Web accessibility

CONCEPT and GUIDELINES

POLITECNICO DI TORINO

Laura Farinetti - DAUIN

e-Lite

Summary

- Inclusive design and accessibility principles
- Web accessibility
- Screen readers
- Standards
  - WCAG
  - WAI-ARIA
- Legge Stanca

WAI-ARIA

WAI-ARIA

- The last ten years have seen the rise of Ajax, JavaScript, HTML5, and countless front-end frameworks
  - The internet is no longer a place of static HTML pages, but it is has become a playground for complex, almost desktop-like web applications, each with their own widgets, controls, and behavior
- Sometimes web development is pushed to the limit… and people with disabilities struggle with these new techniques
- This is not due to disabled JavaScript or insufficient capabilities of current assistive technology (AT)
  - On the contrary, in 2012 WebAIM found that over 98 percent of screen reader users had JavaScript enabled
  - Additionally, ATs like screen readers or refreshable Braille displays are getting better every year

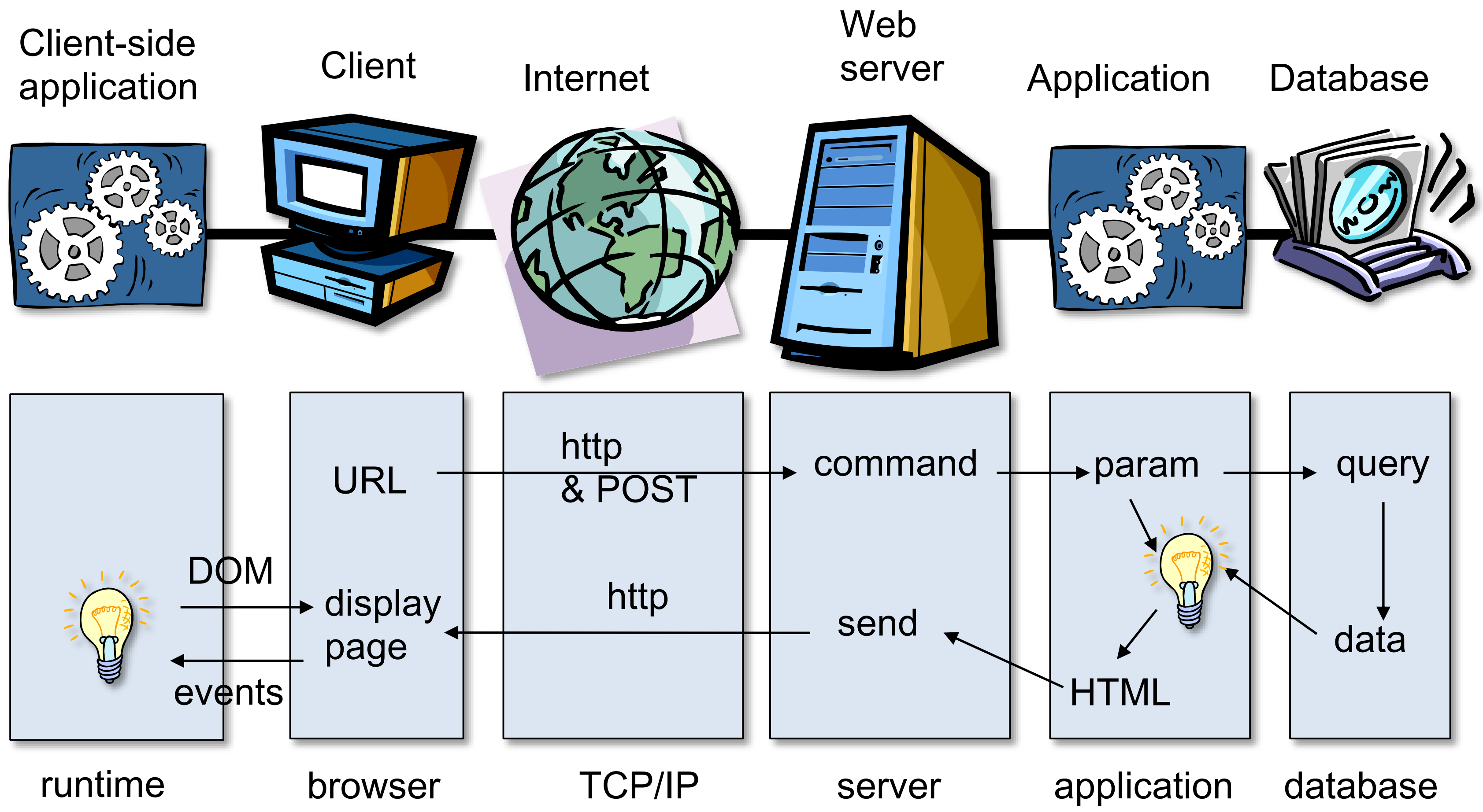
WAI-ARIA

- The problem lies with HTML limited ability to mark up web applications that make heavy use of JavaScript and produce a huge amount of dynamic content
- Four key obstacles can be identified when assistive technologies deal with JavaScript applications
  - Unknown functionality of components
  - Unknown states and properties of components
  - Unreported change of dynamic content
  - Bad keyboard accessibility

