Neutrino Quick Start

180 Pacific Avenue San Francisco, CA 94111
info@subatomicsystems.com

Contents

Introduction

Welcome to Neutrino

Neutrino is a framework that simplifies the process of developing HTML5 Web apps while enabling creative developers to separate front-end application prototyping and design from JavaScript and back-end engineering. With Neutrino, creative developers are able to quickly prototype, produce, and deploy engaging interactive experiences using the same code base and development process regardless of the delivery platform. Whether you are deploying as hybrid apps for iOS and Android or as stand-alone Web apps and embedded mini-sites, the development work flow stays the same.

For support questions, please contact support@subatomicsystems.com.

Neutrino Quick Start

Neutrino Quick Start

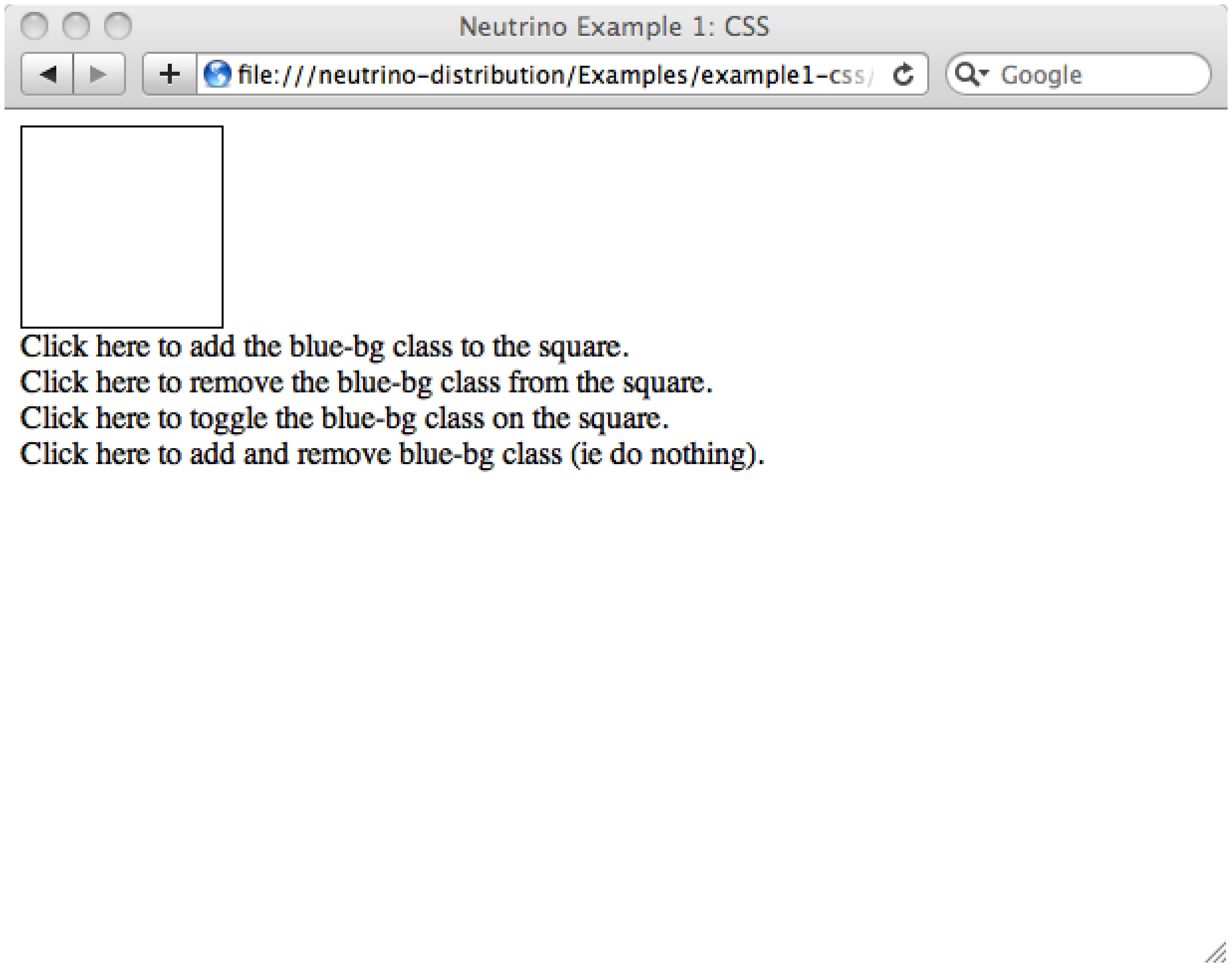
The best way to get up to speed on Neutrino is to see it in action. This guide is a quick tour of Neutrino covering the main functional areas of the framework and how they work.

