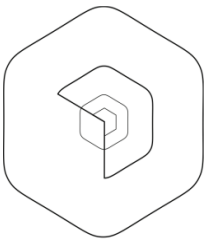
Neutrino Client Developer Guide



SUBATOMIC
SYSTEMS INCORPORATED

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

Contents

Neutrino Client Developer Guide

Welcome to Neutrino

Neutrino is a framework that simplifies the process of developing HTML5 Web apps. Neutrino enables creative developers to decouple 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 as hybrid apps for iOS and Android and as stand-alone Web apps and embedded mini-sites, all using the same code base and development process.

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.

Why Neutrino?

Web application development has moved beyond HTML and Flash. The capabilities of browsers have ushered in an age where everything but the model—and sometimes even part of that—is hosted by the client rather than the server. The result is highly dynamic, sophisticated applications free from the complexity of too much server code.

But this new client-centric model raises its own challenges. HTML and CSS are static. You can't modify or generate HTML—even, for example, to add a CSS class to an element—without writing JavaScript. So the process of developing compelling applications has come to require authors with two distinct skill sets: HTML and CSS programming, on the one hand, and JavaScript programming, on the other. This dependency on multiple skills creates bottlenecks during the development process. It also can lead to higher development costs, since two types of developers are needed to produce a single application.

Neutrino reduces this dependency by minimizing the need for manually written JavaScript in Web applications. Neutrino adds directives to HTML that perform common view-oriented operations. CSS classes can be manipulated, sections of screen can be shown and hidden, and JSON feeds can be

combined with templates, among other things. Neutrino greatly lessens the dependencies between logic and layout, resulting in expedited application development.

Neutrino's additions to HTML are not recognized by any browser. The framework must examine every section of markup and register the appropriate event handlers in order that the magic happens. Therefore markup is not "generated" in what has become a conventional way. In fact, markup is not generated by JavaScript at all. Neutrino keeps all the markup where it belongs - in page source.

How Neutrino Works

Neutrino's internals bear more of a resemblance to established native development frameworks than other JavaScript frameworks. Here's an introduction to Neutrino internals:

Application

Application is the only class that is directly visible in every Neutrino application, as it needs to be instantiated and invoked to get the ball rolling. It holds global state, manages pages, and provides services for other framework clients. In addition, it holds the root dynamics context from which all other contexts ultimately derive.

Window

The Window is a Page associated with the body element. Its function is largely to serve as a default Page for applications without Pages, and as a parent Page for Views outside the page container.

Page

A Page is a combination of HTML, CSS, and JavaScript essentially representing a bookmarkable application state. Neutrino ensures that only one Page is on the screen at any given time, though Views can exist outside the Page container. Page manages a list of child Views.

View

A View is a combination of HTML, CSS, and JavaScript representing a section of screen, under the management of a Page, or, in the absence of Pages, the Window. The association of a View with an HTML element allows Neutrino to manage its visibility, complete with animations, and also allows custom behavior based on visibility lifecycle callbacks or event handlers.

Neutrino Basics

This section covers how to start a Neutrino application from scratch. It gives you an understanding of the default file structure, the standard classes, and the tags needed to build a Neutrino app.

Application Directory Structure

application
Root directory for your application.

application/server
Root directory for web application resources.

application/server/web
Root directory for the web hosted section of the application. This is the location of index.html and other directly accessible resources. Also includes WEB-INF which contains any server application resources.

application/server/web/neutrino
Neutrino resources. Currently just contains directories further subdividing the resource space.

application/server/web/neutrino/assets
Neutrino content assets. Currently contains just online.gif, a tiny image the successful retrieval of which is assumed to indicate connectivity.

application/server/web/neutrino/js
Neutrino JavaScript resources. Currently contains just neutrino.js.

application/server/web/neutrino/css
Neutrino CSS resources. Currently contains just neutrino.css.

application/server/web/css/pages
Application specific Page CSS files.

application/server/web/css/views
Application specific View CSS files.

application/server/web/html/pages
Application specific Page HTML files.

application/server/web/html/views
Application specific View HTML files.

application/server/web/js/pages
Application specific Page JavaScript files.

application/server/web/js/Taglets

Application specific Taglet JavaScript files. Note this is a recommended convention only; Neutrino does not require Taglet JavaScript to be located here.

application/server/web/js/views
Application specific View JavaScript files.

Required Includes

Neutrino requires the addition of its combined CSS and JavaScript files to operate. Copy neutrino/client/css/neutrino.css and neutrino/client/js/neutrino.js into your application's "neutrino" directory and arrange the includes in your index.html. For example:

```
<link
  rel="stylesheet"
  type="text/css"
  nu-href=”neutrino/neutrino.css”
  class=”nu-browser-neutral-css”
  >
</link>
```

Note the nu-href attribute in place of the href attribute one might expect. Referencing the stylesheet with nu-href indicates to Neutrino that the sheet contains browser neutral CSS which should be rewritten on loading to be CSS specific to the requesting browser. See later for more information on browser neutral CSS.

```
<script
  type=”text/javascript”
  src=”neutrino/neutrino.js”
  >
</script>
```

Application Startup

Upon application startup, Neutrino needs to initialize itself and to examine the DOM to discover whether the initial markup includes any Neutrino directives. To start up Neutrino, include this code in your application.

```
<script type=”text/javascript”>

// make the application instance global
var gApplication = null;

function
main ()
{
  gApplication = new neutrino.Application ();
  gApplication.start ();
}

document.addEventListener ("DOMContentLoaded", main, false);
```

```
</script>
```

Views

A Neutrino View is a JavaScript class that is associated with a markup element. Associating a View with an element hands the management of that section over to Neutrino. This means that the section may be referred to via its "view key" in Neutrino API calls for managing its visibility (among other things), that it may potentially perform some custom tasks at various points during its visibility lifecycle, and that it may potentially be called upon to perform tasks when the user interacts with its markup.

Associating a View with an HTML Element

Associating a View with an element is accomplished by adding a custom attribute to the element, as follows:

```
<div nu-view="menu"></div>
```

When Neutrino examines this section of markup, it does the following things:

1. It attempts to load JavaScript from js/views/menu.js.
2. It attempts to instantiate a JavaScript class called MenuView, defaulting to the Neutrino View class if MenuView isn't found.
3. It attempts to load CSS from css/views/menu.css/.
4. It associates the View class with its element.
5. If the element has no markup within it, Neutrino attempts to load HTML from html/views/menu.html, inserting any markup found inside this element.
6. It calls the View to let it know that it has been loaded.
7. The View's parent Page registers some event handlers so that its visibility animations can be managed
8. Neutrino adds the View to its parent Page's map of child Views, indexed by its view key.

The nu-view attribute can have an additional field, delimited by a colon, which determines which resources are loaded during this process. If a View is known not to have any JavaScript, for example, then letting Neutrino know about that prevents a wasted HTTP transaction. The flags are:

`h` load HTML
`c` load CSS
`j` load JavaScript

If the colon is present in the nu-view attribute, then its contents are honored. An empty flags field means "do not load any additional resources". Usually these "load flags" are not included within the attribute until application deployment time, when the resource usage for each View becomes clear.

When declaring a View, you can override the default key which the Neutrino API will use to refer to the view and assign a different key, perhaps to give it a more meaningful name, or to avoid duplicate View keys in the case of lists, etc. To override the default key, add the nu-view-key attribute in the declaration of the view.

In the following example, we declare the menu view, instruct Neutrino to load HTML and CSS for this view but no JavaScript, and assign the view the key "topmenu", overriding the default key.

```
<div nu-view="menu:hc" nu-view-key="topmenu"></div>
```

From the time when the View is loaded, it will continue to receive visibility callbacks when its state changes. For example, if the View is visible when Neutrino loads it, then it will receive before-visible and visible callbacks.

The element associated with a View can be obtained via its "nuElement" property. An element's View, if any, can be retrieved by using neutrino.DOM.getData() with the key "view".

Showing and Hiding Views

Once a View is loaded, it may be referred to by its key in other operations. For example, the `nu-action` attribute allows views to be shown, hidden, and their visibility toggled (among other things). The following piece of markup toggles the visibility of the "topmenu" view declared above --

```
<div nu-action="toggleview: topmenu">
  Click here to toggle the top menu.
</div>
```

The advantage to showing showing and hiding Views like this, rather than just directly manipulating CSS classes, is that Neutrino manages animations and View visibility states automatically. Neutrino ships with standard animations which are used by default. Customizing these animations is simply a matter of providing CSS classes that describe specific customizations. Alternatively, override animation class names can be specified in the `nu-action` for more granularity:

```
<div nu-action="showview: menu/slide-left">Show Menu</div>
<div nu-action="hideview: menu/slide-right">Hide Menu</div>
```

Neutrino determines whether a View is visible according to whether it has the "invisible" class. This is the class that Neutrino adds when a View transitions to the invisible state, and removes when a View transitions from the invisible state. Neutrino uses a class rather than setting the CSS "display" property directly so that users can customize it for different environments.

Pages

Like a View, a Neutrino Page represents a section of screen and in implementation inherits from View. The major difference is that only one Page is visible at a time, and a Page manages its own list of child Views. While a View is just an independent section of a screen, a Page really represents a bookmarkable state of the application; it has a mission.

Like Views, Pages are usually dynamically loaded in response to requests from the application to set the active page. However they are not usually simply declared in markup. Page markup lives in fragments in html/pages, with the page element itself identified by the `nu-page` attribute. This attribute should be set to the Page key, which performs a similar function to a View key, but which is unique across the application.

The process for loading Page resources is very similar to that for loading Views, the only significant difference being the lack of load flags affecting resource selection.