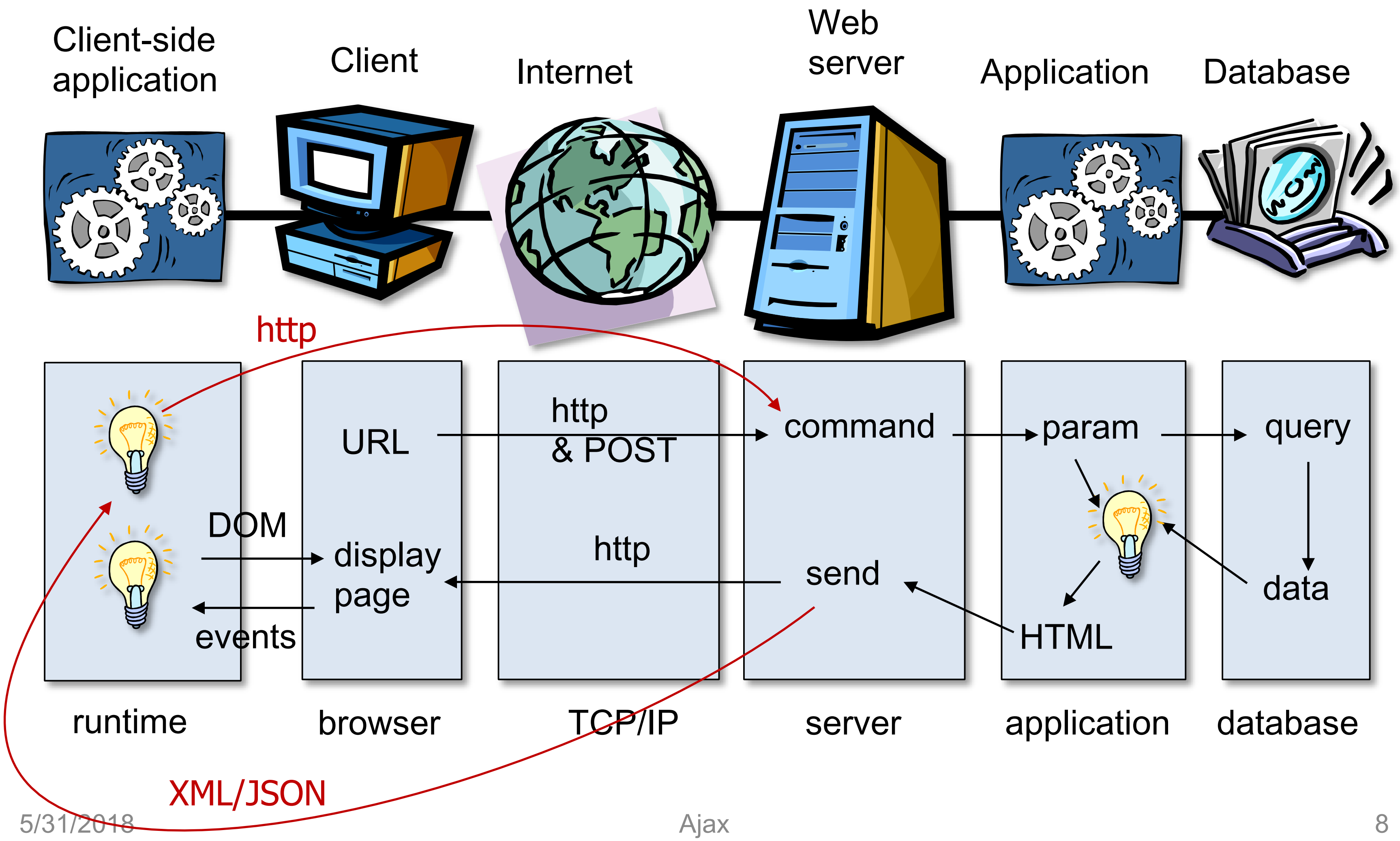
WAI-ARIA

- In general, accessibility issues with rich internet applications can be characterized as
  - Inability to provide the semantic structure of page areas and functionality (e.g., navigation, main content, search, etc.)
  - Inaccessibility of content that is dynamic and may change within the page (e.g., AJAX content updates)
  - Inability to change keyboard focus to page elements (e.g., setting focus to an error message within the page)
  - Difficulty with keyboard and screen reader accessibility with complex widgets and navigation elements (e.g., sliders, menu trees, etc.)
- ARIA can help address many of these issues

Rich-client transactions



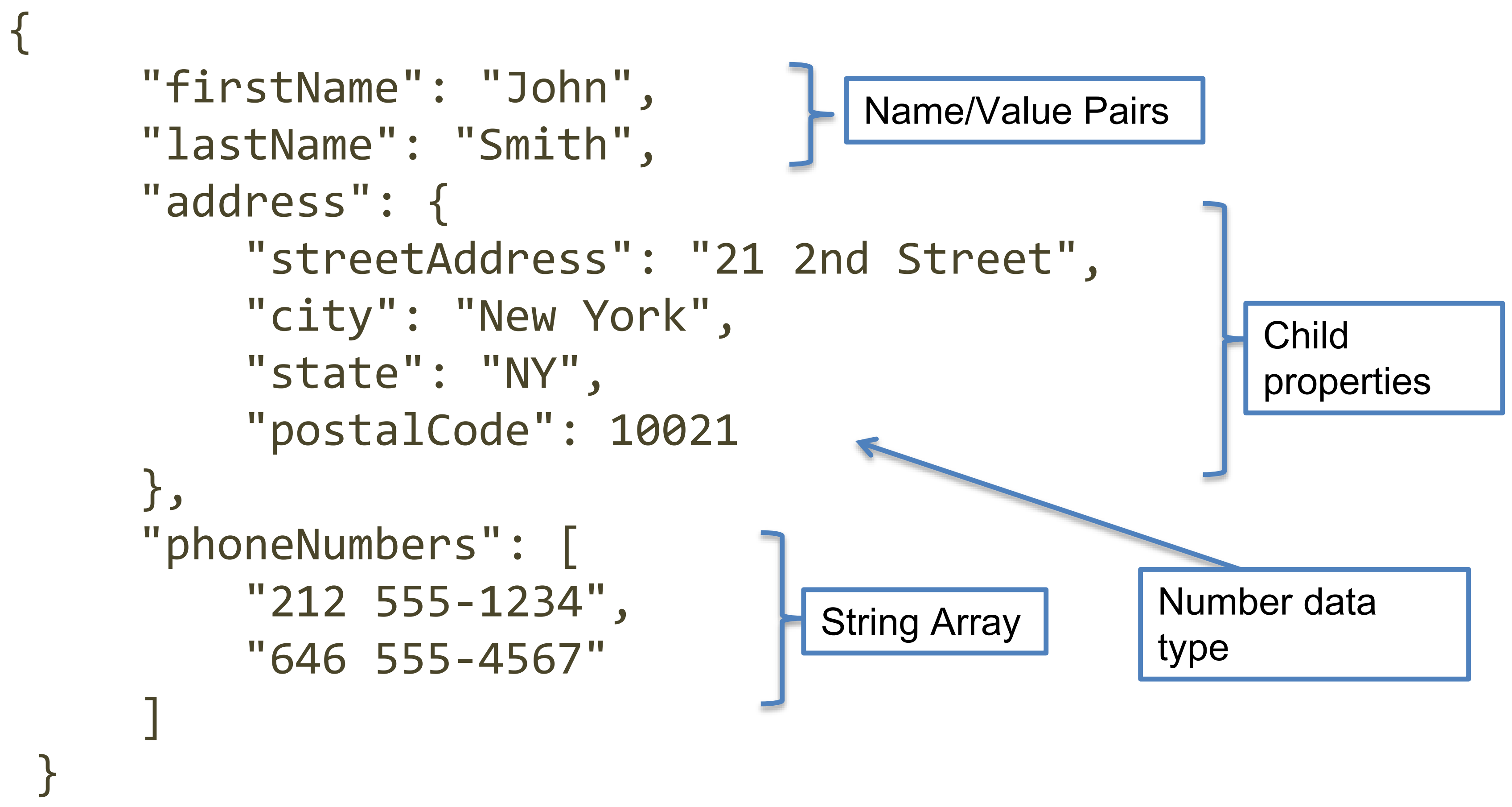
Rich-client asynchronous transactions



JSON

- "JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate" – JSON.org
- Important: JSON is a subset of JavaScript
- JSON is built on two structures
  - A collection of name/value pairs: in various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array. { … }
  - An ordered list of values: in most languages, this is realized as an array, vector, list, or sequence. […]

JSON example

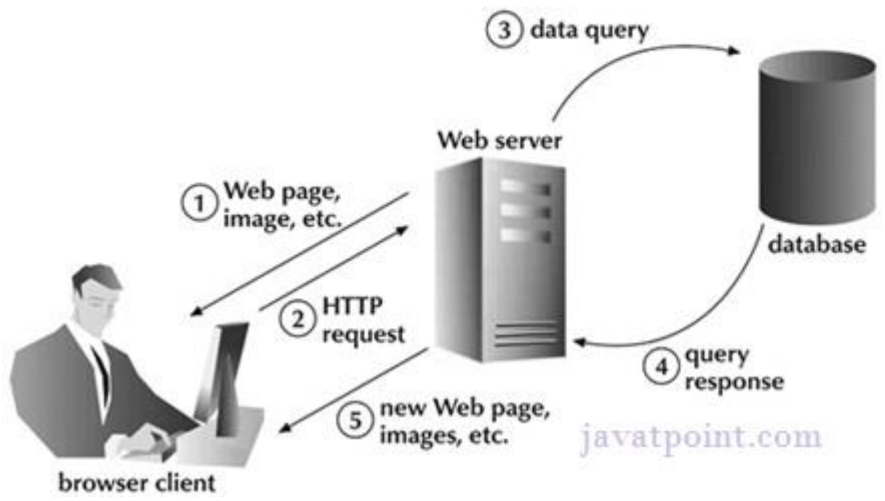


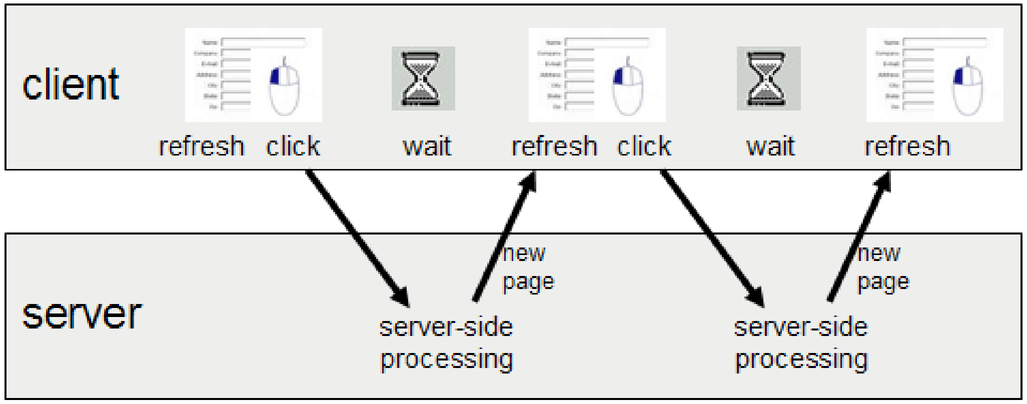
```
{
     "firstName": "John",
     "lastName": "Smith",
     "address": {
         "streetAddress": "21 2nd Street",
         "city": "New York",
         "state": "NY",
         "postalCode": 10021
     },
     "phoneNumbers": [
         "212 555-1234",
         "646 555-4567"
     ]
 }
```

