TDL SCHOOL

Web A11y for Developers

Day 1: Introduction

Who am I?

Hi, my name's Daniel!

I'm a Senior Web Developer &
TDL Course Instructor for 5+ years.

Celtics fan for life! ☘️

✉ daniel.shijakovski@testdevlab.com

dsijakovski98

Daniel Shijakovski

TDL SCHOOL

Agenda

Day 1: Introduction

Day 2: Components

Day 3: Accessible Forms

Day 4: Best Practices

What’s our goal?

Convey the importance of Web Accessibility to developers, from a practical standpoint.

We will look at **real life scenarios** from a developer’s day-to-day work responsibilities.

Before we begin

Check out the Demo website we will be using for this course.

Clone the corresponding GitHub repository of the website.

What does Accessibility mean in practice?

Provide everyone a way to experience and contribute to life, without any unfair disadvantages.

Web Accessibility is not just about **experiencing** the Web.
It is also about **using and contributing to it**.

Accessibility is **NOT** only for **disabled people**.

People with **temporary impairments**, **situational limitations**.
People whose **physical abilities** have deteriorated due to **aging**.

Accessibility as an equation

We as developers are effectively facing an **input-output** equation.

People use different kinds of <u>inputs</u> and <u>outputs</u> to interact with the Web.

Inputs: Keyboards, touchpads, mouse devices, voice commands...

Outputs: Screen readers, Magnifiers, Braille displays...

Our task as developers is simple:

Make our UIs “survive” the different input-output combinations.

WCAG by W3C

WCAG - Web Content Accessibility Guidelines

Shared standard - what does it mean for a website/application to be accessible.
Developed by **W3C's** **Web Accessibility Initiative** project.

It is **NOT a checklist** to be blindly followed.
It is **NOT a guarantee** that your product will be usable.
It is **NOT a thing we can ignore** and leave to the legal dept.

WCAG just **outlines different principles** that need to be supported by our applications - in a standardized way.

WCAG Levels of Accessibility

Level A
The **bare minimum**. We shouldn’t have a product that doesn’t cover level A success criteria, if we can help it.

Level AA
The **standard we should aim for** most of the time. Essential accessibility principles that support a wide range of users.

Level AAA
The **“holy grail”** - an aspiration we should strive for, but realistically not always practical/possible to achieve.

WCAG principles (POUR)

Perceivable
Users must be able to **perceive** our software.

Operable
Users must be able to **use** our software **regardless of the inputs**.

Understandable
Users must be able to **understand** our software.

Robust
Our software must stand the test of time.

The pillars of Accessibility

1. Keyboard controls (Operable principle)

It is our responsibility as developers, to make each component be able to function **with and without a mouse pointer**.

Anything users are able to do with a mouse, they should be able to do with a keyboard.

Examples

Fill out a form, select a different language in a dropdown, look through an image gallery, add a comment etc.

The pillars of Accessibility

2. Color contrast (Perceivable principle)

Any interactive controls, as well as text, images and important graphical elements on the screen, **must have good color contrast**.

This is often ignored at the **design stage** of the pipeline, often with well meaning intentions.

Examples

Form input outlines, icon buttons, low priority important text (captions) etc.

The pillars of Accessibility

3. Labeling (Understandable & Perceivable principle)

Everything on the screen that users can read or interact with, **must be properly labeled**.

Examples

Form inputs (ex. text inputs, select menus, checkboxes etc), icon buttons, images, informative graphical elements (ex. A checkmark icon instead of “Passed” text) etc.

Screen readers

Hardware/Software technology that **parses the content of applications** and reads it out loud using **text-to-speech**.

Screen readers are **for everyone**, not just blind people.

Screen readers for different platforms:

- JAWS, NVDA and Windows Narrator (built-in) for Windows OS
- VoiceOver (built-in) for macOS and iOS
- Orca for Linux
- TalkBack (built-in) for Android

Screen readers

Common Screen reader - Browser combinations

- JAWS → Chrome
- NVDA → Chrome/Firefox
- VoiceOver → Safari
- Narrator → Edge

Ideally our applications should work **perfectly across all Screen reader/Browser combinations**.

This is not practical, so we want to **prioritize the common ones**.

Demo time!

Screen readers: VoiceOver

Common pitfalls: Design vs Dev team

We need to strive to build a culture of constant **communication** and **feedback with empathy**.

