Develop Accessible Web Apps with React

Transcripts for Erin Doyle (https://egghead.io/instructors/erin-doyle) course on egghead.io (https://egghead.io/courses/develop-accessible-web-apps-with-react).

Description

A large number of people are unable to use the web effectively due to an impairment or disability.

As developers there are several tools and techniques we can use to make our web applications accessible, ensuring a great user experience that includes everyone.

We all agree that Accessibility is important. However, it's a broad landscape and can be overwhelming figuring out where to start!

Erin Doyle is an expert in creating accessible React applications and has developed a course that will give you a concrete process for testing, refactoring, and building your applications with accessibility in mind.

After this course, you'll have a jump start on auditing and fixing accessibility issues in your applications and gain a better understanding of your target users and how to approach your web app design from their perspectives.

Check out these community notes for this course on Github (https://github.com/eggheadio-projects/develop-accessible-web-apps-with-react-notes).

Set up ESLint to Audit Accessibility Issues in React

Instructor: [00:02] If you have not yet installed `eslint`, then you will need to do that first, and then include the plugin. It's `eslint-plugin-jsx-a11y`. I am going to save it to our development dependencies.

```
npm install eslint eslint-plugin-jsx-a11y --
save-dev
```

[00:23] Now that that's installed, we need to configure it. You want to go to your `eslint`, config. Mine is in `eslintrc.json` file. We will add a section for plugins. Again, it is `jsx-a11y`.

eslintrc.json

```
{
    "extends":[
        "react-app"
    ],
    "plugins":[
        "jsx-a11y"
    ]
}
```

[00:44] Now if we want to configure the `rules`, we can do that here. You'll just specify whatever rule it is you want to configure. For instance, if I wanted to set `alt-text` to `warn`, I could do that here.

```
{
    "extends":[
        "react-app"
    ],
    "plugins":[
        "jsx-a11y"
    ],
    "rules": {
        "jsx-a11y/alt-text":"warn"
    }
}
```

[00:59] If, instead, you want to extend the recommended or strict set of rules, we can do that as well. We would get rid of our specific rule definitions and add to the `extends` section, `plugin:jsx-a11y`, and we'd either do `strict` or `recommended` for recommended mode.

```
{
    "extends":[
        "react-app",
        "plugin:jsx-a11y/recommended"
    ],
    "plugins":[
        "jsx-a11y"
    ]
}
```

[01:23] Let's go ahead and get ready to run it. If you have not done so previously, we can create a script for the linter. We would add to our `scripts` section in our `package.json` file. I am going to call this `lint`. We run it with the `eslint` command, and you specify the directory that you want to lint. For my project, it's called `src`.

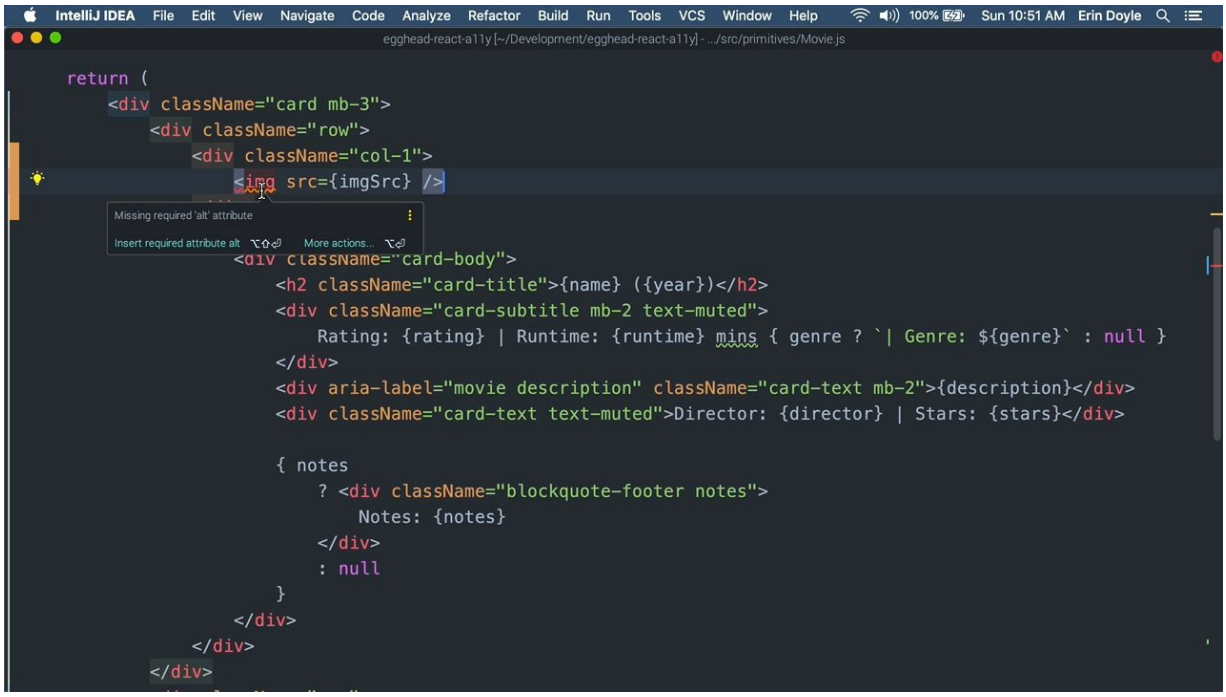
package.json

```
"scripts"{
  "lint": "eslint src"
}
```

