intopia

The accessibility tree, ARIA and accessible names

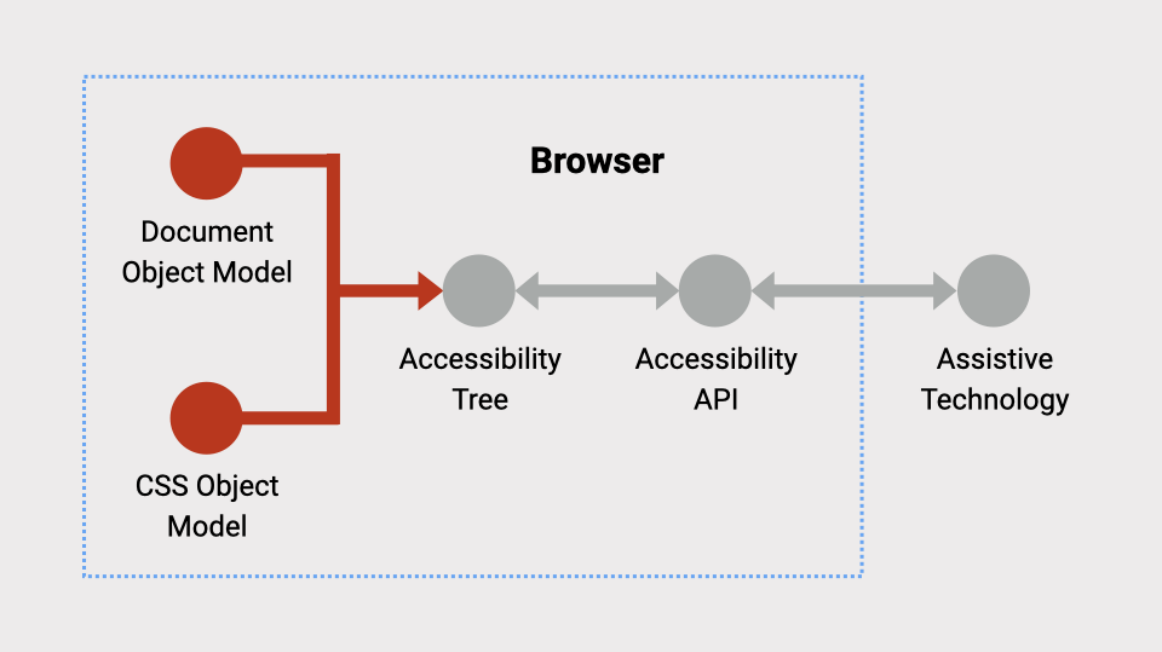
Engineering Session 1

What will we cover?

Part 1: The accessibility tree

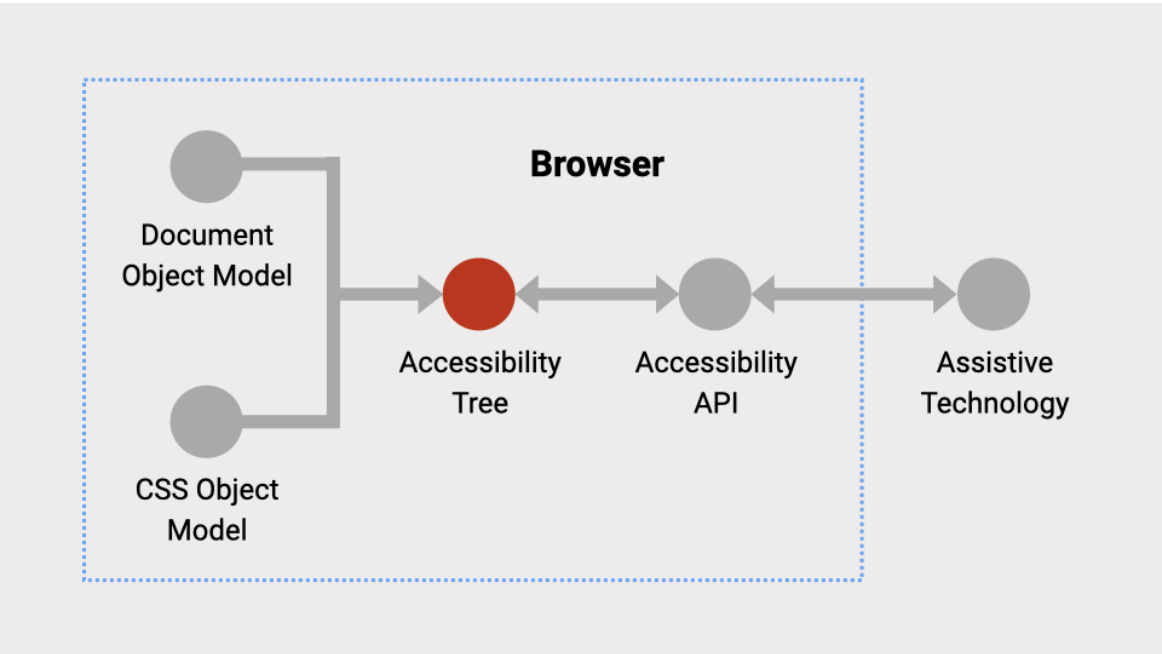
What is the accessibility tree?