Neutrino keeps Page markup in a special element, identified by the `nu-page-container` ID. Neutrino checks for this element at startup and creates it if it doesn't exist. If for any reason you want to load page markup at startup rather than have it dynamically loaded (as the compiler does, as described further on), you can just include the markup inside the page container element. Any markup is legal in there, as long as you identify your page element with the `nu-page` attribute.

Page transitions are conventionally achieved in markup using the `nu-action` attribute with the setpage action. A click on the following element would cause the current Page, if there is one, to be transitioned out with an appropriate animation, and the "detail" page to be transitioned in, again with an appropriate animation.

```
<div nu-action="setpage: detail">
  Click here to go to the detail page
</div>
```

The animations selected for this are the same as the ones which pertain to View transitions, which can in turn be overridden by CSS specificity. In addition, animations can be specified on a per-element basis by adding parameters after the page key, like so:

```
<div
  nu-action="setpage: detail/slide-left/slide-right"
  >
  Click here to go to the detail page.
</div>
```

Pages manage a map of Views, organized by View key. However, since Views can live at Application scope, outside the Page container, Application is called on to show and hide Views. The current Page's Views are searched first for a key match; if there is no match, then Application's Views are searched.

The element associated with a Page can be obtained via its "nuElement" property. An element's Page, if any, can be retrieved by using neutrino.DOM.getData() with the key "page". Note that a Page will have both "view" and "page" data elements, both of which point to the same JavaScript object, as Page inherits from View.

Events

By default, adding a "nu-action" element to an element causes Neutrino to attach desired actions to a particular event. For example, the following code fragment --

```
<div
  nu-action="(click) call: detail"
  nu-action-params="movie_id: 100; movie_title: Dune;"
  >
  click here for parameterised detail click
</div>
```

-- asks Neutrino to call the “detail” method in the enclosing View (or Page, or Application, etc), with the specified parameter set when a click happens on that . Note that “(click)” is the default and can be omitted in the click case.

Any event name can be used between the parentheses, giving Neutrino the power to add actions to any event. For example, here’s code that shows a view upon detecting a mouseover:

```
<div
  nu-action-1="(mouseover) showview: overview"
  nu-action-2="(mouseout) hideview: overview"
  >
  mouseover here to show the "overview" view
</div>

<div
  nu-view="overview" class="invisible"
  >
  this is the OVERVIEW
</div>
```

or calling a method to save an input field’s value when it changes:

```
<input
  nu-action="(change) call: saveName"
  name="name" type="text" required="true"
/>
```

Transitions

When Views and Pages are shown and hidden using nu-action or the corresponding API calls, Neutrino manages the transition between visibility states automatically using transition classes. These classes incorporate CSS3 properties which trigger animations when added, then when the animations complete, Neutrino updates the View’s state. Note that when using transitions, rather than abrupt visibility state changes, the transition classes *must* include CSS3 animation properties.

Example transition class --

```
/* note that this example contains browser neutral CSS */

.nu-transition-visible
{
    animation-name: nu-fade-in;
    animation-duration: 400ms;
    animation-timing-function: linear;
}

@keyframes nu-fade-in
{
    0%
    {
       opacity: 0;
```

```
    }

    100%
    {
       opacity: 1;
    }
}
```

By default, Neutrino uses the “nu-transition-visible” and “nu-transition-invisible” classes as transitory states to visibility and invisibility respectively. These classes simply fade in and out as appropriate. However, custom transitions can be used either by authoring classes with more specificity so that they override the default ones, or by specifying different class names in the nu-action or API call.

Dynamics

Neutrino incorporates a full dynamics engine which allows the resolution of template HTML with data into final markup, with no JavaScript code necessary.

Dynamics can be added to any View simply by adding the nu-dynamic="true" attribute to the View's element. When Neutrino encounters that element, it copies the template markup within the element into the View's instance, then deletes the markup from the DOM. When it comes time to render the View, the template markup is copied back to the element and the dynamics engine then processes it.

NOTE: Only Views with the nu-dynamic attribute set to "true" will have their contents dynamically refreshed. Views inside dynamic Views will only be refreshed when their parent dynamic View is refreshed.
Conventionally, dynamics are implemented with JavaScript either constructing HTML on the fly or managing HTML fragments—a development approach that exacerbates dependencies between skill sets. Neutrino aims to reduce such dependencies by implementing dynamics entirely in markup, so that HTML authors can proceed unfettered by the need to continually make requests of their JavaScript colleagues.

Inside the template element, Neutrino offers two facilities for managing dynamics: Taglets and variables.

Taglets

Neutrino provides special HTML tags which are associated with pieces of JavaScript called “Taglets” which are invoked when the dynamics engine processes markup. These Taglets then implement various kinds of dynamic behavior, which may result in changes to the document. For example, an “if” Taglet would include its children if its conditions were met and omit them if not:

```
<nu:if lhs=”one” rhs=”one”>
this would be in the final markup, as one = one
</nu:if>

<nu:if lhs=”one” rhs=”two”>
this would not be in the final markup, as one != two
</nu:if>
```

Taglets, then, provide an interface to dynamic behavior which is accessible from markup and familiar to HTML authors.

Taglets opaquely manage sections of markup and introduce variables which their children can then reference. However, Taglets do not and should not generate or modify markup. To do so would be to reintroduce the kinds of skill set dependency that Neutrino is designed to reduce.

Of course, you can write your own Taglets and have them perform application-specific operations on your markup. The only requirements are loading JavaScript for the Taglets and implementing *getTagletConfiguration()* in your Application subclass. See later section on Custom Taglets for more details.

In most cases, a Taglet removes its tag from the document once the dynamics processing on it and its children is complete, so you will not see these tags in the browser. But it is perfectly legal to associate Taglets with normal HTML tags, and in those cases the tag might remain in the tree.

Variables

Neutrino's dynamics engine manages the equivalent of a variable stack for use by markup called "context". As new contexts are made, they inherit from the current context, so that variables accessible to the Taglet's parents are also accessible inside the Taglet's tag. The initial context set for a dynamics run includes the View's Page, which inherits from the root context held by Application.

Variables are accessed in markup using this syntax:

```
$variablename;
```

Taglets routinely create contexts that inherit from the existing context, and introduce values into these new contexts for their children to reference or display. Conventionally then, these values are *only* valid for their children, as the new context is simply not accessible from, say, peers of the Taglet which introduced it.

For example, <nu:numberformat> formats a number according to the provided parameters, and introduces the formatted number into context. The formatting can then be used in Taglet's subtree like so:

```
<nu:numberformat value="23.9999" type="fixed" digits="2">
  <!-- numberformat.value is only valid inside <nu:numberformat> -->
  $numberformat.value;
</nu:numberformat>
```

However, sometimes it is necessary to "promote" a variable, say to Page or Application context, so that other sections of markup can access it. The `<nu:set>` Taglet has the power to do this.

Conventionally, the values which Taglets introduce to context have a prefix, which helps identify who put the value there and also contextualizes the name so that it can have application significance. For example:

```
$movie.title;
```

The prefix defaults to the name of the tag (minus any namespacing) and can be overridden by including a "prefix" attribute. Use of prefixes is encouraged as they increase readability.

Note that context variables can be of any type. Usually, strings and numbers are the only types used directly in markup, as objects and functions converted to strings for display purposes are of virtually no use. However, dot syntax can be used to walk JavaScript objects, just like JavaScript does itself.

For example, if the following JavaScript object:

```
var movie = new Object ();
movie.title = "Saving Private Ryan";
```

were inserted into context as "movie", then the template markup $movie.title; would resolve, even though the actual string movie.title is not a directly valid variable name in context.

The ability of Neutrino context variables to walk JavaScript object trees comes in handy especially in one particular area of dynamics processing: JSON feeds. Neutrino's Taglets make integration with JSON as simple as including one tag in your markup. The <nu:json> tag fetches a JSON feed from a URL, parses it into a JavaScript object tree, and puts it into context so that you can then reference its contents using context variables.

For example, suppose the following JSON feed was available through the URL "feeds/dune.json".

```
{
  "title": "Dune",
  "director": "David Lynch",
  "year": 1984
}
```

This feed could be integrated into markup as follows:

```
<nu:json url="feeds/dune.json" prefix="dune">
  <!-- the feed is now in context under the name "dune" -->
  Dune's title is $dune.title;
  Dune's director is $dune.director;
  Dune was released in $dune.year;
</nu:json>
```

Once the dynamics processing is complete, the final markup would look like this (minus the comment):

```
Dune's title is Dune
Dune's director is David Lynch
Dune was released in 1984
```

Using context variable syntax to walk JavaScript trees is very powerful. However, there are limitations to this method. For example, it doesn't support iterating arrays that might be in context. Fortunately, there's the nu:list tag, which iterates through an array in context and provides a copy of its children for each iteration, with the corresponding element of the array placed in context for each.

Let's look at an example. Suppose the following JSON feed was available via the URL "feeds/movies.json".

```
[
  {
    "title": "Dune",
    "director": "David Lynch",
    "year": 1984
  },
```

```
  {
    "title": "Flash Gordon",
    "director": "Mike Hodges",
    "year": 1980
  }
]
```

We could use nu:list to iterate through the entries in the feed as follows:

```
<nu:json url="feeds/movies.json" prefix="movies">
  <nu:list key="movies" prefix="movie">
    <div>$movie.title; ($movie.year;), directed by $movie.director;.</div>
  </nu:list>
</nu:json>
```

This would produce the following final markup:

```
<div>Dune (1984), directed by David Lynch.</div>
<div>Flash Gordon (1980), directed by Mike Hodges.</div>
```

`<nu:list>`, then, picks up the context variable inserted by `<nu:json>`.

For a complete description of the stock Neutrino Taglet set, please refer to the Reference Manual section of this document.