Developers need to be **involved in the design decisions** with our knowledge of accessibility principles.

Accessibility is not the most popular topic.
We may need to be the ones that **share this knowledge** with the rest of the team.

Common pitfalls: Design vs Dev team

Animation vs Reduced motion

Cool looking animations are not an uncommon feature to receive from the design team.

They have their place in the Web experience, but can be **distracting and overwhelming** to people that suffer from **vertigo** or **ADHD**, have a **high sensitivity to motion** etc.

We need to have a way to **accommodate the experience** for them.

Common pitfalls: Design vs Dev team

Animation vs Reduced motion – Solution

1. Have a mechanism that allows for **pausing**, **stopping** or **hiding** the animations for such users.

2.2.2 Pause, Stop, Hide (A)

2. Utilize the CSS query `**prefers-reduced-motion**` to disable all animations for users that have this enabled on their machines.

2.3.3 Animation from Interactions (AAA)

Demo time!

`prefers-reduced-motion`

Common pitfalls: Design vs Dev team

(Missing) focus indicators

No focus indicators (outline: none) are **not an option.**

Keyboard-only users rely on the focus indicators to navigate our websites and know where they are on the page.

The most common way to show a focus indicator is to use the CSS property **`outline`** - but **that's not the only way**.

Common pitfalls: Design vs Dev team

(Missing) focus indicators - Solution

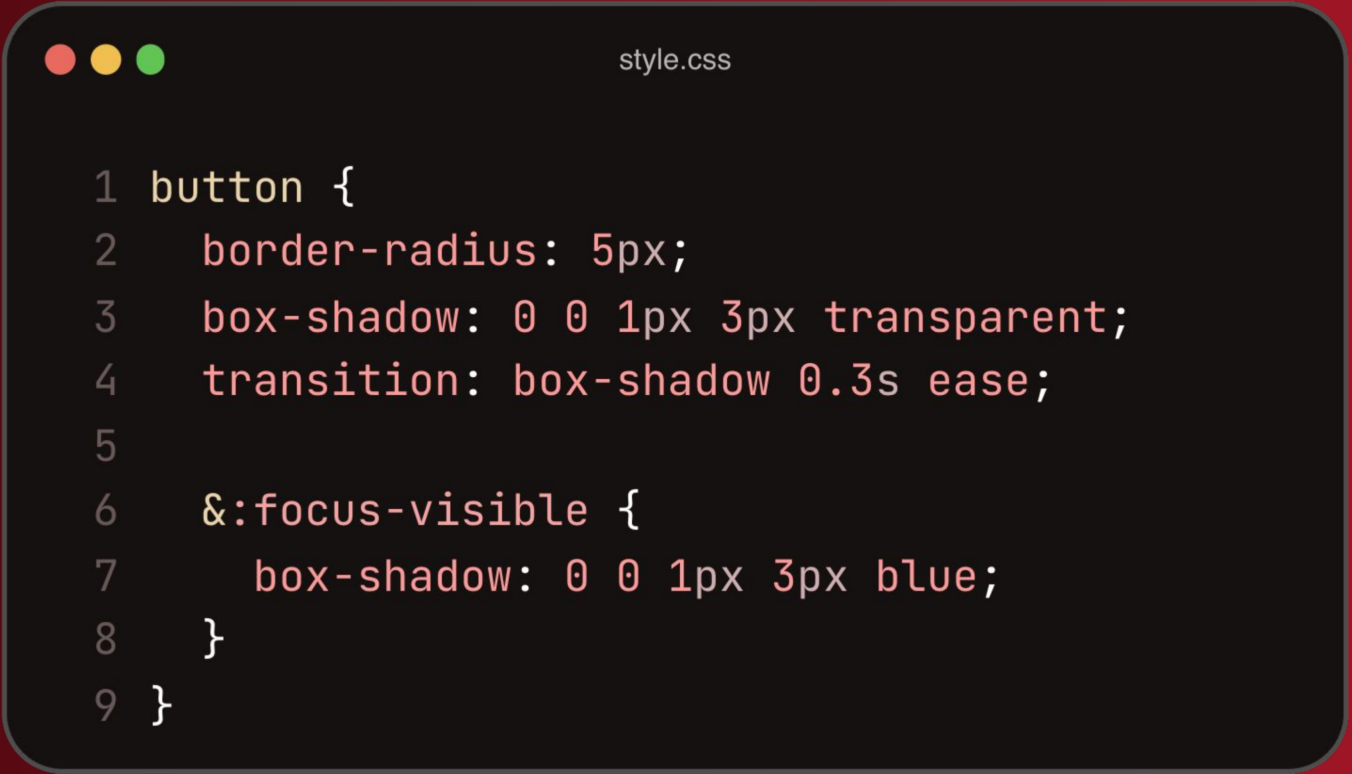
Instead of an outline, we can do the following:

