Introduction

The goal of this course is to convey the importance of Web Accessibility to developers, from a **practical standpoint**.
We will not focus too much on the formal definitions and legal constraints - although they are an important piece of the puzzle. Instead, we will be looking at real life scenarios from a developer's day-to-day work when it comes to implementing and bringing about a more accessible Web.

What does accessibility mean in practice?

Accessibility is a very broad concept that very basically means - **providing everyone a way to experience life and society, as well as contribute to it, more or less the same way, without any unfair disadvantages**.
This idea of an accessible experience for everyone obviously applies to software as well - and we as software developers have the responsibility and privilege of creating an inclusive environment for any user to enjoy, use and contribute to.

Notice how I am not just talking about **experiencing** software, but also **using it** and **contributing to it** as well. Web accessibility is not just the idea of having people able to understand or read our websites easily - it is also about providing **everyone** the ability to do actual meaningful work with our software, whether that is being able to book a flight or pay one's bills etc.

We do actually mean everyone in the most broad sense. A common misconception about accessibility is that it only applies to disabled people which is only one part of the equation. Non disabled people may have **temporary impairments** (they injured their arm and can't type as fast) or **situational limitations** (they are trying to conduct a meeting in a crowded coffee shop). Furthermore, as people age - their vision, memory and physical abilities deteriorate and they need us to help them use our software with relative ease.

Effectively, what we as developers are facing is an input-output equation in that:
Different people differ in the kinds of **inputs** and **outputs** they use to interact with our software.
Those inputs can include: **keyboards, touchpads, mouse devices, voice commands** etc.
Conversely, the outputs can include: **screen readers and magnifiers, braille displays, high contrast screens** etc.

Our task then becomes quite simple in its definition - **make your UI “survive” the different input-output combinations**.

So let’s see how we can go about tackling this equation and start from the beginning. How do we define these accessibility challenges and what do we need to help our users achieve while using our software?

WCAG & POUR principles

The acronym WCAG stands for **Web Content Accessibility Guidelines**. It’s a shared standard discussing the idea of what it means for a Web site or application to be accessible.
It is not a checklist that is to be blindly followed, nor is it a guarantee that your application will be usable at the end of the day. It is also not a thing we can ignore and just “leave it to the legal department”.
WCAG simply **outlines different principles that need to be supported** in our applications - in a standardized way.

There are 3 levels of accessibility that WCAG differentiates: A, AA and AAA.
Level A is the bare minimum. We shouldn’t in good faith have a website or application out there that doesn’t cover the level A success criteria - if we can help it.
Level AA is the standard that we should aim for most of the time. It covers the essential accessibility principles and also expands these criteria to support a wider range of users with different disabilities.
Finally, **level AAA is kind of like the holy grail**, an aspiration we should all strive to achieve, but realistically it is not always practical or possible to do so.

The guidelines mentioned in WCAG are categorized into 4 principles:

- **Perceivable** - users must be able to **perceive** our software - it can’t be invisible to them
- **Operable** - users must be able to **use** our software regardless of their input methods
- **Understandable** - users must be able to **understand** our software - it must be predictable and easy to comprehend
- **Robust** - our software must stand the test of time - it must support legacy platforms as far as we can reach back

A good mnemonic device to remember these principles is using the **POUR** acronym.
These principles of course are further divided into **success criteria** and you can check each and every one of them out here.

We will obviously not be going through each and every one of them but we will definitely mention the relevant ones as we go along.

Accessibility pillars

If we had to boil down Web accessibility into 3 pillars - 3 north stars if you will - that we need to follow, they would be the following:

1. **Keyboard controls** (Operable principle)
2. **Color contrast** (Perceivable principle)
3. **Labeling** (Understandable & Perceivable principles)

Keyboard controls

Very basically, as developers of high quality accessible UIs, it is our responsibility to make each component able to function **with and without a mouse pointer**. This means that anything users are able to do with a mouse, they should be also able to do using only the keyboard.
We need to make sure that our components are **keyboard accessible**.

Color contrast

Perhaps one of the most obvious but surprisingly often overlooked ideas is the idea of having good color contrast across your entire application. This applies to **any interactive controls (buttons, inputs, menus etc), as well as text, images and any and all important graphical elements on the screen**.
This concept of color contrast is often ignored at the **design stage** of the development pipeline, when well meaning ideas of creating good looking UIs fall short of having sufficient contrast. We will look at such scenarios shortly.