[01:51] Another really helpful thing we could do is if every time we run `test` we want to run the linter first, we can add a `pretest` script that will run our linter.

```
"scripts"{
  "pretest": "npm run lint",
  "lint": "eslint src"
}
```

[02:11] Let's go ahead and try this out. I've got a line of code that should trigger the linter and give us the error. Right here, we have `img` and it is missing an `alt` attribute. You'll also notice I've got some squiggly lines here.

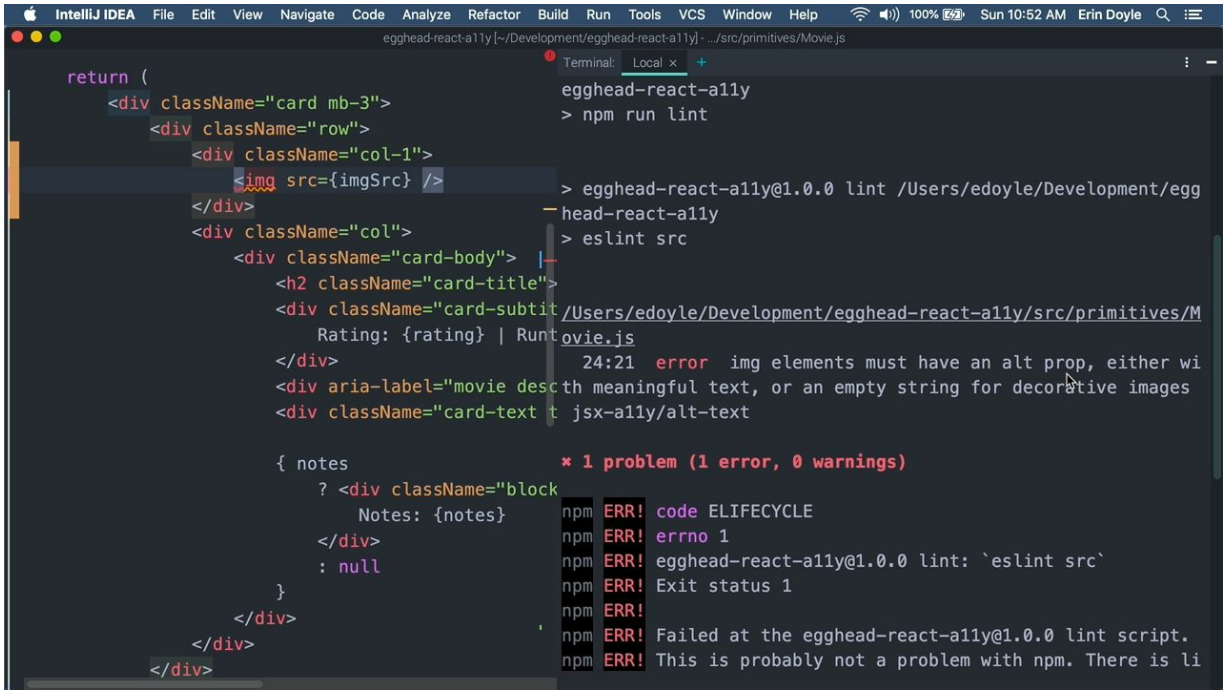


[02:27] Some IDEs will integrate with your `eslint` config and will actually visually show you when there is a finding. We can already see if we hover over that that it knows I am missing my `alt` attribute.

[02:40] Anyway, if you are not using an IDE that integrates with `eslint`, you can continue to run it from the command line. No problem.

[02:47] This should give me an error. Let's go ahead and run it, and see that that happens. I am going to run the `test` script, just to show it will run my linter before it runs tests. I don't have any `test`. We're just going

to see that the linter will show us the error. There it is.



[03:06] Here is my error, `img elements must have an alt prop` That's what we're expecting. If I go ahead and fix that real quick, and let's run it again. This time I am just going to run the lint script directly.

[03:29] Let's see that my finding was fixed, and it was. Here we go. Because the command line isn't showing any more output, that means there were no more errors found.

