TDL SCHOOL

Web A11y for Developers

Day 2: Components

TDL SCHOOL

Agenda

Day 1: Introduction

Day 2: Components

Day 3: Accessible Forms

Day 4: Best Practices

Building accessible components

We will look at some of the **most frequently used UI components** in almost every website or application today.

Each component presents its own accessibility challenges:

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

Web Accessibility Resources

WAI's ARIA Authoring Practices Guide

One of the best resources to learn about Web Accessibility.

We will focus on their **Patterns** resource - how to build common UI components using known accessibility and structural patterns.

Accordion

Composed of 2 parts: **the header** and **the panel**.

Accordion header

Interactable element (ex. button) that’s used for toggling the Accordion’s panel.

Accordion panel

Element that holds the detailed information which we are toggling using the header.
When the Accordion is closed - this content should be **invisible to the user and assistive technologies**.

Accordion: Header

We need to use proper ARIA attributes to **convey information** about:

- **Which element it is controlling** (toggling)
- **What's the state of that toggle** (opened/closed)

The toggling relationship information is conveyed using the ARIA attribute `**aria-controls**`.

The toggled state information is conveyed using `**aria-expanded**`.

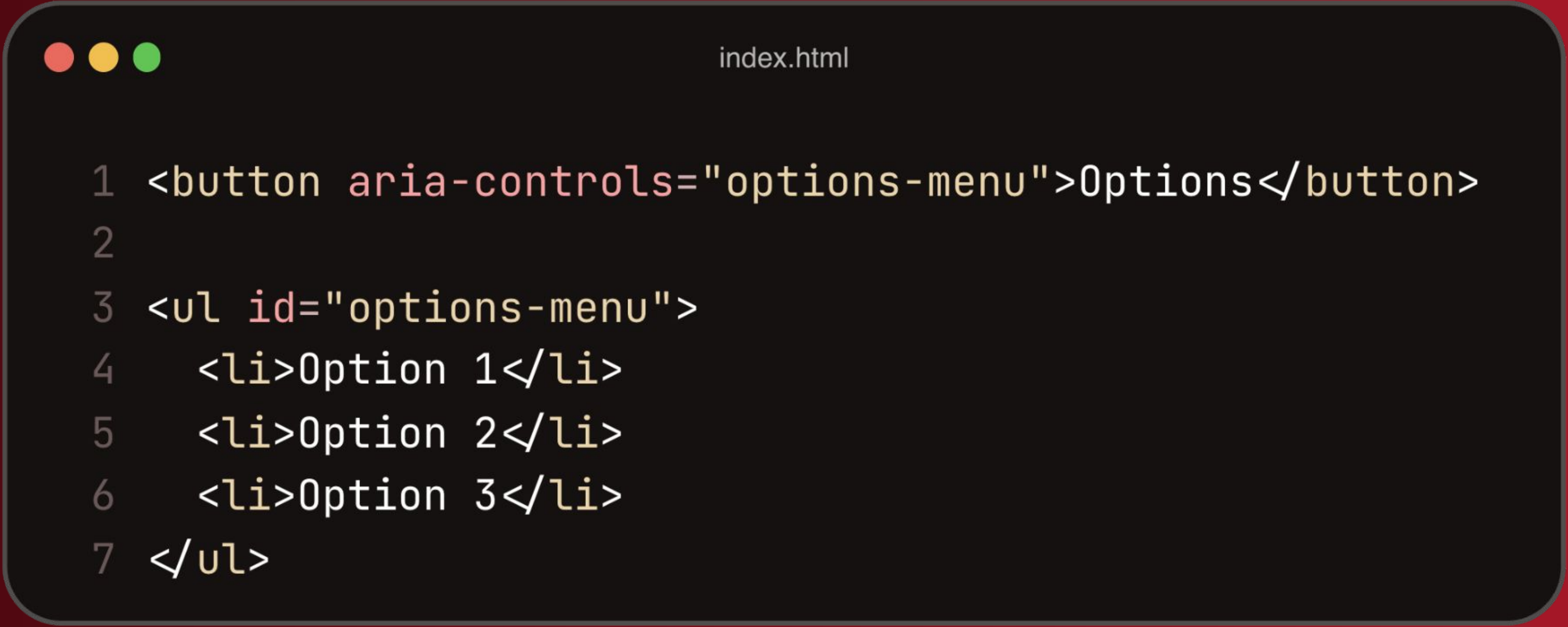
Accordion: Header

aria-controls

Used to convey a relationship between 2 elements - where **one element controls the presence of another element.**

Usage

1. Add the `**aria-controls**` attribute to the **controlling element**
2. Set the **value of the attribute** to the **ID of the element that's being controlled**


index.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>
```

Accordion: Header

aria-expanded

Used to convey **information about state**. It is used on **elements that control other elements that can expand/collapse**.

Usage

1. Add the `**aria-expanded**` attribute to the **controlling element**
2. Set the **value of the attribute** to either “**true**” or “**false**”, depending on if the controlled element is expanded/collapsed
3. Dynamically update the attribute’s value

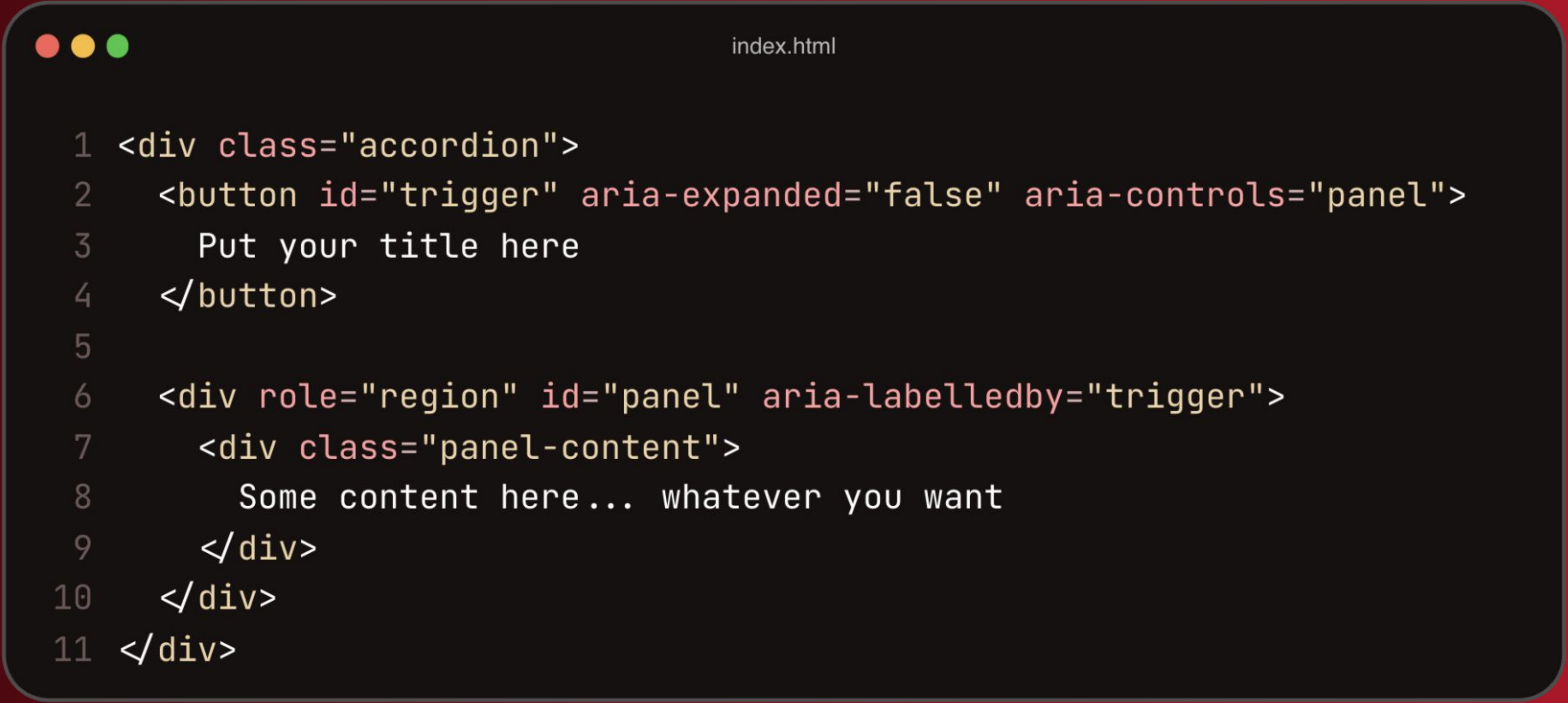
Accordion: Panel

We need to give it the **role** of **“region”** to make it a **landmark** that screen readers can navigate to.

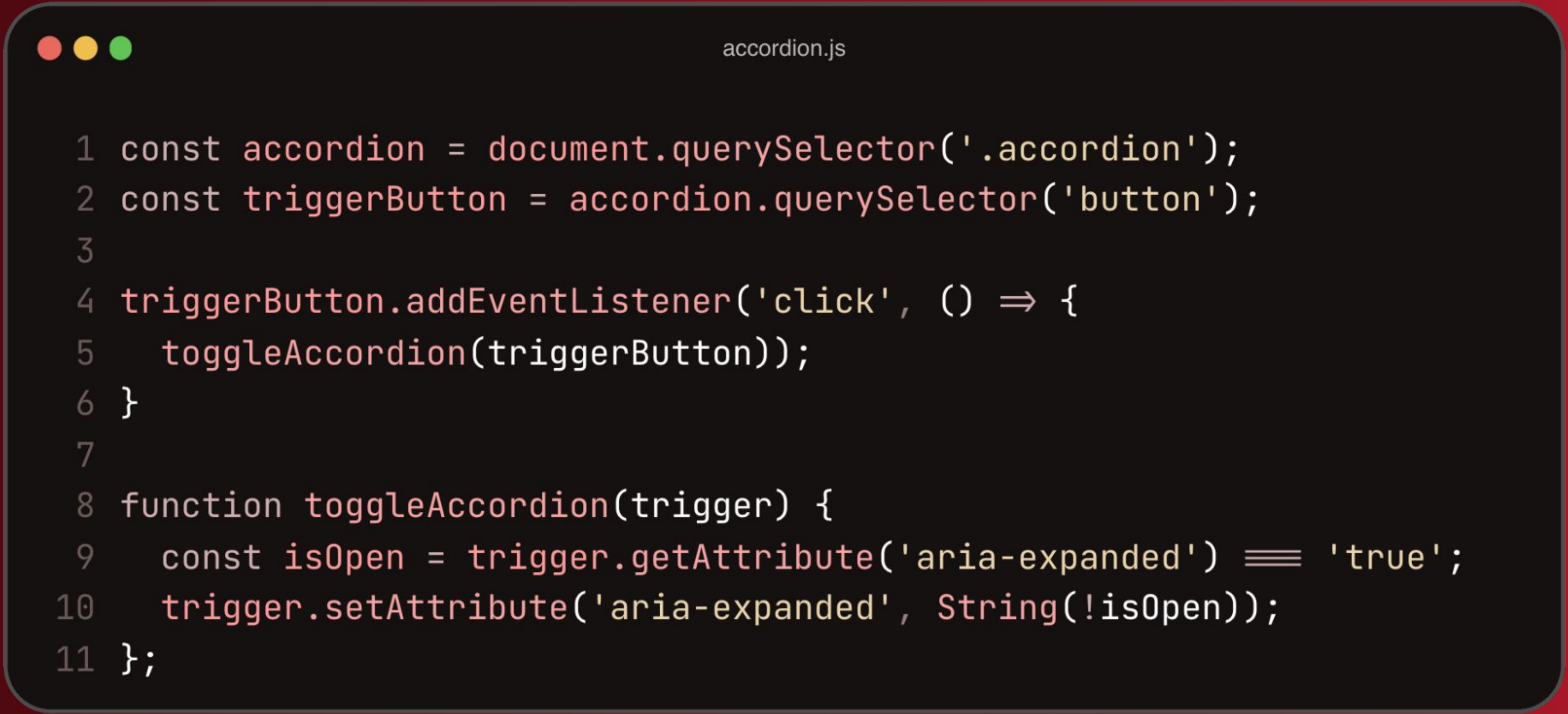
We then need to **label it according to its trigger** (Header) **button.**

Good practice is to use a **single source of truth** for the Accordion’s expanded/collapsed state.

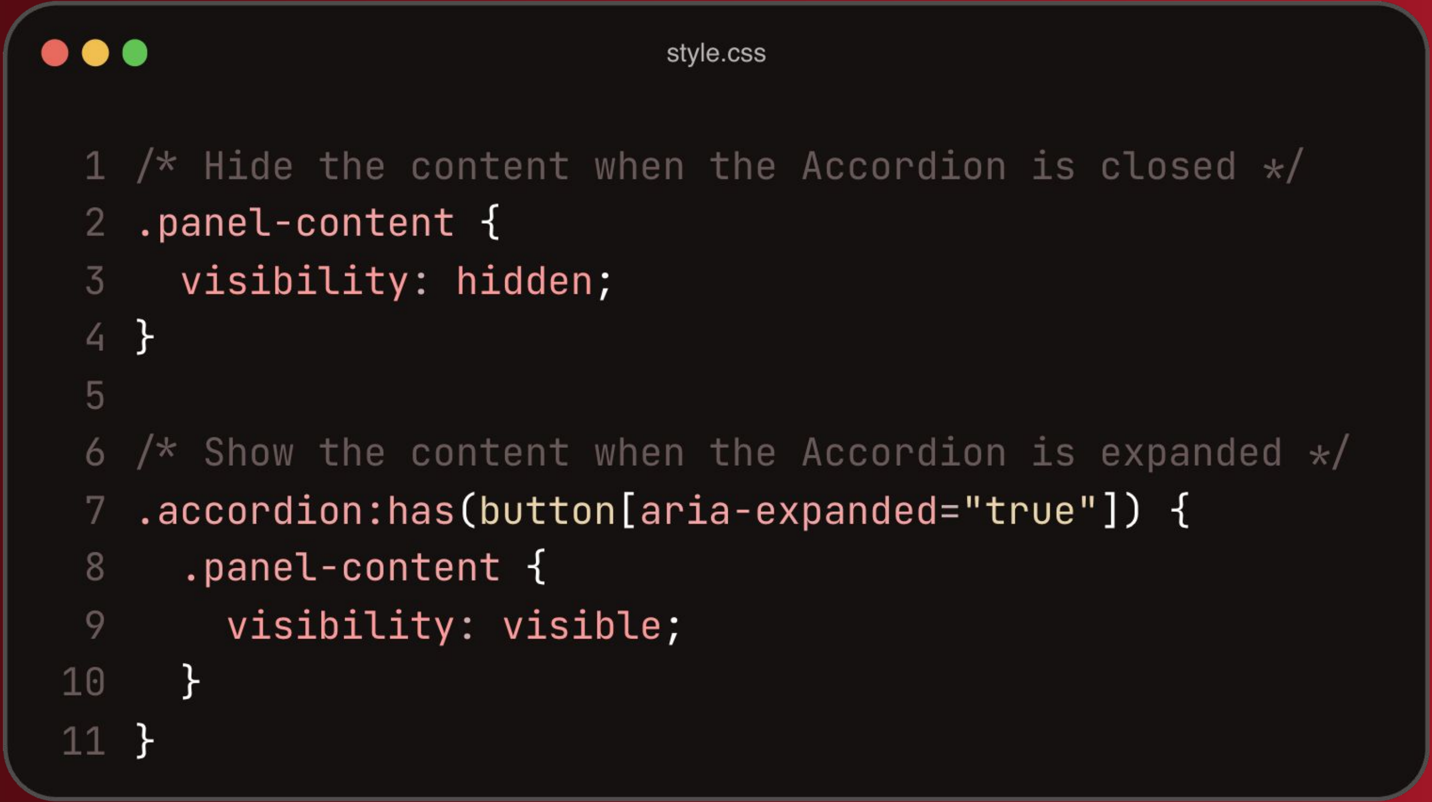
We will use the trigger (Header) button’s `aria-expanded` state.



