Table of Contents

MIND Patterns - Accessibility Patterns for the Web

This book will assist frontend developers in building accessible e-commerce websites and components. It is a *living document* - so please expect continual edits and updates.

The book is work-in-progress. Some patterns are complete, but gaps in the documentation still remain. We are addressing these gaps as quickly as we can.

Pattern Philosophy

These patterns will assist developers with accessibility, but are not considered to be final, packaged code!

All patterns leave additional steps for the frontend developer to complete. This typically means any CSS styling and JavaScript behaviour *not* related to accessibility.

Each pattern follows a Progressive Enhancement strategy (where applicable) and aims to conform to WCAG 2.0 Level AA.

Pattern Organisation

There are 4 main groups of patterns:

1. **M**essaging
2. **I**nput
3. **N**avigation
4. **D**isclosure

These groups spell out the **MIND** acronym. If you ever wonder what group does a pattern fall into - then use your MIND!

Three other important groups complement our patterns: Structure, Anti-Patterns and Techniques.

Pattern Contents

Every pattern includes:

- Overview
- Working examples
- Best practices
- Interaction design
- Developer guide
- ARIA Reference

The book also contains an appendix section with a list of ARIA Essentials, References, Utilities and FAQ.

Pattern Principles

There are 4 guiding principles of accessibility, collectively know as **POUR**:

1. **P**erceivable: People experience content in different ways (sight, hearing, and touch). Content needs to be transferable into recognizable (or perceivable) formats.
2. **O**perable: Content needs to be navigable (or operable) by multiple methods—not just a mouse
3. **U**nderstandable: Web content needs to be understandable. Language should be simple and concise; functionality should be consistent and intuitive.
4. **R**obust: Create web content that works for all (or most!) technologies. This includes operating systems, browsers, and mobile devices.

From a developer perspective, **Operable** and **Robust** are the most important principles!

TL;DR?

Don't feel like reading? You can, if you wish, dive straight into the working examples.

Messaging

Patterns intended for the relaying of message(s) to the user. Form validation report, page level notices and dynamic 'as-you-type' feedback for example.

Pattern Completion/Progress

	Working Examples	Best Practices	Interaction Design	Developer Guide
Inline Notice	✓	✓	✓	✓
Flyout Notice	✕	✓	✕	✕
Form Validation	✓	✓	✓	✓
Input Meter	✓	✕	✕	✕
Input Validation	✓	✓	✓	✓
Page Notice	✓	✓	✓	✓
Time	✓	✓	✓	✕

✓ = complete ✕ = not complete

Flyout Notice

Introduction

A flyout notice is a system-activated flyout. It draws the users attention to a specific UI element on the page.

A flyout notice is typically classified as either high or low priority.

High priority notices might be errors that prevent the user from proceeding with a task.

Low priority notices are typically informational in tone.

A flyout notice might render on the server or client, although client-side rendering is most common.

Working Examples

A flyout notice example is in progress. Check back again soon.

For a real world example, you can also see the pattern in action over on eBay Skin.

Best Practices

Must remain on screen until explicitly closed or dismissed by the user.

Must contain a button to dismiss and close the flyout.

Please refer to the flyout pattern, for all inherited best practices.

Interaction Design

This section provides guidance on device interaction for keyboard, screen reader, mouse & touch.

Keyboard

Please refer to the flyout pattern, for all inherited keyboard interactions.

Screen Reader

Screen reader **must** announce high priority flyout notice when rendered on client.

Screen reader **must** contain high-priority flyout in it's list of regions when rendered on server.

Please refer to the flyout pattern, for all inherited screen reader interactions.

Mouse & Touch

Please refer to the flyout pattern, for all inherited mouse & touch interactions.

Developer Guide

Developer guide coming soon.

ebay

Form Validation

Introduction

Form validation notifies users of invalid and missing data *after* a user has attempted to submit the form.

To validate a field *before* the user attempts to submit the form, please view the Input Validation pattern. You may wish to combine both patterns.

NOTE: Do not rely on built-in browser validation of HTML5 input types (email, tel, date, etc.). HTML5 validation is not widely supported and is not always accessible.

Working Examples

You can take a look at the form validation pattern in action on our examples site.

Best Practices

Typically, any one of the following actions may occur after a form submit:

- reload same page
- load a new page
- run javascript

For errors found after page load or reload, render a server-side Page Notice above the fold.

For errors found by JavaScript, render a client side Inline Notice *above* the form.

In both cases, the notice should contain a list of errors that link directly to their respective error fields.

Interaction Design

Keyboard

Keyboard focus must be set on the error notice.

With keyboard focus on the notice, TAB key will move through links inside of notice. Activating a link will move focus to the invalid field.

Screen Reader

User must receive immediate notification of any errors preventing progress.

Invalid fields must notify user of the invalid state *and* the accompanying error message.

Developer Guide

Our implementation follows the Progressive Enhancement strategy. We build in a layered fashion that allows **everyone** to access the basic content and functionality of a web page.

The three layers are:

1. Content (HTML)
2. Presentation (CSS)
3. Behaviour (JS)

We should never depend upon JavaScript to validate a form. Our baseline functionality must be to validate the form on the server after a full page reload.

JavaScript enhances our baseline experience by validating the form without a full page reload.

Content (HTML)

Full Page Reload

A page notice ensures the notification of any errors after a full page reload. If JavaScript is available, we use focus management technique to set focus to the page notice.

Our page notice must contain links to the error fields containers:

```
<div class="page-notice page-notice--error">
    <h2>Error! Please correct the following fields:</h2>
    <ol>
        <li><a href="#age_container">Age</a> - please enter a valid age (for example,
35)</li>
        <li><a href="#shoesize_container">Shoe-size</a> - please enter a valid shoe si
ze (for example, 8.5)</li>
    </ol>
</div>
```

We must use server-side inline notices to display errors next to each field:

```
<div>
    <div id="age_container">
        <label for="age">Age</label>
        <input aria-describedby="age_notice" aria-invalid="true" id="age" name="age" t
ype="text" value="foo" />
    </div>
    <div id="age_notice">
        <span>Please enter a correct age</span>
    </div>
</div>
```

Use `aria-invalid="true"` to mark fields as invalid for assistive technology.

Enhanced Behaviour

If JavaScript is available we can intercept the form submission to prevent a full page reload.

We can prepare a main inline notice that will aggregate all errors. Place this container directly above the form:

```
<section aria-label="Form error" class="page-notice page-notice--error" tabindex="-1"
role="alert">

</section>
```

Rather than create this error container on the server, you may wish to wait and render it on the client at the appropriate time.

Presentation (CSS)

With or without a full page load, we must visually style the notice and all error fields. Please consult your design systems guide.

Remember, we can (and should) leverage ARIA states in our CSS, like so:

```
.formvalidation input[aria-invalid=true] {
    border: 1px solid red;
}
```

Here we have used the aria-invalid state to style our invalid fields with a red border. No need to create a class!

Behaviour (JS)

Full Page Reload

For validation after full page reload, we can attempt to set focus on the page notice:

```
$('.page-notice--error').attr('tabindex', '-1').focus();
```

Adding a tabindex value of -1 ensures the notice is programmatically focusable.

No Page Reload

Aggregate all errors found by the client within the main Inline Notice directly above the form:

```
<section aria-label="Form error" class="page-notice page-notice--error" tabindex="-1"
role="alert">
    <h2>Error! Please correct the following fields:</h2>
    <ol>
        <li><a href="#age_container">Age</a> - please enter a valid age (for example,
35)</li>
        <li><a href="#shoesize_container">Shoe-size</a> - please enter a valid shoe si
ze (for example, 8.5)</li>
    </ol>
</section>
```

The aggregated inline notice must contain links to the invalid fields.

Set `aria-invalid="true"` on invalid inputs.

ARIA Reference

This section gives an overview of our use of ARIA in this pattern.

aria-required

Applied to the input element to denoted a required field.

We can also use this attribute as a CSS hook.

aria-invalid

Applied to the input element to denote an invalid field.

We can also use this attribute as a CSS hook.

aria-describedby

Applied to the input element to denote the element containing the error message.

aria-live

Apply to each inline notice to notify screenreader that this content will be changed on client.

novalidate

Not an ARIA attribute, but important to apply this on your forms to disable built in validation. Built-in validation messages are not accessible and conflict with our design system guidelines.

The Future

HTML5 also promises us built-in validation on the client for new input types (number, date, url, etc). However, at the time of writing, support across devices (and for accessibility) is not good enough for us to consider their use and they should be avoided until further notice.

Inline Notice

Introduction

A user notification that appears inline next to a region or element.

An inline notice is typically classified as either high or low priority.

High priority notices might be errors that prevent the user from proceeding with a task. High priority notices are typically rendered on the client as the result of a user interaction.

Low priority messages are typically rendered by the server and are informational in tone.

Composite patterns containing Inline Notice are:

- Form Validation (client-side)
- Input Validation
- Input Meter

Working Examples

You can take a look at the inline notice pattern in action on our examples site.

For a real world example, you can also see the pattern in action over on eBay Skin.

Best Practices

Avoid having more than one high-priority notice visible at any time.

Avoid rendering high-priority inline notices on the server. Use a page notice instead. If you cannot use a page notice, then focus must be set on the inline focus.

Avoid using progress bars and spinners in conjunction with client-side notice.

A notice for a success confirmation may not always be necessary.

Interaction Design

Keyboard

Must be possible to TAB to nested interactive elements in a logical and linear order.

Screen Reader

Screen reader does not need to announce low-priority server side notices.

Screen reader must announce client-side content changes to notice.

Screen reader must announce client-side visibility change of hidden notice.

Screen reader should announce element notice when related interactive element gains focus.

Developer Guide

Our implementation follows the Progressive Enhancement strategy. We build in a layered fashion that allows **everyone** to access the basic content and functionality of a web page.

The three layers are:

1. Content (HTML)
2. Presentation (CSS)
3. Behaviour (JS)

For our example, we are going to mock up an implementation based on a typical 'Like' button you might find on any web site these days. This like button will emit a high-priority notice when any error occurs.

Content (HTML)

Let's make it work on the server first, without any JavaScript.

Form

We'll need a form, a hidden form value, and a submit button:

```
<form id="likeForm">
    <input name="like" type="hidden" value="iphone12345" />
    <input type="submit" value="Like" />
</form>
```

Clicking this button will submit the form. When the page reloads our item will either be 'liked' or an error message will appear.

At this stage, without JavaScript, the error message must be a server-side page notice, not an inline notice. Refer to the page notice developer guide for details. Our inline notice is a progressive enhancement of this baseline, server-side experience.

Structure

To add hooks for progressive enhancement, we wrap the submit button inside of a div. Then we append a live region to contain our client side message.

```
<form id="likeForm">
    <input name="like" type="hidden" value="iphone12345" />
    <div class="elementnotice" id="elementnotice-1">
        <input aria-describedby="elementnotice-1-msg" type="submit" value="Like" />
        <div aria-live="polite" id="elementnotice-1-msg">
            <!-- message text goes here -->
        </div>
    </div>
</form>
```

You may want to create this structure on the client, during an initialisation routine, rather than on the server.

Presentation (CSS)

You can style the button and inline notice according to your design system guide.

Remember to toggle the display property of the live region *contents*, not the live region itself.

```
.elementnotice-js [aria-live] p {
    display: block;
}
```

VoiceOver will detect the sub-tree visibility change and notify the user accordingly.

Behaviour (JS)

The goal of JavaScript is to:

1. Prevent the form submit action
2. Perform the service call using AJAX
3. Display any error message next to our button

Step 3 is of interest to us. We must ensure the error message is accessible.

Display Message

Assuming there was an error, we update the live region content:

```
$('.elementnotice [aria-live]').append('<p>There was an error! Please try again.</p>')
;
```

"There was an error! Please try again." is the message displayed on screen and announced by the screen reader. Keyboard focus remains on the button.

Button Description

If the user tabs away from the button, and then tabs back, it would be nice to remind the user of any error message. We can achieve this with the **aria-describedby** property:

```
$('.inlinenotice > input[type=button]').attr('aria-describedby', 'elementnotice-1-msg'
);
```

Remember to assign a unique id to the live region for this to work.

FAQ

When do I use role=region on an inline notice?

For notices rendered once on the server and visible at page load time.

When do I use role=alert on an inline notice?

For all notices rendered on the client. Or for notices rendered once on the server that will later be *updated* by the client.

When do I set keyboard focus on an inline notice?

Always set keyboard focus on high priority notices rendered by the server and visible at page load time. Again, these situations should be rare. Use a page notice instead.

For notices that contain long messages with structured or interactive elements. For example, an error notice displayed by client-side form validation.

Input Meter

Introduction

An input meter gauges user input according to a set of pre-defined levels. For example, a password strength meter may indicate "weak", "medium", "strong", etc.

Working Examples

You can take a look at the input meter pattern in action on our examples site.

Best Practices

We expect to have this section available early 2016.

Interaction Design

We expect to have this section available early 2016.

Developer Guide

We expect to have this section available early 2016.

Input Validation

Introduction

Input validation notifies users of invalid and missing data *before* a user has attempted to submit the form.

To validate a field *after* the user attempts to submit the form, please view the Form Validation pattern. You may wish to combine both patterns.

Working Examples

You can take a look at the input validation pattern in action on our examples site.

Best Practices

The validation of an input may occur on the client at any of the following points:

1. The input gains focus
    - This method is not recommended, because an input should be in a valid or invalid state *before* it receives focus.
2. The input changes value
    - This method is an intrusive way of validating an input. Consider it only in certain, exceptional circumstances. It is intrusive because a notification will fire after every key press.
3. The input loses focus
    - This is usually a good time to do validation as the user has finished entering their value and now moves onto their next task. Consider the input meter pattern for

dynamic feedback to the user after each keystroke. The user must receive polite notification of any error.

4. The form gets submitted
    - When a user attempts to submit the form that the input belongs to, the input value may be validated on the client and/or the server. For more details please consult the Form Validation Pattern.

We recommend methods #3 and #4. We cover method #3 in this pattern, and method #4 as part of the Form Validation pattern.

Interaction Design

This pattern inherits all Inline Notice interaction design.

Keyboard

If the error notice contains interactive controls (e.g. a hyperlink) it must be possible to TAB or SHIFT-TAB to these elements in a logical and linear order.

Screen Reader

The screen reader must announce any content changes that occur inside of the error notice.

The error state and error message should announce when focus is on the input.

Developer Guide

Input validation is a pattern that depends on JavaScript. It progressively enhances the form validation pattern (which validates the entire form).

Content (HTML)

JavaScript will create our actual validation content. In our base markup all we can do is surround our input element with a span to act as our hook for scripting:

```
<div class="input-validation input-validation--number">
    <span>
        <label for="age">Age</label>
        <input aria-required="false" id="age" name="age" type="text" />
    </span>
</div>
```

We use two BEM classes. The base class identifies the block, or widget type. The modifier class identifies the modifier or variation of the widget.

NOTE: We don't use HTML5 type="number" due to current cross-browser issues with HTML5 input types.

Live Region

Let's skip ahead and show the additional markup structure required for JavaScript. This markup can be created server side if you like, but is redundant if JavaScript fails.

As a user interacts and works on a form, inline error notices may become either hidden or shown. For the screen reader to detect error messages when they become visible, they must go inside of a live region:

```
<div class="input-validation input-validation--number">
    <span>
        <label for="age">Age</label>
        <input aria-describedby="age_elementnotice" aria-invalid="false" id="age" name=
"age" type="text" />
    </span>
    <span aria-live="polite" id="age_elementnotice">
        <p><!-- message content will be inserted here --></p>
    </span>
</div>
```

Also, notice the addition of `aria-describedby`. If a user tabs away from the input and then back again, this mechansism ensures the description (our error message) is announced by the screen reader again.

Presentation (CSS)

In terms of CSS and accessibility there's not a great deal we need to cover. Your error messages will be styled according to your design systems guide. Usually some shade of red will do the trick!

Border Highlight

It's good practice to visually highlight an error field with a border. Again, red is always a good choice!

```
.input-validation input[aria-invalid=true] {
    border: 1px solid red;
}
```

JavaScript will add and maintain the aria-invalid state.

Behaviour (JS)

Our pattern truly comes alive with JavaScript. Our goal is to validate on the client and give immediate feedback to the user *without* refreshing the page.

If JavaScript is unavailable, the form validation pattern should be in place as fallback.

Validation

The actual validation of input values is outside the scope of the document, but suffice to say you will need to listen to the 'input' event (or 'keydown' for IE8 and under) and call your validation routine accordingly.

If an error is found we insert and show our error message content. If an error has been fixed we remove the error message.

Again, we cannot yet use HTML5 client-side validation, so remember to add the 'novalidate' attribute to your HTML5 forms.

ARIA Reference

This section gives an overview of our use of ARIA in this pattern.

aria-required

Applied to the input element to notify screen reader that a value is required.

We can also use this attribute as a CSS hook.

aria-invalid

Applied to the input element to notify screen reader that value is invalid.

We can also use this attribute as a CSS hook.

aria-describedby

Applied to the input element in order for the screen reader to announce the related error message when focus is active.

aria-live

Applied to the error message container to notify screen reader that this content will change. A setting of 'polite' means any content changes get appended to the screen reader event queue.

novalidate

Not an ARIA attribute, but very important to apply this on your forms to disable built in validation. Why? Because the built-in validation messages are not accessible and conflict with our design system guidelines.

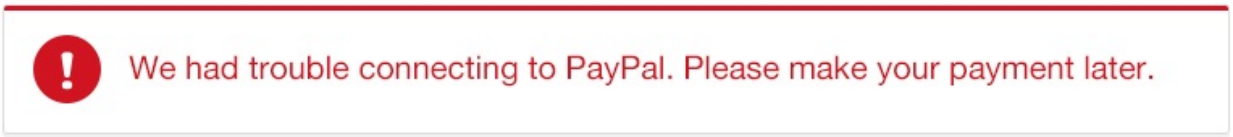
The Future

ARIA also offers us a role of status which is designed to provide status information about an element. A screen reader user can press a key to read that status message.

As of the time of writing, status is not widely supported but it may be something we re-investigate in future. For the time being aria-describedby offers similar behaviour and is more widely supported.

HTML5 also promises us built-in validation on the client for a number of new input types (number, date, url, etc). However, at the time of writing - support across devices is not good enough for us to consider using them and they should be avoided for now until further notice.

Page Notice



Introduction

User notifications which appear prominently at the top of the page.

A page notice is typically classified as either high or low priority.

Page notices are typically rendered server-side and visible at page load time.

Composite patterns containing Page Notices are:

- Form Validation (server-side)

Working Examples

You can take a look at the page notice pattern in action on our examples site.

For a real world example, you can also see the pattern in action over on eBay Skin.

Best Practices

Place page notices near the top of the page content. Ideally page notices should be next to the main page heading (i.e. the H1 tag) and always above the fold.

At most, one page notice region visible at any time.

Notices must not rely on colour alone to convey meaning or tone.

Interaction Design

Keyboard

High priority notices rendered on the server **must** receive keyboard focus. In some cases we do this for client-side rendered notices too (e.g. if the notice contains interactive elements).

Screen Reader

Screen reader must announce client-side changes to visible content.

Screen reader must announce client-side visibility change of hidden content.

Also use an icon to convey meaning or tone.

Screen reader should display notice in it's regions or landmarks list.

Developer Guide

A page notice is a custom page landmark. We use the section tag and region role to denote this semantic information:

```
<section aria-label="High priority notice" class="page-notice page-notice--error" role=
"region" tabindex="-1">
    <h2>We had trouble connecting to PayPal</h2>
    <p>Please make your payment later.</p>
</section>
```

A tabindex value of -1 allows programmatic keyboard focus for high-priority messages.

Page Alert

Notices used by client-side scripting (i.e. without a full page reload), are *alert* regions. We change role of region to alert (alert is a subclass of region).

```
<section aria-label="High priority notice" class="page-notice page-notice--error" role=
"alert" tabindex="-1">
    <h2>We had trouble connecting to PayPal</h2>
    <p>Please make your payment later.</p>
</section>
```

Now a screen reader receives notifications of any client-side changes to the notice.