1. Change in the background of the element
2. Box shadow around the element
3. Do some stylistic trickery with the text content of the element
4. Transformations (scale, move) over the element etc.

2.4.7 Focus Visible (AA)

Demo time!

Focus indicators


style.css

```css
button {
  border-radius: 5px;
  box-shadow: 0 0 1px 3px transparent;
  transition: box-shadow 0.3s ease;

  &:focus-visible {
    box-shadow: 0 0 1px 3px blue;
  }
}
```

Quiz Time!

What's the difference between `**focus**` and `**focus-visible**`?

Common pitfalls: Design vs Dev team

focus

General pseudo-class. Applied when an elements receives focus, **no matter what triggered it**.

focus-visible

Specialized pseudo-class. Applied when **browsers' own heuristics determine** that a focus indicator is needed.

For custom focus indicators - it's better to use `focus-visible`.

Break Time!

Grab a snack!

Common pitfalls: Custom vs Native code

Developers love to **reinvent the wheel**, even when that’s not needed.

Most of us have implemented a **custom component from scratch** instead of using a built-in HTML element, at least once.
This is not always a good idea when it comes to accessibility.

Native HTML elements have accessibility built-in.

Raise your hand if you used a <div>/<span> as a **button**. 🙋🏻‍♂️ ← Me

Common pitfalls: Custom vs Native code

Native HTML elements just work

We don’t need to set up **click handlers**, **keyboard event handlers**, or convey the semantic meaning of the element.

Most of the time, it’s just about the CSS with a little bit of JavaScript for that custom behaviour.

It’s almost never a good idea to do a custom implementation if you can get away with using a native HTML element.

Code Challenge (5-7 min)

Implement a button using a <div>

In order to be a button - an element has to have the following:

- A proper semantic role of “button”
- Accessible keyboard controls - “Enter” and “Space” keys are used to click on it
- Focus management - it is correctly placed in the focus order of the page
- Ability to accept HTML attributes to connect it with forms, disable it etc. (optional for this challenge)

Demo Time!

Implementing a button using a <div>

Quiz Time!

What does **tabindex=”-1”** mean?

Common pitfalls: Custom vs Native code

tabindex=”-1”

Removes an element from the regular focus order on the page.
Users cannot focus on those element with their keyboard.

This **doesn’t mean** that these elements **can’t receive focus**. We can still do that programmatically from our code.

There are times when this is actually exactly what we need.

Common pitfalls: Custom vs Native code

Landmark elements

Too often we see websites and web applications riddled with a bunch of <div> elements.

Modern HTML (HTML5) introduced **landmark elements** to help developers convey the correct semantic meaning of a section on the page.

Each landmark element has a **corresponding landmark role**.

Common pitfalls: Custom vs Native code

Landmark elements

Element	When is it a Landmark?	Role
`<main>`	Always	“main”
`<nav>`	Always	“navigation”
`<aside>`	Always	“complementary”
`<section>`	Only with a label	“region”
`<header>`	Only at top-level	“banner”
`<footer>`	Only at top-level	“contentinfo”

