HowTo: Components – howto-tabs

By Ewa Gasperowicz
Ewa is a contributor to Web**Fundamentals**

By Rob Dodson
Rob is a contributor to Web**Fundamentals**

By Surma
Surma is a contributor to Web**Fundamentals**

Summary

`<howto-tabs>` limit visible content by separating it into multiple panels. Only one panel is visible at a time, while *all* corresponding tabs are always visible. To switch from one panel to another, the corresponding tab has to be selected.

By either clicking or by using the arrow keys the user can change the selection of the active tab.

If JavaScript is disabled, all panels are shown interleaved with the respective tabs. The tabs now function as headings.

Reference

- HowTo: Components on GitHub
- Tabs pattern in ARIA Authoring Practices 1.1

Demo

View live demo on GitHub

Example usage

```
<style>
  howto-tab {
    border: 1px solid black;
    padding: 20px;
  }
  howto-panel {
    padding: 20px;
    background-color: lightgray;
  }
  howto-tab[selected] {
    background-color: bisque;
  }
```

If JavaScript does not run, the element will not match `:defined`. In that case this style adds spacing between tabs and previous panel.

```
  howto-tabs:not(:defined), howto-tab:not(:defined), howto-panel:not(:defin
    display: block;
  }
</style>

<howto-tabs>
  <howto-tab role="heading" slot="tab">Tab 1</howto-tab>
  <howto-panel role="region" slot="panel">Content 1</howto-panel>
  <howto-tab role="heading" slot="tab">Tab 2</howto-tab>
  <howto-panel role="region" slot="panel">Content 2</howto-panel>
  <howto-tab role="heading" slot="tab">Tab 3</howto-tab>
  <howto-panel role="region" slot="panel">Content 3</howto-panel>
</howto-tabs>
```

Code

```
(function() {
```

Define key codes to help with handling keyboard events.

```
const KEYCODE = {
```

```
  DOWN: 40,
  LEFT: 37,
  RIGHT: 39,
  UP: 38,
  HOME: 36,
  END: 35,
};
```

To avoid invoking the parser with **`.innerHTML`** for every new instance, a template for the contents of the shadow DOM is shared by all **`<howto-tabs>`** instances.

```
const template = document.createElement('template');
template.innerHTML = `
  <style>
    :host {
      display: flex;
      flex-wrap: wrap;
    }
    ::slotted(howto-panel) {
      flex-basis: 100%;
    }
  </style>
  <slot name="tab"></slot>
  <slot name="panel"></slot>
`;
```

`HowtoTabs` is a container element for tabs and panels.

All children of **`<howto-tabs>`** should be either **`<howto-tab>`** or **`<howto-tabpanel>`**. This element is stateless, meaning that no values are cached and therefore, changes during runtime work.

```
class HowtoTabs extends HTMLElement {
```

```
  constructor() {
    super();
```

Event handlers that are not attached to this element need to be bound if they need access to `this`.

```
    this._onSlotChange = this._onSlotChange.bind(this);
```

For progressive enhancement, the markup should alternate between tabs and panels. Elements that reorder their children tend to not work well with frameworks. Instead shadow DOM is used to reorder the elements by using slots.

```
    this.attachShadow({mode: 'open'});
```

Import the shared template to create the slots for tabs and panels.

```
    this.shadowRoot.appendChild(template.content.cloneNode(true));

    this._tabSlot = this.shadowRoot.querySelector('slot[name=tab]');
    this._panelSlot = this.shadowRoot.querySelector('slot[name=panel]');
```

This element needs to react to new children as it links up tabs and panel semantically using `aria-labelledby` and `aria-controls`. New children will get slotted automatically and cause `slotchange` to fire, so not `MutationObserver` is needed.

```
    this._tabSlot.addEventListener('slotchange', this._onSlotChange);
    this._panelSlot.addEventListener('slotchange', this._onSlotChange);
  }
```

`connectedCallback()` groups tabs and panels by reordering and makes sure exactly one tab is active.

```
  connectedCallback() {
```

The element needs to do some manual input event handling to allow switching with arrow keys and Home/End.

```
    this.addEventListener('keydown', this._onKeyDown);
    this.addEventListener('click', this._onClick);

    if (!this.hasAttribute('role'))
      this.setAttribute('role', 'tablist');
```

Up until recently, `slotchange` events did not fire when an element was upgraded by the parser. For this reason, the element invokes the handler manually. Once the new behavior lands in all browsers, the code below can be removed.

```
    Promise.all([
      customElements.whenDefined('howto-tab'),
      customElements.whenDefined('howto-panel'),
    ])
      .then(_ => this._linkPanels());
  }
```

`disconnectedCallback()` removes the event listeners that `connectedCallback` added.

```
  disconnectedCallback() {

    this.removeEventListener('keydown', this._onKeyDown);
    this.removeEventListener('click', this._onClick);
  }
```

`_onSlotChange()` is called whenever an element is added or removed from one of the shadow DOM slots.

```
  _onSlotChange() {
    this._linkPanels();
  }
```

`_linkPanels()` links up tabs with their adjacent panels using `aria-controls` and `aria-labelledby`. Additionally, the method makes sure only one tab is active.