Parameters

It's often necessary to communicate context along with an action. For example, consider a list of items fetched from a JSON feed, a click on any one of which opens a detail page for that item. With only nu-action, an application would have to examine the target of the click and derive some environmental information before being able to provide the detail page with context.

Fortunately, Neutrino enables markup authors to communicate information in conjunction with a nu-action. The nu-action-params (or appropriate numerically suffixed equivalent) attribute contains a list of encoded parameters to accompany the action. The encoding is simple name: value; pairs similar to CSS --

nu-action-params="key1: value1; key2: value2;"

Note that due to the particulars of this encoding and Neutrino's context variable syntax, a double semicolon is required when expanding context variables into action parameter attributes --

```
<div
  nu-action="showview: detail"
  nu-action-params="movie_id: $movie.id;; movie_title: $movie.title;;"
  >
  click here for parameterized showview
</div>
```

In all cases, Neutrino parses the parameters and calls *setParams()* on the target View (or Page) *before* calling the action method, setting the Page, or showing the View. By setting parameters first, Neutrino sets properties in the View's parameters bundle, which can then be accessed by JavaScript, and also puts

corresponding values into the View's context so that they are accessible to dynamic markup.

The following is an example of a parameterized list-detail page interface.

```
<!-- html/pages/list.html →

<nu:json
  url="feeds/movies.json"
  prefix="movies"
  >
  <nu:list
    key="movies"
    prefix="movie"
    >
    <div
      nu-action="movie"
      nu-action-params="movie_id: $movie.id;;"
      >
      $movie.title;
    </div>
  </nu:list>
</nu:json>

<!-- html/pages/detail.html →

<!--
  parameterise the movie query with the movie ID
  that came in from the nu-action-params in the list page
-->

<nu:json
  url="feeds/movie.json?id=$params.movie_id;;"
  prefix="movie"
  >
$movie.title; was directed by $movie.director; and released in $movie.year;
</nu:json>
```

Custom Taglets

It's sometimes the case that the stock Taglet set simply doesn't do what an application might require, or that implementing a feature with the stock Taglet set results in complex and ungainly markup. Fortunately, developing custom tags is straightforward.

Step 1: Create a new class which inherits from neutrino.janx.Taglet

The new Taglet will basically be the same as <nu:lowercase> but capitalise the first letter of its argument. We'll call it the Capitalise Taglet.

```
var CapitaliseTaglet = function ()
{
```

```
      // call our superclass's constructor
      neutrino.janx.Taglet.call (this);

      // set up our required attributes
      // we won't get called unless these are defined
      this.requiredAttributes = new Array (1);
      this.requiredAttributes [0] = "value";
};

// set up our inheritance from Taglet
neutrino.inherits (CapitaliseTaglet, neutrino.janx.Taglet);

// ok now override expand() from Taglet to do our stuff
CapitaliseTaglet.prototype.expand =
      function (inElement, inContext, inTreeWalker
{
      // get the value to capitalise
      var   value = inElement.getAttribute ("value");

      // arrange the capitalised version
      var   newValue = inValue.substring (0, 1).toUpperCase ()
            + inValue.substring (1).toLowerCase ();

      // make the new context which will be valid for our children
      var   newContext = new neutrino.janx.DelegateHashMap (inContext);

      // insert the capitalised value into the new context
      newContext.put (this.getPrefixDot (inElement) + "value", newValue);

      // walk our subtree
      inTreeWalker.walkChildren (inElement, newContext);

      // and take this tag out of the tree
      neutrino.DOM.replaceWithChildren (inElement, inElement);

      // return the tag which is to replace us
      // in this case, as in most cases, we are gone
      return null;
}
```

Hopefully this should be straightforward. The new context is required in order that the capitalised value is only valid for the children of this tag. If promotion to other contexts is required, then <nu:set> can be used *inside* this tag, as always. Note also that the call to walk the subtree happens before any tree manipulation; this is required for asynchronous event notification to work properly.

Step 2: Include the JavaScript file in index.html

Taglets are not loaded the same way as Pages and Views, therefore they must be statically loaded like regular JavaScript resources. However, the Compiler will include Taglet JavaScript provided it is located in js/Taglets, so in order to avoid a double inclusion, mark the tag with the "nu-compiler-remove" tag.

```
<script type="text/javascript" src="js/Taglets/capitaliseTaglet.js"
class="nu-compiler-remove"></script>
```

Step 3: Register the Taglet class with the dynamics engine

Neutrino's dynamics engine requires configuration to associate tag names with Taglet instances. To do this, override getTagletConfiguration() in the Application subclass, and return an object which a key value pair --

```
myApplication.prototype.getTagletConfiguration = function ()
{
      var   taglets = new Object ();
      taglets ["myapp:capitalise"] = new CapitaliseTaglet ();

      return taglets;
}
```

Step 4: Test

Once these steps are complete, it should be possible to test the new Taglet from markup --

```
<nu:set key="title" value="DUNE">
      <myapp:capitalise value="$title;">
            $capitalise.value;
      </myapp:capitalise>
</nu:set>
```

If everything is working correctly, the "capitalise.value" entity should resolve to "Dune".

Analytics

Neutrino incorporates granular analytics reporting, including adapters for collection by Subatomic's forthcoming Analytics component or Google Analytics. Other analytics systems can be supported by authoring adapters.

Analytics adapters are installed by assigning an instance of a subclass of neutrino.Analytics to gApplication.nuAnalytics. This instance will then in turn be used to report analytics events, which are generally raised in response to framework events. View and Page level events are raised by the framework by calling View.reportAnalyticsEvent(), allowing collection to be overridden on a per-View or Page basis. The default implementation of this method calls Application.reportAnalyticsEvent(), allowing collection to be overridden globally for the Application. The default implementation of this method calls the installed instance of Analytics to report the event.

The following events reported by the Analytics system in response to framework activity --

Event Name	**Framework event**	**Accompanying Data**
onloaded	View or Page has loaded	View and/or Page key
onbeforevisible	View or Page has started transition to visible state	View and/or Page key
onvisible	View or Page has ended transition to visible state	View and/or Page key
ondomready	View or Page's markup is final	View and/or Page key
onbeforeinvisible	View or Page has started	View and/or Page key

	transition to invisible state	
oninvisible	View or Page has ended transition to invisible state	View and/or Page key
onrefresh	View or Page received refresh request	View and/or Page key
onclick	Successful dispatch of hierarchical method invocation	View and/or Page key (or “application” if satisfied by Application), method name

Note that events raised at the View level will have both View and Page keys, whereas those raised at the Page level will only have Page keys.

Note also that the information ultimately reported to an Analytics system depends on that system's capabilities. The forthcoming Subatomic Analytics component will report all events and preserve all the event properties, but the default Google Analytics adapter only reports "onpagevisible".

Compiler

Neutrino's method of keeping markup, style, and code fragments separate is convenient during development, but because it requires many discrete retrieval transactions it is not generally suited for deployment. To prepare an application for deployment, Neutrino provides a node.js script called the Compiler. This script takes one argument, which is the (usually shortened and lower-case) name of the application.

To work with the Neutrino Compiler, an application must meet these requirements:

1. There must be a <link> with an ID of "application-css", which will be rewritten to include the combined CSS file.

2. There must be a <style> with an ID of "application-js", which will be rewritten to include the combined JavaScript file.

3. If any resources are included manually in index.html, but whose contents are included in the combined JavaScript or CSS files, the elements which include them should be marked with the "compiler-remove" class.

The Compiler does the following:

1. Invokes ./make-(applicationname)-js.cmd to combine all the application specific JavaScript files into (applicationname).js.

2. Invokes ./make-(applicationname)-css.cmd to combine all the application specific CSS files into (applicationname).css.

3. Parses the application's index.html file into a DOM, including jQuery as a script.

4. Creates the page container element inside <body>.

5. Loads the contents of all page markup files into the page container element.

6. Walks the application's markup, loading all view markup files into place.

7. Rewrites the #application-css element to include (applicationname).css.

8. Rewrites the #application-js element to include (applicationname).js.

9. Removes any elements having the "compiler-remove" class.

10. Partially disables the Neutrino loader. HTML loading is left enabled to permit loading markup whose name is not known at compile time.

11. Write the resulting document out to index-compiled.html.

Standard Components

Neutrino includes some standard UI components that can be easily dropped into your applications.

SwipeView

SwipeView allows swipe-based navigation of an area which is a window onto a bigger area. The bigger area is described by an element inside the view's element which has the "nu-container" class. The view can be made swipeable horizontally and/or vertically by the classes "nu-horizontal" and "nu-vertical" respectively on the view's element.

```
<div class="my-swipe-view nu-horizontal"
  nu-view="neutrino.components.SwipeView">
  <div class="nu-container" style="height: 300px; width: 2000px;">
    <div style="background: rgba(0,0,255,0.3); height: 200px;">
      Here is some 'swipeable' content, style me!
    </div>
  </div>
</div>
```

For a working example, see example-swipeview.

GalleryView

GalleryView is a subclass of SwipeView which allows quantised swipe of a window into a list of equally sized elements. The elements must be children of the "nu-container" element. In addition, the elements must have a class whose name is given by the value of the "nu-element-class" attribute on the view element, which defaults to "nu-element". This class name override is provided to allow for the situation where GalleryViews can be concentric.

```
<div nu-view="neutrino.components.GalleryView" class="nu-horizontal"
  style="width: 400px; height: 400px; overflow: hidden;"
  >
  <div class="nu-container" style="height: 400px;">
    <div class="nu-element" style="float: left; text-align: center; width:
400px; height: 400px; background: rgba(0,0,0,0.3);">ONE</div>
    <div class="nu-element" style="float: left; text-align: center; width:
400px; height: 400px; background: rgba(0,100,0,0.3);">TWO</div>
    <div class="nu-element" style="float: left; text-align: center; width:
400px; height: 400px; background: rgba(0,200,0,0.3);">THREE</div>
  </div>
</div>
```

For a working example, see see example-galleryview.

