Best Practices

Up until this lesson, you could say that we mostly focused on a bit more higher level concepts such as common UI components and forms in general. In contrast to that, this lesson will cover slightly more detailed principles of accessibility.

We will start off by talking about **general best practices to follow** in your day-to-day development, followed by some bite-sized **tips and tricks** you can utilize for some quick wins within your apps. We will then finish off the lesson by moving **beyond the actual development stage** and talk about how we can expand the idea of accessibility throughout an entire team of developers, designers, testers, managers etc.

Focus management

As discussed in the previous lessons, focus indicators are extremely important especially to users that rely on the keyboard for their navigation through the Web. As such, **proper focus management** is crucial to the accessibility of any website or application.

Now, from a high level perspective, this means that **we need to make sure the focus order follows the natural visual flow of our page**. There is actually a WCAG success criterion that calls for this kind of behaviour - 2.4.3 Focus Order.

We want our users to be able to reliably and predictably move through each page knowing that the structure that they are seeing on their screens (or hearing of from their screen readers) follows a clear order and all of the focusable elements on the page follow that order as well.

For example, we need to **avoid using "tabindex" values that are larger than 0** so that the focus order of elements is not different from their visual structure and position on the page.

On the other hand, if we look at focus management from a more fine grained perspective, we will find scenarios where we need to **intentionally direct the focus indicator** to specific elements such that it doesn't get lost and/or provides for a smoother experience for our users. Let's see what both of these scenarios mean by looking at 2 examples: **clearable inputs** and **deletable options**.

Clearable inputs

If we happen to have an input component that has a “Clear” icon button which users can press to quickly clear the entire input, we need to make sure that the keyboard experience for users is such that **after pressing on the “Clear” button - the input element is focused**.

This is needed because after we press on the “Clear” button with our keyboard - it will be removed from the DOM - because it is no longer needed since the input is empty. But the focus was on the “Clear” button - that’s how we got to press it in the first place - so where does the focus go? The answer to this question **depends on the browser and the platform** on which we do this, but most of the time **it just moves to the <body>** element. This is sometimes referred to as **“dead focus”**.

Regardless of where it moves, keyboard users don’t and shouldn’t expect this to happen, they just wanted to clear some text from an input - it stands to reason that they probably wanna keep typing in that input. So, what we want to do is to **intentionally move the focus to the input element once we clear it**.

HTML

```
<div class="clearable-input">
  <input type="text" id="clearable-name" placeholder="Your name here..." >
  <button type="button" class="clear invisible" aria-label="Clear">✕</button>
</div>
```

JavaScript

```
const input = document.querySelector('.clearable-input input');
const clearButton = input.nextElementSibling;

input.addEventListener('input', () => {
  clearButton.classList.toggle('invisible', input.value.length === 0);
});

clearButton.addEventListener('click', () => {
  clearButton.classList.add('invisible');

  input.value = '';
  input.focus();
});
```

Deletable options

Similarly to the previous example of a clearable input - if we have some sort of **list of items** such that any item can be removed from the list by (for example) pressing on a “Remove” icon button next to the item - we also need to make sure that **once we delete that item - the focus moves to the appropriate logical next position**.

Now, it may not be immediately obvious what that next logical position is, so here’s a helpful guide to follow:

- If we’re deleting the **last element** - the focus should move to the **item before it**
- If we are deleting **any other element** - the focus should move to the **item after it**

Okay, so we now know on which item in the list the focus should go - but do we know on what exact element within the item should we put that focus on? The answer to this is “**Not necessarily**”.
This will depend on the type of items you have in the list. If **the entire item element is clickable and/or interactable**, maybe we want to **move the focus on that entire item element**. If on the other hand **the item is not interactable** and we only have some text and a “Remove” button next to it - then we would have to **move the focus on that next item’s “Remove” button**.