If this function becomes a bottleneck, it can be easily optimized by only handling the new elements instead of iterating over all of the element's children.

```
  _linkPanels() {
    const tabs = this._allTabs();
```

Give each panel a `aria-labelledby` attribute that refers to the tab that controls it.

```
    tabs.forEach(tab => {
      const panel = tab.nextElementSibling;
      if (panel.tagName.toLowerCase() !== 'howto-panel') {
        console.error(`Tab #${tab.id} is not a` +
          `sibling of a <howto-panel>`);
        return;
      }

      tab.setAttribute('aria-controls', panel.id);
      panel.setAttribute('aria-labelledby', tab.id);
    });
```

The element checks if any of the tabs have been marked as selected. If not, the first tab is now selected.

```
      const selectedTab =
        tabs.find(tab => tab.selected) || tabs[0];
```

Next, switch to the selected tab. **`selectTab()`** takes care of marking all other tabs as deselected and hiding all other panels.

```
      this._selectTab(selectedTab);
    }
```

`_allPanels()` returns all the panels in the tab panel. This function could memorize the result if the DOM queries ever become a performance issue. The downside of memorization is that dynamically added tabs and panels will not be handled.

This is a method and not a getter, because a getter implies that it is cheap to read.

```
    _allPanels() {
```

```
      return Array.from(this.querySelectorAll('howto-panel'));
    }
```

`_allTabs()` returns all the tabs in the tab panel.

```
    _allTabs() {
```

```
    return Array.from(this.querySelectorAll('howto-tab'));
  }
```

`_panelForTab()` returns the panel that the given tab controls.

```
  _panelForTab(tab) {
```

```
    const panelId = tab.getAttribute('aria-controls');
    return this.querySelector(`#${panelId}`);
  }
```

`_prevTab()` returns the tab that comes before the currently selected one, wrapping around when reaching the first one.

```
  _prevTab() {
```

```
    const tabs = this._allTabs();
```

Use `findIndex()` to find the index of the currently selected element and subtracts one to get the index of the previous element.

```
    let newIdx =
      tabs.findIndex(tab => tab.selected) - 1;
```

Add `tabs.length` to make sure the index is a positive number and get the modulus to wrap around if necessary.

```
    return tabs[(newIdx + tabs.length) % tabs.length];
  }
```

`_firstTab()` returns the first tab.

```
_firstTab() {
  const tabs = this._allTabs();
  return tabs[0];
}
```

`_lastTab()` returns the last tab.

```
_lastTab() {
  const tabs = this._allTabs();
  return tabs[tabs.length - 1];
}
```

`_nextTab()` gets the tab that comes after the currently selected one, wrapping around when reaching the last tab.

```
_nextTab() {
  const tabs = this._allTabs();
  let newIdx = tabs.findIndex(tab => tab.selected) + 1;
  return tabs[newIdx % tabs.length];
}
```

`reset()` marks all tabs as deselected and hides all the panels.

```
reset() {
  const tabs = this._allTabs();
  const panels = this._allPanels();

  tabs.forEach(tab => tab.selected = false);
  panels.forEach(panel => panel.hidden = true);
}
```

`_selectTab()` marks the given tab

as selected. Additionally, it unhides the panel corresponding to the given tab.

```
  _selectTab(newTab) {
```

Deselect all tabs and hide all panels.

```
    this.reset();
```

Get the panel that the `newTab` is associated with.

```
    const newPanel = this._panelForTab(newTab);
```

If that panel doesn't exist, abort.

```
    if (!newPanel)
      throw new Error(`No panel with id ${newPanelId}`);
    newTab.selected = true;
    newPanel.hidden = false;
    newTab.focus();
  }
```

`_onKeyDown()` handles key presses inside the tab panel.

```
  _onKeyDown(event) {
```

If the keypress did not originate from a tab element itself, it was a keypress inside the a panel or on empty space. Nothing to do.

```
    if (event.target.getAttribute('role') !== 'tab')
      return;
```

