AccDC - A Fast Route to Access for Web Applications

Presenter: Bryan Garaventa

Founder and Developer (WhatSock.com)

SSB BART Group is the primary sponsor of WhatSock at CSUN.

SSB's Accessibility Management Platform (AMP) is a turn-key, web-based solution that provides enterprise-class organizations access to auditing, testing, reporting, training, and best practices to rapidly conform to Section 508, the Web Content Accessibility Guidelines (WCAG), the Americans with Disabilities Act (ADA), and other leading accessibility requirements.

This outline, including all related coding examples, can be downloaded from:

http://whatsock.com/csun/2014

AccDC - Accelerated Dynamic Content

AccDC is a free development resource for international businesses, organizations, and academic institutions wishing to incorporate Accessible Innovation within future web technologies.

"Accessible Innovation" refers to the creation of new technologies that include Accessibility as a core platform feature of development.

This is accomplished through the following:

1. The AccDC API is a JavaScript based Dynamic Content Management System that automates the rendering of dynamic content to ensure accessibility for screen reader and keyboard only users.
2. The AccDC Technical Style Guide is designed to provide reliable and consistent interaction designs that are accessible to the highest percentage of people possible, and to establish a baseline for Functional Accessibility that can be utilized, built upon, studied, and tested against.

3. AccDC Bootstrap is an HTML parser that renders advanced, accessible interactive controls using semantic HTML markup.

AccDC API

At its core, AccDC is a JavaScript based API that manages dynamic content objects, called AccDC Objects, by automating accessible rendering processes for Rich Internet Applications.

This works by acting as a standalone interface, or as a scaled down plug-in for jQuery, Dojo, or MooTools, that extends the functionality of each by tying into their respective rendering processes.

This normalization makes it possible to share UI components built using AccDC Objects between jQuery, Dojo, and MooTools, without requiring modification, since all such functionality ties into the same AccDC API commands.

AccDC Objects

The AccDC API manages all of the following rendering processes separately within each AccDC Object instance:

- Script flow control for sequential processing
- Rendering of HTML markup, DOM nodes, or content from AJAX queries
- Parent, child, and sibling relationship mapping
- Auto-positioning of layered content relative to specific DOM nodes
- CSS styling overrides for full visual customization
- Content overrides for full language customization
- Handler overrides for full event customization
- Rendering overrides for full behavior customization
- Global and sibling overrides for property and method prototyping
- Load latency for full programmatic configuration before rendering

Additionally, the AccDC API automates all of the following accessibility features for rendered dynamic content through AccDC Objects:

- Configurable beginning and ending boundary information for screen reader users

- Configurable ARIA heading structures for screen reader users
- Configurable dynamic Close link for screen reader and keyboard only users
- Configurable DOM insertion to manage reading order for screen reader users
- Configurable auto-focus positioning for screen reader and keyboard only users
- Configurable role and state information for screen reader users
- Invokable or automatic announcement of text or DOM node content for screen reader users

Accessible Component Modules

The Accessible Component Module collection is included within the AccDC Technical Style Guide.

An Accessible Component Module (ACM) is a fully functional; scalable; configurable; JavaScript based widget, that plugs directly into the AccDC API, and has been fully tested to ensure the greatest level of accessibility for the highest percentage of screen reader and keyboard only users possible.

Since each ACM uses only standard JavaScript and AccDC API commands, the same module code can be shared between jQuery, Dojo, and MooTools with equal functionality and accessibility without requiring modification.

The ACM collection includes all of the following interactive widget types:

- ARIA and Non-ARIA powered Accordions
- ARIA and Non-ARIA Tabs
- ARIA Data Grids (Read-only, Selectable, and Editable)
- ARIA powered Date Pickers
- ARIA Listboxes (Static, Dropdown, Single Select, Multiselect, and Sortable)
- ARIA Menus (Flyout and Vertical)
- ARIA Radio Buttons
- ARIA Sliders (Horizontal and Vertical)
- ARIA Toggles, Checkboxes, Links, and Buttons
- ARIA Trees
- Banners
- Carousels, Slideshows, and Wizards
- Drag and Drop (Keyboard Accessible)
- Footnotes
- Inline Form Field Validation Errors and Dynamic Help Tooltips

- Modals
- Popups
- Progress Bars
- Scrollable Divs
- Tooltips
- Web Chat and Dynamic Message Announcement (Instructional)

The accompanying AccDC Technical Style Guide (TSG) provides specific code level guidance for successfully implementing each ACM type, as well as the intended functionality, browser and screen reader differences, and ARIA related pitfalls to be aware of for each when applicable.

