ARIA – Part One

"The Joy of ARIA"

February, 2016

Prepared by David O'Neill

Introduction

This article is Part One of a three-part series for those who wish to learn more about ARIA. The intended audience is web development professionals, including designers, coders, testers and user support specialists. The article assumes the reader has a basic understanding of HTML and is familiar with the use of client-side scripting and dynamic CSS. An appreciation of the unique challenges that people with disabilities face using the Web is also beneficial. Having said that, the articles presented in this series are self-contained and quite suitable for learning ARIA, regardless of the reader's Web programming experience.

Part One, "The Joy of ARIA", introduces the reader to ARIA and demonstrates how to get started with the Landmarks and Document Structure Roles to make a Website more accessible.

What exactly is ARIA?

ARIA is an acronym which stands for "Accessible Rich Internet Applications", which might best be considered as both a goal statement and a detailed specification for improving the accessibility of your Web application. We use ARIA as a means to make modern Internet Applications both **Accessible** and **Rich**. But what exactly do these terms mean?

> **A**ccessible: Meaning an Internet Application that can be perceived, operated, understood and compatible with all types of users, including those with disabilities.
>
> **R**ich: Meaning an Internet Application with interactive features, including dynamic content and complex controls that extend beyond the original intent of basic HTML elements.

ARIA is a technical standard which defines a framework for enhancing HTML with semantic markup which is needed in some Web applications to be "understood" by assistive technology.

More specifically, ARIA is a collection of HTML attributes used to define the accessible **Roles**, **Properties** and **States** of otherwise inaccessible HTML markup.

What ARIA Isn't

Before we dive further into the details of ARIA it is important to dispel the common myths and clarify some of the misconceptions.

ARIA is not a silver bullet. ARIA solves the specific need of providing sematic information for HTML elements where native semantics are insufficient for accessibility. ARIA is not intended as a substitute for existing best practices for accessibility. A thorough working knowledge of accessibility standards, such as WCAG 2.0, remains vital to building accessible applications. Further, ARIA by itself is insufficient to satisfy all the requirements of accessibility for the many rich controls found in modern web applications. ARIA itself provides no behavior. Behavior is supplied by a combination of the User Agent and the functionality coded by the website developer. The developer must be prepared to use ARIA in conjunction with CSS and JavaScript to support standard design patterns for keyboard operability, focus management and dynamic styles. Always remember, ARIA is simply part of the larger solution for web accessibility.

ARIA is not harmless. Of all the misconceptions floating around the web concerning ARIA, the claim that it is best to implement ARIA attributes when in doubt because "ARIA is harmless". Actually, ARIA can be quite disruptive! By design, ARIA Roles, Properties and States will override the native semantics of HTML elements to which they are applied. If done carelessly, accessibility can be adversely affected. ARIA is intended to add missing semantics when these semantics are not available. It is sometimes appropriate to use ARIA in conjunction with HTML elements that have similar native semantics, but this technique should only be used as part of a well-designed progressive enhancement strategy. In these cases, where ARIA semantics are not supported, the browser will have a "fail-safe" alternative in the native semantics of the HTML element. As a rule, never apply ARIA haphazardly and only use ARIA to override native HTML semantics if it is part of a tested progressive enhancement strategy.

ARIA is not redundant. You may have noticed that HTML5 has some new elements that overlap with the ARIA Roles ontology. Many folks believe that this overlap makes ARIA redundant and perhaps unnecessary. This is a false assumption. There are many Roles, Properties and States defined by ARIA that are not defined by HTML5. Further, developers have a propensity to create custom HTML elements despite the availability of native HTML elements which are intended for the same purpose. These custom elements need ARIA to supply the semantics that are missing for accessibility.

In summary, ARIA is a valuable extension of HTML5 that requires careful planning and dedicated efforts to implement properly.

WAI-ARIA History

The current WAI-ARIA 1.0 specification has a long history dating back to two former W3C Working Groups which were started in 2007:

- Roles for Accessible Rich Internet Applications (WAI-ARIA Roles)
- States and Properties Module for Accessible Rich Internet Applications (WAI-ARIA States and Properties)

In 2008, the efforts of these groups were united under the Protocols & Formats Working Group of the Web Accessibility Initiative to create a single specification called "Accessible Rich Internet Applications (WAI-ARIA) Version 1.0". After six years, and several working drafts, WAI-ARIA 1.0 became an official Recommendation of the W3C on March 20th, 2014 and was incorporated into the W3C's HTML5 Recommendation on the October 24, 2015. Though it took many years to publish the WAI-ARIA 1.0 Recommendation, the underlying goal of the specification has never changed. ARIA aims to enable accessibility in modern, dynamic Web Applications.

HTML, Accessibility API and Assistive Technology

To better understand the problem that ARIA solves, we need to explore how the Accessibility API of most operating platforms exposes information about HTML elements to Assistive Technology. Most HTML elements found in the Document Object Model (DOM) of a Web page are added as a node to an "Accessible Tree". Each node of the Accessible Tree contains accessibility information including the Role, Properties and State of the related HTML element. This information collectively represents the element's "semantics". Semantics can be native, explicit or even implicit. Most HTML tags have native semantics which are directly mapped within the Accessible Tree by the Accessible API. ARIA is used to define additional accessibility semantics which can be explicitly applied to HTML elements. These explicit semantics generally supplement, and in some cases override, the native semantics of the HTML element. Finally, since ARIA support pre-dates HTML5, many of the new HTML5 tags have implicit semantics based on similar ARIA Roles. It is important for the ARIA developer to be aware of all three forms of semantics.