FAQ

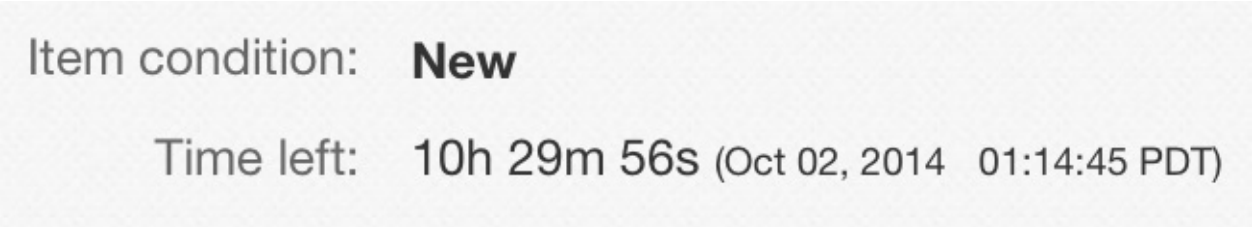
When do I set keyboard focus on a page notice?

Low priority notices never receive keyboard focus.

All high-priority page notices rendered on the server should receive keyboard focus.

Some high-priority notices rendered on the client should receive keyboard focus. Typically these are notices that contain long messages, structured data or interactive elements.

Time



Introduction

Time-based patterns can be used to indicate the elapsed time from a start point (stopwatch), the time remaining until an end point (countdown), or simply the current time (clock).

For sighted users, a clock, countdown or stopwatch will update every second visually. For non-sighted users we must be careful **not** to convey this 'ticking' of the timer aurally,

---

Working Examples

You can take a look at the page notice pattern in action on our examples site.

Best Practices

Screen reader **must not** announce updates every second because this is way too verbose/noisy, and besides, it will quickly become out of sync with the onscreen content due to lag.

We must use alerts to audibly convey only the most important thresholds. For example "Less than 30 seconds left".

Alerts must be spaced at least 15 seconds apart. This will give the screen reader enough time to announce the new value, even on the slowest screen reader speech rate, in any language.

Interaction Design

Keyboard

As the time pattern contains no interactive elements, there are no specific call-outs for keyboard interaction.

Screen Reader

Without losing their current point of focus, the screen reader must be alerted of any *critical* points in the passage of time.

Developer Guide

We expect to have this section content available early 2016.

Input

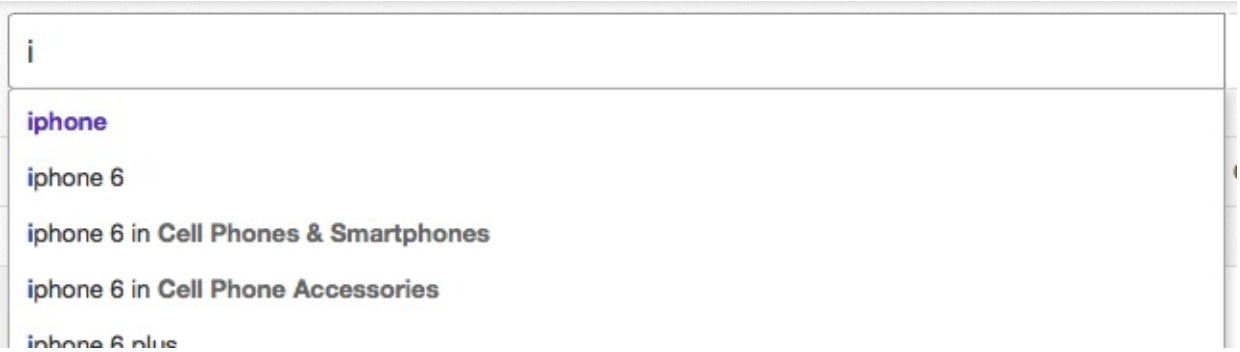
Patterns intended for the capture of user input. Radio buttons, autocomplete and drag & drop for example.

Pattern Completion/Progress

	Working Examples	Best Practices	Interaction Design	Developer Guide
Autocomplete	✓	✕	✕	✕
Button	✓	✓	✓	✕
Checkbox	✓	✕	✕	✕
Combobox	✓	✓	✓	✕
Confirm	✕	✕	✕	✕
Listbox	✕	✓	✕	✕
Menu	✓	✓	✓	✕
Radio	✓	✓	✓	✓
Star Rating	✓	✕	✕	✕
Switch	✕	✕	✕	✕

ebay

Auto-Complete



Introduction

Auto-complete adds additional behaviour to the combobox pattern. The textbox *autofills* with the value of the active list option. Up & down arrow keys change the active list option.

Examples of auto-complete are the URL bar in browsers, and the main search field in search engines.

Note that it is also possible to have an auto-complete *without* a listbox, although less common. An *inline* auto-complete adds a suggestion after the caret as the user is typing.

Updates

- Feb 22nd, 2016
  - Issue with Safari and Voiceover not announcing combobox options is now fixed.

Working Examples

You can take a look at the autocomplete pattern in action on our examples site.

Best Practices

A combobox is an enhancement of textbox. Likewise, auto-complete is an enhancement of combobox. Auto-complete should gracefully degrade to combobox behaviour. Combobox should always gracefully degrade to textbox behaviour.

We expect to have this section content complete early 2016.

Interaction Design

This section provides pattern interaction designs for different user and input types.

Keyboard

Auto-complete keyboard focus can *appear* to be in two places at the same time (the textbox and the listbox). In actual fact, keyboard focus always stays on the textbox. The aria-activedescendant property controls the pseudo-focus inside of the listbox.

With focus in the empty combobox, type any letter. If any of the available choices begin with the letter typed, those choices appear in a dropdown flyout. Any subsequent text input will further filter the list of suggestions. If the value does not match any of the available choices the drop-down list is not displayed.

With the listbox visible, up and down arrow keys will navigate the options. The textbox value will always reflect the highlighted option.

Screen Reader

NOTE: There is no common consensus for how auto-complete should behave in a screen reader. The behaviour we prescribe here may differ from other examples and guidelines you might see.

We expect to have this section content available early 2016.

Developer Guide

Auto-complete is an extension of the combobox pattern. Everything that applies to combobox pattern also applies to the autocomplete. The only addition here is that the value of the combobox must update as the user cycles through the available list options.

FAQ

Why is role=application used?

The listbox role does not seem to trigger application mode in JAWS. For this reason we force the application mode by applying the role to the root element.

How is this different from a combobox?

The only difference is that the combobox value will update as the user cycles through listbox options.

Button

Introduction

A command with one of the following possible action types:

1. Submit form
2. Reset form
3. Run JavaScript

The first two types of button are for use inside of a form, due to their specific behaviour.

The third and final type has no action, other than to fire a click event, and is not restricted to forms. It is a general purpose button created for the intention of running client side script. This type of buttons appears in the following patterns:

- Bubble-Help
- Expando
- Carousel
- Menu
- Dialog

Working Examples

You can take a look at the button in action on our examples site.

Best Practices

The button label **must never** be 'click here'.

A button label with verb and no noun is an *ambigious label*. For example, 'Add to Cart', 'Place Bid' or 'Select'. Ambiguous buttons are fine for sighted users, but can present problems to users of AT who may not be able to judge the context of the verb.

Never use a button to open a new page. Cmon, that's what a link is for!

Under certain circumstances it is okay for a button command to update the hash fragment of the URL. For example, if building a single page app experience and explicitly managing the History API.

Use buttons to open any kind of flyout, dialog or menu overlay.

Remember that buttons outside of forms do nothing when clicked until JavaScript is available. Either render the button on the client or disable the button in the server side markup before enabling it on the client.

Interaction Design

Keyboard

SPACEBAR and **ENTER** should activate button.

TAB and **SHIFT-TAB** should move to the next or previous interactive page element respectively.

Screenreader

Button must be invokable by screen reader virtual cursor (e.g. VO+SPACE in VoiceOver).

Screen reader should announce role of 'button' and the button text or accessible label.

Screen reader should announce any extra button behaviour. For example: expanded, collapsed, popup or submit.

Developer Guide

Buttons are straightforward, and 100% accessible, so long as you remember to use the button tag. Yet there are two types of button that might need extra care and attention:

1. Ambiguous Buttons

2. Links that become Buttons (hijaxed links)

TODO: add guide for icon buttons.

ARIA Reference

This section lists all relevant ARIA roles, states and properties for a button.

aria-haspopup

Informs AT that this button opens a menu when clicked.

aria-controls

Informs AT that this button controls another element.

aria-expanded

Informs AT of the expanded state of any controlled element. Used in conjunction with aria-controls.

aria-label

Creates a label for the button. **Warning!** This label will override any inner text that may be present. This property is most commonly used for critical icon buttons (i.e. buttons with icon and no text).

aria-describedby

Informs AT of any extended description or context related to this button. Note that this property has no effect on the accessible label of the button.

Checkbox

Introduction

Allows selection of zero or more items in a group of options.

HTML provides native checkboxes that are fully accessible and their use is **highly** recommended.

As we all know though, these native checkboxes are traditionally difficult to style. Our developer guide shows how use of modern CSS and icon fonts can restyle native, accessible checkboxes.

Recent Updates

- May 19th, 2015
  - Checkbox updated to a pure CSS and icont font solution. No ARIA or JavaScript required. The sole caveat is a custom keyboard focus indicator (dotted outline).

Working Examples

You can take a look at the checkbox pattern in action on our examples site.

Best Practices

If grouping two or more related checkboxes, the checkboxes should have a *group* label. Typically we use a fieldset and legend to achieve this.

We expect to have this section content available early 2016.

Interaction Design

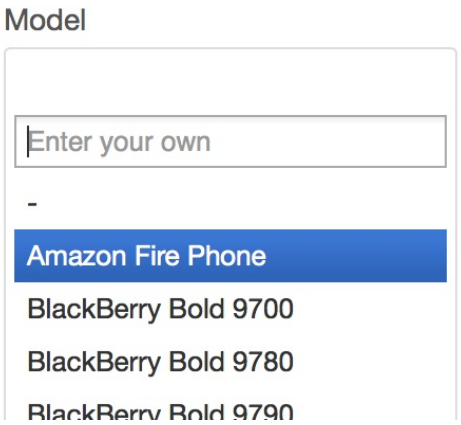
We expect to have this section content available early 2016.

Developer Guide

We expect to have this section content available early 2016.

ebay

Combobox



Introduction

An enhanced textbox that allows free text input, selection from a list, or a combination of both. Hence the name 'combobox'.

A combobox is a composite pattern containing a textbox, listbox and button.

Updates

- Feb 22nd, 2016
  - Issue with Safari and Voiceover not announcing combobox options is now fixed.
- May 13th, 2016
  - Developer Guide added

Working Example

You can take a look at the combobox pattern in action on our examples site.

You can get an idea of the required markup structure by viewing our bones project.

Best Practices

Each row in the list of options allows only a single action. It is not possible to have multiple actions per row, e.g. select, edit and delete.

Interaction Design

Keyboard

Combobox keyboard focus will *appear* to be in two places at the same time (the textbox and the listbox). In actual fact, keyboard focus always stays on the textbox. The aria-activedescendant property controls the pseudo-focus inside of the listbox.

With keyboard focus on the combobox, down arrow key will expand the listbox.

Alternatively, you may wish to auto-expand the listbox as soon as focus lands on the combobox. In this case, you may wish to exclude the button.

There is no strict requirement for the combobox button to be in the keyboard tabindex. This is because the listbox can be expanded using the down arrow key.

Up and down arrow keys navigate the list of options.

Pressing ENTER key on any option will set the combobox value and close the list of options.

Pressing ESC key will close the listbox if visible, otherwise clear the textbox value.

Screen Reader

The screen reader will announce the input as 'text edit', 'combobox' or words to those effect, depending on level of ARIA support.

When keyboard focus is on the textbox, screen reader should announce "*n* suggestions available. Use up and down arrow keys to navigate" - or words to those effect. Typically this instructional text is not provided by screen readers and so must be coded in page.

Mouse and Touch

Clicking or tapping the combobox button will toggle the listbox display.

Alterntively, you may wish to auto-expand the listbox when tapping inside the combobox input.

Clicking or tapping an option will fill the combobox with that value. For long forms, the list of options must close **without** triggering form submit.

Developer Guide

Combobox is a good example of progressive enhancement. Until JavaScript is loaded or initalised, the textbox operates as a regular textbox. For example, a user can still enter and submit a value using the plain old textbox. The ability to choose a value from a list of pre-defined options is considered the *enhancement* that will be available with JavaScript.

Textbox

We start with a label and textbox:

```
<span class="combobox" id="combobox-0">
  <label for="combobox_0-input">Game Console</label>
  <input id="combobox_0-input" name="console" type="text" placeholder="Playstation 4,
Xbox One, etc."/>
  <!-- button goes here -->
  <!-- description goes here -->
  <!-- listbox goes here -->
</span>
```

We have added our elements inside of a `.combobox` wrapper element. This wrapper acts as our module root and hook for CSS & JavaScript.

Remember: the textbox does not yet have a role of combobox, it is added later with JavaScript.

A button, description and listbox elements will be appended to this wrapper. It is up to you whether you wish to render these elements server-side or client-side. There are pros and cons to both approaches, which we will discuss below.

Button

The button should immediately follow the textbox. This button allows mouse and touch users to expand the combobox.

```
<button type="button" disabled>Expand</button>
```

It is sometimes useful to render this button on the server to avoid a flash of content or layout issues. If rendered on the server, the button should be in a disabled state. This is because the button is useless without JavaScript. The button can be enabled after all client-side initialisation for the widget is complete.

Description

At the time of writing, screen readers don't do a good job of letting the user know how to interact with a combobox or how many options it contains. To solve this problem, we can add offscreen text.

```
<span id="combobox_0-description" class="clipped">Combobox has 6 options. Use up and d
own keys to navigate.</span>
```

It is **strongly** recommended to add this description element on the client-side, and only after we are sure the widget is fully functional.

Next, in order for the screen reader to announce this description, we must add an `aria-describedby` attribute to the combobox. Again, this should be added using JavaScript on the client-side.

```
<input id="combobox_0-input" name="console" type="text" placeholder="Playstation 4, Xb
ox One, etc." aria-describedby="combobox_0-description" />
```

Now the screen reader announces the description whenever focus lands on the combobox.

Live Region

If the number of options changes based on the value of the combobox (i.e. filtering occurs) this new information must be related to assistive technology. The description element can be converted to a live region in order to achieve this.

```
<span id="combobox_0-description" class="clipped" role="status" aria-live="polite">Com
bobox has 6 options. Use up and down keys to navigate.</span>
```

The new attributes are `role=status` and `aria-live=polite`.

Note: aria-live attribute is technically redundant here, because status role has an implicit aria-live setting of `polite`. We add it to double-up on our support for older browsers and AT that might not support the status role.

Listbox

After the description, we add the listbox of options. The listbox may render on the server or the client. Like the disabled button above, it is wise to put the listbox in a semantically hidden state if rendering on the server. To do so, use the `aria-hidden` state (or HTML5 `hidden` attribute).

```
<ul id="combobox_0-listbox" role="listbox" aria-hidden="true">
  <li role="option" id="nid-0">Playstation 3</li>
  <li role="option" id="nid-1">Playstation 4</li>
  <li role="option" id="nid-2">Xbox 360</li>
  <li role="option" id="nid-3">Xbox One</li>
  <li role="option" id="nid-4">Wii</li>
  <li role="option" id="nid-5">Wii U</li>
</ul>
```

Using JavaScript we now convert the textbox to a combobox, by adding role=combobox. We also create the properties and state that connect the combobox to the listbox:

```
<input id="combobox-0-input" name="console0" type="text" placeholder="Playstation 4, X
box One, etc." role="combobox" aria-expanded="false" autocomplete="off" aria-owns="com
bobox_0-listbox" aria-describedby="combobox_0-instructions">
```

The new attributes are `role` , `aria-expanded` , `autocomplete` and `aria-owns` .

Keyboard Navigation

Our elements are now in place, but how does a keyboard user navigate to the options? We cannot use TAB key because focus must stay on the combobox (so that user can type characters). As with any widget, the answer lies in the arrow keys. Up and down arrow keys are the way to select our combobox options.

If focus must remain on the combobox, how then do we also have focus on the listbox options? The answer is that we don't. Focus always remains on the combobox and instead we have a kind of *pseudo-focus* on the options. We call the option with pseudo-focus the active descendant. And guess what? Yes, there is an ARIA attribute for this called `aria-activedescendant` . This attribute is placed on the combobox element. The attribute value is the ID of the currently active (pseudo-focussed) option.

To make all of this easier, we recommend using a plugin such as jquery-active-descendant. After your HTML structure is in place, simply activate the plugin on the widget and up/down arrow keys will update the necessary states. Use CSS to style the active descendant in any way you like.

JAWS Keyboard Navigation

At the time of writing we have found that role of combobox does not force JAWS into application mode. This means that ARROW keys will be intercepted by JAWS, thus making UP/DOWN arrow key navigation problematic. In order to force JAWS into application mode, we apply a wrapper element with `role=application`.

```
<span role="application">
  <span class="combobox" id="combobox-0">
    <!-- label, input, description, etc -->
  </span>
</span>
```

We hope that this workaround will only be a temporary measure until we see better support for combobox role in JAWS.

We expect to have this section finalised in 2016.

ARIA Reference

This section gives an overview of ARIA usage, *within the context of this pattern*.

role=application

We have found that this wrapper role is necessary for correct behaviour of keyboard accessibility in JAWS and IE.

role=combobox

This attribute changes the role of the text input from `textbox` to `combobox`. We recommend applying this attribute on the client-side with JavaScript.

role=status

This role converts the offscreen text description/instructions into an ARIA live region.

role=listbox

The list of suggestions has a role of listbox

role=option

Each listbox item has a role of option.

aria-describedby

This property connects the combobox to the offscreen description/instructions.

aria-owns

This property connects the combobox to the listbox.

aria-expanded

Conveys the expanded state of the combobox.

aria-label

Provides the expand/collapse button with an accessible label, in the case where it has no visible text (i.e. an icon button).

Confirm

Modal overlay prompting the user to confirm (e.g. yes/no) their action.

The behaviour is identical to a JavaScript confirm dialog.

We expect to have this code pattern available early 2016

Working Examples

We expect to have this section content available early 2016.

Best Practices

We expect to have this section content available early 2016.

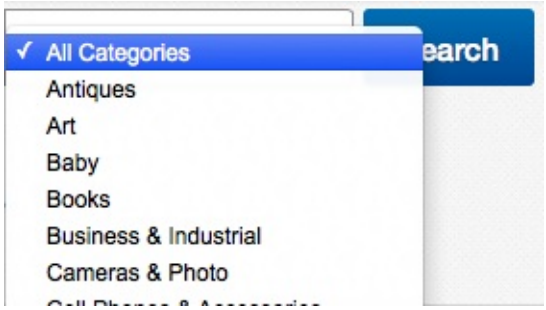
Interaction Design

We expect to have this section content available early 2016.

Developer Guide

We expect to have this section content available early 2016.

Listbox



Introduction

Allows selection of one item from a group of options.

Listbox is commonly used when requiring a form submission for full page reload.

HTML already gives us a native listbox control that is 100% keyboard and screenreader accessible without any need for JavaScript:

```
<label for="car">Choose a car</label>
<select id="car" name="car">
    <option value="volvo">Volvo</option>
    <option value="saab">Saab</option>
    <option value="opel">Opel</option>
    <option value="audi">Audi</option>
</select>
```

The implementation discussed here only applies when we wish to create a *custom* listbox. Usually the reason we do this is because CSS support is not sufficient to meet the visual design.

But first, ask yourself this question, "Does the design add something truly beneficial that is worth this extra code?" If not, you should always just stick with the native HTML implementation.

Working Example

You can take a look at the custom checkbox pattern in action on our examples site.

