DHTML Developer Checklist

DHTML Developer Checklist

Introduction

Web 2.0-style user interfaces, built using DHTML, JavaScript, and AJAX, have become increasingly popular within community source applications in recent months. While these technologies do offer an exciting opportunity to use more recognizable and direct user interface idioms, they also represent a potential barrier to Web accessibility and usability. Developers who create new user interfaces using these technologies need to be aware of the challenges they present, and understand the techniques that can help ensure a usable experience for all users.

This checklist is intended to provide developers with a quick overview of a few of the most important issues to watch out for when developing DHTML and AJAX-based user interfaces. It may also be suitable as a starting point for thinking about UI technical governance, ensuring all new user interfaces address these issues before being included in a release. Fluid is committed to helping community source developers implement these strategies in their own code, and is working to grow a library of well-designed user interface components that will help to ease the burden of creating accessible, reusable designs.

On This Page

Still need help?

Join the infusion-users mailing list and ask your questions there.

Checklist Summary

Developers working on DHTML user interfaces should keep in mind, at the very least, the following techniques:

1. Keyboard support: everything that works with the mouse should also be usable with the keyboard.
    - Follow ARIA conventions when using the tabindex attribute
    - Use consistent and familiar keyboard controls (Tab key, Arrow keys, Enter, and Spacebar)
2. Portal friendliness: Ensure JavaScript is self-contained and properly namespaced
    - Don't use the global namespace
    - Avoid overriding built-in types
    - Generate unique element ids in your markup
3. Label everything: use ARIA to convey UI semantics

Checklist in Detail

1. Keyboard Support: everything that works with the mouse should also be usable with the keyboard

Comprehensive keyboard controls are often overlooked by developers, but are one of the best ways to improve the overall accessibility of your design. A large number of assistive technology users depend on the keyboard to access everything on the web; keyboard accessibility also improves efficiency and convenience for all users.

The basic rule is this: *everything* that is available with the mouse should also be usable by keyboard. This includes any kind of direct manipulation tasks such as drag and drop, as well as hover effects, tooltips, and similar behaviour.

Suggested Keyboard Behaviour for DHTML Widgets

- The **Tab key** should provide focus to the widget as a whole. For example, tabbing to the Lightbox puts focus on the overall thumbnail container.
- The **arrow keys** should allow for selection or navigation within the widget. For example, using the left arrow key in the Lightbox shifts focus to the next image thumbnail.
- When the widget is not inside a form, both the **Enter and Spacebar keys** should select or activate the control.
- Within a form, the **Spacebar** should select or activate the control, while the **Enter** key should submit the form's default action.
- If in doubt, mimic the standard desktop UI behaviour of the control you are creating.

Technical Details

The *tabindex* attribute allows you to specify which items are focusable using the Tab key. Widgets should be given a tabindex value of 0 or higher. Sub-elements within the widget, such as menu items, selection items, and so on, should be given a tabindex value of -1. This will remove them from the default tab order and allow you to provide custom JavaScript handlers for arrow key events. Fluid is working on JavaScript libraries that will make adding your own keyboard handlers for this behaviour much easier, but in the meantime we suggest reusing widgets from accessible toolkits such as Dojo wherever possible.

Example Markup Using Tabindex

Here's an example of how you would use the tabindex attribute for a typical menubar interface:

```
<!--
 * The menu bar itself will be focussable with the Tab key.
 * All of the menu items themselves should be accessible using the arrow keys by adding JavaScript handlers for
these keys.
 * Notice that I have to explicitly set the tabindex of focussable items such as <a> tags to "-1" to assure
they aren't available in the tab order.
 -->
<li id="menubar" tabindex="0">
  <li id="fileMenu">File
      <ul>
        <li><a href="/open" tabindex="-1">Open File...</a></</li>
        <li><a href="/save" tabindex="-1">Save As...</a></li>
        <li><a href="/print" tabindex="-1">Print...</a></li>
      </ul>
  </li>
  <li id="editMenu">Edit
      <ul>
        <li><a href="/cut" tabindex="-1">Cut</a></li>
        <li><a href="/copy" tabindex="-1">Copy</a></li>
        <li><a href="/paste" tabindex="-1">Paste</a></li>
      </ul>
  </li>
</li>
```

For more detailed information about keyboard navigation, check out the Mozilla Developer Center's Key-navigable custom DHTML widgets article.

To test your browser's handling of the tabindex attribute, consult the Tabindex Focus Navigation Tests article.

