Building accessible components

There are several common UI components that are prevalent in almost every website or Web application - each of them presenting their own unique accessibility challenges and we will look at some of the most popular ones during this lesson.

Every component that we will look at, will teach us **different accessibility techniques** which are themselves **common patterns that can be used in more complex widgets and scenarios**. While discussing each component, we will talk about its characteristics in terms of its:

- **HTML (semantic) structure**
- **ARIA attributes**
- **Keyboard accessibility rules**
- **Focus management rules**

...and other unique traits that may appear along the way.

Before we start, I would like to share with you one of the most useful resources when it comes to learning the rules on how these components should operate - the WAI Patterns.

WAI stands for **Web Accessibility Initiative** and is an **initiative** formed within the **W3C (World Wide Web Consortium)** with the mission of **establishing clear Web Accessibility standards and guidelines**.

Their ARIA Authoring Practices Guide is one of the best accessibility resources available today and I highly recommend you spend some time looking through it. I personally have learned a tremendous amount just from that website alone.

One of their resources is the Patterns page that describes **how some common UI components should work from an accessibility standpoint**.
We will distill the information they provide into the most crucial parts and build out some of the components mentioned on their page, from scratch.

Accordion

One of the simplest looking common components out there - the accordion - can teach us a lot about ARIA attributes, as well as using correct semantics and is a good place to start our building process.

The Accordion component is composed of 2 parts: **the header** and **the panel**.

Accordion header

The Header needs to be **an interactable element**, for example a **button**, such that users can click it to open/close the Accordion panel.
The button must have the following ARIA attributes so that it properly conveys the information to assistive technologies that **it is controlling the panel opening and closing.**

aria-controls

This attribute is used when we want to convey a relationship between 2 elements - where **one element controls the presence of another element**.
We add the `aria-controls` attribute on the **controlling element** - with the value being **the ID of the element that's being controlled**. Of course, this means that we must add that ID as an attribute to the element that's being controlled.

```html
HTML
<button aria-controls="options-menu">Options</button>

<ul id="options-menu">
  <li>Option 1</li>
  <li>Option 2</li>
  <li>Option 3</li>
</ul>
```

aria-expanded

This attribute is used to actually **convey information about state**. We want to use `aria-expanded` on **elements that control other elements that can expand/collapse** (open/close).
Our responsibility as developers is to **dynamically update the value of this attribute** as the controlled element expands/collapses.

Accordion panel

The panel is the content that holds the actual detailed information which we are showing/hiding using the header button.
When the Accordion is closed - **this content should be invisible to the user and assistive technologies (screen readers)**.

We want to give the panel a **role** of **"region"** to make it a **landmark that screen readers can navigate to**. We then need to add the `aria-labelledby` attribute so it connects to the trigger button. That way screen reader users will know what content the region contains.

Combining all of the above, we can construct a very simple Accordion component.

HTML
```html
<div class="accordion">
  <button id="trigger" aria-expanded="false" aria-controls="panel">
    Put your title here
  </button>

  <div role="region" id="panel" aria-labelledby="trigger">
    <div class="panel-content">
      Some content here... whatever you want
    </div>
  </div>
</div>
```

We would need a **single source of truth** to tell us whether the accordion is expanded or collapsed. For that we will use **the trigger button's `aria-expanded` state**.

JavaScript
```javascript
const accordion = document.querySelector('.accordion');
const triggerButton = accordion.querySelector('button');

triggerButton.addEventListener('click', () => {
  toggleAccordion(triggerButton)
});

function toggleAccordion(trigger) {
  const isOpen = trigger.getAttribute('aria-expanded') === 'true';
  trigger.setAttribute('aria-expanded', String(!isOpen));
};
```

```css
.panel-content {
  visibility: hidden;
}

.accordion:has(button[aria-expanded="true"]) {
  .panel-content {
    visibility: visible;
  }
}
```

Note: The full code is deliberately omitted from these materials due to brevity, but you can check out the code to see how it's done. There are some **neat animation and CSS selector tricks** that make this work with a single source of truth.
Feel free to check out the more in-depth look on the Authoring Practices Guide for constructing an Accordion component.

Tabs

Tabs are essentially composed of 2 parts as well: **the actual Tablist and the Tab panels**.