Rich-client asynchronous transactions

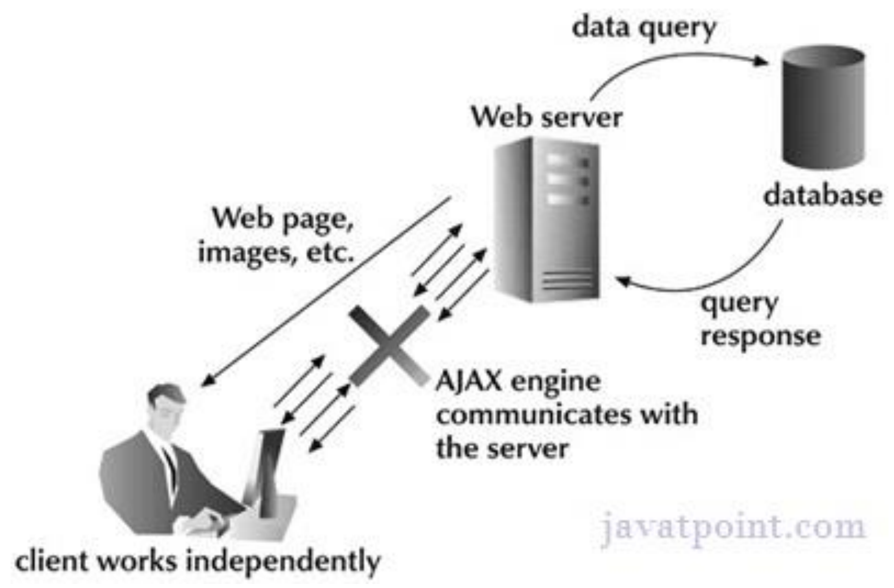
- In 2005, Jesse James Garrett wrote an online article titled “Ajax: A New Approach to Web Applications” (www.adaptivepath.com/ideas/essays/archives/000385.php)
- This article outlined a technique that he referred to as Ajax, short for Asynchronous JavaScript+XML, consisting in making server requests for additional data without unloading the web page, for a better user experience
- Garrett explained how this technique could be used to change the traditional click-and-wait paradigm that the Web had been stuck in since its start

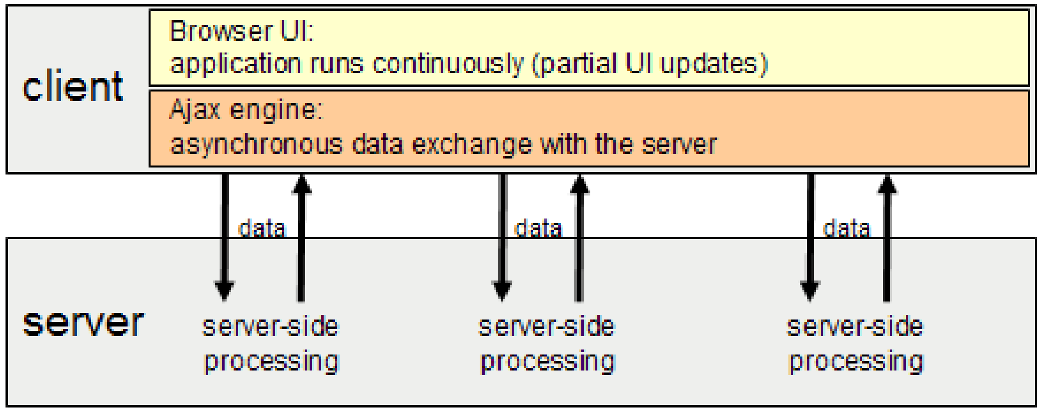
Synchronous (classic) web application model



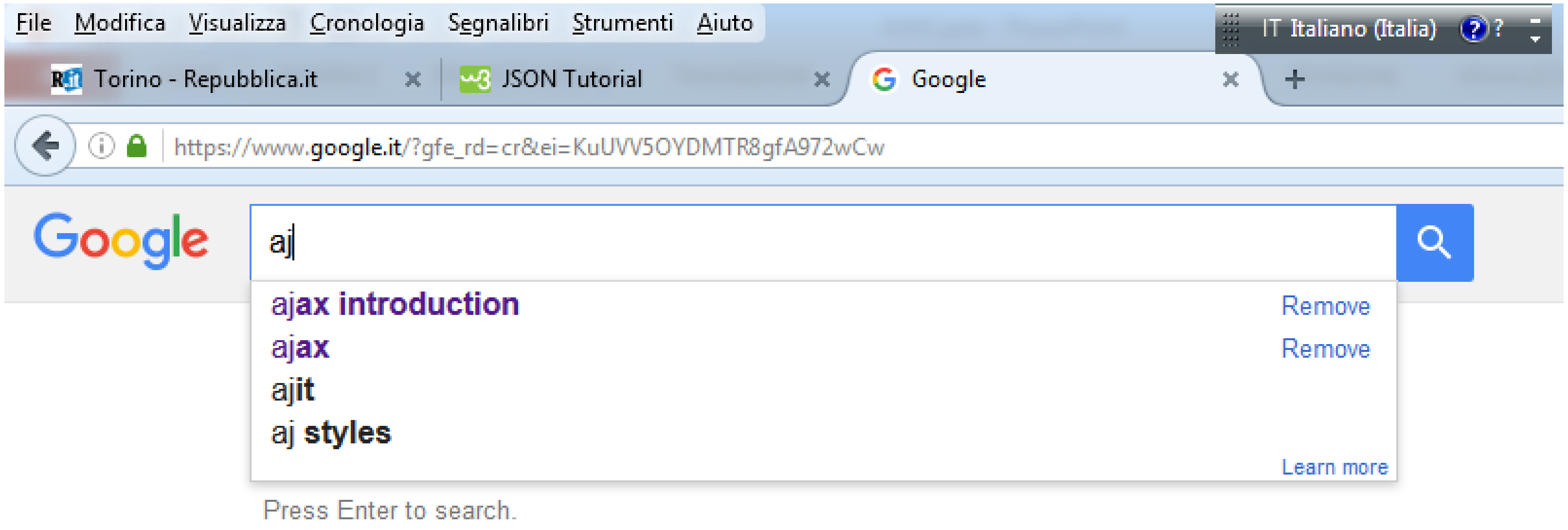


Asynchronous web application model





Example


File Modifica Visualizza Cronologia Segnalibri Strumenti Aiuto
IT Italiano (Italia)
Torino - Repubblica.it
JSON Tutorial
Google
https://www.google.it/?gfe_rd=cr&ei=KuUVV5OYDMTR8gfA972wCw
aj
ajax introduction
Remove
ajax
Remove
ajit
aj styles
Learn more
Press Enter to search.

