Main rich interface components accessibility guidelines

Date	Version	Author	Statut / Comments
August 3rd, 2018	3.0	Atalan	These guidelines follow ARIA 1.0 specifications. Please refer to WAI-ARIA Authoring Practices 1.1 for the latest version of ARIA. Next release of these AcceDe Web guidelines: March 2019.
August 10, 2020	4.0	Atalan	Major updates to ensure compliance with RGAA 4.
June 15, 2023	4.0	Atalan	Improved document accessibility.
October 24, 2023	4.1.2	Atalan	Guidelines adjustments according to APG.
April 18, 2024	4.1.2	Atalan	Guidelines adjustments according to APG.

In partnership with:

Air Liquide – Atos – BNP Paribas – Capgemini – EDF – Generali – L'Oréal – SFR – SNCF – Société Générale – SPIE – Total

Observers:

AbilityNet (UK) – Agence Entreprises & Handicap – AnySurfer (Belgium) – Association des Paralysés de France (APF) – Association Valentin Haüy (AVH) – CIGREF – Design For All Foundation (Spain) – ESSEC – Handirect – Hanploi – Sciences Po – Télécom ParisTech

Table of contents

Introduction

Context and objectives

This documentation provides guidelines to make common rich interface components more accessible and to ensure WCAG 2.1 compliance.

This manual is part of a set of four complementary manuals that can be downloaded from the AcceDe Web website:

- Graphic and functional accessibility guidelines.
- HTML and CSS accessibility guidelines (this manual).
- **Main rich interface components accessibility guidelines.**
- Accessibility guidelines for editors (template).

Who should read this document, and how to use it?

This document should be provided to stakeholders and/or contractors performing the technical specifications or development of rich application components. It complements project specifications. Its recommendations may be supplemented or removed depending on the context of use, such work can be carried out by the project owner.

It is important that these recommendations be used in the process of selecting or creating rich application components.

Note

The online version of these guidelines comes with many examples, links to complementary resources, etc. It is available at: www.accede-web.com.

Contact

Please send any comments about this document to Atalan, the coordinator of the AcceDe Web project, at the following email address: accede@atalan.ca.

You can also find more information about the methodological instructions of the AcceDe Web project on the website www.accede-web.com.

License agreement

Accordions

Principle

Accordions are dynamic components that optimize the display of content in a limited space with expand/collapse mechanisms on a group of panels.

They are usually controlled by a button over each panel, which toggles its contents in and out of view.

This code is based on the "Accordion" design pattern found in the ARIA Authoring Practices Guide (APG) of the W3C.

Core HTML base

```
<h2>
  <button aria-expanded="false" aria-controls="accordion-panel-
1">Panel 1 header</button>
</h2>
<div id="accordion-panel-1">
  [Panel 1 content (hidden)]
</div>
<h2>
  <button aria-expanded="true" aria-controls="accordion-panel-
2">Panel 2 header</button>
</h2>
<div id="accordion-panel-2">
  [Panel 2 content (visible)]
</div>
<h2>
  <button aria-expanded="false" aria-controls="accordion-panel-
3">Panel 3 header</button>
</h2>
<div id="accordion-panel-3">
  [Panel 3 content (hidden)]
</div>
```

ARIA roles, states and properties

- Each panel header must be marked up with `<button>`.
- Each panel header tab must be surrounded by a header tag (`<h1>` to `<h6>`), depending on the context in which the accordion is placed.
- The aria-expanded attribute must be added to each header tab. Its value will be set dynamically based on the state of the accordion:
    - `aria-expanded="true"` when the associated panel is open.
    - `aria-expanded="false"` when the associated panel is closed.
- Each header tab must be attached to its panel by means of the `aria-controls` attribute:
    - Each panel must have an `id` attribute set to a unique value.
    - Each panel header tab must have an `aria-controls` attribute set to the value of the `id` attribute of the associated panel.

Keyboard interactions

Enter or **Spacebar**

- If the keyboard focus is on the header tab of a closed panel, one of these keys opens the associated panel. If the accordion authorizes the expansion of only one panel at the time, and if another panel is already open, it closes the panel.
- If the keyboard focus is on the header tab of an open panel, one of these keys closes the associated panel if the accordion authorizes the collapse of that panel. Some accordions only allow one panel to be expanded at any time. In that case, the open panel cannot be closed by activating its associated heading tab.

Note

In the HTML code example, heading level 2 (`<h2>`) is used to mark up the panel headers. The heading level must be adapted to the context of the page: it is important to maintain a logical heading hierarchy in the page.

For instance, if an `<h2>` heading introduces the accordion, then each panel header becomes a child of this level 2 heading and must be marked up with `<h3>`.

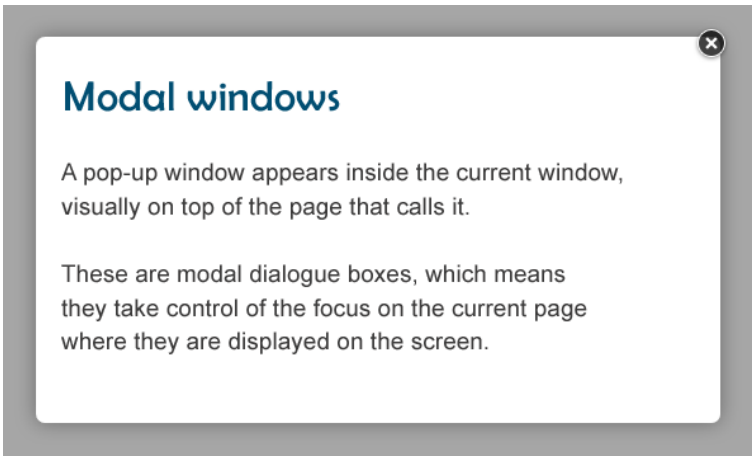
Components

The components of the “Accordeons” are shown here because their level of accessibility is considered good or very good.

However, before using them in your project, it is important to check that the specifications presented above have been respected. Some components may require some adjustments.

Modal window

Principle



A modal window appears inside the current window, and is displayed *above* the page that calls it.

Modal windows take control of the current page as long as they are displayed on the screen.

This sheet is based on the "Modal window" design pattern detailed in ARIA Authoring Practices Guide (APG) of the W3C.

Core HTML base

The HTML base of a modal window differs, depending on whether or not it has a title displayed on the screen.

Modal window with a title displayed on the screen

```
<div role="dialog" aria-modal="true" aria-labelledby="modal-heading">
  <button>Close</button>
  <h1 id="modal-heading">[Modal heading]</h1>
  [Modal content]
</div>
```

Modal window without a title displayed on the screen

```
<div role="dialog" aria-modal="true" aria-label="[Modal
heading]">
    <button>Close</button>
    [Modal content]
</div>
```

ARIA roles, states and properties

- `role="dialog"` must be applied to the container in the modal window
- `aria-modal="true"` must be applied to the container in the modal window
- If the title of the modal window is displayed on the screen, it must be attached to the modal window via the attribute `aria-labelledby`:
    - The title of the modal window must have an `id` attribute with a unique value.
    - The container of the modal window must have an `aria-labelledby` attribute with the value of the id attribute of the modal window title.
- If the title of the modal window is not displayed on screen, `aria-label` must be applied and provided on the modal window container.

Keyboard interactions

Tab

When the modal window is displayed, this key moves the keyboard focus successively through the interactive elements in the modal window. If the focus is on the last interactive element in the modal dialog box when the tab is pressed the keyboard focus moves to the first interactive element in the modal window.

