Positron Quick Start

Welcome to Positron

Positron simplifies development of web applications by extending HTML to allow behavioural elements to be scripted purely with markup.

Positron Quick Start

The best way to get up to speed on Positron is to see it in action. This guide is a quick tour covering the main functional areas of the framework and how they work.

Just a few ground rules before jumping in: it's assumed you're using a Macintosh and you have a reasonably modern default browser. Positron uses node.js to serve examples, it's also assumed your machine will happily run that.

Look in the examples folder in the distribution for a Terminal script called run-examples.command. This starts a little server to serve the examples, and directs your browser to its root page. Open it, and you should see a basic site displayed in your browser.

Instant Gratification: CSS Manipulation

In the root of the page displayed by run-examples, click on "Example 1: CSS".

Clicking on a link should do what its relevant caption says. If it doesn't, check the developer console for browser errors.

Open index.html in your editor, or inspect the page source using your browser (don't inspect

elements, you want the original source the browser loaded, not the current state). Note that apart from doing a little Positron initialization, there is no Javascript in the file.

So how does this little application work?

Take a look at Example 1 and notice the markup for the elements which are acting as links:

```
<div pos-action=”addclass: blue-bg/.square”>
  Click here to add the blue-bg class to the square.
</div>
```

Positron works by allowing markup authors to annotate code with extra attributes to determine behaviour. In this case, the pos-action attribute is specifying that upon a click on that element, the CSS class blue-bg should be added to all elements matching the .square selector. The other elements remove the class and toggle the class using the actions removeclass and toggleclass respectively.

Note that the last “link” in the example appears to do nothing, but it’s actually showing how to do multiple actions per click --

```
<div
  pos-action-1="addclass: blue-bg/.square”
  pos-action-2=”removeclass: blue-bg/.square"
  >
  Click here to add and remove blue-bg class.
</div>
```

pos-action allows much more than CSS class manipulation, as we’ll see. But for now, play with addclass, removeclass, and toggleclass.

Organisation 1: Views

This application effectively does the same thing as Example 1, except it uses a Positron View to encapsulate the markup and styles for the square. Also it allows showing and hiding of the square using more pos-action operations.

First take a look at the element holding the square --

```
<div pos-view="square" class="pos-invisible"></div>
```

The pos-view attribute declares that there is a Positron View associated with the element. When Positron encounters this, it attempts to find HTML, CSS, and Javascript resources for the view. In this case, square.html and square.css will be loaded from the html/views and css/views folders respectively. If we had needed any Javascript for this example, then we could have added square.js in js/views and that would have been loaded too.

Take a look at this element in the browser when the app is loaded and you'll see the markup from square.html included inside it.

However, pos-view goes further than loading resources. Adding pos-view to an element allows the page section contained by that element to be referred to by its View Key, which is normally the name of the View. Any Positron operations which operate on View Keys can then be applied to that page section.

For example, take a look at an element where a click affects the visibility of the view --

```
<div pos-action="showview: square">
  Click here to show the square.
</div>
```

Note the parameter to showview is the same as the value of pos-view in the element that declared the view.

At this point you may be forgiven for asking, *"why we didn't just use addclass and removeclass with a class containing "display: none;" instead of showing and hiding a view?"*

When you click to show and hide the square, notice that the square does not appear or disappear immediately - it fades in and out. This is because when you use showview, hideview, and toggleview, Positron manages the transitions between visible and invisible states, automatically applying animations. Fading is the default animation; you can override the default animation easily by either authoring a CSS class with more specificity or specifying a CSS class as an extra parameter in pos-action. See the later section on animations for more details. For now, play with declaring, showing, and hiding views.

Organisation 2: Pages

We've touched on Positron Views and how they mark a section of screen for special treatment. But authoring an entire application with just Views for organizational help would quickly get unmanageable. Enter Positron Pages. Take a look at Example 3 for a simple demonstration of Pages.