The Coding Arena, also included within the AccDC TSG, is an interactive training center and sandbox for experimental testing and development.

Engineers and Assistive Technology vendors can use the Coding Arena to test current levels of accessibility for specific ARIA Widget types that map specifically to the ARIA Roles Model documentation, and verify which browser and AT combinations best support this functionality in practice.

Moreover, the more familiar that engineers become with the keyboard functionality of specific widget types and how they interact with Assistive Technologies, the easier it will be for engineers to identify when similar widgets are not accessible through comparative analysis.

AccDC Bootstrap

AccDC Bootstrap is a separate JavaScript module that can optionally be run after the AccDC API and all relevant Accessible Component Modules have been loaded, which will then automatically convert standard HTML elements into interactive widgets that maintain the same level of accessibility for screen reader and keyboard only users.

This makes it possible to implement shared accessible widgets across unlimited site pages without requiring separate JavaScript configuration for each.

The module functions by automatically recognizing key class names in the markup, matching this class name with the corresponding Accessible Component Module, and then adding all widget functionality including keyboard and screen reader support.

The HTML5 'data-*' attribute is used to configure specific widget functionality such as accessible names for screen reader users, content locators, and behavioral flags, which is all included within the HTML markup. (The asterisk '*' refers to the name of the widget setting to modify.)

AccDC Bootstrap currently supports all of the following Accessible Component Modules:

- ARIA powered Accordions
- ARIA Tabs
- ARIA powered Date Pickers
- ARIA Menus (Flyout and Vertical)
- ARIA Toggles
- ARIA Trees
- Banners
- Carousels and Slideshows
- Footnotes
- Modals
- Popups
- Scrollable Divs
- Tooltips

Additionally, the AccDC Bootstrap module is fully customizable, and can be edited to add or remove ACM widgets as desired, or modify the functionality of supported widgets to fit any behavioral requirements.

Underlying Principles for Accessibility

In order to ensure the greatest level of accessibility for the highest percentage of people possible, all of the functionality within the AccDC API, the AccDC Component Modules, and AccDC Bootstrap, has been constructed with strict adherence to the following principles.

- Keyboard Accessibility
- Screen Reader Accessibility
- Cognitive Accessibility

Keyboard Accessibility

Keyboard Accessibility refers to full keyboard accessibility for all user types, with or without a screen reader running.

All functionality must be accessible from the keyboard, and the user interaction model for each widget must be intuitive.

If functionality cannot be activated from the keyboard using standard keyboard interaction commands such as the Tab, Arrow, and Enter keys, then the widget is not accessible.

In the case of a simulated button or link, ensuring that the element is focusable and that it can be activated by pressing Enter from the keyboard, will satisfy this requirement.

Similarly, simulated Checkboxes and Toggles must also be toggleable using the Spacebar.

Complex widget types however, such as simulated Listboxes, Menus, Tabs, Trees, Grids, Sliders, Radio Buttons, and ToolBars, require more comprehensive keyboard interaction designs.

In order to prevent confusion when screen readers process key events differently, such as when pressing Enter in Virtual Buffer mode will activate the 'click' event instead of the key event as expected, Keyboard Accessibility should always be tested with no screen reader running.

Screen Reader Accessibility

Screen Reader Accessibility is closely related to Keyboard Accessibility, however there are important differences that must be taken into account.

One of these, is the concept of textual equivalents.

In order for screen readers to convey what type of element has focus, and how a user is supposed to interact with it, a textual equivalent must be provided to convey this information.

This is typically handled automatically by the browser and the Accessibility API within the Operating System, by setting appropriate roles such as 'link' for A tags, 'heading' for H# tags, and so on for all standard HTML elements.

However, when simulated controls are built, such as when using a Span tag as a button, none of these role mappings are present, making the purpose of the element impossible for screen readers to detect.

This is where ARIA comes into play, providing the ability to set desired role mappings, plus state and property information, that will be detectable by screen readers.

A simple example of this is a Slider control that is keyboard accessible. It may be possible to tab to the Slider, then use the arrow keys to adjust the value, but none of this information will be conveyed to a screen reader user, making it impossible for them to detect the control as a Slider or to know which value is currently selected.

However, the addition of role='slider' will convey the element as a slider, and the addition and dynamic updating of aria-valuemin, aria-valuemax, aria-valuenow, and aria-valuetext accordingly will convey the correct state and value information for screen reader users.