Let's examine how native semantics enable accessibility in a static HTML example. As we have just learned, core HTML tags such as `<p>`, `<h1>`, `<ul>`, `<li>`, `<a>`, `<img>` and `<table>` have native semantics, including a Role, which define their purpose. A `<p>` tag is used to contain a paragraph of text while an `<h1>` tag contains text which is treated as a heading at level 1. These native semantics are mapped automatically through accessibility APIs so that they can be

accessed, “understood” and leveraged by assistive technology. When these HTML tags are used properly by Web developers, assistive technology will greatly enhance the accessibility of the underlying HTML content. For example, a screen reader may offer users the ability to navigate by Roles such as heading-by-heading or paragraph-by-paragraph.

HTML elements also support various Properties which are expressed in the form of attribute/value pairs. Properties are used to define the characteristics of an HTML element and are often essential for accessibility. A perfect example is associating a `<label>` element with an `<input>` element using the “`for`” and “`id`” Properties as shown below;

```
<label for=”fname”>First Name: </label>
<input type="text" id="fname"/>
```

The “`for`” attribute of the `<label>` element explicitly relates to the “`id`” attribute of the `<input>` element through the value of “`fname`”. These relational Properties are essential for assistive technology such as a screen reader. When focus is placed on the `<input>` element, the screen reader will announce the Role of “text input” followed by the label “First Name:”. If this relationship were not explicitly expressed through the Properties, the burden of determining the purpose of the `<input>` element would fall upon the user of assistive technology.

A State is a special type of attribute/value pair used to express characteristics of an HTML element that typically changes with user interaction or as a result of a process within the Web application. A good example would be an `<input>` element that can be disabled, or perhaps checked. In each case, HTML provides attribute/value pairs which are mapped to the Accessibility Tree by the Accessibility API for use by assistive technology. This is critical because it is common for the State of an element to be represented visually, however the State must also be translated into alternative forms such as speech.

Let’s pull the concepts of Role, Properties, and State together in one simple static HTML example.

```
<input type="checkbox" id=”consent” name="consent" checked=”true”>
<label for=”consent”>I agree with the terms</label>
```

Using the semantics found in the Accessible Tree, a screen reader would announce the following phrases once focus is placed on the checkbox:

“I agree with the terms”, “Checkbox”, “Checked”

This accessible experience was enabled entirely from the native Roles, Properties, and State of the HTML elements!

But what happens when we push the boundaries of basic HTML using CSS and JavaScript to create web pages with dynamic interfaces that look and feel more like desktop applications? Dynamic CSS and asynchronous JavaScript enable web developers to create incredibly rich controls such as tabs, accordions, fly-outs, menus, modal dialogs, expandable trees and

progress bars. These are all built from core HTML elements, but the presentation and behavior patterns are very different. Are the native Roles, Properties and States of standard HTML elements sufficient to support the **A**ccessibility needs in these modern **R**ich **I**nternet **A**pplications? The answer was no, not until WAI-ARIA.

Components of ARIA

ARIA extends the native semantics of HTML with a specific set of Roles, Properties and States used to enhance accessibility. There are many Roles, many Properties and many States to consider when implementing ARIA. The choices you make can't be arbitrary. WAI-ARIA is a formal ontology with a taxonomy that defines how the Roles, Properties and States relate to each other. The ARIA standard is actually a hierarchy of "classes" with specific inheritance chains for Roles, Properties and States.

ARIA Roles

At the core of WAI-ARIA is a collection of Roles which are divided into four categories;

1. Abstract Roles
2. Landmark Roles
3. Document Structure Roles
4. Widget Roles

Abstract Roles are foundational and are never used directly by Web developers. Instead, ARIA provides Web developers with usable concrete Roles that are derived from one or more of these Abstract Roles. The abstract roles defined by the ARIA 1.0 ontology are;

- `command`
- `composite`
- `input`
- `landmark`
- `range`
- `roletype`
- `section`
- `sectionhead`
- `select`
- `structure`
- `widget`
- `window`

Landmark Roles are concrete Roles that are derived from the abstract `landmark` Role base. They are used to identify navigational regions within your Web page. The current list of Landmark Roles includes;

- `application`
- `banner`
- `complementary`
- `contentinfo`
- `form`
- `main`
- `navigation`
- `search`

Document Structure Roles describe the structures that organize the content found on a Web Page. They are concrete Roles that are derived from a combination of abstract and other concrete bases. The current list of Document Structure Roles includes;

- `article`
- `columnheader`
- `definition`
- `directory`
- `document`
- `group`
- `heading`
- `img`
- `list`
- `listitem`
- `math`
- `note`
- `presentation`
- `region`
- `row`
- `rowgroup`
- `rowheader`
- `separator`
- `toolbar`

Widget Roles are used to create the more complex user interfaces and typically require dynamic styles and client scripting to support their standard design patterns. Widget Roles can be standalone or composite. Widget Roles will be explored in more detail in Part 2 of this ARIA series.