Use react-axe to Audit Accessibility Issues at Runtime during Development

Instructor: [00:00] To begin using `react-axe`, first we need to install it. We can do so by running `npm install`, we're going to save this to our development dependencies, and it's called `react-axe`. Now we can see that in our development dependencies here in our `package.json` file.

package.json

```
"devDependencies": {
    "react-axe": "3.3.0",
  },
```

Now we need to initialize it.

[00:26] You're going to want to go to whatever file it is that starts up your application. For me and my project here, it's in `index.js`. We want to get `react-axe` initialized before we render our first component in our react application.

[00:41] We want to make sure that `react-axe` is only running in our development environment, because it's going to be logging errors to the console, and it also has a little bit of a performance hit, so we only want to run it in development, not in our `production` application.

[00:56] We can make sure it only runs in our development environment by wrapping it in a block where we check the environment. We're going to check that this is not `production`. Now inside our block we will dynamically import `react-axe`, and we will pass to the `axe` constructor the `React` and `ReactDOM` objects.

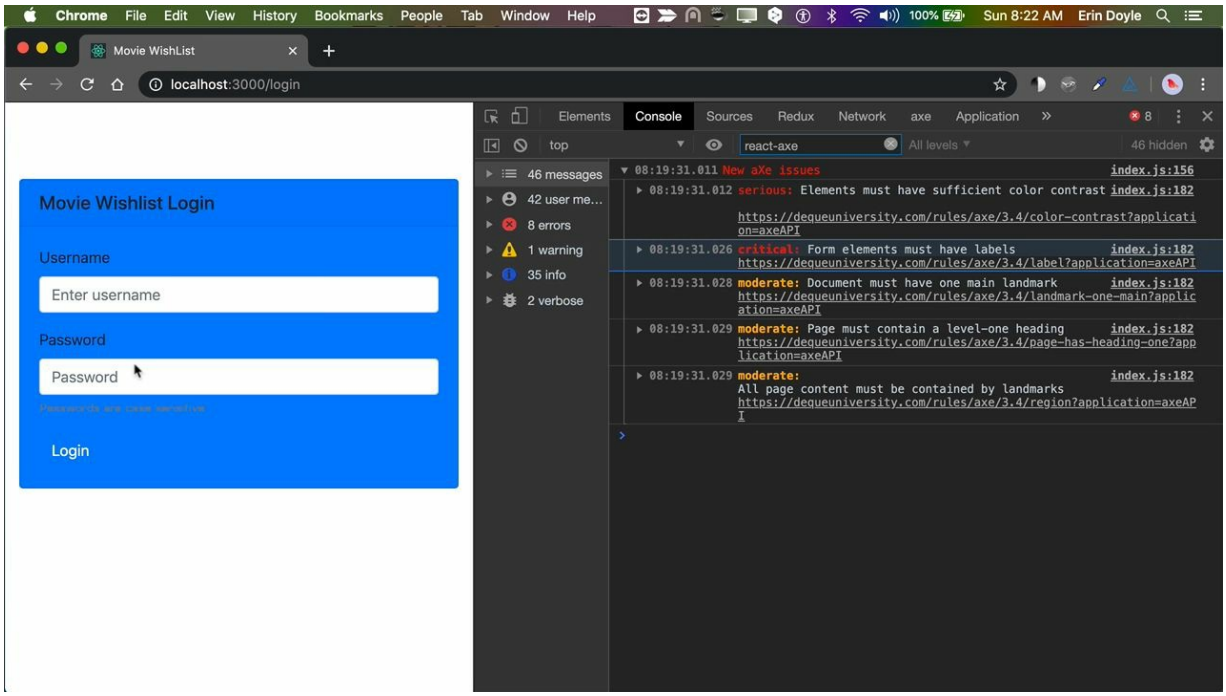
index.js

```
if (process.env.NODE_ENV !== "production") {
  var axe = require("react-axe");
  axe(React, ReactDOM, 1000);
}
```