Now, here is one of the most important pitfalls of using ARIA within interactive widget designs: All role mappings must precisely match programmatic focus movement, and all role mappings and programmatic focus movement must strictly adhere to the W3C ARIA Roles Model. Similarly, only supported states and properties should be used in combination with each role.

The reason being, if focus movement doesn't precisely match an element with a valid role attribute, then the role of the element will not be accurately conveyed to screen reader users.

For some interactive widget types such as Tabs, Listboxes, Grids, Menus, Comboboxes, and Trees, the aria-activedescendant attribute can be used to manage selection from the top level container element instead of setting focus on individual child nodes. Nevertheless, the elements that receive programmatic focus must include a valid ARIA role.

Also, the use of an invalid state or property, such as aria-pressed on an element with role='link', will not be recognized. (Aria-pressed is only valid on elements with role='button' for instance.)

It is critical to ensure that focus movement within Keyboard Accessibility always properly matches the placement of ARIA role attributes when designing simulated interactive widgets. Similarly, all relevant supported state and property attributes must be placed within the same element that contains a valid role.

Example:

****

What am I?

****

Answer: A focusable Toggle Button that is not pressed.

Since ARIA is not specific to screen readers, the proper use of ARIA as indicated above, will improve accessibility for other Assistive Technology types such as Voice Navigation and Screen Magnification software as support for ARIA increases in the future.

Another important aspect to be aware of, is the difference between JAWS and NVDA regarding 'focus' event triggering.

When using JAWS in Virtual Buffer Mode, it is possible to arrow down the page and read all page content without triggering the 'focus' handler on any of the element nodes. (The one exception to this is form fields when Auto Forms Mode is enabled.)

However, when using NVDA in Browse Mode (equivalent to Virtual Buffer Mode in JAWS), the act of reading content on the page will automatically trigger any attached 'focus' handlers.

This is especially important to keep in mind when constructing interactive widgets like Menus, Editable Grids, Listboxes, Radio Buttons, and Tabs, where 'focus' handlers may be used to trigger specific actions.

Cognitive Accessibility

Lastly, Cognitive Accessibility refers to the visual aspect of accessibility, such as the visual tracking of focus movement, sufficient color contrast for low vision users, intuitive reading and tab order, making sure that important information doesn't disappear when High Contrast mode is enabled, and providing audio textual equivalents for the hearing impaired such as captions when applicable.

Conclusion

The AccDC API is designed to provide the most reliable JavaScript interface possible for rendering dynamic content accessibly with equal results across jQuery, Dojo, and MooTools.

All of the interaction designs within the AccDC Technical Style Guide and Coding Arena combine the three principles, Keyboard, Screen Reader, and Cognitive Accessibility, in order to provide the most accessible component designs possible for engineers, educators, and AT vendors to compare and test functionality within a fully interactive sandbox environment.

AccDC Bootstrap provides a method for engineers to implement shared Accessible Component Modules across unlimited site pages using standard HTML markup to configure functionality, while at the same time ensuring the same level of accessibility demonstrated within the Coding Arena.

References

- AccDC API: http://whatsock.com
- AccDC Technical Style Guide: http://whatsock.com/tsg
- AccDC Bootstrap: http://whatsock.com/bootstrap
- The Roles Model | Accessible Rich Internet Applications (WAI-ARIA): http://www.w3.org/TR/wai-aria/roles
- Supported States and Properties | Accessible Rich Internet Applications (WAI-ARIA): http://www.w3.org/TR/wai-aria/states_and_properties
- WebAIM: Screen Reader User Survey #5 Results: http://webaim.org/projects/screenreadersurvey5/
- How Browsers Interact with Screen Readers and Where ARIA Fits in the Mix | SSB BART Group: https://www.ssbbartgroup.com/blog/2013/01/02/how-browsers-interact-with-screen-readers-and-where-aria-fits-in-the-mix/
- Why There are Only Two Ways to Make ARIA Widgets Programmatically Focusable for Screen Reader Users | SSB BART Group: https://www.ssbbartgroup.com/blog/2013/10/22/why-there-are-only-two-ways-to-make-aria-widgets-programmatically-focusable-for-screen-reader-users/
- The Importance of Keyboard Accessibility & Why ARIA Widgets Don't Work as Expected in Voice Navigation Software | SSB BART Group: https://www.ssbbartgroup.com/blog/2013/07/08/the-importance-of-keyboard-accessibility-why-aria-widgets-dont-work-as-expected-in-voice-navigation-software/

Contact

Bryan Garaventa

bryan.garaventa@whatsock.com

WhatSock.com