Assigning an ARIA Role to an HTML element is rather straight forward. Simply use the name of the Role as the value for the `role` attribute on an HTML element. For example, we can assign the `navigation` Role to the un-ordered list element with the following HTML code;

```
<ul role="navigation">
```

We can use this very same approach to assign any ARIA Role to any HTML element.

Assigning an ARIA Role is not necessarily the same as implementing an ARIA Role. To properly implement an ARIA Role, we often need to implement its supported Properties and States.

ARIA Properties and States

Every ARIA Role will also have Properties and States which are classified as Required, Supported or Inherited. For example, the Properties and States of the `banner` Role are all inherited from the `landmark` abstract role. The `banner` Role, like all Roles derived from `landmark`, will therefore support the following Inherited Properties and States;

- `aria-atomic`
- `aria-controls`
- `aria-describedby`
- `aria-dropeffect`
- `aria-flowto`
- `aria-haspopup`
- `aria-label`
- `aria-labelledby`
- `aria-live`
- `aria-owns`
- `aria-relevant`
- `aria-busy (state)`
- `aria-disabled (state)`
- `aria-expanded (state)`
- `aria-grabbed (state)`
- `aria-hidden (state)`
- `aria-invalid (state)`

Supported Properties and States are applicable to a specific Role and any child Roles derived from it. For example, the `slider` Role inherits the Properties and States of the `input` and `range` Widget Roles, but also has one Supported Property called `aria-orientation`. Use of this Supported Property is illustrated below;

```
<div role="slider" aria-orientation="horizontal" …>
```

The `slider` Role also has several Properties which are required and must be implemented by the Web developer. These Required Properties include;

- `aria-valuemax`
- `aria-valuemin`
- `aria-valuenow`

Finally, there are a set of Global Properties and States that can be applied to any HTML element, independently of the assigned ARIA Role. It might surprise you to learn that ARIA does not always require the use of a Role, however it should make sense considering the goal of ARIA is to supplement HTML with accessibility semantics when needed. There are many cases where the native semantic Role of an HTML element would simply benefit from additional ARIA Properties and States. The Global Properties and States include;

- `aria-atomic`
- `aria-busy (state)`
- `aria-controls`
- `aria-describedby`
- `aria-disabled (state)`
- `aria-dropeffect`
- `aria-flowto`
- `aria-grabbed (state)`
- `aria-haspopup`
- `aria-hidden (state)`
- `aria-invalid (state)`
- `aria-label`
- `aria-labelledby`
- `aria-live`
- `aria-owns`
- `aria-relevant`

The complete list of Properties and States for a specific ARIA Role is the distinct union of Global, Required, Supported and Inherited Properties and States. Always take the time to study the hierarchy of an ARIA Role and its complete list of Properties and States before applying it to your HTML elements. All too often beginner ARIA developers misapply Roles and misuse or omit Properties and States. The results are unpredictable and generally inaccessible HTML!

You may be wondering, what is the actual difference between an ARIA Property and an ARIA State? Fundamentally they are the same. Properties and States are both used to represent data values that are essential for establishing accessible semantics of our HTML elements for use by assistive technology. They are both implemented as simple HTML attribute/value pairs. They even share the same `aria-` prefix and present identically within your HTML markup. So how are they different?

The primary difference lies between the intent of an ARIA Property versus the intent of an ARIA State. ARIA Properties tend to feature values that are static and are often used to express relationships between elements in an HTML document. For example;

```
aria-labelledby = “labelID”
```

The `aria-labelledby` Property is used to associate one element as the label of another element in the same HTML document. This type of relationship is generally static and not likely to change during the life of the Web application. For this reason, `aria-labelledby` is considered a Property rather than a State. Please note, there is no restriction in ARIA preventing dynamic changes to Property values. Any reference to the static tendency of Property values is simply a generalization to help ARIA developers conceptualize the differences between a Property and a State.

An ARIA State typically expresses a value that is dynamic in nature, often changing in response to user interaction or by other application processes. For example;

```
aria-busy = “true”
```

The `aria-busy` State is used to inform assistive technology that an HTML element, or one of its child elements, is currently being updated and may not be ready for the user. This comes in handy in many situations where content is initially being loaded or asynchronously being updated in response to a user interaction. Once the HTML element is loaded or refreshed, the `aria-busy` property can be removed or reset to “`false`”. Generally speaking, the values expressed by an ARIA State will require dynamic management.

Property and State Values

Notice how we keep discussing Properties and States in terms of attribute/value pairs? We already discussed the convention of using the `aria-` prefix in the naming of the attributes that express ARIA Properties and States. We now need to explore the legitimate values that we can use with these attributes. Attribute values can be one of several possible types in ARIA;

- Boolean: Represented with either `true` or `false` (default value) values.
- Boolean Undefined: Represented with `true`, `false` and `undefined` (default value).
- Tristate: Represented with `true`, `false` (default value) and `mixed` values.
- ID: A reference to a valid ID of an element in the same document.
- ID List: A list of one or more ID references which are space delimited.
- Integer: A numerical value without a fractional component.
- Number: Any real numerical value.
- String: Unconstrained value type.
- Token: One of a limited set of allowed values.
- Token List: A list of one or more tokens which are space delimited.