Taking Neutrino for a Ride

Instant Gratification: CSS Manipulation

Look in the examples folder in the distribution and find a folder called **example1-css**. In that folder, find index.html and open it in Safari.

You should see something like this -



Click here to add the blue-bg class to the square.
Click here to remove the blue-bg class from the square.
Click here to toggle the blue-bg class on the square.
Click here to add and remove blue-bg class (ie do nothing).

Neutrino Example 1: CSS

Clicking on a link should do what its relevant caption says.

Open **index.html** in your editor, or inspect the page source using your browser. Note that apart from doing a little Neutrino 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  >
  Click here to add the blue-bg class to the square.
</div>
```

Neutrino works by allowing markup authors to annotate code with extra attributes to determine behavior. In this case, the **nu-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.

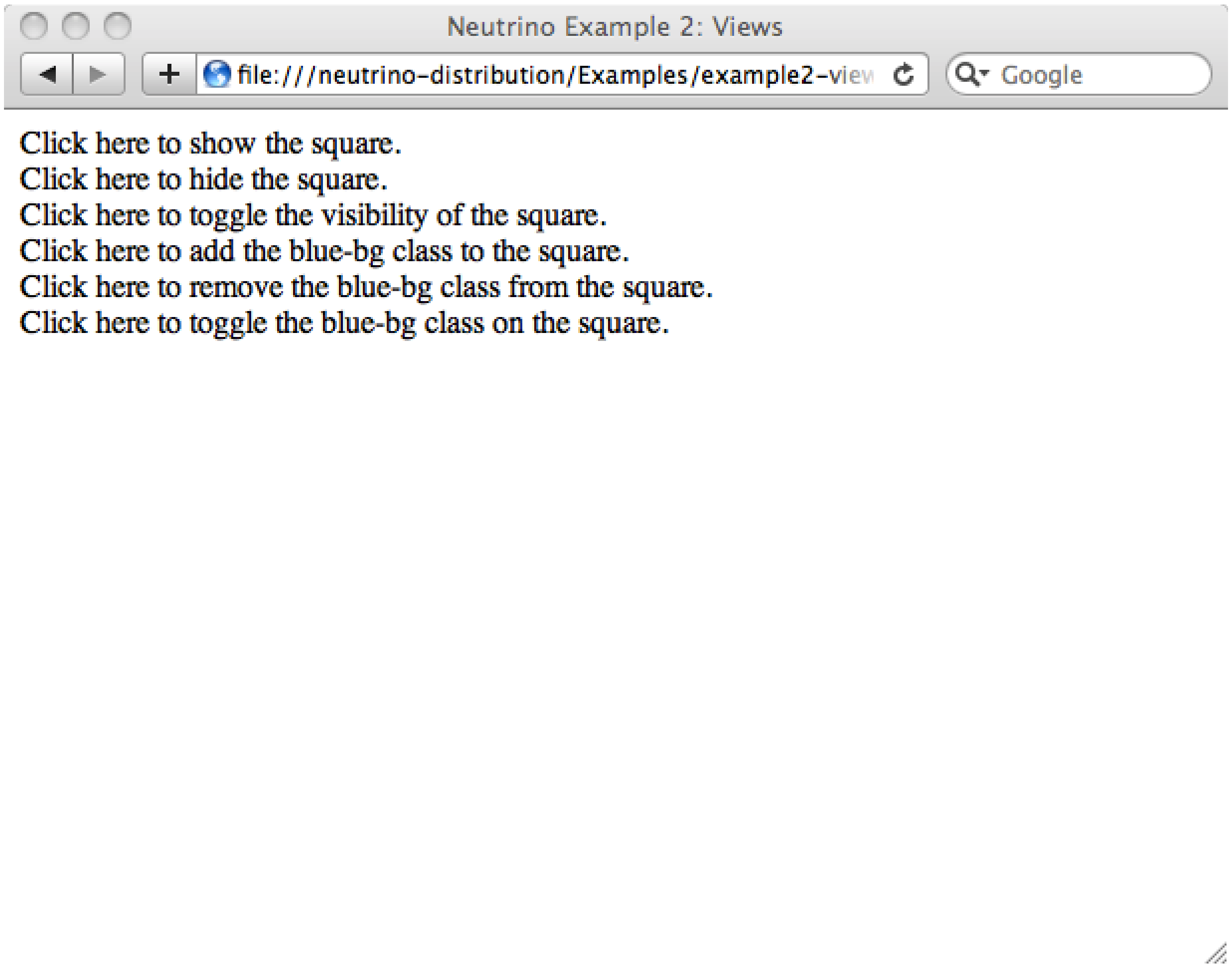
Note that the last “link” in the example appears to do nothing, but it’s actually showing how to do multiple actions per click --

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

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

Organization 1: Views

Locate **example2-views** in the distribution and open the **index.html** with Safari. You should see something like this --


Neutrino Example 2: Views
file:///neutrino-distribution/Examples/example2-view
Google

Click here to show the square.
Click here to hide the square.
Click here to toggle the visibility of the square.
Click here to add the blue-bg class to the square.
Click here to remove the blue-bg class from the square.
Click here to toggle the blue-bg class on the square.

Neutrino Example 2: Views

Clicking on a link should do what the caption says.

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

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

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

The nu-view attribute declares that there is a Neutrino **View** associated with the element. When Neutrino 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, **nu-view** goes further than loading resources. Adding **nu-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 Neutrino 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 nu-action="showview: square">
      Click here to show the square.
</div>
```