Shift + **Tab**

This key combination has the same behavior as the **Tab** key, but in the reverse order. If the keyboard focus is on the first interactive element in the modal window when the key combination is pressed, the focus moves to the last interactive element in the modal window.

Esc

When the modal window is displayed, the modal window closes and the keyboard focus is placed on the interactive element that triggered the modal window.

Expected behavior

When the modal window is displayed (open)

- The keyboard focus is dynamically placed on the first interactive element contained in the modal window.
- The keyboard focus must be confined to the modal window and tabulating must not be possible on the rest of the page (below the modal window).
- The modal window can be closed with the **Esc** key.

When the modal window is hidden (closed)

- The keyboard focus must be repositioned on the element that opened the modal window opening.
- Ideally, the modal window is removed from the DOM. However, if the modal window remains present in the source code, `display: none` or `visibility: hidden` must be applied to its container.

Components

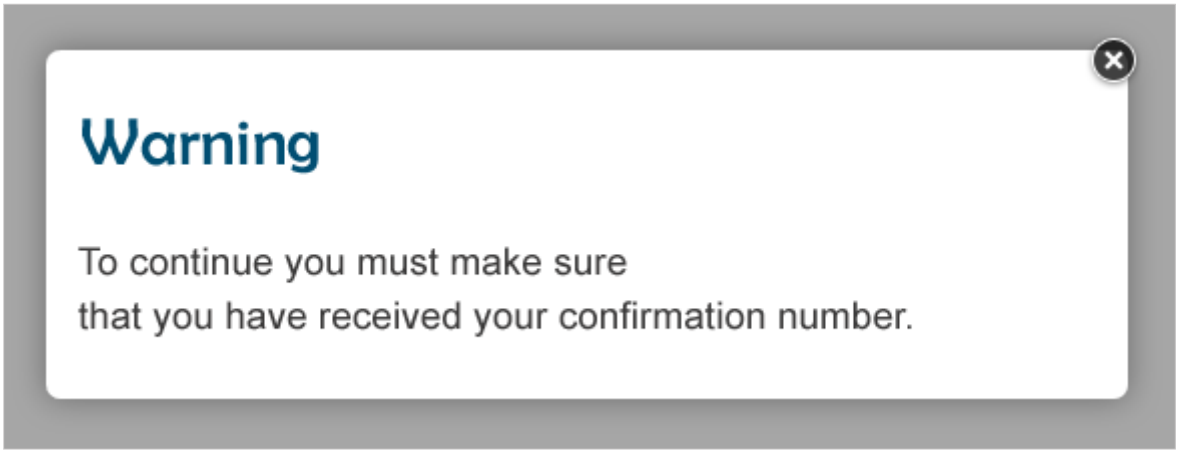
These "Modal window" components are proposed here because their level of accessibility is considered good or very good.

However, before using them in your project, it is important to check that the specifications presented above have been respected. Certain components may need some adjustments before they can be used in your project.

Alert dialog

Principle

A modal alert box is a subtype of a modal dialog window.



It sends a short alert or prompts confirmation from the user, and are appropriate when:

- The message is no longer than one phrase.
- Punctuation is not necessary to understand the message. For example, modal alert boxes are not appropriate to state that a specific syntax, such as "MM/DD/YYYY", is the expected format for a date field.
- The message does not contain information that the person will need later, such as a phone number.
- The message does not contain interactive elements, such as a link to a resource.

This code is based on the "Alert Dialog" design pattern found in the ARIA Authoring Practices Guide (APG) of the W3C.

Core HTML

The HTML base of a modal alert window is different depending on whether or not there is a title on the screen.

Alert dialog with a displayed title

```
<div role="alertdialog" aria-modal="true" aria-labelledby="modal-heading" aria-describedby="modal-content">
```

```
  <button>Close</button>
  <h1 id="modal-heading">[Modal window title]</h1>
  <p id="modal-content">[Modal window content]</p>
</div>
```

Alert dialog without a displayed title

```
<div role="alertdialog" aria-modal="true" aria-label="[Modal
window title]" aria-describedby="modal-content">
   <button>Close</button>
   <p id="modal-content">[Modal window content]</p>
 </div>
```

ARIA roles, states and properties

- `role="alertdialog"` must be applied to the container of the alert dialog.
- `aria-modal="true"` must be applied to the container of the alert dialog.
- If the title of the alert dialog is displayed it must be programmatically associated to the alert dialog with the `aria-labelledby` attribute:
  - The title of the alert dialog must contain an `id` attribute set to a unique value.
  - The container of the alert dialog must have an `aria-labelledby` attribute set to the value of the id attribute of the alert dialog.
- If the title of the alert dialog is not displayed an `aria-label` attribute must be applied and its value set to the container of alert dialog.
- The message must be programmatically associated with the alert dialog using the `aria-describedby` attribute:
  - The message must contain an `id` attribute set to a unique value.
  - The container of the alert dialog must have an `aria-describedby` attribute set to the value of the id of the message.

Keyboard interactions

Tab

When the alert dialog is displayed, this key successively moves the keyboard focus through each interactive element in the alert dialog. If the keyboard focus is on the last interactive element in alert dialog when the key is pressed, the keyboard focus moves to the first interactive element in the alert dialog.

Shift + **Tab**

This key combination has the same behavior as the **Tab** key, but in reverse order. If the keyboard focus is on the first interactive element in the alert dialog when the key combination is pressed, the keyboard focus moves to the last interactive element in the modal box.

Esc

When the alert dialog is displayed, closes the alert dialog, and moves the keyboard focus to the interactive element that triggered the alert dialog opening.

Expected behavior

When the alert dialog is displayed

- The keyboard focus is dynamically placed on the first interactive element in the modal alert window.
- The keyboard focus must be confined to the modal alert window and tabulating must not be possible on the rest of the page (below the modal alert window).
- The modal alert window can be closed using the **Esc** key.

When the modal alert window is closed

- The keyboard focus must be repositioned on the element that triggered the opening of the modal alert window.
- Ideally, the modal alert window is removed from the DOM. However, if the modal alert window is still present in the source code, then `display: none` or `visibility: hidden` must be applied to its container.

Components

The "Modal Alert Window" components are shown here because their level of accessibility is considered good or very good.

However, before using them in your project, it is important to check for compliancy with the specifications presented above. Certain components may require some adjustments.

“Show more” buttons

Principle

"Show more" buttons are dynamic components that display additional content just before the content, each time the button is activated. Content is added with each press, as long as additional hidden content remains available.

The "Show more" button disappears when all the additional hidden content has been displayed on the page.

Core HTML base

Before activation of the button

```
<div>[Default content]</div>
<button>["Show More" button label]</button>
```

After activation of the button

```
<div>[Default content shown before activation]</div>
<div tabindex="-1">[Additional information which is added using the button]</div>
<button>["Show More" button label]</button>
```

ARIA roles, sates and properties

Following the activation of the button, `tabindex="-1"` must be added to the container of the new content.

Keyboard interaction

Enter and **Spacebar**

When the keyboard focus is on the button, either of these keys will show additional hidden information, as long as there is hidden information available.

Expected behavior

- When the keyboard focus is on the button, additional information can be shown by hitting the **Spacebar** and **Enter keys**. For this, listen to the click event.
- When the button is activated, the keyboard focus is dynamically placed on the container of the new additional information.
- When there is no more additional information to show, the button disappears.

Notes

If the button only adds interactive elements, the `tabindex="-1"` attribute can be omitted and the focus can be simply positioned on the first interactive element which appears after the button is activated.