Media & MediaClient

The Media and MediaClient components work together to provide a simple markup-only way of playing and manipulating media. The Media component hosts the HTML5 audio or video tag and dispatches events to MediaClients associated with it; MediaClient components then show state and call the Media component back via its API.

A video-only Media component might be declared as follows --

```
<div nu-dynamic="true"
  nu-component="neutrino.components.MediaView" nu-view="media"
  >
  <video>
    <nu:list key="params.sources" prefix="source">
      <source src="$source;"></source>
    </nu:list>
  </video>
</div>
```

The Media component is configured via the Neutrino standard parameter mechanism. A list of media source URLs is expected in the "sources" property of the parameters, but in order to aid configuration from markup, a single source URL may be set via the "source" parameter (note that MediaView overrides setParams() to convert "source" to "sources", so it will always be a list in markup).

In addition, the component expects a media type field (audio or video) in the "type" field, and an optional "autoplay" property determines whether the media tag starts playing immediately. Once the component is configured, and its markup is final, it looks for a media tag in its children and registers event listeners for all the common media events.

A simple MediaClient component which displays the length of the media once it is available might be declared as follows --

```
<div nu-dynamic="true"
  nu-component="neutrino.components.MediaClientView" nu-view="duration"
  nu-media-view="media" nu-media-events="loadedmetadata"
  >
  <!-- this guards against the view showing nonsense if initially visible -->
  <nu:ifnot lhs="$params.mediastate;" rhs="">
    <div>
      $params.mediastate.duration.hours;h
      $params.mediastate.duration.minutes;m
      $params.mediastate.duration.seconds;s
      $params.mediastate.duration.milliseconds;ms
    </div>
  </nu:ifnot>
</div>
```

The nu-media-view attribute declares that this MediaClientView is associated with the MediaView which has the "media" View key. The nu-media-events attribute declares which media events this MediaClientView accepts.

Any events originating on the media tag managed by MediaView are dispatched to any MediaClientViews that are declared to receive them, via onMediaEvent() methods. The MediaView passes the event and a state object which contains some handy information some handy status information. The default implementation of this method simply puts the event and state object into

context via setParams() and then refreshes itself.

The state object contains the following information --

loadedmetadata (boolean) - whether the loadedmetadata event has been received
playing (boolean) - whether the media is currently playing
ended (boolean) - whether the media has finished playing
canplay (boolean) - whether the canplay event has been received
canplaythrough (boolean) - whether the canplaythrough event has been received

duration (object) - real time (hms/ms) media duration
progress (object) - percent, ratio, and real time (hms/ms) of media load progress
play (object) - percent, ratio, and real time (hms/ms) of media play progress
remaining (object) - percent, ratio, and real time (hms/ms) remaining to play

These properties are always available, however their values may not be correct unless the relevant events have fired to populate them. For example, the duration block is only relevant after the “loadedmetadata” event has fired. MediaClientViews are expected to coordinate their references to this information with the events for which they are registered.

There follows another example of a MediaClientView which displays load progress in blue and play progress superimposed in red.

```
<div style="width: 200px; border: 1px solid black;">
  <div class="nu-media-client"
    nu-dynamic="true"
    nu-component="neutrino.components.MediaClientView" nu-view="progressbar"
    nu-media-view="media" nu-media-events="progress,timeupdate"
    style="width: 200px; height: 50px;"
    >
    <nu:if lhs="$params.mediastate;" rhs="OBJECT">
      <div style="width: $params.mediastate.progress.percent;%; height: 50px;
background-color: blue;">
        <div style="width: $params.mediastate.play.percent;%; height: 50px;
background-color: red;">
        </div>
      </div>
    </nu:if>
  </div>
</div>
```

It is hoped that enough information is provided for virtually any media component to be authored exclusively in dynamic markup. However, if custom functionality is needed which requires Javascript, then an application specific View need only extend neutrino.components.MediaClientView and override onMediaEvent().

MediaClientView also implements a Javascript API, which can be called by nu-action or otherwise, and which is proxied to the MediaView to do common media related tasks --

play()
This causes the media to play from the current time position.

pause()

This causes the media to pause, unsurprisingly.

setPositionFromFraction(inFraction)

This sets the media's current time property to the specified fraction multiplied by the media's duration. The range of fraction is 0..1.

setPositionFromPercent(inPercent)

This sets the media's current time property to the specified percentage of the media's duration. The range of percent is 0..100.

In addition, MediaClientView has default down, move, and up handlers for implementing scrub controllers. Simply call onMouseDown, onMouseMove, and onMouseUp from nu-action on the actual element itself (not an element within the view) for scrubbing to work automatically.

For a working example of the media components, see example-media.

Neutrino Client Reference

Neutrino Attributes

Upon encountering a section of markup, Neutrino recognizes the following attributes.

Note regarding method dispatch: Neutrino searches for specified methods by walking the clicked elements' parents and looking for enclosing Views and Pages. If a match is not found, then the Window and Application objects are searched. The method name comparison is case-sensitive. A match stops the search.

nu-action and nu-action-params

nu-action="(eventName) action1: param1/param2/param3/..."
Signifies that a the event specified by eventName (default is click) firing on this element should cause the specified action to occur. The currently supported actions are:

addclass: selector/class
Adds the specified space-separated CSS classes to the specified CSS selector.

removeclass: selector/class
Removes the specified space-separated CSS classes from the specified CSS selector.

toggleclass: selector/class
Toggles the specified space-separated CSS classes on the specified CSS selector.

showview: viewkey/transitioninclass
Shows the view with the specified key, optionally with a CSS class to serve as a transition-in class.

hideview: viewkey/transitionoutclass
Hides the view with the specified key, optionally with a CSS class to serve as a transition-out class.

toggleview: viewkey/transitioninclass/transitionoutclass
Toggles the visibility of the view with the specified key, optionally with CSS classes to serve as transition in and out classes.

setpage: pagekey/transitionoutclass/transitioninclass
Transitions out the current page, if any, and transitions in the page identified by the page key. If the page referenced by the page key has not yet been loaded, it is loaded before the transition begins. Note that the order of the transition in and out class names is reversed from toggleview.

Parameters for call, showview, and toggleview's "show" phase can be provided via a *nu-action-params* attribute. The format of this attribute is CSS-style key-value pairs:

nu-action-params="key1: value1; key2: value2;"

Note that any number of actions can be attached to an element. Multiple actions are denoted by ascending numeric suffixes on the "nu-action-" stem, with parameters for each action denoted by the suffix "-params" . The suffix can begin with zero or 1. The actions are performed in suffix sequence.
The sequence of actions can start with nu-action, nu-action-0, or nu-action-1.

nu-app

nu-app="appName"

On the body element, specifies the name of the application. Neutrino uses this name to set the document's title. The default name is "Neutrino".

nu-component

nu-component="className"

In conjunction with nu-view, specifies the name of a JavaScript class to serve as the code for this class. Useful in situations where several Views have the same code but different markup and styling. Note that an attempt is made to instantiate a class with the same name as the attribute value, then the attribute value with the "View" suffix, and finally an attempt is made to load a view from views/js/classname.js.

If nu-component is present, Neutrino makes no attempt to load a JavaScript class referenced by nu-view.

nu-page

nu-page="pageKey"
On the enclosing element of Page markup, or on an immediate child of the Page container, this specifies that this element is the page element. Required.

nu-page-title

nu-page-title="pageTitle"
On a page element, specifies the title of the page. When the page is shown, the document title will be constructed from the application name, optionally from *nu-app* on <body>, which defaults to "Neutrino", and this page title, which defaults to the page key.

nu-progress-id

nu-progress-id="progressID"
On an element with nu-view, specifies the ID of another element which contains "progress" markup, which is shown while the View's dynamics are being resolved.

nu-session

nu-session="true|false"
Signifies that this element should be shown or hidden (via visible/invisible classes) based on whether there is a session active (via gApplication.session.active). If this flag is true, the element is only shown if there is a session, and vice versa.

nu-start-page

nu-start-page="pageKey"
On the body element, defines which page will load on application startup.

nu-start-page-delay

nu-start-page-delay=”delay”
On the body element, defines the delay between application startup and loading the initial page. Value is in milliseconds. The default is 1ms.

nu-swipe

nu-swipe=”methodStem”
Signifies that mouse or touch down, move, and up on this element should cause the methods Start, Move, and End of the named method stem to be called via regular method dispatch. Will probably be deprecated in favour of multiple nu-actions.

nu-view

nu-view=”viewKey:flags”
Signifies that a Neutrino View should be associated with this element. This involves:

- attempting to instantiate the view directly, either using the view key as a class name, using the view key in combination with the “View” suffix, or loading JavaScript code, if any, from js/views/viewkey.js
- instantiating the JavaScript class, if any, with the name ViewKeyView
- loading CSS, if any, from css/views/viewkey.css
- loading markup, if any, from html/views/viewkey.html

If the element with nu-view has any child elements, then no attempt is made to load dynamic markup.

Case is honored when determining JavaScript class names, but not when determining file paths or internal view or page keys. So this view reference:

```
<div nu-view="ButtonBar">…</div>
```

would result in ButtonBarView being instantiated, loaded from js/views/buttonbar.js, with the view key “buttonbar”.

The flags included after the colon in the view key determine which resources should be loaded. This is a deploy-time optimization step to prevent the loading of resources which are known not to be required. Currently supported flags are h, j, and c, signifying HTML, JavaScript, and CSS respectively; the presence of the flag means an attempt to load that resource is made. If the colon is present in the view key field, then the flags following it are honored, so an empty flag field means no attempt to load resources is made.

nu-view-key

nu-view-key=”viewKey”
Overrides the view key for situations where nu-view does not provide uniqueness for external referencing via nu-action etc. An example would be views reused within other views inside a tab view, etc.

nu-view-params

nu-view-params=”key1: value1; key2: value2;”
Provides default parameters for the View specified with nu-view, they are active after onLoad() time.