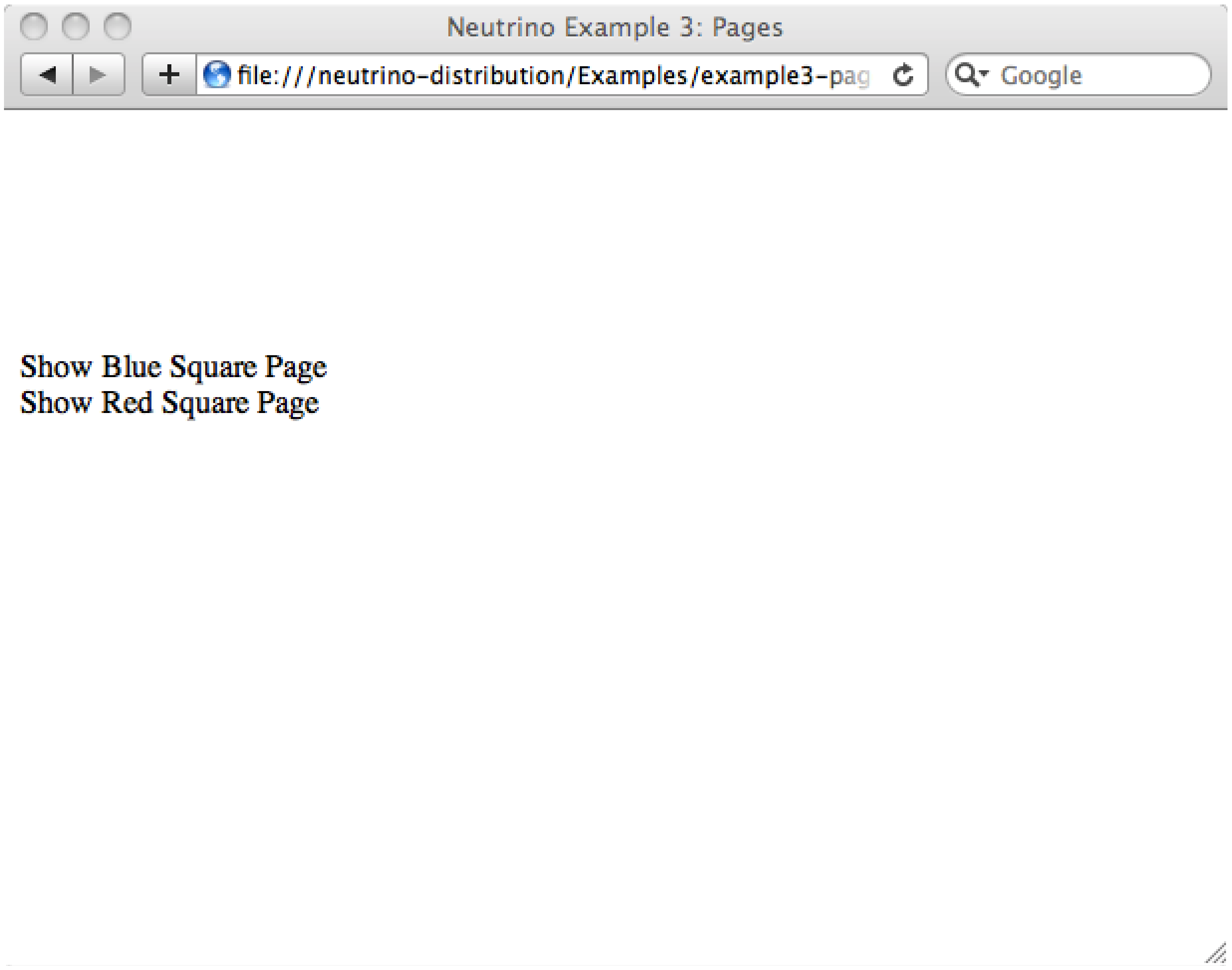
Note the parameter to **showview** is the same as the value of **nu-view** in the element that declared the view.