The following example illustrates a Token value in action;

```
aria-live=”polite”
```

The ARIA Property uses an attribute named `aria-live` with a valid token value of “`polite`”. Other valid token values for the `aria-live` attribute are “`off`” and “`assertive`”.

The following example illustrates an ID value type in action:

```
aria-controls="tabpanelID"
```

The ARIA Property uses an attribute named `aria-controls` with an ID value of "`tabpanelID`". The value of "`tabpanelID`" must match the `id` attribute value of an existing HTML element within the current document.

The following example illustrates a Boolean value type in action;

```
aria-disabled="true"
```

The ARIA State uses the attribute named `aria-disabled` with a Boolean value of "`true`". The only other valid values for this type of attribute would be "`false`".

Be sure to understand the proper value types used for each of the ARIA Properties and States. This is particularly important for the Token and Token List value types.

How is ARIA useful?

When implemented properly, ARIA can make our Web applications far more accessible. This applies to both new design efforts and incremental integration of ARIA into existing implementations. Some of the specific benefits from the proper use of ARIA include;

- Content that is better organized and easier to navigate.
- Clarity to relationships between otherwise disconnected content
- Accessible notification and delivery of asynchronously updated content
- A more complete and consistent representation of complex user control semantics

Some of these benefits are easily enabled, others can be more challenging. Though ARIA itself is not a new specification, it has only recently been incorporated into HTML5. Both ARIA and HTML5 have varying levels of support across browsers and assistive technology. For this reason, ARIA is best implemented with a progressive enhancement approach.

Progressive enhancement starts by providing an accessible user experience based entirely on core HTML and progressively builds on that accessible experience in layers using ARIA, dynamic CSS and unobtrusive JavaScript. This layered progressive approach ensures that everyone has equal access to the base content and functionality of your Web application, while simultaneously enabling enhanced accessibility experiences in browser/assistive technology combinations which can support it.

Though there is no set formula for implementing ARIA progressively, one approach might simply be to organize your enhancements based on the distinct ARIA Role categories;

- Landmark Roles
- Document Structure Roles
- Widget Roles

The balance of Part 1 of this ARIA series explores accessibility enhancements using Landmark Roles and Document Structure Roles. We will explore accessible Widget Roles in Part 2.

ARIA Landmark Roles

Landmarks refer to a specific category of ARIA Roles which identify the most common regions found in classic Web page design. Can you think of a Web application that does not feature at least one group of navigation links? How about a main content area? Or a site search feature? Landmarks identify these regions and assistive technology uses them to help users comprehend and navigate Web pages more effectively.

Landmarks are the easiest feature of ARIA to implement and provide substantial accessibility benefits. Simply add Landmark Roles to the appropriate HTML elements in your page Templates. The following is a complete list of ARIA Landmarks with usage guidance, assistive technology impact and a simple code snippet for your reference.

banner

A region that contains mostly site-oriented content, rather than page-specific content.

`role="banner"` – Almost every website has some common content that appears on every page, usually positioned at the top spreading along the width of the viewport. This common content usually includes a logo image and perhaps a slogan expressed as text, however there are no requirements or restrictions regarding the actual content used. The HTML element containing this content should be identified with the `banner` Landmark. In HTML5 the <header> element is likely used for this content. Though the native semantics of the HTML5 `<header>` element are similar to the semantics of the `banner` Role, using the `banner` Landmark is recommended as part of a progressive enhancement strategy. Remember, ARIA Landmarks actually pre-date HTML5 section elements and tend to offer greater support in older browsers. The following code snippet would be used in an HTML5 implementation;

```
<header role="banner"> … </header>
```

If you are not developing using HTML5, you should apply the `banner` Landmark role to the containing `<div>` element as illustrated in the following code snippet:

```
<div role="banner">
```

As a rule, there should be only one banner landmark per `document` Role and one `banner` Landmark per `application` Role found on a Web page.

Most current assistive technologies provide strong support of the `banner` Landmark. Screen Readers will typical identify `banner` Landmarks and HTML5 `<header>` elements by announcing “banner” or “banner landmark” followed by the content contained by the element. In some cases, screen readers may announce both the HTML5 `<header>` element and the ARIA `banner` Landmark, resulting in “header banner” or “header banner landmark”. The specific text announced will vary across browser and assistive technology combinations.

complementary

A supporting section of the document, designed to be complementary to the main content at a similar level in the DOM hierarchy, but remains meaningful when separated from the main content.

`role=”complementary”` – Many Web page designs feature a main content surrounded by secondary supporting content where the main and secondary content are contained in separate HTML elements. The `main` Landmark is used on the HTML element containing the main content and the `complementary` Landmark is used on the HTML elements containing secondary content that is in fact relevant to the main content. For example, a Web page may feature a main area to display central content on a particular subject with a side panel displaying a list of related articles directly related to the subject. An HTML code snippet, including the ARIA Roles;

```
<div role="main">…central content…</div>
<div role="complementary">…related article list…</div>
```

It should be noted that the implicit ARIA Role of an HTML5 `<aside>` element is `complementary`, however the implicit ARIA Role of the `<aside>` element can be overridden by applying one of several other legitimate ARIA document roles for this element type, including `article` and `note`.