```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>
```



```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));
};
```


style.css

```css
/* Hide the content when the Accordion is closed */
.panel-content {
  visibility: hidden;
}

/* Show the content when the Accordion is expanded */
.accordion:has(button[aria-expanded="true"]) {
  .panel-content {
    visibility: visible;
  }
}
```

Demo Time!

Building an Accordion component

Accordion

Disclaimer: The full code is deliberately omitted on the screenshots for brevity.

Check out the full implementation on the repo.

Check out the more in-depth look on the APG Patterns for an Accordion component.

Tabs

Composed of 2 parts: the **Tablist** and the **Tab panels**.

Tablist

The list that contains the actual Tab buttons.

Tab panels

Individual pieces of content that correspond to a given Tab.

Tabs: Tablist

It is itself composed out of 2 parts: The **Tabs container** and the individual **Tab buttons**.

The Tabs container should have the **role** of “**tablist**”, while each individual Tab button should have the **role** of “**tab**”.

We need to connect each Tab button to the Tab Panel that it is associated with - using `` `**aria-controls**` ``.

We also need to convey information about which Tab is currently selected - using `` `**aria-selected**` ``.

Tabs: Tablist

aria-selected

Used to convey information on **which element is selected out of a group of elements** (whatever “selected” means for that group).

Only one Tab can be selected at a time.

Usage

1. Set the `**aria-selected**` attribute to **“true” on the currently active Tab**
2. Set the `**aria-selected**` attribute to **“false” on the other Tabs**

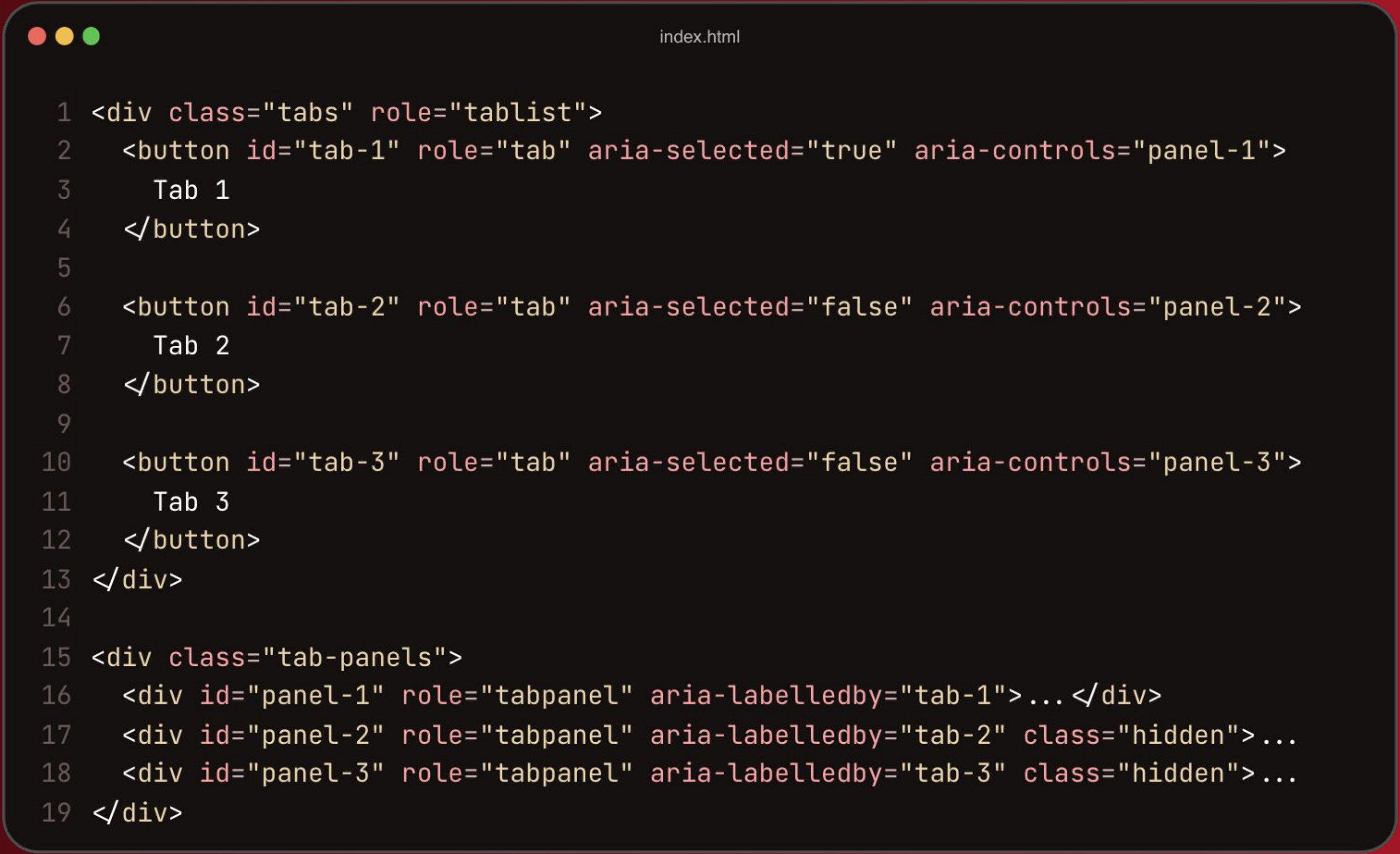
Tabs: Tab panels

Each panel needs to have the **role** of **“tabpanel”**.

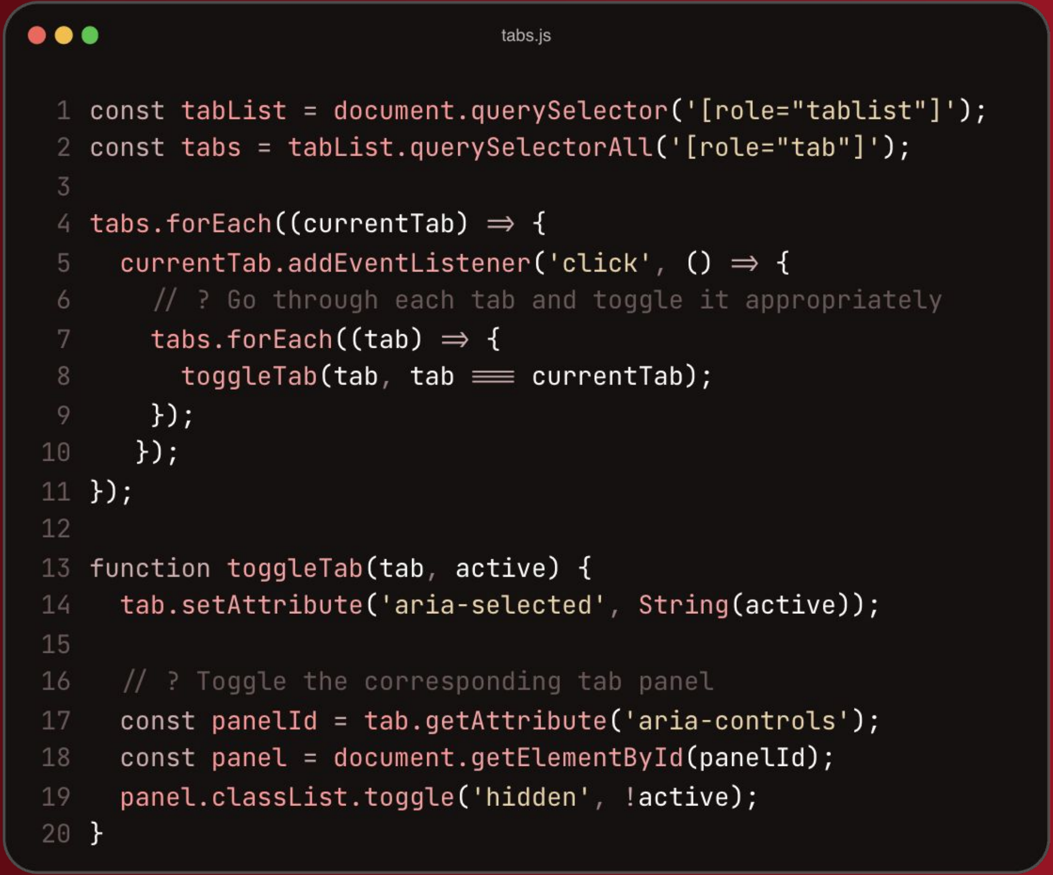
We also need to **label each panel according to their Tab button**. We can do this using `aria-labelledby`.

How we show/hide the panels is then up to us:

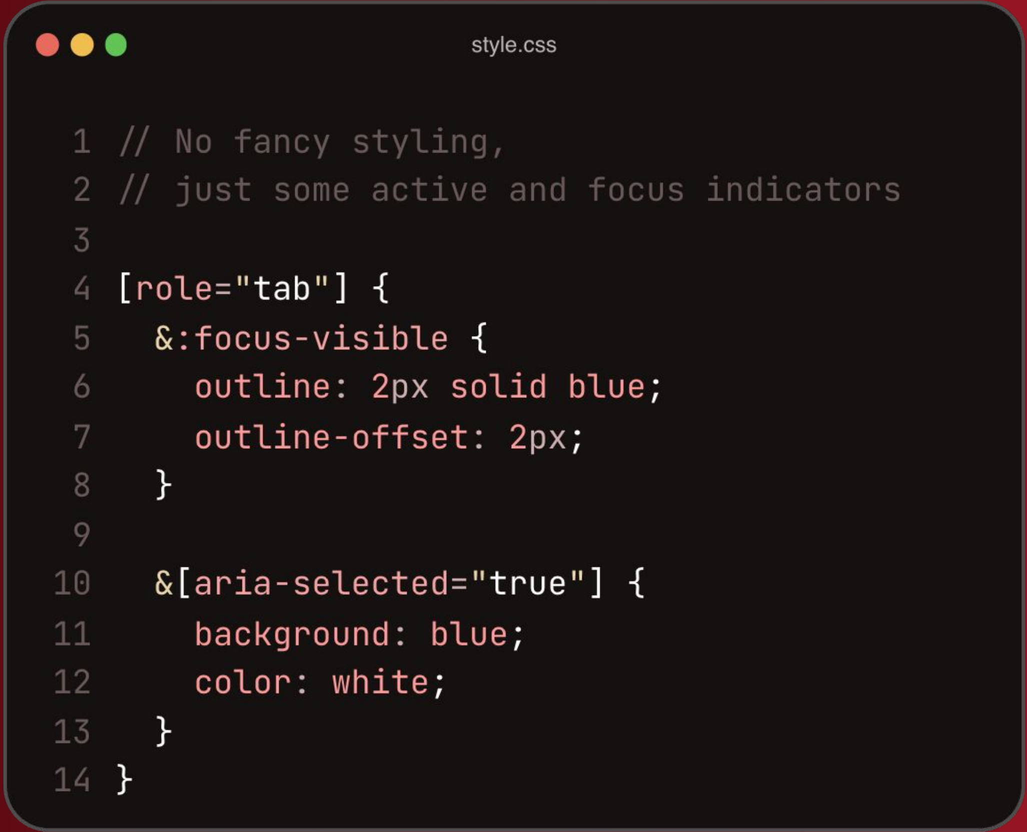
- Simply toggle their **`display`** property to **`none/block`**
- Use **`aria-hidden=”true”`** **on inactive panels** to keep them in the DOM (ex. For animation purposes)


index.html

```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>
```



```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);
}
```



```css
// No fancy styling,
// just some active and focus indicators

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

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

Demo Time!

Building a Tabs component

Tabs: Keyboard controls

Q: What would happen if we had 10+ Tabs and **keyboard users** wanted to **focus on a link** that's in the **first Tab panel's content**?

A: They would need to **"Tab" through all 10+ Tabs** to get to it.

This is a **usability nightmare** and we need to do something about it.

Enter, "**roving tabindex**".

Tabs: Keyboard controls

Roving tabindex

Focus management technique – makes sure **only 1 element within a group of elements is part of the focus order**.

How does it work?

1. Set `**tabindex=”0”**` on the element we want to be focusable
2. Set `**tabindex=”-1”**` on the rest of the elements in the group

We want to set `tabindex=”0”` on the **currently active Tab button** and `tabindex=”-1’` on the rest of them.

Tabs: Keyboard controls

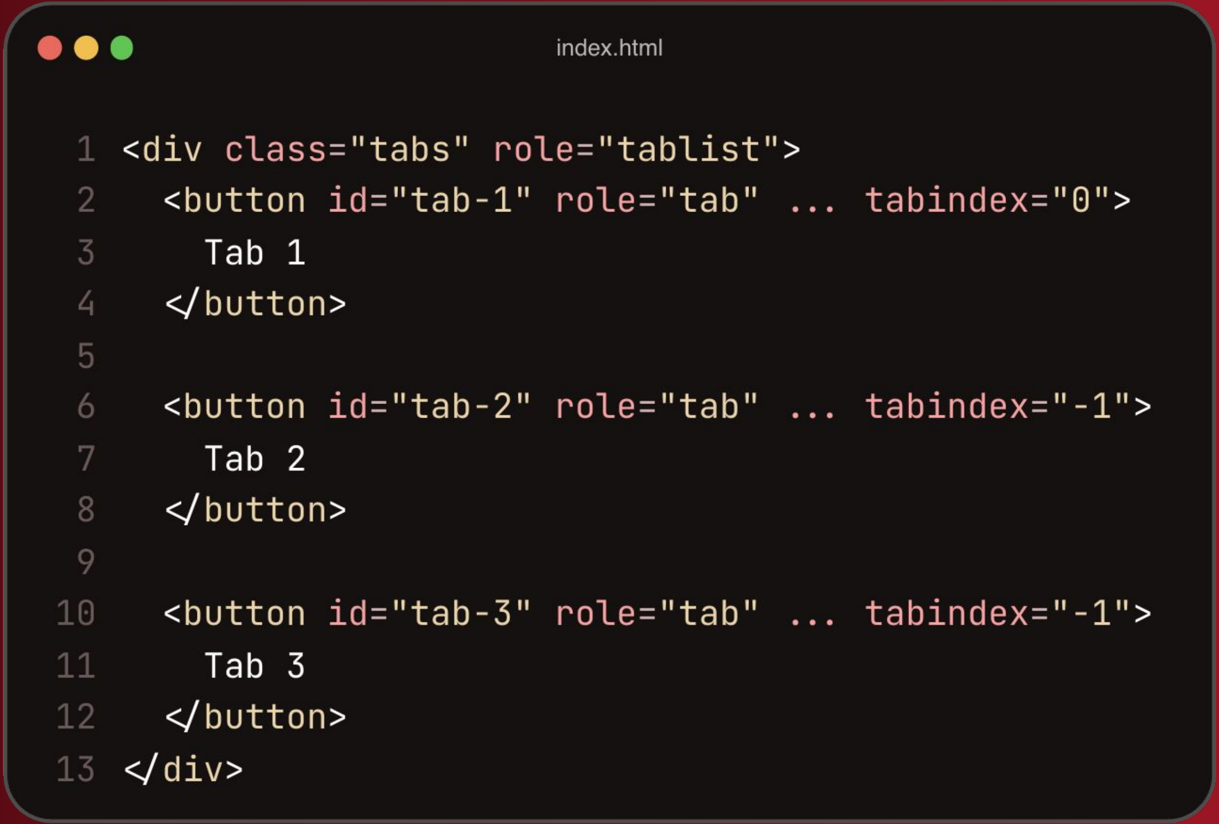
Roving tabindex

By moving the `tabindex` property - we make sure that pressing "Tab" while being focused on the currently active Tab button - will move the focus **inside the corresponding Tab panel's content**.