WAI-ARIA

- "Web Accessibility Initiative Accessible Rich Internet Applications (WAI-ARIA)
- A technical specification that defines a way to make Web content and Web applications more accessible to people with disabilities
  - W3C Recommendation WAI-ARIA 1.1 (14/12/2017)
- Provides attributes for extending HTML markup with roles, states and properties to expose Web applications to Assistive Technologies
- ARIA was designed to be recognized only by assistive technology and does not affect the DOM or the style in any way

WAI-ARIA

- WAI-ARIA provides web authors with
  - Roles to describe the structure of the Web page, such as headings, regions, and tables (grids)
  - Roles to describe the type of widget presented, such as “menu”, “treeitem”, “slider”, “progressmeter”, …
  - Properties to describe the state widgets are in, such as “checked” for a check box, or “haspopup” for a menu
  - Properties to define live regions of a page that are likely to get updates (such as stock quotes), as well as an interruption policy for those updates—for example, critical updates may be presented in an alert dialog box, and incidental updates occur within the page
  - Properties for drag-and-drop that describe drag sources and drop targets
  - A way to provide keyboard navigation for the Web objects and events, such as those mentioned above

WAI-ARIA core components

- Roles
  - ARIA roles define what an element is or does
  - Most HTML elements have a default role that is presented to assistive technology
  - E.g., a button has a default role of "button" and a form has a default role of "form“
  - With ARIA, you can define roles that are not available in HTML, or you can override HTML default roles
  - Example: <form role="search">: in HTML, all forms have the same semantics but with ARIA, you can add to the semantics of a particular form to define it as the search form
- Types of ARIA roles
  - Landmark roles (the most important)
  - Widget roles
  - Document structure roles
  - Abstract roles (ontology, not for developers)
  - https://www.w3.org/WAI/PF/aria/roles

WAI-ARIA core components

- Properties
  - ARIA properties define properties or meanings of elements
  - You can extend HTML native semantics by defining properties for elements that are not allowed in standard HTML
  - Example: <input aria-required="true">: this property will cause a screen reader to read this input as being required (meaning the user must complete it)
- States
  - ARIA states are properties that define the current condition of an element
  - They generally change based on user interaction or some dynamic variable
  - Example: <input aria-disabled="true">: this property will cause a screen reader to read this input as being disabled

WAI-ARIA core components

- Keyboard navigation
  - ARIA also provides keyboard navigation methods for the web objects and events

- ARIA roles, states, and properties can be defined in markup or they can be defined and dynamically set and changed using scripting
- ARIA states and property attributes always start with "aria-" (e.g., aria-required="true")

HTML5 and ARIA

- Use ARIA only if necessary
  - If you can use a native HTML element [HTML5] or attribute with the semantics and behaviour you require already built in, instead of re-purposing an element and adding an ARIA role, state or property to make it accessible, then do so
  - Example:

```
<!------ avoid these if possible ------>
<span role="button">...</span>
<div role="link">...</div>

<!------ these are preferred ------>
<button type="button">...</button>
<a href="link">...</a>
```

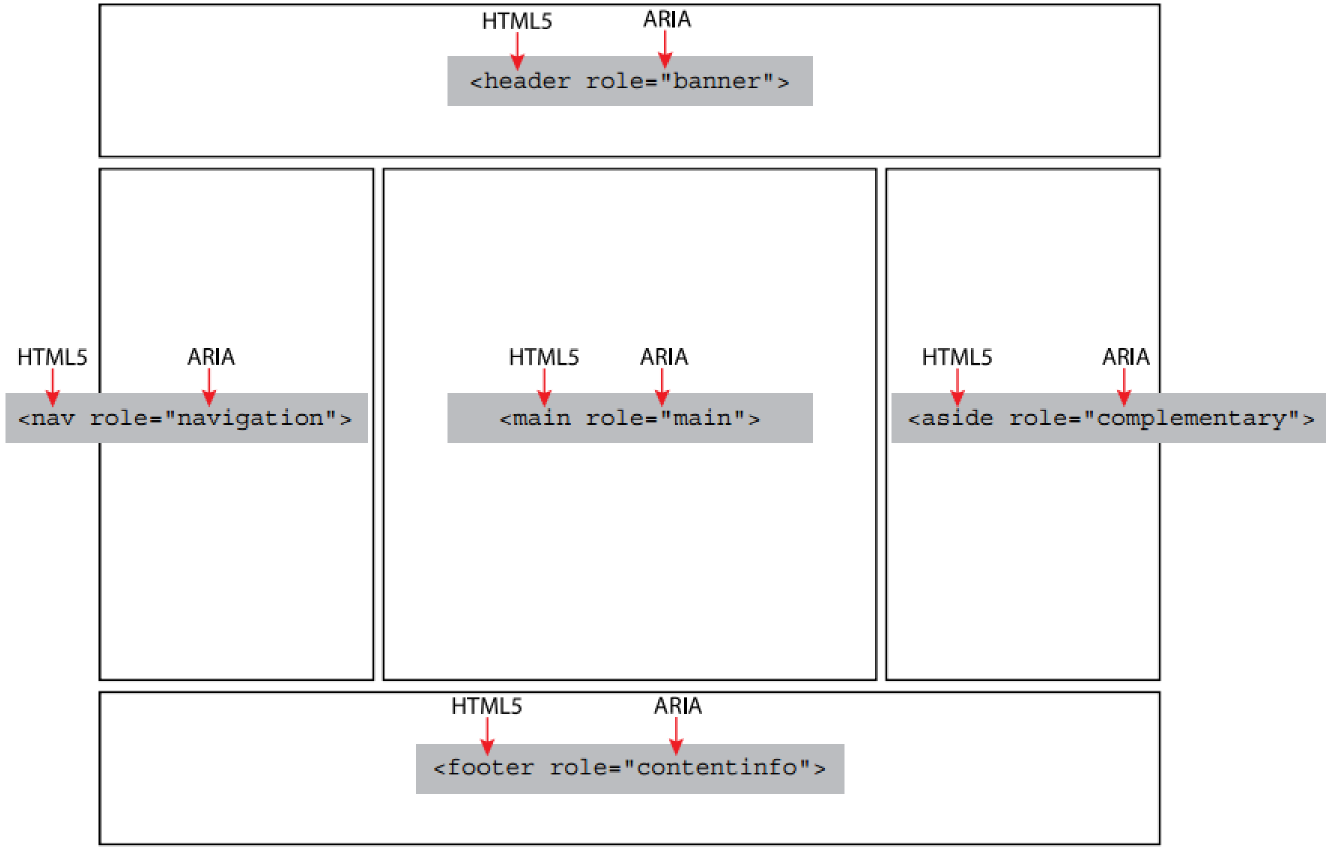
ARIA roles

- ARIA provides a rich role taxonomy that enables developers to classify otherwise meaningless tags
- This prepares the tags for assistive technologies by revealing the functionality or the part they play in the overall web document

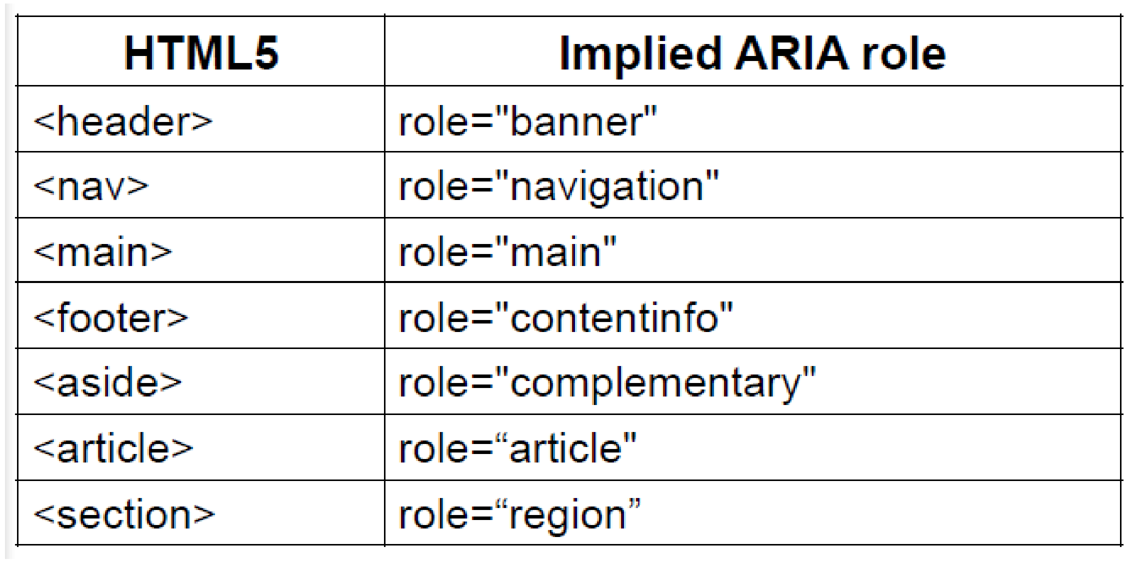
```
<ul id="myTab" class="nav nav-tabs" role="tablist">
  <li class="active"> <a href="#home" role="tab"
      data-toggle="tab">Home</a> </li>
  <li> <a href="#profile" role="tab"
      data-toggle="tab">Profile</a> </li>
  <li> <a href="#articles" role="tab"
      data-toggle="tab">Articles</a> </li>
</ul>
```