Neutrino Class System

Neutrino employs a very simple, standards-based method of implementing class functionality, which has few conventions and rules.

```
// js/views/menu.js

var MenuView = function ()
{
  // call superclass constructor from our constructor
  neutrino.View.call (this);
}

// arrange the inheritance between MenuView and neutrino.View
neutrino.inherits (MenuView, neutrino.View);

// override a method from neutrino.View
// calling the superclass
MenuView.prototype.onDOMReady = function ()
{
  neutrino.View.prototype.onDOMReady.call (this);
}
```

Dynamic Tags

Neutrino dynamic tags enable powerful integration of HTML with dynamic elements. If an element has a “nu-view” attribute, and also has a “nu-dynamic” attribute whose value is “true”, then Neutrino will engage the dynamics engine to work on that View’s children.

Neutrino’s dynamic tags all have one thing in common. If the tag inserts a variable into context, then it will optionally have a prefix which is derived from any “prefix” attribute and defaults to the name of the tag (minus the nu: or any other namespacing). This is mainly to help identify where variables are coming from, but also to contextualize them according to your application.

nu:changecase

```
<nu:changecase
  value=”...” (required)
  mode=”upper|lower”
  prefix=”...”
  >
```

This tag changes the case of its “value” attribute and puts the new value into context as prefix.value.

Note that <nu:changecase> recognises <nu:uppercase> and <nu:lowercase> as tag names, in which case the “mode” attribute is not required. The appropriate tag entries in the configuration must of course be there, but the Application class does this by default.

nu:comment

```
<nu:comment>
```

This tag is a markup comment only and is pulled from the DOM on detection. It's designed to offset the mess resulting from including HTML comments in dynamic markup.

nu:date

```
<nu:date
  ms="..." (required)
  format="..." (required)
  >
```

This tag converts "milliseconds since 1970" format dates into readable ones. The "ms" attribute must contain the milliseconds value, and the "format" attribute can optionally be a formatting string. The formatting string can contain any of the following tokens, which will be substituted by fields from the date.

HH	2 digit 24-hour hour
mm	2 digit minute
ss	2 digit second
dd	2 digit numerical day
EE	3 character day of week name
MM	3 character month of year name
yyyy	4 digit year
Z	7 character timezone

Note that these fields come from processing the output of the regular JavaScript Date *toString()* call. Other options are not yet available.

nu:get

```
<nu:get
  key="..." (required)
  prefix="..."
  >
```

This tag retrieves items from context and puts them back into context. This might not immediately appear to be desired functionality. However, it's not currently possible to substitute into substituted text, so the following won't work --

```
$application.$messagekey;;
```

With nu:get, this is possible via --

```
<nu:get key="application.$messagekey;">$get.value;</nu:get>
```

nu:if

```
<nu:if
  lhs="..."
  rhs="..."
  operation="..."
  >
```

This tag implements basic equivalence. If the "lhs" and "rhs" attributes are exactly equal, then the children of the tag replace the tag in the tree. If not, the tag and its children are deleted.

Unlimited numbers of additional terms can be added via numerical suffixes off the "lhs-" and "rhs-" stems. The numerical suffixes can start from zero or one. If multiple terms are provided, the default operation which causes the entire expression to be true or false is "and". This can be overridden with the "operation" attribute.

nu:ifnot

```
<nu:ifnot
  lhs="..."
  rhs="..."
  operation="..."
  >
```

This tag implements basic not-equivalence. If the "lhs" and "rhs" attributes are exactly equal, then the tag and its children are deleted. If they are not exactly equal, then the children of the tag replace the tag in the tree.

Unlimited numbers of additional terms can be added via numerical suffixes off the "lhs-" and "rhs-" stems. The numerical suffixes can start from zero or one. If multiple terms are provided, the default operation which causes the entire expression to be true or false is "and". This can be overridden with the "operation" attribute.

nu:json

```
<nu:json
  url="..." (required)
  offlineurl="..."
  jsonp="true|false"
  cachekey="..."
  cachelifetime="..."
  prefix="..."
  >
```

This tag grabs a JSON feed from a URL and puts it into context. Note this tag is asynchronous; the final state of this tag's children is not assured until *onDOMReady()* has been called on the enclosing view. The "cachekey" and "cachelifetime" attributes control whether the results of the retrieval can be cached and the lifetime of that cache entry.

For example, with this JSON feed:

```
{
  "firstname": "Father",
  "secondname": "Christmas",
  "address": "Lapland"
}
```

And this template markup:

```
<nu:json url="..." prefix="xmas">
  <div>first name is $xmas.firstname;</div>
  <div>second name is $xmas.secondname;</div>
  <div>address is $xmas.address;</div>
</nu:json>
```

The resulting final markup would be:

```
<div>first name is Father</div>
<div>second name is Christmas</div>
<div>address is Lapland</div>
```

nu:link

```
<nu:link
  href="..." (required)
  type="..."
  class="..."
  >
```

This tag inserts a <link> tag into <head> with the specified href, type, and class attributes. Will probably be deprecated in favour of a more general Taglet that inserts into <head>.

nu:list

```
<nu:list
  key="..." (required)
  offset="..."
  limit="..."
  searchkey="..."
  searchvalue="..."
  prefix="..."
  >
```

This tag allows iteration of a JavaScript array object found in context by the key specified by the (required) "key" attribute. For each element in the list, a copy of the tag's children is included, with the element included in that copy's context. The iteration of the list starts at zero, unless an "offset" attribute is provided, and iterates through the entire list, unless a "limit" attribute is provided.

For example, with this JavaScript array in context under the name "numbers":

```
[
```

```
  "one",
  "two",
  "three"
]
```

And the following template markup:

```
<div>
  <nu:list key="numbers" prefix="number">
    <div>number is $number;</div>
  </nu:list>
</div>
```

The resulting final markup would be:

```
<div>
  <div>number is one</div>
  <div>number is two</div>
  <div>number is three</div>
</div>
```

The offset, limit, searchkey, and searchvalue attributes allow filtering of the list. Offset and limit together specify the start index and maximum iterations into the list. Searchkey and searchvalue specify that only objects in the list with matching properties are included.

<nu:list> also introduces some meta information into context regarding the collection --

prefix.meta.globalindex - the index into the collection
prefix.meta.globalcount - the number of elements in the collection
prefix.meta.index - the index into the subcollection (relative to offset and limit)
prefix.meta.count - contains the number of elements in the subcollection
prefix.meta.isfirst indicates whether this element is the first in the subcollection
prefix.meta.islast indicates whether this element is the last in the subcollection

Note that the metadata relative to the subcollection does not take elements matched by search criteria into account -- prefix.meta.count is not therefore a count of so matched elements.

nu:log

```
<nu:log
  value="..."
  valuekey="..."
  >
```

This tag logs either the value in the "value" attribute, or the context object whose key is in the "valuekey" attribute.

nu:map

```
<nu:map
  key="..." (required)
  prefix="..."
```

```
  >
```

This tag is the equivalent of <nu:list> for JavaScript Objects (called maps in other languages, and here for clarity). For each element in the map which has a string key and a non-function value, a copy of the subtree is produced, contextualised with key and value for this element. Metadata is handled similarly to <nu:list>.

nu:numberformat

```
<nu:numberformat
  value="..." (required)
  type="fixed|precision" (required)
  digits="..." (required)
  prefix="..."
  >
```

This tag is effectively a markup interface to the JavaScript *toFixed()* and t*oPrecision()* methods of the number type. It formats numbers and puts the ensuing value into context as prefix.value.

nu:replace

```
<nu:replace
  value="..." (required)
  replace="..." (required)
  with="..." (required)
  prefix="..."
  >
```

This tag does straightforward text replacement with its arguments and puts the ensuing value into context as `prefix.value`.

nu:set

```
<nu:set
  key="..." (required)
  value="..."
  valuekey="..."
  context="view|page|window|application"
  >
```

This tag sets variables in context. Usually variables are introduced by tags, and have only tag context, but sometimes it is necessary to manage them manually, especially when promotion to another context is needed. For example, if you needed to effectively do something based on a logical "or" between two <nu:if> conditions, you could hold a flag in page context so that the value would be valid outside the <nu:if> tags --

```
<nu:set key="redorblue" value="false" context="page"></nu:set>

<nu:if lhs="$view.colour;" rhs="red">
  <nu:set key="redorblue" value="true" context="page"></nu:set>
</nu:if>
```

```
<nu:if lhs="$view.colour;" rhs="blue">
  <nu:set key="redorblue" value="true" context="page"></nu:set>
</nu:if>

<nu:if lhs="$redorblue;" rhs="true">
  view is red or blue
</nu:if>
```

nu:xml

```
<nu:xml
  url="..." (required)
  offlineurl="..."
  jsonp="true|false"
  cachekey="..."
  cachelifetime="..."
  prefix="..."
  >
```

This tag grabs an XML feed from a URL, converts it to a JavaScript object tree using *neutrino.Utils.elementToJs()*, and puts it into context. This tag is asynchronous; the final state of this tag's children is not assured until *onDOMReady()* has been called on the enclosing view.

The "cachekey" and "cachelifetime" attributes control whether the results of the retrieval can be cached and the lifetime of that cache entry.

Class Reference

Application

(public methods)

Application()
Constructor. Initialises member property state, caches sections of the request including the URL and parameters, determines logging situation, caches browser specifics. Registers hashchange handler for bookmarkable state.

start()
Usually called by main(), immediately after constructing Application. Makes the page container and other required containers, configures Window as the default page, starts the preloader, preloads any extant pages in the page container, converts any static CSS to browser neutral, walks <body>, and attempts to determine an initial page.