This is the case with a "View more news" button, for example, that triggers links to news items:

```
<ul>
  <li><a href="…">News Story 1</a></li>
  <li><a href="…">News Story 2</a></li>
  <li><a href="…">News Story 3</a></li>
</ul>
<button>View more news stories</button>
```

When the button is activated, the keyboard focus is sent to the link "News Story 4" which be the new link that appears.

There's no need for `tabindex="-1"` because the `<a>` element can receive the keyboard focus by default.

```
<ul>
  <li><a href="…">News Story 1</a></li>
  <li><a href="…">News Story 2</a></li>
  <li><a href="…">News Story 3</a></li>
  <li><a href="…">News Story 4</a></li>
  <li><a href="…">News Story 5</a></li>
  <li><a href="…">News Story 6</a></li>
</ul>
```

Radio buttons customized using ARIA

Principle

Radio buttons are form elements that are used to select a single option out of a group of possible options.

Radio buttons are "customized" when they are not built using the standard HTML code for radio buttons, found in the specification `<input type="radio"/>`.

By following the recommendations below the default behavior of standard HTML radio buttons can be reproduced **in situations where the standard HTML code for radio buttons cannot be used**.

This code is based on the "Radio Group" design pattern found in ARIA Authoring Practices Guide (APG) of the W3C.

Core HTML base

```
<div role="radiogroup" aria-labelledby="question">
  <h2 id="question">Question</h2>
  <div role="radio" aria-checked="false" tabindex="-1">
    <img aria-hidden="true" src="radio.svg" alt="Not selected: "
/>
    Choice 1
  </div>
  <div role="radio" aria-checked="true" tabindex="0">
    <img aria-hidden="true" src="radio-checked.svg"
alt="Selected: " />
    Choice 2
  </div>
  <div role="radio" aria-checked="false" tabindex="-1">
    <img aria-hidden="true" src="radio.svg" alt="Not selected: "
/>
    Choice 3
  </div>
</div>
```

ARIA roles, states and properties

- `role="radiogroup"` must be placed on the radio button group.
- `role="radio"` must be placed on the container of each radio button.
- The `tabindex` attribute must be placed by default on the container of each radio button, and its value set dynamically according to the state of the radio buttons:
    - If no button is selected: `tabindex="0"` on the first and last radio button of the group and `tabindex="-1"` on the other radio buttons.
    - If one radio button is selected: `tabindex="0"` on the selected button, `tabindex="-1"` on the other radio buttons.
- `aria-hidden="true"` must be placed on each image simulating a radio button.
- The `aria-checked` attribute must be placed on the container of each radio button. Its value must be set dynamically according to the state of the associated radio button:
    - `aria-checked="true"` when the radio button is selected.
    - `aria-checked="false"` when the radio button is not selected.
- The group of radio buttons must be attached to the label of the group using the `aria-labelledby` attribute:
    - The label of the group should have a unique `id` value.
    - The radio button group should have an `aria-labelledby` attribute marked up with the value of the `id` attribute of the group label.

Keyboard interaction

Keyboard interactions are the same as for standard radio buttons in HTML. The only difference is that keyboard focus is placed on the container of the radio button and not only on the radio button.

Tab and **Shift** + **Tab**

When the user enters a group of radio buttons by pressing the **Tab** key, the focus moves to the selected radio button in the group. When the focus is on a selected radio button, the next tab allows the user to leave the group of radio buttons.

If no radio button is selected when the radio buttons group is opened from the keyboard, the keyboard is focused:

- On the first radio button in the group if the **Tab** key was pressed.
- On the last radio button if the **Shift** + **Tab** key combination was pressed.

Up arrow and **Left arrow**

When the focus is on one of the radio buttons, each of these arrows moves the focus to the previous radio button in the group and selects that radio button. If the focus is on the first radio button of the group and one of these arrows is pressed, then the keyboard focus moves to the last radio button in the group and selects it.

Down arrow and **Right arrow**

When the focus is on one of the radio buttons, each of these arrows moves the focus to the next radio button in the group and selects that radio button. If the focus is on the last radio button of the group and one of these arrows is pressed, then the keyboard focus moves to the first radio button in the group and selects it.

Spacebar

When the keyboard focus is on a radio button, the spacebar selects that radio button and deselects the other radio button which had previously been selected in the group.

Note

The image source and its alternative text must be modified to correspond to the state of the associated button.

Note that the `<img />` tag is provided in the example of code above, but it could be replaced with a scalable vector image `<svg>`.

Components

The "Radio button components customized with ARIA" are shown here because their level of accessibility is considered good or very good.

However, before using them in your project, it is important to check for compliancy with the specifications presented above. Certain components may require some adjustments.

Radio buttons customized using CSS

Principle

Radio buttons are form elements that allow the user to select a single option out of a group of possible options.

Radio buttons are "customized using CSS" when they are built using the standard HTML code for radio buttons as found in the specification `<input type="radio" />` but the standard radio buttons are visually hidden and replaced with CSS images, icon fonts, or specific styles.

There are several ways of achieving this. The example below shows how to simulate the behaviour of standard default HTML radio buttons using CSS pseudo-elements and background images, while ensuring that they remain accessible.

Core HTML base

```
<fieldset>
  <legend>Question</legend>
  <input type="radio" id="answer-1" name="question"
value="answer-1" />
  <label for="answer-1">Answer 1</label>
  <input type="radio" id="answer-2" name="question"
value="answer-2" />
  <label for="answer-2">Answer 2</label>
  […]
</fieldset>
```

CSS base

```
label {
  padding: 0 0 0 2rem;
  position: relative;
}

input[type=radio] {
  position: absolute;
  opacity: 0;
}

input[type=radio] + label::before,
input[type=radio] + label::after {
  content: '';
  position: absolute;
  border-radius: 50%;
```

```
}

input[type=radio] + label::before {
  left: 0.5rem;
  top: 0.2rem;
  display: inline-block;
  width: 0.8rem;
  height: 0.8rem;
  border: 0.05rem solid black;
  background: white;
}

input[type=radio]:checked + label::after {
  left: 0.7rem;
  top: 0.4rem;
  border: 0.25rem solid black;
}

input[type=radio]:focus + label::before {
  outline: 0.05rem dotted;
}
```