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**, Neutrino 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 **nu-action**. See the later section on animations for more details. For now, play with declaring, showing, and hiding views.

Organization 2: Pages

We've touched on Neutrino **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 Neutrino **Pages**. Take a look at **Example 3** for a simple demonstration of **Pages**.


Neutrino Example 3: Pages
file:///neutrino-distribution/Examples/example3-pag
Google

Show Blue Square Page
Show Red Square Page

A **Page** embodies an application state; only one Page can be visible at once. When Neutrino is asked to make a page active, it automatically hides the current page (if one exists) 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 **nu-link** attribute --

```
<div nu-link="blue">Show Blue Square Page</div>
```

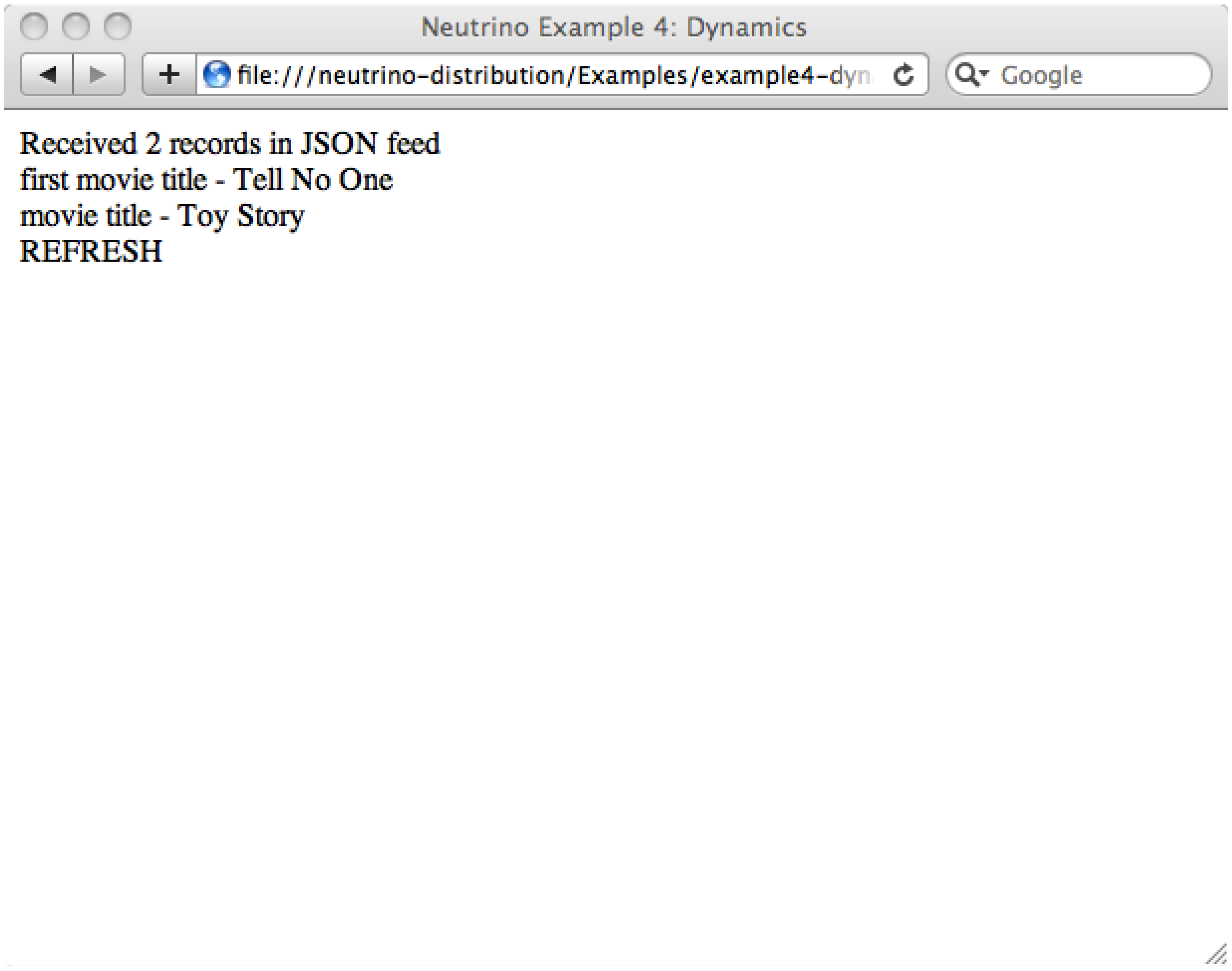
Note that despite the fact that Pages are exclusive, this doesn't mean that only a Page can be shown. Pages live in a special element called the Page