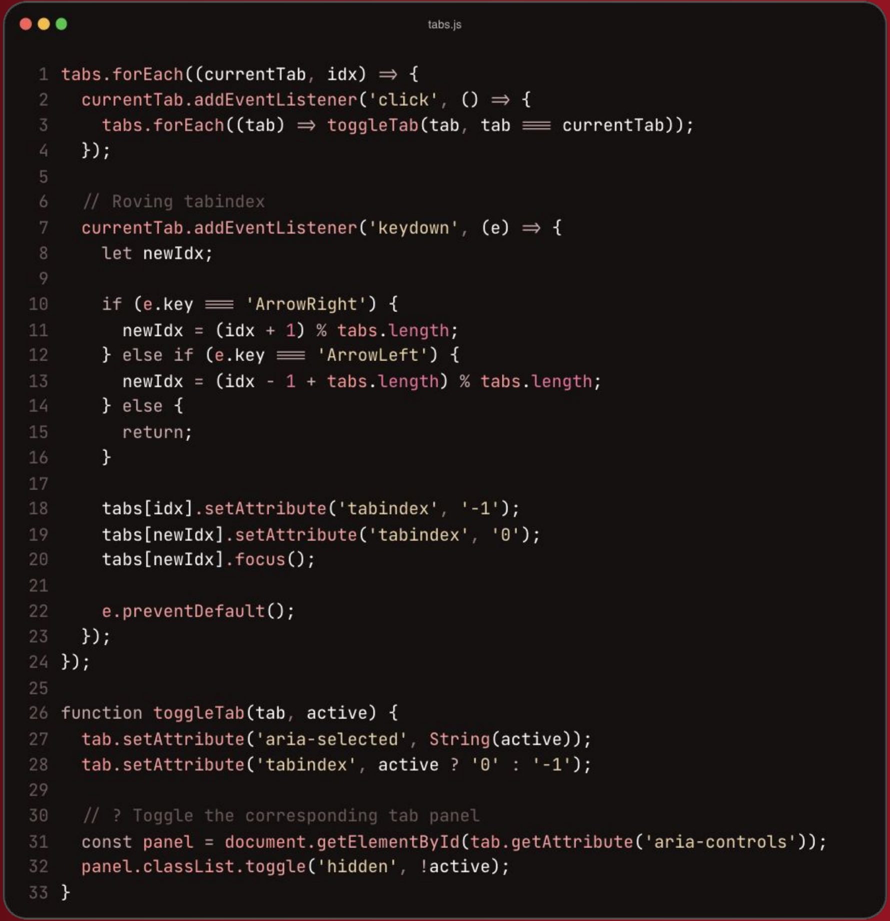
Now, when we have 10+ (or even 50+) Tabs, users only need to press "Tab" once and the focus will move inside the active Tab's content.

Q: But how do we then move through Tab buttons?

A: Using the keyboard's Arrow keys!



```html
<div class="tabs" role="tablist">
  <button id="tab-1" role="tab" ... tabindex="0">
    Tab 1
  </button>

  <button id="tab-2" role="tab" ... tabindex="-1">
    Tab 2
  </button>

  <button id="tab-3" role="tab" ... tabindex="-1">
    Tab 3
  </button>
</div>
```



```js
tabs.forEach((currentTab, idx) => {
  currentTab.addEventListener('click', () => {
    tabs.forEach((tab) => toggleTab(tab, tab === currentTab));
  });

  // Roving tabindex
  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);
}
```

Demo Time!

Tabs: Keyboard controls

Tabs

Disclaimer: The full code is deliberately omitted on the screenshots for brevity.

Check out the full implementation on the repo.

Check out the more in-depth look on the APG Patterns for the Tabs component.

Break Time!

Take a deep breath and relax

Modal

Terminology

There are lots of terms that this component has been called: Dialog, Modal, Overlay, Popup, Popover etc.

Check out this amazing article on the terminology debate around these kinds of components.

For the sake of this course, we will use 2 terms:
Modal and **Dialog**.

Modal vs Dialog

Modal

Appears **in front of the rest of the UI** and makes the rest of the UI **unreachable** (not interactable).

Examples: Filling in a form, confirming a deletion of an important resource etc.

Technical definition: **Modal Dialog**.
The “Modal” part means that it blocks the rest of the UI.

Modal vs Dialog

Dialog

Appears **in front of the rest of the UI** but doesn’t block the rest of the UI. Users can still interact with the entire page.

Examples: Pop-up chats with AI assistants, find-and-replace windows in our IDEs etc.

Technical definition: **Non-Modal Dialog**.

Modal

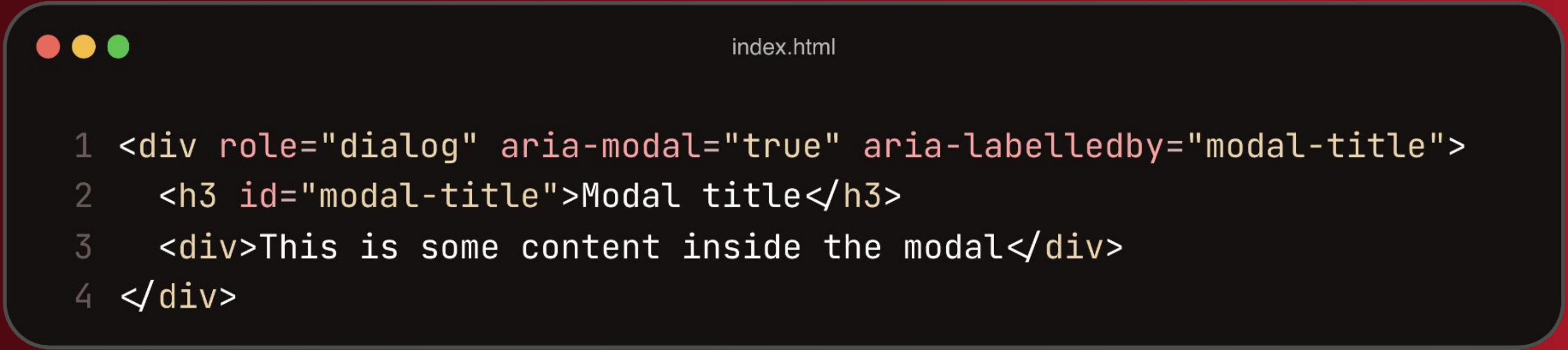
HTML Structure

The Modal container needs a **role** of “**dialog**”.

Next, we need to set the ARIA attribute `**aria-modal**` to “**true**”.

This tells assistive technologies that our component should block the rest of the UI behind it, while it is open.

Then we need to **label the Modal container**. This is usually done by connecting it to the **title (heading) element** inside the Modal.


index.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>
```

Modal

HTML Structure

We will also need a **backdrop element** to block out the rest of the UI (Ex. Make it darker or blur it).

Finally, we need a **button that will open the Modal**. We call this the **Modal trigger**.

To properly connect the trigger with the Modal, we need to use the ARIA attribute `**aria-haspopup**`.

Modal

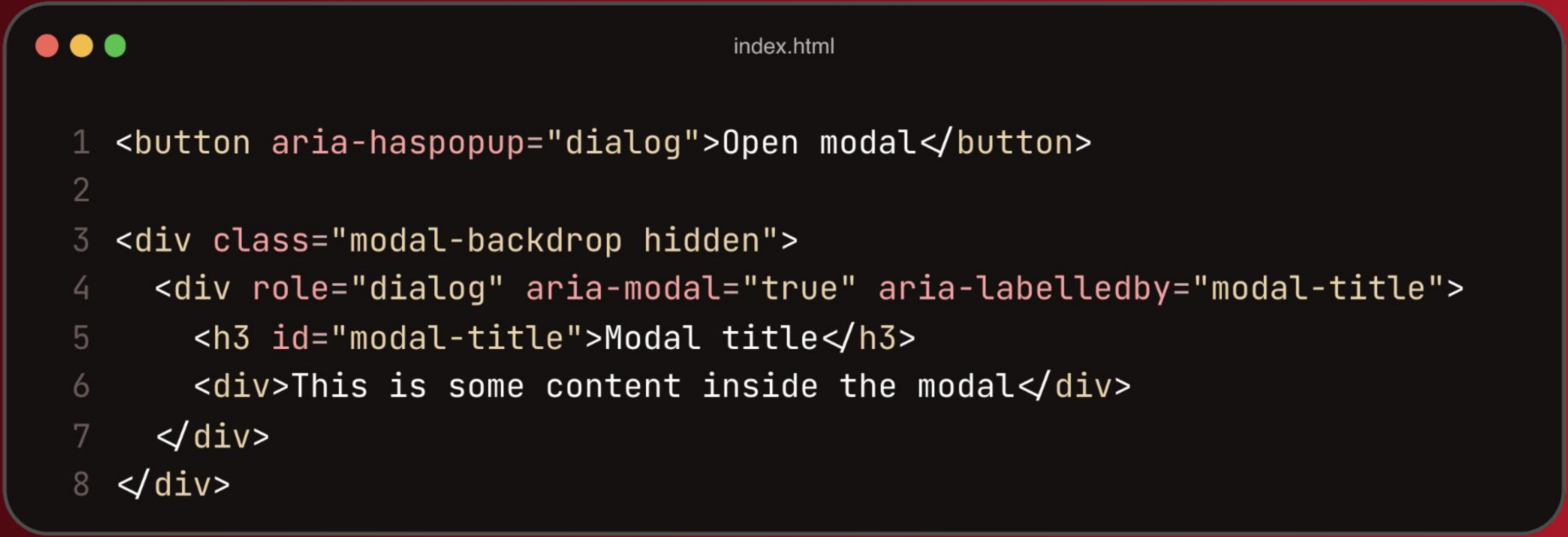
aria-haspopup

Used on elements that **open some kind of “popup”**.

Popup - term HTML uses to refer to elements such as: Menus, Trees, Listboxes, Grids and Dialogs.

The **value of `aria-haspopup`** must match the **role of the popup element** being opened.

If we don’t know the role of that popup element, we can use **“true” as a fallback value**.


index.html