Most current assistive technologies provide strong support of the `complementary` Landmark. Screen Readers will typically identify `complementary` announcing “complementary” or “complementary landmark” followed by the content contained by the element.

contentinfo

A large perceivable region that contains information about the parent document.

`role="contentinfo"` – Almost every Web page has an area at the very bottom of the page reserved for content such as footer links, credits and a copyright statement. The HTML element containing this content should be assigned the `contentinfo` Landmark. There can be only one `contentinfo` Landmark per `document` and there can be only one `contentinfo` Landmark per `application`.

HTML5 provides a `<footer>` tag which is often used to contain similar content. Sticking to our progressive enhancement strategy, the `contentinfo` Landmark can and should be applied to the final `<footer>` element on the page. Do not apply the `contentinfo` Landmark to other `<footer>` elements that may appear in other sections within a `document`, as that would violate the one `contentinfo` Landmark per `document` rule. The following HTML5 code snippet illustrates the best practice for the Role;

```
<footer role="contentinfo">…footer content…</footer>
```

The ARIA `contentinfo` Landmark has somewhat stronger support by browsers and assistive technology than the HTML5 `<footer>` element. Screen readers that support both the `contentinfo` Landmark and the HTML5 `<footer>` element will generally announce them both as "contentinfo" or "contentinfo landmark".

form

A landmark region that contains a collection of items and objects that, as a whole, combine to create a form. See related search.

`Role="form"` – Forms require special attention to accessibility. The WCAG 2.0 specification has specific guidelines regarding the presentation and operability of form controls which should be followed to ensure accessibility. The ARIA `form` Landmark is used to identify a region within the page that contains related form controls. These form controls may include standard HTML form elements, custom scripted controls, and hyperlinks. If the `form` Landmark is used, authors should provide a visible label for the `form` Landmark using the `aria-labelledby` Property. If the purpose of the form is to support search functionality, the `search` Landmark should be used instead of the `form` Landmark. A simple HTML example implementing the `form` Landmark is provided below;

```
<div role="form" aria-labelledby="LoginFormHeading">
<h1 id="LoginFormHeading">Login Form</h1>
…Form input elements…
</div>
```

If you are developing with HTML5 you should use the native `<form>` tag in conjunction with the ARIA `form` Role and `aria-labelledby` Property as illustrated below.

```
<form role="form" aria-labelledby="LoginFormHeading">…</form>
```

The ARIA `form` Landmark has somewhat stronger support by browsers and assistive technology than the HTML5 `<form>` element. Screen readers generally announce the `form` Landmark as “form” or “form landmark”.

main

The main content of a document.

`role=”main”` – One of the common accessibility features found Web sites is the classic “Skip to Main Content” link. This link is usually hidden from a visual perspective, but available to screen readers in the DOM. The purpose of this link is to provide users with an easier way to bypass repetitive content and reach the main content on a Web page. The `main` Landmark is intended to provide similar capability. The `main` Landmark identifies the primary content that is the central topic of the document. There should only be one `main` Landmark per document and only one `main` Landmark per application within and HTML page. Assistive technology that supports navigation by Landmarks will leverage the `main` Landmark as a non-obtrusive alternative for "skip to main content" links. An HTML code snippet, including the `main` and `complementary` Landmark Roles;

```
<body role=”document”>
<div role="main">…article markup…</div>
<div role="complementary">…related article list…</div>
</body>
```

navigation

A collection of navigational elements (usually links) for navigating the document or related documents.

`role=”navigation”` – Most Web pages have one or more sets of links that are used to provide primary or secondary navigation. HTML5 also provides the `<nav>` element for this purpose. A progressive enhancement strategy for accessibility would be to use both the HTML5 `<nav>` element and the ARIA `navigation` Landmark as illustrated below;

```
<nav role="navigation">
<ul>
<li><a href="#">Link1</a></li>
<li><a href="#">Link2</a></li>
</ul>
</nav>
```

search

A landmark region that contains a collection of items and objects that, as a whole, combine to create a search facility.

`role="search"` – The search Landmark is a specialize case of the `form` Landmark. If your form controls serve the purpose of a search function, use the specialized case of the `search` Landmark over the more general `form` Landmark. You should provide a visual label for your `search` Landmark using `aria-labelledby`. The following HTML example illustrates the proper use of the `search` Landmark.

```
<form role="search" id="form" aria-labelledby="searchHeadingId">
            <h2 id="searchHeadingId">Search Form</h2>
            <label for="searchInputId">Search Expression:</label>
            <input id="searchInputId" type="text">
            <input type="button" value="submit">
</form>
```

There is strong accessibility support for both ARIA `search` Landmark Role. Screen readers will typically announce either "search" or "search landmark" followed by the element referred by `aria-labelledby` Property. In the case of our example, JAWS would announce "Search Form" followed by "Search Region".