Common pitfalls: Custom vs Native code

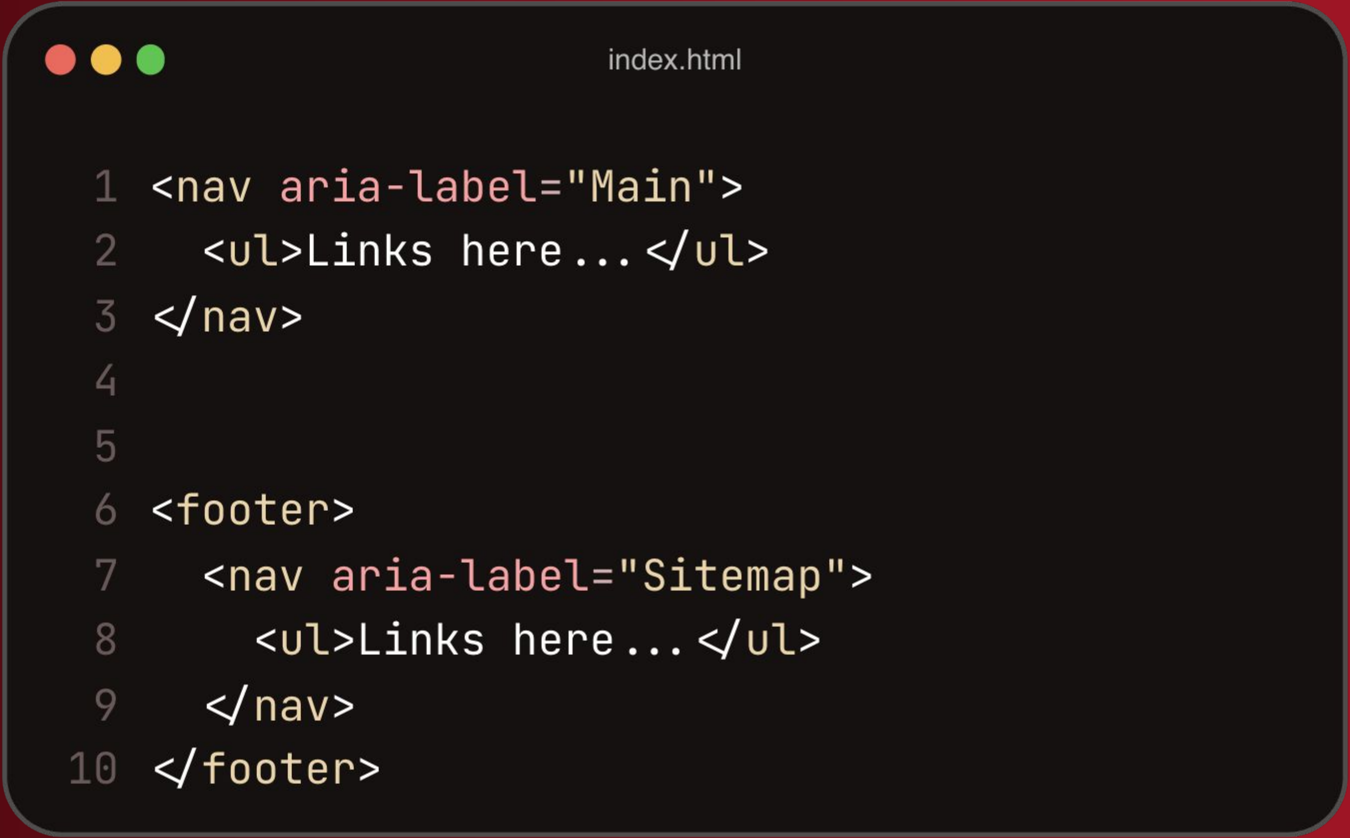
Landmark elements

<main>

Used to wrap the **main content** on your page - use **one per page**.

<nav>

Used to denote sections where **navigational links** are located.
If there are multiple <nav> elements on one page - label them.


index.html

```html
<nav aria-label="Main">
  <ul>Links here...</ul>
</nav>


<footer>
  <nav aria-label="Sitemap">
    <ul>Links here...</ul>
  </nav>
</footer>
```