Tablist (Tabs)

The list of Tabs is the most crucial part of the Tabs component. It is composed out of a **container for the tabs** - **tablist** and the **actual tabs**.
The tabs container should have the **role** of **"tablist"** while each tab element should have the **role** of, well, **"tab"** - simple enough.

Then we need to make sure we have the proper ARIA attributes for each tab.
We of course need to **programmatically connect each tab to the tab panel that it controls** (shows/hides) using `aria-controls` - similarly to how we did with the Accordion header and panel.

aria-selected

This attribute is used with **groups of selectable elements** (such as tabs) **where only one element can be selected at a time** (whatever "selected" means for our case).
We need to add this attribute with the **value of "true"** to the currently selected (active) tab - and dynamically update this value as the users toggle through tabs.

Tab panels

The tab panels are the actual individual pieces of content that correspond to each tab and they need to have a **role** of **“tabpanel”**.
In order for us to connect them with their assigned tab, we need to do the following:

- Assign an **ID** to each panel so that the **tab can connect to it via `aria-controls`**
- Add an **`aria-labelledby`** attribute referencing the **tab button’s content**

After that, it’s up to us how we want to show/hide the actual tab panels.

We can go with a simple `display: block` and `display: none` toggle, or we can **visually hide them but keep them in the DOM** using **`aria-hidden=”true”`** on the non-active panels.
The latter one is used for cases when, for example, we want to **animate our tab panels**.

So let’s do it, let’s construct the Tabs component and see what tricky challenges we face along the way:

```html
<div class="tabs" role="tablist">
  <button id="tab-1" role="tab" aria-selected="true" aria-controls="panel-1">
    Tab 1
  </button>

  <button id="tab-2" role="tab" aria-selected="false" aria-controls="panel-2">
    Tab 2
  </button>

  <button id="tab-3" role="tab" aria-selected="false" aria-controls="panel-3">
    Tab 3
  </button>
</div>

<div class="tab-panels">
  <div id="panel-1" role="tabpanel" aria-labelledby="tab-1">...</div>
  <div id="panel-2" role="tabpanel" aria-labelledby="tab-2" class="hidden">...
  <div id="panel-3" role="tabpanel" aria-labelledby="tab-3" class="hidden">...
</div>
```

Note: This may not be the final version of the tabs’ HTML structure - but we will get to that a bit later. For now, let’s add the JavaScript controls and necessary styles.

JavaScript
```javascript
const tabList = document.querySelector('[role="tablist"]');
const tabs = tabList.querySelectorAll('[role="tab"]');

tabs.forEach((currentTab) => {
  currentTab.addEventListener('click', () => {
    // ? Go through each tab and toggle it appropriately
    tabs.forEach((tab) => {
      toggleTab(tab, tab === currentTab);
    });
   });
});

function toggleTab(tab, active) {
  tab.setAttribute('aria-selected', String(active));

  // ? Toggle the corresponding tab panel
  const panelId = tab.getAttribute('aria-controls');
  const panel = document.getElementById(panelId);
  panel.classList.toggle('hidden', !active);
}
```

Nothing fancy for the styling, just make sure your tab buttons have a **distinctive look when they are active vs when they are inactive**.

CSS
```css
[role="tab"] {
  &:focus-visible {
    outline: 2px solid blue;
    outline-offset: 2px;
  }

  &[aria-selected="true"] {
    background: blue;
    color: white;
  }
}
```

Notice we are using **`aria-selected` as the source of truth to know if a Tab is active**.

And that's it! We now have fully functioning Tabs... or do we?

Keyboard controls

Now when we built the Accordion, we didn't really talk about the keyboard controls because we basically got all of them built-in for free - because we used a button for the trigger - but Tabs are a little bit different in this regard.

The challenge we need to tackle in this case is **keyboard users**. They can navigate the page by focusing on the interactive elements and just tabbing through until they get to their desired place in the page.
Now, if we have a component such as our Tabs as currently implemented, imagine instead of 3 tabs there were 10+ tabs in the list: **Users would have to tab through all 10+ tab buttons to get to the first interactable element in the first tab panel**.
This is a usability nightmare so we need to do something about it.

Roving tabindex

Roving tabindex is a technique that makes sure **only 1 element within a group of elements is part of the focus order** - all the other ones are not.
We do this by assigning a **tabindex of "0" to the element we want to be focusable (active)** and also assigning a **tabindex of "-1" to all other elements in the group**.
Then, when we want to change the focusable/active element in the group, we just move the **tabindex="0"** value to the desired element and make all the other ones have a **tabindex of "-1"** and repeat the process.

