Positron Glossary

Action

An *action* is an operation which achieves something. This could be as simple as adding a class to an element, or as complex as posting information to a server and interpreting the result. As an MVC paradigm, an action most closely resembles a controller.

In Positron, *actions* are Javascript classes which are referenced from HTML markup. When a *trigger* fires, the *action* does its stuff. Any *trigger* can be associated with any *action*, and in general the *trigger* and corresponding *action* are not aware of each other's details.

When Positron encounters an *action* in markup, its name is looked up in its configuration, resulting in a class name. This is used to instantiate the appropriate class, and associate the instance with the corresponding *trigger* instance so they can talk to each other.

In the following example, "addclass" is the name of the *action* -

```
<div pos-action="(click) addclass: blue/.square">
  Click here to turn all the squares blue.
</div>
```

- which is then resolved to the class name "positron.action.AddClassAction" for actual execution.

Attribute

An *attribute* is a property of an HTML element, which adds some metadata information to the element. A simple example of this would be the "class" *attribute*, which describes which CSS classes pertain to an element.

Positron allows the developer to associate a Javascript class, called an *attributelet*, with an *attribute*, which is instantiated and invoked when first encountered. Positron itself uses the *attribute* mechanism to register its own attributes *pos-view* and *pos-action*.

When Positron encounters HTML, it looks each *attribute* name up in its configuration, resulting in a class name. The class is then instantiated and called to process its *attribute*.

In the following example, “pos-action” is the name of the *attribute* -

```
<div pos-action=”(click) addclass: blue/.square”>
  Click here to turn all the squares blue.
</div>
```

- which is then resolved to the class name “positron.attribute.ActionAttribute” for actual execution.

Context

Positron allows the developer to reference data entities directly in markup. However, for various reasons, these entities do not come directly from the Javascript variable space. In order for a piece of data to be accessible from markup, it must first be placed in a *context*. In programming terms, *context* is equivalent to *scope* - a space within which a certain piece of data can be accessed.

Generally, data has *tag scope*, which means that it is accessible from markup only inside the tag which introduced it. In the following example -

```
<pos-json url=”/session/user” prefix=”user>
The logged-in user’s name is $user.name;
</pos-json>
```

- the “pos-json” tag introduces the “user” variable into its *tag context*. Its value is valid only inside the “pos-json” tag.

Scoping data like this is helps manage access and prevents pollution of the Javascript variable space. It’s similar to how local variables work in native programming languages such as C++ and Java (ie, not Javascript).

Positron also allows the developer to shift the context of a piece of data to make it accessible to markup above its originating context. In the following example -

```
<pos-set key=”user” valuekey=”user” context=”page” />
```

- the “pos-set” tag introduces a reference to the data item referenced by the “user” key in the current context into *page context*, so that any piece of markup in the current page can reference it. In the following example -

```
<pos-json url="/session/user" prefix="user>
  <pos-set key="user" valuekey="user" context="page" />
</pos-json>
The logged-in user's name is $user.name;
```

- the reference to "user" is valid, as the "pos-set" tag promoted it to *page context*.

Controller

In the context of Positron, *Controller* is a chunk of markup with a name, declared by adding the "pos-controller" attribute to a normal element. It's very similar to *View* but is not showable, ie it has no visibility whatsoever, and Positron will mark Controllers as invisible if they are not so marked when first encountered. The markup within a Controller is not executed until it is invoked by the "runcontroller" action or manually using the JavaScript API.

Controller are useful for encapsulating markup-hosted code which has no visibility - eg, it could use <pos-sync> to copy elements to another visible view.

In the following example, the clockinterval View runs the clockproto Controller every second to update the rotation of the second hand in the clock View --

```
<div pos-view="clockinterval:">
  <div
    pos-action="(interval: 100ms) runcontroller: clockproto"></div>
</div>

<div pos-controller="clockproto:">
  <pos-date>
    <pos-sync view="clock">
      <pos-set
        name="seconddegree"
        expression="$date.seconds; / 60 * 360"
        >
        <img
          id="second-hand"
          src="images/second-hand.png"
          style="transform: rotate($seconddegree;deg)"></img>
      </pos-set>
    </pos-sync>
  </pos-date>
</div>
```

```
<!-- the elements in this view get updated via the sync tag above -->
<div pos-view="clock:" class="clock">
  <img id="second-hand" src="images/second-hand.png"></img>
</div>
```

See example7-clock for a more complete demonstration of Controller.

Event

In web development terms, an event is a message between two pieces of code which isolates the sender from the recipient. Events are commonly used to communicate user input, such as mouse activity or a change made to the value of a form field.

Positron allows the developer to associate actions with triggers, a common form of which is an event handler. When the event is caught, then the action is executed, and so on. Some actions, such as *showview*, have accompanying parameters, so that the target view can use some of the information contained within the event. However, to avoid making the view dependent on structure pertaining to a particular event type, Positron allows for a piece of code to map information from the event structure into the parameters heading for the target. Such a piece of code is called an *eventlet*.

Eventlets are not declared in markup. They are associated with event types behind the scenes. Common uses for events would be to copy object references from event structures into action parameters, for later consumption in view markup.

Tag

Tag is really shorthand for *element*, an HTML node with a name, and potentially attributes and children. Traditionally, *tags* are used to declare page structure and layout hints for the browser, such as <div> for layout blocks, or <input> for data acceptance from the user.

Positron allows the developer to associate a Javascript class, called a *taglet*, with a *tag*, which is then instantiated and invoked when first encountered. Positron itself uses this mechanism to register the *tags* comprising its own standard *taglet* library.

When Positron encounters HTML, it looks each *tag* name up in its configuration, resulting in a class name. The class is then instantiated and called to process its *tag*.

In the following example, “pos-json” is the name of the *tag* -

```
<pos-json url=”/session/user” prefix=”user>
The logged-in user’s name is $user.name;
</pos-json>
```

- which is then resolved to the class name “positron.tag.JSONTag” for actual execution.

Trigger

A *trigger* is a condition which results in the execution of a Positron *action*. This is commonly an event being caught, but can in theory be anything. One such example is the “interval” trigger, which causes its associated *action* to fire every so often.

Positron allows the developer to associate a Javascript class, called a *triggerlet*, with the name of a particular trigger, which is then instantiated and invoked when first encountered. The *triggerlet* then sets up the conditions for itself, and causes its *action* to fire when those conditions are met. Positron itself uses this mechanism to register the *triggerlets* comprising its own standard library.

One common use for *triggers* is to register for regular events and apply logic to allow compound event types such as *long-tap* or *double-click*.

In the following example, “click” is the name of the *trigger* -

```
<div pos-action=”(click) addclass: blue/.square”>
  Click here to turn all the squares blue.
</div>
```

- which is then resolved to the class name “positron.trigger.ClickTrigger” for actual execution.

Please see the Positron Reference for more details of specific actions, attributes, events, tags, and triggers, and the Developers Guide for details of how to implement them.

View/Selector Paradigm

In many actions and tags, Positron uses a convention known as the view/selector paradigm to identify target elements on which the operation takes place. This is similar in concept to a CSS selector, and indeed can incorporate a CSS selector. There are two fields in the selector, either of which is optional. The first field specifies the key of a view - if this is specified, then the view’s associated element becomes the root of any further selection. The second field specifies a CSS selector - if this is specified, then the selector is used to further select elements off the root element.

Examples --

“menu” - select the associated element of the “menu” view.

“menu/.square” - select anything with the class “square” within the associated element of the “menu” view

“.square” - select anything in the document with the class “square”