WAI-ARIA for landmark roles

- HTML5 elements and ARIA roles are complementary
  - Including both of them in your site provides a solid code structure and good navigation around the page



HTML5 semantic tags and ARIA landmark roles

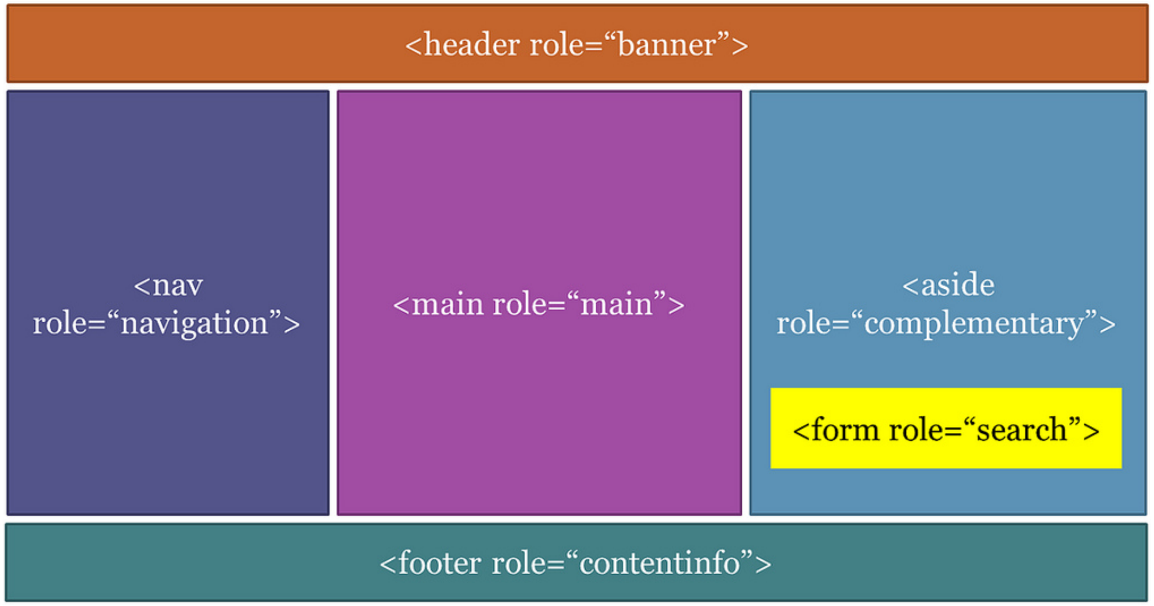


HTML5	Implied ARIA role
<header>	role="banner"
<nav>	role="navigation"
<main>	role="main"
<footer>	role="contentinfo"
<aside>	role="complementary"
<article>	role=“article"
<section>	role=“region”

- If HTML5 sectioning elements are used without understanding the associated landmark structure, assistive technology users will most likely be confused and less efficient in accessing content and interacting with web pages

Landmark roles

- Landmarks provide a powerful way to identify the organization and structure of a web page
  - Support keyboard navigation to the structure of a web page for screen reader users
- All content should reside in a semantically meaningful landmark in order that content is not missed by the user



Landmark roles

- Eight roles, each representing a block of content that occurs commonly on web pages
  - role=”banner”
  - role=”navigation” (e.g., a menu)
  - role=”main” (the main content of the page)
  - role=”complementary” (e.g., a sidebar)
  - role=”contentinfo” (meta data about the page, e.g., a copyright statement)
  - role=”search”
  - role=”form”
  - role=”application” (a web application with its own keyboard interface)
- If a role is used more than once on a page, the aria-label attribute should also be used in order to distinguish between the two regions
  - <div role=”navigation” aria-label=”Main menu”>
  - <div role=”navigation” aria-label=”User menu”>

Landmark roles usage

- Step 1: Identify the logical structure
  - Break the page into perceivable areas called "areas"
  - Typically, designers indicate areas visually using alignment and spacing of content
  - Regions can be further defined into logical sub-areas as needed
- Step 2: Assign landmark roles to each area
  - Assign landmark roles based on the type of content in the area
  - banner, main, complementary and contentinfo landmarks should be top level landmarks
  - Landmark roles can be nested to identify parent/child relationships of the information being presented

Landmark roles usage

- Step 3: Label each area
  - If a specific landmark role is used more than once on a web page, it should have a unique label
  - If a area begins with a heading element (e.g. h1-h6), it can be used as the label for the area using aria-labelledby attribute
  - If a area does not have a heading element, provide a label using the aria-label attribute
  - Avoid using the landmark role as part of the label: a navigation landmark with a label “Site Navigation” will be announced by a screen reader as “Site Navigation Navigation”, the label should simply be “Site”

Widget roles

- 25 widget roles
  - alert, alertdialog, button, checkbox, dialog, gridcell, link, log, marquee, menuitem, menuitemcheckbox, menuitemradio, option, progressbar, radio, scrollbar, slider, spinbutton, status, tab, tabpanel, textbox, timer, tooltip, treeitem
- 9 composite roles
  - Typically act as containers that manage other contained widgets
  - combobox, grid, listbox, menu, menubar, radiogroup, tablist, tree, treegrid

Document structure roles

- Describe structures that organize content in a page
  - Usually not interactive content
  - article, columnheader, definition, directory, document, group, heading, img, list, listitem, math, note, presentation, region, row, rowgroup, rowheader, separator, toolbar

States and attributes

- Similar features
  - Both provide specific information about an object, and contribute to the nature of roles
- Major difference
  - the values of properties (e.g., aria-labelledby) are often less likely to change throughout the application life-cycle than the values of states (e.g, aria-checked) which may change frequently due to user interaction
- aria-prefixed markup attributes
- Categorized as
  - Widget attributes
  - Live Region attributes
  - Drag-and-Drop attributes
  - Relationship attributes

States and attributes

https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/ARIA_Techniques#States_and_properties

Widget attributes

- aria-autocomplete
- aria-checked
- aria-current
- aria-disabled
- aria-expanded
- aria-haspopup
- aria-hidden
- aria-invalid
- aria-label
- aria-level
- aria-multiline
- aria-multiselectable
- aria-orientation
- aria-pressed
- aria-readonly
- aria-required
- aria-selected
- aria-sort
- aria-valuemax
- aria-valuemin
- aria-valuenow
- aria-valuetext

Live region attributes

- aria-live
- aria-relevant
- aria-atomic
- aria-busy

No ARIA is better than Bad ARIA

Drag & drop attributes

- aria-dropeffect
- aria-dragged

Relationship attributes

- aria-activedescendant
- aria-controls
- aria-describedby
- aria-flowto
- aria-labelledby
- aria-owns
- aria-posinset
- aria-setsize

WAI-ARIA in 5 steps

1. Alert users to what each element or widget is: the element’s role (such as a menu or a tree)
2. Alert the user to each element properties and important relationships (such as “aria-haspopup”, “aria-describedby” and other labels)
3. Alert the user to what each element is doing: the element’s state (such as “aria-expanded” or “aria-disabled”)
4. Alert users to any changes in the element’s state
5. Make sure the widget is keyboard accessible and the focus is predictable
   - Interactive controls should receive focus through the keyboard
   - Events can be triggered through the keyboard
   - How to trigger events should be intuitive to the user

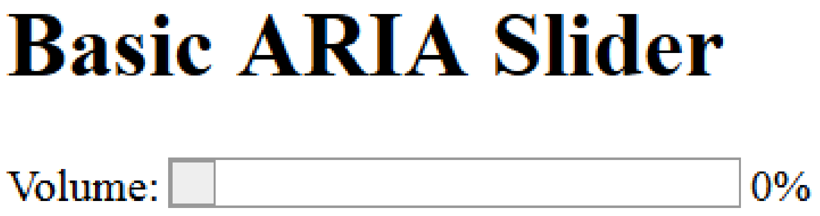
States and Properties

- The W3C provides some detailed WAI-ARIA best practices to follow
  - https://www.w3.org/TR/wai-aria-practices/