Overrides on start() usually call the superclass first, then overwrite nuServerHost and other common application-specific properties.

callMethodOnEnclosingView(inMethod, inElement, inEvent, inArgument, inParams)
Default method dispatcher. Walks the specified element's parent elements looking for Views (or Pages). If a View is found, then a check is made for a function with the specified name. If the search exhausts

the DOM, then Application is also checked. If a match is found, any parameters passed are set in the recipient, via *setParams()*, before the method is invoked.

createLoadTreeWalker()
Provided so that applications can determine the LoadTreeWalker class used by the framework.

createOnBeforeVisibleTreeWalker()
Provided so that applications can determine the OnBeforeVisibleTreeWalker class used by the framework.

createOnVisibleTreeWalker()
Provided so that applications can determine the OnVisibleTreeWalker class used by the framework.

createOnBeforeInvisibleTreeWalker()
Provided so that applications can determine the OnBeforeInvisibleTreeWalker class used by the framework.

createOnInvisibleTreeWalker()
Provided so that applications can determine the OnInvisibleTreeWalker class used by the framework.

createView()
Provided so that applications can determine the default View class that is instantiated in the case where no class specific to that View is found or specified.

generateCacheToken()
If nuDevMode is set, generate a unique token to be used as a cache-busting parameter on the next URL to be requested. Currently set from the current time.

getMessage(inKey, inContext)
Retrieve a message from nuMessages by key, satisfy any included context references, and return a final string.

getPage(inKey)
Retrieve a page from nuPages by key.

setPage(pageKey, transitionInvisibleClass, transitionVisibleClass, parameters)
Transition out the current Page, if any, and transition in the Page referenced by the specified key. Use custom transitions if specified. Set parameters on the incoming page if specified.

unloadPage()
Unload a page, removing its markup and CSS. Currently unsupported.

getView(viewKey)
Retrieve a view by key, from the current Page or Window.

showView(viewKey, parameters, transitionVisibleClass)
Show the View with the specified key. Use custom transition-in class if specified. Set parameters on the View beforehand, if specified.

hideView(viewKey)
Find a view by key, from the current Page or Window, and start its transition from visible to invisible.

toggleView(viewKey)
Find a view by key, from the current Page or Window, and start a transition to reverse its visibility state.

showProgressView()
If <body> contains a progress-view-key attribute, show the View that it specifies. Called before the asset loader begins and before a page transition.

hideProgressView()
If <body> contains a progress-view-key attribute, show the View that it specifies. Called after the asset loader finishes and after a page transition.

makePhoneInterface()
Make the instance of PhoneInterface most appropriate to the user agent. Currently unsupported.

setCheckOnline(inEnable, inPeriod)
Set whether Neutrino makes periodic checks to determine whether there is network connectivity.

(private methods)

setInitialPage()
Check for an initial Page key in the "nu-start-page" attribute on <body>. If an initial Page is specified, then it is set after the delay, if any, specified in "nu-start-page-delay" attribute on <body>. The default delay is 1ms.

addPageEventListeners()
After a Page is loaded, add the relevant listeners to its element so that transitions etc can be handled.

initialiseJanx()
Initialise the dynamics engine. Configure the Taglet manager with the default Taglet set, and call the Application's *getTagletConfiguration()*, if implemented, to add additional Taglets. Make the Janx instance and insert initial elements into the root Janx context.

janxify(inElement, inContext)
Run the dynamics engine on the specified element using the specified context.

janxifyTo(inFromElement, inToElement, inContext, inAppend)
Run the dynamics engine on the specified element using the specified context, but store the final markup in another specified element. Currently this is implemented by duplicating the template markup tree into the destination markup tree, rather than the dynamics engine being able to process between two trees.

janxifyText(inText, inContext)
Run the dynamics engine on the specified string using the specified context.

makeContainers()
Make any required Neutrino containers within the DOM. Currently these are the Page container and the browser specific <style> at the end of <head>.

preloadAssets()

Load the “preload.json” file from the server and if it contains any URLs, set the asset loader off loading them. During the time the asset loader is busy, the progress View is shown, if its key is configured on <body>.

onAssetLoadStart(inAssetLoader)
Called by the asset loader prior to an attempt being made to load the first asset in preload.json.

onAssetLoadProgress(inAssetLoader, inIndex, inCount, inPercentage)
Called by the asset loader after every attempt to load an asset. The current item index, the total number of assets to be loaded, and a percentage completion are passed.

onAssetLoadFinish(inAssetLoader)
Called by the asset loader when the attempt to load the last asset in preload.json completes.

preloadPages()
Register any page markup extant in the page container when the application loads.

processCSS()
Find <style> and <link> tags with the class “nu-browser-neutral-css”, convert their styles to browser specific ones, and store the resulting style information in the <style> tag with the ID “nu-browser-specific-css”.

setPageInternal(inPageKey, inTransitionInvisibleClass, inTransitionVisibleClass, inParams)
Actually go ahead and transition out the current Page and transition in a new one. Called from the onhashchange handler, and driven from state stored when setPage() changes the document URL or when the user changes the URL in the location bar.

setParams(inParams)
Update nuParams from the passed-in object. Matching parameter keys are overwritten, others are left untouched.

setupLogging()
Set up nuLogKeywords. Check for a nu-log or nu-logging parameter in the URL, match up keywords from it with the set, and configure the nuLogMask appropriately.

isLogging(inMask)
Check to see whether Neutrino is configured to log for a particular component by checking its mask in nuLogMask.

setupParameters()
Copy parameters from the URL to nuParams.

setUserAgentFlags()
Configure nuBrowser from information found in the user-agent string.

onBeforeRequest(inRequest)
Called by the various network requesters to allow Application to monkey with the request before it goes out. Common uses for this include add cache-busting parameters, etc.

onPageAnimationStart(inEvent)
Called when an animation (transition) on a Page element starts.

onPageAnimationEnd(inEvent)
Called when an animation (transition) on a Page element ends. Used to manage Page transitions between visibility states.

walk(inElement, inRunDynamics, inMakeVisible)
Called to process an element and its subtree that Neutrino has not seen before. Runs the load walker, on-before-visible walker, and optionally, the on-visible walker. Whether to run dynamics during onBeforeVisible() is also optional.

walkChildren(inElement, inRunDynamics, inMakeVisible)
Called to process the subtree of an element that Neutrino has not seen before. Runs the load walker, on-before-visible walker, and optionally, the on-visible walker. Whether to run dynamics during onBeforeVisible() is also optional.

(properties)

nuAnalytics
Instance of a class which is handling analytics calls generated by the framework. Defaults to an instance of neutrino.DummyAnalytics which does nothing. Can be set to an instance of neutrino.Analytics in an overridden constructor or start() in order to enable reporting to the Neutrino Server analytics facility.

nuAppName
Name of application. Obtained from “nu-app” attribute of <body>, defaulted to “Neutrino”.

nuBrowser
Object containing details of the user agent. Contains flags such as isWebKit, etc.

nuCache
Object containing application-wide cached entities. Is public; the only framework client is AjaxTaglet and its subclasses.

nuCheckOnline
Boolean determining whether the framework makes periodic checks to determine whether there is network connectivity. If the framework can successfully load the URL “/neutrino/assets/online.gif”, then connectivity is assumed.

nuCheckOnlinePeriod
Period of periodic online check.

nuCheckOnlineSequence
Current value of cache-busting attribute added to the URL used to determine whether there is network connectivity.

nuCheckOnlineTask
JavaScript setInterval() task which checks for network connectivity.

nuDevMode
Determine whether Neutrino adds a cache-busting parameter to each network request. Defaults to true, recommend turning off by clearing it in main(), between the Application constructor and the call to start().

nuIsOnline
Whether the “check online” system currently thinks there is network connectivity.

nuJanx
Instance of neutrino.janx.Janx used to perform all dynamic transformations.

nuLoadWalker
Instance of neutrino.LoadTreeWalker used to walk DOM subtrees and load any referenced Neutrino entities.

nuLogKeywords
Object containing all possible values for the granular logging system with their corresponding mask values.

nuLogMask
The currently active log mask, which determines which framework subsystem(s) are allowed to log.

nuMessages
Object containing localised messages, indexed by key. Not currently populated by the framework, but the contents are available via getMessage().

nuOnBeforeVisibleWalker
Instance of neutrino.OnBeforeVisibleWalker used to walk DOM subtrees and call onBeforeVisible() and run the dynamics on any found Views.

nuOnVisibleWalker
Instance of neutrino.OnVisibleTreeWalker used to walk DOM subtrees and call onVisible() on any found Views.

nuPage
Instance of neutrino.Page which is the currently visible Page.

nuPages
Object containing the currently loaded Pages, indexed by key.

nuPageTransitionInvisibleClass
The name of the transition-invisible class associated with the current page transition. Maintained as state through the setPage/setPageInternal gap.

nuPageTransitionVisibleClass
The name of the transition-visible class associated with the current page transition. Maintained as state through the setPage/setPageInternal gap.

nuParams
Object containing parameters set on Application. Parameters come from the document’s HREF or by explicit calls to setParams() which typically happen when method dispatch resolves to Application scope and has accompanying parameters. Parameters so set are mirrored into nuRootJanxContext, prefixed with “param.”.

nuRootJanxContext

The Janx context from which all other Janx contexts ultimately derive. Usually contains application configuration information and anything set in Application's parameters.

nuServerHost
Scheme, host, and port of the hosting server. Mostly optional, but available for insertion into markup etc via application.serverhost just in case.

nuSessionWalker
Instance of neutrino.SessionTreeWalker used to walk DOM subtrees and show and hide elements appropriately according to the value of the "nu-session" attribute.

nuTagletManager
Instance of neutrino.janx.TagletManager, which manages access to the Taglet list. Pretty much a do-nothing class which is ultimately going away.

nuTaglets
Object containing the currently registered Janx Taglets, indexed by tag name. Setup by initialiseJanx(), modified by getTagletConfiguration(), if implemented by Application subclass.

nuURL
Copy of document.location.HREF, without the query or anchor sections.

nuURLScheme
The scheme section from document.location.HREF.

nuWindow
Instance of neutrino.Page, associated with <body>. Serves as the default Page if there are no Pages, and as the Page for any Views declared outside the page container.