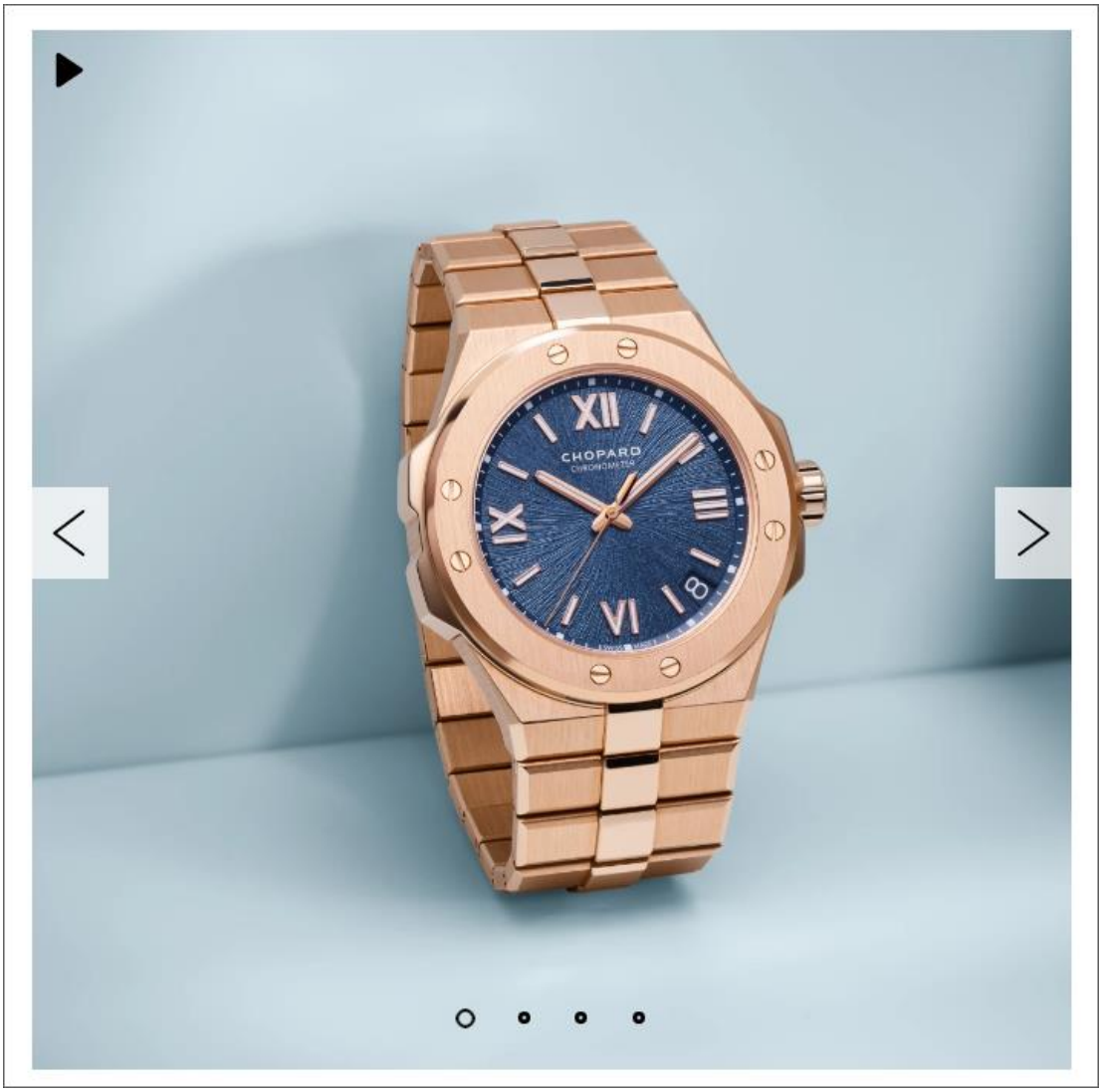
Carousels

Principle

Carousels are dynamic modules that make it possible to fit several images in a small area. They are controlled by a navigation system of scrolling panels, which is sometimes automatic.

They usually contain a visible slide with flanked by "Previous" and "Next" buttons, to scroll through the various slides of the carousel. They are often associated with page navigation buttons.

If the carousel is “auto-scrolling”, a “Pause” button is displayed to pause and resume scrolling.



This auto-scrolling carousel has a pause/play button, arrows and bullets points for navigation.

This code is based on the "Carousel with Tabs for Slide Control" design pattern found in ARIA Authoring Practices Guide (APG) of the W3C.

Core HTML base

```
<button><img src="pause.png" alt="Play slide carousel"
/></button>
<button><img src="previous.png" alt="Previous slide" /></button>
<div aria-live="polite">
  <div role="tabpanel" id="slide-1" aria-roledescription="slide"
aria-label="1 of 4">
    [Content of the first panel (shown, because associated tab is
selected)]
  </div>
  <div role="tabpanel" id="slide-2" aria-roledescription="slide"
aria-label="2 of 4">
    [Content of the second panel (hidden)]
  </div>
  <div role="tabpanel" id="slide-3" aria-roledescription="slide"
aria-label="3 of 4">
    [Contenu of the third panel (hidden)]
  </div>
  <div role="tabpanel" id="slide-4" aria-roledescription="slide"
aria-label="4 of 4">
    [Content of the fourth panel (hidden)]
  </div>
</div>
<div role="tablist" aria-label="Slides">
  <button role="tab" aria-selected="true" aria-controls="slide-
1"><img src="active-bullet.png" alt="Slide 1" /></button>
  <button role="tab" tabindex="-1" aria-selected="false" aria-
controls="slide-2"><img src="bullet.png" alt="Slide 2"
/></button>
  <button role="tab" tabindex="-1" aria-selected="false" aria-
controls="slide-3"><img src="bullet.png" alt="Slide 3"
/></button>
  <button role="tab" tabindex="-1" aria-selected="false" aria-
controls="slide-4"><img src="bullet.png" alt="Slide 4"
/></button>
</div>
<button><img src="next.png" alt="Next slide" /></button>
```

ARIA roles, states and properties

- `role="tablist"` and `aria-label="Slides"` must be placed on the element which encapsulates the tabbed interface component.

- `role="tab"` must be placed on each tab.
- The `tabindex="-1"` attribute must be placed on each tab that are not selected.
- The `aria-selected` attribute must be applied to each tab. Its value must be set dynamically according to the state of the associated tab:
  - `aria-selected="true"` on the selected tab.
  - `aria-selected="false"` on the other tabs.
- The `aria-live` attribute must be applied to the slide container. Its value must be dynamically set according to the scrolling state of the carousel:
  - `aria-live="polite"` if the carousel does not scroll automatically.
  - `aria-live="off"` if the carousel scrolls automatically.
- `role="tabpanel"`, `aria-roledescription="slide"` and `aria-label="1 of 4"` must be placed on each tab panel. The values of this last attribute must be filled in dynamically, depending on the number and total number of slides.
- Slides that are not displayed should be hidden using the CSS class `display: none;` or `visibility: hidden;`. If this is not possible, apply the following recommendations:
  - The `aria-hidden` attribute must be applied to each slide. Its value must be set dynamically according to the state of the associated slide:
    - `aria-hidden="false"` on the displayed slide.
    - `aria-hidden="true"` on the other slides.
  - `tabindex="-1"` must be applied dynamically to each interactive element in the hidden slide. This attribute must be removed from elements in the slide which are displayed.
- Each tab must be programmatically associated with its corresponding slide by the `aria-controls` attribute:
  - Each slide must have an `id` attribute set to a unique value.

- Each tab must have an `aria-controls` attribute set to the value of the `id` for the associated slide.

Keyboard interaction

Tab

When the user tabs into the tabbed interface component, this key places the focus on the selected tab in the group. When the focus is on a tab, pressing the **Tab** key allows the user to leave the tabbed interface component.

Shift + **Tab**

This key combination has the same behavior as the **Tab** key, except in the reverse order.

Left arrow

When the focus is on a tab, this key moves the keyboard focus to the previous tab in the slide show and selects this tab. If the keyboard focus is on the first tab in the group when the key is pressed, the keyboard focus moves to the last tab in the group and selects it.

Right arrow

When the focus is on a tab, this key moves the keyboard focus to the next tab in the slide show and selects this tab. If the keyboard focus is on the last tab in the group when the key is pressed, the keyboard focus moves to the first tab in the group and selects it.

Home

Moves focus to the first tab and shows the first slide.

End

Moves focus to the last tab and shows the last slide.

Enter or **Spacebar**

When the keyboard focus is on the navigation buttons, either of these keys displays the next or previous slide.

When the keyboard focus is on the pause button, either of these keys toggles between pause and play.