Labeling

Last but definitely not least, we have the concept of labeling. Everything that's on the screen and can be useful to the user - either to read or interact with - needs to be properly labeled. This includes of course any kinds of **form controls and inputs (text inputs, select menus, checkboxes etc)** but also the often forgotten **icon buttons**, as well as things like **images**, **informative graphical elements (ex. A checkmark icon instead of "Passed" text)** etc.

There are of course many other concepts that are included in the process of creating an accessible Web site/application, but the bulk of it is included in those 3 "pillars of accessibility". If you get those 3 things right - you will have solved most of the critical accessibility issues in your application.

Screen Readers

We really cannot go further without discussing one of the most important assistive technologies out there - screen readers.
They are very powerful pieces of technology (both hardware and software) that essentially **parse the content of any application** and read it out loud using **text-to-speech**.

A common misconception about screen readers is that **only blind people use them**, when in actuality they are used by a number of different people with varying degrees of disabilities if any. They are an amazing tool for both all kinds of users to be able to **use and contribute to** the Web (as well as other spaces like mobile and desktop applications), as well as software developers and testers to be able to improve the accessibility of their software.

There are many screen readers out there, each mostly used in combination with a specific platform or a browser.
We have JAWS and NVDA which are mostly used on Windows, as well as Windows Narrator which is built-in to Windows machines. Then we have VoiceOver for MacOS and iOS devices. There's also TalkBack for Android devices, Orca for Linux machines etc.

As far as browsers go, there are also common combinations recognised such as:

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

Ideally we would want our websites and applications to **work perfectly across all screen readers in combination with all browsers**, but this is not very realistic and often not at all practical, as there are many combinations we would need to go through in our testing to make sure they all work perfectly, which takes up a lot of time.
What's more, not all screen readers work the same way on all platforms and browsers, so there may be some edge cases where a certain screen reader (ex. VoiceOver) doesn't work as expected on Firefox for example, while it works perfectly on Safari.

Instead what we want to do is **prioritize the most common combinations** for our use case. Now, while we cannot really find out what exact screen readers our users are using, we can figure out what platforms and browsers they are using with basic analytics tools and figure out the most common screen reader used with them.

WebAIM (Web Accessibility in Mind), are a non-profit organization from Utah and they have amazing resources on accessibility, including **surveys in which they collect information from real users** on their usage of screen readers.
If we cannot collect this information ourselves, these surveys are a great place to get an idea of what screen readers we might want to prioritize for our accessibility testing.

Performing manual screen reader accessibility testing is probably the closest we can get to actually testing our product's accessibility with real world users. While **real world user testing is always the ideal way to go**, sometimes this is not possible due to **financial or time constraints**, so adding screen readers to our team's testing workflow is a must in that case.

Throughout this course, we will see different pieces of UI, components, forms etc. It is always a good idea to **test out our UI using screen readers when developing**, so with that in mind - each time we implement a component or some kind of accessible UI down the line, try to use a screen reader to go through it and make sure the read outs make sense and that it is clear what you are doing and where you are on the page at all times.

This course will use VoiceOver as the default screen reader for showing demonstrations, but you can use whichever one you want.

Demo: Screen readers introduction and cheat sheets

Common pitfalls

Design team vs Development team

We touched on this part previously when talking about the issues that can occur during the **design stage** of the development pipeline, so let's look at it in a bit more detail.
Oftentimes, designers - having very well intentions - may overlook or completely ignore fundamental accessibility rules in the name of creating good looking, slick UIs.

In such cases, we need to strive to build a **culture of constant communication and feedback with empathy**, where we as developers can be involved in the design decisions with our knowledge of accessibility principles, to make sure they are not ignored or violated in any way. It is important we do this with **empathy in mind**. Remember, accessibility is not the most popular topic on which everyone is informed, so sometimes we need to be the ones that **share this knowledge** with the rest of the team.

Animation vs Reduced motion

It is not uncommon for our design team to present us with a cool looking animation that needs to be added to our website. Maybe an infinite scrolling list of client logos or some funky cool looking movement in the hero banner, like this one.

Now, these kinds of features add a level of liveliness to our pages, but they can often be disorientating and downright distracting or even overwhelming to a lot of users.
People that suffer from vertigo, have ADHD or just have a high sensitivity to motion and other similar groups will not appreciate all of that movement no matter how cool it looks.

