as subclassing, static and instance members, super-method invocation, and linkage from the instance to the class. It also adds the concept of *Advice*¹ which is a conceptual cross between aspects from AOP and Ruby mixins.

BaseClass will primarily be used by calling its *subClass* method to create a subclass. (TODO: see reference) The *subClass* method takes a single argument which is an Object containing the instance members which will be added to the class's prototype. There are a couple of special properties which can be passed in the members object and play a special role for the created class:

initialize Called when the class is instantiated with the new operator. All arguments passed to the constructor are passed to this method.

override If present, then this array contains the names of methods which override corresponding methods on the super-class. Each will be annotated to support invocation of the super method. See ??.

staticInitialize If present on a class or super-class, this method will be called on the prototype object during class construction. See ??.

className This string is the logical name of the class and will be copied to the newly created constructor's *className* field.²

Calling *subClass* will yield a new constructor representing the newly created sub-class. The constructor will have the following static members:

superType A reference to the constructor for the class from which this class was derived.

className The name of the class (may be undefined if no name was given).

applyAdvice(adviceName, options) Applies static advice to the class

subClass(members) Create a sub-class.

¹Following AOP nomenclature, this would have been more aptly named Aspect, but the distinction was lost during the wee hours of the morning when this feature was added.

²Defining classes by way of the *Protowidget.define(...)* method will set this field automatically.

Advice An object that contains advice which is eligible to be applied to this class or its instances.

Every instance of a Class derived from BaseClass will contain a *type* field which references the constructor used to create it. See the API reference for BaseClass for more information.

2.2 Advice

The Protowidget class system contains a facility inspired by Aspect Oriented Programming and Ruby mixins. Since JavaScript classes are never “closed”, the impetus to create deep class hierarchies in order to perform highly specialized functions is diminished. Instead, it is easier in many cases to systematically alter a class or its instances to enable a mixture of functionality that could not be otherwise achieved in a single-inheritance environment. Protowidget defines the concept of *Advice* for this purpose.

In Protowidget, a piece of Advice (known as an Advice Function) is just a JavaScript function which is called on a class prototype (static) or an instance (dynamic) in order to alter the class or instance in some way. The advice function is invoked as if it is a member of the instance being altered so that the *this* reference within the function refers to the instance being altered (or the *prototype* if being applied statically). Advice Functions can take an optional argument which will be passed through to the Advice Function from the *options* argument of the *applyAdvice* method. If specified, it should be an object with optional parameters needed to apply the advice.

Advice is applied by calling the *applyAdvice* method on an instance or on the class constructor. This method will invoke the *findAdvice* method to lookup the relevant advice by name for the class on which it is called. The *findAdvice* method searches the *Advice* field of all class constructors up the inheritance chain until it finds an entry with the given name.

While the concept of Advice is powerful enough as a programming device, it is made even more so when coupled with Protowidget’s HTML integration. When declaring a widget in HTML, a list of Advice names and options can be specified. When the widget is instantiated all of the listed advice will be applied to the instance. This allows for massive customization of a widget’s behavior in a declarative fashion. This is especially useful for complex widgets such as Text Fields which can exhibit a widely varying set of behaviors based on the context in which they are used. For example, the TextField widget could be instantiated with advice telling it to only accept numeric input, and export its value to the model in real-time. In other widget systems, this effect would require either a dizzying number of specialized sub-classes or extremely complicated widgets with conditional logic to determine when various features should be activated.

2.3 Components

The Component class extends BaseClass to provide support for managing properties on classes. It provides the following features:

- Declarative generation and definition of getters and setters (see *properties* field, *defineProperty(...)*, *defineProperties(...)*)
- Property change events (see *addPropertyListener*, *removePropertyListener*, *firePropertyChange*)
- Support for custom property get/set behavior (by declaring *__set__prop* and *__get__prop* methods)
- Property path navigation in the form of attributes (see *getAttribute*, *setAttribute*)
- Binding a property to the value of another through an attribute declaration (see *bindProperty*, *unbindProperty*)
- Generic property access by name (see *getProperty*, *setProperty*)

This section will present the concepts needed to understand components. Refer to the API reference for further details.

2.3.1 Properties

By default, properties correspond with fields on the instance. They share the same name, so a property with the name “foo” will be represented as a field on the object named “foo.” Getters and setters for the property are named by the Java conventions whereby the first character of the property name is capitalized and appended to either “get” or “set”. For example, a property named “foo” would be stored in field “foo” on the object instance and accessed with *getFoo()* or *setFoo('bar')*. When properties are stored according to this default scheme, it is acceptable to read the value through a raw field reference (eg. *object.foo*). The property must be set, however, using either the dedicated setter method (eg. *setFoo('bar')*) or the generic *setProperty* method (eg. *setProperty('foo', 'bar')*). If fields are updated directly, then the system will be unable to notify listeners of property change events. Bypassing the setter/*setProperty* method

should only be done during object initialization or under other special circumstances when the implications are understood. In general, reading properties should be done through the `getProperty` or `getter` method even though it is normally possible to read the field directly. It may be desirable in some circumstances, however, to read the field value directly. Accessing properties in this way should generally be limited to code that is part of the same logical unit, since it breaks the public api abstractions.