No ARIA is better than Bad ARIA

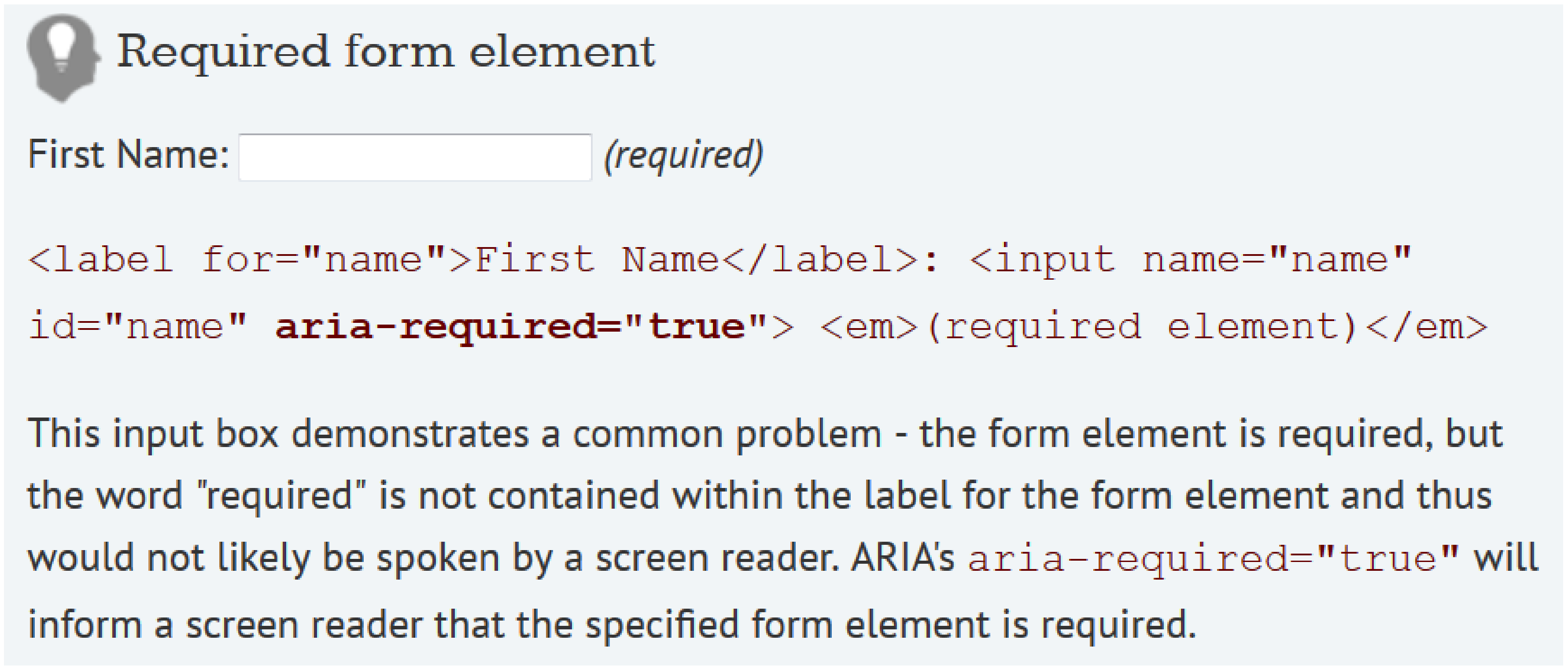
```
<li class="active">
  <a id="tab1" href="#home" role="tab" aria-controls="home"
     data-toggle="tab">Home</a>
</li>
```

Example: slider



```
<div class="clearfix">
  <span id="sliderLabel" class="floatLeft">Volume:</span>
  <div id="sliderRail1" class="sliderRailfloatLeft">
    <button class="sliderThumb" id="sliderThumb1" role="slider"
     aria-labelledby="sliderLabel" aria-valuemin="0" aria-valuemax="100"
     aria-valuenow="0" aria-valuetext="0%"></button>
  </div>
  <span id="sliderValue1"class="floatLeft">0%</span>
</div>
```

Example



Required form element

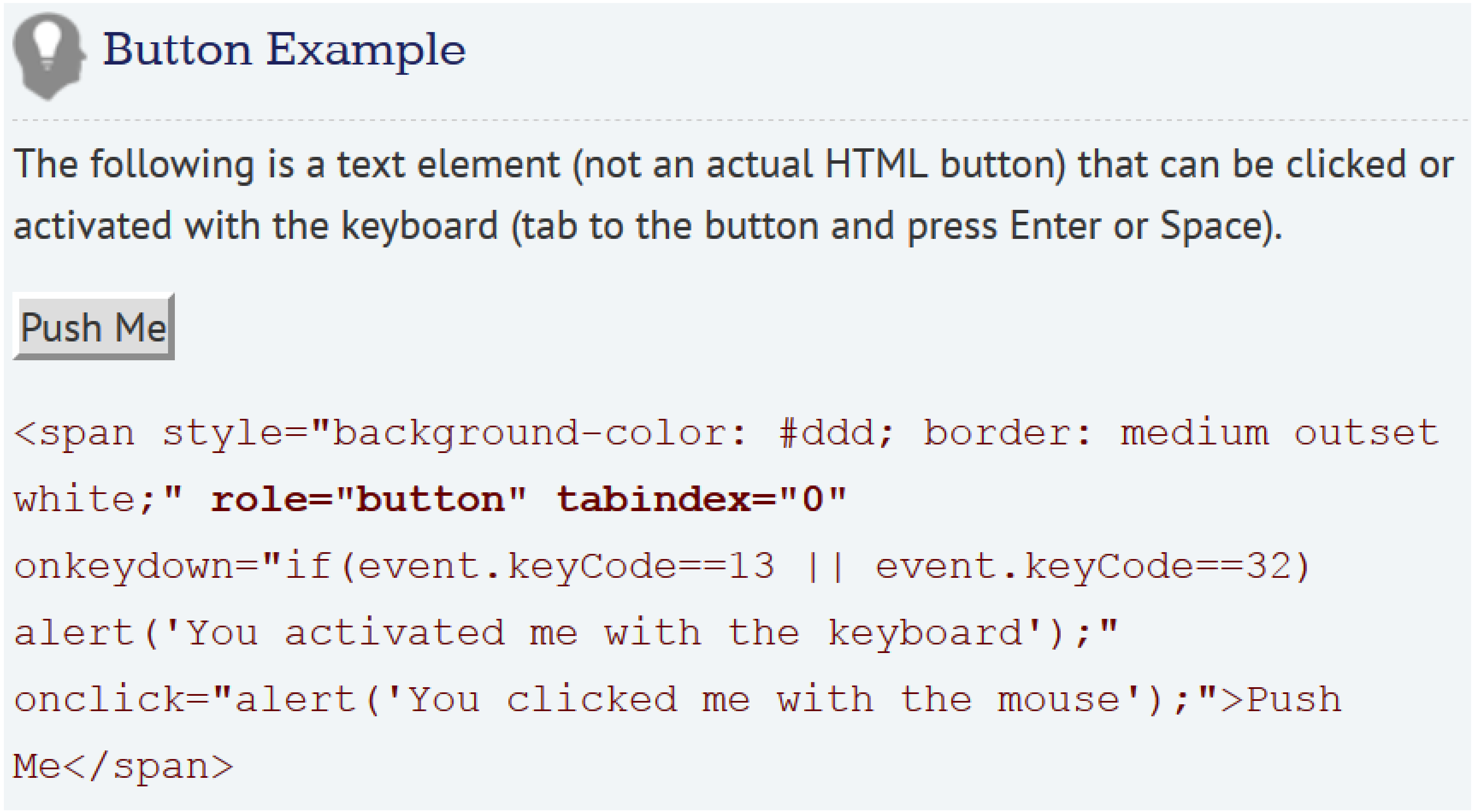
First Name: *(required)*

```
<label for="name">First Name</label>: <input name="name"
id="name" aria-required="true"> <em>(required element)</em>
```

This input box demonstrates a common problem - the form element is required, but the word "required" is not contained within the label for the form element and thus would not likely be spoken by a screen reader. ARIA's `aria-required="true"` will inform a screen reader that the specified form element is required.