The final Landmark to discuss is the ARIA `application` Role. The `application` Role is currently classified as a Landmark Role in WAI-ARIA 1.0 recommendation. It has been reclassified to the Document Structure category in the working draft of the WAI-ARIA 1.1 specification. As part of that reclassification, the `application` Role will no longer be derived from the `landmark` abstract Role. Rather, it will be derived from the `structure` abstract Role. WAI developers should always use the current recommendation of the WAI-ARIA standard, which is the 1.0 release. The reclassification and new inheritance chain specified in the WAI-ARIA 1.1 for the `application` Role does not change its intent or primary implementation technique.

application

A region declared as a web application, as opposed to a web document.

`role="application"` – The `application` Role is perhaps one of the most misunderstood ARIA Roles because of the seemingly unpredictable behavior that it causes across various assistive technologies. Note the phrase "seemingly unpredictable". In actuality, the `application` Role is fairly predictable once you

understand its purpose with respect to the different “modes” in which assistive technology supports user interaction.

Assistive technology serves as a translator between the end user and an underlying application. An example would be a screen reader which provides speech and keyboard capabilities to enable a non-sighted user to interact with a Web browser. One of the many responsibilities of the screen reader is to present the content inside the Web browser to the user in a format that is both perceivable and operable. Screen readers handle the operable requirement by intercepting user keyboard events to facilitate document reading and navigation. This mode, often referred to as “virtual cursor mode”, works fine until the user encounters interactive content which directly requires use of the keyboard. When the user sets focus on interactive content a screen reader will switch out of “virtual cursor mode” and into a mode that allows all or some keyboard events to pass directly through to the browser application. This second mode is often referred to as “forms mode” or “application mode”. The actual number and names of the modes will vary from one assistive technology to another. For purposes of understanding the ARIA `application` Role, let’s simplify and assume there are only two;

1. Virtual cursor mode (intercepts and processes keyboard events)
2. Forms mode (passes keyboard events to the browser application for processing)

The need for a “forms mode” should be fairly obvious, but an example always helps. Assume we are navigating a Web page using a screen reader in “virtual cursor mode”. JAWS would allow us to navigate by Landmark using the r and “`shift + r`“ keys. What happens if the user encounters a form with a text input field while browsing? The screen reader must switch from “virtual cursor” mode to “forms” mode and allow the user’s keystrokes to pass through and reach the text input. Mode switches generally happen automatically in response to the type of HTML element that receives focus. Screen readers “know” when to switch out of “virtual cursor” mode into “forms” mode. The mode can also be manually toggled by the end user. Switching from one mode to another is usually accompanied by an audible signal.

Everything was fine with these modes, until the arrival of the infamous ARIA `application` role. The ARIA `application` Role explicitly directs assistive technology to switch from “virtual cursor” mode to “forms” mode, and it does a great job at it! The problem is in the use of the `application` Role by developers who do not fully understand the consequences of their actions. If you apply the `application` Role to an HTML element, you must ensure that all of the content contained by that element must be natively focusable and keyboard operable. A paragraph of text (`<p>` tag) is neither natively focusable nor keyboard operable. You should assume it will not be accessible when contained by HTML which has the ARIA `application` Role applied. A text input field (`<input type=”text”…>`) is natively focusable and is keyboard operable. You can

safely assume it will be accessible when contained by HTML which has the ARIA `application` Role applied. If you have text content that needs to be expressed within an HTML container of Role `application`, you must use the `document` Role on that text to make it accessible. More details on the `document` role are provided later in this article.

Here is a code snippet that shows the use of the ARIA `application` Role, including non-interactive text content that will be accessible using the `document` Role;

```
<div id="application" role="application" aria-labelledby="appId">
      <h2 id="appId">Registration Service</h2>
      <p tabindex="0" role="document">Focusable instructions</p>
      <label for=”fname”>First Name:</label>
      <input type=”test” id=”fname”>
</div>
```

When a screen reader reaches the outer element with the `application` Role applied, it will announce “application” followed by “Registration Service”. This is facilitated by the `aria-labelledby` Property. Typically, the user presses the `enter` key to access the content contained by the `application`. The native keyboard tab control will allow the user to navigate the focusable elements contained by the `application`. Note that the `<p>` element has a `tabindex` value of `”0”` and an ARIA Role of `document`. The `tabindex` setting makes the element focusable and the `document` role instructs the screen reader to switch back to “virtual cursor mode”, allowing the user to reach and read text contained by the paragraph element.

We will revisit the application Role in Part 2 of this series when we discuss implementing Widgets with ARIA.

We have officially covered all of the ARIA Landmark Roles. To recap;

- Landmarks identify the most common regions of a Web page
- Landmarks enable assistive technology to help users comprehend and navigate Web pages more effectively.
- Landmarks should be implemented with the newest HTML5 elements as part of a progressive enhancement strategy
- Use of the application Landmark Role should be reserved for HTML elements containing natively focusable and keyboard operable or Widgets that have been custom developed with such support (to be discussed in Part 2)
- Always refer to the base abstract landmark Role and to determine which ARIA Properties and States are valid for use with Landmarks

Document Structure Roles

The final topic for Part 1 of this ARIA series is ARIA Roles for Document Structure. Document Structure roles are similar to Landmarks in that they are used to organize content on a page. The primary difference is that Landmarks are intended for use in navigation while Document Structure Roles are not. The Document Structure Roles are generally considered non-interactive.