The technique's name fits well here - we are moving around the `tabindex` property within a group of elements.
The reason why this technique is helpful in this case is because **it makes only 1 tab button actually focusable at a time**, so pressing "Tab" while focused on the currently active tab will move the focus within the tab's panel (or to the next-in-line focusable element outside the tabs component altogether).
Regardless where the focus moves next, this is exactly what we want! Users no longer need to tab through all 10+ tab buttons just to get to the next focusable element.

But now the question becomes - **how do we move through the tabs with our keyboard**?
We will do that using the **arrow keys** of our keyboard. Depending on the **direction of the tabs** (in this case they are horizontal) we will use the "ArrowLeft" and "ArrowRight" keys to move through the tabs and cycle back when we get to the edges.

Combining this keyboard navigation with the “Roving tabindex” technique - we will finalize our Tabs component and make it fully accessible. Let’s tweak the HTML and JavaScript to reflect this final addition:

HTML
```html
<div class="tabs" role="tablist">
  <button
    id="tab-1"
    role="tab"
    aria-selected="true"
    aria-controls="panel-1"
    tabindex="0">
    Tab 1
  </button>

  <button
    id="tab-2"
    role="tab"
    aria-selected="false"
    aria-controls="panel-2"
    tabindex="-1">
    Tab 2
  </button>

  <button
    id="tab-3"
    role="tab"
    aria-selected="false"
    aria-controls="panel-3"
    tabindex="-1">
    Tab 3
  </button>
</div>
```

JavaScript
```javascript
const tabList = document.querySelector('[role="tablist"]');
const tabs = tabList.querySelectorAll('[role="tab"]');

tabs.forEach((currentTab, idx) => {
  currentTab.addEventListener('click', () => {
    // ? Go through each tab and toggle it appropriately
    tabs.forEach((tab) => {
      toggleTab(tab, tab === currentTab);
    });
  });

  currentTab.addEventListener('keydown', (e) => {
    let newIdx;

    if (e.key === 'ArrowRight') {
      newIdx = (idx + 1) % tabs.length;
    } else if (e.key === 'ArrowLeft') {
      newIdx = (idx - 1 + tabs.length) % tabs.length;
    } else {
      return;
    }

    tabs[idx].setAttribute('tabindex', '-1');
    tabs[newIdx].setAttribute('tabindex', '0');
    tabs[newIdx].focus();

    e.preventDefault();
  });
});

function toggleTab(tab, active) {
  tab.setAttribute('aria-selected', String(active));
  tab.setAttribute('tabindex', active ? '0' : '-1');

  // ? Toggle the corresponding tab panel
  const panel = document.getElementById(tab.getAttribute('aria-controls'));
  panel.classList.toggle('hidden', !active);
}
```

Note: The full code is deliberately omitted from these materials due to brevity, but you can check out the code to see how it's done.
Feel free to check out the more in-depth look on the Authoring Practices Guide for constructing a Tabs component.

Modal

The Modal is a tricky piece of UI to build correctly, mainly because it has a number of moving parts that need to come together to make it work.
Given that, we will break down all the steps needed to create a Modal **from scratch** and go through them one by one to create our own custom Modal component. After that, we will see an easier method for using Modal components with today's modern HTML.

First things first - let's settle on some **terminology**. There are a lot of terms thrown around, labeling different kinds of UI elements such as **Dialogs, Modals, Overlays, Popups, Poppovers etc** and I would like to direct you to this amazing article on the terminology debate if you want to learn more about the exact definitions and distinctions between these UI elements.
For the sake of this course (and our sanity), we will simplify our definitions to 2 terms: **Modal and Dialog**.

The Modal is the component that we will look at today - which is an element that appears **in front of the rest of the UI** and also **makes the rest of the UI unreachable** until the action in the Modal is completed. Think of use cases like **filling in a form** or **confirming a deletion of an important resource** and similar cases like that.
The technical definition of this element is actually a **"Modal Dialog"** - the "modal" part suggesting the "block the rest of the UI" functionality.

The Dialog on the other hand is quite similar to the Modal, with the difference being that **it doesn't block the rest of the UI from being interacted with**. This essentially means that Dialogs in this sense, shouldn't contain important and critical actions that users need to complete. Think of **pop up chats with an AI assistant** or a **find-and-replace window in our IDEs etc**.
The technical definition of this element is actually a **"Non-Modal Dialog"** - meaning it is not blocking the rest of the UI.