2.3.1.1 Custom 'get' logic

When a property is read via `getProperty` or its `getter` (which internally calls `getProperty`), the default behavior is to return the value of the object field with the given property name. This behavior can be overridden by defining a special method on the class/object. This method is named `__get__property`, where *property* is the name of the property. The method takes zero arguments and must return the value of the property.

2.3.1.2 Custom 'set' logic

Similar to custom 'get' logic, custom set logic can be defined in a method named `__set__property`, where *property* is the name of the property. The method takes a single parameter: the new value. In order to determine whether the property has changed (in order to fire events), the `__set__` method must return the original value in effect prior to making the change.³

2.3.1.3 Defining properties

Properties are implicitly created when accessed via any of the methods that operate on properties: `getProperty`, `setProperty`, `addPropertyListener`, `removePropertyListener`, `bindProperty`, and `unbindProperty`. Getters and setters, however, must be declared manually. If a property needs to be accessed via `getter` and `setter`, then one of the following steps must be taken in order to have the `getter` and `setter` created:

1. Calling the `defineProperty` method with the name of the property. This method can be called on the prototype (ie. `MyClass.prototype.defineProperty`) to define class getters and setters. Alternatively, it can be called on an instance to define getters/setters for just that instance (ie. `myObj.defineProperty('foo')`).
2. Bulk defining properties with the `defineProperties` method. The method takes a varying number of parameters, each one naming a property to define. The method can be called in the same context as `defineProperty`.
3. Adding a 'properties' field to the members object passed to the `Component.subClass` method. The value should be an array of property names to be defined on the class.

2.3.2 Scope

Whenever creating a system where named entities need to interact with each other, it is wise to consider carefully the question of *scope*. Much has been lost in otherwise robust systems by not defining adequate scoping rules (or leaving them out entirely and just assuming a global namespace).

In `Protowidget`, every `Component` (and by extension, every `Widget`) has associated with it a *scope*. Internally, the scope is implemented as a `Protowidget.Scope` object located on the component as the `_scope` field. The scope is used as a basis for resolving *attribute* references (see the next section).

Each scope has a *Scope Id*, a *Parent Scope*, and a *Delegate* – that is to say that every scope has a name, references a parent scope, and contains a reference to an object that defines the properties for the scope (usually a `Component/Widget`). There is one root `Scope`, named *root* whose *delegate* is the root widget (`Protowidget.RootWidget`). This can be thought of as a global scope of sorts in that properties set on the `RootWidget` can be accessed from anywhere.

Most widgets simply inherit the scope of their parent, which will typically be the root scope. Any widget or component can introduce its own scope, but this is typically reserved for special cases, namely collections and iteration.

Since components/widgets can introduce a new scope or inherit their parent's, and since once a scope relationship has been established, it cannot be changed, it can be said that `Protowidget` implements the concept of *lexical scoping*. If you don't know what that means, don't worry. Put simply, it means that the scope structure follows the structure of the widget hierarchy, which itself follows the lexical structure of the defining HTML document.

The next section discusses *attributes* which are the primary mechanism for interacting with scopes.

³TODO: This is not presently how it works

2.3.3 Attributes

Protowidget Attributes are the primary mechanism for navigating from a source object to a target object. In this way, they can be thought of as a simple expression language of sorts. All interaction with attributes is done through the *Component.getAttribute* and *Component.setAttribute* methods. Both take an attribute specification and alternately return the associated value or set the receiver to a new value.

ATTRIBUTE SYNTAX: [scope::][prop1[.prop2[...]]]

In its simplest form, an attribute specification just resembles a normal JavaScript property reference with periods separating path components. When getting an attribute, encountering a null/undefined value in the property path will cause the *getAttribute* method to immediately return null. When setting an attribute, this situation will log a warning.

The slightly more complicated form involves prefixing the property path with a scope qualifier. In this form, the scope chain is searched upward until a scope with the given name is found. The first scope with the given name is used as the target of the attribute get/set.

There are a few “special” scope names:

- The empty scope name (denoted by “::”) always refers to the current scope.
- *root* refers to the root scope, as defined by *Protowidget.RootWidget*
- *this* refers to a pseudo-scope whose target is the object receiving the *getAttribute/setAttribute* call

In simple applications, the *scope::* prefix will not be needed. It is used heavily, however when working with repeaters (covered later) in which each repeating item assigned its own scope.