In this case, we need to discuss alongside the design team - the possibility of adding a **pausing mechanism to our infinite animations**.
In fact the 2.2.2 Pause, Stop, Hide success criterion is put in place for scenarios such as these ones. We need to provide a way for users to control these automatically playing animations and stop them if they deem them distracting.

Another thing we can do here is use the CSS query for **`prefers-reduced-motion`** to not display any kinds of potentially distracting animations (pause them or stop them altogether). This kind of solution refers to the 2.3.3 Animation from Interactions success criterion that talks about users not being harmed by distracting animations.

```css
CSS
@media (prefers-reduced-motion: reduce) {
 .animating-ticker {
    animation: none;
  }
}
```

The goal is to keep these kinds of distractions to a minimum and having these conversations with our designers are a crucial part of doing so.

Demo: prefers-reduced-motion usage

(Missing) Focus indicators

Eliminating focus indicators just for the UI to “look pretty” is definitely not an option. Still, there are many teams out there that do this and it is our job to call out this behaviour and prevent it from happening in our own applications.

The reason for this should be quite obvious - users that rely on their keyboard to navigate our website need to know where they are on the page, and one of the ways this is accomplished is through **showing a focus indicator ring around the element that is currently in focus**.

The 2.4.7 Focus Visible success criterion deals with this very scenario.
Now, there are cases where the stylistic look of our website is such that just simple outlines around our interactive elements are not satisfactory (they just clash with the look and feel of the website).

There are alternatives to using the simple default outline to denote an element receiving focus, for example we can use a **box shadow** around the element with a **small blur radius**. This way we can introduce some small transitions to make our elements feel more “alive” while also providing a clear focus indicator.
Remember, **outline ≠ focus indicator**. We can have pretty much anything be a focus indicator to our elements (a box shadow, a pseudoelement like `before` or `after`, maybe even a background change etc) - **as long as we are consistent with that approach** and of course make sure that your custom focus indicator is **clearly visible to users**.

```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;
  }
}
```

Demo: Focus indicator variations

Quiz

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

- **focus** - a general pseudo-class that gets applied to an element whenever it receives focus, **no matter what triggered it**
- **focus-visible** - a more specialized pseudo-class that gets applied whenever **browsers' own heuristics** determine that a focus indicator is most needed. This is usually when users focus an element using their keyboard or any kind of assistive technology.

The pseudoclass **focus-visible** is much better suited for styling the focus indicators on elements such as buttons, as (most likely) we don't really want that indicator to appear when we click on them for example.

These are just some of many discussions we need to have alongside our design team to make sure we **catch obvious accessibility issues as early as possible**.

As we will see throughout this course - **the concept of accessibility needs to be present throughout the entire process of creating a software product**, so collaboration between the designers and developers is crucial.

Custom vs Native implementation

Whether it's the ego or pride in us, or just the desire to impress folks, we developers sometimes love to reinvent the wheel.
There have been countless examples of developers **implementing a custom component from scratch instead of just using the native built-in HTML components** which can do the job just as well - and often better, especially from an accessibility perspective.

Of course this is not a set-in-stone rule, as sometimes the design specifications are such that we really do need some custom behaviour, but most of the time - we can get away with using a native HTML feature. The best example for this is the simple **button**.

Custom buttons

Raise your hand if at some point you have implemented or seen someone implement a button using a **<div>** or a **<span>**.

I will be the first to raise my hand because starting off - I too fell into the trap of seeing a custom button-like functionality and immediately saying bye-bye to the native <button> HTML element. And while again, it is true that some requirements may ask for a custom implementation, most of the time **we just need to use a button.**

The reason for this is very simple - HTML buttons (and native elements in general) just work. They work how we want them, they work exactly how we expect them to and they are accessible out of the gate.
We don't need to worry about **handling proper mouse click events, as well as keyboard "Enter" and "Space" press events**. We don't need to worry about **the semantic meaning of the element** etc.

The hard truth that I too had to accept is that most of the time, it's just a matter of having the skills to **style the button** according to the specs - while perhaps sprinkling in some custom behaviour via JavaScript.
It's almost never a good idea to go with a custom implementation if you can get away with using a native HTML element.