Example

https://webaim.org/techniques/aria/



Button Example

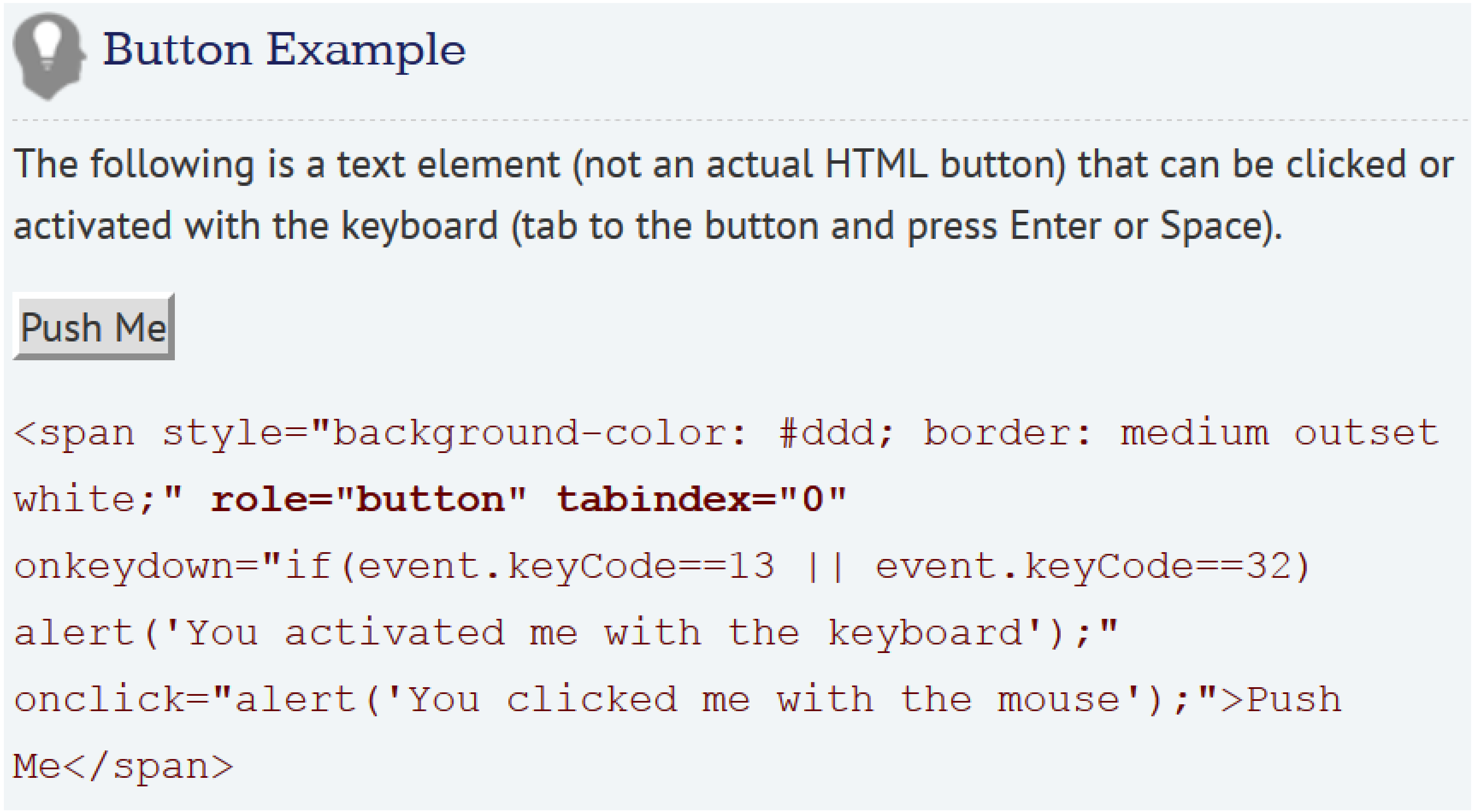
The following is a text element (not an actual HTML button) that can be clicked or activated with the keyboard (tab to the button and press Enter or Space).

Push Me

```
<span style="background-color: #ddd; border: medium outset
white;" role="button" tabindex="0"
onkeydown="if(event.keyCode==13 || event.keyCode==32)
alert('You activated me with the keyboard');"
onclick="alert('You clicked me with the mouse');">Push
Me</span>
```

Example

https://webaim.org/techniques/aria/



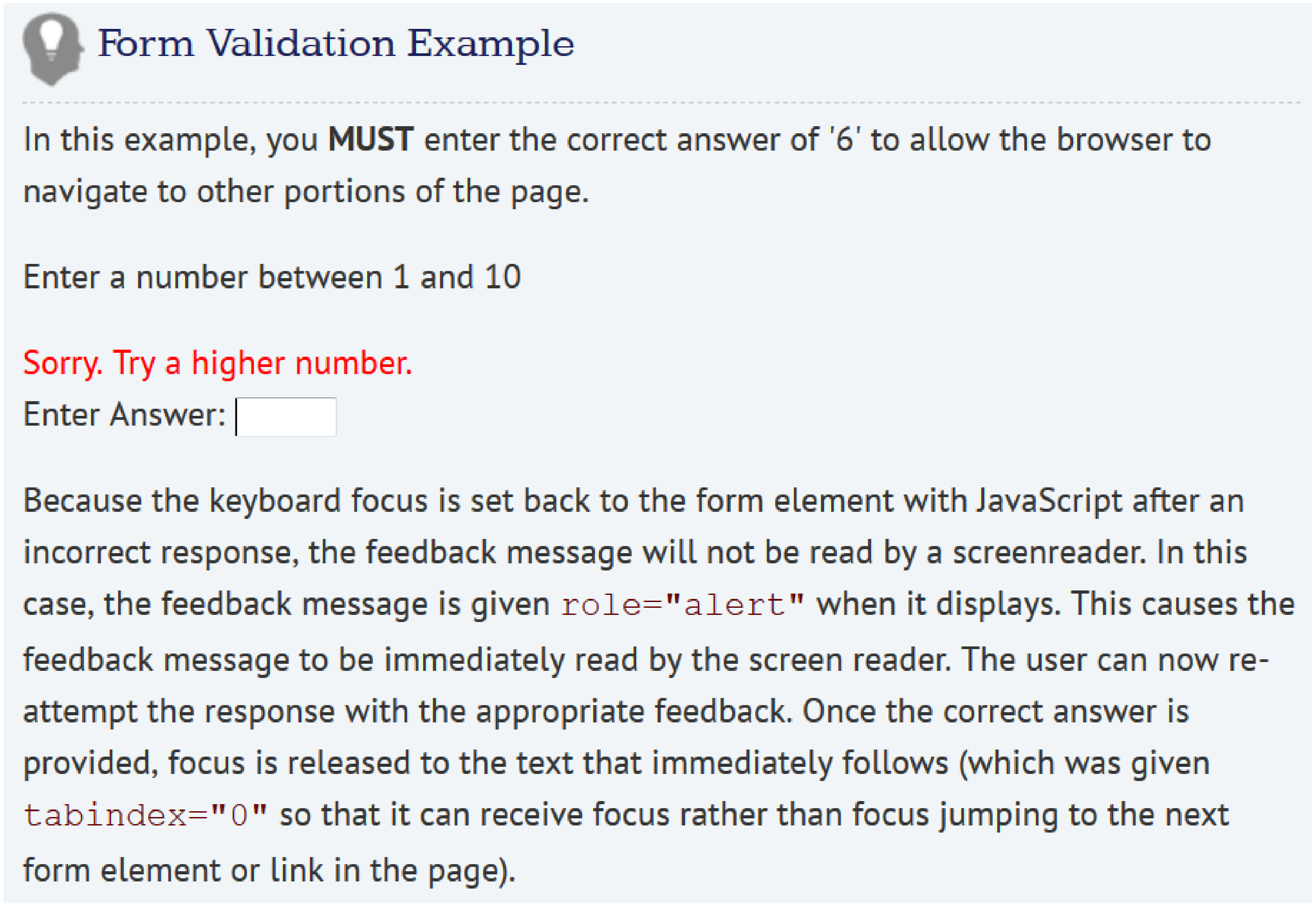
Button Example

The following is a text element (not an actual HTML button) that can be clicked or activated with the keyboard (tab to the button and press Enter or Space).

```
<span style="background-color: #ddd; border: medium outset
white;" role="button" tabindex="0"
onkeydown="if(event.keyCode==13 || event.keyCode==32)
alert('You activated me with the keyboard');"
onclick="alert('You clicked me with the mouse');">Push
Me</span>
```

Example

```
<p class="incorrect" id="feedback" role="alert">
   Sorry. Try a higher number.
</p>
```



Form Validation Example

In this example, you **MUST** enter the correct answer of '6' to allow the browser to navigate to other portions of the page.