Common pitfalls: Custom vs Native code

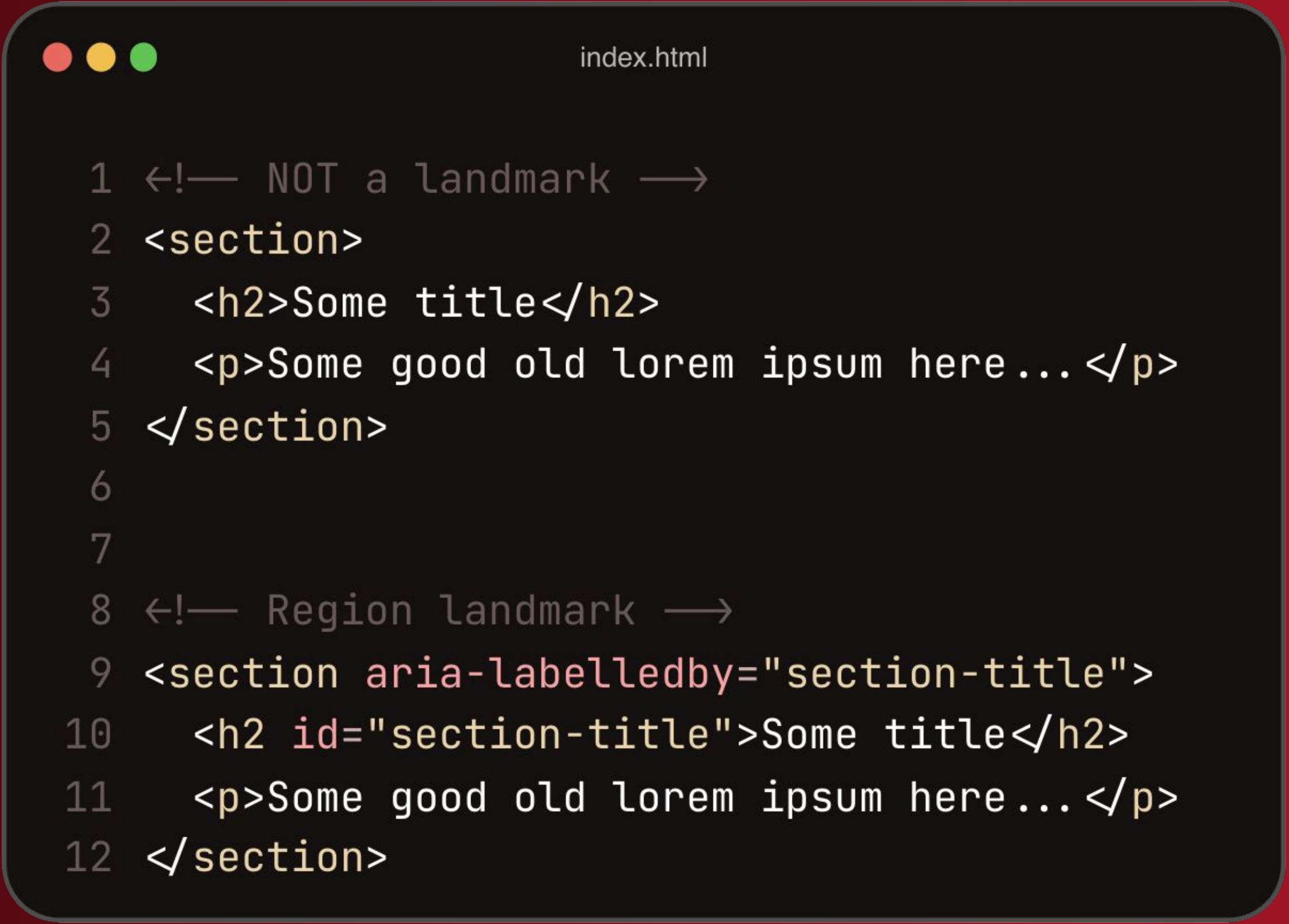
Landmark elements

<aside>

Used to denote **complimentary sections** on our page (ex. sidebars). Same labeling rules apply as for the <nav> element.

<section>

On its own it has no semantic meaning. It becomes a **“region” landmark element when we label it** (ex. Using \`aria-labelledby\`).


index.html

```html
<!-- NOT a landmark -->
<section>
  <h2>Some title</h2>
  <p>Some good old lorem ipsum here...</p>
</section>


<!-- Region landmark -->
<section aria-labelledby="section-title">
  <h2 id="section-title">Some title</h2>
  <p>Some good old lorem ipsum here...</p>
</section>
```

Common pitfalls: Custom vs Native code

Landmark elements

<header>

Usually used at the **top-level** of your page (ex. Where the main navigation, logo, user auth state etc. are located).

It can also be used **inside a <section> or an <article>**.
In such cases - don’t add the role of “banner” to it - we must have **only one “banner” per page**.

Common pitfalls: Custom vs Native code

Landmark elements

<footer>

Usually used at the **top-level** of your page - mainly at the very end of the page.

It can also be used within <section> and <article> elements.
Same rules apply for its **“contentinfo” role**, as for the “banner” role - we can only have **one per page**.