Best Practices

Must not use multiple select attribute. For multi-select consider a Menu or Checkboxes.

Must belong inside of a form element, and that form element **must** have a submit button.

Clicking an option in the listbox should *not* submit the form, it should only select that option.

If you require similar behaviour, but without the submit button, please consider the Menu pattern instead.

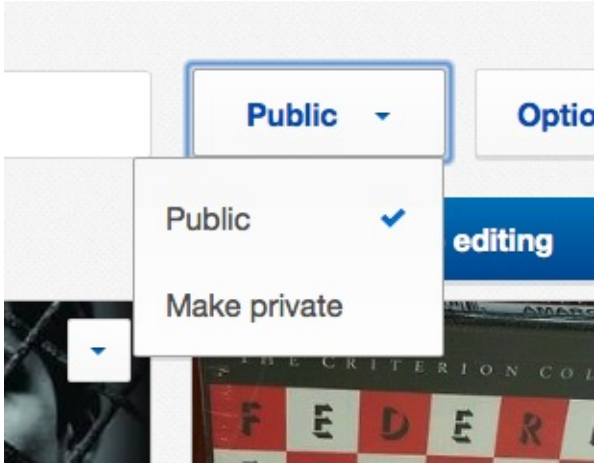
Interaction Design

We expect to have this section content available early 2016.

Developer Guide

We expect to have this section content available early 2016.

Menu



Introduction

A menu is a special kind of click-activated flyout. The flyout contains two or more menuitem commands. Menuitem commands can have menuitem, menuitemradio or menuitemcheckbox behaviour.

A menu is commonly used when requiring a partial page refresh *without* using a form or full page reload. For example: search result filtering and sorting.

If you desire links instead of menuitems, please use the Faux Menu pattern instead. The distinction between menu items and links is important! A menu item is a command that executes JavaScript, whereas a link is a command that navigates to a url.

HINT: A menu is often called a popup menu or dropdown menu, because the menu *pops-up* or *drops down* from the button. We prefer to just call it a menu, thus focussing on what it *does*, as opposed to what it *looks like*.

Working Examples

You can take a look at the menu pattern in action on our examples site.

You can get an idea of the required markup structure by viewing our bones project.

Best Practices

Selecting a menu item **must not** fully reload the page. Menu commands should perform an action on the same page (e.g. partial page re-render after AJAX).

If you need to reload the page consider a navigation or form submit pattern instead.

Interaction Design

Keyboard

ENTER, SPACEBAR or DOWN-ARROW keys will open menu when keyboard focus is on button and move keyboard focus to first menu item.

UP and DOWN arrow keys will navigate keyboard focus through the menu items.

ENTER or SPACEBAR key will activate menu items.

Pressing TAB key on menuitem will exit and close the menu.

Pressing ESC key on menuitem will close the menu and return focus to the button.

Screen Reader

Button will be announced as a popup button or popup menu button and menu's expanded state will be announced.

Menuitems will be identified as menuitem, menuitemradio or menuitemcheckbox.

Checked state of radio and checkbox menu items will be announced.

Developer Guide

It *is* technically feasible to build a progressively enhanced menu; the HTML could contain buttons, checkboxes or radios that submit a form, and with JavaScript we could enhance those form controls into menuitems, menuitemcheckboxes, and menuitemradios respectively.

However first you might want to consider the *importance* of your actual dropdownmenu functionality. Because remember that progressive enhancement is primarily concerned with ensuring that only all *important* content is available to all users.

If the functionality is based around filtering and sorting search results for example (e.g. no database updates) and not deemed critical to the user experience, then you can probably forego the progressive enhancement approach.

On the other hand, if the functionality is based around adding, modifying or deleting content (e.g. database is updated) then you should make sure this functionality is available when JavaScript is unavailable.

For our developer guide we will focus on a menu that can be used to filter search results.

Content (HTML Template)

All of our content will be created and rendered with JavaScript. None of it will be rendered on the server. Why? Well because if we render the menu HTML on the server but then JavaScript is unavailable, we end up with a broken menu button on the page. Clicking it will do nothing.

It's important to remember that any buttons outside of a form will always be a no-op without JavaScript and therefore should really be rendered on the client, then we can be sure JavaScript is available to operate the button.

Encapsulate

Like all of our other patterns, we create a container div to encapsulate the contents of our widget:

```
<div class="menu">
    <!-- widget elements go here -->
</div>
```

Button

Then we add our button:

```
<div class="menu">
    <button aria-controls="menu1" aria-expanded="false" aria-haspopup="true" type="but
ton">Search Options</button>
    <!-- menu and menuitems will go here -->
</div>
```

We use ARIA to set the buttons properties and states. It makes sense to add these ARIA attributes with JavaScript.

For further information on all ARIA attributes used in this pattern, please see the ARIA reference section at the end of this page.

Menu

Our menu will have 3 groups: a group of menuitems, a group of menuitemradios, and a group of menuitemcheckboxes. Each group must be separated with a separator tag.

```
<div class="menu">
    <button aria-controls="menu1" aria-expanded="false" aria-haspopup="true" type="but
ton">Search Options</button>
    <div id="menu1" role="menu">
        <div role="presentation">
            <div role="menuitem" tabindex="0">More Results</div>
            <div role="menuitem" tabindex="-1">Less Results</div>
        </div>
        <hr />
        <div role="presentation">
            <div aria-checked="true" role="menuitemradio" tabindex="-1">Name sorted</d
iv>
            <div aria-checked="false" role="menuitemradio" tabindex="-1">Price sorted</
div>
            <div aria-checked="false" role="menuitemradio" tabindex="-1">Date sorted</
div>
        </div>
        <hr />
        <div role="presentation">
            <div aria-checked="true" role="menuitemcheckbox" tabindex="-1">Buy It Now</
div>
            <div aria-checked="true" role="menuitemcheckbox" tabindex="-1">Auction</div
>
        </div>
    </div>
</div>
```

NOTE A div tag already has an implicit role of presentation, so why specify it again explicitly? When testing in various screen readers, I noticed more consistent behaviour with the role specified explicitly. At some point in time I will go back and review to see if this is still necessary.

We use ARIA to set the roles and states of each item.

We also initialise our rovingtabindex by ensuring only 1 menuitem (the first) is in the tab order. JavaScript will update this rovingtabindex state when the user interacts with the menu.

Ungrouped Menus

If you do not require groups and separators, you can also use the slightly more compact list-based markup to create your menu and menuitems:

```
<div class="menu">
    <button aria-controls="menu1" aria-expanded="false" aria-haspopup="true" type="but
ton">Open Menu</button>
    <ul id="menu1" role="menu">
        <li role="menuitem" tabindex="0">Button 1</li>
        <li role="menuitem" tabindex="-1">Button 2</li>
    </ul>
</div>
```

Checkpoint

At this point we have our template markup and we have our initial ARIA states and rovingtabindex state.

Presentation (CSS)

With CSS we will move the menu items inside of a hidden flyout.

We expect to have this section content available early 2016.

Behaviour (JS)

With JavaScript we will implement our keyboard and mouse interaction design and make sure our model/state is kept in sync with the view.

We expect to have this section content available early 2016.

Useful Plugins

We have some experimental jQuery plugins that may assist you with creation of an accessible menu widget:

- jquery-button-flyout
  - Useful for implementing a button that opens a non-modal flyout
- jquery-roving-tabindex
  - Useful for implementing the arrow key behaviour to change menu items

ARIA Reference

role=menu

Informs assistive technology that this is a menu containing menuitems, menuitemradios or menuitemcheckboxes.

role=presentation

Informs assistive technology that the divs around groups of menu items are for presentation purposes only and should not be added to accessibility tree.

role=menuitem

Informs assistive technology that this menu command has button behaviour.

role=menuitemradio

Informs assistive technology that this menu command has radio button behaviour.

role=menuitemcheckbox

Informs assistive technology that this menu command has checkbox behaviour.

aria-haspopup

Informs assistive technology that the button controls a menu. The name of this property is slightly misleading in that it implies it can be used for *any* kind of popup. This is not the case! This property is only for use with Popup Menus (not dialogs, tooltips, or popup navs, for example).

aria-controls

Inform assistive technology of which menu this button controls.

aria-expanded

Informs assistive technology whether the popup menu is expanded or not. And yes, this state goes on the button, not the menu.

aria-checked

Informs assistive technology whether the menuitemradio or menuitemcheckbox is checked or not. Notice we do not use aria-selected.

Radio

Introduction

Allows selection of a single item in a group of options.

If requiring multi-selection, please consider checkboxes or listbox instead.

HTML gives us native radio buttons that are fully accessible by default. Unfortunately, many designers frown upon the native radio button styling. This issue is compounded by the fact that native radio buttons are traditonally very difficult to re-style.

Fortunately for us, modern browsers give us a few tricks up our sleeve to combat these old problems.

Working Examples

You can take a look at the custom radio group pattern in action on our examples site.

You can get an idea of the required markup structure by viewing our bones project.

Best Practices

Radio buttons **must be** grouped together and a group **must contain** at least two radio buttons.

Each group of buttons **must** be labelled.

Each individual button **must** be labelled.

Native radios must belong inside of a form element, and that form element must have a submit button.

Interaction Design

This section provides the pattern interaction design for keyboard and screen reader.

If you find yourself in doubt about the interactions of a custom radio group, then simply observe how regular HTML radio buttons behave.

Keyboard

Tabbing into the group will set focus on the currently checked radio button. If no button is currently checked, focus will move to the first button or the last depending on whether TAB or SHIFT+TAB was used to enter the group.

TAB and **SHIFT+TAB** will leave the radio button group.

RIGHT ARROW and **DOWN ARROW** will focus and select the next button in group.

LEFT ARROW and **UP ARROW** will focus and select the previous button in the group.

SPACEBAR selects the currently focussed radio button. This is usually used in conjunction with CTRL+ARROW combination mentioned above.

ENTER submits the form that the radio group belongs to.

Screen Reader

Moving keyboard focus or virtual cursor onto a radio button will announce the input's:

- group label
- label
- type (radio)
- state (checked or unchecked)

The order of these announcements may vary depending on screen reader.

Developer Guide

Our developer guide will show how to build accessible radio buttons in several variations.

1. HTML Radio Buttons (native style)
2. HTML Radio Buttons (custom style)
3. ARIA Radio Buttons

We start with the most basic native HTML and increase in complexity from there.

HTML Radio Buttons (native style)

Here they are in all their glory. I hope everybody is comfortable creating native HTML radio buttons:

```
<div class="radios">
    <fieldset>
        <legend>Listing Format</legend>
        <input id="listingformat_all" type="radio" name="listingformat" value="all" />
        <label for="listingformat_all">All Listings</label>
        <input id="listingformat_auction" type="radio" name="listingformat" value="auc
tion" />
        <label for="listingformat_auction">Auction</label>
        <input id="listingformat_bin" type="radio" name="listingformat" value="bin" />
        <label for="listingformat_bin">Buy it Now</label>
    </fieldset>
</div>
```

The fieldset creates the grouping semantics. The legend creates the group label.

HTML Radio Buttons (custom style)

Native HTML radio buttons are 100% accessible by default, but may not match your design system look & feel.

In this section we show how CSS and font icons can give you the visuals you desire, *without* any JavaScript.

We are going to:

1. *Hide* the native radio button
2. Generate icon-font content in it's place
3. Assign a focus outline style to the icon-font

Invisible Input

We make the native input invisible, and keep it in place, using opacity and absolute positioning:

```
.radios input[type="radio"] {
  position: absolute;
  opacity: 0;
  z-index: 2;
}
```

We keep the radio in place, so that we can continue to click on it. Although invisible, it is still the native inputs that receive clicks and keyboard focus. This is important so that the underlying input maintains it's event handling and state.

The second thing to remember is that clicking the label of a radio button will also toggle it's checked state.

Generating Icon-Font Content

First of all, we setup our font-face, using the eBay Skin icon font:

```
@font-face {
  font-family: "vq-icon-font";
  src: url('//ir.ebaystatic.com/cr/v/c1/skin/v2.5.5/fonts/vq-icon-font.eot');
  src: url('//ir.ebaystatic.com/cr/v/c1/skin/v2.5.5/fonts/vq-icon-font.eot?#iefix') fo
rmat('embedded-opentype'), url('//ir.ebaystatic.com/cr/v/c1/skin/v2.5.5/fonts/vq-icon-
font.woff') format('woff'), url('//ir.ebaystatic.com/cr/v/c1/skin/v2.5.5/fonts/vq-icon
-font.ttf') format('truetype'), url('//ir.ebaystatic.com/cr/v/c1/skin/v2.5.5/fonts/vq-
icon-font.svg#vq-icon-font') format('svg');
  font-weight: normal;
  font-style: normal;
}
```

The generated icon-font requires absolute positioning. Therefore first set it's parent to use relative positioning:

```
.radios input[type="radio"] + label {
  position: relative;
  padding-left: 1.5rem;
}
```

We also add some left padding to account for the spacing required.

Now add the generated icon-font content before the label:

```
.radios input[type="radio"] + label::before {
  font-family: "vq-icon-font";
  position: absolute;
  content: "\e62b";
}
```

The default icon has an unchecked state. We can use the `:checked` pseudo-style to update the content when the native input is in a checked state:

```
.radios input[type="radio"]:checked + label::before {
  content: "\e62d";
 }
```

Font-Icon Focus Outline

Remember that the keyboard will always have focus on the real, native inputs. However, we cannot see the *real* focus indicator because it has 0 opacity. To workaround this, we can create a custom focus outline around the generated content:

```
.radios input[type="radio"]:focus + label::before {
  outline: 1px dotted #767676;
}
```

A dotted border is a good choice, mimicking that of various browsers (such as Firefox).

ARIA Radio Buttons

If icon-fonts are not an option, or perhaps your radios do not need to submit any form data, you may wish to consider ARIA radio buttons.

ARIA radio buttons are made up entirely from DIVs and SPANs.

```
<div class="aria-radio-group" id="aria-radios1">
    <div class="aria-radio-group__legend" id="aria-radios1__legend">Auction Type</div>
    <div class="aria-radio-group__fieldset" role="radiogroup" aria-labelledby="aria-ra
dio-group__label">
        <div role="presentation">
            <span role="radio" aria-checked="false" aria-describedby="aria-radios1__le
gend" aria-labelledby="aria-radios1_label1"></span>
            <span class="aria-radio__label" id="aria-radios1_label1" aria-hidden="true"
>All Listings</span>
        </div>
        <div role="presentation">
            <span role="radio" aria-checked="false" aria-describedby="aria-radios1__le
gend" aria-labelledby="aria-radios1_label2"></span>
            <span class="aria-radio__label" id="aria-radios1_label2" aria-hidden="true"
>Auction</span>
        </div>
        <div role="presentation">
            <span role="radio" aria-checked="false" aria-describedby="aria-radios1__le
gend" aria-labelledby="aria-radios1_label3"></span>
            <span class="aria-radio__label" id="aria-radios1_label3" aria-hidden="true"
>Buy it Now</span>
        </div>
    </div>
</div>
```

Notice the application of aria-hidden on the labels. This prevents an issue where Voiceover which treats the labels as members of the radiogroup, thus doubling the number of reported radio items in the group. Don't worry, the label will still continue to function as a labelling element. Likewise, for similar reasons, the 'legend' is also placed outside of the radiogroup.

Be warned: JavaScript will be required. In fact, a **lot** of JavaScript will be required. You will need to recreate the exact behaviour of native HTML radio buttons in full. It is no easy task.

Perhaps a better approach is to progressively enhance the native radios into ARIA radios. This way, the under-the-hood native radios continue to provide all accessibility benefits, whilst the ARIA buttons provide the custom look and feel.

ARIA Reference

This reference applies only to the ARIA radio buttons. No ARIA is needed for native HTML radio buttons.

role="radiogroup"

Informs assistive techno.logy that the fieldset element contains a group of radio buttons

role="radio"

Informs assistive technology that the span element is a radio button.

aria-hidden

Removes the native radio buttons from the accessibility tree and screen readers will ignore them (their values will still be submitted with the form).

Removes the label element from the accessibility tree for the benefit of VoiceOver which will otherwise treat the label as a member of the radiogroup role.

aria-checked

Informs the screen reader whether the radio button is in a checked or unchecked state.

aria-labelledby

Informs the screen reader of the custom radio button's label.

Informs the screen reader of the radiogroup's label.

The Future

Rather than using icon fonts or creating custom elements, what if we could just remove all the built-in styles associated with native radio button elements?

```
input[type=radio] {
    appearance: none;
}
```

Unfortunately this rule is not supported in any version of Internet Explorer.

Switch

Introduction

There are two different types of switch: a light switch and a toggle switch.

A light switch is a visual treatment of a checkbox. It is sometimes also referred to as a 'pill switch' - because it's shape often resembles a pill.

A toggle switch is a visual treatment of a radio button group.

Both types of switch can be used to store form data to be submitted to the server.

We expect to have this pattern available early 2016.

Star Rating

Introduction

Interactive star rating pattern. Typically allows user to allocate a rating of 1 to 5, and may also allow half-values.

For a non-interactive star rating (e.g. to display a global/average rating) a foreground image with alt text may be used.

Working Examples

You can take a look at the star rating pattern in action on our examples site.

More info to follow...

Navigation

Patterns that allow the user to navigate to a different URL location or page anchor.

Pattern Completion/Progress

	Working Examples	Best Practices	Interaction Design	Developer Guide
Breadcrumbs	✓	✓	✓	✓
Link	✓	✓	✓	✓
Faux Menu	✓	✓	✓	✗
Faux Tabs	✗	✗	✗	✗
Skip-To	✓	✓	✓	✓
Tile	✓	✓	✓	✗

Breadcrumbs

Home → Clothing, Shoes & Accessories → Men's Shoes → Boots

Introduction

Breadcrumbs let the user know their current position in the site hierarchy and provide a means to navigate upwards through the hierarchy chain.

The term 'breadcrumbs' comes from the trail of breadcrumbs left by Hansel and Gretel in the popular fairytale.

Working Examples

You can take a look at the breadcrumbs pattern in action on our examples site.

You can get an idea of the required markup structure by viewing our bones project.

Best Practices

Breadcrumbs are typically used as secondary navigation on sites that are organized in a hierarchical manner.

The breadcrumb pattern has two conventions that set it apart from a typical list of ordered links.

1. The 'greater than' symbol > is used to separate each link
2. The last breadcrumb should not be a link

Breadcrumbs **must** have a heading of 'You are here', or words to those effect. This heading can be hidden offscreen.

Interaction Design

Breadcrumb interaction is similar to a simple list of ordered links.

Keyboard

TAB key moves from breadcrumb to breadcrumb. The last breadcrumb is not a link and therefore must not be in the tab order.

Screen Reader

The screen reader will recognize the breadcrumbs as a navigation landmark.

Each breadcrumb will be announced as 'link', followed by the link text.

The screen reader will announce each separator symbol as 'greater than'. This is the standard eBay convention for separating breadcrumbs.

Developer Guide

Breadcrumbs require no JavaScript, they are simply an ordered list of links inside of a navigation landmark region:

```
<nav aria-labelledby="breadcrumbs_heading" class="breadcrumbs" role="navigation">
    <h2 id="breadcrumbs_heading">You are here</h2>
    <ol>
        <li><a href="http://www.ebay.com">Home</a></li>
        <li><a href="http://www.ebay.com">Clothing, Shoes &amp;amp; Accessories</a></li>
        <li><a href="http://www.ebay.com">Men's Shoes</a></li>
        <li><span>Boots</span></li>
    </ol>
</nav>
```

The heading ('You are here') can be hidden offscreen using the .clipped class.

The seperator content can be generated with CSS:

```
.breadcrumbs li::after {
    content: '>';
}
.breadcrumbs li:last-child::after {
    content: '';
}
```

This generated content is inserted between the closing and tags.

Link

A hyperlink (or 'link') points to a resource (URL) or to a specific element within a document (URL fragment).

Links are commands followed by humans or machines (a spider or crawler for example).