So with that out of the way - let's build a Modal!
To start off, we will look at the **HTML structure of a Modal component**, then we will move on to the **toggle (keyboard) controls** and **focus management rules** that go along with it.
Finally, we'll see how **modern HTML** makes building this kind of a component easier than ever.

HTML Structure

When we talk about the HTML structure of a Modal, we really mean the HTML structure of the **Modal container element**.

This element needs a **role** of **“dialog”** to start with.
Next, we need to add the ARIA attribute **`aria-modal`** and set its value to **“true”**. We do this because the Modal component we are building will essentially **block the rest of the UI from being used until the Modal is closed**. This ARIA attribute tells assistive technologies that that is how the Modal will work.

After that we just need to label the Modal container, either by using **`aria-label`** and setting a primary label, or (and this is the preferred way) by using **`aria-labelledby`** and programmatically connecting it with the ID of the **title of the modal** that’s inside the Modal container. We usually build our Modals to have a title (some heading) and we can use this element to act as the primary label of the Modal.

HTML
```html
<div role="dialog" aria-modal="true" aria-labelledby="modal-title">
  <h3 id="modal-title">Modal title</h3>
  <div>This is some content inside the modal</div>
</div>
```

Of course it must start off as hidden and for that we can just set a **`display: none`** on it.
We also need a so-called **backdrop** element to **block the rest of the UI from interacting with it**. Finally, we will add a **button that will open the Modal**.

Now, that button is also a very important piece of this puzzle, we’ll call it the **Modal’s trigger**. This button needs a little bit of attention as well, because we need to tell assistive technologies **what kind of element does this button trigger (open)** and for that we will need `aria-haspopup`.

aria-haspopup

This attribute is added to **elements that open some kind of popup**. “Popup” is a term that HTML uses to include elements such as: **menus, trees, list boxes, grids and dialogs**.
Now, the element that has this ARIA attribute set on it, must control an element that has a **role equal to the value of the `aria-haspopup` attribute**.

In our case, because our Modal component has the role of **dialog** - the `aria-haspopup` attribute on the Modal’s trigger button will also have the value of **dialog**.

This is of course because **we know the role** of the element that our trigger is toggling, but if we didn't know this - we could use the "catch-all" value of **`aria-haspopup="true"`** - a more broad value but accessible nonetheless.

Combining all of that, we can finalize our Modal's initial HTML structure:

```html
<button aria-haspopup="dialog">Open modal</button>

<div class="modal-backdrop hidden">
  <div role="dialog" aria-modal="true" aria-labelledby="modal-title">
    <h3 id="modal-title">Modal title</h3>
    <div>This is some content inside the modal</div>
  </div>
</div>
```

Okay, that was pretty easy. Now let's get to the **toggle controls** and **focus management rules**.

Toggle controls

Our toggle controls logic has to do a lot of heavy lifting here, so we will start simple.
Let's first wire up our **trigger button to be able to open its corresponding Modal**.

There are multiple ways to do this, but we will do it by **giving the Modal an ID and adding a custom property `data-modal` to the trigger button with the value of the Modal's ID**.

```html
<button data-modal="modal-1" aria-haspopup="dialog">Open modal</button>

<div class="modal-backdrop hidden">
  <div id="modal-1" role="dialog" ...>
    <h3 id="modal-title">Modal title</h3>
    <div>This is some content inside the modal</div>
  </div>
</div>
```

Then, we can add a simple "click" listener to our trigger button that will open the Modal.

```javascript
const modalTrigger = document.querySelector('[data-modal]');

modalTrigger.addEventListener('click', () => toggleModal(modalTrigger, true));

function toggleModal(trigger, open) {
  const modalId = trigger.getAttribute('data-modal');
  const modal = document.getElementById(modalId);
  const backdrop = modal.closest('.modal-backdrop');

  backdrop.classList.toggle('hidden', !open);
}
```

Simple enough, now we need a way to close the Modal and we can actually do this a couple of ways - the most common of which are the following:

- Close the Modal using a **“Close” icon button inside the Modal itself**
- Close the Modal by **pressing the “Escape” key on the keyboard** (crucial for keyboard-only users)
- Close the Modal by **clicking outside of it (optional)**

Again, let’s start with the easy part and add the “Close” icon button logic.