[01:25] Then finally, this third argument is a timing delay in milliseconds for how long `react-axe` will wait after a component renders, before it will begin analysis again. We're going to go with `1000`. Now that we've got `react-axe` initialized, we can go ahead and use it.

[01:44] Here's a sample application I have with some accessibility issues baked in. I've got Chrome dev tools open over here on the right. Note that `react-axe` works best with Chrome. It's noted to work OK with Firefox and Safari. It's highly recommended that you use it with Chrome at this time.

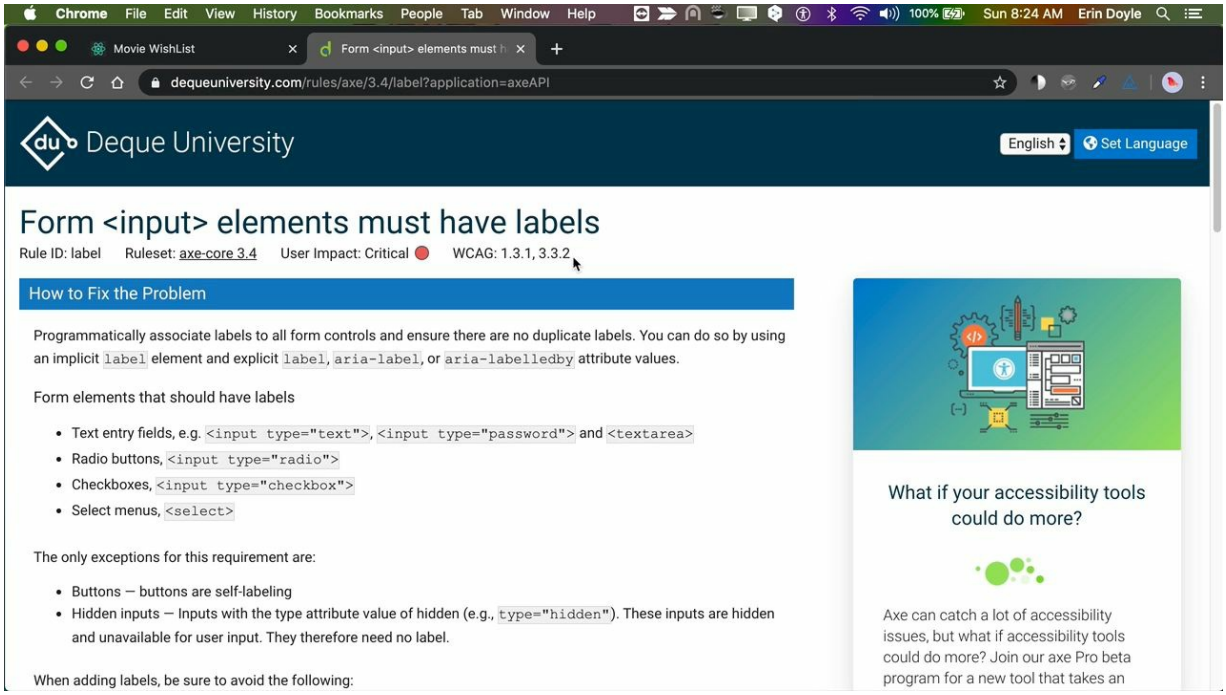
[02:02] You can see I have a number of issues being reported in the console now by `react-axe`. They're ordered by their severity, so we have `serious`, `critical`, `moderate`. A really nice feature of `react-axe` is that it dedupes findings.