With that being said, if you must implement a custom button (for whatever reason) - you need to make sure you have your bases covered by implementing all the features that a native button provides. Let's go against our better judgement now and **implement a button using a <div>**.

To start off, we need to make sure that assistive technologies such as screen readers can know that this element is indeed a button. For that reason, we have to give it the proper **role** of **"button"**.

```
HTML
<div role="button">Click me</div>
```

Next, we need to make sure that our custom button **can receive focus** (is part of the regular focus order on the page). To do that, we can add the **tabindex** property with the value of **"0"**.

```
HTML
<div role="button" tabindex="0">Click me</div>
```

Quiz

What does a **tabindex** of **“-1”** mean?
Elements that have a tabindex of “-1” are **removed from the regular focus order on the page**. This means that **users cannot focus on those elements with their keyboard**.
However, it is important to note that this does not mean these elements **can’t receive focus** - because we can still do that **programmatically from our code**.

There are actually very good use cases for this specific attribute-value combination which we will look at more in-depth during the next lesson. Okay, back to our custom button.

Well, we ought to be able to click it right? A button is no good if it doesn’t do anything when you click it. So let’s add a click handler.

```javascript
const customButton = document.querySelector('div[role="button"]');

customButton.addEventListener('click', () => {
 alert('Hi!');
});
```

Okay, we are getting close, but still we are missing our **keyboard accessibility**. What if I wanna press the button using the “Enter” or “Space” keys? Well, we would need to add another event listener for that.

```javascript
const customButton = document.querySelector('div[role="button"]');
const onClick = () => alert('Hi!');

customButton.addEventListener('click', () => onClick());

customButton.addEventListener('keyup', (e) => {
  if (e.key === ' ' || e.key === 'Enter') {
    e.preventDefault();
    onClick();
  }
});
```

A couple more things before we really have an actual working button - **generic labels like "Learn more", "Click here" etc. are not very accessible and won't be able to tell users (on their own) what they (links, buttons) actually do**.
Sure, if you look at that page visually maybe you get some context around that button that will explain what it does, but for a keyboard only user that has just tabbed onto your button - this label means nothing.

For this reason, we also need to add an **accessible label** to the button. We will talk more in depth about labeling later but for now, let's just add a regular **`aria-label`**.

```html
HTML
<div role="button" tabindex="0" aria-label="Prompt an alert to say Hi!">
  Click me
</div>
```

And last but not least, we need to style it up with some default styles that are given to us for free when we have regular buttons - **hover, active and focus states**.
We will also add a "pointer" cursor to make it really obvious that this is a button.

```css
CSS
div[role="button"] {
  /* Basic styles for a button here... */

  cursor: pointer;

  &:hover {
    background-color: #eee;
  }

  &:active {
    background-color: #bbb;
  }

  &:focus-visible {
    outline: 2px solid blue;
  }
}
```

…and now we are done.

I hope that we now have a shared appreciation for all of the things a simple HTML <button> element gives us for free without us having to worry about it.

The moral of the story is to definitely **use native built-in functionality whenever possible** and only reach for custom logic when you absolutely have no other option left.

Landmark elements

The same thing can be said about HTML landmark elements - there is no need to clutter our websites with countless <div> elements all over, especially when it comes to the main sections of a page.

The **<main>**, **<nav>**, **<aside>**, **<section>**, **<header>** and **<footer>** elements were introduced in HTML5 for the specific reason of giving our pages more **semantic structure and meaning**.

Too often these elements are ignored and replaced with a simple <div> element instead. Sometimes, those <div> elements have a \`role\` attribute like **\`role="main"\`** but that's just more code instead of just writing **<main>**.

Each landmark element has a corresponding landmark role::

Element	**Default Landmark Status**	**Role (when a landmark)**
`<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`

Main

Pretty self-explanatory - used to wrap the main content on your page. We should **only use one main element per page**.

Nav

Used to denote sections on a page where **navigational links** are located. This can be your main top-level navigation, a sitemap inside the footer etc.

It is important to **label your <nav> elements if you have multiple instances on one page**, in order for assistive technologies to be able to distinguish between them.

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


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

Aside

Used to denote **complimentary sections on our page, like sidebars for example**. Same labeling rules apply as for the <nav> element - **if we have multiple <aside> elements on one page - we need to label them** so that they are distinct from each other.