The specific Document Structure Roles are;

- `article`
- `columnheader`
- `definition`
- `directory`
- `document`
- `group`
- `heading`
- `img`
- `list`
- `listitem`
- `math`
- `note`
- `presentation`
- `region`
- `row`
- `rowgroup`
- `rowheader`
- `separator`
- `toolbar`

The reader is encouraged to independently review the definitions and usage of each of these Document Structure Roles. In this article, we will explore several of the most important Roles available to ARIA developers.

region

A large perceivable section of a web page or document, that is important enough to be included in a page summary or table of contents.

Superclass: `section`

`role="region"` – Not all of our page content falls neatly into a specific ARIA Landmark Role, however we still may want to organize the content into a "perceivable" container. This is the exact purpose of the `region` Role. Proper use of the region Role requires the developer to specify an accessible name using the `aria-labelledby` Property.

Example illustrating how the `region` Role can be used to identify an area on a page which contains key performance statistics.

```
<div role="region" aria-labeledby=”stat1Id”>
      <h2 id=” stat1Id”>Performance Statistics</h2>
      … Performance statistics content …
</div>
```

Most screen readers will treat a `region` Role as if it were a Landmark, allowing the user to quickly navigate to these regions on the page. Some assistive technology may integrate the naming element identified by the `aria-labelledby` property into a page summary or table contents feature.

document

A region containing related information that is declared as document content, as opposed to a web application.

Superclass: `structure`

`role=”document”` – Recall the discussion of the `application` Role? The purpose of the `document` Role is the exact opposite of the purpose of the `application` Role. The `document` Role instructs assistive technologies to switch to “virtual cursor mode”. If the primary content in a Web page is non-interactive text (which is generally the case), the `<body>` element should be assigned the `document` Role, but is not required. If your Web page is predominately a container for highly interactive components, consider using the `application` role on the `<body>` element. In these cases, you will need to use `document` Role on the non-interactive text elements contained within the `application` body.

Every `document` Role should have a title or label. If the entire page is treated with a `document` Role, the HTML `<title>` element will satisfy this requirement, otherwise a visible text element should be associated with the `document` Role using the `aria-labelledby` Property.

Example illustrating a `<body>` element which is assigned the `document` Role, setting the assistive technology into “virtual cursor” mode automatically. The assistive technology will switch when it reaches the `<div>` with the `application` Role.

```
<body role="document">
… Page content presented in virtual cursor mode …
<div role=”application” labelledby=”appID”>
      <h2 id=”docID”>Title for Application Content</h2>
      … Interactive content…
</div>
```

```
… More page content presented in virtual cursor mode …
</body>
```

Example illustrating a `<body>` element which is assigned the `application` Role, setting the assistive technology into “forms mode” automatically. The assistive technology will switch to “virtual cursor mode” when it reaches any element assigned the `document` Role.

```
<body role="application">
… Interactive Page Content presented in forms mode …
<div role=”document” labelledby=”appID”>
      <h2 id=”docID”>Title for Document Content</h2>
… Page content presented in virtual cursor mode …
</div>
… More interactive Page Content presented in forms mode …
</body>
```

Screen readers will announce “application” when they reach an HTML element with the Role of `application`. Screen readers will announce “document” when they reach an HTML element with the Role of `document`.

article

A section of a page that consists of a composition that forms an independent part of a document, page, or site.

Superclass: `document`, `region`

`role=”article”` - Self-contained content that has meaning independently of the page should be expressed with the `article` Role. HTML5 also provides the `<article>` tag for this purpose. Use them both as part of a progressive strategy. A common use of the `article` Role is for blog entries and therefor the Role supports nesting. The parent `article` could be a blog post or some other form of syndicated content. User comments or feedback could be organized using nested HTML elements with the `article` Role. The WAI-ARIA 1.1 working draft introduces a `feed` Role which will be a scrollable widget intended for use with a list of `article` elements.

The `article` Role is presently derived from both the `document` Role and the `region` Role. This means that assistive technologies should switch into “virtual cursor mode” when they encounter an `article`. Though the `article` Role is not designated as navigational (descendant of `landmark`) it is often treated as a navigable region by assistive technology.

The following HTML illustrates a basic example of a nested elements with `article` Roles;

```
<article role="article">
<h2>February Blog Post</h2>
<p>…blog content…</p>
<article role="article">
<h3>Comment Title</h3>
<p>…comment content…</p>
</article>
</article>
```

The article Role and the HTML5 `<article>` may not yet supported by all assistive technologies, however both techniques should be employed as part of a progressive strategy.

group

A set of user interface objects which are not intended to be included in a page summary or table of contents by assistive technologies.

Superclass: `row`, `rowgroup`, `select`, `toolbar`

`role="group"` – Many of the Widget Roles that will be discussed in Part 2 of this series will require organizing HTML elements in logical collections that are meaningful to the construction of the Widget itself, but irrelevant in terms of the content found on the page. The `group` Role can be used in nested elements.