Browsers use the **DOM and some CSS** (where relevant) to generate an Accessibility Tree.

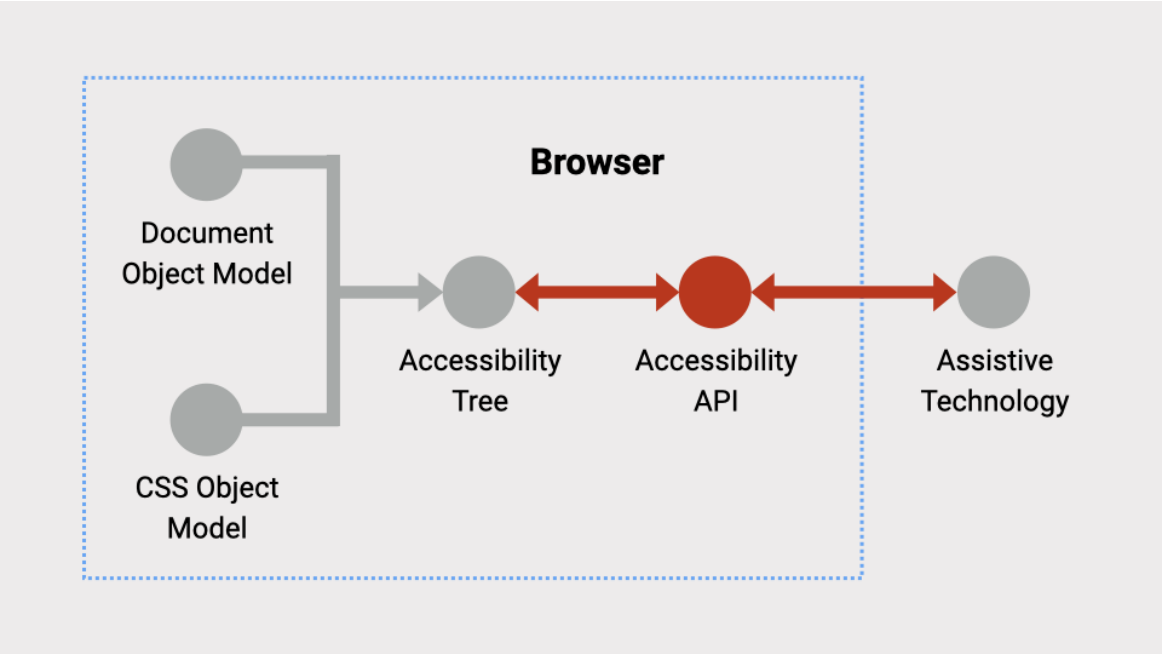


This **accessibility tree** consists of information about specific HTML elements, including their:

- Name (e.g. “Submit”, “Full name”)
- Role (e.g. button, link, textbox)
- State (e.g. checked, expanded, disabled)
- Value (e.g. input from the user)



Accessibility APIs communicate this information from the accessibility tree to assistive technologies - such as screen readers.



Each browser supports different Accessibility APIs, which means they each **produce their own unique accessibility tree**.

Any **questions or comments**?

Activity: Viewing the accessibility tree

Accessing the activity:

Go to the All about mammals page.

We will use the **Chrome browser** for this activity as it has very easy-to-use accessibility tree.