It all depends on the specific list of items that you are implementing, but the underlying principle of the intentional focus management stays the same - **once we remove an item from a list (using the keyboard), the focus needs to move to an appropriate logical place** so that users don’t get confused by the unexpected removal of their focus indicator.

```HTML
<ul class="options-list">
  <li>
    Task 1 <button class="remove" aria-label="Remove item">×</button>
  </li>
  <li>
    Task 2 <button class="remove" aria-label="Remove item">×</button>
  </li>
  <li>
    Task 3 <button class="remove" aria-label="Remove item">×</button>
  </li>
</ul>
```

JavaScript

```
const optionsList = document.querySelector('.options-list');
const resetListButton = optionsList.nextElementSibling;

// Event delegation for better performance - fewer click listeners
optionsList.addEventListener('click', (e) => {
  const removeButton = e.target.closest('button.remove');
  if (!removeButton) return;

  const currentOptions = Array.from(optionsList.querySelectorAll('li'));
  const deletingLastElement = currentOptions.length === 1;

  const listElement = removeButton.parentElement;
  const idx = currentOptions.indexOf(listElement);

  if (idx === -1) return;

  let newIdx;

  if (idx === currentOptions.length - 1) {
    // Deleting last element - move focus to previous element
    newIdx = Math.max(idx - 1, 0);
  } else {
    // Deleting any other element - move focus to next element
    newIdx = idx + 1;
  }

  if (deletingLastElement) {
    // Focus an "Add item" button or something similar
    resetListButton.focus();
  } else {
    // Focus the new item (depends on what you need to focus on your own items)
    currentOptions[newIdx].querySelector('button.remove')?.focus();
  }
  // Delete the clicked item
  listElement.remove();
});
```

These are just a few of many scenarios where we need to be mindful of the focus indicator, so that we minimize (ideally completely remove) the “dead focus” situation.

Tooltips

Probably the most widely used piece of UI in any interactive web application - **the tooltip** presents an interesting accessibility challenge and **it is often implemented and/or used incorrectly** from that perspective.

Tooltips (literally meaning just **“tips for tools”**) are those small popups that appear next to some interactive element that help us understand that element’s meaning. For example, let’s say we have a cool looking icon button that’s not widely used throughout the Web - we may want to show a tooltip next to it when users hover over it so that they can understand what that icon button does.

These things are used almost everywhere, so given that - it’s important that we understand **how to properly set them up and use them** in our applications.

What are Tooltips used for?

Tooltips can serve 2 purposes - they can be used as a **primary label** for an element, or as a **helpful description** of an element.

Tooltips as labels

We generally want to use them as **primary labels** when there is **absolutely no space for us to include a visible label**. We do this by **programmatically connecting** the tooltip element to the element it is trying to label, using `aria-labelledby`.

```HTML
<button aria-labelledby="notifications-tooltip">
  <notification-bell-icon>
</button>

<div id="notifications-tooltip" role="tooltip">
  <div class="tooltip-content">Notifications</div>
</div>
```

Notice that we also need to use the **“role”** of **“tooltip”** on the tooltip element itself. After this, we want to set up our **hover** and **focus** states on our button, so that the tooltip is shown only in that situation.

An important note regarding tooltips - **they must stay open while the mouse is hovering on them too**. There are many ways to do this - in our case we will just add a bit of a transition delay for the tooltip opening and closing, so that the mouse pointer has enough time to enter the tooltip after exiting the button.

There is of course the question of **positioning**, as it is very difficult to use **absolute positioning** if the control element (the button) and the tooltip are siblings, such as in the example above. In this case, we can use the super new CSS functionality - [anchor positioning] which will allow us to position the tooltip as if it was “anchored” next to the button.

```CSS
button {
  anchor-name: --tooltip-anchor;

  & + [role="tooltip"] {
    position: absolute;
    position-anchor: --tooltip-anchor;
    top: anchor(center);
    left: anchor(right);

    opacity: 0;
    pointer-events: none;
  }
}
```

This positioning mechanism is very new, so I highly encourage you all to do some research on it, on your own time, as it can really help you out in situations like these.

Okay, we positioned the tooltip, now we want to make sure that it appears when we **hover** over or when we **focus** on the button.

```CSS
button:is(:hover, :focus-visible) + [role="tooltip"] {
  opacity: 1;
  pointer-events: all;
}
```

We also want the tooltip to **stay open while it is being hovered**, so we add that as well.

```CSS
button:is(:hover, :focus-visible) + [role="tooltip"],
[role="tooltip"]:hover {
  opacity: 1;
  pointer-events: all;
}
```