Expected behavior

- Among all the tabs, only one can be selected at a time and only the active tab can receive the focus.
- When an inactive tab is selected, the previously selected tab becomes inactive and the focus is on the newly selected tab.
- Only the slide(s) associated with the currently selected pagination tab are displayed. Other slides are hidden with `display: none;` or `visibility: hidden;` or optionally with `aria-hidden="true"`.
- It is not possible to tab to any interactive element in the hidden slides. If the hidden slides are not hidden thanks to CSS, the `tabindex="-1"` attribute must be added dynamically to these elements. The attribute is then removed when the associated slide is visible.
- The arrow keys are used to navigate the list of tabs and to select the current tab.
- The value of the `aria-selected` attribute must be modified dynamically each time the state of the associated tab is updated.
- The `tabindex="-1"` attribute must also be modified dynamically each time the state of the tab is updated.
- If used, the value of the `aria-hidden` attribute must also be modified dynamically each time the state of the associated slide is updated.
- The text alternative for the image button "Play slide carousel" must be updated when the image button is active, for example, "Pause slide carousel".

Notes

- Note that it is also possible to replace `<img />` in buttons with text, scalable vector graphics `<svg>`, or icon fonts.
- In the case of a carousel containing no pagination bullets, all keyboard interactions and expected behaviors specific to the tab system are to be ignored. Thus, in the core HTML base, navigation bullets will be removed, and the `role="tabpanel"` attribute present on slides will be replaced by `role="group"`.

Components

The “Carousel” components are shown here because their level of accessibility is considered good or very good.

However, before using them in your project, it is important to check for compliancy with the specifications presented above. Certain components may require some adjustments.

Checkboxes customized using ARIA

Principle

Checkboxes are form elements that are used to select a single option out of a group of possible options.

Checkboxes are "customized using ARIA" when they are not built using the standard HTML code for checkboxes as found in the specification: `<input type="checkbox" />`.

Below is a way of reproducing the behavior of a standard default HTML checkbox, **when standard HTML cannot be used**.

This code is based on the "Checkbox" design pattern found in ARIA Authoring Practices Guide (APG) of the W3C.

Core HTML base

```
<div aria-labelledby="question" role="group">
  <h2 id="question">Question</h2>
  <ul>
    <li>
      <div role="checkbox" aria-checked="false" tabindex="0">
        <img aria-hidden="true" src="checkbox.svg" alt="Not
selected: " />
        Choice 1
      </div>
    </li>
    <li>
      <div role="checkbox" aria-checked="true" tabindex="0">
        <img aria-hidden="true" src="checkbox-checked.svg"
alt="Selected: " />
        Choice 2
      </div>
    </li>
    <li>
      <div role="checkbox" aria-checked="false" tabindex="0">
        <img aria-hidden="true" src="checkbox.svg" alt="Not
selected: " />
        Choice 3
      </div>
    </li>
  </ul>
</div>
```

ARIA roles, states and properties

- `role="group"` must be added to the group of checkboxes.
- `role="checkbox"` and `tabindex="0"` must be added to the container of each checkbox.
- `aria-hidden="true"` must be applied to all the images simulating a checkbox.
- The `aria-checked` attribute must be applied to the content of each checkbox. Its value must be set dynamically according to the state of the associated checkbox:
    - `aria-checked="true"` when the check box is checked.
    - `aria-checked="false"` when the check box is not checked.
- The checkbox group must be programmatically associated with the group via the `aria-labelledby` attribute:
    - The tag for the group must have an `id` attribute set to a unique value.
    - The checkbox group must have an `aria-labelledby` attribute set to the value of the `id` attribute of the check box group.

Keyboard interaction

Keyboard interaction is the same as for classic HTML, except that the keyboard focus is on the checkbox container, and not only on the checkbox itself.

Spacebar

When the keyboard focus is on the checkbox container, this key toggles between the checked and unchecked states.

Tab

This key moves focus to each checkbox in the logical order before leaving the group.

Shift + **Tab**

This key combination has the same behavior as the **Tab** key, except in the reverse order.

Note

The image source and its alternative text must be modified according to the state of the associated checkbox.

Note that `<img />` tags can also be replaced with scalable vector graphics `<svg>`, or icon fonts.

Components

The “Checkboxes customized in ARIA” components are shown here because their level of accessibility is considered good or very good.

However, before using them in your project, it is important to check for compliancy with the specifications presented above. Certain components may require some adjustments.

Checkboxes customized using CSS

Principle

Checkboxes are form elements that are used to select one or more options out of a group of options.

Checkboxes are "customized using CSS" when they are built using the standard HTML code for radio buttons as found in the specification `<input type="checkbox" />` but the standard radio buttons are visually hidden and replaced with CSS images, icon fonts, or specific styles.

There are several ways of achieving this. The example below shows how to simulate the behavior of standard default HTML checkboxes using CSS pseudo-elements and background images, while ensuring that they remain accessible.

Core HTML base

```
<fieldset>
  <legend>Question</legend>
  <ul>
    <li>
      <input type="checkbox" id="answer-1" name="answer-1" />
      <label for="answer-1">Answer 1</label>
    </li>
    <li>
      <input type="checkbox" id="answer-2" name="answer-2" />
      <label for="answer-2">Answer 2</label>
    </li>
    […]
  </ul>
</fieldset>
```

CSS base

```
label {
  position: relative;
  padding: 0 0 0 2rem;
}

input[type=checkbox] {
  position: absolute;
  opacity: 0;
}

input[type=checkbox] + label::before,
```

```css
input[type=checkbox] + label::after {
  content: '';
  position: absolute;
  display: inline-block;
}

input[type=checkbox] + label::before {
  left: 0.5rem;
  top: 0.15rem;
  width: 0.9rem;
  height: 0.9rem;
  border: 0.05rem solid black;
  background: white;
}

input[type=checkbox]:checked + label::after {
  left: 0.6rem;
  top: 0.28rem;
  height: 0.8rem;
  border-left: 0.8rem solid black;
}

input[type=checkbox]:focus + label::before {
  outline: 0.05rem dotted;
}
```

Customized tooltips

Principle

Tooltips are messages that allow the user to obtain additional information about an item. The message is displayed on hover and also when keyboard focus is on the element.

A tooltip is "customized using ARIA" when it is not built using the standard HTML code as found in the specification, which is the title attribute.

This code is based on the "Tooltip" design pattern found in ARIA Authoring Practices Guide (APG) of the W3C.

Core HTML base

```
<a href="#" aria-describedby="extrainfo">[Link text]</a>
<div role="tooltip" id="extrainfo" >[Content of the tooltip]</div>
```

ARIA roles, states and properties

- `tabindex="0"` must be applied to the element which will cause the tooltip to be displayed, if the tooltip is not reachable by default using the keyboard.
- `role="tooltip"` must be applied to the tooltip container.
- The element which triggers the tooltip must be programmatically associated with the tooltip via the `aria-describedby` attribute:
    - The container of the tooltip should have an `id` attribute set to a unique value.
    - The element that triggers the tooltip should contain an `aria-describedby` attribute set to the value of the id attribute of the container for the tooltip.

Keyboard interaction

Esc

When the tooltip is visible, this key hides the tooltip.

Expected behavior

- The tooltip must be displayed when the triggering element:

    - Is hovered over by the mouse.
    - Receives keyboard focus.
- The tooltip must be hidden when the triggering element:
    - Is not hovered over.
    - Does not have keyboard focus.
- Hitting the **Esc** key hides the tooltip.
- The tooltip must remain visible as long as the mouse is hovering over the triggering element.
- When the tooltip is hidden, it must have `display: none;` and/or `visibility: hidden;`. Or alternatively, it can be removed from the source code.