Finding the “Accessibility” tab

Step 1: Opening Chrome Developer Tools

- Right-click anywhere on the page and select “Inspect”.
- This will open Chrome Developer Tools.

Step 2: Finding the “Accessibility” tab

- The “Elements” tab will be active by default.
- This shows the DOM tree and “Styles” panel.
- The “Accessibility” tab is to the right of “Styles”.
- You may need to click the “More tabs” icon.

Step 3: Select the “Accessibility” tab

- Select the “Accessibility” tab.
- This tab displays:
  - “Accessibility Tree”
  - “ARIA Attributes”
  - “Computed Properties”
  - “Source Order Viewer”

Step 4: Viewing “Computed Properties”

We will spend most of our time exploring the “Computed Properties” window.

The activity

1. Inspect the “Find out more” link

In the accessibility panel, find the element’s `name` and `role`.

Answer

- Name: *Find out more*

- Role: *link*

All important elements **must have names and roles**, so they can be understood by assistive technologies.
(We will look at accessible names in detail soon)

2. Inspect the image

In the accessibility panel, find the element's `name` and `role`:

Answer

- Name: *A common wombat standing on the forest floor*
- Role: *img*

3. Inspect the "Full name" `<input>`

In the accessibility panel, find the element's `name` and `role`:

Answer

- Name: *Full name*
- Role: *textbox*

4. Inspect the "All about mammals" heading

In the accessibility panel, find the element's `role` and `level`:

Answer

- Role: *heading*
- Level: *1*

5. Inspect the “Land mammals” table

In the accessibility panel, find the `<table>` element’s `name` and `role`:

Answer

- Name: *Land mammals*
- Role: *table*

Inside the table, all essential elements **also have roles**:

Answer

- `<tr>` role = *row*
- `<th>` role = *columnheader*
- `<td>` role = *gridcell*

6. Inspect the “Vombatus” content

Does this element provide any information in the accessibility panel:

Answer

- *Accessibility node not exposed*

Some less important elements are **not exposed in the accessibility tree**.

7. Inspect the “Phone” `<input>`

In the accessibility panel, find the element’s `description`:

Answer

- Description: *Include area code*

Descriptions are used to **provide additional information** for some elements.
(We will look at accessible descriptions in detail soon)

8. Inspect the “Email” `<input>`

Is this element defined as `required` in the accessibility panel?

Answer

- Required: *true*

This tells assistive technologies **that the form control is required** and must be filled in before submitting the form.

9. Add text into the “Email” `<input>`

Does the element now have a `value` in the accessibility panel?

Answer

- Value: *"abc@com.au"*

This tells assistive technologies what the user has added to the form field - **allowing the user to review the information** before submitting the form.

10. Select an option from the dropdown

Select “Aardvark” from the “Favourite mammal” dropdown. Does the `<select>` element now have a `value` in the accessibility panel?

Answer

- Value: *"Aardvark"*

11. Check a checkbox

Check “Yes” from the “Bats” checkbox group. Does the checkbox now have a `checked` status in the accessibility panel?

Answer

- Checked: *true*

This tells assistive technologies that the form control **has been checked**.

12. Click the “Submit” button

This will create some fake form errors. Inspect the “Phone” `<input>`. Is this element defined as `invalid` in the accessibility panel?

Answer

- Invalid user entry: *true*

This tells assistive technologies that the form control **is currently invalid and needs to be resolved**.

13. Changing an element’s role

Add `role="button"` to the “Classification” heading in the DOM. What is the element’s `role` in the accessibility panel?

Answer

- Role: `button`

This shows how we can completely change the nature of an element **in the accessibility tree**.

> *“With great power comes great responsibility”*

Benjamin Franklin "Ben" Parker

Lots of properties

There is a wide range of possible properties that can be presented in the accessibility tree, **depending on the element**:

```
Name: [ accessible name as a text string ]
Role: [ pre-defined list of roles ]
Description: [ description as a text string ]
Value: [ current value as a text string ]
Required: true | false
Expanded: true | false
Checked: true | false
Disabled: true | false
Described by: [element #id]
Labeled by: [element #id]
```

Any **questions or comments**?

Part 2: An introduction to ARIA

What is ARIA?

ARIA is the abbreviation of **Accessible Rich Internet Applications**.