Of course we will need to update our HTML structure to add the button and to actually **mark it as a button that can close the Modal** - we will **set a custom `data-dismiss` attribute** on it.

```html
<div class="modal-backdrop hidden">
  <div id="modal-1" role="dialog" ...>
    <div>
      <h3 id="modal-title">Modal title</h3>
      <button aria-label="Close Modal" data-dismiss>✕</button>
    <div>

    <div>This is some content inside the modal</div>
  </div>
</div>
```

```
JavaScript
const modalTrigger = document.querySelector('[data-modal]');

modalTrigger.addEventListener('click', () => toggleModal(modalTrigger, true));

const modalId = modalTrigger.getAttribute('data-modal');
const modal = document.getElementById(modalId);
const closeButtons = modal.querySelectorAll('[data-dismiss]');

closeButtons.forEach((closeButton) => {
  closeButton.addEventListener('click', (e) => {
    e.stopPropagation();
    toggleModal(modalTrigger, false);
  });
});

function toggleModal(trigger, state) {
  const modalId = trigger.getAttribute('data-modal');
  const modal = document.getElementById(modalId);
  const backdrop = modal.closest('.modal-backdrop');

  backdrop.classList.toggle('hidden', !state);
}
```

Next we need to tackle the **keyboard controls - closing the Modal using the Escape key**. We will do that by having our **global Window object** listen to any “Escape” key presses and if a Modal is open during that press - it will simply close it.

```
JavaScript
window.addEventListener('keydown', (e) => {
  if (e.key !== 'Escape') return;

  const openBackdrop = document.querySelector('.modal-backdrop:not(.hidden)');
  if (!openBackdrop) return;

  const modalId = openBackdrop.firstElementChild.id;
  const activeTrigger = document.querySelector(`[data-modal="${modalId}"]`);

  e.preventDefault();
  toggleModal(activeTrigger, false);
});
```

Awesome! Now, let’s make sure we **close the Modal when we click outside of it**. This is very simple for us because we already have that **Modal backdrop** element and we can add a click listener to it, **just make sure the click is not coming from the Modal itself**.

At this point, some code may be duplicated or reused in less-than-ideal ways. These examples intentionally keep modal state handling simple for demonstration purposes, even though more elegant solutions exist. Since **this course focuses on Web Accessibility**, we won’t spend much time optimizing data handling.

```javascript
const modalTrigger = document.querySelector('[data-modal]');

// Window Escape key listener here...

// Trigger click listener here...

const backdrop = getBackdrop(modalTrigger);

backdrop.addEventListener('click', (e) => {
  if (e.target !== backdrop) return;

  e.preventDefault();
  toggleModal(modalTrigger, false);
});

// Close buttons click listeners here...

function getBackdrop(trigger) {
  const modal = document.getElementById(trigger.getAttribute('data-modal'));
  return modal?.closest('.modal-backdrop');
}

function toggleModal(trigger, open) {
  const backdrop = getBackdrop(trigger);
  backdrop.classList.toggle('hidden', !open);
}
```

Okay, so now we can open and close our Modal, both using our mouse and keyboard. We also added a click-away listener that closes the Modal as a nice UX improvement.

Let’s now look at the **focus management** rules and techniques that Modals require.

Focus management

When talking about a Modal component, its focus management rules require us to do 2 things: manage the focus when we **open and close the Modal** and **trap the focus inside the Modal** until it's closed. Let's do the easier one first.

To cover the first rule, we must make sure that:

- When we're **opening** the Modal - we **move focus within the Modal container**
- When we're **closing** the Modal - we **move focus on the trigger element that opened it**

To be able to move focus within the Modal when opening it - we can add a **`tabindex="-1"`** on the Modal container and focus it from our code. Conversely, when closing the Modal, we can just focus the trigger button from our code, since we already have access to it.

```HTML
<div class="modal-backdrop hidden">
  <div id="modal-1" role="dialog" tabindex="-1" ... >
    <div>
      <h3 id="modal-title">Modal title</h3>
      <button aria-label="Close Modal" data-dismiss>✕</button>
    <div>
    <div>This is some content inside the modal</div>
  </div>
</div>
```

```JavaScript
function toggleModal(trigger, open) {
  const backdrop = getBackdrop(trigger);
  backdrop.classList.toggle('hidden', !open);

  if (open) {
    backdrop.firstElementChild?.focus();
  } else {
    trigger.focus();
  }
}
```

Now, let's tackle the final part of the Modal equation - **focus trapping**.