[02:18] For instance, here, I have two form inputs that have the same issue, and they're only reported here once keeping the console logging nice and clean. When I hover over each of these, it goes ahead and highlights the elements on the page.

Highlight (https://res.cloudinary.com/dg3gyk0gu/image/upload/v1576545911/transcript-images/02_react-use-react-axe-to-audit-accessibility-issues-at-runtime-during-development-highlight.jpg)

[02:33] If I want to learn more about this finding, they've got links for each of the findings where you can go and learn about the issue. Here they list the severity level which WCAG standard is being violated with this finding, info on how to fix the problem, why it's important, and more information and links to more resources on this issue.



[03:02] This is a great way to learn more about accessibility and how to solve accessibility problems. If I resolve this issue here, now we can see that that `Critical` finding has been resolved and it's no longer being reported in the console.

[03:20] Now if we want to configure `react-axe` to behave differently than the default, we have the ability to pass in a configuration object where we can add new rules, modify existing rules, modify how it reports the console, all sorts of different things we can do to configure `react-axe` further. We can do this by creating a `config` object.

[03:45] If, for instance, we want to modify the rules, we add our `rules` property which is an array, and it takes an object for each rule we want to either provide or modify. For this example, we're going to modify the `radiogroup` rule.

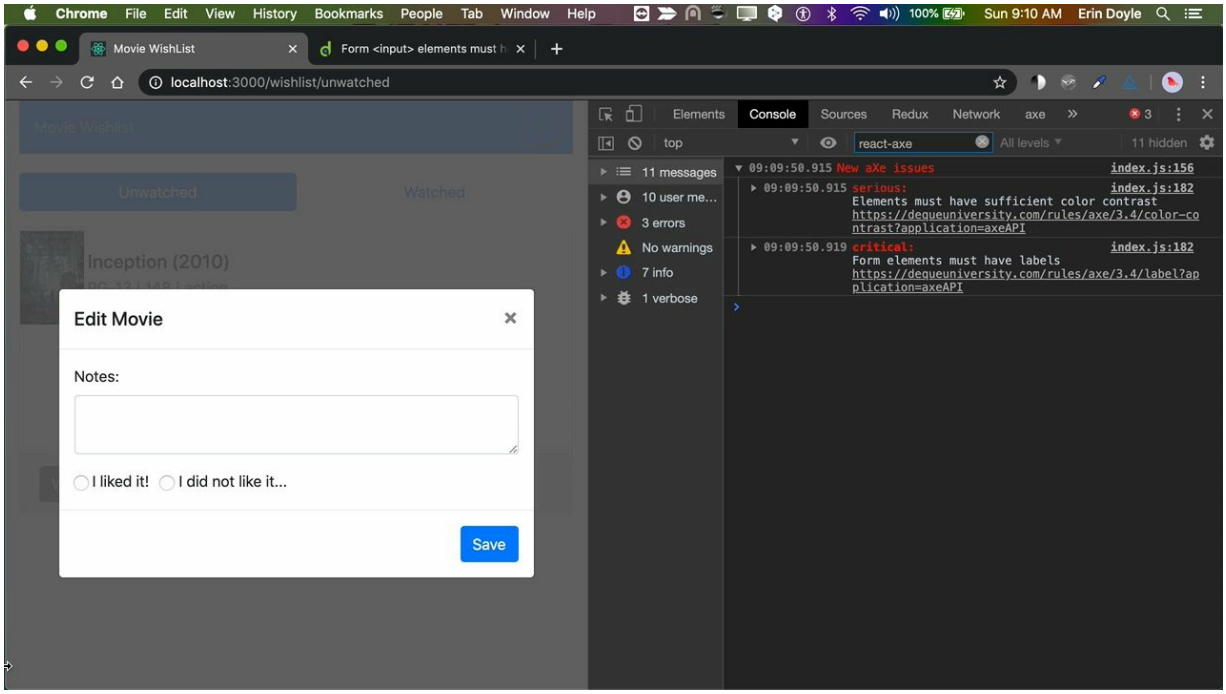
```
var config = {
  rules: [
    {
      id: "radiogroup",
      enabled: true
    }
  ]
};
```