Don’t handle modifier shortcuts typically used by assistive technology.

```
      if (event.altKey)
        return;
```

The switch-case will determine which tab should be marked as active depending on the key that was pressed.

```
      let newTab;
      switch (event.keyCode) {
        case KEYCODE.LEFT:
        case KEYCODE.UP:
          newTab = this._prevTab();
          break;

        case KEYCODE.RIGHT:
        case KEYCODE.DOWN:
          newTab = this._nextTab();
          break;

        case KEYCODE.HOME:
          newTab = this._firstTab();
          break;

        case KEYCODE.END:
          newTab = this._lastTab();
          break;
```

Any other key press is ignored and passed back to the browser.

```
        default:
          return;
      }
```

The browser might have some native functionality bound to the arrow keys, home or end. The element calls `preventDefault()` to prevent the browser from taking any actions.

```
      event.preventDefault();
```

Select the new tab, that has been determined in the switch-case.

```
      this._selectTab(newTab);
    }
```

`_onClick()` handles clicks inside the tab panel.

```
    _onClick(event) {
```

If the click was not targeted on a tab element itself, it was a click inside the a panel or on empty space. Nothing to do.

```
      if (event.target.getAttribute('role') !== 'tab')
        return;
```

If it was on a tab element, though, select that tab.

```
      this._selectTab(event.target);
    }
  }
  customElements.define('howto-tabs', HowtoTabs);
```

`howtoTabCounter` counts the number of `<howto-tab>` instances created. The number is used to generated new, unique IDs.

```
  let howtoTabCounter = 0;
```

`HowtoTabsTab` is a tab for a `<howto-tabs>` tab panel. `<howto-tab>` should always be used with `role=heading` in the markup so that the semantics remain useable when JavaScript is failing.

A `<howto-tab>` declares which `<howto-panel>` it belongs to by using that panel's ID as the value for the `aria-controls` attribute.

A `<howto-tab>` will automatically generate a unique ID if none is specified.

```
class HowtoTab extends HTMLElement {
```

```
  static get observedAttributes() {
    return ['selected'];
  }
```

```
  constructor() {
    super();
  }
```

```
  connectedCallback() {
```

If this is executed, JavaScript is working and the element changes its role to `tab`.

```
    this.setAttribute('role', 'tab');
    if (!this.id)
      this.id = `howto-tab-generated-${howtoTabCounter++}`;
```

Set a well-defined initial state.

```
    this.setAttribute('aria-selected', 'false');
    this.setAttribute('tabindex', -1);
    this._upgradeProperty('selected');
  }
```

Check if a property has an instance value. If so, copy the value, and delete the instance property so it doesn't shadow the class property setter. Finally, pass the value to the class property setter so it can trigger any side effects. This is to safe guard against cases where, for instance, a framework may have added the element to the page and set a value on one of its properties, but lazy loaded its definition. Without this guard, the upgraded element would miss that property and the instance property would prevent the class property setter from ever being called.

```
_upgradeProperty(prop) {
```

```
  if (this.hasOwnProperty(prop)) {
    let value = this[prop];
    delete this[prop];
    this[prop] = value;
  }
}
```

Properties and their corresponding attributes should mirror one another. To this effect, the property setter for `selected` handles truthy/falsy values and reflects those to the state of the attribute. It’s important to note that there are no side effects taking place in the property setter. For example, the setter does not set `aria-selected`. Instead, that work happens in the `attributeChangedCallback`. As a general rule, make property setters very dumb, and if setting a property or attribute should cause a side effect (like setting a corresponding ARIA attribute) do that work in the **`attributeChangedCallback()`**. This will avoid having to manage complex attribute/property reentrancy scenarios.

```
  attributeChangedCallback() {

    const value = this.hasAttribute('selected');
    this.setAttribute('aria-selected', value);
    this.setAttribute('tabindex', value ? 0 : -1);
  }

  set selected(value) {
    value = Boolean(value);
    if (value)
      this.setAttribute('selected', '');
    else
      this.removeAttribute('selected');
  }

  get selected() {
    return this.hasAttribute('selected');
  }
}
customElements.define('howto-tab', HowtoTab);

let howtoPanelCounter = 0;
```

`HowtoPanel` is a panel for a **`<howto-tabs>`** tab panel.

```
class HowtoPanel extends HTMLElement {

  constructor() {
    super();
  }

  connectedCallback() {
    this.setAttribute('role', 'tabpanel');
    if (!this.id)
      this.id = `howto-panel-generated-${howtoPanelCounter++}`;
  }
}
```

```
  customElements.define('howto-panel', HowtoPanel);
})();
```

Last updated July 2, 2018.