The web wouldn't be much of a web without links. Let's treat them with the respect they deserve by ensuring we never do anything silly to break them!

Links also form the foundation of certain progressive enhancement patterns (tabs, lightboxes, etc). In such cases JavaScript over-rides, or augments, the default behaviour of the link.

Working Examples

You can take a look at the link in action on our examples site.

Best Practices

The link text **must never** be 'click here'.

The link text **must always** adequately communicate the destination.

Links that share the same link text **must all** go to the same URL.

HREF attribute **must always** contain a valid URL.

HREF attribute **must never** be 'javascript:'. Please read the JavaScript HREF anti-pattern for more information and examples.

Link text **must not** be ambiguous to users of assistive technology. Offscreen text is the recommended approach. Please read the ambiguous label anti-pattern for more information and examples.

Link **must** warn user if it will open in a new window or tab. An icon with alternative text is the recommended approach. Please read the Open New Window anti-pattern for more information and examples.

Interaction Design

Keyboard

Link **must** activate with ENTER key (not SPACEBAR).

Link **must not** prevent default scrolling behaviour of SPACEBAR and ARROW keys.

Screen Reader

Link **must** announce with role of 'link'.

Link **must** invoke using virtual cursor command (e.g. VO+SPACE in Voiceover).

Link **must** announce 'new window or tab' behaviour *before* click.

Developer Guide

There are two types of link that need extra care and attention:

1. Ambiguous Links
2. Links that Open in a New Window or Tab

FAQ

Should I use offscreen text or aria-label?

If adding context to an ambiguous link, use offscreen text.

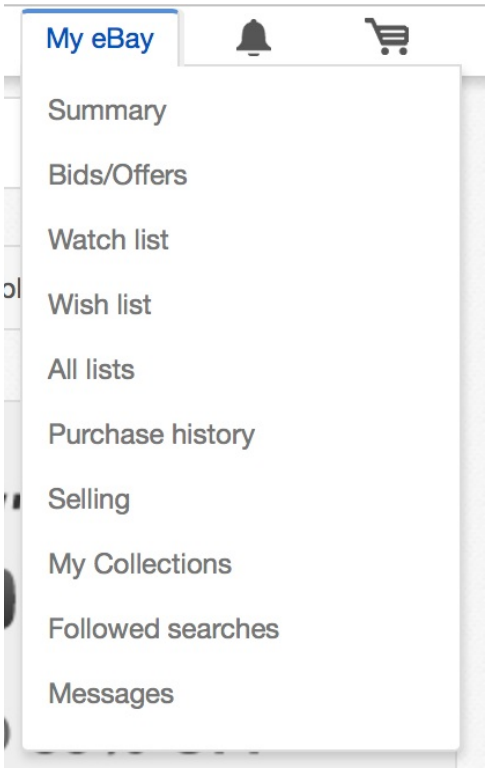
If link tag is empty (e.g. if it displays an icon), use aria-label.

However, Voiceover on iOS has problems with empty link tags. Thus, for now, we recommend always using offscreen text.

Further Reading

- http://webaim.org/techniques/hypertext/hypertext_links

Faux Menu



Introduction

A flyout containing a list of links. Clicking any link will perform a full-page navigation.

If you desire a menu that triggers a partial page reload instead of full page reload, please use the Menu pattern instead.

The distinction between menuitems and links is important! A menuitem is a command that executes JavaScript, whereas a link is a command that navigates to a url.

Working Examples

You can take a look at the faux menu pattern in action on our examples site.

You can get an idea of the required markup structure by viewing our bones site.

Best Practices

Avoid using faux menu for important and/or frequently navigated links. Links in a hidden overlay will not show up in screen reader's list of links until the overlay is visible. This issue may be circumvented to some extent by use of a navigation landmark region around the faux menu.

If your flyout must contain a mix of menuitems and links, you should use the Menu pattern to create a hybrid menu. It is more accessible to put links into a menu than putting menuitems into a faux menu.

Interaction Design

Keyboard

ENTER or SPACEBAR on button will open flyout. Keyboard focus will be moved to first link inside flyout.

TAB key will navigate links.

ENTER key will activate links.

TABBING out of flyout will close flyout.

ESC will close the flyout and return focus to the button.

Screen Reader

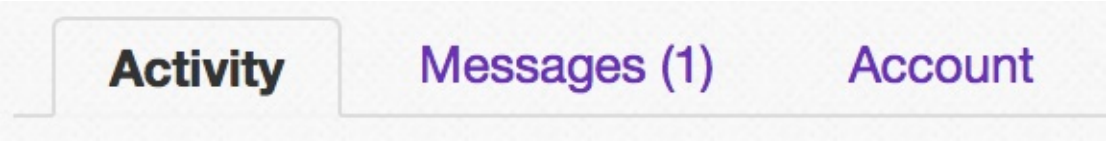
Flyout's expanded state will be announced when focus is on button.

Links **must** be identified as links (e.g. not menuitems, buttons or anything else).

Developer Guide

We expect to have this section content available early 2016.

Faux Tabs



Introduction

Style a list of links with the appearance of tabs.

Navigation or Disclosure?

Faux tabs is a navigation pattern. The "tabs" are simply a list of links with no additional javascript behaviour.

For dynamic tabs that hide & show content, please consult the disclosure tabs pattern.

Working Examples

We expect to have this section content available early 2016.

In the meantime you can consult the bones project for required markup.

Best Practices

We expect to have this section content available early 2016.

Interaction Design

Keyboard

Tab key navigates the faux tabs.

Enter key activates the faux tab (i.e. it navigates to the link href).

The 'selected' faux tab is not in the tabindex.

We expect to have this section content available early 2016.

Developer Guide

We expect to have this section content available early 2016.

In the meantime you can consult the bones project for required markup.

Pagination

Introduction

Allows user to move back and forwards through pages of results, or jump directly to any specific results page.

Working Examples

You can take a look at the breadcrumbs pattern in action on our examples site.

You can get an idea of the required markup structure by viewing our bones project.

Best Practices

We expect to have this section content available early 2016.

Interaction Design

We expect to have this section content available early 2016.

Developer Guide

We expect to have this section content available early 2016.

Skip-To

Introduction

The skip-to pattern is a simple internal link that allows users to navigate from one page region to another.

A skip-to link is most commonly used by keyboard users to skip past the main navigation in the site header. A skip-to link may also appear elsewhere on the page, as in the "see details" link on eBay's view item page.

It is also not uncommon to see a "Back to Top" link in the footer of a website. This is also an instance of the skip-to pattern.

If the skip-to link is of little use to mouse or touch users, then hide it offscreen until the moment a keyboard user tabs onto it.

Working Examples

You can take a look at the skip-to pattern in action on our examples site.

Best Practices

Must set keyboard focus on target element.

May hide itself from view until focused.

Interaction Design

For all users and devices, activating the skip to link sets keyboard focus on the target element. The browser will automatically scroll the target element into view if necessary.

Developer Guide

Our implementation follows the Progressive Enhancement strategy; we build in a layered fashion that allows **everyone** to access the basic content and functionality of a web page.

The three layers are:

1. Content (HTML)
2. Presentation (CSS)
3. Behaviour (JS)

The skip-to content is fully accessible without CSS and JavaScript.

Content (HTML)

Our content contains a source link and a target element.

Target

In our target element we specify a unique id, and set a tabindex value of -1.

```
<div id="mainContent" role="main" tabindex="-1">
    <!-- all of your main page content goes here -->
</div   >
```

The tabindex of -1 is important. It allows the browser to programmatically set focus on the target element.

Source

Our source link references the target id:

```
<a class="skipto" href="#mainContent">Skip to main content</a>
```

Checkpoint

Our HTML is now complete, and we have a fully functional skip to link without any JavaScript.

How does this work without JavaScript? If the URL contains a hash fragment, the browser will scroll to the element with the same id and set focus on the element.

Presentation (CSS)

The goal of the presentation layer is to hide the skip to link until it has keyboard focus.

Hiding Offscreen

We use a CSS clipping technique to hide the element offscreen:

```
.skipto {
    position: absolute;
    clip: rect(1px 1px 1px 1px); /* IE6, IE7 */
    clip: rect(1px, 1px, 1px, 1px);
    padding: 0;
    border: 0;
    height: 1px;
    width: 1px;
    overflow: hidden;
}
```

Revealing

To reveal, we update these styles using the `:focus` pseudo selector:

```
.skipto:focus {
    clip: none;
    height: auto;
    width: auto;
}
```

Hiding Click Outline

You may notice that if you click with your mouse on the target area, a focus indicator appears around the element. This is due to the tabindex.

We can remove this outline with CSS:

```
.skiptotarget:focus {
    outline-width: 0;
}
```

This class will be applied with JavaScript. Note that this is one of the very few times when it is okay to remove the focus outline of an element.

Showing Keyboard Outline

Of course, we must always show a keyboard focus indicator. We create another modifier class that will also be applied later using JavaScript.

```
.skiptotarget--active:focus {
    outline-width: thin;
}
```

Checkpoint

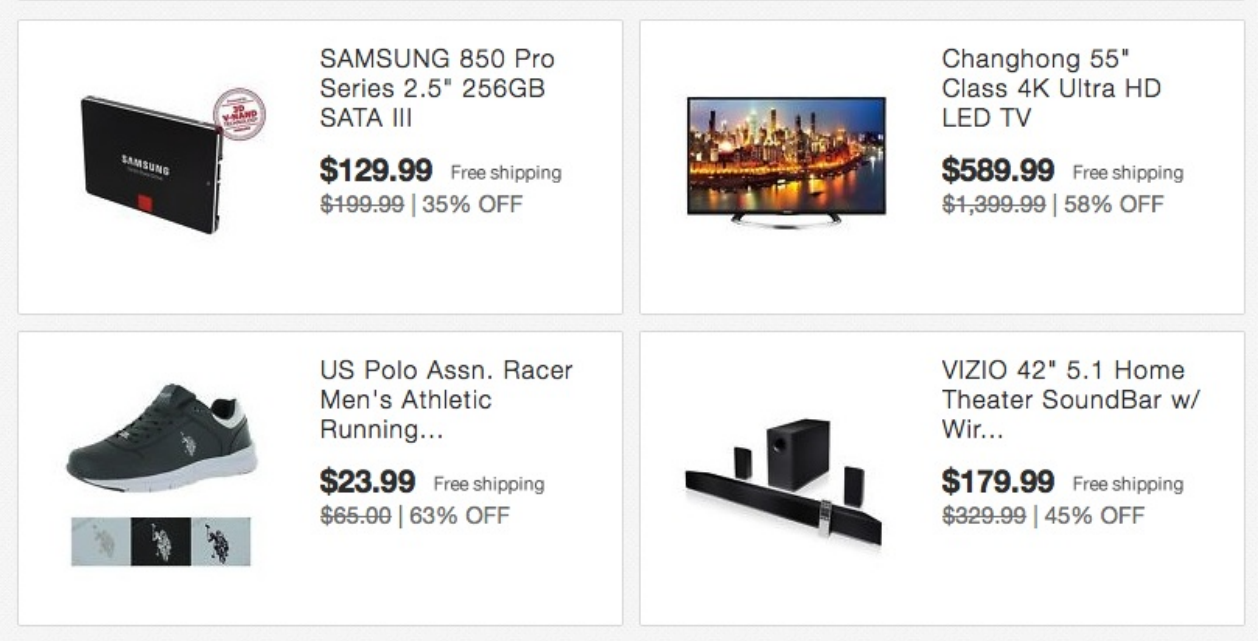
Our CSS is now complete. Because our new classes will be added with JavaScript, our functionality remains keyboard accessible.

Behaviour (JavaScript)

JavaScript allows us to only show the keyboard focus indicator when the link has been activated with keyboard.

```
$targetEl.addClass('skiptotarget');

$sourceLink.on('click', function onClick(e) {

    // add class to allow kb focus outline
    $targetEl.addClass('skiptotarget--active');

    // set programmatic focus on target element
    $targetEl.focus();

    // onblur removes class that allows outline
    $targetEl.one('blur', function(e) {
        $targetEl.removeClass('skiptotarget--active');
    });
}
```

Tile



Introduction

A tile is a large, touch-friendly area that provides a small summary-like experience of a single target URL.

Light Tile

For tiles with a small amount of text (and maybe a graphic or two), you can simply wrap these elements in an anchor tag and refer to the Link pattern We refer to this kind of tile as a *light* tile.

Heavy Tile

For tiles with longer text and structural elements such as lists, subheadings and tables, the contents cannot simply be wrapped in an anchor tag for fear of the link text becoming too verbose and unwieldy for assistive technology. We refer to this kind of tile as a *heavy* tile.

Working Examples

You can take a look at the light tile and heavy tile patterns in action on our examples site.

Best Practices

Tile can have one link destination only. Anything more than one link is considered a card.

Tile **should have** a solid outline to identify the touch region.

Light tile is simply an anchor wrapping the contents of the tile.

Light tile can not contain any interactive elements. This ensures that the tile is a single stop in the tab order.

Light tile **must not** contain long descriptive text or structural elements such as lists, tables and widgets.

Heavy tile **must only** contain one hyperlink and no other interactive elements. This ensures that the tile is a single stop in the tab order.

Interaction Design

Light tiles inherit all interaction design from the Link Pattern.

For a heavy tile, it is recommended to supress the default focus indicator of the link inside the tile, and instead display a focus indicator around the entire tile. We discuss how to do this in the developer guide below.

Developer Guide

We expect to have this section content available early 2016.

Disclosure

Patterns that disclose/reveal content to the user only when needed. Expand/collapse, tabs and menus for example.

Progressive disclosure is an interaction design technique often used in human computer interaction to help maintain the focus of a user's attention by reducing clutter, confusion, and cognitive workload. This improves usability by presenting only the minimum data required for the task at hand.

Pattern Completion/Progress

	Working Examples	Best Practices	Interaction Design	Developer Guide
Accordion	✓	✕	✕	✕
Bubble Help	✓	✓	✓	✓
Carousel	✓	✓	✓	✕
Dialog	✓	✓	✓	✓
Expando	✓	✓	✓	✕
Flyout	n/a	✓	✓	n/a
Lens	✓	✓	✓	✓
Tabs	✓	✓	✓	✓
Tooltip	✓	✓	✓	✓

Accordion

Allows inline stacking of content panels; zero or more panels of content can be visible at any time.

Often used in sidebars for the progressive disclosure of options and filters.

Working Examples

You can take a look at the accordion pattern in action on our examples site.

You can get an idea of the required markup structure by viewing our bones site.

Best Practices

We expect to have this section content available early 2016.

Interaction Design

We expect to have this section content available early 2016.

Developer Guide

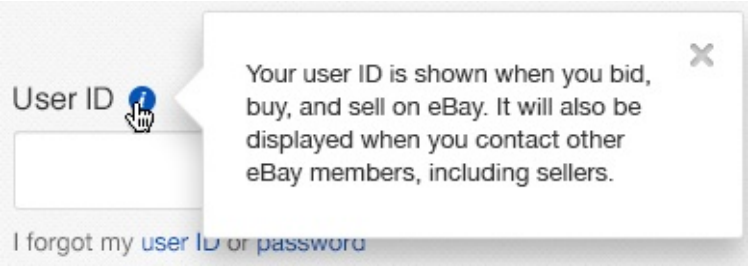
We expect to have this section content available early 2016.

Useful Plugins

We have some experimental jQuery plugins that may assist you with creation of an accessible accordion widget:

- jquery-roving-tabindex
  - Useful for implementing the arrow key behaviour to change accordion tabs

Bubble-Help



Introduction

Bubble-Help is a click-activated flyout, containing advisory or supplementary content related to an onscreen element or text.

Thematically, you can think of bubble help content fulfilling the same role as footnotes in a page.

The bubble help contains a button and an overlay. The button toggles the visibility of the overlay. The overlay contains the help content.

Working Examples

You can take a look at the bubble-help pattern in action on our examples site.

Best Practices

A good help bubble follows the principal of progressive disclosure. It should reveal the right information, at the right time.

A good help bubble should be no more than one or two paragraphs in length. For lengthier content, consider instead using a dialog.

The help bubble may contain links and images, but should **not** contain any nested controls or widgets. For example, do not place a tabs widget or carousel widget inside the Bubble Help!

The overlay should not trap keyboard focus or screen reader virtual cursor. If you desire modal behaviour please consider a dialog pattern instead.

Do not use **aria-haspopup**. Despite the name of this property, it is actually intended for menus only. Some screen readers will announce "Menu" or "Submenu", which is unsuitable for this help bubble.

Interaction Design

This section provides the pattern interaction design for keyboard, screen reader, mouse & touch.

Keyboard

Activating the button will toggle the display of the overlay. Keyboard focus remains on the button.

Pressing TAB key moves keyboard focus from the button to the first interactive element in the overlay. If the overlay contains no interactive elements, focus skips to the next interactive element on the page.

With keyboard focus on an interactive element in the overlay, ESC key will close the overlay and return focus to the button.

Screen Reader

Navigating to the button will announce the button's label (e.g. 'Help) and state (i.e. expanded/collapsed).

Activating the button will toggle expanded/collapsed state. Some screen readers will announce the button's new state automatically (expanded/collapsed).

With virtual cursor on button in expanded state, navigating virtual cursor to next item will move inside the overlay.

OPTIONAL: When bubble help is changed to 'expanded' state, screen reader will announce the overlay contents. This generally works best for bubble-helps with a small amount of content. See 'Live Region' section further below for more details.

Mouse

Clicking the button will toggle the display of the overlay.

Touch

Tapping the button will toggle the display of the overlay.

Developer Guide - Basic

Our first, basic implementation will get us quickly up and running with the accessibility requirements of the bubble help pattern. It is not designed with progressive enhancement in mind. If JavaScript is unavailable or fails to load for any reason, the button will be non-operational.

HTML

The goal of our content layer is to add the button and overlay. The structure and DOM positioning of these elements is of utmost importance.

```
<span id="bubblehelp_0" class="bubblehelp">
  <button class"bubblehelp__btn" aria-controls="bubblehelp_0-overlay" aria-expanded="f
alse" aria-label="Help" type="button"></button>
  <div class="bubblehelp__overlay" id="bubblehelp_0-overlay">
    <!-- bubble help contents go here -->
  </div>
</span>
```

The button element must be immediately followed by the overlay element. Keyboard focus and virtual cursor will move seamlessly from the button into the overlay.

CSS

The goal of the CSS layer is to style the button and overlay. How the overlay is opened and closed is of most importance to us.

```
.bubblehelp {
  position: relative;
}
.bubblehelp__overlay {
  display: none;
  position: absolute;
  z-index: 1;
}
.bubblehelp__btn[aria-expanded=true] + .bubblehelp__overlay {
  display: block;
}
```

The overlay must have `display: none` when hidden/closed. This ensures that screen readers cannot access the overlay content when in a hidden/closed state.

Notice that the display is controlled by toggling the aria-expanded attribute of the button. This ensures that the UI remains in sync with the button state.

JavaScript

The goal of the JavaScript is to hide or show the overlay when the button is toggled. This can be achieved by setting the aria-expanded state of the button.

Hide/show animations are also possible of course, but fall outside the scope of accessibility.

Developer Guide - Progressive Enhancement

Our next implementation follows the Progressive Enhancement strategy. We build in a layered fashion that allows **everyone** to access the basic content and functionality of a web page.

The bubble help begins as a simple internal hyperlink that jumps to the footnotes section of the page.

Content

The goal of our content layer is to provide related help content for any given page element.

Identify Element or Text

First we identify the page control or text that requires help content. We will use the "Gregor Samsa" text in the following paragraph:

```
<p>One morning, when Gregor Samsa woke from troubled dreams, he found himself transfor
med in his bed into a horrible vermin.</p>
```

Create Button

Our help button actually starts it's life as an anchor element. We use an internal link pointing to the section of the page containing our help content (identified by the ID of 'help1'):

```
<p>One morning, when Gregor Samsa <span class="bubblehelp"><a href="#help1"><sup>[1]</
sup></a></span> woke from troubled dreams, he found himself transformed in his bed int
o a horrible vermin.</p>
```

Notice we have also wrapped our anchor in a span element that forms the root of our **widget** for CSS and JavaScript hooks.