Common pitfalls: Custom vs Native code

Landmark elements

Before 2016/17 it was still necessary to add the landmark role attribute, so that Screen Readers would read them correctly.

Today this is not the case and we can just use the landmark elements as is.

If you are working on a legacy product that needs to **support legacy browsers/screen readers** - use **both the landmark element and its associated landmark role**.

Take Home Challenge

Repair a broken page using proper landmark elements

Use the landmark elements we covered during this lesson and try to fix all the issues on the Broken Intro page of the Demo website.

Keep in mind the rules of each landmark element to make sure assistive technologies are not confused by (for example) multiple <nav> elements etc.

See how many issues you can find and then compare your solution with the one in the solutions branch.

Common pitfalls: Keyboard accessibility

Every interaction that users can do with **their mouse** - they need to be able to do using **only their keyboard** (with rare exceptions).

It is most often left out within **custom implementations of components.**
Depending on the (custom) component, we need to make sure we cover:

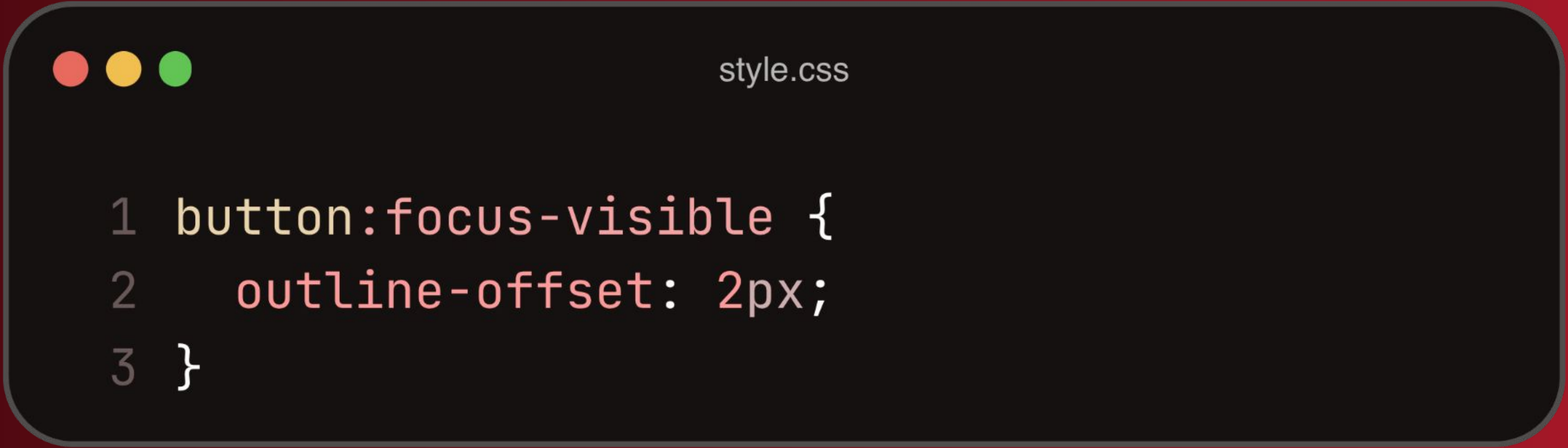
- **Keyboard events** (ex. “Enter” and “Space” for buttons)
- **Focus management** (will be covered during the next lesson)
- **Navigation throughout the component** (Arrow keys, “Tab” etc.)

Common pitfalls: Contrast issues (focus)

Focus indicators need to be not only present, but also **clearly visible** and **distinguishable** from their surrounding environment.

Common examples of bad focus indicators include ones with:

- Insufficient color contrast to the element/environment
- Too thin outlines, not noticeable at 100% zoom
- **The same color as the background of their element and are right next to each other**