DOM

neutrino.DOM.listen (inElement, inEventName, inFunction, inCapturePhase, inThis)
Register a callback for an event on an element. Will use jQuery if available. Capture phase and this parameters are optional.

neutrino.DOM.unlisten (inElement, inEventName, inFunction, inCapturePhase, inThis)
Unregister a callback for an event on an element. Capture phase and this parameters are optional, but must match the parameters originally supplied to listen() for this event handler.

In the following CSS and Data handling methods, the "inSelectorOrElement" parameter may either be a string which contains a selector or an element object.

neutrino.DOM.addClass (inSelectorOrElement, inClassName)
Adds the specified space-separated list of class names to the found elements.

neutrino.DOM.removeClass (inSelectorOrElement, inClassName)
Removes the specified space-separated list of class names from the found elements.

neutrino.DOM.toggleClass (inSelectorOrElement, inClassName)
Toggles the specified space-separated list of class names on the found elements.

neutrino.DOM.hasClass (inSelectorOrElement, inClassName)
Returns whether the found element has the specified single class.

neutrino.DOM.getData (inSelectorOrElement, inKey)
Returns the data item indexed by the specified key on the found element.

neutrino.DOM.putData (inSelectorOrElement, inKey, inValue)
Sets the data item indexed by the specified key on the found element to the specified value.

neutrino.DOM.find (inElement, inSpec)
Returns a list of elements within the specified element which match the specified specification.

neutrino.DOM.findImmediateChildWithClass (inElement, inClass)
Returns the first immediate child on the specified element which has the specified class.

neutrino.DOM.showElementsBySelector (inSelector)
Removes the "nu-invisible" class from all elements matching the specified selector.

neutrino.DOM.hideElementsBySelector (inSelector)
Adds the "nu-invisible" class to all elements matching the specified selector.

neutrino.DOM.getChildElements (inElement)
Returns a list of immediate children of the specified element which are elements.

neutrino.DOM.getElementList (inSelectorOrElement)
If the "inSelectorOrElement" parameter is a string, return a list of elements which match the selector. If the parameter is an object, assume that it is an element, and return a list with that element as its sole entry.

neutrino.DOM.getParents (inElement)
Return a list of parents of the specified element, ordered by proximity to the element.

neutrino.DOM.globalEval (inScript)
Assume that the "inScript" parameter contains JavaScript and add it to the global JavaScript execution context.

neutrino.DOM.moveChildren (inFromElement, inToElement)
Move all children of the "inFromElement" parameter to the "inToElement" parameter.

neutrino.DOM.insertChildrenBefore (inParentNode, inBeforeElement)
Insert all children of the "inParentNode" parameter as siblings of the "inBeforeElement" parameter.

neutrino.DOM.replaceWithChildren (inReplaceElement, inParentElement)
Insert all children of the "inParentElement" parameter as siblings of the "inReplaceElement" parameter, then remove the "inReplaceElement" parameter.

neutrino.DOM.removeChildren (inElement)
Remove all children of the specified element.

Utils

(public methods)

neutrino.Utils.elementToDuplicateChildren (inNode)
Track children of the specified node that have duplicate tag names. Part of the largely deprecated less-than-faithful elementToJs2() system.

neutrino.Utils.elementToJs (inNode)
Convert an XML tree to a JavaScript object tree the supported faithful way. More awkward to traverse from markup, though.

neutrino.Utils.elementToJs2 (inNode)
Convert an XML tree to a JavaScript object tree the largely deprecated less-than-faithful way. Easier to traverse from markup, and handy in areas where (for example) attribute handling is not required.

neutrino.Utils.jsToElement (inJS)
Converts a JavaScript object tree to a DOM tree. Essentially the opposite of elementToJs().

neutrino.Utils.getURLContents (inRequest)
General purpose method to retrieve the contents of a URL. It effectively clones and abstracts the jQuery ajax() call, and indeed will use jQuery if available and fall back to XHR when not. Note however that JSONP is not currently supported without jQuery.

url - the URL whose contents to retrieve
data - URL parameters in key=value&key1=value1 format. preceding ? is not expected.
offlineURL - the substitute URL to use if Neutrino thinks there is no connectivity
offlineData - parameters for substitute URL
type - HTTP access method, default is GET
async - whether the request should be asynchronous
dataType - json/jsonp/text/xml
success - success handler function
error - error handler function

neutrino.Utils.handleEvent = function (inEvent, inElement)
Handles events registered by nu-action. Forwards to handleActions().

neutrino.Utils.handleAction = function (inEvent, inElement, inAction, inParams)
Handles a single action from a list registered by nu-action, called by handleActions(). Switches on the opcode in the action to call methods, manipulate CSS classes, show and hide Views, change Pages, etc.

neutrino.Utils.handleActions = function (inEvent, inElement)
Handles events registered by nu-action. Runs through any nu-action or numerically suffixed nu-action attributes, parsing them, ensuring that they match the handled event, and handing each in turn off to handleAction(). Each action in the list is executed regardless of whether actions succeed or fail.

neutrino.Utils.parseAction = function (inActionString)
Parses an action of the form “(event) opcode: param1/param2/...” into a more easily manageable JavaScript object structure.

neutrino.Utils.parseParams = function (inParamString)

Parses a parameter string of the form “key1: value1; key2: value2;”, for use with nu-action-params or suffixed equivalent, into a simple JavaScript object.

neutrino.Utils.stripSpaces = function (inString)
Strip leading and trailing spaces from a JavaScript string.

neutrino.Utils.unparseAction = function (inAction)
Unparses an action object of the form produced by parseAction() back into the form “(event) opcode: param1/param2/...”. Used by LoadTreeWalker after a virtual event type is mapped back to a real event type and the appropriate action attribute is rewritten so that it will match the ultimately occurring event.

neutrino.Utils.unparseParams = function (inParamString)
Unparses a parameter object of the form produced by parseParams() back into the form “key1: value1; key2: value2;”. Used by Application.setPage() to construct a hash of the target location, the setting of which then results in an onhashchange event.

neutrino.Utils.validateEmailAddress = function (inAddress)
Validates an email address. This is a reasonably thorough but not exhaustive validation.

View

On calling superclasses: views that override methods in the Neutrino View class must, under normal circumstances, call the corresponding method in View in addition to doing anything else. For example, if MenuView overrode onLoaded(), it might look like this:

```
MenuView.prototype.onLoaded = function ()
{
  // maybe do some stuff before calling the superclass here

  // call View's onLoaded(), but with our "this"
  neutrino.View.prototype.onLoaded.call (this);

  // maybe do some stuff after calling the superclass here
}
```

(public methods)

YourView()
View constructor. Set any initial state.

configure(inKey, inElement, inPage)
Called shortly after loading to associate the View with its parent element and arrange its Janx context, etc.

onAnimationStart(inEvent)
Called when an animation on an element within the View starts.

onAnimationEnd(inEvent)
Called when an animation on an element within the View ends.

onAsyncStart()

Called when an asynchronous operation started by a Taglet within the View starts. In the View class, this results in nuAsyncCount being incremented.

onAsyncEnd(inData)
Called when an asynchronous operation started by a Taglet within the View starts. In the View class, this results in nuAsyncCount being decremented.

onLoaded()
Called when the view is loaded into the DOM, regardless of whether it's visible or not. Note that any views referenced by the view's markup are not guaranteed to be loaded at this point.

onBeforeVisible(inRunDynamics)
Called when a view begins the transition from the invisible to visible state, either when its enclosing page is called to show it, or when an enclosing view or page is shown. In the former case, this is called when the transition-visible animation starts.

Note that if the view is dynamic, then the process of resolving dynamics begins in *onBeforeVisible()*. Note also that if the view is static, then *onDOMReady()* is called from *onBeforeVisible()*.

Whether to run dynamics or not is optional, as it's often the case that onBeforeVisible() is called as part of a dynamics run which started with a View that is enclosing this one.

onVisible()
Called when a view ends the transition from the invisible to visible state, either when its enclosing page is called to show it, or when an enclosing view or page is shown. In the former case, this is called when the transition-visible animation ends.

onBeforeInvisible()
Called when a view begins the transition from the visible to invisible state, either when its enclosing page is called to hide it, or when an enclosing view or page is hidden. In the former case, this is called when the transition-invisible animation starts.

onInvisible()
Called when a view ends the transition from the visible to invisible state, either when its enclosing page is called to hide it, or when an enclosing view or page is hidden. In the former case, this is called when the transition-invisible animation ends.

onDOMReady()
Called once the markup for a particular view is final. If the view is static, or contains only synchronous dynamic operations, then this will be called from *onBeforeVisible()*. If the view contains asynchronous dynamic operations, such as <nu:json>, then this will be called when all such operations complete.