Add Content

We will now add our help content to the base markup. We can think of this help content as being a *footnote* in the page:

```
<section aria-labelledby="notes_heading" id="footnotes" role="region">
  <h2 id="notes_heading">Footnotes</h2>
  <ol>
    <li>
      <div id="help1">
        <p>Gregor Samsa appears to be based upon Kafka himself. As when Kafka suffered
 from <a href="http://en.wikipedia.org/wiki/Insomnia">insomnia</a>, he feared he was r
epulsive and a burden to his family, during which time his sister was his caretaker.</p
>
      </div>
    </li>
  </ol>
</div>
```

Viola! With some simple HTML, our bubble-help is already functional, albeit in a primitive fashion.

Presentation

The goal of our CSS is to style the anchor to appear more button-like and to present the help content inside of an overlay.

We must also consider that our browser maybe in a non-JavaScript situation and act accordingly. There are some things we can do safely *before* we are sure that JavaScript is available. Likewise there are some CSS rules we only want to apply *after* we are sure JavaScript has initialised the widget.

Before JS Initialisation

We can style the anchor with a sprite and hide it's text content like so:

```
.bubblehelp > a {
  content: '';
  display: inline-block;
  background: url('sprite.png') no-repeat -75px -104px;
  width: 16px;
  height: 16px;
}
.bubblehelp > a sup {
  display: none;
}
```

Note that you could also use font icons or SVG here, that is entirely up to you.

After JS Initialisation

We setup a class that will be used for the overlay:

```
.bubblehelp--js {
  position: relative;
}
.bubblehelp--js .bubblehelp__overlay {
  background: white;
  display: none;
  position: absolute;
  z-index: 1;
}
```

The important rules to note here are `position: absolute` and `z-index`. They allow our content to overlay any other element on the page.

In some situations, you may need to use `position: fixed` instead. Fixed positioning requires JavaScript to maintain overlay position when scrolling or resizing the window.

Behaviour

The goal of our JavaScript layer is to move the content into a new overlay element. Behaviour must also be added for the anchor element to control the overlay.

Locating the Content

The help content will most likely be positioned at the bottom of the page, far away from the help link.

The href attribute of our source anchor contains an ID reference to the target anchor:

```
<span class="bubblehelp">
  <a href="#help1"><sup>[1]</sup></a>
</span>
```

We can use the ID attribute to locate the content element:

```
var $this = $(this),
    $anchor = $this.find('> a'),
    href = $anchor.href,
    contentId = href.substring(href.indexOf('#')+1),
    $content = $('#' + contentId),
```

Again, remember that we do not dictate that you scrape the content of the bubble help in this fashion. Your content can come from anywhere, it can even be injected with JavaScript after an AJAX call if you wish.

Create Overlay

The content element can be re-purposed into an overlay by applying classname 'bubblehelp__overlay'.

```
$content.addClass('bubblehelp__overlay');
```

Remember that we created the rules for this class in our previous CSS step.

Live Region

To make assistive technology aware of the help content when shown, we can wrap it in a live region:

```
$liveRegion = $('<span aria-live="polite"/>');
$liveRegion.append($content);
```

Here we create a new live region element and append our content to it. A screen reader will now announce any text or visibility changes to it's contents.

NOTE: for VoiceOver to identify changes to the live region, the live region element itself remains in the DOM at all times. Any hide, show, add or remove operation must happen on the *children* of the live region, not the live region itself.

DOM Order

The live region is positioned directly after the anchor element. This ensures that the screen reader and tab order will naturally flow from button into overlay.

```
$anchor.after($liveRegion);
```

Enhancing the Anchor

We add the following roles, states and properties to the anchor element:

```
$anchor.attr('role', 'button');
$anchor.attr('aria-label', 'Help');
$anchor.attr('aria-controls', contentId);
$anchor.attr('aria-expanded', 'false');
```

To understand why these attributes are added, please consult the ARIA Reference section of this page below.

Toggle Button

The button toggle logic must toggle the overlay display **and** the ARIA expanded state.

Remember that anchor elements do not trigger 'click' events on SPACEBAR key. A spacebar event handler must be added manually.

Useful Plugins

The following jQuery plugins may assist you in creation of Bubble Help pattern:

- jquery-button-flyout

ARIA Reference

aria-live

Informs assistive technology to announce the contents of this element whenever the contents, or visibility of the contents, changes.

role=button

Informs assistive technology that our anchor element behaves like a button instead of a link.

aria-label

Provides a more accessible button label for assistive technology, over-riding the inner text of the anchor element.

aria-controls

The button *controls* the expanded state of the flyout.

aria-expanded

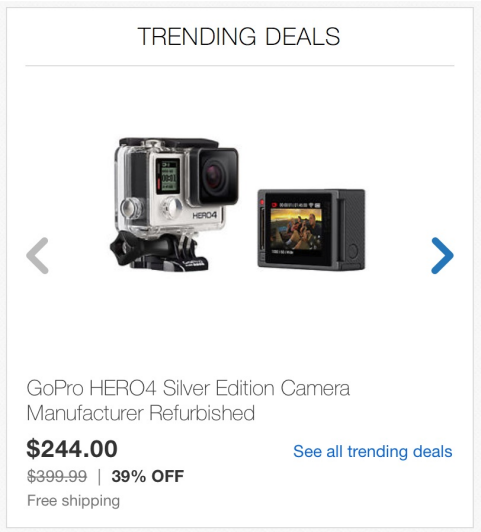
The expanded state of the button and flyout.

FAQ

This is different to the way BootStrap 'Popovers' work. Why?

The fundamental difference is that the Bootstrap Popovers get their content from a data attribute, whereas the pattern presented here gets it's content from an actual element. This allows us to more easily place HTML such as hyperlinks & images inside of our popovers and, unlike Bootstrap, the popover content will be fully accessible *without* JavaScript.

Carousel



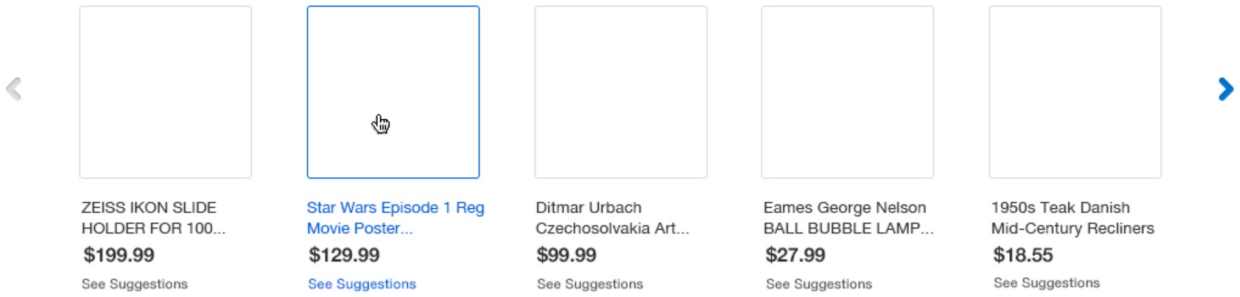
Introduction

Creates a viewport around a list of items, where *n* items are visible at any time. We say that each viewport of items constitutes a *slide*.

Two pagination buttons ('Previous' & 'Next') will move to the previous slide or next slide respectively. Extra controls may allow access to *specific* slides.

There are two variants of carousel:

- A carousel with one item in the viewport is a **Slideshow** Carousel (pictured above).
- A carousel with more than one item in the viewport is a **Filmstrip** Carousel (pictured below).



Working Examples

You can take a look at the carousel pattern in action on our examples site.

You can get an idea of the required markup structure by viewing our bones site.

Terminology

Viewport
Visible content area of the carousel. Displays a single slide.

Slide
Contains one or more items.

Item
A discrete unit of content inside of a slide.

Peek
A partially visible preview of an item on the next or previous slide. Peeks serves as a visual indicator that more slides exist. A non-visual equivalent to a Peek would be the active (or non-"aria-disabled") state of the previous or next buttons.

Pagination
Any button that changes the current slide.

Best Practices

All carousels **must** have a heading.

Carousel heading **should** end with the word "Carousel"; it can be displayed as offscreen text.

Filmstrip carousel items **must** be marked up as a list.

All slideshow carousel items **must** have a heading.

Must use buttons for all carousel controls. Controls inside items are not considered carousel controls.

Content in the viewport **must** conform to accessibility guidelines.

Peeks **must not** be focusable or interactive.

If the slides can automatically progress, then the carousel **must** include a pause/play button.

- Slideshow Carousels **should not** start automatically
- Filmstrip Carousels **must not** start automatically
- Automatic progression **must** pause when keyboard focus enters widget
- Automatic progression **must** pause when mouse hovers over widget

Interaction Design

Keyboard

Pressing TAB key while focus is on 'Previous' button **must** move keyboard focus to first focus-able element in view port.

Items off-screen, outside of view port, **must not** be keyboard focus-able. Keyboard user must use 'Previous' and 'Next' buttons to control the viewport.

Pressing SHIFT+TAB key while focus is on 'Next' button **must** move keyboard focus to last focus-able element in view port.

When on first slide, the 'Previous' button should remain in tab-order but **must** be visually disabled.

When on last slide, the 'Next' button should remain in tab-order but **must** be visually disabled.

Keyboard focus order summary:

1. Previous button
2. Controls (links, buttons, etc) belonging to items visible in the viewport
3. Next button
4. Pause/Play button, if it exists
5. Jump to slide pagination buttons, if it exists

Focus management summary:

- Activate previous or next button: focus **must** stay on Previous or Next button.
- Activate Pause/Play button: focus **must** stay on the Pause/Play button.

Screen Reader

Previous and Next buttons **must** have label “Show Previous Trending Deals”, or words to that effect.

Items outside of the viewport **must not** be reachable with the virtual cursor.

When on first slide, screen reader **must** announce 'Previous' button as disabled.

When on first slide, screen reader **must** announce 'Next' button as disabled.

When moving virtual cursor from item to item, screen reader *might* announce list index position.

When activated, the previous and next buttons **must** announce which items are in the viewport. Example: "Trending Deals now showing slide 4 of 6".

Play button label **must** be “Play Trending Deals”, or words to that effect.

Pause button label **must** be “Pause Trending Deals”, or words to that effect.

When autoplay changes slides, screen reader **must** announce what changed. Example: “Trending Deals now showing slide 4 of 6”. See role=status in the ARIA reference below.

Mouse

We expect to have this section content available mid to late 2016.

Touch

We expect to have this section content available mid to late 2016.

Developer Guide

We expect to have this section content available mid to late 2016.

ARIA Reference

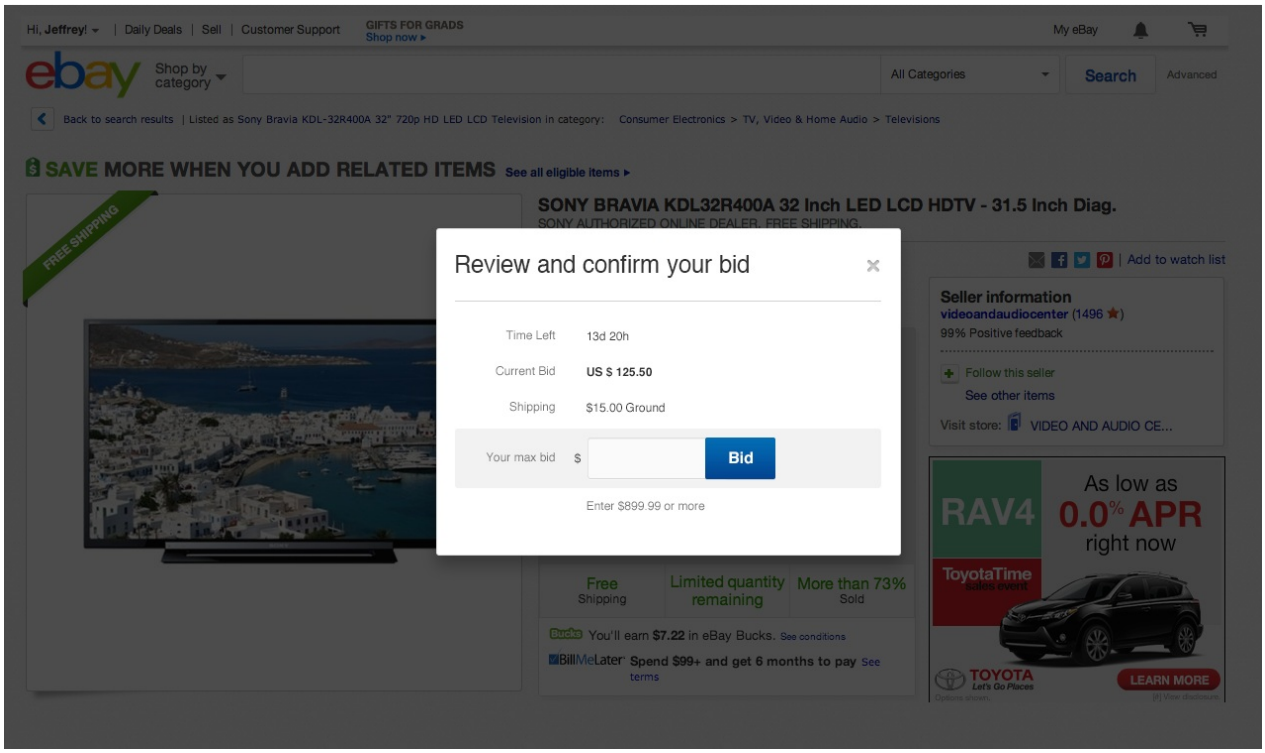
aria-disabled

Used to notify user that carousel controls are in a disabled state. Screen readers *might* announce 'disabled,' 'dimmed,' 'unavailable,' or similar.

role=status

Used to create a live status region for updates to carousel viewport, e.g "Trending Deals Carousel now showing slide 4 of 6”.

Dialog



Introduction

Heavy, modal overlay used for Messaging, Input or Progressive Disclosure of content.

This content could be a prompt for user input, an important message, or even a full page-like experience.

Dialog is a *base*-pattern, with the following concrete implementations:

- Alert
- Confirm
- Input
- Lens

Working Examples

You can take a look at the dialog pattern in action on our examples site.

You can get an idea of the required markup structure by viewing our bones site.

Best Practices

A dialog is typically opened in one of two ways:

- **Click-Activated** Explicitly opened and closed by clicking an associated button control
- **System-Activated** Automatically opened by the system/application on page load or at some other arbitrary time

Opening dialogs that are not requested by user (i.e. system-activated) are a violation of WCAG Guideline 3.2.5 (Level AAA) and therefore should be reserved for exceptional circumstances only.

Must only be opened on click event or system event.

Must never be opened on hover/focus event.

Must contain a close button.

Must contain at least one interactive element (this can be the close button).

Must be labelled by an onscreen element (a heading for example) or explicit attribute.

Must start heading hierarchy at level 2.

Must trap keyboard focus until closed.

Must add high-contrast mask over page background.

Must use appropriate ARIA role of dialog or alertdialog.

If the dialog content is deemed critical then it **must be** available without JavaScript, either at an alternative URL or elsewhere on the same page.

Interactions

Keyboard

Pressing SPACE or ENTER on associated button will open the click-activated dialog.

When dialog is opened, keyboard focus is immediately set on an interactive element inside of window (this is usually the close button).

Pressing TAB or SHIFT+TAB will cycle through all interactive elements inside of the dialog.

Pressing ENTER or SPACE on close button, or pressing ESC at any time, will hide dialog; keyboard focus is returned to the element that had focus before dialog was opened.

Screen Reader

For click-activated dialog, screen reader reads the associated button label (no other aria attributes are required, e.g. do not use aria-haspopup).

When dialog opens, contents and role (dialog or alertdialog) are announced by screen reader, followed by the label of interactive control that gains keyboard focus.

Reads close button label when close button has focus.

Virtual cursor is confined to the contents of the dialog.

Announces "Leaving dialog" (or words to those affect) when keyboard focus leaves dialog (i.e. when dialog is closed).

Pointer

Clicking the close button will hide the dialog.

Developer Guide

Content (HTML)

Trigger Element

We are going to add a button element that triggers a dialog when clicked:

```
<button type="button" class="dialog-button">Open Dialog</button>
```

Dialog Role

The dialog element itself must have a role of dialog:

```
<div role="dialog">

</div>
```

Document

To assist older screen reader versions (NVDA in particular), the immediate child of a dialog must have a role of document:

```
<div role="dialog">
    <div role="document">

    </div>
</div>
```

In time, as screen reader support improves, the requirement for this role may go away.

Header and Body

Next we create the header and body structure:

```
<div role="dialog">
    <div role="document">
        <header role="banner">

        </header>
        <div>

        </div>
    </div>
</div>
```

Dialog Label and Close Button

Every dialog must be labelled. We can use aria-labelledby to point to a suitable labelling element inside of our dialog.

Every dialog requires a close button, we must make this the first interactive element in the dialog.

```
<div role="dialog">
    <div role="document">
        <header role="banner">
          <h2 id="dialog_title">Dialog Example</h2>
          <button aria-label="Close Dialog">X</button>
        </header>
        <div>
            <!-- dialog contents go here -->
        </div>
    </div>
</div>
```

Alternatively, if no suitable heading or labelling element exists, we can specify the label in an aria-label property instead:

```
<div aria-label="Dialog Example" role="dialog">
```

Don't forget to internationalize this attribute text!

Presentation (CSS)

Once we have our markup in place, there are many different ways to visually, and uniquely, style a dialog.

Visually, there is only one thing that all dialogs require, and that is a background mask.

Background Mask

In order to make a visual distinction between the dialog content in the foreground, and the page content in the background, we must create a high-contrast mask between them:

```
[role=dialog] {
    background-color: rgba(0,0,0,0.85);
    height: 100%;
    left: 0;
    position: fixed;
    top: 0;
    width: 100%;
}
```

By design, this mask prevents mouse and touch from interacting with the underlying page. It does not prevent keyboard or screen reader from going there.

Behaviour (JS)

No matter how a dialog is opened, and how it is styled, it will always have the same accessibility requirements once in the open state.

The goal is to trap keyboard focus, trap the screen reader virtual cursor, and allow the user to close the dialog and return to their previous page position.

Trap Keyboard Focus

While open, a dialog must prevent keyboard focus from returning to the background page. In addition, keyboard focus must wrap from the last interactive element back to the first, and vice versa.

There are several different JavaScript techniques to achieve such behaviour. We provide an example jquery-keyboard-trap plugin as an example.

Trap Screen Reader

In addition to preventing keyboard users from interacting with non-dialog content, we must also prevent screen reader users from doing the same with their virtual cursor.

Any content that is not inside of the dialog element **must** be hidden from screen reader users. We can achieve this by applying aria-hidden="true" to all direct descendants of the body element while the dialog is in an open state.

You **must also** remember to *un*hide these elements when the dialog is closed, otherwise the page will be completely hidden & innacessible to screen readers!

Again, we provide an example jquery-screenreader-trap plugin to assist with this behaviour.

Prevent Page Scroll

It is desirable to prevent the background page from scrolling when interacting with the foreground dialog content. This can be achieved by simply applying overflow:hidden to the html body:

```
body.has-dialog {
    overflow: hidden;
}
```

The presence of this class would be toggled by our JavaScript on dialog hide/show event.

ESC Key

Pressing the ESC key at any time must close the dialog:

```
$(document).on('keyup', function onDocumentKeyup(e) {
    if (e.keyCode === 27) {
        $dialog.trigger('close');
    }
});

$dialog.on('close', function onDialogClose() {
    $dialog.detach();
    $opener.focus();
    $('body').removeClass('has-dialog');
});
```

Note that if the dialog was opened by clicking a button, we must set keyboard focus back to that button.