Focus trapping

Focus trapping is a focus management technique that essentially **keeps the ability to focus on elements within a certain section of the page**. This is done when certain UI elements (such as a Modal) require the user's attention and cannot let them move forward with interacting with the rest of the page until they complete a certain action or dismiss that UI element.

There are quite a few **third party libraries that provide this functionality out of the box** and we will talk about such solutions at a later point in time, but for now - let's do it ourselves. Effectively what we want to do is **keep the focus within the Modal** while it is open. This means that when the user is focused on the **last focusable element in the Modal** and presses "Tab" on their keyboard - the focus should move to the **first focusable element in the Modal** and vice versa.

We never let the focus out of the constraints of the Modal container until of course they either complete the action required in it, or just close it using the methods we covered a couple of paragraphs ago.

Now, to do this we need to have a reference (a list) of **all focusable elements within the Modal** and this can be a tricky thing to pull off. It's actually the trickiest thing to do in this entire functionality.
It really depends on how your Modal is intended to work. Do you definitely know that your Modal will have (for example) only buttons and links? Is your Modal populated with dynamic content, in which case you won't know what kind of focusable elements are in there?

There are different ways to tackle this problem, but for my money the easiest way to do it is to (and in this case I will make an exception to the third-party-libraries-bad attitude and just say) install the **"focusable"** library**.** It's extremely small - it only exports a **long string which is a list of all valid CSS selectors that can make for a focusable element**.

After we grab all of the focusable elements in our Modal, we just need to keep the user from putting focus outside of it, like so:

JavaScript

```javascript
modal.addEventListener('keydown', (e) => {
  if (e.key !== 'Tab') return;

  const focusables = getFocusables(modal);
  const firstFocusable = focusables[0];
  const lastFocusable = focusables[focusables.length - 1];

  if (e.shiftKey) { // Going backward
    if (e.target === firstFocusable) { // Focus the last focusable element
      e.preventDefault();
      lastFocusable.focus();
    }
  } else { // Going forward
    if (e.target === lastFocusable) { // Focus the first focusable element
      e.preventDefault();
      firstFocusable.focus();
    }
  }
});
```

And that’s it! That’s all that it takes to build a custom Modal from scratch. Maybe not so easy to start off with, but if you have the time and resources to devote - definitely worth it saving a couple of megabytes on a component library.

UX Improvements

Let’s talk about some improvements and tweaks we can make to our Modal, depending on the situation and use case.

Talking about the focus management, depending on the content of your Modal - if you know that you will have a **form** inside it as the main action to perform - it is probably a good idea to **focus the first input in the form when opening the Modal**. This makes it easier for users to start performing the action they wanted instead of needing 1 (or 2+ if we have a close button and some other focusable elements before the form) “Tab” presses to get to the actual form.

There can also be scenarios where **some of the components inside the Modal are quite interactive**.
For example there is a Menu or a Dropdown inside one of your Modals and **we need to make sure their keyboard controls don’t clash with the Modal’s keyboard controls**.

If the Dropdown inside the Modal uses the “Escape” key to close the popup dropdown element, we need to make sure that **that event doesn’t propagate and also activate the “Escape” key event that closes the Modal**.

We also need to take into account the fact that our page height might be bigger than the viewport - meaning the main content is scrollable - in which case we want to **block the scrollability of the main content while a Modal is open**. This is because the Modal, again, is an element that provides an action that the user must perform and cannot interact with anything else until they do (or close the Modal).

Modern CSS makes this very easy to implement, as we can just **hide the overflow of the <body> element** when it has a Modal that’s open.

```css
CSS
body:has(.modal-backdrop:not(.hidden)) {
  overflow: hidden;
}
```

There are all kinds of tricky edge case scenarios that we need to make sure work correctly across our entire application regardless of the content inside the Modal. This is admittedly very tricky to do, so maybe there’s a better solution?

Note: Check out the full code to see the **implementation of the `getFocusables()` function**, as well as the full implementation of the Modal component.
Feel free to also check out the more in-depth look on the Authoring Practices Guide for constructing a Modal component.

Modern implementation

If the above code seemed a little too overwhelming, it’s because it is. We all know that developers don’t and won’t get much time to devote to such detailed implementations of every single piece of UI from scratch.