Container which is identified by the **nu-page-container** ID. Neutrino 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 Neutrino developer when starting a new project.

Real Power: Dynamics

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


Neutrino Example 4: Dynamics
file:///neutrino-distribution/Examples/example4-dyn
Google

```
Received 2 records in JSON feed
first movie title - Tell No One
movie title - Toy Story
REFRESH
```

Neutrino Example 4: Dynamics

Note the **nu-dynamic** attribute on the movies View --

```
<div nu-view="movies:" nu-dynamic="true"></div>
```

This says that the view is dynamic, as in, its contents may change each time it is refreshed. This element therefore contains template markup which is combined with dynamic environmental entities to produce final markup for the View. Note that the template markup is copied away and hidden until it is time to transform it into final markup, so you may find that your View has no markup when viewed in your browser's inspector.

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

```
<nu:json url="feeds/movies.json" prefix="movies">
    <div>Received $movies.meta.entryCount; records in JSON feed</
div>
    <nu:list key="movies.data" prefix="movie">
      <div>movie title - $movie.title;</div>
    </nu:list>
</nu:json>
```

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

```
<nu:json url="feeds/movies.json" prefix="movies">
    <div>Received $movies.meta.entryCount; records in JSON feed</
div>
    <nu:list key="movies.data" prefix="movie">
      <div>movie title - $movie.title;</div>
    </nu:list>
</nu:json>
```

Within template markup, Neutrino 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.

```
<nu:json url="..." prefix="..." jsonp="true|false">
```

The 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 **prefix** attribute. If no prefix attribute is specified, then the name is simply **json**. 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.

```
<nu:list key="..." prefix="...">
```