Enter a number between 1 and 10

Sorry. Try a higher number.

Enter Answer:

Because the keyboard focus is set back to the form element with JavaScript after an incorrect response, the feedback message will not be read by a screenreader. In this case, the feedback message is given `role="alert"` when it displays. This causes the feedback message to be immediately read by the screen reader. The user can now re-attempt the response with the appropriate feedback. Once the correct answer is provided, focus is released to the text that immediately follows (which was given `tabindex="0"` so that it can receive focus rather than focus jumping to the next form element or link in the page).

Dynamic content updates

- When content changes dynamically within a web page, it may cause accessibility problems
  - What happens if a screen reader is currently reading an element that is updated?
  - If the updated content is important, should you interrupt the user and set focus immediately to the new content, do you simply inform the user of the update, or do you do nothing?
  - How do you set focus or allow the user to jump to the updated content?
- With WAI-ARIA the developer can identify regions that dynamically change as a live region
  - A live region allows content updates in a way that a screen reader understands
  - It also allows the developer to add additional functionality to alert the user, provide controls for the live region, determine the amount of new content that would be read, …

Live regions

- To create a live region, the developer adds the aria-live property to the element
  - The value, or politeness level (or alternatively the intrusiveness level) specifies what a screen reader should do when the element is updated
- aria-live="off“ tells the screen reader to not announce the update
  - Should be used for non-important or irrelevant content updates
- aria-live=“polite“ notifies the user of the content change as soon as he/she is done with the current task
  - This might take the form of a beep or an audio indication of the update, and the user can then choose to directly jump to the updated content
  - Should be the most common for content updates, especially for things like status notification, weather or stock updates, chat messages, etc.

Live regions

- aria-live=“assertive” will result in the user being alerted to the content change immediately or as soon as possible
  - Assertive would be used for important updates, such as error messages

- Example

```
<div id="myTabContent" class="tab-content"
     aria-live="polite">
    ...
</div>
```

Enhanced Keyboard Navigation

- In HTML, the only elements that could receive keyboard focus with the TAB key are links and form elements
- With scripting, however, you can add mouse interactivity to nearly any element (e.g. spans, paragraphs, …)
- The functionality of these non-focusable elements cannot be made accessible to screen reader and keyboard-only users
- ARIA enables every HTML element to receive keyboard focus by extending the "tabindex" attribute so that it can be applied to any element

```
<li class="tab active">
   <a id="tab1" href="#home" role="tab" aria-controls="home"
      aria-selected="true" data-toggle="tab" tabindex="0">Home</a>
</li>
```

Enhanced Keyboard Navigation

- The tabindex attribute has three distinct uses:
  - tabindex="1" (or any number greater than 1) defines an explicit tab order; this is almost always a bad idea
  - tabindex="0" allows elements besides links and form elements to receive keyboard focus; it does not change the tab order, but places the element in the logical navigation flow, as if it were a link on the page
  - tabindex="-1" removes the element from the default navigation flow but allows it to receive "programmatic" focus, i.e. focus can be set to the element through scripting, links, etc.
    - Example: a modal dialog window

LEGGE STANCA

Legge stanca

- D.M. 9 luglio 2004 che regola l'accessibilità dei siti web in Italia
  - «Disposizioni per favorire l'accesso dei soggetti disabili agli strumenti informatici»
  - Definita come la capacità dei sistemi informatici di erogare informazioni fruibili, senza discriminazioni, anche da parte di coloro che a causa di disabilità necessitano di tecnologie assistive o di configurazioni particolari

Soggetti destinatari della legge

https://www.webaccessibile.org/articoli/legge-stanca-guida-ai-22-requisiti-tecnici/

- Pubbliche amministrazioni
- Enti pubblici economici, aziende private concessionarie di servizi pubblici
- Aziende municipalizzate regionali
- Enti di assistenza e di riabilitazione pubblici
- Aziende di trasporto e di telecomunicazione a prevalente partecipazione di capitale pubblico
- Aziende appaltatrici di servizi informatici

Requisiti tecnici

- 22 requisiti tecnici, ridotti a 12 nel 2013
    - Riferimento: WCAG 2.0, livello AA
- Requisito 1 – Alternative testuali: fornire alternative testuali per qualsiasi contenuto di natura non testuale in modo che il testo predisposto come alternativa possa essere fruito e trasformato secondo le necessità degli utenti, come per esempio convertito in stampa a caratteri ingranditi, in stampa Braille, letto da una sintesi vocale, simboli o altra modalità di rappresentazione del contenuto.
- Requisito 2 – Contenuti audio, contenuti video, animazioni: fornire alternative testuali equivalenti per le informazioni veicolate da formati audio, formati video, formati contenenti immagini animate (animazioni), formati multisensoriali in genere.

Requisiti tecnici

- Requisito 3 – Adattabile: creare contenuti che possano essere presentati in modalità differenti (ad esempio, con layout più semplici), senza perdita di informazioni o struttura.
- Requisito 4 – Distinguibile: rendere più semplice agli utenti la visione e l'ascolto dei contenuti, separando i contenuti in primo piano dallo sfondo.
- Requisito 5- Accessibile da tastiera: rendere disponibili tutte le funzionalità anche tramite tastiera.
- Requisito 6- Adeguata disponibilità di tempo: fornire all'utente tempo sufficiente per leggere ed utilizzare i contenuti.
- Requisito 7- Crisi epilettiche: non sviluppare contenuti che possano causare crisi epilettiche.

Requisiti tecnici

- Requisito 8- Navigabile: fornire all'utente funzionalità di supporto per navigare, trovare contenuti e determinare la propria posizione nel sito e nelle pagine.
- Requisito 9- Leggibile: rendere leggibile e comprensibile il contenuto testuale.
- Requisito 10- Prevedibile: creare pagine web che appaiano e che si comportino in maniera prevedibile.
- Requisito 11- Assistenza nell'inserimento di dati e informazioni: aiutare l'utente ad evitare gli errori ed agevolarlo nella loro correzione.
- Requisito 12- Compatibile: garantire la massima compatibilità con i programmi utente e con le tecnologie assistive.

https://www.webaccessibile.org/categorie/legge-stanca/guida-ai-22-requisiti-tecnici/

References

- Accessible Rich Internet Applications (WAI-ARIA) 1.1
  – https://www.w3.org/TR/wai-aria/
- ARIA landmarks example
  – https://www.w3.org/TR/2017/NOTE-wai-aria-practices-1.1-20171214/examples/landmarks/index.html
- WebAIM: accessibility to Rich Internet Applications
  – https://webaim.org/techniques/aria/
- Legge Stanca
  – http://www.camera.it/parlam/leggi/04004l.htm

License

- This work is licensed under the Creative Commons “Attribution-NonCommercial-ShareAlike Unported (CC BY-NC-SA 3,0)” License.
- You are free:
  - to Share - to copy, distribute and transmit the work
  - to Remix - to adapt the work
- Under the following conditions:
  - Attribution - You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
  - Noncommercial - You may not use this work for commercial purposes.
  - Share Alike - If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.
- To view a copy of this license, visit http://creativecommons.org/license/by-nc-sa/3.0/