Section

The <section> element plays an interesting role.
On its own, it’s just a grouping element with no semantic meaning, however if we want it to become a landmark element (so called **region landmark**) we need to **label it** using either `aria-label` or `aria-labelledby`.

```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>
```

Header

Generally used at the **top level - the "banner" of your application**. This is where usually the top level navigation is located, along with some branding and user-specific information.
There are also use cases where the <header> element is used inside a <section> or an <article> - in these cases we **must not add a role of "banner" to that kind of header**. The "banner" role is only reserved for the top-level header and **we should only have one "banner" landmark per page**.

Footer

Generally used at the bottom of your page, although similarly as the <header> this element can also be used as the last child of a <section> or an <article> element.
Same rules apply in such situations - **we can only have one element with the role of "contentinfo" per page**.

Interestingly enough, sometime before 2016/17 it was still necessary to add the landmark role attribute to make sure that screen readers would read landmark elements correctly - even though they were semantic landmark elements.

Luckily today we don't have to do this anymore and can just use <main> and <header> etc and will have all major screen readers and modern browsers covered.

If however you are working on a very legacy product that needs to **support "old" technology (legacy browsers, screen readers etc)**, keep this in mind and use **both the landmark element and its associated role.**

Missing keyboard accessibility

We need to make sure that every interaction (there are of course rare exceptions) that users can do with a mouse device - is able to be accomplished using only a keyboard.
Usually keyboard accessibility is not implemented correctly within **custom implementations of components**.
Built-in native components have keyboard accessibility taken care of, so the moment we start to build our own custom component - it is our job to take care of that keyboard logic as well.

Depending on the component, we need to make sure we cover:

- Keyboard press events ("Enter" and "Space" for buttons for example)
- Focus management (will talk about this more during the next lesson)
- Navigation throughout the component (Arrow keys, Tab etc)

During our next lesson, we'll spend a lot of time talking about how we can make sure we don't miss anything related to keyboard accessibility when building out custom components.

Insufficient contrast issues (focus indicators)

The issue of color contrast is its own broad topic - in fact, it's one of the three pillars that we mentioned at the beginning of this lesson.
For this pitfall, we will focus on a small but very important and often overlooked part of color contrast issues - focus indicators.

Good visible focus indicators are crucial to the visibility of the interactive elements on our websites. Users (and especially keyboard-only users) need to be able to **clearly see** what element they are focusing on, at any given point in time.
Our job as developers is to make sure that not only these focus indicators are present when an element receives focus, but also that **they are clearly visible and distinguishable from their surrounding environment**.

This again calls back to the **need for collaboration between developers and designers** because this issue can be easily solved during the design stage, if we are careful not to miss it.
Common examples of bad focus indicators include:

- Indicators with insufficient color contrast to the element/environment
- Indicators that are too thin and not noticeable at 100% zoom
- Indicators that are the same color as the background of the interactive element and are right next to each other

This third one is a very prominent issue even throughout various component libraries.
A good strategy to battle such scenarios and make sure both your buttons and their focus indicators remain clear and visible throughout the entire layout of your application is to **use a focus indicator with a bit of offset from the button**.

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

This way the focus indicator will be **visually separated from the button itself**, making it more distinguishable.

Using too much/wrong ARIA

As the Web Accessibility Initiative's ARIA authoring practices guide says: "No ARIA is better than bad ARIA."
Before we throw in a bunch of ARIA attributes on our components, **we need to be aware of their purpose** and ask ourselves if they are truly necessary.

More often than not, there are better semantic alternatives (like native built-in elements instead of custom components), but also simpler solutions to a problem than just covering it up in ARIA attributes and hoping for the best.

```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>
```

We will look at some appropriate ARIA attributes that are used alongside some common components during the next lesson.

Relying on third party libraries for everything

This pitfall obviously doesn't just concern Web accessibility but the entirety of Web development. We have admittedly become **too trigger happy to include a library in our project** when we need some functionality which we cannot immediately figure out how to implement on our own.

Now there are obviously **many scenarios where using a third party library is very appropriate**, as we don't want to reinvent every wheel out there. There are comprehensive solutions and battle tested libraries that help us out and do a lot of heavy lifting and are valid choices to reach for in different scenarios: **Routing, Authentication, Component libraries** etc.