2.3.4 Property Listeners

One of the primary benefits of using properties on Components instead of raw field access is the ability for the framework to track and raise property change events. Capturing these events is done by attaching property listeners to a named property on an object instance. This is done by calling the *addPropertyListener(propName, fcn)* method, specifying the property name to monitor and a listener function. The listener function will be invoked whenever the property changes value. It will be passed the following parameters:

1. Reference to the object whose property changed
2. Property name
3. New Value
4. Old Value

Property listeners need to be explicitly removed through a call to *removePropertyListener(propName, fcn)*.

2.3.5 Bindings

All of the other features of the Component class principally serve to enable Bindings. A binding is a linkage between a source attribute and a target property such that when the source attribute changes, the target property will be updated. A binding is established on the target end of the relationship by calling the *bindProperty(name, spec)* method. The *name* is the property name to bind and the *spec* is a binding specification (see below) which defines the binding. A property can be unbound by calling *unbindProperty(name)*.

The binding specification can either be a literal string value or a binding expression. A binding expression is one that follows the following syntax:

```
#{JavaScript expression}
```

The *JavaScript expression* can be any legal JavaScript expression. The expression may contain attribute references surrounded by backticks (`). These backtick attributes will be replaced with the actual value of the attribute as resolved by *getAttribute*. All components of the attribute will have property listeners attached to them such that the expression will be re-evaluated and applied whenever any

property in the attribute specification changes. Note that this does not just apply to the final property but works for all components of the attribute.

If a binding expression *only* contains an attribute specification and no other JavaScript expression, then the syntax can be shortened to just the attribute surrounded by backticks(``).

The following are valid binding specifications:

`#{'root::product.name'}` References the `product.name` property on the root widget.

`#{'position' * 50}` Evaluates to the `position` property on the outermost scope multiplied by 50.

`some value` A literal string

`#{3*12}` Evaluates to 36 and never updates

`'currentItem::firstName'` References the `firstName` property on the `currentItem` scope

The attribute syntax for binding expressions is extended to include flags that control the manner in which the binding takes place. These are primarily useful as optimization hints to disable various features for read-only attribute references. Currently, the only optimization involves prefixing the attribute specification with a tilde (~). If this is done, then all intermediate parts of the attribute will not be monitored for changes (unless if any of them evaluate to null/undefined, in which case, they will only be monitored for changes until the entire attribute path evaluates to !null).

2.4 Widgets

As the name implies, `Protowidget` exists to enable widgets. Up to this point, various foundational concepts have been presented that are necessary to fully understand how widgets operate. Widgets have the following characteristics:

- Are instances of the `Protowidget.Widget` class/subclass (extends `Protowidget.Component`)
- Is associated with an HTML DOM element (with a few rare, low-level exceptions)
- Has a parent widget (except the root widget)
- Inherits a scope from their parent or introduce a new one
- Is instantiated and configured by adding custom attributes to the corresponding DOM element

Most of `Protowidgets` actually extend `DomWidget` instead of `Widget`. `DomWidget` adds a number of features for HTML integration that are not needed or desired for some low-level uses. The remaining discussion about widgets will deal exclusively with subclasses of `DomWidget`.

2.4.1 HTML Integration

Widgets are defined and configured by annotating HTML elements with custom properties. At document load time (in response to the `onload` event), `Protowidget` will scan the loaded document for elements with `Protowidget` custom attributes and take appropriate action. In most applications, this scan only happens once; however methods exist for scanning additional parts of the document that have been modified or reloaded. In general, this is discouraged. Instead, there are various templating widgets that encapsulate this behavior and ensure that their contents are always in sync with `Protowidget`.

Due to the vararies of different HTML user-agents, `Protowidget` supports a number of different mechanisms for specifying its attributes. The debate rages over the "correctness" of annotating HTML documents with custom attributes in this manner. Opponents of this technique typically rely on elaborate CSS class names to specify their bidding. The argument goes that CSS class names can be used and the resultant document will still be valid, whereas if custom attributes are used, the document will not validate. The `Protowidget` philosophy on this front is made of the following points:

- Using visual CSS class names to control programatic behavior is more of a violation of the *spirit* of the various standards than is using custom attributes. Pragmatically, there is nothing wrong with either approach, but if someone wants to criticize the use of custom attributes, then they also need to evaluate their abuse of CSS classes.

- Custom attributes work in every browser and are highly convenient.
- Pure XML syntax may be used. In this way it is possible to construct a DTD such that the document validates. Protowidget leaves the actual construction of the DTD as an exercise to obsessive users.
- An alternative, HTML friendly attribute syntax is supported. This will never pass any kind of validation, but users are free to leverage it.
- The 'X' in XHTML stands for eXtensible. It is indeed extensible, but the designers of the spec did not give much thought to actual feasible ways to make extensions validate (and browser makers compounded the problem). Protowidget will not be held back by the inability to properly validate extensions.