[04:04] This rule is disabled by default so we're going to enable it. Once we've created our `config` variable, we can go ahead and pass that as the fourth argument to the `axe` constructor.

```
if (process.env.NODE_ENV !== "production") {
  var axe = require("react-axe");
  axe(React, ReactDOM, 1000, config);
}
```

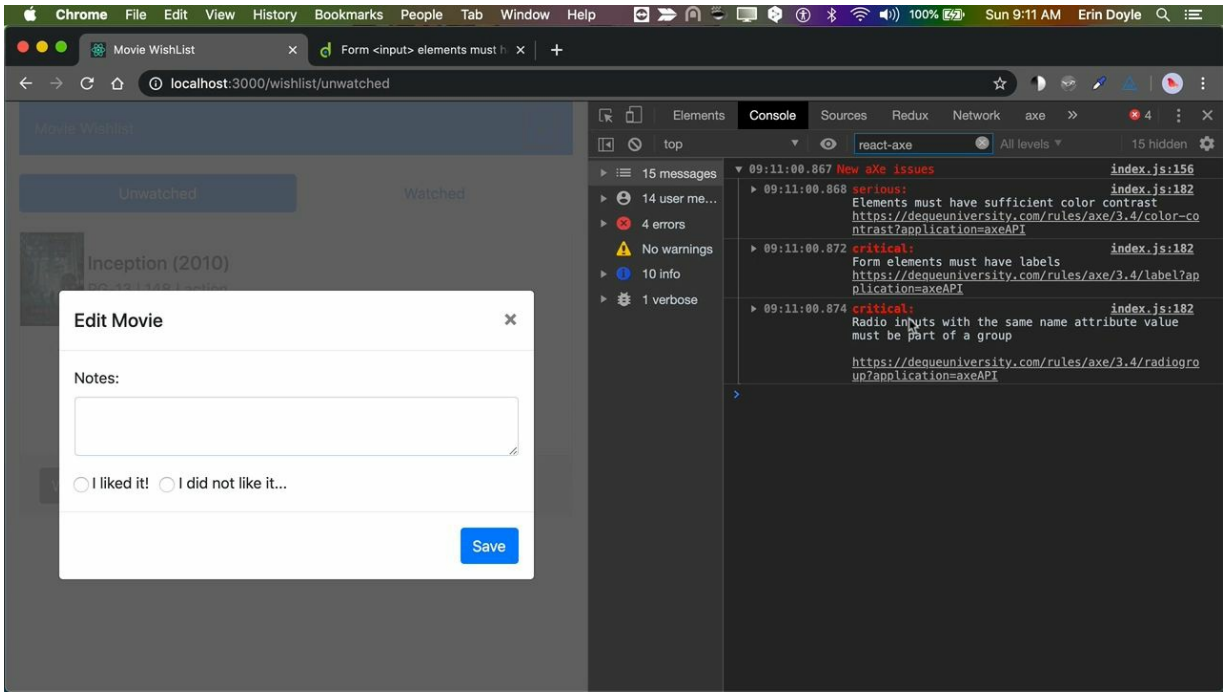
Let's go ahead and change this back to `false` so we can see the before and after.

[04:21] Here we have `modal` in our sample application that has a couple radio buttons. We do have a couple other findings, but one thing we don't have, is whether these radio buttons are in a radio button group.



Let's go ahead and turn that rule back on.