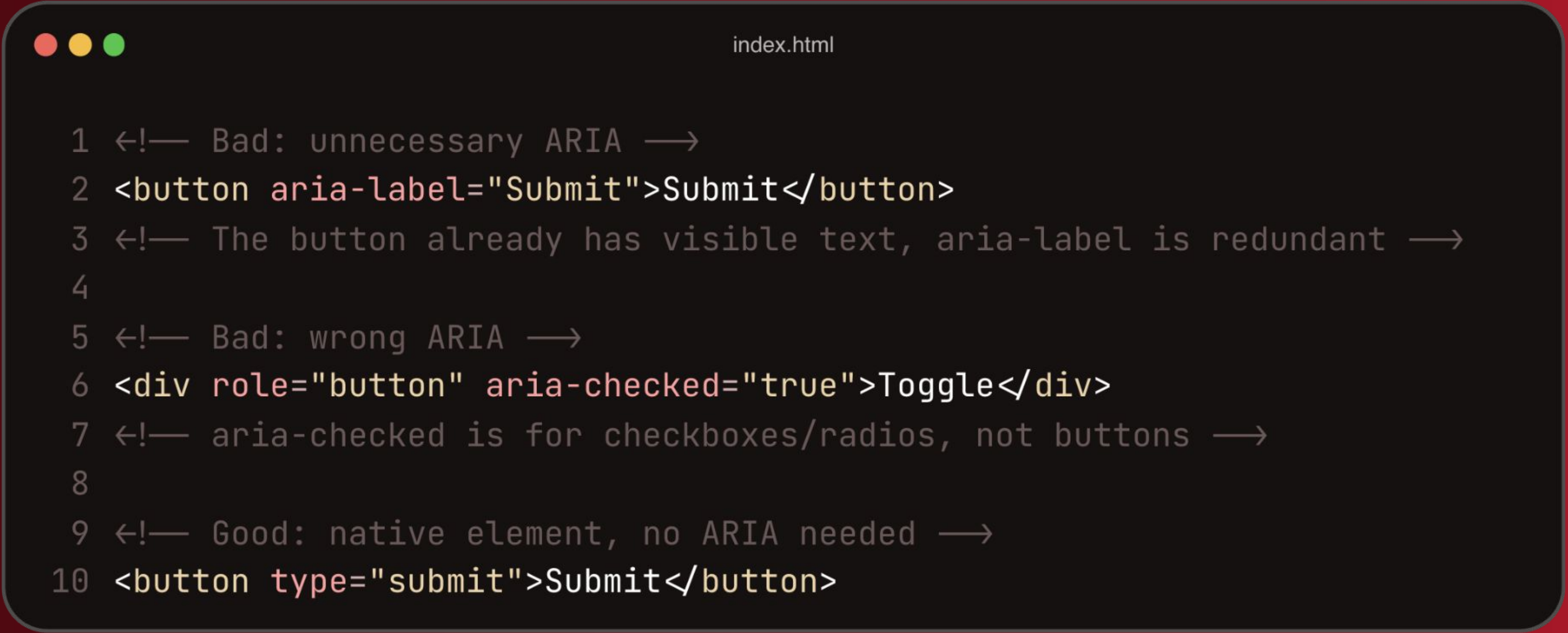
```css
button:focus-visible {
  outline-offset: 2px;
}
```

Common pitfalls: Too much/wrong ARIA

“No ARIA is better than bad ARIA” - The Web Accessibility Initiative’s ARIA Authoring Practice Guide (APG).

We need to be **aware of the purpose** of the ARIA attributes before we use them on our components.

Often there are better alternatives (ex. Just use a built-in HTML element) to solve that kind of problem.



index.html

```html
<!-- Bad: unnecessary ARIA -->
<button aria-label="Submit">Submit</button>
<!-- The button already has visible text, aria-label is redundant -->

<!-- Bad: wrong ARIA -->
<div role="button" aria-checked="true">Toggle</div>
<!-- aria-checked is for checkboxes/radios, not buttons -->

<!-- Good: native element, no ARIA needed -->
<button type="submit">Submit</button>
```

Common pitfalls: Libraries everywhere

There are many scenarios where **third party libraries** are very appropriate: Routing, Authentication, Component libraries etc.

There are also situations where we use them when we need some **specific custom logic** in one or two places in our application.

When we use a third party library, **we are putting our trust** in the author of that library.

Accessibility is also a very tricky thing to get right, especially for custom components.

Common pitfalls: Libraries everywhere

Full disclosure, there are libraries that provide **comprehensive accessible components**: Base UI, Headless UI, Radix UI etc.

We don’t need such complex libraries in our code **if we only need a couple of custom functionalities or components** (ex. Modal or an Autocomplete component).

We can take some time to implement them ourselves. This will give us **more control over the accessibility** of our code.

The **next lesson** will cover this exact topic - **building custom UI components**.