Useful Plugins

We have some experimental jQuery plugins that may assist you with creation of an accessible menu widget:

- jquery-keyboard-trap
  - Useful for implementing modal behaviour for keyboard
- jquery-screenreader-trap
  - Useful for implementing modal behaviour for screen reader
- jquery-focusable
  - Useful for finding the list of focusable elements inside of the dialog

ARIA Reference

This section gives an overview of ARIA usage within the context of this pattern.

role=dialog

Informs the assistive technology that the user is inside of a dialog.

role=document

For backwards compatibility for older screen readers. Without this role they cannot navigate the virtual cursor inside of dialog window.

role=banner

Identifies the banner/header section of the dialog. A user knows that this area typically contains the dialog heading and close button.

aria-labelledby

Informs the assistive technology of the onscreen text used to label the dialog.

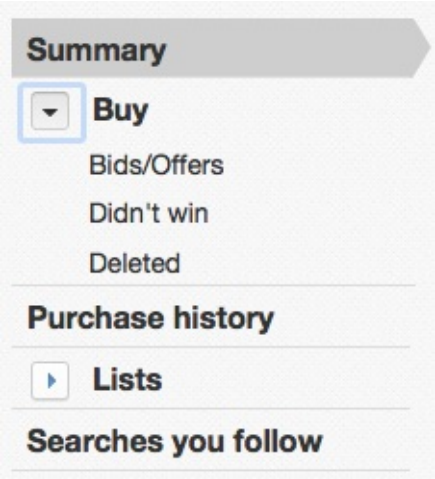
aria-label

Sets an explicit label value for the dialog, overriding any onscreen labelling text.

aria-hidden

Applied to all direct descendants of the body element (except the dialog element!) while dialog is in an open state, causing the screen reader to ignore any non-dialog content.

Expando



Introduction

A region of content that can be expanded and collapsed, forcing any adjacent content to be pushed vertically or horizontally across the page accordingly.

Expandos are often used to declutter secondary content (e.g. 'Show more'), navigation links (usually a toggle icon in the shape of an arrow) and form options.

Working Examples

You can take a look at the expando pattern in action on our examples site.

Best Practices

Must be controlled by a button, checkbox or radio button.

Interaction Design

Keyboard

For regular buttons, pressing ENTER or SPACE on button will expand or collapse content accordingly.

For radio buttons, changing the selected option will expand or collapse content accordingly.

For checkboxes, checking or unchecking the checkbox will expand or collapse content accordingly.

Screen Reader

With virtual cursor on button, checkbox or radio button, screen reader will announce current expanded/collapsed status.

When content is expanded, screen reader will announce contents of expanded region.

Developer Guide

We expect to have this section content available early 2016.

The Future

The HTML5 <details> element provides expand collapse behaviour without any need for CSS or JavaScript. Unfortunately there is insufficient browser support to consider using it right now.

Flyout

Introduction

Non-modal overlay which discloses additional content about an existing page element. The flyout 'points to', or is 'related to', this exisisting page element.

Flyout is a critical pattern. It is used in several other composite patterns:

- Tooltip
- Bubble-Help
- Flyout Notice
- Menu
- Faux Menu

Working Examples

You can take a look at the flyout pattern in action on our examples site.

Best Practices

Flyouts are typically activated via any combination of the following events:

- **Click**: Explicitly opened on click event of the page element or an associated control
- **Hover**: Implicitly opened on mouseover event of the page element or an associated control
- **Focus**: Implicitly opened on focus event of the page element or an associated control
- **System**: Automatically opened on page load event or at some arbitrary time

Focus activated flyouts are only recommended for the tooltip use case. Opening a flyout on focus in other cases can result in a poor user experience for keyboard users. For example, if forced to tab through non-primary content or navigation inside of the flyout.

Flyouts **must not** trap keyboard focus or mask page background, i.e. they must be non-modal. If you desire modal behaviour please consider a dialog pattern instead.

Flyouts **must** insert themselves into the DOM directly after the associated page element or control.

Flyouts can hold any kind of non-interactive or interactive content. However, for overlays that require input, consider using a dialog instead.

System activated flyouts **must not** be time sensitive. System activated flyouts **must** contain a button to close/dismiss the flyout.

Interaction Design

This section provides pattern interaction designs for all users.

Keyboard

Keyboard focus **must** move directly from associated control to first focusable child of flyout. If flyout has no focusable child element, focus goes to next element in page.

Pressing ESC key whilst focus is within overlay will close the flyout. Focus **must** then return to the page control that opened the flyout.

Focus activated flyouts **must** open when associated page elemtn or control gains keyboard focus.

Click activated flyouts **must** open and close upon invoking associated page control. This page control is usually a button.

Screen Reader

Reading order **must** flow directly from associated page element or control into flyout content.

Screen reader **must** announce present and/or content of high priority system-activated flyouts.

Mouse

Hover activated flyout **must** open when hovering over it's associated page element or control.

Click activated flyout **must** open and close when clicking it's associated page element or control.

System activated flyout **must** close when clicking it's close button.

Touch

Hover activated behaviour can be problematic or impossible for touch. You may wish to consider using the Bubble Help pattern instead.

Click activated flyout **must** open and close when tapping it's associated page element.

System activated flyout **must** close when tapping it's close button.

Developer Guide

Let's examine generic click-activated and hover-activated flyouts.

The key things to consider when implementing a flyout are:

1. Placement of overlay element in relation to host
2. Using CSS display absolute or fixed
3. Using aria-expanded state to toggle CSS display

Click-Activated Flyout

Click-activated flyouts typically occur on buttons. Clicking the button opens the overlay.

NOTE: Click-activated flyouts can never be used on links. This is because the click should always open the URL. Links do support hover-activated flyouts however.

Markup

For improved readability of our example code, BEM element classes will be applied to all notable elements. You could drop these classes if you deem them to be too verbose or unecessary.

```
<span class="flyout flyout--button">
  <button class="flyout__button">Toggle Flyout</button>
  <div class="flyout__overlay">
      <h3>Any kind of HTML control can go inside of a flyout...</h3>
      <p>A link: <a href="http://www.ebay.com">www.ebay.com</a></p>
      <p>A button: <button>Click Me</button></p>
      <p>An input: <input type="text" aria-label="Dummy textbox"></p>
      <p>A checkbox: <input type="checkbox" aria-label="Dummy checkbox"></p>
      <p><strong>...but avoid ARIA widgets such as tabs and carousels!</strong></p>
  </div>
</span>
```

Notice placement of the button element immedietely before the overlay element. This allows natural tab order flow from the button into the flyout.

A generic flyout such as this may contain any kind of content. However, we recommend avoiding complex widgets or form controls.

Styling the Overlay

The flyout overlay must be absolute or fixed positioned:

```
.flyout__overlay {
    display: none;
    position: absolute;
    z-index: 1;
}
```

Of course we have hidden the overlay by default. We use the aria-expanded state to control the display:

```
.flyout__button[aria-expanded=true] ~ .flyout__overlay {
    display: block;
}
```

Behaviour

Work in progress...

Hover-Activated Flyout

Work in progress...

Lens

Introduction

Dialog containing drill-down detail about an object. The object could be anything from an item for sale on eBay, a football team on a sports site, or an online news article.

The term 'lens' is a metaphor for holding a magnifying glass over an object for a more detailed view. We view the item through a lens, without leaving the current page.

Best Practices

Please follow dialog best practices.

Interaction Design

Please follow dialog interaction design.

Developer Guide

Our implementation follows the Progressive Enhancement strategy; we build in a layered fashion that allows **everyone** to access the basic content and functionality of a web page.

The three layers are:

1. Content (HTML)
2. Presentation (CSS)
3. Behaviour (JS)

Content (HTML)

The goal of our content layer is to enable navigation to an item detail view. In our example, this item will be an eBay auction.

Identify Element

We will use a simple "View Item" link:

```
<a href="http://www.ebay.com/itm/221451251366">view item</a>
```

Widget Hooks

We add our classname:

```
<a class="lens" href="http://www.ebay.com/itm/221451251366">view item</a>
```

That's it! A good 'ol hyperlink is all we need, ensuring that our core functionality (viewing the item detail) is accessible to everyone. Who said progressive enhancement was difficult?

Presentation (CSS)

Please follow the dialog base-pattern presentation implementation.

Behaviour (JS)

The goal of our behaviour layer is to convert the basic hyperlink into a button control that will open our lightbox.

Modal Construction

We construct the modal entirely with JavaScript. Please follow the dialog base-pattern content implementation for further details on the structure of the markup.

Hijack Link

With JavaScript available, we no longer want the default hyperlink behaviour of navigating to a new page. So we must prevent the click event from performing it's default behaviour:

```
$link.on('click', function onLinkClick(event) {
    event.preventDefault;
});
```

Entity Parsing

Retrieving the item detail content from a REST API via AJAX is beyond the scope of this document, but one thing we should point out is that we already have access to the unique item id right there in the href attribute.

If anything else needs to be passed to the client-side plugin or REST API, we could store it as JSON in a data attribute. For example, if you wanted to pass the unique item id in this manner:

```
<a class="lens" data-lens='{"id":"221451251366"}' href="http://www.ebay.com/itm/221451
251366">view item</a>
```

Transform Link

With our behaviour now complete, our link no longer behaves like a link. It is more like a button. ARIA allows us to change the semantic nature of our link element like so:

```
$link.attr('role', 'button');
```

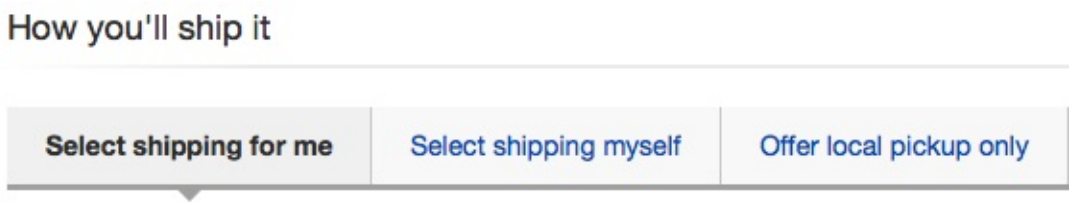
Screen readers will now announce the anchor tag as a button, giving users a better indication that they will stay on the same page when the button is pressed.

Summary

The lens pattern does not deviate very far from the dialog pattern.

The key thing to note is that we use the principles of Progressive Enhancement and Hijax to 'hijack' the behaviour of the link with JavaScript to open a lightbox dialog instead of navigating to a new page.

Tabs



Introduction

Allows layered stacking of content panels; only one panel of content is visible at any time.

Navigation or Disclosure?

The tabs pattern removes page clutter by dynamically hiding & showing content. These tabs are **not** intended for reloading the page or navigating to a new URL.

If you wish to create a set of links styled *visually* as tabs, with no JavaScript behaviour, please use a simple list of links. Consult the faux tabs pattern for details.

Working Examples

You can take a look at the tabs pattern in action on our examples site.

You can get an idea of the required markup structure by viewing our bones site.

For a real life example, you can also see the tabs pattern in action on our eBay Skin site.

Best Practices

Tabs widget **must** have a heading. All tabs **should** be thematically related to this heading. For example, a set of 'Shipping Services' tabs might contain a tab each for USPS, FedEx and UPS.

Each tab panel **must** also contain an offscreen heading. This hidden heading text is the same as it's corresponding tab text. The level of this panel heading **must** be one level lower than the heading of the widget.

Interaction Design

The tabs pattern comprises of a tablist, tabs and tabpanels. Just one tab can be 'selected' at any given time.

Changing the selected tab will display the corresponding tabpanel and hide all others.

Keyboard

The tablist exhibits widget behaviour; that is, TAB key must enter the widget and next TAB key must exit widget.

With keyboard focus on the active tab, ARROW keys move focus and change the selected tab.

When a user leaves and then returns to the widget, keyboard focus **must** go to the last active tab.

Screen Reader

A screen reader can navigate the tablist using arrow keys (as described above) *or* virtual cursor next/prev item commands.

Virtual cursor navigation may move from tab to tab *without* changing the active tab selection. The invoke command (e.g. VO+SPACE on Voiceover) selects the tab under the virtual cursor.

Screen reader **must** announce selected tab's selected state.

Developer Guide

Our sample implementation follows the Progressive Enhancement strategy; we build in a layered fashion that allows **everyone** to access the basic content and functionality of a web page.

The three layers are:

1. Content (HTML)

2. Presentation (CSS)
3. Behaviour (JS)

The tabs and their related content elements can be fully visible and accessible without CSS and JavaScript as simple hyperlinks and page anchors respectively.

Content (HTML)

The goal of our content is to add all of our tabs and panel content to the page.

For the purposes of this example, all content will come from the server on first page load. If you have a lot of content, you may wish to consider lazy loading the content of each panel with AJAX.

Tabs and Panels

The tabs begin life as simple same-page navigation links, linking to the content anchors (panels) below it on the same page:

```
<div class="tabs tabs-horizontal">
    <h2>My eBay</h2>
    <ul>
        <li><a href="#buying">Buying</a></li>
        <li><a href="#bought">Bought</a></li>
        <li><a href="#selling">Selling</a></li>
        <li><a href="#sold">Sold</a></li>
    </ul>
    <div>
        <div id="buying" tabindex="-1">
            <h3>...</h3>
            <p>...</p>
        </div>
        <div id="bought" tabindex="-1">
            <h3>...</h3>
            <p>...</p>
        </div>
        <div id="selling" tabindex="-1">
            <h3>...</h3>
            <p>...</p>
        </div>
        <div id="sold" tabindex="-1">
            <h3>...</h3>
            <p>...</p>
        </div>
    </div>
</div>
```

We call this markup structure our bones; our CSS and JavaScript will be expecting this **exact** DOM structure convention.

This structure has been chosen carefully. It allows us to display tabs horizontally *and* vertically simply by changing the second class (to tabs-horizontal or tabs-vertical).

Checkpoint

That's it! Our content is available to anyone in a non-CSS and non-JS state.

Presentation (CSS)

The goal of our CSS is to prevent a flash of unstyled content (FOUC) and visually style the links to look like folder style tabs.

How you choose to style the links is outside the scope of this document, because every website likes to make their tabs look slightly different!

Flash of Unstyled Content (FOUC)

FOUC may occur before JavaScript initialises the widget. To alleviate this we can set a fixed height on the tab panel container:

```
/* before js init */
.tabs > div {
    height: 150px;
    overflow-y: auto;
}
/* after js init */
.tabs-js > div {
    height: auto;
}
```

We have chosen an arbitrary value of 150px for our example. After JS initialises the widget, it's height will grow or shrink to match the content of the currently selected panel. Of course if fixed height is what you desire, then you can leave the fixed value in place.

Checkpoint

Our tabs now appear visually like tabs, and the panel content is still fully operable without JavaScript (albeit with ugly vertical scrollbars).

Behaviour (JS)

The goal of our JavaScript is to implement our interaction design.

Plugin Boilerplate

We start by caching references to our most important elements:

```
(function ( $ ) {
    $.fn.tabs = function tabs () {
        return this.each(function onEach() {
            var $tabsWidget = $(this),
                $tablist = $tabsWidget.find('> ul'),
                $tabs = $tablist.find('li'),
                $links = $tablist.find('a'),
                $panelcontainer = $tabsWidget.find('> div'),
                $panels = $panelcontainer.find('> div');

            // implementation goes here
        });
    };
}( jQuery ));
```

Our selectors are based on our bones markup convention.

Widget Init

Let's mark our widget as initialised:

```
$tabsWidget.addClass('tabs-js');
```

Now our CSS rules for our progresively enhanced widget will kick in.

ARIA Roles

How does a screen reader know this is a tabs widget? We must add ARIA roles.

Roles only need to be added once:

```
$tablist.attr('role', 'tablist');
$tabs.attr('role', 'tab')
$panels.attr('role', 'tabpanel')
$links.attr('role', 'presentation').removeAttr('href');
```

Notice the last statement, which essentially now turns our old same-page links into meaningless span elements, so that they don't conflict with our tabs.

ARIA States

How does a screen reader know which tab is currently selected and which panel is visible? We must add ARIA states.

States must be monitored and then bound to any changes in the view:

```
$tabs
    .attr('aria-selected', 'false')
    .first()
        .attr('tabindex', '0')
        .attr('aria-selected', 'true');
$panels
    .attr('aria-hidden', 'true')
    .first()
        .attr('aria-hidden', 'false');
```

By default we make the first tab the selected tab.

ARIA Properties

How does a screen reader know which panel belongs to which tab? We must add ARIA properties.

Properties are usually just added once and then left alone:

```
$tabs.each(function onEachTab(idx, tabEl) {
    var $tab = $(tabEl),
        tabId = $tabsWidget.attr('id') + '_tab_' + idx,
        panelId = $tabsWidget.attr('id') + '_panel_' + idx;

    $tab
        .attr('id', tabId)
        .attr('aria-controls', panelId);

    $panels.eq(idx)
        .attr('id', panelId)
        .attr('aria-labelledby', tabId);
});
```

All panels are now labelled and controlled by their respective tab.

Roving Tab Index

If there are many tabs it would require many TAB key presses to navigate past the widget, therefore tabs should be navigated with ARROW keys instead.

Only one tab can be focussable at any given time. This is always the selected/active tab, so that when a user tabs away from the widget and then back again, focus will return to the active tab.

This behaviour is known as a roving tabindex. We provide a sample jquery.rovingtabindex.js plugin for you to reference.

Prevent Page Scroll

When keyboard focus is on the widget, we must prevent arrow keys and spacebar from scrolling page. JQuery make this trivial for us:

```
$(document).on('keydown', '[role=tab]', function(e) {
    var keyCode = e.keyCode;
    if (keyCode === 32 || keyCode === 38 || keyCode === 40) {
        e.preventDefault();
    }
});
```

Notice the 2nd parameter which scopes the keydown event to a particular selector/element. Very handy!

Pointer Interaction

Finally, don't forget to delegate mouse clicks and touch events:

```
$tablist.on('click', function(e) {
  // update view and ARIA states here
});
```

We will leave these implementation details for pointer devices up to you - it shouldn't be too hard!

Final Checkpoint

We have enhanced our markup with ARIA roles, states and properties for screen reader users, and implemented keyboard & mouse behaviour.

Your next steps would be to make it look pretty and optimise/rework your JS as necessary - adding in any extra behaviour you may need - but remembering not to break accessibility of course!

Useful Plugins

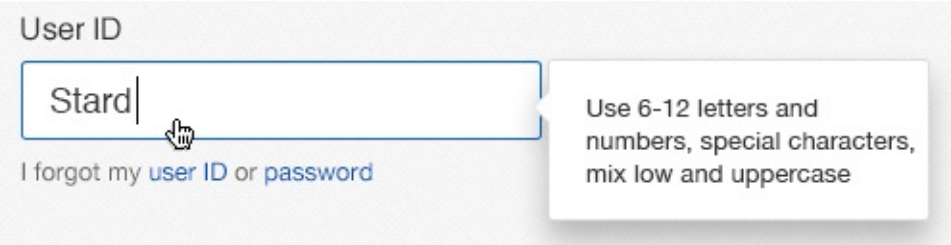
We have some experimental jQuery plugins that may assist you with creation of an accessible tabs widget:

- jquery-roving-tabindex
  - Useful for implementing the arrow key behaviour to change tabs

References

- W3C Design Patterns: Tabpanel

Tooltip



Introduction

Tooltip is a hover and focus activated flyout. The flyout contains advisory or supplementary content related to an interactive page element.

You can think of a tooltip as inline content you would place in parenthesis.

In desktop software tooltips are commonly used in relation to toolbar button. Web tooltips are more commonly experienced in relation to links, buttons and form controls.

Working Examples

You can take a look at the tooltip pattern in action on our examples site.

Best Practices

Tooltips on interactive elements work well for mouse and keyboard. They can trigger the tooltip on hover or on focus respectively. Also, the tooltip does not interfere or prevent the behaviour of the interactive element. For example, links can still be followed, and buttons can still be activated. On touch screens however, this dual behaviour can be problematic or impossible.