If we are animating (transitioning) the showing and hiding of the tooltip, you might want to add **a small transition delay** on the tooltip initially, and **remove it once it is shown**.

```CSS
button {
  anchor-name: --tooltip-anchor;

  & + [role="tooltip"] {
    position: absolute;
    position-anchor: --tooltip-anchor;
    top: anchor(center);
    left: anchor(right);

    opacity: 0;
    poiter-events: none;

    transition: opacity 0.3s ease;
    transition-delay: 0.1s;
  }
}

button:is(:hover, :focus-visible) + [role="tooltip"],
[role="tooltip"]:hover {
  visibility: visible;
  opacity: 1;

  transition-delay: 0s;
}
```

Tooltips as helpful descriptions

We can also use tooltips to provide **additional helpful descriptions** to an already labeled (visually or not) element. The set up is pretty much the same, except for the ARIA attribute used to provide the description, which obviously will be **aria-describedby**.

```HTML
<button aria-label="Notifications" aria-describedby="notifications-tooltip">
  <notification-bell-icon>
</button>

<div id="notifications-tooltip" role="tooltip">
  <div class="tooltip-content">
    Check out the latest activity across your network
  </div>
</div>
```

Open discussion: How should we handle these tooltips on mobile devices?

Toggletips

While tooltips show their content when we hover over or focus on an interactive element (ex. button), **toggletips** work by **showing their content on mouse click**.
This is because **toggletips are technically buttons** and their sole purpose is to show some extra information about a particular element.

Think of long complicated forms with vaguely labeled inputs - having a toggletip next to specific inputs would make it very helpful for users to get more context on what that input represents and what information it requires us to fill in.

Toggletips, by their nature, **can only serve as additional descriptions - not labels**. This is because labels provide **essential information** on what something is and having that information behind a button that will only show it after a click - makes for bad user experience. Moreover, this additional description is **not directly connected to the interactive element** with something like **aria-describedby**.

Toggletips essentially say to users: “Click me to reveal some hidden information”, so initially this information is not available (ex. When we focus on the toggletip button).
In order to make sure this information is read out loud by a screen reader **only when users click on the toggletip** - we can take advantage of **live regions**.

We can have an empty live region alongside the toggletip button and **programmatically fill it out** with some content once we click on it to reveal it. The way we provide the actual content can differ depending on your use case, but for now we will just use a simple custom data attribute **data-toggletip** with its value being the actual additional information needed.

HTML

```
<label for="emp-id">Employee ID</label>
<input id="emp-id" type="text" inputmode="numeric" placeholder="Employee ID">

<button aria-label="More info" data-toggletip="Last 4 digits of your birthday">
  i
  <div role="status"></div>
</button>
```

Then, we can programmatically show the content inside of the “status” live region when we click the toggletip, and remove it when we **move focus out of it**, or **click outside of it** - basically **on the “blur” event**.

JavaScript

```
const toggletips = document.querySelectorAll('[data-toggletip]');

toggletips.forEach((toggletip) => {
  const liveRegion = toggletip.querySelector('[role="status"]');
  const content = toggletip.getAttribute('data-toggletip');

  toggletip.addEventListener('click', (e) => {
    liveRegion.textContent = e.target === toggletip ? content : '';
  });

  toggletip.addEventListener('blur', () => {
    liveRegion.textContent = '';
  });
});
```

The styling and positioning is pretty much the same as the tooltip element - we can use **CSS anchor positioning to position the live region element** wherever we want and **hide it when it is empty** (has no content inside of it).

The reason we have these distinctions between a tooltip and a toggletip is because they are often used interchangeably - more often than not, a tooltip is used where a toggletip is needed. The rules of thumb to follow are these:

If you have a “More information” icon button that is next to an element - **then you need a toggletip**. The sole purpose of that icon button is to show and hide the additional information that it contains.
If however, you have **elements that require labels**, but you are short on space, or perhaps you want to provide some **additional description** to an element but you can’t or don’t want to put an extra “info” icon button next to the element - you want to use a tooltip.

Tips & Tricks

This section of today’s lesson is dedicated to **very quick and small wins** you can integrate into your codebase and your workflow today, which will make your applications less prone to accessibility issues.

document.activeElement

A very easy way to find out what’s the **currently focused element** on the pages is to use **`document.activeElement`**. This is a reference that the Document object constantly updates to point to the element that currently has focus on the page.

This is very helpful for **accessibility debugging**, especially when we have things like **big hidden menus and sidebars** and need to figure out where your focus indicator is going. Combine that with Chromium browsers’ DevTools’ Live Expressions feature and we can easily monitor the currently focused element on the page and make our debugging experience that much easier to deal with.