```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>
```

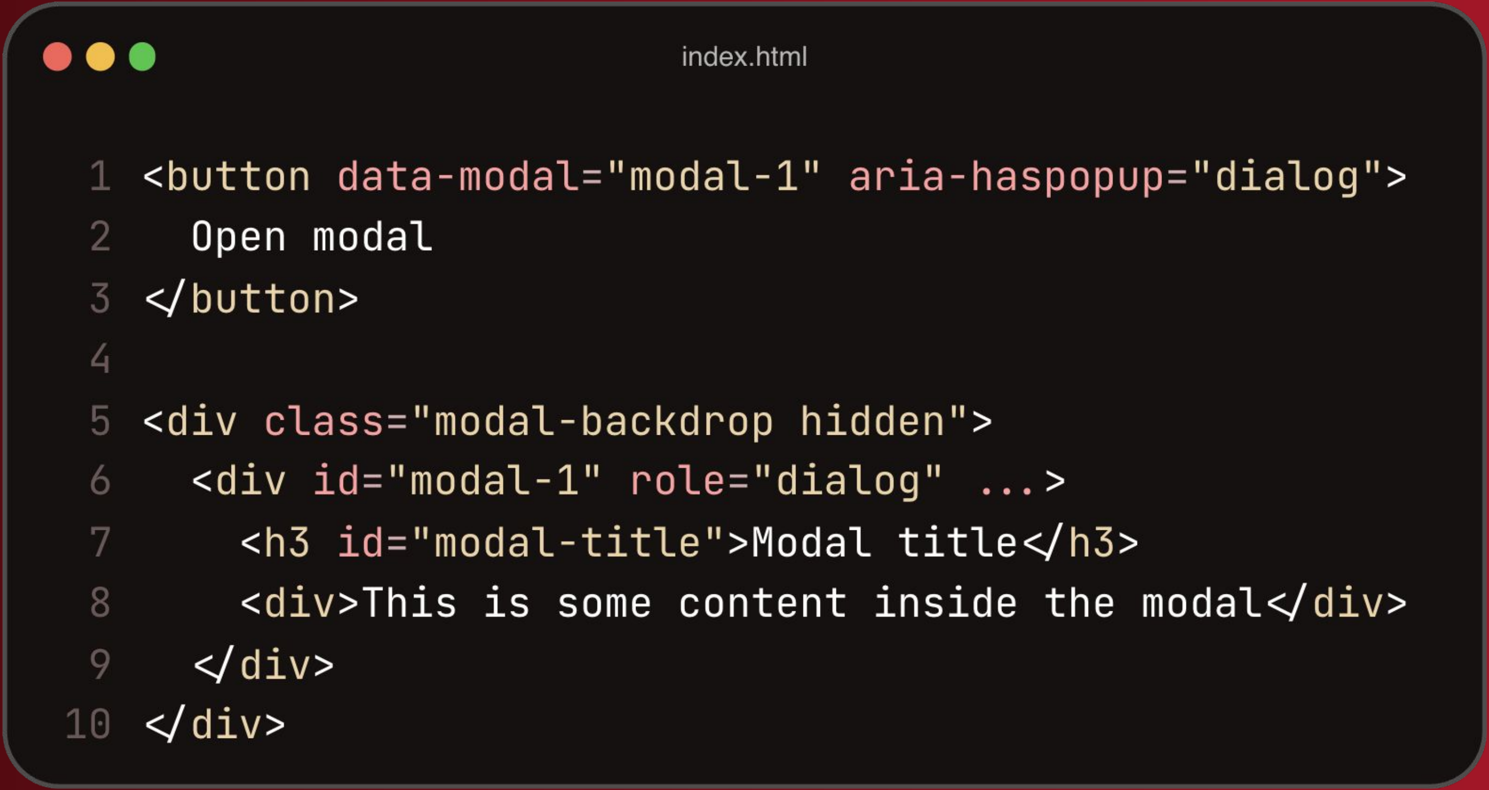
Modal

Toggle controls - Opening the Modal

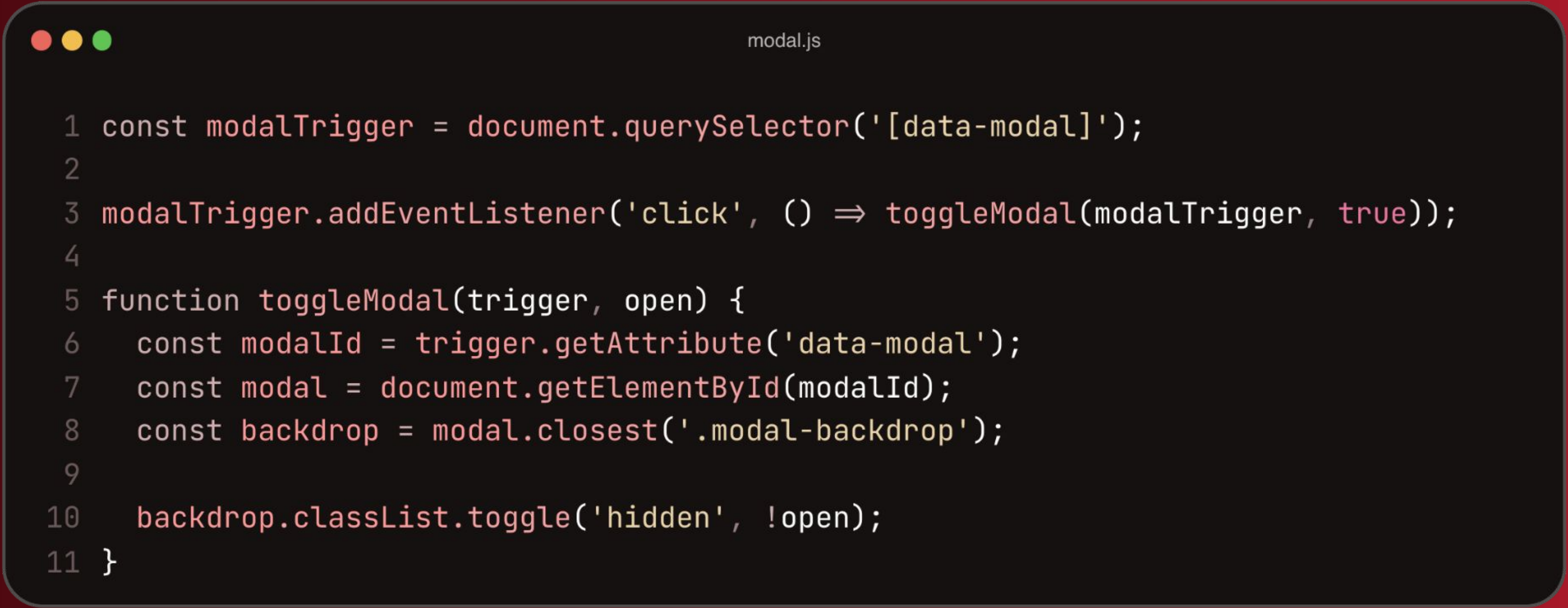
We need to **wire up** the trigger button to be able to open its corresponding Modal.

To do this, we will use a **custom data attribute `data-modal`** on the trigger button.

We will set the **ID of the Modal** container as the **value of that custom data attribute**.
Then we can set up our JavaScript logic for opening the Modal.



```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>
```


modal.js

```
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);
}
```

Demo Time!

Building a Modal component

Modal

Toggle controls - Closing the Modal

We need to make sure that out Modal can be closed a couple of different ways:

1. Using a “Close” icon button inside the Modal itself
2. Pressing the “Escape” key on the keyboard
3. Clicking outside of the Modal (optional)

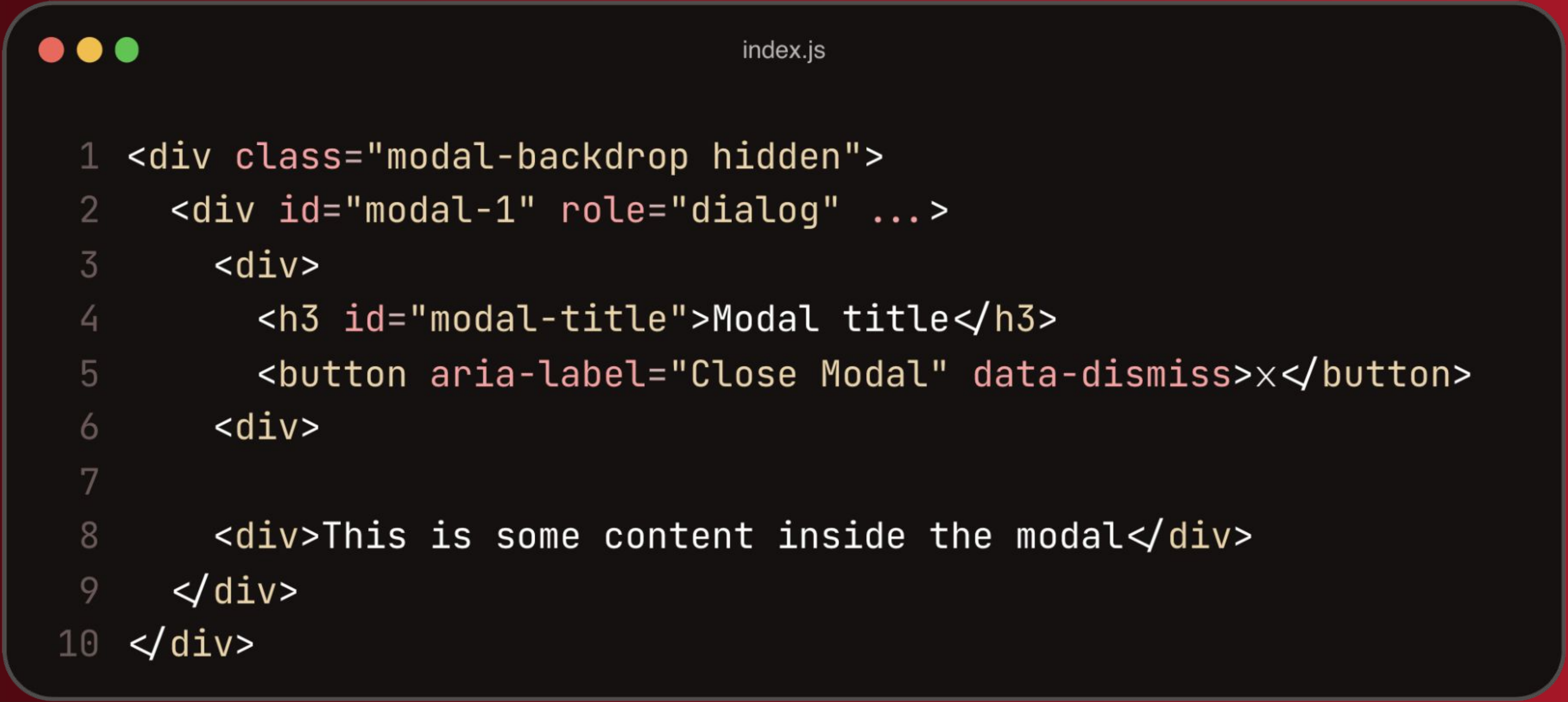
Modal

Toggle controls - Closing the Modal

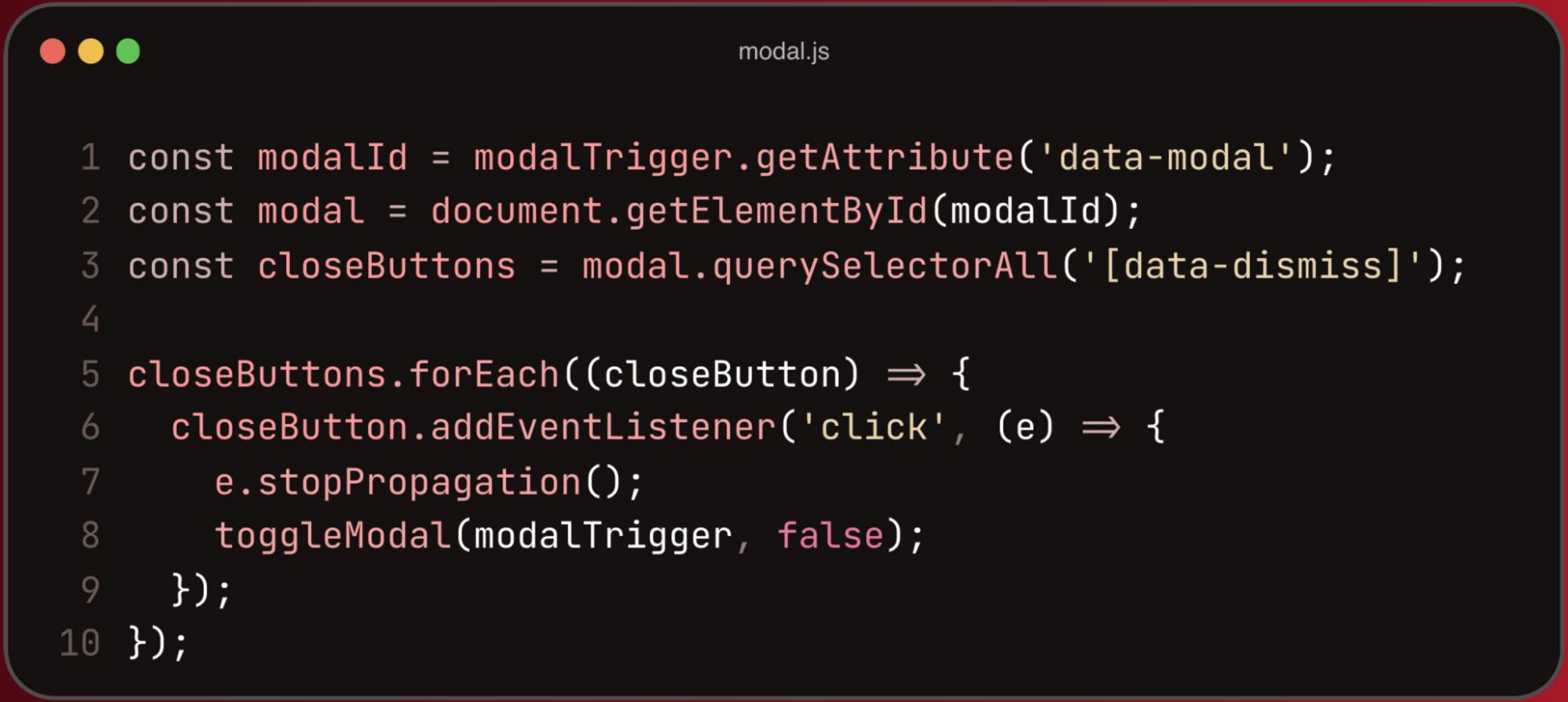
To close the Modal using a “Close” icon button, we first need to update our HTML structure and add that button.

Second, we need a way to be able to **reference all buttons that can close the Modal from within it**.

We will use a custom data attribute called `**data-dismiss**` for this.


index.js