A good tooltip follows the principal of progressive disclosure. That is to say, a good tooltip reveals the right information, at the right time.

A good tooltip should be no more than one sentence in length. If you wish to display more than one sentence, please consider using the Bubble Help pattern.

Never rely solely on the title attribute and **never** apply the tooltip to non-interactive elements.

Do **not** use aria-haspopup for tooltips. Ordinary tooltips are not considered popups in this context.

Tooltip content is important, and should exist in the base markup rendered on the server. CSS and JavaScript enhancements will present the content inside of an overlay.

Interaction Design

This section provides the pattern interaction design for keyboard, screen reader, mouse & touch.

Keyboard

Navigating to the interactive element will show the tooltip after a short delay.

The next TAB key press will move focus to the first interactive element inside of the tooltip (if applicable).

The tooltip hides when neither the interactive element or it's content have focus.

Screenreader

Navigating to the interactive element will announce the content of that element, as normal. The assistive technology will announce the tooltip content after a short delay. This delay is configurable in most screen readers.

Mouse

Hovering over or setting focus on the interactive element will show the tooltip after a short delay.

The tooltip dissapears when the mouse leaves the interactive element or tooltip overlay.

Touch

Many touch devices do not support hover interactions!

You may wish to instead consider using the Bubble Help pattern.

Developer Guide

Our implementation follows the Progressive Enhancement strategy. We build in a layered fashion that allows **everyone** to access the basic content and functionality of a web page.

The three layers are:

1. Content (HTML)
2. Presentation (CSS)
3. Behaviour (JS)

The tooltip content is fully visible and accessible without CSS and JavaScript.

Content (HTML)

The goal of our content layer is to add helpful tooltip content to an interactive, focusable page element.

Identify Element

First we identify the interactive page element that requires a tooltip.

We will use the "Add to Cart" button below:

```
<button>Add to Cart</button>
```

Add Content

We now need to add our tooltip content to the base markup.

We place our content directly after the button inside of a span element:

```
<button>Add to Cart</button>
<span>Your cart contains <a href="http://cart.payments.ebay.com/sc/view">6 items</a></
span>
```

Notice that our tooltip content contains a nested hyperlink. This is a common product requirement. Native tooltips (i.e. the title attribute) do not support this requirement. We must ensure that any content *inside* of our tooltip overlay is accessible too.

ARIA

Now let's add our ARIA attributes:

```
<button aria-describedby="tooltip1">Add to Cart</button>
<span id="tooltip1" role="tooltip">Your cart contains <a href="http://cart.payments.eb
ay.com/sc/view">6 items</a></span>
```

The span element now has `role=tooltip` and a unique `id`. The page element references this id in it's own `aria-describedby` attribute.

Widget Hooks

We wrap our page element and tooltip elements together in a new parent element that forms the root element for our widget:

```
<span class="tooltip">
  <button aria-describedby="tooltip1">Add to Cart</button>
  <span id="tooltip1" role="tooltip">(Your cart contains <a href="http://cart.payments
.ebay.com/sc/view">6 items</a>)</span>
</span>
```

We give our widget a class of 'tooltip' which acts as the hook for our CSS and JavaScript.

Checkpoint

Our markup is now complete. The link, it's tooltip and the link inside of the tooltip are all functional and accessible.

Presentation (CSS)

We *could* use pure CSS and display the tooltip flyout using the `:hover` and `:focus` pseudo selectors. However, we would quickly run into accessibility issues. When keyboard users tab away from the button, the tooltip overlay is immediately hidden. This behaviour prevents focus on any links *inside* of the tooltip overlay. We **need** to use JavaScript to prevent this behaviour.

Positioning

To be able to position the tooltip with JavaScript, we set either absolute or fixed positioning:

```
.tooltip--js [role=tooltip] {
    display: none;
    position: absolute;
    z-index: 1;
}
```

Notice we are using the `tooltip--js` selector. JavaScript will add this modifer class when the widget has fully initialised. Our tooltip isn't hidden *until* that time. This ensures that the tooltip content is readable and accessible without JavaScript.

When to use fixed positioning rather than absolute? Fixed position prevents possible cropping and constrained width issues. For example, in cases where the ancestor element has a fixed width or height. The trade off is that we must use JavaScript to update the tooltip position on window scroll and window resize.

Basic Styling

Then we add some basic tooltip styling, such as background color, border and padding:

```
.tooltip--js [role=tooltip] {
    background-color: LightYellow;
    border: 1px solid #ccc;
    display: none;
    position: absolute;
    padding: 0.4em;
    text-align: center;
    white-space: nowrap;
    z-index: 1;
}
```

This is only an example set of styles. You should consult your own design system library for specifics.

Checkpoint

Our presentation is now complete. At this point, the content remains fully visible and accessible.

Behaviour (JS)

The goal of our JavaScript layer is control the position & visibility of the tooltip content.

Visibility

The tooltip must display when hovering with mouse or focusing with keyboard.

Our tooltip element is a descendant of the widget element. This DOM hierarchy makes it trivial to use mouse and focus events to hide and show the tooltip.

```
$tooltipWidget.on('focusin', function onFocusIn(e) {
    show();
});

$tooltipWidget.on('focusExit', function onFocusExit(e) {
    hide();
});
```

NOTE: `focusExit` is a custom event. Read below for more details.

Keyboard Focus

The jQuery 'focusout' event fires every time an element inside of our widget loses focus. This event occurs even if focus goes directly to another element inside of the widget. How can we be notifed only when focus has left the entire tooltip widget?

To solve this issue we created the jquery-focus-exit plugin. This plugin triggers a custom `focusExit` event when focus exits the widget.

Final Checkpoint

Our accessibility behaviour is now complete.

This pattern lays the foundation for creating an accessible tooltip.

The following additional behaviour is outside the scope of this guide:

- Re-positioning the tooltip bubble when near edge of screen
  - A good candidate for a jQuery plugin (if it doesn't already exist)
- Adding `mouseenter` and `mouseleave` event handlers
  - The tooltip must display for mouse users too, remember!

Compatibility

This section provides compatibility support matrices for most common browser + screenreader combinations:

	IE JAWS14	Safari VoiceOver	Chrome TalkBack	IE NVDA	IE WinEyes	IE Narrator
Announces tooltip content	Yes	Yes	tbd	Yes	tbd	tbd
Ignores tooltip parenthesis	**No**	Yes	tbd	Yes	tbd	tbd
Announces role=tooltip	**No**	Yes	tbd	Yes	tbd	tbd
Tooltip content navigable	Yes	Yes	tbd	Yes	tbd	tbd

FAQ

This is different to the way Bootstrap Tooltip works. Why?

The fundamental difference is that the Bootstrap Tooltip get it's content from the title attribute, whereas the pattern presented here gets it's content from an element's innerHTML. This allows us to more easily place HTML such as hyperlinks inside of our tooltips and, unlike Bootstrap, the tooltip content will be fully accessible *without* JavaScript.

Why do we need JavaScript at all? Can't we do a tooltip with just CSS?

A pure CSS solution only gets us so far, it could not cover the following situations:

- Delay before showing tooltip
- Keyboard focus management
- Re-positioning logic when scrolling or near edge of screen

Structure

Headings, landmarks, forms and tables help us structure our page content into logical, navigable regions.

Pattern Completion/Progress

	Working Examples	Best Practices	Interaction Design	Developer Guide
Form	✕	✕	✕	✕
Heading	✕	✓	✓	✕
Image	✕	✓	✕	✕
Region	✕	✓	✓	✕
Table	✕	✓	✕	✕

Form

We expect to have further information on this pattern available early 2016.

Heading

1. Web Content Accessibility Guidelines
    1. Contents
    2. Earlier guidelines[edit]
    3. WCAG 1.0[edit]
        1. WCAG Samurai[edit]
    4. WCAG 2.0[edit]
    5. Legal obligations[edit]
    6. References[edit]
    7. External links[edit]
    8. Navigation menu
        1. Personal tools
        2. Namespaces
        3. Variants
        4. Views
        5. More
        6. Search
        7. Navigation
        8. Interaction
        9. Tools
        10. Print/export
        11. Languages

Introduction

Upon visiting a web page for the first time, a sighted user will typically scan the headings of the page to create a visual model of it's content. On subsequent visits to the page, the user recollects this visual model.

A non-sighted user will typically want to do exactly the same, using their screen reader to scan for headings to create a mental model.

The hierarchy of these headings dictates the HTML4 document outline. This HTML4 document outline is less important now with the advent of HTML5 and it's newer, improved document outline. Much more important is the impact of headings to screen reader accessibility.

Working Examples

We expect to have working examples available in early 2016.

Best Practices

Every page requires a single H1 heading. This heading describes the content and purpose of the page.

Every page region requires a single H2 heading. This heading describes the content and purpose of the region. It is good practice to wrap every page region in a div or section tag.

Every page region can contain multiple H3 - H6 headings, but they *must* be placed in the markup in strict hierarchical order.

Interaction Design

Headings are not interactive elements and therefore have no keyboard interaction.

A heading may give the illusion of being an interactive element if it contains a link, or is wrapped in a link, but the heading element itself never receives focus.

Screen reader will identify heading by it's level, e.g. 1 to 6.

Image

There are two types of image:

1. content images
2. decoration images

Working Examples

We expect to have working examples available in early 2016.

Best Practices

Every image can be classed as important *content* or non-important *decoration*.

Every image **must** have an alt attribute. There are no exceptions.

Decoration images **must** have an empty alt attribute value. This informs screen reader to skip over this element, otherwise the long, ugly URL/filename of image will be read.

Content images **must** have a meaningful, descriptive alt attribute value that should be defined by your content team.

If a content image is adjacent to descriptive text, for example in the case of an item title right before or after the item image, then the alt text of the image can be left blank.

If possible, always try and use CSS background images or generated content (e.g. font icons) for decoration images.

Avoid using CSS background images for content images because they are disabled by default in Windows high contrast mode (this setting can be overridden in your application CSS however).

Interaction Design

Image tags are not interactive by default and therefore should not be keyboard focussable.

A screen reader will skip over a decoration image (with empty alt text) and will read the alt text of content images.

Region

Upon visiting a web site for the first time, a sighted user will typically scan the page to create a visual model of it's main regions. On subsequent visits to the site, the user will quickly recollect this visual model.

A screen reader user will typically want to do exactly the same to create a *mental* model, but this task can be easy or difficult depending on the HTML markup of the page.

Like headings, regions can be used as navigational aids because screen reader vendors like JAWS and NVDA provide keyboard shortcuts to move between them. For example, in JAWS 15 you can press the R key to go to the next region or Shift + R to go to the previous one.

Regions are synonymous with HTML5 sections.

Working Examples

We expect to have working examples available in early 2016.

Best Practices

There are two types of region:

Landmark regions identify the main regions of a page that are common to most websites. Site banner/logo, the search box, site navigation, main content area, sidebar, etc. HTML5 and ARIA provide us with a limited list of landmark tags and roles respectively.

Custom regions identify the main areas of a page that are common to *your* website. Search filters, user profile, notifications, help, etc. HTML5 and ARIA provide us with the section tag and region role respectively, as the building blocks to create any number of custom regions.

Example

The following example shows how we can markup the main regions of a page using a combination of HTML5 and ARIA:

```
<body>
    <header role=“banner”>
       <img src=“sitelogo.jpg” />
       <nav role="navigation">
        …
       </nav>
       <div role="search">
        …
       </div>
    </header>
    <div role=“main”>
        <section role=“region” aria-labelledby="userprofile_heading">
           <h2 id="userprofile_heading">User Profile</h2>
           …
        </section>
        …
    </div>
    <aside role=“complementary”>
        <h2>Advertisement</h2>
        …
    </aside>
    <footer role=“contentinfo”>
        …
    </footer>
</body>
```

In this example, we have added a custom user profile region inside of our main content. Custom regions **must** be labelled. The screen reader will identify the region in our example as 'User Profile region' (or words to those affect).

Note that the HTML5 <main> tag is not yet supported by any version of Internet Explorer so we avoid using it for now.

Interaction Design

Regions are not interactive elements and therefore have no keyboard interaction.

Screen readers will identify regions by their label, and all regions (landmarks & custom) are available with screen reader shortcut keys. Note that VoiceOver identifies both types of regions as 'landmarks'.

Table

A table is used to represent tabular data in rows and columns.

Working Examples

You can take a look at the table pattern in action on our examples site.

Best Practices

Every table requires a caption element. The summary attribute is deprecated in HTML5.

Every table cell should have a logical column header or row header.

Do **not** use tables for row/column layout of page or modules. See the Layout Tables Anti-Pattern for more information.

Interaction Design

Keyboard, Mouse & Touch

The only special consideration for table is sortable columns. Typically this is achieved by adding a toggle button as the contents of the table header cell. The usual rules for button apply.

Screen Reader

Table **must** be visible in screen reader list of tables.

Screen reader **must** be navigable with screen reader table command. For example, in VoiceOver this command is `VO+CMD+T`.

Screen reader **must** announce caption of table.

Screen reader **must** announce value of every cell.

Screen reader **must** announce value of column header when instructed to do so. For example, in VoiceOver this command is `VO+C`.

Sortable column **must** be announced by screen reader.

Current sort state of sortable columns **must** be announce by screen reader.

When moving from column to column, screen reader announces new column scope. Typically this scope is the value of the column header.

When moving from row to row, screen reader announces new row scope. Typically this scope is the value of the row header.

We expect to have further information on this pattern available by early 2016.

Techniques

Across all of the code patterns presented in this book, there are several cross-cutting sub-patterns & techniques. These techniques are usually implemented with JavaScript.

List of Techniques

Techniques: Alt Text

An alternative means to convey information that is otherwise only provided visually in images or icons, used in the following situations:

- Embedded image text
- Image links
- Adjacent alt text

Embedded image text

If the image contains text (usually ads and promo banners), we must convey this text in the alt attribute.

```
<img src="fashion_promo.jpg" alt="Explore eBay Fashion" />
```

Image links

Alt text is used as the link text.

```
<a href="http://www.ebay.com/itm/12345">
    <img src="lumia_950.jpg" alt="Microsoft Lumia 950" />
</a>
```

Adjacent alt text

If the alt text would repeat nearby or adjacent text, we can leave it's value empty so that the screen reader will ignore it.

```
<a href="http://www.ebay.com/itm/12345">
    <img src="lumia_950.jpg" alt="" role="presentation" />
    <p>Microsoft Lumia 950</p>
</a>
```

`role=presentation` will prevent our internal WAE tool from flagging this as a missing alt attribute.

Techniques: Focus Trap

A focus trap can be implemented to prevent keyboard focus from leaving the confines of a UI control. Most commonly used in modal dialogs.

We expect to have this section content available early 2016, in the meantime you can reference the jquery-keyboard-trap plugin on GitHub & NPM.

Techniques: Icon Fonts

Icons can be placed on static elements or interactive controls. Icons can be deemed critical or non-critical.

Non-Critical Icons

An icon is deemed non-critical if it is purely presentational, or if the information can be found in nearby or adjacent text.

```
<p>
  <span class=”icon-warning” aria-hidden="true">
    <!-- icon font generated here with CSS -->
  </span>
  Warning: this item is about to expire
</p>
```

The adjacent text clearly contains the word "Warning" and so the element that generates the icon can safely be set to aria-hidden=true.

Critical Icons

Critical icons are icons with no supplementary text.

```
<p>
  <span class=”icon-warning” role="img" aria-label="Warning: this item is about to exp
ire">
    <!-- icon font generated here with CSS -->
  </span>
</p>
```

Critical icons are most common in buttons:

```
<button class=”hamburger” aria-label=”Menu”>
  <!-- icon font generated here with CSS -->
</button>
```

Techniques: Live Regions

If an area of a page is dynamically updated without a full page reload, we *might*, but not always, need to inform the user.

```
<div aria-live="polite">
    <p>Newcastle 0 - Barcelona 0</p>
</div>
```

Whenever the contents or visibility of this live region changes, the screen reader will announce the changed content.

```
<div aria-live="polite">
    <p>Goal!</p>
</div>
```

Screen reader announces “Goal!”

```
<div aria-live="polite">
    <p>Newcastle 1 - Barcelona 0</p>
</div>
```

Screen reader announces “Newcastle 1 - Barcelona 0"

Techniques: Offscreen Text

Offscreen text allows additional context or information about a control or element for assistive technology. Offscreen text helps in the following situations:

- Link opens in new window or tab
- Ambiguous label
- Repeated Ambiguous label
- Offscreen headings

TIP: offscreen text is also commonly known as screen reader text or clipped text.

All of the examples in this page make use of the `.clipped` class to create offscreen text:

```
/* clip element visibility, making it accessible to screen reader only */
.clipped {
    position: absolute !important;
    clip: rect(1px 1px 1px 1px); /* IE6, IE7 */
    clip: rect(1px, 1px, 1px, 1px);
    padding: 0 !important;
    border: 0 !important;
    height: 1px !important;
    width: 1px !important;
    overflow: hidden;
}
```

Link opens in new window or tab

Launching a new browser window can be a major frustration for users of assistive technology.

```
<a href="http://www.ebay.com/help" target="_blank">Help<span class="clipped> - opens i
n new window or tab</span></a>
```

Offscreen text (inside the clipped span tag) helps to set expectations in advance.

Ambiguous label

A label whose context can only be determined visually, programmatically or with manual effort.

```
<button>Buy it Now<span class=”clipped”> - iPhone 4</span></button>
```

Offscreen text simplifies the context of the control for assistive technology.

Repeated Ambiguous label

An ambiguous label that is used many times on the same page. When a screen reader displays the list of controls on the page, it will be impossible to determine one from the other.

```
<button>Buy it Now<span class=”clipped”> - iPhone 4</span></button>
<button>Buy it Now<span class=”clipped”> - Nokia Lumia 635</span></button>
<button>Buy it Now<span class=”clipped”> - iPhone 6</span></button>
```

Offscreen text improves the uniqueness of each label.

Offscreen headings

If the visual design does not incorporate onscreen headings, we must provide them offscreen.

```
<h2 class=”clipped”>Search refinements</h2>
```

When do we provide offscreen headings? A good rule of thumb is that every *visually* significant module or section of the page should have a heading.

Techniques: Skip to Main Content

A means for keyboard users to bypass header navigation and search.

```
<!-- the skip to anchor is the first interactive element in the DOM -->
<a href=”#mainContent” class="clipped">Skip to Main</a>

<!-- the anchor target wraps your main content -->
<div id=”mainContent” role=”main”>
  ..
</div>
```

Please refer to the Skip-To pattern for guidance regarding CSS and JavaScript.

The Anti-Patterns

An anti-pattern (or antipattern) is a common response to a recurring problem that is usually ineffective and risks being highly counterproductive.

List of Anti-Patterns

- Ambiguous Label
- JavaScript HREF
- Layout Tables
- Non-interactive Hover
- Open New Window
- Tabindex-itis
- Title Tooltip

Anti-Pattern: Ambiguous Label

A typical e-commerce sight is rife with buttons and links such as 'Buy it Now', 'Add to Cart', 'Select', and 'Delete' etc. We call these labels *ambigious labels* because the label does not explicitly call out the item, or noun, that it refers to.

Use clipped text to ensure labels are unambiguous to AT:

```
<a href="http://www.ebay.com/cart/add">Add to Cart<span class="clipped"> - iPhone 6, 32GB</span></a>

<button>Place Bid<span class="clipped"> - iPhone 6, 32GB</span></button>
```

Clipped text remains hidden to sighted users but visible to AT. Please refer to the Utilities page for more information on the clipped class.

NOTE: An ambiguous label is not an anti-pattern for sighted users. Sighted users gather context from visual surroundings.

JavaScript HREF

Never use "javascript:" or "#" as the value of an href attribute, because this value will be read by the screen reader.

The purpose of the href attribute is to specify a **URL only.** If you wish to run JavaScript behaviour on element click, use a button or progressively enhanced link instead.