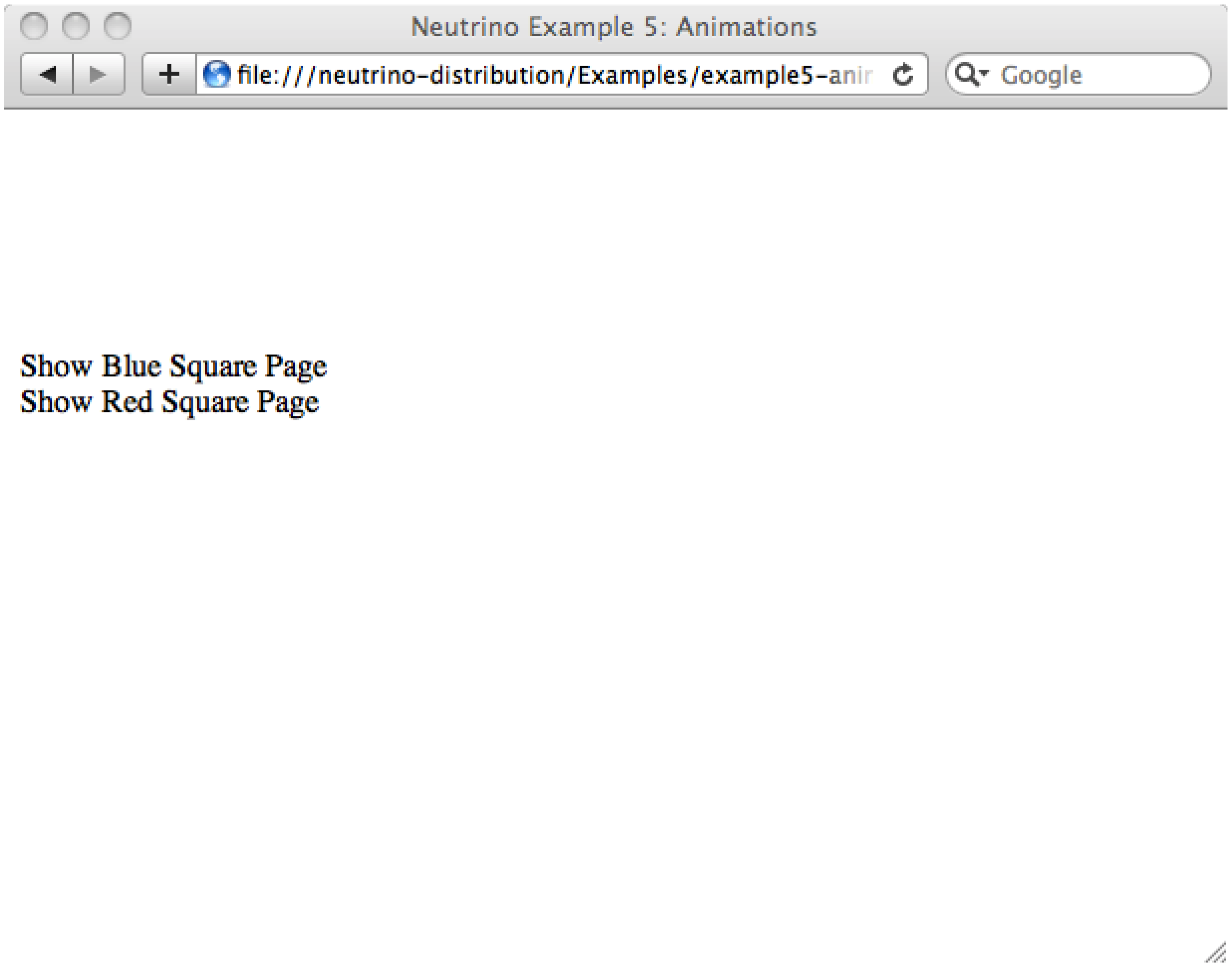
The tag above looks in context for a Javascript array object and provides a copy of its children for each element in the array. Each element is then made available in context for the appropriate child and receives a name for the prefix attribute or simply “list” if no prefix attribute is specified.

The easiest way of understanding how Neutrino’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 JSON file and then modify the template to display it instead of (or in addition to) the movie title.

Example 5: Animations

Neutrino knows that when your **Views** and **Pages** appear and disappear, they need to do so with accompanying animations. As we’ve seen, Neutrino 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.


Neutrino Example 5: Animations
file:///neutrino-distribution/Examples/example5-anir
Google

Show Blue Square Page
Show Red Square Page

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 **nu-link** attribute operates in conjunction with two other optional attributes specifying the animation to apply to the incoming and outgoing pages --

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

The **nu-page-transition-in** and **nu-page-transition-out** 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, perhaps scoped to an ID on the page's element, 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 nu-action** operations --