Development linter stylesheets

It is not uncommon for developers to **forget to add alt-text to images** or **forget to add an accessible label to an icon (svg)**. In cases such as these, we have automated tools that can catch this, but we can save ourselves the embarrassment of having our accessibility CI fail because we forgot alt-text, by **leveraging CSS** to our advantage.

```CSS
/* Images that don't have an "alt" attribute */
img:not([alt]) {
  outline: 6px solid red !important;
  outline-offset: 2px;
}
```

```
/* SVG elements that are not hidden and not labelled */
svg:not([aria-hidden="true"]):not([aria-label]):not([aria-labelledby]) {
  outline: 6px solid darkorange !important;
  outline-offset: 2px;
}
```

We need to be careful with these though, because they can quickly get out of hand, so we may need to determine some rules within our application, like: “Is an image that’s labeled using `aria-label` or `aria-labelledby` considered valid?”.

It really depends on how deep you want to go with these CSS selectors indicators, but you can set up some pretty extensive styles for some obvious low-hanging fruits to make sure they are glaring on the screen. Think along the lines of **buttons implemented using a / element**, elements that have a **tabindex with a value larger than 0**, **anchor elements with missing `href` attributes** etc.

Visually hidden elements

These are elements that are **present in both the DOM and the Accessibility tree, but are not visually shown on the screen**, nor do they interrupt or break the layout in the process.
Such elements can be used to add **additional screen-reader-only descriptions** to certain sections of a page or a component.

There’s this famous CSS class called **`sr-only`** (which stands for “Screen Reader only”) which almost every CSS framework out there has and we can use to make an element visually hidden but still available to assistive technologies.

```
CSS
.sr-only {
  clip: rect(0 0 0 0);
  clip-path: inset(50%);
  height: 1px;
  overflow: hidden;
  position: absolute;
  white-space: nowrap;
  width: 1px;
}
```

Some use cases where we might want to use visually hidden elements include:

Loading message within a loading indicator

```HTML
<div>
  <div class="spinner"></div>
  <span class="sr-only">Loading images...</span>
</div>
```

Invisible Live regions that announce changes

```HTML
<div role="status">
  <span class="sr-only">15 results found</span>
</div>
```

Providing additional context to vaguely named links/buttons

```HTML
<a href="https://testdevlab.com">
  Read more <span class="sr-only">about the company I work at</span>
</a>
```

There are so many more examples, we would be here all day if we went through all of them. The idea is that we can provide all kinds of additional information to users that rely on assistive technologies, in order to **give them more context about what's happening on our page** at any time and we can do that using just a simple CSS class.

High Contrast mode

We talked about focus indicators a good amount during this course, so let's talk about them some more, shall we? So, we said that we can use alternatives to the **outline** property to show a focus indicator - for example, a box shadow around the focused elements.

This does work, but there's a common mistake developers make when doing this and that's setting **`outline: none;`** on focused elements that have a different focus indicator (ex.

box-shadow). Now, this might seem strange at first because why would we not hide the outline since we are showing an alternative focus indicator, right?

Well, that mistake rears its head once someone uses our website with **"High Contrast mode"** (forced colors on Chromium browsers) enabled.
High Contrast mode is an accessibility feature that users can enable on their devices, which will then **make their devices use a limited color palette with a high contrast**, so that people with low vision can read the content on their screens easier.

When users are using our website with High Contrast mode, and we have a custom focus indicator (ex. box-shadow) set up along with `outline: none;` - 2 things will happen:

1. The custom focus indicator won't be shown, because **box shadows are hidden in High Contrast mode**
2. The outline will not be shown because we disabled it entirely

To mitigate this, we can swap out the **`outline: none;`** with an **`outline-color: transparent;`** statement - which will still hide the outline in normal contrast mode, but **will actually show it** when using High Contrast mode.

DevTools features

While we are on the topic of DevTools and their **forced colors** feature, let's quickly go over some helpful tools that the (Chromium browsers) DevTools provide.