Bad:

```
<a href="javascript:;" id="share">Share on Facebook</a>

<script>
    $('#share').on('click', function(e) {
        // do fb share stuff
    });
</script>
```

Screen reader will announce "JavaScript colon semi-colon Share on Facebook". The call to action content is now unclear!

Good:

```
<!-- a button created with JavaScript -->
<button id="share">Share on Facebook</button>

<script>
    $('#share').on('click', function(e) {
        // do fb share stuff
    });
</script>
```

The screen reader will announce "Share on Facebook. Button". Much better!

NOTE: Buttons outside of forms should always be created and inserted with JavaScript; this prevents a 'no-op' situation when clicking the button before the script is available or ready.

Even better

Use progressive enhancement:

```
<a href="http://www.facebook.com/share?id=12345" id="share">Share on Facebook</a>

<script>
    $('#share').attr('role', 'button');
    $('#share[role=button]').on('click', function(e) {
        this.preventDefault();
        // do fb share stuff
    })
</script>
```

The screen reader will announce "Share on Facebook. Button" or fallback to "Share on Facebook. Link" if JavaScript is not ready or available.

Anti-Pattern: Layout Tables

Do not use the HTML table tags (<table>, <tr>, <td>, etc.) for page or module layout.

For example, do not use table rows and columns to arrange the header, sidebar and main content areas of a page or module:

```
<!-- This is bad! Please do not copy! -->
<body>
    <table>
        <tr>
            <td colspan="2">
                <!-- page header/banner -->
                <img src="sitelogo.jpg" />
            </td>
        </tr>
        <tr>
            <td>
                <h2>Sidebar</h2>
                <!-- sidebar content -->
            </td>
            <td>
                <h1>Page Header</h1>
                <!-- main content -->
            </td>
        </tr>
        <tr>
            <td colspan="2">
                <!-- page footer -->
            </td>
        </tr>
    </table>
</body>
<!-- Urgh, I felt dirty writing that. The things I do for you all. -->
```

The table tags should only be used to represent tabular relationships between data. The table tags have semantics that will cause problems for screen reader users if used for layout.

For example, in the code example above, the screen reader would pre-announce all elements inside the sidebar as "row 2 column 1", and all elements in the main content with "row 2 column 2" etc.

CSS can, and should, be used to achieve row/column layouts in pages and modules and this layout is transparent to the screen reader user.

Legacy Code

If you have legacy code that you wish to make accessible, and are unable to rewrite your entire markup and CSS, a quick fix is to apply **role=presentation** to the table tag:

```
<table role="presentation">
    <!-- rows and columns go here as normal -->
</table>
```

With this role, visual layout will be maintained but all table, row and column semantics will be removed from the accessibility tree.

The CSS Table Model

Annoyed that you can't use tables for layout anymore, because you find that super useful and CSS floats are a pain?

Take a look at The CSS table model which lets you layout DIVS and SPANS using similar table-like syntax, without the semantics.

Anti-Pattern: Non-interactive Hover

Do not add mouse hover behavior to non-interactive elements. The behaviour will not be accessible to keyboard users.

Example to come...

Anti-Pattern: Open in New Window

Forcing links to open in a new window is an anti-pattern because we are taking control away from the user and forcing web-browsing behaviour upon them. If a user wishes to open a link in a new window, they can do so by using their mouse or keyboard shortcut, without our intervention.

If, for whatever reason, you find that you must open a link in a new window, you should pre-warn all users of this behaviour, ideally by use of an icon inside the link (with alt text for screen readers).

```
<a href="http://www.ebay.com/shop" target="_blank">Shop Now<img src="new-window-icon.j
pg" alt=" - opens in new window or tab" /></a>
```

Or alternatively, using a background image:

```
<a href="http://www.ebay.com/shop" target="_blank">Shop Now<span class="icon-new-windo
w" aria-label=" - opens in new window or tab"></span></a>
```

Anti-Pattern: Tabindex-itis

Do not apply the tabindex attribute to non-interactive elements (headings and paragraphs, for example). Doing so will add an additional, unnecessary tabstop for keyboard users.

A common mis-perception is that "screenreaders need to be able to access headings, so we must add a tabindex". But screenreaders *can* access headings perfectly well *without* tabindexes.

An unneccessary tabstop can confuse users, because any attempt to interact with the element (using SPACE or ENTER key) will provide no feedback to user.

An unneccessary tabstop can increase physical discomfort for some users - for example if using a binary switch rather than a traditional keyboard.

Anti-Pattern: Title Tooltip

HTML title attributes are often perceived as an accessibility bonus, but the opposite is true. Content being put in the title attribute is being hidden from many of your users (see below). If information is being hidden from your users, and you are okay with that, then the information is probably not necessary in the first place.

HTML 5.1 includes general advice on use of the title attribute:

Relying on the title attribute is currently discouraged as many user agents do not expose the attribute in an accessible manner as required by this specification (e.g. requiring a pointing device such as a mouse to cause a tooltip to apear, which excludes keyboard-only users and touch-only users, such as anyone with a modern phone or tablet).

The W3C states that:

Implementing this technique (supplementing link text) with the title attribute is only sufficient if the title attribute is accessibility supported (it is not, as stated above). If the value of the title is essential to understanding the purpose of the link for all users, then the content of this attribute needs to be available to all keyboard users (not only those with text-to-speech software) for this technique to be accessibility supported.

Anti-Pattern: Disabling Pinch-to-Zoom

In a `<meta name="viewport"... >` element

- Don't set `user-scalable="no"`
- Don't use `maximum-scale`

Why not?

Users with low vision depend on the ability to zoom content.

- We **must not** disable the use of the native browser zoom.
- We **must not** dictate how much a user can zoom the page.

The user has the right to make their experience useful for them even though others might not like the way it looks. The ability to pinch/zoom is an expected gesture for these devices, we should not remove this functionality.

Appendix

Thank you for taking the time to read this book. I hope it has proved useful!

In this final section you can find a list of frequently asked questions, references, utilities and ARIA essentials.

ARIA Essentials

This page lists the essential ARIA roles, states & properties that every developer should be familiar with, including a brief overview on their usage.

aria-label & aria-labelledby

Label an element using one of these properties.

Use aria-labelledby if there is visible text onscreen that can be used to label the element (a nearby heading, for example).

Use aria-label when there is no suitable labeling text onscreen (this issue is common with icon buttons, for example).

These attributes are not recommended if the element already has a natural label such as inner-text or an associated label tag, as the ARIA attribute may take precedence, therefore overriding the natural label.

aria-describedby

Describe an interactive element using existing onscreen text.

A description is different from a label. Labels are typically short and to the point, whereas descriptions are longer and more verbose.

For example, using this attribute we could describe the inline error message of a form field, or the title of the item belonging to an 'Add to Cart' button.

role=button

Informs the user that this element can be clicked or pressed to trigger an action.

Commonly used to convert a plain DIV or SPAN into an interactive button. Also useful in converting hijaxed hyperlinks into buttons.

Of course, supplying this role alone is not sufficient to convert a DIV or SPAN into a fully functioning, keyboard-accessible button. That behaviour must be added with JavaScript. Better yet, just try and use a native button element which will give you the screen reader role and keyboard behaviour for free.

aria-expanded

Indicate whether the element, or another element it controls, is currently expanded or collapsed.

Often used on 'expando' or 'show more' buttons that dynamically hide & show content. Also used on buttons and comboboxes that control menu and auto-complete overlays respectively.

aria-live

Announce any content or visibility changes inside an element that may be outside the user's area of focus.

Changes can be notified assertively, interrupting the screenreaders current announcement, or politely after the current announcement has finished.

By default, only the updated content inside the element will be announced, rather than the entire content of the element. This behaviour can be changed with the aria-atomic property.

It is good practice to try and use live-regions to announce *what* content has changed, rather than the actual content itself.

role=dialog & role=alertdialog

Dialog informs the user that they are inside of a modal window. A modal window typically prompts the user to enter some information, confirm a response or to browse new content.

Alertdialog effectively converts the entire dialog into an assertive live-region. This behaviour may be desirable in cases where the window contains updating content, error messages or workflows.

Every modal window requires a role of dialog or alertdialog.

WARNING! Our final set of ARIA attributes listed below are deemed dangerous and therefore essential knowledge in order to *avoid* their misuse!

---

aria-haspopup

Used on a button or menuitem that triggers a menu (or submenu) overlay.

The term 'menu' is important here because this property is to be used in conjunction with the menu and menubar roles only.

The JAWS screenreader will announce 'has menu', so you can see that despite the name of this attribute, it is **not** intended to be used on elements that open other kinds of popup overlay - such as modals, tooltips, flyouts, etc.

aria-hidden

Removes an element for screenreader users only.

Caution must be exercised when applying this attribute because it is rare that we want to give screenreader users a different experience, or different content, to other users.

This attribute may only be used if intended to improve the experience for users of assistive technologies by removing redundant or extraneous content.

Note that setting aria-hidden to true is not necessary if the element already has CSS 'display: none' or the HTML5 hidden attribute.

role=presentation

Remove the default semantic meaning of an element.

You can think of this role as essentially converting an element into a meaningless DIV (block) or a SPAN (inline) but retaining it's default styling.

For example, when applied to a table, the screenreader would no longer announce it as a table or it's children as rows or columns, but it would still appear visually as a table. This behaviour can be desirable for pages or modules that use tables for layout rather than CSS.

Again, this is another attribute that must be used with extreme care.

role=application

The majority of web pages, and web developers, will never need to use this role.

By applying this role to an element, you are basically saying that you, as the developer, will now handle all keypresses inside that element rather than the screenreader. So when a screen reader user presses their arrow keys in an attempt to move their virtual cursor, nothing will happen, because you are now handling the behaviour of those keys inside this element.

Applying this role to an element can have severe consequences to screenreader users. Applying this role to the body tag for example, can render the entire page unusable.

Even if you *think* you need this role, you most probably don't, so please exercise extreme caution. You have been warned!

At the current time of writing the patterns that need role="application" are:

- Combobox

Checklist

A manual checklist is of course no substitute for automated testing, or static analysis, of your page. A checklist can though help us catch a certain class of issues that might not be surfaced by such tools, or that might result in false positives.

General

What?	Why?
Ensure your page has a valid doctype e.g. `<!doctype html>` for HTML5	Standards mode helps improve consistent screen reader behaviour.
Ensure your html tag has a valid lang attribute e.g. `<html lang="en">`	Informs the screen reader of which language to read your page in.
Ensure your head section has a valid title element	Informs the screen reader of the purpose and/or content of the page.
Ensure unique ids for all elements	Duplicate IDs can prevent screen reader from reading form labels.
Ensure dynamically generated ids are unique	Duplicate IDs can prevent screen reader from reading form labels.
Ensure HTML attributes are used to convey state rather than classnames e.g. `<button disabled>` rather than `<button class="disabled">`	Class names provide no semantics to screen readers
Ensure ARIA attributes are used to convey state rather than classnames e.g. `<a role="tab" aria-selected="true">` rather than `<a class="tab" class="selected">`	Class names provide no semantics to screen readers
Ensure `<i>` tags are not used as CSS icon holders. Use a `<span>` element instead.	These tags have specific semantics to screen readers.
Ensure a single h1 heading exists on the page	This tells the user where they are and the purpose of the page.
Ensure that all main page regions have a single h2 heading	This will help a screen reader user make a mental model of the page.

relevant ARIA roles (e.g. banner, nav, main, complementary)	This will help a screenreader user scan & navigate the page.
Ensure all page landmarks are labelled (except role="main").	This will provide a human readable description of the landmark.
Ensure that your main content wrapper has id="mainContent", tabindex="-1", role="main"	This is required by "Skip to Main Content" link in eBay global header.
Do not set a tabindex attribute other than -1 or 0	Only the DOM order should be used to dictate order of keyboard navigation. A value of -1 allows programmatic focus with JavaScript. A value of 0 adds element to natural tab order. Use with care.
Do not set a non-negative tabindex attribute on non-interactive elements or text	Non-interactive elements must never be in the tab order.
Ensure non-modal overlays are placed adjacent in the DOM to their nearest visual element	Allows natural tab order and screen reading order from page into flyout overlay
Ensure noscript tag is only used as a fallback for lazy loaded images	Noscript tag is an anti-pattern. Use progressive enhancement instead.

Autocomplete

Autocomplete is an extension of Combobox. In addition to the checklist below, please follow all items in Combobox checklist.

What?	Why?
Ensure text input has `aria-autocomplete="list"`	Informs assistive technology that list of suggestions (listbox) will be displayed

Carousel

A carousel is a composite containing two buttons and a viewport. The viewport contains *n* items and the buttons update the content of this viewport.

What?	Why?
Ensure the carousel is preceded by a heading	Screen reader users (all users, infact) want to know what the carousel contains
Ensure a list is used to markup the carousel items	A list informs screen reader user of number of items and current position in list
Ensure items outside of viewport are not in taborder	Only the visible items should be in tab order
Ensure items outside of viewport are hidden from screen reader using `aria-hidden="true"`	Only the visible items should be available to screen reader
Ensure any slideshow behaviour can be paused	Slideshows are distracting to users with cognitive disabilities
Ensure any auto-play slideshow behaviour loops only once	Some users prefer to wait for slideshow to end before interacting with page
Ensure filmstrip carousels have left/right pagination buttons	Keyboard user needs a way of paginating the carousel viewport
Ensure all pagination controls have an accessible label e.g. `<button class="icon-arrow-right" aria-label="Show next items" />`	Critical icon buttons need an accessible label for screen readers

Combobox

Combobox is a composite pattern containing a textbox, listbox (in flyout) and optional button (to expand/collapse).

What?	Why?
Ensure text input has `autocomplete="off"`	Disables built-in HTML5 autocomplete which interferes with this control
Ensure textbox has `role="combobox"`	Informs assistive technology that this input has an associated listbox
Ensure textbox has `aria-expanded` attribute	Informs assistive technology of listbox state (true or false
Ensure textbox has `aria-activedescendant` state	Informs assistive technology of current active suggestion in list
Ensure listbox has a unique id	The textbox input needs to reference this ID
Ensure suggestion listbox has `role="listbox"`	Informs assistive technology of listbox semantics
Ensure each option in listbox has `role="option"`	Informs assistive technology of listbox option semantics
Ensure each option in listbox has unique id	Used in conjunction with aria-activedescendant
Ensure active option in listbox has `aria-selected="true"`	Required by voiceover (but not other screen readers) to identify the active listbox option
Ensure listbox is navigable with up/down arrow keys	Allows keyboard user to make select an option
Ensure combobox has offscreen status region detailing number of suggestions	In some screen readers, combobox functionality will not be discoverable without this hint

Dialog

Our dialogs are always modal. It *is* feasible for a dialog to be non-modal, but these cases are typically less common.

What?	Why?
Ensure dialog has `role="dialog"`	Assistive technology will inform user of dialog behaviour when focus enters dialog
Ensure dialog is labelled with `aria-labelledby` or `aria-label`	Assistive technology will inform user of dialog purpose when focus enters dialog
Ensure dialog contains at least one interactive element	This allows us to set focus on a window element on open.
Ensure dialog has keydown event handler for `ESC` key	This allows keyboard users to quit the dialog
Ensure keyboard focus cannot leave dialog	Our dialog has modal behaviour
Ensure screen reader virtual cursor cannot leave dialog	Our dialog has modal behaviour

Forms

Coming soon.

Links & Buttons

Coming soon.

Images

Coming soon.

Live Regions

Coming soon.

Tables

Coming soon.

Stay tuned. more checklists for custom controls to follow...

FAQ

Listed below are frequently asked questions. If you have a question that is not answered here, please start a discussion.

How do I hide text offscreen?

Apply the `.clipped` class (or it's rules) to the element that you want to be visually hidden. The element will remain accessible to screen readers on desktop and mobile devices.

```
.clipped {
  position: absolute !important;
  clip: rect(1px 1px 1px 1px); /* IE6, IE7 */
  clip: rect(1px, 1px, 1px, 1px);
  padding: 0 !important;
  border: 0 !important;
  height: 1px !important;
  width: 1px !important;
  overflow: hidden;
}
```

For more information please read the excellent "When and How to Visually Hide Content" by Dennis Lembree.

Also, strictly speaking it's no longer 'offscreen'. The content remains *within* the area of the screen but is clipped in size.

When using Safari I can’t TAB through the links on the page. What’s going on?

You must enable the following option in Safari's preferences:

```
Safari -> Preferences -> Advanced -> Accessibility -> "Press Tab to highlight each ite
m on a webpage"
```

And here’s how to do the same for Firefox:

```
System Preferences -> Keyboard -> Keyboard Shortcuts -> Full Keyboard Access -> "All c
ontrols"
```

I'm using Microsoft Remote Desktop for Mac to test with JAWS on a Windows machine. My keyboard does not have an INSERT key. How do I send the INSERT command?

It is true that Mac keyboards do not have an INSERT key, and this is a known issue with MRD. The workaround is to install a hotkey application such as Karabiner.

Once installed, check the checkbox "Enable at only Remote Desktop Connection Client - Use F12 as insert".

Now you should be able to activate any JAWS commands that require the INSERT key (e.g. opening the Links List).

Why do we cater for users without JavaScript!?

It's true that only a very small percentage of users browse the web without JavaScript. Sometimes this is by choice, and sometimes not (a strict company security policy may disable JavaScript for all employees, for example).

We are not going to go out of our way to cater for these individuals.

However, there are other *unknown* factors which can potentially place the user into a non-JS or partial-JS state, for example:

- A script fails to load due to poor connectivity
- A script executes much later or slower than expected
- A script fails due to incompatible browser support

> “We don’t have any non-JavaScript users”
> No, *all* your users are non-JS while they’re downloading your JS
>
> *Jake Archibald (@jaffathecake) May 28, 2012*

The only realistical way of defending against the *unknown* is by building our pages in the Progressive Enhancement way; we make sure our most important content and functionality[1] is always available to users in **any** browser, in **any** situation.

1

[1] Of course, in certain situations, some functionality simply cannot be replicated without JavaScript - a single page app (SPA) which relies heavily on realtime, client-side events, might fall under this category.

How do I know where page headings are required?

Identifying page headings is the responsibility of the design team.

As a rough guideline, know that every logical, visual *block* of the page **must** have a heading.

Sometimes a heading will be hidden visually offscreen and available only to screenreaders. Again, defining these types of headings is the responsibility of the design team.

Should SPACEBAR or ENTER key activate links & buttons?

Both the SPACEBAR and ENTER key should activate a button.

Only the ENTER key should activate a link. Pressing the spacebar whilst keyboard focus is on a link should scroll the page down.

Should I use aria-label or aria-labelledby?

Use aria-label in cases where a text label is not visible on the screen. If there is visible text labeling the element, use aria-labelledby instead.

Should I use radio, checkbox or select?

A radio group allow only one choice to be selected.

A checkbox group allows zero or more choices to be selected.

A select group allow one or more choices to be selected.

References

In the making of this book, the following material was referenced:

- W3C: HTML5 Spec
- W3C: ARIA Spec
- W3C: WAI-ARIA 1.0 Design Patterns
- W3C: ARIA in HTML
- W3C: ARIA Authoring Practices

Utilities

In an effort to make the example code more concise and readable, several shivs, plugins and classes were used in the creation of the examples.

Shivs

- ES5 Shim
- HTML5 Shiv

Plugins

- jquery-next-id
- jquery-common-keydown
- jquery-focusable
- jquery-roving-tabindex
- jquery-active-descendant
- jquery-button-flyout
- jquery-link-flyout
- jquery-focus-exit
- jquery-mouse-exit
- jquery-keyboard-trap
- jquery-screenreader-trap
- jquery-prevent-document-scroll-keys

Classes

```
/* clip element visibility, making it accessible to screen reader only */
.clipped {
    position: absolute !important;
    clip: rect(1px 1px 1px 1px); /* IE6, IE7 */
    clip: rect(1px, 1px, 1px, 1px);
    padding: 0 !important;
    border: 0 !important;
    height: 1px !important;
    width: 1px !important;
    overflow: hidden;
}
```