ARIA **defines ways to make websites and web apps more accessible** to people with disabilities.

ARIA is:

- A set of custom HTML attributes.
- These attributes add information to the accessibility tree.
- This information be used to help assistive technologies.

WAI-ARIA 1.2 became the **W3C Recommendation** on 06 June 2023.

What problems is ARIA trying to solve?

ARIA aims to solve two problems

- Dynamically injected content.
- Non-native widgets/components.

ARIA is like a **polyfill** between the HTML we have now and the HTML we wish we had or may soon have!

In fact, our aim is to **gradually avoid or even remove ARIA** as the HTML specification expands over time.

Some examples of HTML expansion:

- The `<dialog>` element and related `open` attribute.
- The `inert` attribute.
- The `<details>` element.
- The `popover` attribute.

Roles, states and properties

ARIA can be used to adjust how elements are **presented in the accessibility tree**.

- Roles
- Properties
- States

ARIA roles

ARIA roles can be used to **add or change the semantic meaning** of HTML elements in the accessibility tree.

ARIA roles are **HTML attributes** written as:
`role="[role name]"`

```
<div role="button"></div>
<div role="combobox"></div>
<div role="menu"></div>
```

```
<div role="application"></div>
<div role="banner"></div>
<div role="alert"></div>
<div role="dialog"></div>
```

ARIA properties

ARIA properties can be used to **provide additional information** to HTML elements in the accessibility tree.

ARIA properties are **HTML attributes** written as:
`aria-[property]="[value]"`

```
<div aria-controls="aaa"></div>
<div aria-details="bbb"></div>
<div aria-errormessage="ccc"></div>
<div aria-owns="ddd"></div>
<div aria-relevant="eee"></div>
```

ARIA states

ARIA states can be used to inform users of the **current state of elements** in the accessibility tree.

ARIA states are **HTML attributes** written as:

```
aria-[state]="[value]"
```

```
<div aria-busy="true"></div>
<div aria-busy="false"></div>

<div aria-current="page"></div>
<div aria-current="step"></div>
<div aria-current="location"></div>

<div aria-disabled="true"></div>
<div aria-disabled="false"></div>
```

ARIA can’t do everything!

- Modify an element’s visual appearance.
- Modify an element’s behaviour.
- Add focusability to elements.
- Add keyboard functionality to elements.

Support for ARIA

How can we determine if ARIA is supported by **each of the different browsers and assistive technologies**?

A11ySupport.io documents how individual ARIA attributes are **supported across browsers**.

HTML test cases documents how individual HTML elements and ARIA attributes are **supported across some screen reader/browser combinations**.

Where possible, you should conduct your own tests to **determine if the relevant ARIA attributes are supported**.

Any **questions or comments**?

Activity: Creating a fake checkbox

This is a demonstration of how ARIA can be used to make a poorly built component **more accessible**. It is not a recommended practice.

Accessing the activity:

Creating a fake checkbox - start.

Imagine you have to make a checkbox group, but you have to to use `<div>` elements **instead of native checkboxes**.

As you will see:

- **Some basic JavaScript** is already in place.
- **Keyboard focus** is in place via `tabindex="0"`.

However, this widget is completely **inaccessible to assistive technologies**. Let's look at it in the accessibility panel.

- The elements have **no role defined**.
- The elements have **no checked state defined**.
- The `<h3>` heading is **not programmatically associated** with the checkbox group.

Step 1:

Add `role="checkbox"` to each of the fake checkboxes.

```
<div
  class="checkbox"
  tabindex="0"
  role="checkbox"
>
  Lettuce
</div>
```

```
<div
  class="checkbox"
  tabindex="0"
  role="checkbox"
>
  Tomato
</div>
```

```
<div
  class="checkbox"
```

```
  tabindex="0"
  role="checkbox"
>
  Mustard
</div>
```

Step 2:

- Add `aria-checked="true"` to the first fake checkbox.
- Add `aria-checked="false"` to other fake checkboxes.

```
<div
  class="checkbox"
  tabindex="0"
  role="checkbox"
  aria-checked="true"
>
  Lettuce
</div>
```

```
<div
  class="checkbox"
  tabindex="0"
  role="checkbox"
  aria-checked="false"
>
  Tomato
</div>
```

```
<div
  class="checkbox"
  tabindex="0"
  role="checkbox"
  aria-checked="false"
>
  Mustard
</div>
```

If we inspect any of the fake checkboxes in the accessibility tree, **they now have roles and states**.