Note

The major advantage of a custom tooltip over its standard HTML counterpart (`title` attribute) is that the custom tooltip is also accessible for keyboard users.

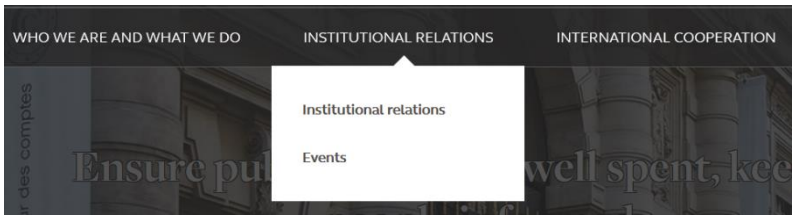
Components

Customized tooltip components are shown here because their level of accessibility is considered good or very good.

However, before using them in your project, it is important to check for compliancy with the specifications presented above. Certain components may require some adjustments.

Drop-down menu

Principle



A drop-down menu is usually a series of “buttons” displayed side by side. A sub-menu is displayed below the activated button.

Core HTML base

```
<nav role="navigation" aria-label="Main menu">
  <ul>
    <li>
      <button aria-expanded="false">First-level button with a hidden submenu</button>
      <ul class="non-visible">
        <li><a href="…">Second-level link</a></li>
        <li><a href="…">Second-level link</a></li>
        […]
      </ul>
    </li>
    <li>
      <button aria-expanded="true">First-level button with a displayed sub-menu</button>
      <ul class="visible">
        <li><a href="…">Second-level link</a></li>
        <li><a href="…">Second-level link</a></li>
        […]
      </ul>
    </li>
    <li><a href="…">First-level link</a></li>
    […]
  </ul>
</nav>
```

ARIA roles, states and properties

- The `<nav role="navigation">` tag must be used to structure the menu.
- The `aria-label` attribute must be included in the same `<nav role="navigation">` tag and set with the name of the corresponding menu.
- Nested `<ul>` and `<li>` tags must be used to structure the first-level buttons and sub-menu links.
- Each first-level button must be marked with a `<button>` tag.
- The `aria-expanded` attribute must be included in each first-level button. Its value must be dynamically set according to the status of the associated sub-menu:
    - `aria-expanded="false"` when the associated sub-menu is collapsed.
    - `aria-expanded="true"` when the associated sub-menu is expanded.

Keyboard interactions

Enter or **Spacebar**

- If the keyboard focus is positioned on a first-level button of a collapsed sub-menu, expand the associated sub-menu.
- If the keyboard focus is positioned on a first-level button on an expanded sub-menu, collapse the associated sub-menu.

Esc

If the keyboard focus is positioned on one of the items in a displayed sub-menu, this key moves the keyboard focus to the first-level button that triggered the sub menu display, and then closes the sub menu.

Note

The sub-menus that are not displayed must be hidden using `display: none` and/or `visibility: hidden`.

Hamburger menu

Core HTML base

```
<nav role="navigation" aria-label="Main menu">
  <button aria-expanded="true">
    <svg aria-hidden="true" focusable="false">[…]</svg>
    Menu
  </button>
  <ul class="visible">
    [Main navigation menu]
  </ul>
</nav>
```

ARIA roles, states and properties

- The `<nav role="navigation">` tag must be used to structure the hamburger button and menu.
- The `aria-label` attribute must be included in the same `<nav role="navigation">` tag and set with the name of the corresponding menu (e.g. `aria-label="Main menu"`).
- The hamburger button must be marked with a `<button>` tag.
- The `aria-expanded` attribute must be applied to the hamburger button that controls the menu. Its value must be set dynamically according to the status of the menu:
    - `aria-expanded="true"` when the menu is expanded.
    - `aria-expanded="false"` when the menu is collapsed.

Keyboard interactions

Enter and **Spacebar**

When the keyboard focus is positioned on the hamburger button, these keys alternately display/hide the menu.

Esc

If the keyboard focus is positioned on one of the menu items, **Esc** moves the keyboard focus to the hamburger button that triggered the menu display, and then closes it.

Expected behavior

- When the keyboard focus is positioned on the hamburger button, the menu can be displayed/hidden using the **Spacebar** and **Enter** keys. To do this, listen to the `click` event.
- When the menu is collapsed, it must be hidden using `display: none;` and/or `visibility: hidden;`.
- The default `aria-expanded` attribute value of the hamburger button must be modified dynamically each time the menu status changes.

Note

If the hamburger button is not located immediately before the HTML menu, then it is important to technically associate the menu with the hamburger button that controls it.

This association must be declared through the attribute `aria-controls`:

- The `id` attribute of the menu must have only one value.
- The hamburger button `aria-controls` attribute must have the same value as the menu's `id` attribute.

```
<nav role="navigation" aria-label="Main menu">
  <button aria-expanded="true" aria-controls="main-menu">
    <svg aria-hidden="true" focusable="false">[…]</svg>
    Menu
  </button>
  […]
  <ul id="main-menu" class="visible">
    [Main navigation menu]
  </ul>
</nav>
```

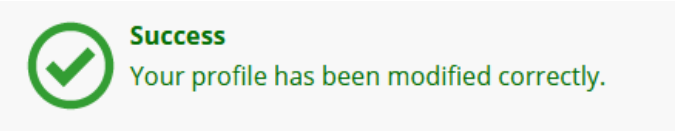
Components

The "Hamburger menu" components are shown here because their level of accessibility is considered good or very good.

However, before using them in your project, it is important to check for compliancy with the specifications presented above. Certain components may require some adjustments.

Notification messages

Principle



Notification messages are dynamic components that announce non-critical information or warnings.

They grab the reader's attention through a short message that appears all at once on the screen, without reloading the page or interrupting the activity.

Core HTML base

When the page is loaded

```
<div role="status"></div>
```

When the alert is triggered

```
<div role="status">
  <p>Your message has been sent.</p>
</div>
```

ARIA roles, states and properties

The `role="status"` attribute must be applied to the container of the notification message.

Expected behavior

The `role="status"` attribute must be statically present when the page is loaded.

The container must then be dynamically populated when the notification is triggered.

Note

Notification messages must not disappear automatically from the screen after a certain time.

They must disappear only following a deliberate action by the user (closing "X", display of a new page, etc.).

Alert messages

Principle

Alert messages are a special case of notification messages that are used to report a critical error or warning.

They grab the reader's attention through a short message that is displayed all at once on the screen, without reloading the page or interrupting the activity.

This code is based on the "Alert" design pattern found in ARIA Authoring Practices Guide (APG) of the W3C.

Core HTML base

When the page is loaded

```
<div role="alert"></div>
```

When the alert is triggered

```
<div role="alert">
  <p>Seating may not be available on this trip.</p>
</div>
```

ARIA roles, states and properties

The `role="alert"` attribute must be applied to the container of the alert message.

Expected behavior

The `role="alert"` attribute must be statically present when the page is loaded.