Emulator

Chromium browsers' DevTools have a very powerful **emulation capability** such that we can:

- See how users with enabled "Reduced Motion" would experience our page
- Emulate High Contrast mode color palettes
- Simulate color deficiencies such as Protanopia (no red), Deuteranopia (no green), Tritanopia (no blue), and Achromatopsia (no color, grayscale)

Most modern browsers (Chrome, Firefox, Edge) also have the capability to show the **Accessibility tree** visualized the same way as the DOM tree within the "Inspect" panel.

Accessibility tree

The Accessibility tree is a virtual tree-like data structure similar to the DOM. It is constructed as a hierarchical structure of nodes and elements that help assistive technologies parse the HTML content and help screen readers navigate and properly announce the content of Web pages.

We can see how this tree looks for our application, by opening the “Inspect” panel in the DevTools of most modern browsers and toggling the “Accessibility Tree” icon button on the top-right corner of the DevTools viewport.

Using this Accessibility Tree visualizer, we can clearly see how screen readers see our application, which elements are properly hidden, which ones are correctly labeled, do they have the correct roles etc. It is an extremely useful tool for developers to consult while building features on a daily basis.

Beyond Development

To finish off this lesson, we will talk about how we can **integrate Accessibility as a cultural characteristic** of our entire team. We will go beyond just the development stage and look at all of the ways we can bring the topic of accessibility closer to our designers, our testers and pretty much everyone else on our team.

Accessibility in the Pipeline

If we were to imagine a common development pipeline like the one below:

```HTML
Design → Development → Code Review → Testing → Deployment
```

What would be your answer if I asked you: “Where would Accessibility be placed here?”
There technically is no wrong answer, as wherever you say - that’s where Accessibility should be placed. In fact, the actual right answer is that **Accessibility should be present throughout the entire pipeline**.
At each step, from the Design stage to the Deployment stage - we want to have a conscious effort of including Accessibility in our process, no matter what we are working on.

You may have heard of the phrase “Shift-Left testing”, if you haven’t, it basically represents this idea that if we imagine that pipeline from above as going from left to right, kind of like how

time moves forward - **we want to catch and resolve (accessibility) issues as early as possible in the development pipeline.**
This means that we want to “shift left” (i.e. do it as early as possible) the process of identifying and fixing accessibility issues, so that they are **cheaper and easier to fix**.

Design stage

We touched on this at the beginning of this course - designers play a crucial role in the accessibility of our applications. We as developers need to be in constant communication with them and share our thoughts and ideas between one another, so that we can make sure that we are building an accessible product.

Be on the lookout for any potential **color contrast issues**, **missing focus indicators**, ask questions about **animations**, font sizes, spacing, hover states, error states etc. and **lead with empathy** in the process to better the product that you are making together.

Development stage

This stage is our time to shine. Do your best to take the lessons we covered in this course: the **techniques and patterns for building accessible UI components**, the **rules and guidelines for creating accessible forms** and take note of today’s wide ranging set of quick wins you can achieve with not that much effort.

Do your best to rely on **native HTML** behaviour as much as possible, **understand the ARIA attributes** that you are introducing and make sure to cover any **keyboard controls** and **focus management** flows if needed.

You now have the knowledge and skills necessary to create accessible websites and applications, so there’s nothing stopping you from doing so.

Testing stage

It is crucial for any team that wants to create an accessible product, that they are fully aligned on **what it means for that product to be accessible**.
In our case this means that we need to make sure that we **include the accessibility requirements** of any given feature - **in the definition of done** for that particular feature. Our testers need to enforce these requirements and they themselves should ideally have the skills to be able to properly test for these accessibility issues.

Code Review & Deployment stages

These 2 stages can be mentioned together because essentially what we want to make sure we have is a **comprehensive automated process** (as well as manual code reviews) that will **catch obvious accessibility issues** before they even get to a QA engineer to test them.

This means including **accessibility linters** both during our development process and in our CI/CD pipeline.

Accessibility linters

There are many linting tools for accessibility purposes out there. Below is a list of a few of the most common ones used today across different Web frameworks.

React

eslint-plugin-jsx-a11y is probably the most common one, used as a plugin for your ESLint configuration file. It analyzes **static code** so that you can be notified of a potential accessibility issue as you are writing your code.
@axe-core/react can be integrated in any React app and it will look for accessibility issues on the actual rendered pages in your browser.

Vue

eslint-plugin-vuejs-accessibility is specifically designed to look for accessibility issues in **`.vue`** files. Works similarly to eslint-plugin-jsx-a11y in that it can be integrated with your ESLint configuration file.
vue-axe-next works by analyzing and reporting on accessibility issues it finds during the runtime of your application. It is based on the axe-core engine.