A Page embodies an application state; only one Page can be visible at once. When Positron is asked to make a page active, it automatically hides the current page (if one is showing) with an associated animation and loads and displays the new one, again with an associated animation. In Example 3, you can see that the old page fades out and the new one fades in... all at the same time. Setting Pages is done with the setpage action --

```
<div pos-action="setpage: blue">Show Blue Square Page</div>
```

Note that despite the fact that Pages are exclusive, this doesn't mean that only a Page can be shown. Pages live in a special element called the Page Container which is identified by the

“pos-page-container” ID. Positron ensures that only one Page inside the container is visible at any given time. However, outside the Page container, anything goes. The area outside of the Page container is usually used to display menus or anything else that exists outside Page space.

Organizing an application into Pages and Views is one of the initial tasks facing a Positron developer when starting a new project.

Real Power: Dynamics

Fear not: Positron offers facilities for combining server data feeds with HTML templates. In grand Positron tradition, everything can be done with markup, no Javascript required. Take a look at Example 4.

In Positron applications, all markup is dynamic, which is to say that entities can be placed within it to reference variables, perform logical decisions, include external data elements (such as JSON feeds), or do just about anything else...

Take a look at a section of the template markup for the movies View --

```
<pos-json url="feeds/movies.json" name="movies">
  <div>Received $movies.meta.count; records in JSON feed</div>
  <pos-list key="movies.data" name="movie">
    <div>movie title - $movie.title;</div>
  </pos-list>
</pos-json>
```

Whoa! What’s going on with those funky tags and dollar signs and stuff?

Within template markup, Positron uses custom tags to perform dynamic operations that have no equivalent in regular HTML. These custom tags do their job and then disappear; you won’t see them in final view markup. The dollar sign and semicolon syntax denotes a reference to a variable which is then substituted out when its value is known.

The <pos-json> tag above grabs a JSON feed from a URL and makes the corresponding Javascript object available to its children as a variable by the name given in the “name” attribute. Note that variable names can traverse Javascript object structures using the same syntax as Javascript - “one.two” refers to the property “two” in the object “one”, and so on.

The <pos-list> tag looks in context for a Javascript array object and provides a copy of its children for each element in the array. Each element is then made available in context for the appropriate child and receives a name for the “name” attribute or simply “list” if no prefix attribute is specified.

The easiest way of understanding how Positron’s dynamics engine works is to play with it, rather than read about it. One good exercise would be to add another property to each record in the movies.json file and then modify the template to display it instead of (or in

addition to) the movie title.

Animations

Positron knows that when your Views and Pages appear and disappear, they need to do so with accompanying animations. As we've seen, Positron supplies default fade in and fade out animations so that the user receives feedback. However, if you want to supply your own animations, or override them on a case-by-case basis, you can do that too.

Example 5 shows how to add custom animations to a page entry and exit sequence.

You might want a page to enter or exit from a particular side depending on how the app gets there or where it's going next. The setpage action operates in conjunction with two other optional attributes specifying the animation to apply to the incoming and outgoing pages --

```
<div
  pos-action="setpage: blue/slide-in-from-left/slide-out-to-right"
  >
  Show Blue Square Page
</div>
```

The extra attributes, where present, override the default animations specified by CSS. Note that if you want to override an animation permanently for a Page or View, you can provide a CSS class with more specificity, that will bind tighter.

It's also possible to override the animation that is used when views appear and disappear, by adding extra elements to the showview, hideview, and toggleview pos-action operations --

```
<div pos-action="showview: menu/fade-in">
<div pos-action="hideview: menu/fade-out">
<div pos-action="toggleview: menu/fade-in/fade-out">
```

Extensibility

Positron ships with a collection of handy widgets to do fun stuff, but of course it's not possible to anticipate everything an application might need. Hence, the framework is completely open and extensible. If don't want pos- as your tag prefix, you can change it. If you want to add extra triggers, actions, attributes, or tags, simply implement them, add them to the config, and then reference them from markup.

Example 6 contains custom trigger, action, attribute, and tag code as a demonstration of Positron's extensibility.

Conclusion

This quick start guide has hopefully presented a clear demonstration of and relatively shallow

learning curve to Positron’s core functionality. The next steps are to play with the examples, break and fix code, and get familiar with the facilities before moving on to the Developer Guide.