```
<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>x</button>
    <div>

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



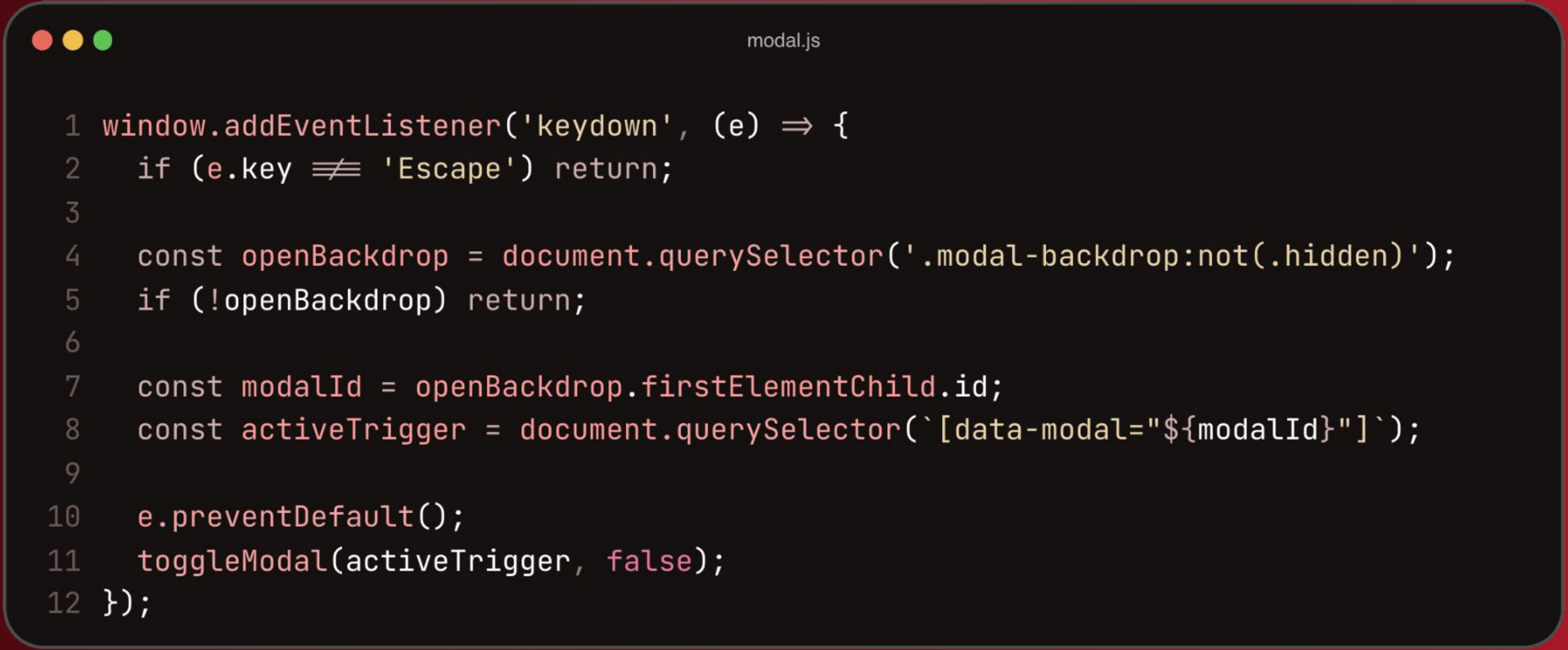
```javascript
const modalId = modalTrigger.getAttribute('data-modal');
const modal = document.getElementById(modalId);
const closeButtons = modal.querySelectorAll('[data-dismiss]');

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

Modal

Toggle controls – Closing the Modal

To close the Modal by pressing the “Escape” key on the keyboard, we just need to:

1. Add a **“keydown” event listener** to the **global Window object**
2. Check to see if the “Escape” key has been pressed **while a Modal is open** – if it is, close it



modal.js

```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);
});
```

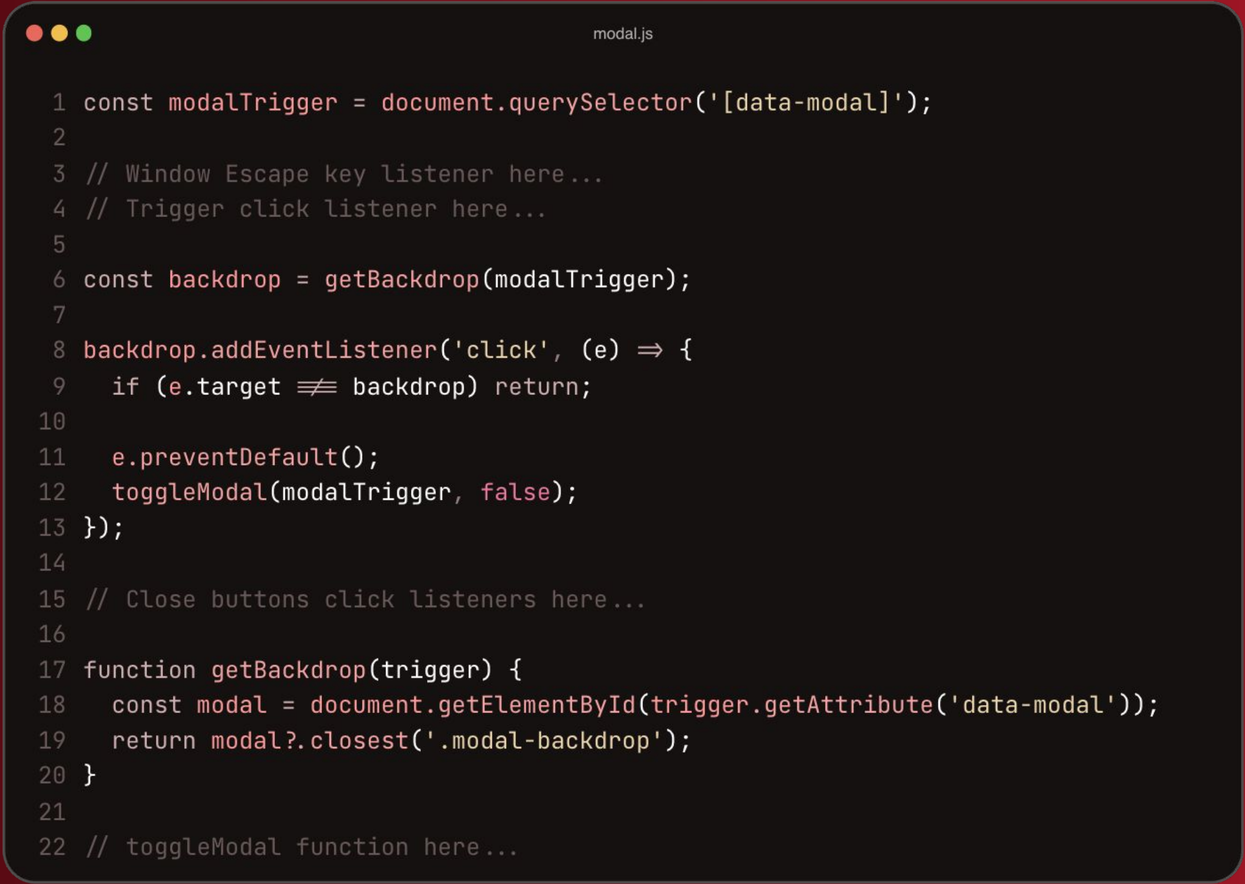
Modal

Toggle controls – Closing the Modal

To close the Modal by clicking outside of it, we can just keep track of **“click” events on its backdrop**.

If the user clicked on **exactly the backdrop** and not the Modal inside it, we can close the Modal.

We can reorganize our code as well, to not have as much code duplication while performing the toggling.


modal.js