However, there are also situations where we need some **specific custom logic or functionality, maybe in one or two places in our application** and instead of putting in a bit of effort to try and figure out how to implement it ourselves - we reach for a third party library.
Again, there isn't an inherent problem with doing that, but there certainly is a bit of danger.

When we include a third party library in our application, we are **putting our trust in the author of that library** - hoping that they have the perfect solution for us and haven't messed anything up themselves.
Combine that with the fact that **accessibility is a very tricky thing to get right for custom components** - which is a common use case for people reaching for third party libraries - we have ourselves a **ticking time bomb of accessibility issues**.

I want to again emphasize that there are **absolutely libraries out there that provide comprehensive accessible components, such as Base UI, Headless UI, Radix UI** etc.
However, if we only need a custom autocomplete component, or maybe a modal or something similar - and not a fully fledged component library/design system - we can definitely spend some time to try and implement it correctly on our own as opposed to installing a large package with all kinds of functionality we might not need.

This approach to Web development and especially accessible UI development will give us **more control over the accessibility of our components** and we won't be dependent on another person’s custom implementation of a search box breaking because of a bug they made that has nothing to do with us.

The next lesson will cover this exact topic - we will go over the **implementation of some common UI components** and hopefully by the end of it, we won’t be as reliant on other people’s code anymore.

Automation testing is NOT enough

We haven’t talked about the testing part of Web accessibility yet and while we will cover it a bit more during the following lessons - we must make one thing very clear at the start - **automation testing is absolutely not enough when it comes to accessibility**.

There are different numbers thrown out there as it pertains to the **percentage of accessibility issues** that different automated tools can catch.
Possibly the most popular tool in this space - Dequeue System’s axe-core engine claims to catch up to 57% of all accessibility issues - which of course is not enough.

Automation is a great tool to have when testing for accessibility issues, especially for very obvious violations and best practices, but it fails to catch very important issues in our applications:

Meaningful labels/alternative text

An automation tool can notify us if we have a missing label or alt-text on an image. They cannot however tell us if that label/alt-text that we put there **makes logical sense or is meaningful in any way.**

We need to manually confirm that all of our labels (visible or not), as well as descriptions, alternative texts etc. make sense within the context of their surroundings.

```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">
```

The above HTML will pass an automation test with flying colors, but imagine you are a user that uses a screen reader and comes across this - you would be pretty disappointed to say the least.

Keyboard interactions

One of the pillars of accessibility is not at all covered by automation tools, that's a shame but it makes sense. No tool can test out all of your custom widgets and components to make sure that the **focus order makes sense, that all keyboard events are handled properly according to the role of that element** etc.
We of course need manual testing to be able to cover such scenarios - and keyboard accessibility is obviously the primary candidate for that.

Visual/Cognitive issues

Automation tools cannot tell us whether we have **distracting animations, confusing or overwhelming text content, issues with zoomability and responsiveness** - we need to manually go through and test everything out ourselves.

Now it is important to note that there is **not always a malicious intent behind teams that rely on automation testing to cover their accessibility issues** - and I have worked for some teams that do that. Sometimes it's the **lack of knowledge and information sharing**, or perhaps sly marketing on the part of the automation tools that can put teams in this position.

This is why we need to approach the issue of accessibility throughout our entire team with empathy and educate people on this very overlooked topic, to make sure that all of us are on the same page and working to accomplish the same goal - making the Web more accessible.

Manual accessibility testing is not an easy task and depending on the application's complexity - it does take quite a bit of time to perform correctly. Be that as it may, **we absolutely need to make sure that we are utilizing it** as another tool to flush out as many accessibility issues as we can.

Conclusion

Web accessibility is a goal that we should strive to achieve for many reasons, not the least of which being that **it's just the right thing to do**.
We as developers must be equipped with as many tools in our toolbelt as possible, so that we can tackle the technical challenges that Web accessibility presents.

Today we established the 3 pillars of accessibility: Keyboard controls, color contrast and labeling - which present the bulk of issues that we need to deal with when improving the accessibility of any website or application.
We also talked about some of the most common pitfalls that any team can fall into when building accessible software and saw various solutions on how to avoid them in the future.

As mentioned multiple times during this lesson - the next lesson will be all about how we can **implement some of the most common UI components with accessibility in mind**. We will talk about the challenges each of those components pose and how to deal with them accordingly. We will also look at some **common UI accessibility patterns** that can arise while building out those components and much more.