Step 3:

Add `role="group"` to the parent container so that we can create a fake `<fieldset>`.

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

Step 4:

Add `aria-labelledby="group-label"` to the parent container so we can give it an accessible name.

```
<div
  role="group"
  aria-labelledby="group-label"
>
</div>
```

Step 5:

Add `id="group-label"` to the `<h3>` element. Now it will operate like a fake `<legend>`.

```
<h3 id="group-label">Choose some toppings</h3>
```

If we inspect the parent container in the accessibility tree, **it now has an accessible name and a role**.

Review the Activity:

Creating a fake checkbox - finished.

Part 3: Accessible names

What are accessible names?

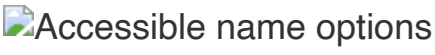
Accessible names are short text strings that authors associate with an element to **provide assistive technology users with a label for the element**.

Accessible names have **two primary purposes** for assistive technologies users:

- Convey the purpose or intent of the element.
- Distinguish the element from other elements on the page.

Accessible names are conveyed to assistive technologies **via the accessibility tree**.

Chrome's accessibility tree shows **all the possible options** that could be used to provide the accessible name.



Chrome also displays the **final computed accessible name** as a text string.

Any **questions or comments**?

The `aria-labelledby` attribute

This is where a primary element is **given a label** (another name for an accessible name) by a secondary element's text string.

```
<section aria-labelledby="aaa">
  <h3 id="aaa">Contact details</h3>
</section>

Section accessible name: "Contact details"
```

The secondary element **does not need to be a child** of the primary element.

```
<p id="bbb">Buy Lawn Mower</p>
<button aria-labelledby="bbb">Buy</button>

Button accessible name: "Buy Lawn Mower"
```

More than one label can be applied to the primary element via space-separated values in the `aria-labelledby` attribute.

```
<p id="ccc">Buy Lawn Mower</p>
<button aria-labelledby="ccc ddd">Buy</button>
<p id="ddd">On special</p>

Buttons accessible name: "Buy Lawn Mower On special"
```

You can even use the **text string from within the element itself** to create an

aria-labelledby value.

```
<p id="yyy">Lawn Mower</p>
<button id="xxx" aria-labelledby="xxx yyy zzz">Buy</button>
<p id="zzz">On special</p>

Buttons accessible name: "Buy Lawn Mower On special"
```

An accessible name can be applied to an element via aria-labelledby **even if the element is hidden**.

```
<p id="ccc" style="display: none">Buy now</p>
<button aria-labelledby="ccc">Buy</button>

Buttons accessible name: "Buy now"
```

An element with an empty text string will **generate an empty accessible name**.

```
<button aria-labelledby="xyz">Hello world</button>
<p id="xyz"></p>

Buttons accessible name: ""
```

Activity:
Adding `aria-labelledby`

Accessing the activity:

- Adding aria-labelledby - start
- Adding aria-labelledby - finished

Example 1: Adding a name to a section

How could we **provide an accessible name** for the `<nav>` element using content from another element?

Add `aria-labelledby="aaa"` to the `<nav>` element:

```
<nav aria-labelledby="aaa">
  <h4>About us</h4>
</nav>
```

Add `id="aaa"` to the `<h4>`:

```
<nav aria-labelledby="aaa">
  <h4 id="aaa">About us</h4>
</nav>
```

Check the accessibility tree to see if the `<nav>` **now has an accessible name**.

Example 2: Adding a complex name to a button

The “Buy” button may not have **enough context for screen reader users**.

We can **provide additional context** by using relevant information on the page to create an accessible name.

Add `aria-labelledby="xxx yyy zzz"` to the `<button>`.

```
<button
  aria-labelledby="xxx yyy zzz"
>Buy</button>
```

Add `id="xxx"` to the `<button>`.

```
<button
  id="xxx"
  aria-labelledby="xxx yyy zzz"
>Buy</button>
```

Add `id="yyy"` to the `<span>` associated with “Lawn Mower” text.

```
<span id="yyy">Lawn Mower</span>
```

Add `id="zzz"` to the `<span>` associated with “On special” text.

```
<span id="zzz">On special</span>
```

Check the accessibility tree to see if the `<button>` **now has a complex accessible name**.