```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');
}

// toggleModal function here...
```

Demo Time!

Modal: Closing controls

Modal

Focus Management

When toggling a Modal, we need to manage the focus such that:

- When **opening the Modal**, move focus within the **Modal container**
- When **closing the Modal**, move focus on the **trigger button**

We also need to make sure that we implement **focus trapping.**

Focus trapping is a focus management technique that **keeps the focus from going out of the Modal container.**

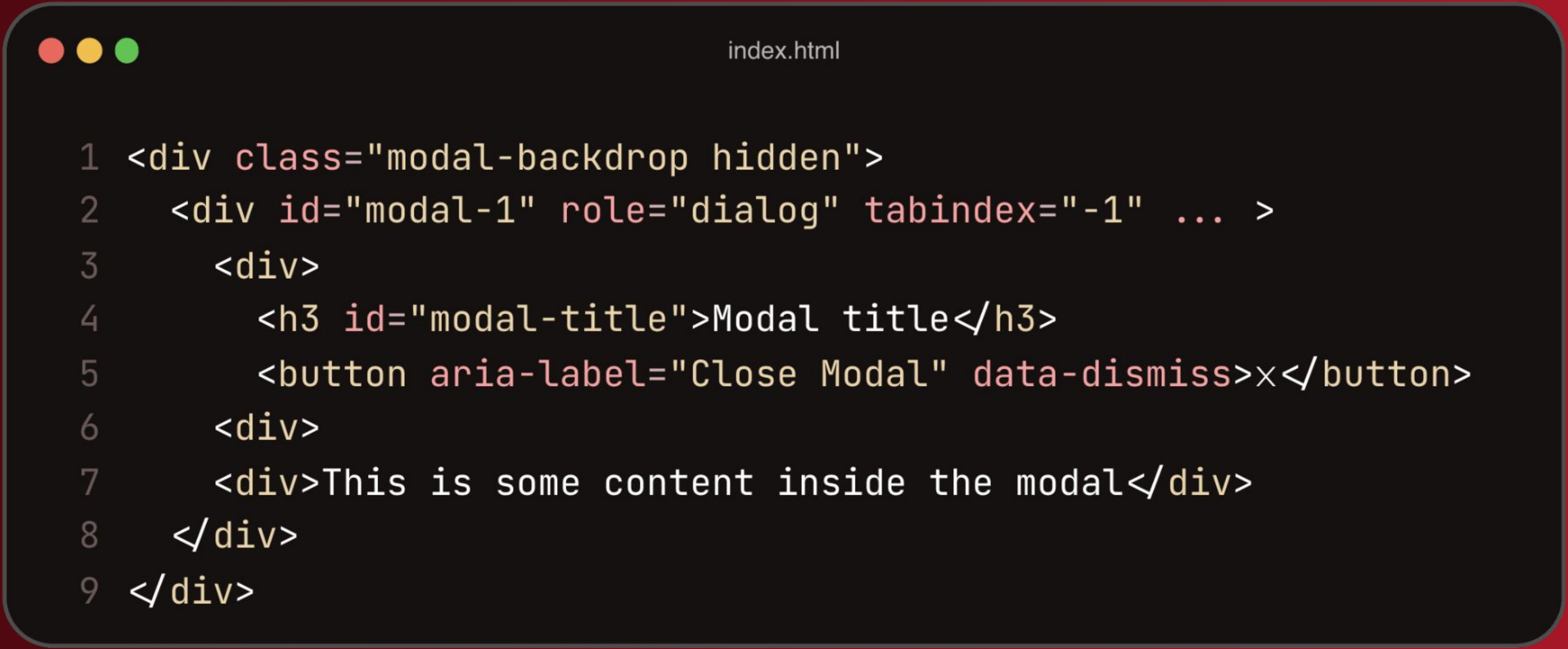
Modal

Focus Management - Toggling the Modal

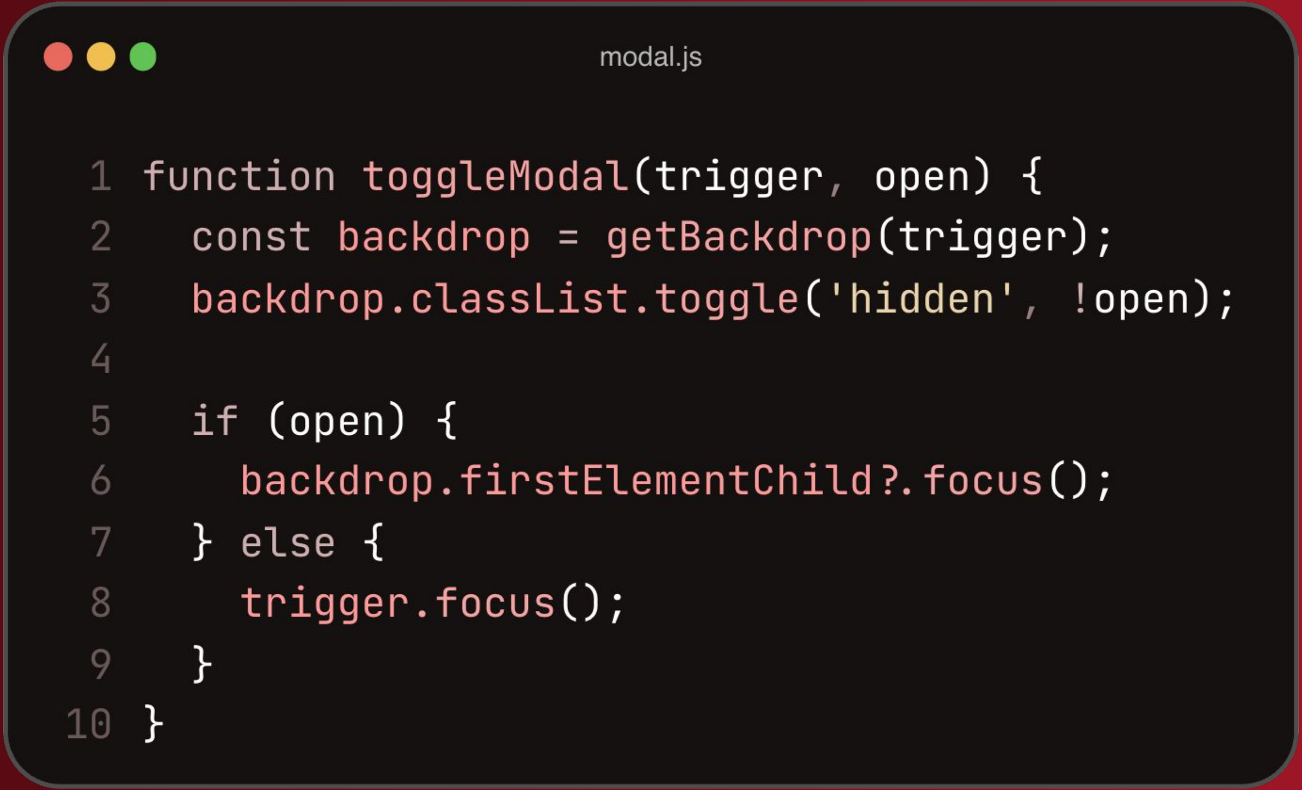
This is a good use case for **`tabindex=”-1”`**.

We can add this attribute to our Modal and programmatically focus it when we open it.

When we close the Modal, we can just move focus on the trigger button instead.



```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>x</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();
  }
}
```

Modal

Focus Management - Focus Trapping

Our Modal needs to **block the rest of the UI** while it is open.

This means that while a Modal is open - **users should not be able to focus** on any elements on the page other than the ones inside the Modal.

Focus Trapping is the technique we will implement to make this functionality happen.

Note: There are third party libraries that provide this as well.

Modal

Focus Management - Focus Trapping

What we need to do is:

1. Get a reference to all focusable elements inside the Modal
2. Keep track of the “Tab” press event on the Modal

When users are focused on the **last focusable element** and pressing “Tab” - move the focus to the **first focusable element**.

When users are focused on the **first focusable element** and pressing “Tab” while going backwards (pressing “Shift” as well),we move the focus to the **last focusable element**.

Modal

Focus Management - Focus Trapping

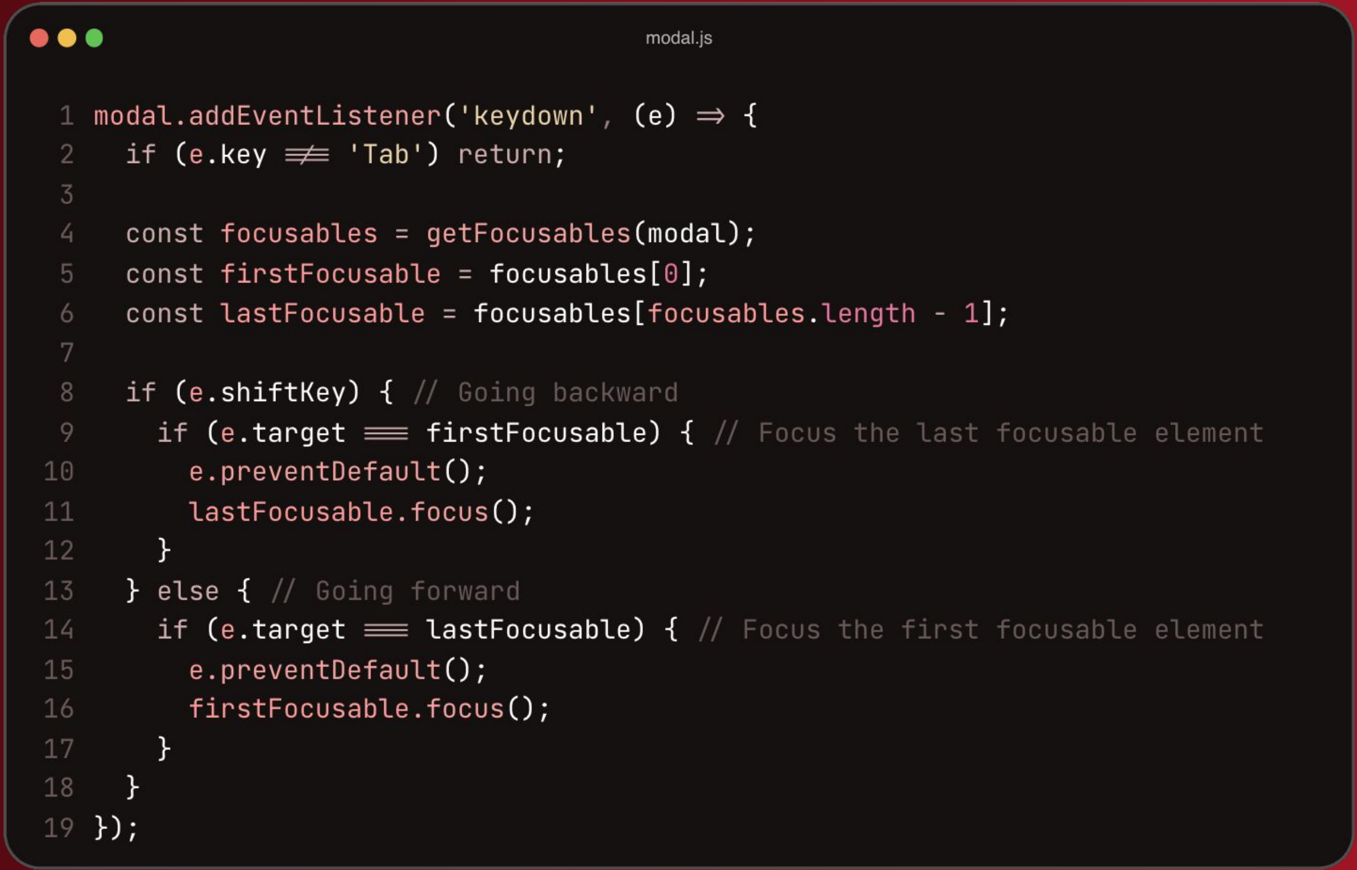
Q: How do we get a reference to all focusable elements?

A: It depends.

Easy way to do it is to use the “focusable” library.

Extremely small library.

Exports a string - **CSS selector of all possible focusable elements.**