Angular

@angular-eslint/eslint-plugin-template is Angular's ESLint plugin that lints template HTML files. It has a bunch of accessibility rules that it checks for, which can help us while developing our application.
We can also **manually integrate the base axe-core engine** into our Angular application so that it provides accessibility scanning at runtime.

Svelte

While maybe not the most popular framework out today, Svelte is worth mentioning in this conversation, mostly because their custom compiler has built-in accessibility linting and it will produce warnings when we write inaccessible HTML automatically.

We can add these kinds of linting tools and use them both during our local development, and in our CI/CD pipeline to act as guardrails for any accessibility regressions.

There are of course similar libraries we can add as plugins to our **automation testing framework** so that we even include end-to-end accessibility testing, such as @axe-core/playwright or cypress-axe.

This is what we mean when we say **Accessibility needs to be present throughout the entire development pipeline**, from start to finish.

Browser extensions

Apart from these tools that we can integrate into our development and testing codebases, today there are also a number of browser extensions that provide free and easy accessibility scanning services.
They can be useful to both developers, while they are building new features, as well as to testers, so that they can perform quick sanity checks to make sure that some of the basic accessibility issues are covered.

Starting from probably the most well known of all - Chrome's built-in Lighthouse can perform Accessibility, SEO, Performance and other audits on our applications right there in the browser as we are making it.
Then we have the Axe DevTools extension, which has both a free and a paid version, though I've found that the free version works perfectly well for most cases. They also have their own linter which if you are using **VS Code** - you can install it as an extension. It's called axe Accessibility Linter and it provides amazing insights for developers.

There is also WAVE by WebAIM which is also a great extension to have, as well as many many more - each of them providing quality insights into the state of Accessibility in your application.

These tools are yet another thing you can add in your "toolbelt" and can reach for them while you are doing your everyday tasks and I guarantee you they will make your life so much easier when it comes to tracking down quirky but simple accessibility issues.

No mouse days

The A11y Project started this movement called “No mouse days” as a means to **spread awareness of accessibility** throughout the world and it aims to bring the experience of users that rely on keyboard-only usage of the Web, closer to everyone else.

The rules of “No mouse days” are pretty simple, they require us to designate **1 day in the week where the entire team will just not use a mouse for the whole day** (lunch break and urgent production issues do not apply).

There is actually an NPM package called no-mouse-days which can be added to any JavaScript codebase and it will **hide the mouse pointer** so that developers and testers testing on a local environment are forced to use their keyboard to navigate the application they are building.
The idea is that this can lead us to find potential accessibility issues within our application we didn’t even know existed until then.

Integrating “No mouse days” within a team can definitely bring more awareness and insights into **how our applications work from an accessibility standpoint**. They can also add a level of excitement to an otherwise (perhaps) mundane week of work and break that monotone cycle for a team - if we can manage to convince our team to get on board with this idea.

Knowledge sharing

Last but definitely not least - we have good old **knowledge sharing**.
From **writing comprehensive and understandable documentation**, to **holding accessibility courses** and **training sessions**, we must be the ones that will champion this movement of bringing the culture of accessibility closer to the teams we work with.

Don’t be afraid to share these insights with your colleagues and exchange ideas and opinions on how to tackle accessibility challenges in your future projects.
When you see an **opportunity to add useful documentation** on how a certain functionality should work both in general and from an accessibility perspective - do it, write it down for others to see and be the ones that will set the example on how to **lead with empathy**.

Conclusion

The path to creating a more accessible Web is not possible without collaboration. We need everybody on the same page and striving to achieve the same goal - make the thing we are building accessible to as many people as we can.

It is crucial that we do this with empathy, patience and understanding in our hearts, knowing that what we are doing is really important and at the end of the day, if we can help even one additional person to have a better experience on the Web - it's worth the effort.

Thank you all for joining, listening, asking questions and just being there for the duration of this course - I truly hope you learned something new throughout these couple of days and that you can start using the skills and knowledge you gained, in your next development challenges.

If you have any questions, or just want to connect and chat about the Web, programming or accessibility, feel free to shoot me an email at daniel.shiajkovski@testdevlab.com.

Thank you again everyone, cheers!