The `aria-label` attribute

The `aria-label` attribute provides a label **directly to the element itself**.

```
<button aria-label="Close and return">
  Close
</button>
```

An element with an `aria-label` value that contains only a space character will **generate an empty accessible name**.

```
<button aria-label=" ">Hello world</button>

Buttons accessible name: ""
```

Activity: Adding `aria-label`

Accessing the activity:

- Adding aria-label - start
- Adding aria-label - finished

Adding an aria-label

In this example, the `<button>` element has a visible text label and an accessible name of “Close”.

While the visible text label may provide enough context for sighted users, the accessible name **may not provide context for other users**.

So, we may want to provide this element with a **meaningful accessible name**.

Add `aria-label="Close and return to banking"` to the `<button>`.

```
<button aria-label="Close and return to banking">
  Close
</button>
```

Check the accessibility tree to see if the `<button>` element **now has a new accessible name**.

Activity: Reviewing accessible names

Accessing the activity:

- Reviewing accessible names
- Reviewing a concatenated accessible name

Part 4: Accessible descriptions

What are accessible descriptions?

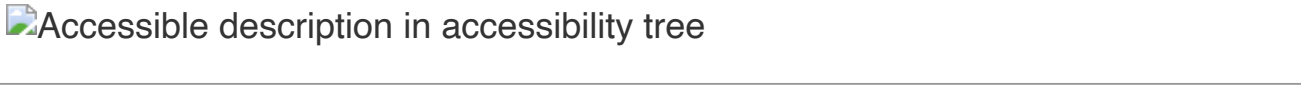
In some cases, accessible objects may need more than an accessible name to **provide additional context.**

For example, **help text or error messages** associated with individual form

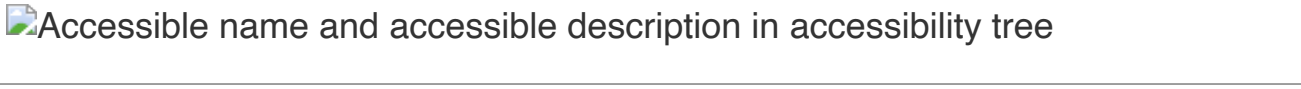
controls.

This additional information is referred to as an **accessible description** if it is exposed in the accessibility tree.

Accessible descriptions are **defined as text strings**.



Accessible names and descriptions are **two totally separate concepts**. They fill two different slots in the accessibility tree.



Any **questions or comments**?

The `aria-describedby` attribute

This is where a primary element is **given a description** by a secondary elements text string.

```
<label for="a">Phone</label>
<input
  id="a"
  type="text"
  aria-invalid="true"
  aria-describedby="i1"
```

```
>
<p id="i1">Error: Number must include all 8 digits</p>
```

As with `aria-labelledby`, the secondary element **does not need to be a child** of the primary element.

More than one description can be applied to the primary element. They need to be space-separated values in the aria-describedby attribute.

```
<label for="a">Phone</label>
<p id="hint1">Include an area code</p>
<input
  id="a"
  type="text"
  aria-invalid="true"
  aria-describedby="hint1 error1">
<p id="error1">Error: Number must include all 8 digits</p>
```

Any **questions or comments**?

Activity: Adding accessible descriptions

Accessing the activity:

- Adding accessible descriptions - start
- Adding accessible descriptions - finished

Example 1

Programmatically associate the `help-text` element with the `<input>` and **check the accessibility tree**.

```
<label for="phone">Phone number</label>
<span id="aaa">Make sure to include your area code</span>
<input aria-describedby="aaa" id="phone" type="text">
```

Example 2

Add a `title` attribute with a value of “Ensure a valid email address” to the `input` and **check the accessibility tree**.

```
<label for="email">Email address</label>
<input title="Ensure a valid email address" id="email" type="text">
```

Example 3

Programmatically associate the `help-text` and `error-message` elements with the `<input>` and **check the accessibility tree**.

```
<label for="name">Name</label>
<span id="yyy">Add your full name</span>
<input aria-describedby="zzz yyy" id="name" type="text">
<span id="zzz">Error: You must include a name</span>
```