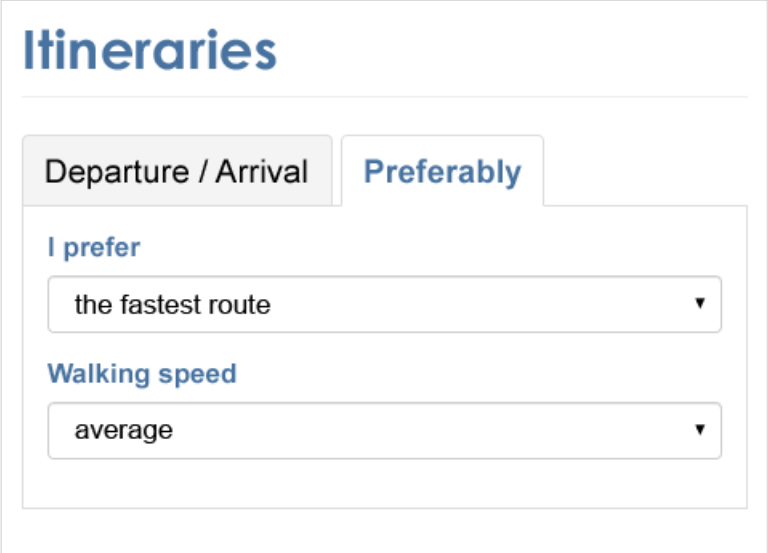
This container must then be dynamically populated when the alert is triggered.

Note

Alert messages must not disappear automatically from the screen after a certain time. They must disappear only following a deliberate action by the user (closing "X", display of a new page, etc.).

Tab panels

Principle



Tab panels are dynamic modules that optimize visible space on a web page, through a system of elements which control whether panels are visible or hidden.

They usually appear as a list of items placed next to the selected tab, designed to display content that relates to it. Only one tab can be activated at the time.

This code is based on the "Tabs with Automatic Activation" design pattern found in ARIA Authoring Practices Guide (APG) of the W3C.

Core HTML base

```
<div role="tablist">
  <button role="tab" id="tab-1" tabindex="-1" aria-
selected="false" aria-controls="panel-1">Tab 1</button>
  <button role="tab" id="tab-2" aria-selected="true" aria-
controls="panel-2">Tab 2</button>
  <button role="tab" id="tab-3" tabindex="-1" aria-
selected="false" aria-controls="panel-3">Tab 3</button>
  <button role="tab" id="tab-4" tabindex="-1" aria-
selected="false" aria-controls="panel-4">Tab 4</button>
</div>
<div role="tabpanel" id="panel-1" aria-labelledby="tab-1"
tabindex="0">
  [Content of the first panel (hidden)]
</div>
```

```
<div role="tabpanel" id="panel-2" aria-labelledby="tab-2"
tabindex="0">
  [Content of the second panel (displayed, because the associated
tab is selected)]
</div>
<div role="tabpanel" id="panel-3" aria-labelledby="tab-3"
tabindex="0">
  [Content of the third panel (hidden)]
</div>
<div role="tabpanel" id="panel-4" aria-labelledby="tab-4"
tabindex="0">
  [Content of the fourth panel (hidden)]
</div>
```

ARIA roles, states and properties

- `role="tablist"` must be placed on the element which encapsulates the tabbed interface component.

If the tabs are oriented vertically, the `aria-orientation="vertical"` attribute must also be applied.

- `role="tab"` must be placed on each tab.
- `role="tabpanel"` must be placed on each tab panel.
- The `tabindex="0"` attribute must be applied to each panel.
- Each tab must be associated with its panel via the `aria-controls` attribute:
  - Each panel should have an `id` attribute set to a unique value.
  - Each tab should have the `aria-controls` attribute set to the value of the `id` attribute of the associated panel.
- Each panel must be associated with the tab that controls it via the `aria-labelledby` attribute:
  - Each tab should have an `id` attribute set to a unique value.
  - Each panel should have an `aria-labelledby` attribute set to the value of the `id` attribute of the tab that controls it.
- The `aria-selected` attribute must be applied to each tab. Its value must be set dynamically according to the state of the associated tab:

- `aria-selected="true"` on the selected tab.
- `aria-selected="false"` on the other tabs, which have not been selected.

- The `tabindex="-1"` attribute must be placed on each non-selected tab. It must be set dynamically according to the state of the associated tab.

Keyboard interaction

Tab and **Shift** + **Tab**

When the user tabs into the tabbed interface component, the **Tab** key places the focus on the selected tab in the group. When the focus is on a tab, pressing the **Tab** key leaves the group of tabs and the focus is placed on the displayed panel.

Left arrow

When the focus is on a tab, this key moves the keyboard focus to the previous tab in the group and selects this tab. If the keyboard focus is on the first tab in the group when the key is pressed, the keyboard focus moves to the last tab in the group and selects it.

If the tabs are oriented vertically, the **Up arrow** must also have this behavior.

Right arrow

When the focus is on a tab, this key moves the keyboard focus to the next tab in the tabbed interface component and selects this tab. If the keyboard focus is on the last tab in the group, this key will move keyboard focus to the first tab in the group and selects it.

If the tabs are oriented vertically, the **Down arrow** must also have this behavior.

Note

Panels not displayed must be hidden with `display: none` and/or `visibility: hidden`.

Components

The "Tabs" components are shown here because their level of accessibility is considered good or very good.

However, before using them in your project, it is important to check for compliancy with the specifications presented above. Certain components may require some adjustments.

Show / hide panels

Principle

Expand/collapse panels are dynamic components designed to optimize the display of content in limited spaces by means of an “expand/collapse” system.

They can be controlled by the user, usually by activating a button at the top of the panel.

This code is based on the "Disclosure (Show/Hide)" design pattern found in ARIA Authoring Practices Guide (APG) of the W3C.

Core HTML base

```
<button aria-expanded="true">[Label for the button]</button>
<div class="visible">
  <p>[Contents of expanded panel]</p>
</div>
```

ARIA roles, states and properties

The `aria-expanded` attribute must be added to the button which controls the panel. Its value must be dynamically set based upon the status of the associated expanded panel:

- `aria-expanded="true"` when the associated panel is open.
- `aria-expanded="false"` when the associated panel is closed.

Keyboard interactions

Enter or **Spacebar**

When the keyboard focus is on the button, either of these keys will toggle the associated panel open or closed.

Expected behavior

- When the keyboard focus is on the button, the panel can be opened or closed using the **Spacebar** and **Enter** keys. For this, listen to the `click` event.
- When the panel is closed, it must be hidden with `display: none;` and/or `visibility: hidden;`.
- The value of the `aria-expanded` attribute must be modified dynamically each time the state of the associated panel is updated.

Note

In the particular case where the action button is not located immediately before the HTML code of the associated expanded panel, it will be necessary to facilitate access to this panel:

- On button activation, the focus is automatically moved into the associated expanded panel:
  - At the level of the first panel element, if it is interactive,
  - otherwise, at the level of the panel container (adding the `tabindex="-1"` attribute to make it focusable).
- When the focus is in the panel and comes out of it, it should be set:
  - At the level of the button that enabled it to be opened, after tabbing back following the first interactive element in the panel.
  - At the next interactive element located immediately after the button that opened it in the HTML code, after tabbing forward following the last interactive element in the panel.
- When the panel is opened, the **Escape** shortcut should be used to close it, repositioning the focus at the button.
- Associate the button with its panel via the `aria-controls` attribute:
  - The expanded panel must have an `id` attribute filled in with a unique value.
  - The button must have an `aria-controls` attribute populated with the value of the associated expanded panel's `id` attribute.

```
<button aria-expanded="true" aria-controls="expandable-
panel">[Button name]</button>
[...]
<div id="expandable-panel" class="visible" tabindex="-1">
  <p>[Content of the expandable panel whose first element is not
interactive]</p>
</div>
```

Components

The "Expandable panel" components are shown here because their level of accessibility is considered good or very good.