If we do - that’s great! But more often than not, this kind of component is just too widely used and too non-important to the actual product we are building that we will just **reach for a component library** to do the heavy lifting for us.

It's admittedly a lot of code to manage ourselves, so it seems like the logical solution in many cases and I have not been an exception to this situation as well.

Luckily for us, modern HTML allows us to have our cake and eat it too - meaning we can skip on both implementing all that code ourselves while also not needing to install a third party library and bloat our code for this type of use case.

I'm talking of course about the shiny new <dialog> element. If you haven't heard about it or used it before, it's HTML's native solution to Modals and Dialogs and all other similar components.
This element makes it as easy as just **writing the simple markup (which we'll see right now) and hooking up the trigger button to it**.

The markup itself is very simple, because we don't need to worry about many of the previously manually setup roles and ARIA attributes. The only thing we'll actually need to keep is the \`aria-labelledby\` attribute, to connect it with the title, that's it.

HTML
```html
<button data-native-modal="native-modal" aria-haspopup="dialog">
  Open native modal
</button>

<dialog id="native-modal" aria-labelledby="native-modal-title">
  <div>
    <h3 id="native-modal-title">Modal title</h3>
    <button aria-label="Close Modal" data-dismiss>✕</button>
  </div>

  <div>This is some content inside the modal</div>
</dialog>
```

Notice we still **keep a custom \`data-native-modal\` and \`data-dismiss\` attribute** to each button respectively, so that we can **wire them up to work with the Modal they are controlling**.
In the case that you have multiple Modals in your application (and you most likely will), it is important to have a way to connect each trigger button to its corresponding Modal.

Be that as it may, our JavaScript is now much cleaner, as we only need to call **one function to open the Modal** and then **one other function to close the Modal**. That's it!

```javascript
const modalTrigger = document.querySelector('[data-native-modal]');

const modalId = trigger.getAttribute('data-native-modal');
const nativeModal = document.getElementById(modalId);
const closeButtons = nativeModal.querySelectorAll('[data-dismiss]');

closeButtons.forEach((closeButton) => {
  closeButton.addEventListener('click', () => nativeModal.close());
});

modalTrigger.addEventListener('click', () => nativeModal.showModal());
```

We need very little CSS to actually place our Modal at the center, too!

```css
dialog {
  margin: auto;
}
```

That's it, everything else is just additional styling we would add to the Modal container box.

The native <dialog> element gives us a lot of the functionality we had to implement ourselves, for free. We have **automatic focus trapping within the Modal**, we have **correct focus management when opening/closing the Modal**.
In fact, if we add the **`autofocus`** attribute to any focusable element inside the Modal - that element will automatically be focused for us first when opening the Modal - we don't have to implement our own edge cases there as well!

We get that **backdrop** element for free as well, as it is a **pseudoelement** of the <dialog> element. As such, we can style it as we want with the simple **`dialog::backdrop`** CSS selector.

There are many more interesting features that the <dialog> element gives us, like **configuring how we want to be able to dismiss it** and it even **gives us the ability to treat it as a Modal or a Non-Modal Dialog - depending on how we use it**.

If we open the `dialog` using the **`.showModal()`** function (like we did in our example) - this element will behave like a Modal Dialog - a.k.a the regular Modal component we built out from scratch where it **blocks the rest of the UI until it is dismissed**.

If on the other hand we open it using the **`show()`** function - this element will behave like a Non-Modal Dialog - a.k.a the Dialog component we mentioned at the start of this lesson, meaning it **will not block the rest of the UI** and **it will not have a backdrop**.

Oh and another thing, as if all this wasn't enough, with the native <dialog> element - we don't need to worry about **z-index positioning**. This is because the <dialog> element, when open, gets placed in a **special layer** called the **`#top-layer`** which is **guaranteed to always be positioned in front of every other element on your page**.

This is an amazing advancement in modern HTML and I highly encourage you to look up all of its glorious features on the MDN documentation on the <dialog> element.

Conclusion

Today we saw how to **build out some of the most common UI components** and talked about all the minor details that go along with them, making them accessible to use.
Try to go over these implementations on your own time, play around with the code and think about how you can start incorporating the accessibility features we talked about today - in your own codebases.

Feel free to reach out to me at daniel.shijakovski@testdevlab.com if you have any questions or just want to talk about building out accessible UI - I'd love to hear from you!