In short, Protowidget takes a similar approach to validation as the browsers: It makes it possible to write valid documents but does not offer much help or incentive on its own to do so.

2.4.1.1 Custom Attributes

All Protowidget custom attributes are prefixed with either a 'pw.' or a 'pw:'. The prefix cannot be changed, and the user must make sure that proper namespaces are declared if targeting XHTML documents. The discussions below assume this prefix and do not explicitly state it.

Note that in general user-agents do not preserve the case of attributes when accessed through the DOM. For this reason, if an attribute needs to represent a mixed case name, it should be done by changing the attribute to lower-case and separating the capitalized parts with a dash. For example, if an attribute needed to be referenced to set the property "fooBar", the corresponding HTML attribute name would be "foo-bar".

The following table summarizes the custom attributes available to all widgets:

In addition to these standard attributes, any widget descended from DomWidget will also process the following:

- Unrecognized prefixed attributes are converted to camel-case and set on the widget with a call to `setProperty`
- Attributes of the form *pw.style.name* will be converted to a style reference. Any valid CSS style can be set in this way or bound as a property (for example constraining the width of one widget to some other numeric value). If the CSS style conveys a unit of size, it will always be converted to/from an integer with a unit of pixels.

The following properties exist for all DomWidget subclasses and can be set or bound with the appropriate HTML attribute:

- *visible* – When true, the `show()` method is called. When false, the `hide()` method.
- *instanceClass* – Controls the prefix used when registering Widget specific CSS styles. See CSS integration.

2.4.1.2 CSS Integration

All DomWidgets integrate with CSS by way of their *instanceClass* property. If not explicitly set, it defaults to the string formed by concatenating "pw_" with the full Widget class name with all periods converted to underscores. See the following examples:

- Button widget: Will have CSS class 'pw_Button'
- Navigator.SelectorWidget: Will have CSS class 'pw_Navigator_SelectorWidget'

In addition, widgets may call their *addInstanceCssClass* and *removeInstanceCssClass* in response to various events. For example, the Navigator.SelectorWidget calls `addInstanceCssClass('disabled')` if the disabled flag is set to true in the model. This results in the following CSS class being added to the element: *pw_Navigator_SelectorWidget_disabled*.

By leveraging CSS in this manner, it is very easy to attach visual characteristics to widget states.

Table 2.1: Custom Attributes

NAME	DESCRIPTION	EXAMPLE	PSEUDO CODE
type	Specifies the widget class that is to be created and bound to this element. The named class must exist within the Protowidget.Types object or one of its sub-objects.	<pre><a pw.type='Button'> </pre>	<pre>info=new WidgetInfo(element) w=new Protowidget.Types.Button(parent, info)</pre>
attach	Semi-colon delimited lists of attribute names in which to store an instance of this widget. The attribute is evaluated relative to the parent.	<pre><div pw.type='Text' pw.attach='myTextWidget'> </div></pre>	<pre>info=new WidgetInfo(element); w=new Protowidget.Types.Text(parent, info); parent.setAttribute('myTextWidget', w);</pre>
advice	Semi-colon delimited list of advice names to apply to the widget instance.	<pre><div pw.type='TextField' pw.advice='ExportRealTime'> </div></pre>	<pre>info=new WidgetInfo(element); w=new Protowidget.Types.TextField(parent, info); w.applyAdvice('ExportRealTime');</pre>
skip	Do not process this element or its children.	<pre><div pw.skip='true'>blah</div></pre>	
id	Sets the widget.id property and attaches to its parent scope with the given name.	<pre><div pw.type='Button' pw.id='myButton'> </div></pre>	<pre>info=new WidgetInfo(element); w=new Protowidget.Types.Button(parent, info); w.id='myButton'; parent.setAttribute('myButton', w);</pre>
scope	Introduces a new scope with this element. Rarely used except on repeaters in which the introduced scope name used is for each list item.	<pre><div pw.type='PrototypeDataListRepeater' pw.list='myList' pw.scope='item'> </div></pre>	<pre>info=new WidgetInfo(element); info.scope='item'; info.list='myList'; w=new Protowidget.Types.PrototypeDataListRepeater(parent, info); info=new WidgetInfo(span); info.text='item:name'; s=new Protowidget.Types.Text(w, info);</pre>

2.5 Models

2.6 Templates

2.7 Input Controls

2.8 Repeaters

2.9 Integration with HTML

2.10 Logging

Chapter 3

Core API Reference

TODO

Chapter 4

Widget Reference

TODO