```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();
    }
  }
});
```

Demo Time!

Modal: Focus management

Modal: UX Improvements

Forms inside Modals

If our Modal has a **form inside it** - would be nice to **move the focus on the first input** when opening it, instead of the Modal container.

Interactive components inside Modals

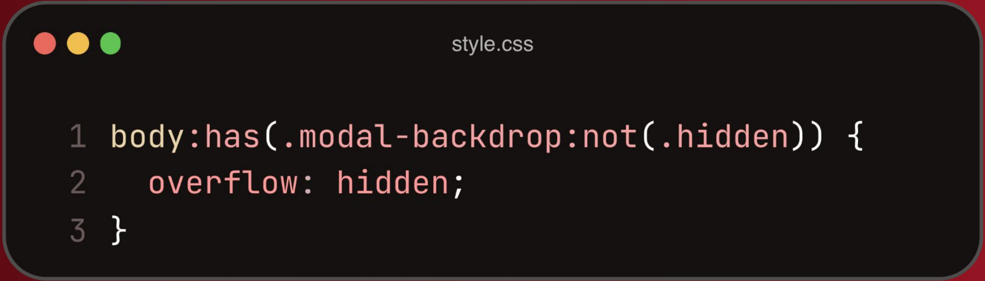
If we have something like a Dropdown menu inside our Modal, we need to make sure their keyboard controls don’t clash with the Modal’s.

For example, if the “Escape” key closes an open Dropdown menu, we need to make sure **the event doesn’t propagate** and trigger the Modal’s “Escape” key listener and close the Modal too.

Modal: UX Improvements

Blocking scroll

Most of the time, our page behind the Modal will be scrollable (bigger than the viewport).
In this case we need to **block the scrollability of the main content.**



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

Modal

Disclaimer: The full code is deliberately omitted on the screenshots for brevity.

Check out the full implementation on the repo.

Check out the more in-depth look on the APG Patterns for the Modal component.

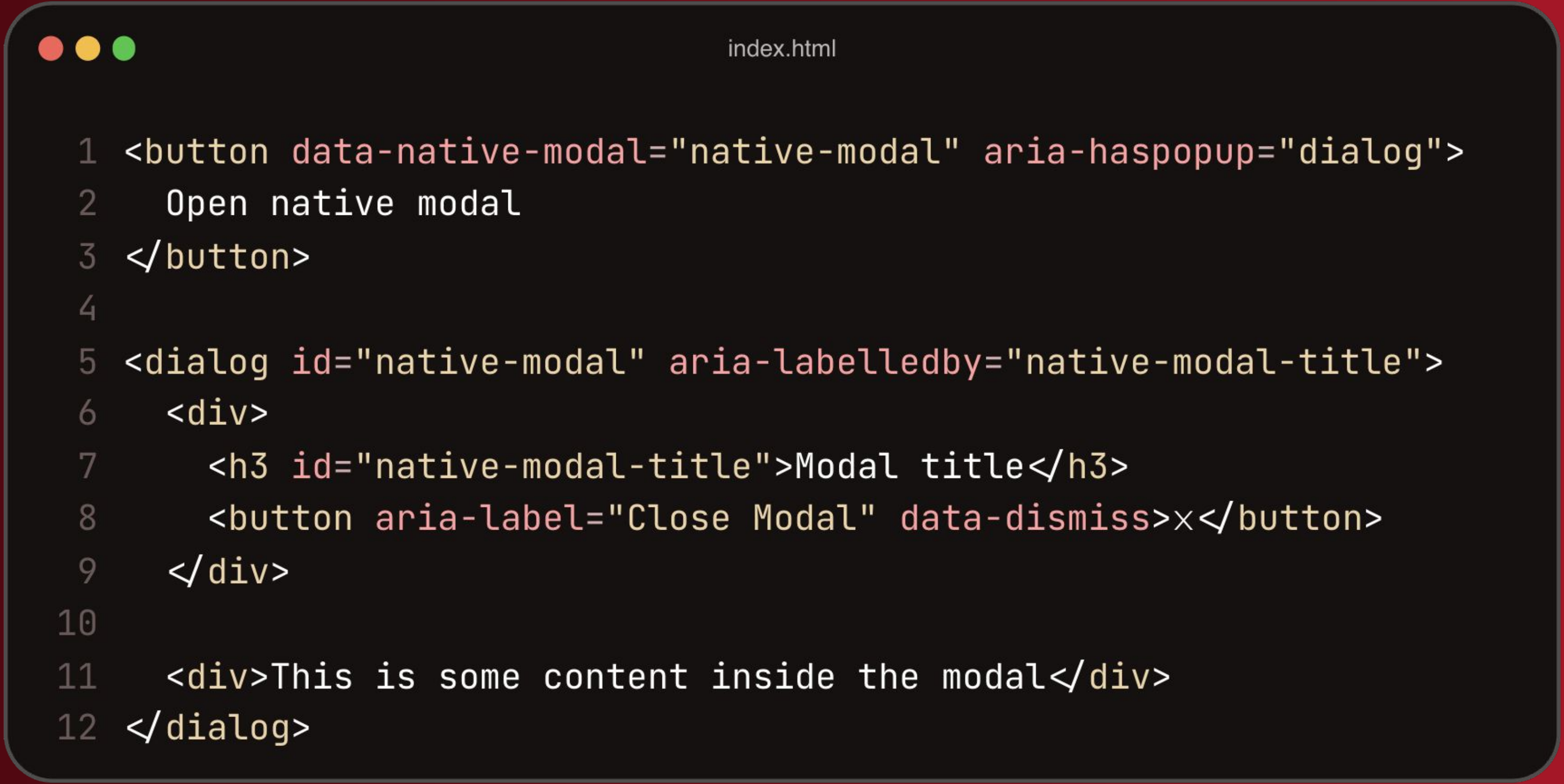
Modal: Modern implementation

Modern HTML introduced the **<dialog>** element for creating Modals and Dialogs and similar components.

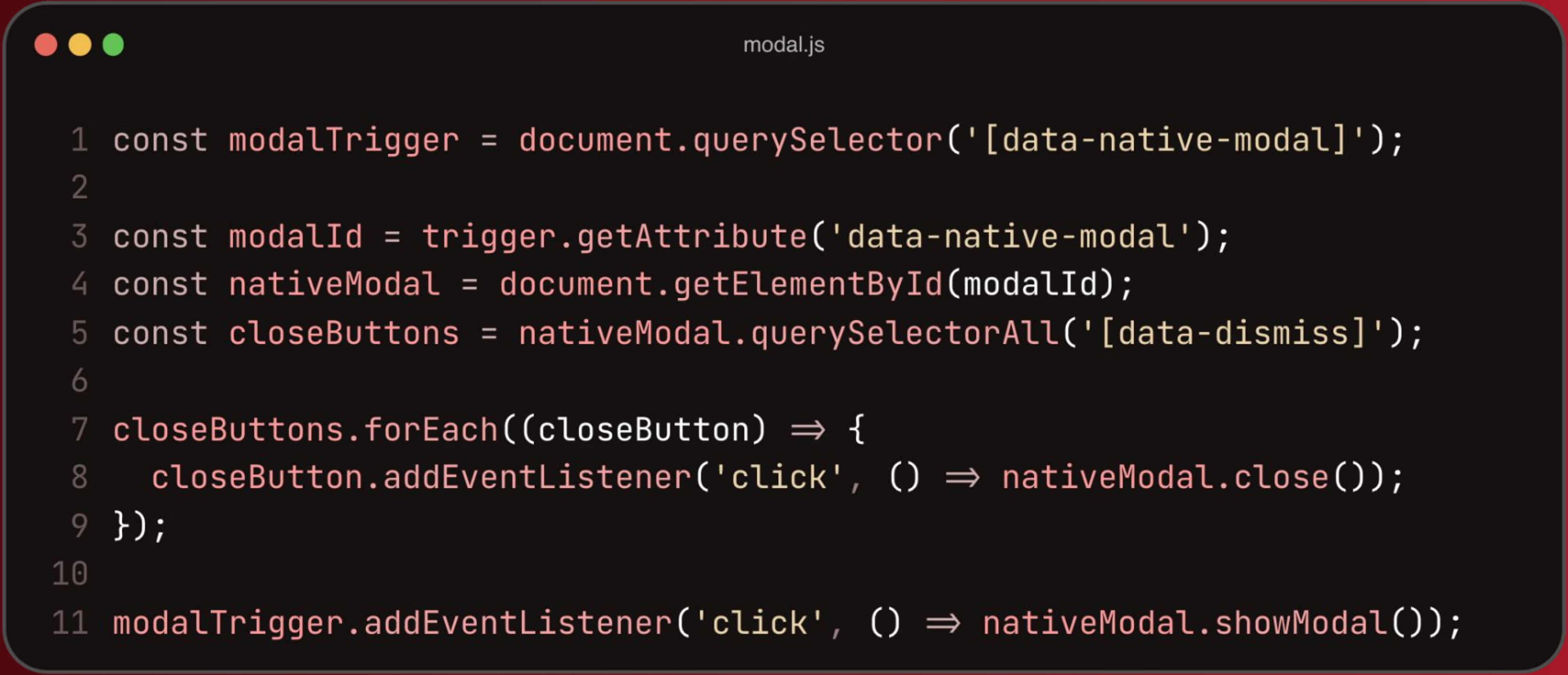
All we have to do is write the markup and hook up the trigger button to it. That's it!

We just need to keep the Modal's **labeling mechanism using `aria-labelledby`**.

Obviously we'll need to also keep the **custom data attributes to reference the correct trigger and dismiss buttons**.


index.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>x</button>
  </div>

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


modal.js

```js
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());
```

Modal: <dialog> perks

- Free **focus trapping**
- **Proper focus management** when opening/closing the Modal
- **Autofocuses the first form input** element inside it, when opening the Modal
- **Free backdrop element** as a pseudo-class `::backdrop`
- Versatility to use as a **Modal** using **`.showModal()`** or a **Dialog** using **`.show()`**
- Guaranteed positioning in front of the rest of the page (#top-layer)

Demo Time!

Building a Modal using <dialog>

Conclusion

We covered a lot today.

We built out some of the most common UI components from scratch and saw some interesting **accessibility techniques and patterns** that emerge during the process.

Go over the code on your own time, play around with it and see if you can solve the **Homework tasks** on your own.

You can always consult the solutions branch if you get stuck.

Thank you!

See ya tomorrow!