Common pitfalls: Auto tests are NOT enough

The most popular tool in Accessibility Automation testing - **Dequeue System's axe-core engine** claims to only catch up to 57% of accessibility issues.

Automation is a great tool to have when testing for accessibility issues - **but it is not enough**.

While it does catch some obvious accessibility violations and informs on some best practices - it fails to catch very important issues in our applications:

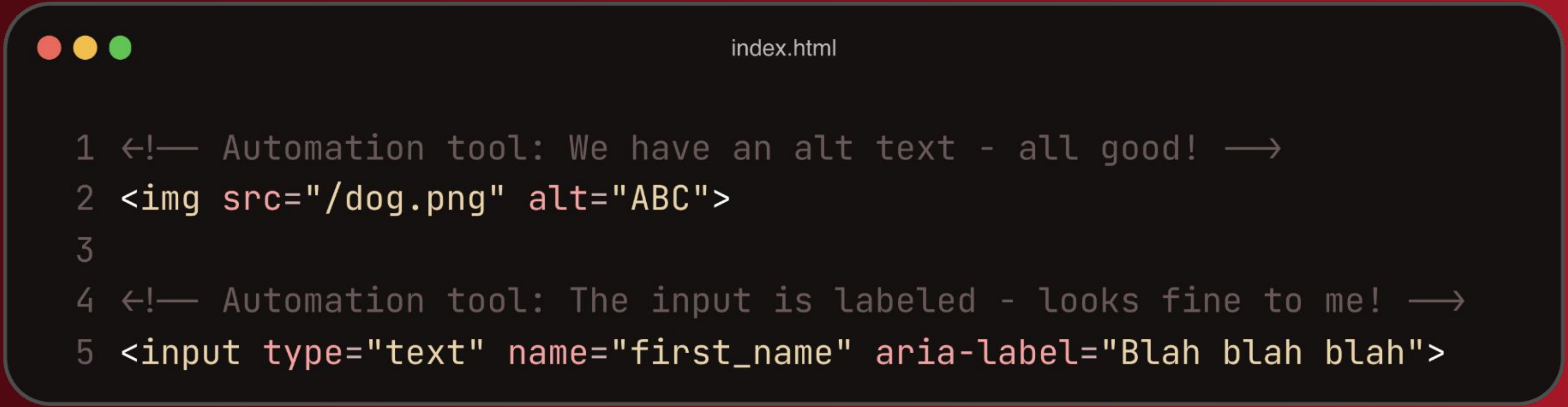
Common pitfalls: Auto tests are NOT enough

Meaningful labels/alternative text

Automation tools can notify us when we have a **missing label** or a **missing alt-text** on an image.

They cannot tell us if that label/alt-text **makes logical sense or is meaningful in any way**.

We need to **manually confirm** that all of our **labels**, **descriptions**, **alternative texts** etc. make sense within the context of their surroundings.


index.html

```html
<!-- Automation tool: We have an alt text - all good! -->
<img src="/dog.png" alt="ABC">

<!-- Automation tool: The input is labeled - looks fine to me! -->
<input type="text" name="first_name" aria-label="Blah blah blah">
```

Common pitfalls: Auto tests are NOT enough

Keyboard interactions

No automation tool can tell you **if the focus order within your page makes sense**, or that **all keyboard events are handled properly** according to the role of that element.

We need to make sure that we **manually test the focus order** and expected **keyboard interactions** with each component on the page.

Common pitfalls: Auto tests are NOT enough

Visual/Cognitive issues

Automation tools cannot tell us if we have:

- Distracting animations
- Confusing or overwhelming text content
- Issues with zoomability or responsiveness

We need to manually go through and test these things out ourselves.

Common pitfalls: Auto tests are NOT enough

There is not always a malicious intent or arrogance behind teams that rely on automation tests for their accessibility coverage.

Sometimes it’s the **lack of knowledge and information sharing** that can put even the most well intentioned teams in that position.

This is why it is important to approach the topic of Accessibility with **empathy** and be able to educate people on it.
We all have the same goal here - make the Web more accessible.

Conclusion

Web accessibility is a goal that we should strive to achieve for many reasons. Not the least of which is that **it's just the right thing to do.**

We as developers need to be **equipped** with as many tools in our "toolbelt" as possible - so that we can tackle the technical challenges that Web accessibility presents.

Empathy is where we start and the code is next.

Thank you!

See ya tomorrow!