So for the **binding type** “data-bind” there is currently three different types of target.

* Property
* Attribute
* Style

To specify a target type we use the following syntax: <targetType>-<target>, e.g.

* property-innerText:<expression>
* attribute-src:<expression>
* style-fontSize:<expression>

For each **binding type** there can be an optional default **target type**, for example, the default target type the data-bind binding type is property. When a default type is available the expression can be simplified, for example:

* innerText:<expression>

In addition to the binding type supporting a default target type, it may also specify a default target, which may be dependent on the tag being bound to. For example, the default **target** for **target type** property on a text input tag could be “value”

* <input data-bind=”firstName” />

Let’s consider the same sample the same example within the data-action binding type. There is currently only one type of supported action.

* action

Which in <targetType>-<target> form would equate to:

* action-click:<expression>

Because on is the default targetType for an action we can invoke from the binding

* click:<expression>

An because click is the default type for a button we can omit the click

* <button data-action=”doAction()” />

Actions differ to properties in that they may specify multiple targets, e.g.

* <button data-action=”on-[mousedown mouseup]:doAction()” />

# Things to explore

## What if actions were placed within the data-bind attribute as per other frameworks?

Based on the above examples we would need to type one of the following:

To bind fully to an action:

* <button data-bind=”on-click:doAction()” />

To bind to the default target of the action target type:

* <button data-bind=”on-:doAction()” />

This syntax here is actually pretty consistent, although more wordy, and IMO doesn’t describe intent quite as well.

## What if every type of binding could support multiple targets?

Based on the above examples for actions we could actually support binding to multiple properties with one expression for all types of bindings.

* <img data-bind=”property-[width,height]:size” />

Or

* <img data-bind=”[width,height]:size” />

This is pretty cool, and very consistent.

## What if a target has custom expression requirements?

Using the mechanisms described above we can have different types of expression for different binding type/target type combinations, but what if we wanted additional properties for a specific target.

The only use case I can think of at the moment at the moment is for binding to custom controls, for example we may have an auto complete box we are using, so we may define a custom target type **autoComplete**, with targets such as **searchText, selectedItem, etc.** This would lead to expressions such as.

* data-bind=”{autoComplete-searchText:searchText}”
* data-bind=”{autoComplete-selectedItem:selectedPerson}”

Interestingly there still doesn’t actually appear to be much of a need for custom options here, we are simply binding to a different target type, that has the ability to manage a set of custom target properties, and their corresponding capabilities, e.g. setting, getting and change notification.

Really custom requirements should always be possible with new target types, however, these target types may not be able to support tag based default properties as these are only available at bind time.

WE NEED TO CONSIDER ALLOWING NON QUALITIFED TARGET TYPES, E.G. IF WE CAN’T SPECIFY A SUB PROPERTY THEN MAKE THE - OPTIONAL

## How should we handle templates?

Again we have two options for binding templates, we can either use a new binding type, or we can use a new target type. Below are some examples of the various syntaxes:

* <div data-template=”id:’person’, url:’url’” ></div>
* <div data-template=”id:’person’, [url:’url](file:///D:\Work\Photon\Wobbly\Web\Source\Web\Photon.Web\'url)’, each:people, if:” ></div>

data-bind=”dom-if:expression”

data-bind=”dom-each:people”

# Efficient Binding

1. Must support elements with binding attributes being removed from the DOM during initialization (this action is performed by flow-if, flow-each) type bindings.
2. Should support switching the data context without walking the DOM (this should be easy)
3. Should be fast
4. Should support “prepare” and “commit” phases, this is useful for controls that need to read a set of values and apply atomically, e.g. the selector control.

The flow control bindings “if” and “each” both modify the DOM by removing their child elements and converting them to templates. Because the binding process uses selectors to evaluate which nodes can be bound we end up attempting to bind the removed elements.

There are several general approaches that can be taken to overcome issue:

1. Remove binding attributes from the remove nodes, or mark them in some way so they are ignored.
2. Perform a prepare operation on flow-if and flow-each bindings so they can remove their content before other bindings are evaluated, (this would actually require two passes on older browsers, which could lead to slower performance).
3. Walk tree’s depth first testing each node for the attributes we are interested in (slow)
4. Ignore bindings on nodes that are detached from the tree without a data context, this would require us to walk up the tree for nodes we already know are not valid.

Another issue with the current approach is that content is removed for flow bindings by depth. Consider the example below, when the “each” binding is encountered it will remove its content and store it as a template, it will then process the “if” content for each item. This will result in following steps being taken for each item.

1. Render item template (which contains an “if”)
2. Bind the “if”
3. Remove the content for the “if” and store as template
4. Apply the template for the “if” if the condition is met

e**a**ch

If

Clearly this is incredibly wasteful as we are rendering all of the content for the “if”, removing it and then re-applying it if its condition is met, and we are doing this for each item!!!

## A Better Approach to Flow Bindings

A better approach would be to pre-process the tree in reverse, from leaf nodes up to the root.

Each node takes a copy of its content and stores it as a template in a cache by id. The id of the template will then be associated with the node using an attribute (data-template-id).

Consider the following content:

<div data-flow="each:1">

<span>

<div data-flow="if:leaf1">

Leaf Content 1

</div>

</span>

<div data-flow="each:leaf2">

Leaf Content 2

</div>

</div>

After pre-processing out DOM would contain a single element:

<div data-flow="each:1" data-template-id="2"></div>

And the cache will contain the following items:

|  |  |
| --- | --- |
| Id | Content |
| 0 | Leaf Content 1 |
| 1 | Leaf Content 2 |
| 2 | <span>  <div data-flow="if:leaf1" data-template-id="0">  </div>  </span>  <div data-flow="each:leaf2" data-template-id="1">  </div> |

### When do we clear the cache?

So when do we clear items from the cache?

As can be seen from the example above each item in the cache is associated with a root, when the root is removed then the cache entries associated with the root can safely be removed.

Two operations should help us here,

**CreateTemplateFromContent**

Takes content and turns it into a cache entry

Scans the content for references to cache entries and marks those entries as being referenced by the current entry.

|  |  |  |
| --- | --- | --- |
| Id | Referenced By | Content |
| 0 | 2 | Leaf Content 1 |
| 1 | 2 | Leaf Content 2 |
| 2 | - | <span>  <div data-flow="if:leaf1" data-template-id="0">  </div>  </span>  <div data-flow="each:leaf2" data-template-id="1">  </div> |

**CleanNode**

Look for data-template-id attribute, if the attribute exists then lookup the template in the cache, if the template is not marked with a dependency then clean up its tree.

Example, removing anything with references to data-template-id 0 or 1 will have no effect on the cache.

One immediate optimization that can be made is this; if any kind of binding node is encountered whilst walking up the hierarchy we can take its data context, e.g. we don’t have to wait until we hit a data context node.