2. Portal friendliness: all JavaScript code should be self-contained and make rigorous use of namespaces

Here are some techniques to make your code more portal-friendly:

- Namespace everything: don't use the global namespace
- Don't override the behaviour of built-in types such as Object or Array
- Generate fully unique element IDs for your markup
- Use JavaScript toolkits that follow these techniques

Namespacing: Portals represent a particularly complex environment for browser-based scripting. The core issue is the fact that portlets are intended to be fully self-contained fragments of markup, independent from the rest of the page. This places significant extra demand on the architecture of your JavaScript code; it must be fully namespaced and isolated from other, potentially arbitrary, JavaScript code running simultaneously.

In other words, don't place any variables in the global namespace, except for a single variable that defines your own personal namespace. Also, avoid extending or overriding the behaviour of native JavaScript objects such as Array or Object, as this will unexpectedly affect all other JavaScript code running on the page. As a rule, avoid JavaScript toolkits that don't follow these conventions; using the global namespace or extending built-in types will wreak havoc in a portal!

Multiple Placements: Another complication is the issue of multiple placements of the same portlet. In a portal, more than one instance of a portlet can appear on a single page. This requires all markup to be unique to each instance of the portlet such that there won't be conflicts or ambiguity when manipulating DOM elements by id. In other words, ensure that your generated markup includes a unique prefix for all element ids. This additional level of markup namespacing can be provided by tools such as the Portlet taglibs for JSP, or the built-in functionality of a portlet-friendly presentation framework such as RSF.

Versioning: The problem of versioning is a significant one in portals. Different portlets may well depend on different versions of a particular JavaScript library. In an ideal case, library dependencies could be expressed on a per-portlet basis. However, all current JavaScript toolkits don't provide sufficient support to enable the use of more than one toolkit version at a time. The best solution, in this case, is to encourage community consensus on toolkit versions and maintain all portlets against a particular, agreed-upon version of the toolkit.

3. Label Everything: Attach ARIA user interface metadata to all your DHTML elements

The biggest accessibility problem with DHTML is a lack of semantics. Most Web 2.0-style user interfaces bring richer controls similar to those found on the desktop. The problem is that HTML doesn't provide enough built-in information to properly describe these types of controls to users of assistive technology.

Current assistive technology is well-equipped to work with common UI controls found on the desktop. Indeed, the fact that many DHTML widgets more closey mimic their desktop counterparts provides the opportunity to make the web much more accessible and familiar than it has previously been. The challenge is ensuring that the assistive technology knows enough about the meaning of the markup to correctly represent these controls. In other words, we need to add additional information to the markup in order to ensure that assistive technologies understand the characteristics and behaviour of our DHTML user interfaces.

The Problem of Insufficient Semantics

At the core of the DHTML accessibility problem is a lack of UI semantics; HTML doesn't give us enough richness to properly describe user interfaces to assistive devices. DHTML controls are often built using generic grouping elements such as `<div>` and `<span>`, or by overloading existing types such as lists. These tags don't provide us with any further indication of what the control does, or what properties and states it possesses. So imagine a tab-based navigation bar in a web application. The markup might look something like this:

```
<ul id="tabNavigation">
  <li class="tab">Home</li>
  <li class="tab">Books</li>
  <li class="tab">Music</li>
  <li class="tab">DVD</li>
</ul>
```

While the markup may look and feel like a group of tabs visually, we have no explicit labels or semantics to communicate this behaviour to the browser and assistive technology.

Using ARIA to Add Semantics

ARIA's goal is to fill in the semantic gap missing from HTML. Created by the W3C, ARIA is a specification describing additional markup for Accessible Rich Internet Applications. It allows you to add specific attributes to your XHTML elements in order to help specify the roles, properties, and states of a particular interface control. Roles describe the type of user interface control, for example: menus, tabs, sliders, grids, trees, and tool tips. States and properties describe the various modes or settings a control can have, for example: disabled, selected, busy, readonly, etc.

Using ARIA, our tab example above would look like this:

```
<ul id="tabNavigation" role="tablist">
  <li class="tab" role="tab">Home</li>
  <li class="tab" role="tab">Books</li>
  <li class="tab" role="tab">Music</li>
  <li class="tab" role="tab">DVD</li>
</ul>
<div role="tabpanel">
  The contents of the tab panel go here.
</div>
```

The addition of the namespaced ARIA roles for tabs, tablists, and tabpanels will be sufficient to convey the meaning of the interface to users of assistive technology, ultimately making this interface far clearer and more accessible to all users.

While ARIA takes a bit of work to understand, it is fundamentally required to make DHTML user interfaces accessible to a broad range of users. The added benefit is that ARIA provides further semantic richness to your markup that can be used for CSS selectors, DOM manipulation, and so on.