An example HTML snippet using a `group` Role in an unordered list of elements with the `treeitem` Role is provided below;

```
<ul role="group">
<li role="treeitem">Group Item 1</li>
<li role="treeitem">Group Item 2</li>
<li role="treeitem">Group Item 3</li>
</ul>
```

The above code snippet is intended solely for the purpose of illustrating the `group` Role. Implementing a `tree` Widget is an independent topic which will be discussed in more depth in Part 2 of this ARIA series.

presentation

An element whose implicit native role semantics will not be mapped to the accessibility API.

Superclass: `structure`

`role="presentation"` - The `presentation` role is used to remove the native semantics of an HTML element from the Accessibility Tree. This includes the implied Role, implied Properties and implied State of the element. For example, the following two HTML snippets would result in the identical semantics and content exposure;

```
<h1 role="presentation">Heading Text</h1>
<span>Heading Text</h1>
```

As you can see, removing the semantics of an element with the `presentation` Role will not affect the exposure of the actual content contained by the element.

If the element has required owned elements (elements that are required for the element to be valid) the implied Role, implied Properties and Implied State of the required owned elements will also be removed from the Accessibility Tree. The following example illustrates two code snippets with identical semantics and content exposure;

```
<ul role="presentation">
<li>Item 1</li>
<li>Item 1</li>
</ul>

<span>
<span>Item 1</span>
<span>Item 1</span>
</span>
```

The `presentation` Role will extend to any of the labeling elements associated with the element. This applies to HTML elements with native label elements, such as a `<table>` and `<caption>` element. Applying the `presentation` Role to the `<table>` will also result in the `presentation` Role being applied to its `<caption>`.

The `presentation` Role will cause assistive technology to ignore the non-global supported ARIA Properties and States applied to the HTML element, but will not affect the exposure of any Global ARIA Properties and States that are applied.

The `presentation` Role will have no effect on focusable elements. This is to ensure that the element remains fully perceivable and operable.

According to the WAI-ARIA 1.0 specification, the presentation role should be limited to a few uses;

- An element whose content is completely presentational (like a spacer image, decorative graphic, or clearing element);
- An image that is in a container with the `img` role and where the full text alternative is available and is marked up with `aria-labelledby` and (if needed) aria-describedby;

- An element used as an additional markup "hook" for CSS; or
- A layout table and/or any of its associated rows, cells, etc.

The `presentation` Role should be used carefully in a progressive enhancement strategy.

ARIA by Example

The best way to learn ARIA is through example. Appendix A illustrates an HTML document that implements HTML5, ARIA Landmarks and a number of the ARIA Document Structure Roles discussed in this article. The reader is encouraged to use the source code provided with a screen reader to experience the impact of ARIA on accessibility.

1. Download a trial copy of JAWS
2. Create your own HTML file from the provided source
3. Load your HTML document into Firefox or Internet Explorer
4. Navigate from top to bottom of the page using the arrow keys
5. Try the tab key
6. Navigate by Landmark (R and Shift-R in JAWS during Virtual Cursor Mode)
7. See if you can tell the difference between the document and application Roles
8. Make changes to the Roles, Properties and States
9. See what happens when you misapply ARIA

The best way to learn is by doing!

Appendix A – ARIA by Example

HTML5 Source

```
<!DOCTYPE HTML>
<html lang="en">
<head>
    <title>ARIA by Example - Part 1 Landmarks and Document Structure</title>
    <link rel="stylesheet" href="ARIAStyles.css" type="text/css">
</head>
<body role="document">
    <header id="banner" role="banner">We could put a logo and slogan text in this
area!</header>
    <nav id="navigation" role="navigation">
        <ul>
            <li><a href="#">First Option</a>
            <li><a href="#">Second Option</a>
            <li><a href="#">Third Option</a>
            <li><a href="#">Fourth Option</a>
            <li><a href="#">Fifth Option</a>
        </ul>
    </nav>
    <main id="main" role="main">
        <h1>ARIA Landmarks and Document Structure Roles</h1>
        <p>This page implements common ARIA roles in support of accessibility.</p>
        <div id="application" role="application" aria-labelledby="appId">
            <h2 id="appId">My Embedded Application Content</h2>
            <p tabindex="0" role="document">This text is accessible within the
application because of the document Role!</p>
            <textarea readonly rows="3" cols="35" aria-disabled="true">A disabled HTML5
textarea element accessible text.</textarea>
        </div>
        <div id="articles">
            <article id="article1" role="article" aria-labelledby="article1HeadingID">
                <h2 id="article1HeadingID">February Blog Post</h2>
                <p>blog content</p>
                <article role="article">
                    <h3>Comment Title</h3>
                    <p>comment content</p>
                </article>
            </article>
        </div>
    </main>
    <aside id="complementary" role="complementary">
        <h2 id="searchHeadingId">Search Form</h2>
        <search role="search" id="search" aria-labelledby="searchHeadingId">
            <label for="searchInputId">Phrase:</label>
            <input id="searchInputId" type="text">
            <input type="button" value="Search">
        </search>
    </aside>
    <footer id="contentinfo" role="contentinfo">Copyright 2016</footer>
</body>
</html>
```