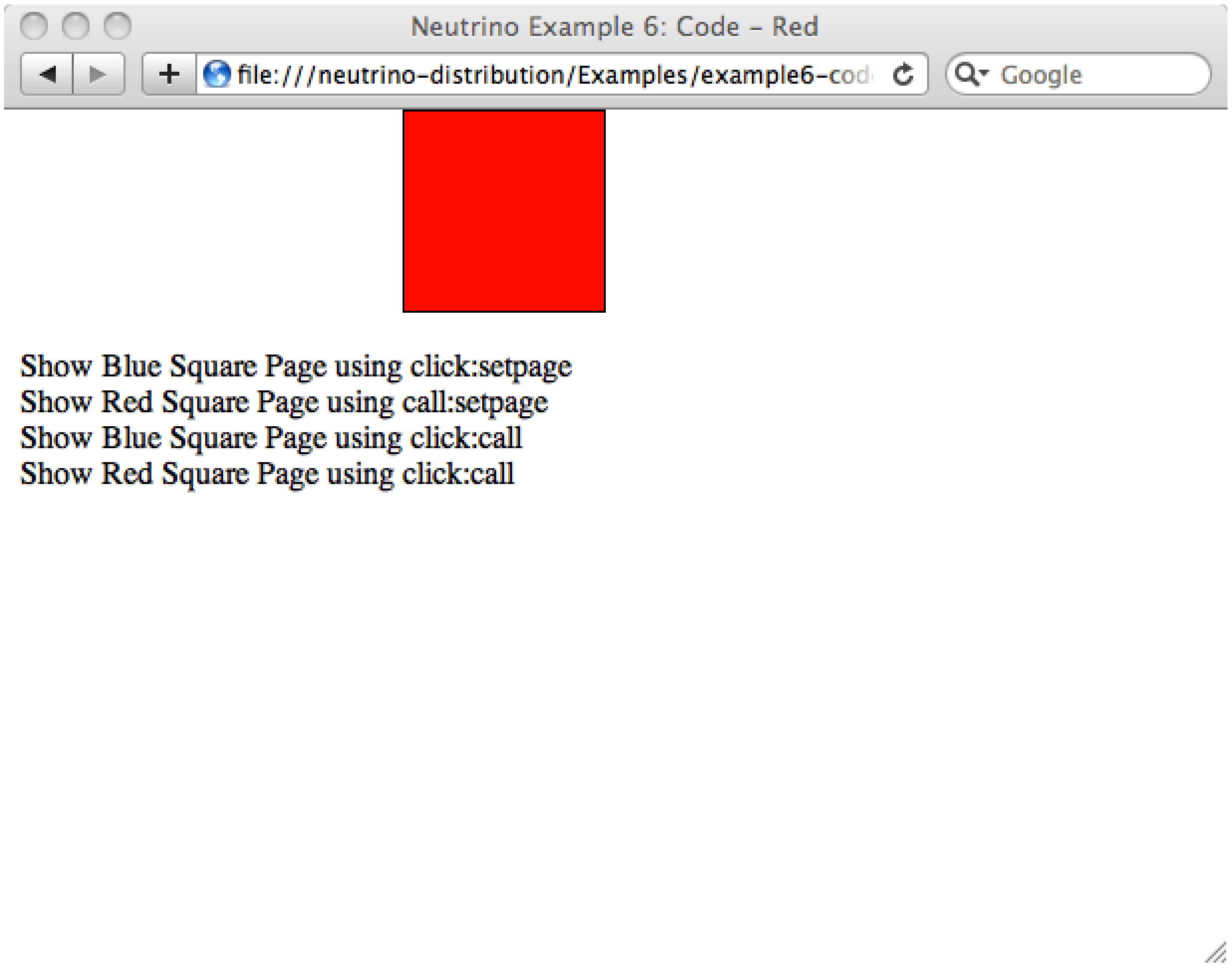
```
<div nu-action="showview: menu/fade-in">
<div nu-action="hideview: menu/fade-out">
<div nu-action="toggleview: menu/fade-in/fade-out">
```

Example 6: Code

Oh no! I thought you said there would be no code?!?

Sometimes there are situations that call for additional code. A likely scenario: submitting user input to a server, then deciding what to do with the information returned. Neutrino makes it possible for HTML/CSS authors to script common view operations without resorting to Javascript and provides facilities whereby code can be called at certain times.

Take a look at **Example 6**.


Neutrino Example 6: Code – Red
file:///neutrino-distribution/Examples/example6-cod
Google

Show Blue Square Page using click:setpage
Show Red Square Page using call:setpage
Show Blue Square Page using click:call
Show Red Square Page using click:call

Note the inclusion of Javascript files in **js/views** and **js/pages**. These files contain subclasses of Neutrino's View and Page classes which override various methods to indicate when they are called. These are so-called View lifecycle callbacks which are called automatically by Neutrino when Views and Pages move between states.

For example, when a View or Page becomes visible, the `onVisible()` method is called in the associated Javascript class.

To get called when the user clicks on an element, add a **nu-action** attribute whose value is "`call:`" followed the name of the method to be called --

```
<div nu-action="call: onElementClicked">CLICK HERE</div>
```

Note that Neutrino will search not only the enclosing View (if any), but also the enclosing Page, Window, and (eventually) the Application for that method. Once a match is found, the search stops.

Conclusion

This quick start guide has hopefully presented a clear demonstration of and relatively shallow learning curve to Neutrino’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 Client Developer Guide and Reference Manual.