However, before using it in your project, it is important to check for compliancy with the specifications presented above. Depending on the version used, the components may need some adjustments before they can be used in your project.

Customized sliders

Principle

Sliders are form elements that allow the user to select a unique value, by moving a cursor on a graduated scale.

Sliders that are "customized" are not built using the standard HTML code as found in the specification `<input type="range" />`.

The following code shows how to reproduce the behavior of HTML sliders, if the native sliders cannot be used.

This code is based on the "Slider" design pattern found in ARIA Authoring Practices Guide (APG) of the W3C.

Core HTML base

```
<p id="label">Slider label</p>
<button role="slider" aria-valumin="0" aria-valuemax="10" aria-
valuenow="8" aria-labelledby="label">
  <img src="cursor.png" aria-hidden="true" alt="" />
</button>
[…]
```

ARIA roles, states and properties

- `role="slider"` must be placed on the slider.
- The `aria-valuemin` attribute must be added to the cursor. Its value must be set to the minimum value allowed by the slider.
- The `aria-valuemax` attribute must be added to the cursor. Its value must be set to the maximum value allowed by the slider.
- The `aria-valuenow` attribute must be added to the cursor. Its value must be set dynamically to the current value of the cursor.

- The cursor must be programmatically associated with its label text using the `aria-labelledby` attribute.
    - The label of the slider must have an `id` set to a unique value.
    - The cursor must have an `aria-labelledby` attribute set to the value of the `id` of the slider.
- `aria-hidden="true"` must be applied to the image of the cursor.

Keyboard interaction

The keyboard interaction is the same as for standard HTML sliders. The only exception is that the focus is set directly on the cursor and not on the whole slider.

Tab

When the user tabs into the slider, the focus is placed on the cursor, hitting the **Tab** key again causes the focus to leave the slider.

Shift + **Tab**

This key combination provides the same behavior as the **Tab** key, but in the reverse order.

Up arrow and **Right arrow**

When the keyboard focus is on the cursor, either of these keys moves the slider and increases the value by one step.

Down arrow and **Left arrow**

When the keyboard focus is on the cursor, either of these keys moves the slider and decreases the value by one step.

Home

When the keyboard focus is on the cursor, this key moves the cursor to its minimum.

End

When the keyboard focus is on the cursor, this key moves the cursor to its maximum.

Page up

When the keyboard focus is on the cursor, this key moves the cursor and increases the value by a predefined number of steps.

Page down

When the keyboard focus is on the cursor, this key moves the cursor and decreases the value by a predefined number of steps.

Expected behavior

- When the keyboard focus is on the cursor, the focus will stay on the cursor until the **Tab** key is pressed.
- When the keyboard focus is on the cursor, the slider value can be modified using the following keys: **Up arrow**, **Down arrow**, **Home**, **End**, **Page up** and **Page down**.
- The value of the `aria-valuenow` attribute must be modified dynamically to update the slider and must be identical to the value of the slider.
- The value of the slider cannot be higher than the `aria-valuemax` value.
- The value of the slider cannot be lower than the `aria-valuemin` value.

Note

The choice of the number of steps to "jump" when pressing the **Page up** and **Page down** keys is open.

On the other hand, it is important to note that digital information conveyed by the `aria-valuenow` attribute is not always clear. This may be the case, for example, when a slider is used to select one of the seven days of the week:

```
<p id="event-day">Day of the week</p>
<button role="slider" aria-valumin="0" aria-valuemax="6" aria-
valuenow="3" aria-labelledby="event-day">
  <img src="cursor.png" aria-hidden="true" alt="" />
</button>
[…]
```

In these situations, the optional `aria-valuetext` element must be used in parallel to the value, in order to translate the current value into something more understandable:

```
<p id="event-day">Day of the week</p>
<button role="slider" aria-valumin="0" aria-valuemax="6" aria-
valuenow="3" aria-valuetext="Thursday" aria-labelledby="event-
day">
  <img src="cursor.png" aria-hidden="true" alt="" />
</button>
[…]
```

If it is used, the value of the `aria-valuetext` attribute must be updated dynamically as the cursor position changes.

Note that the `<img />` tags found in the code above can be also replaced with scalable vector graphics `<svg>`, or icon fonts.

Components

The "Customized sliders" components are shown here because their level of accessibility is considered good or very good.

However, before using them in your project, it is important to check for compliancy with the specifications presented above. Certain components may require some adjustments.

Customized spinbuttons

Principle

Spinbuttons are form elements that allow the user to set a numeric value. They are presented as a text field associated with two buttons used to increase or decrease the value of the field by one increment.

Spinbuttons which are "customized" are not built using the standard HTML code as found in the specification `<input type="number" />`, but by a text field which is associated with the two images, icon fonts or specific styles, to show the increase and decrease "buttons".

The following code shows how to reproduce the behavior of HTML spinbuttons, **when the native spinbuttons cannot be used**.

This code is based on the "Spinbutton" design pattern found in ARIA Authoring Practices Guide (APG) of the W3C.

Core HTML base

```
<p id="label">Spinbutton label</p>
<div>
  <input type="text" role="spinbutton" aria-valumin="0" aria-
valuemax="10" aria-valuenow="8" aria-labelledby="label" />
  <img aria-hidden="true" src="plus.png" alt="+" />
  <img aria-hidden="true" src="minus.png" alt="-" />
</div>
```

ARIA roles, states and properties

- `role="spinbutton"` must be applied to the spinbutton.
- The `aria-valuemin` attribute must be applied to the spinbutton. Its value must be set to the minimum value allowed for the spinbutton.
- The `aria-valuemax` attribute must be applied to the spinbutton. Its value must be set to the maximum value allowed for the spinbutton.
- The `aria-valuenow` attribute must be applied to the spinbutton. Its value must be set dynamically to that of the spinbutton.

- The spinbutton must be programmatically associated with its label using `aria-labelledby`:
  - The spinbutton label should have an `id` attribute set to a unique value.
  - The spinbutton should have the `aria-labelledby` attribute set to the value of the `id` of the label of the spinbutton.
- The `aria-hidden="true"` attribute must be added to each image that simulates an increase or decrease button.

Keyboard interaction

Keyboard interaction is the same as for classic HTML spinbuttons.

Tab

When the user tabs to the spinbutton, the focus is placed in the text field. When the focus is in the text field pressing the **Tab** key allows the user to leave the spinbutton.

Shift + **Tab**

This key combination has the same behavior as the **Tab** key but in the reverse order.

Up arrow

When the focus is on the text field, this key increases the value of the text field one step.

Down arrow

When the focus is on the text field, this key decreases the value of the text field one step.

Home

When the focus is on the text field, this key decreases the value of the text field to its minimum.

End

When the focus is on the text field, this key increases the value of the text field to its maximum.

Expected behavior

- The keyboard focus remains on the text field during the use of the spinbutton, and remains there until the **Tab** key is pressed.
- When the focus is on the text field, the value of the spinbutton can be modified by typing values on the keyboard or by using the **Up arrow**, **Down arrow**, **Home** and **End**.
- The increase and decrease buttons are not keyboard focusable, but can be operated with a mouse.
- The value of the `aria-valuenow` attribute must be updated dynamically so that it is always identical to the value of the spinbutton.
- The value of the spinbutton cannot be greater than the `aria-valuemax` attribute.
- The value of the spinbutton cannot be less than the `aria-valuemin` attribute.