Note that the timing of this call is independent of the timing of *onVisible()*, which denotes that CSS says the view is visible, in itself no reflection on the state of any asynchronous operations.

onBeforeUnload()
Called before a Page is unloaded. Unloading is currently unsupported.

onFormSubmission(inEvent)

HTML5 form validation equivalence method. Note that the framework does not register an event handler automatically for forms; this must be done manually by calling this method from nu-action/onchange on a <form> tag.

onSwipeStart(inEvent)
Default handler, called on touch or mouse down on an element marked with nu-swipe. Will ultimately be deprecated, replaced by nu-action/down, etc.

onSwipeMove(inEvent)
Default handler, called on touch or mouse move on an element marked with nu-swipe. Will ultimately be deprecated, replaced by nu-action/move, etc.

onSwipeEnd(inEvent)
Default handler, called on touch or mouse up on an element marked with nu-swipe. Will ultimately be deprecated, replaced by nu-action/up, etc.

clickNext()
Click handler for View's largely deprecated paging system.

clickNext()
Click handler for View's largely deprecated paging system.

isQuiescent()
Determine whether a View is in transition between visibility states. Deprecated; currently hardwired to remove any transition classes from the View's element and always return true.

isVisible()
Determine whether a View is visible, as defined as whether its element has the "invisible" class.

reportAnalyticsEvent(inEventName, inDetail)
Called by the framework to report an analytics event related to this View. Called on the View so as to facilitate overrides to customise the event reported.

setParams(inParams)
Called mostly by nu-action in response to a parameterised action. Overwrites *only matching parameters* in View state, also mirrors changes to nuJanxContext prefixed with "param.".

checkForDynamics()
Called once per View lifecycle to check to see whether the View has a nu-dynamic attribute which is set to "true". If so, then the View's element's inner HTML is copied into View state and all children of the View's element are removed.

janxify()
Called from onBeforeVisible() to satisfy the View's dynamics, if any. If the View is static, or, after the Janx run, there are no outstanding asynchronous transactions, then onDOMReady() is called. Note that onDOMReady() is called after a short setTimeout(), so that the rendering engine can process the removal of the "invisible" class before onDOMReady() assumes that its elements have size, etc.

updatePageIndicators(inData)
Call to update state for View's largely deprecated paging system.

validateEmail(inInput)
HTML5 form validation handler for form fields of type “email”.

validateNumber(inInput)
HTML5 form validation handler for form fields of type “number”.

validateURL(inInput)
HTML5 form validation handler for form fields of type “url”.

validatePattern(inValue, inPattern)
HTML5 form validation handler for the “pattern” attribute.

(properties)

nuAsyncCount
Count of outstanding asynchronous transactions, usually resulting from network calls by <nu:json> or similar. If the View is dynamic, then onDOMReady() will not be called unless all asynchronous transactions have completed.

nuConfigureCalled
Boolean signalling whether neutrino.View.prototype.configure() has been called on this View. Part of protective measures to alert the developer if an override fails to call the corresponding method in its superclass.

nuConstructorCalled
Boolean signalling whether neutrino.View(), ie the constructor, has been called on this View. Part of protective measures to alert the developer if an override fails to call the corresponding method in its superclass.

nuCheckedForDynamics
Boolean signalling whether this View has checked its attributes for nu-dynamic and stashed away its innerHTML for later processing during onBeforeVisible().

nuElement
Instance of DOM Element which represents the element associated with this View.

nuFormValidators
Object containing functions which implement validation operations, indexed by the <input> tag type. For use with the HTML5 form validation equivalence system.

nuJanxContext
Current Janx context for this view. Inheritance chain depends on where the dynamics run started, but if this View has its dynamics resolved alone, then the inheritance runs through Page to Application.

nuKey
The key by which this View is uniquely identified within the Page, or if this View is a Page, within the Application. Set by nu-view or nu-view-key.

nuOnBeforeVisibleWalker
Instance of neutrino.OnBeforeVisibleWalker used to walk DOM subtrees and call onBeforeVisible() and run the dynamics on any found Views.

nuOnVisibleWalker
Instance of neutrino.OnVisibleWalker used to walk DOM subtrees and call onVisible() on any found Views.

nuOnBeforeInvisibleWalker
Instance of neutrino.OnBeforeInvisibleWalker used to walk DOM subtrees and call onBeforeInvisible() on any found Views.

nuOnInvisibleWalker
Instance of neutrino.OnInvisibleWalker used to walk DOM subtrees and call onInvisible() on any found Views.

nuParams
Object containing parameters set on this View. Parameters come from explicit calls to setParams() which typically happen when method dispatch resolves to this View and has accompanying parameters. Parameters so set are mirrored into this view's nuJanxContext, prefixed with "param.".

nuTransitionVisibleClass
Instance of neutrino.OnInvisibleWalker used to walk DOM subtrees and call onInvisible() on any found

nuTransitionInvisibleClass
Instance of neutrino.OnInvisibleWalker used to walk DOM subtrees and call onInvisible() on any found

nuPage
Instance of neutrino.Page which is this View's parent page. If this View is actually a Page, this is undefined.

Page

Page inherits from View.

On calling superclasses: Pages that override methods in the Neutrino Page class must, under normal circumstances, call the corresponding method in Page in addition to doing anything else.

(public methods)

configure(inKey, inElement, inPage)
Overridden from View. Sets up nuJanxContext to inherit from Application's nuRootJanxContext. Sets "page" data off nuElement to point to "this". Instantiates nuViews.

getView(inViewKey)
Retrieves the View referenced by the specified View key, if any.

hideView(inViewKey, inTransitionInvisibleClass)
Starts the invisible transition for the View specified by the key. Uses the specified transition-invisible class if any, otherwise uses the standard one.

isQuiescent()

Returns whether all the Views in the page are quiescent. Currently, View.isQuiescent() is hardwired to return true, as quiescence is deprecated. Will be removed.

loadView(inViewName, inViewKey, inElement, inFlags)
Calls the loader to load view resources, adds event listeners for lifecycle management, and adds the view to nuViews.

showView(inViewKey, inParams, inTransitionVisibleClass)
Starts the visible transition for the View specified by the key. If any parameters are specified, set them on the target View. Uses the specified transition-visible class if any, otherwise uses the standard one.

unloadView(inViewKey)
Unloads the CSS for the View specified by the key, and removes the View from nuViews. Unloading is currently unsupported.

(private methods)

addViewEventListeners(inView)
Adds event listeners for animation start and end (for all supported browsers) to point to onAnimationEnd() in the View class and onViewAnimationEnd() in the Page class, enabling View visibility state management.

onViewAnimationStart(inEvent, inView)
Called when an animation pertaining to a View managed by this Page starts.

onViewAnimationEnd(inEvent, inView)
Called when an animation pertaining to a View managed by this Page ends. Used to detect the end of visible and invisible animations, notifying the corresponding View of visibility changes.

(properties)

nuViews
Object containing the currently loaded Views for this Page, indexed by view key.

Initial Page

There are a few ways of setting the initial page:

Put a “nu-start-page” attribute on the body element

- Position a “menu-type” View outside the page container whose elements can set the current page via nu-action/setpage.
- Call *gApplication.setPage()* with the page key, after the call to *gApplication.start()* in main(), providing that *gApplication.setInitialPage()* did not decide on a page to load.

Browser Neutral CSS

At time of writing, the CSS3 syntax has yet to be ratified by the W3C, resulting in fragmentation. Therefore, in order to be assured of wide browser compatibility, CSS authors currently have to write four separate browser-specific directives instead of one browser-neutral one.

However, Neutrino helps out here by offering to convert the future standard W3C syntax into the browser-specific dialect appropriate to the requesting user agent. In this way, CSS authors can write standards compliant code.

In addition, Neutrino currently rewrites styles containing “vw” and “vh” units to incorporate the browser’s viewport dimensions. Note that the computed dimensions do not include an allowance for any kind of chrome around the viewport.

The Neutrino CSS loader does this automatically for its dynamically loaded CSS classes. For manually included styles, simply use “nu-href” instead of “href” on <link>, and the "nu-browser-neutral-css" class on <style>.

Standard Transitions

Neutrino includes several animations that can be used as transitions, either in classes with more specificity than the default ones or in places where override classes can be specified—for example, in `showview` actions.

nu-view-fade-in

A simple linear fade in from opacity 0 to opacity 1.

nu-view-fade-out

A simple linear fade out from opacity 1 to opacity 0.

nu-view-slide-in

A linear horizontal slide right from -500px to 0px.

nu-view-slide-out

A linear horizontal slide left from 0px to -2000px.

nu-view-slide-out-to-left

A linear horizontal slide left from 0px to -1200px.

nu-view-slide-out-to-right

A linear horizontal slide right from 0px to 1200px.

nu-view-slide-out-to-top

A linear vertical slide up from 0px to -1200px.

nu-view-slide-out-to-bottom

A linear vertical slide down from 0px to 3200px.

nu-view-slide-in-from-top

A linear vertical slide down from -1200px to 0px.

nu-view-fall-forward

A rotation around the X axis producing a "falling" effect.

nu-view-open-to-left

A rotation around the Y axis producing a effect similar to opening a page going forward through a book.

nu-view-open-to-left

A rotation around the Y axis producing a effect similar to opening a page going backward through a book.

nu-view-close-to-left

A rotation around the Y axis producing a effect similar to closing a page going forward through a book.

nu-view-close-to-right

A rotation around the Y axis producing a effect similar to closing a page going backward through a book.

Log Flags

Neutrino's logging system is activated by adding the “nu-log” parameter to the page URL.

The parameter contains a list of logging keywords, separated by a comma.

The currently accepted logging keywords are:

Keyword	Description
all	Turn on all the logging, equivalent to passing all the recognized keywords.
action	Nu-action parsing and dispatch.
analytics	Analytic events.
app	Application-scope activity.
assetloader	Assetloader (preload.json) activity.
async	Asynchronous dynamic activity.
cache	Additions and expirations from the app-wide cache.
components	Activities from components (a “catch-all” tag).
css	Rewriting browser neutral CSS into browser specifics.
janx	Dynamics.
view	View lifecycle events.