[04:38] Let's change `enabled` to `true`, now we can see that finding being reported here in `react-axe`'s logging output. There it is, pointing to our radio buttons.



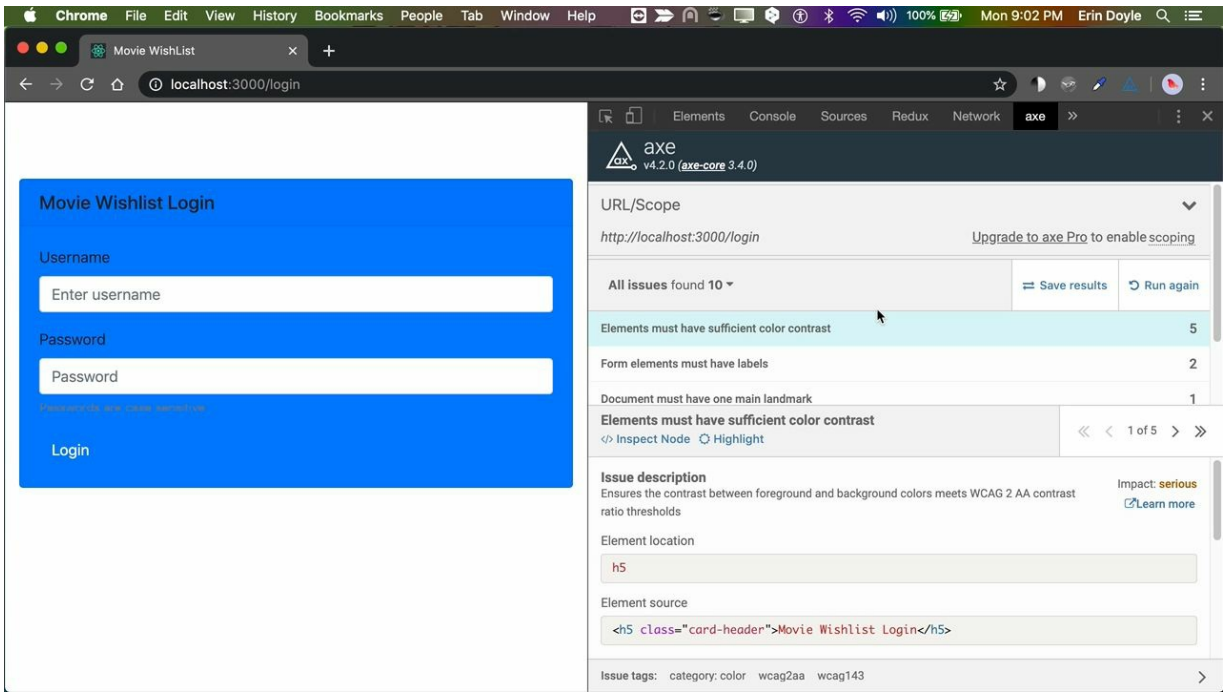
Once again, if we want to know more about this finding and what we can do to fix it, we can just click on the link provided in the console. If we go ahead and fix this issue, we can see that that finding is no longer being

reported.

Use the axe Browser Extension to Audit a Web Page for Accessibility Issues

Instructor: [00:00] In order to install the axe browser extension, you'll need to either look in the Chrome Web Store for Chrome (https://chrome.google.com/webstore/detail/axe-web-accessibility-tes/lhdoppojpmngadmnindnejefpokejbdd), or in the browser add-ons for Firefox (https://addons.mozilla.org/en-US/firefox/addon/axe-devtools/), depending on which browser you wish to use. You just install the extension in the browser of your choice.

[00:26] Now that I've installed the extension in Chrome, let's go ahead and use it. Here, I have a sample web application with some accessibility issues baked in. If I open Chrome DevTools, I now have a tab for axe. If I select that and click the Analyze button, it will list whatever accessibility issues it's found for the current page.



[00:53] If you select an issue, you can choose to highlight each of the elements that have been found with that issue. Within the issue description, you can learn more about what the issue is. It shows you the source of the issue for the selected element. It lists the severity level. Here's a link to learn more about this issue.

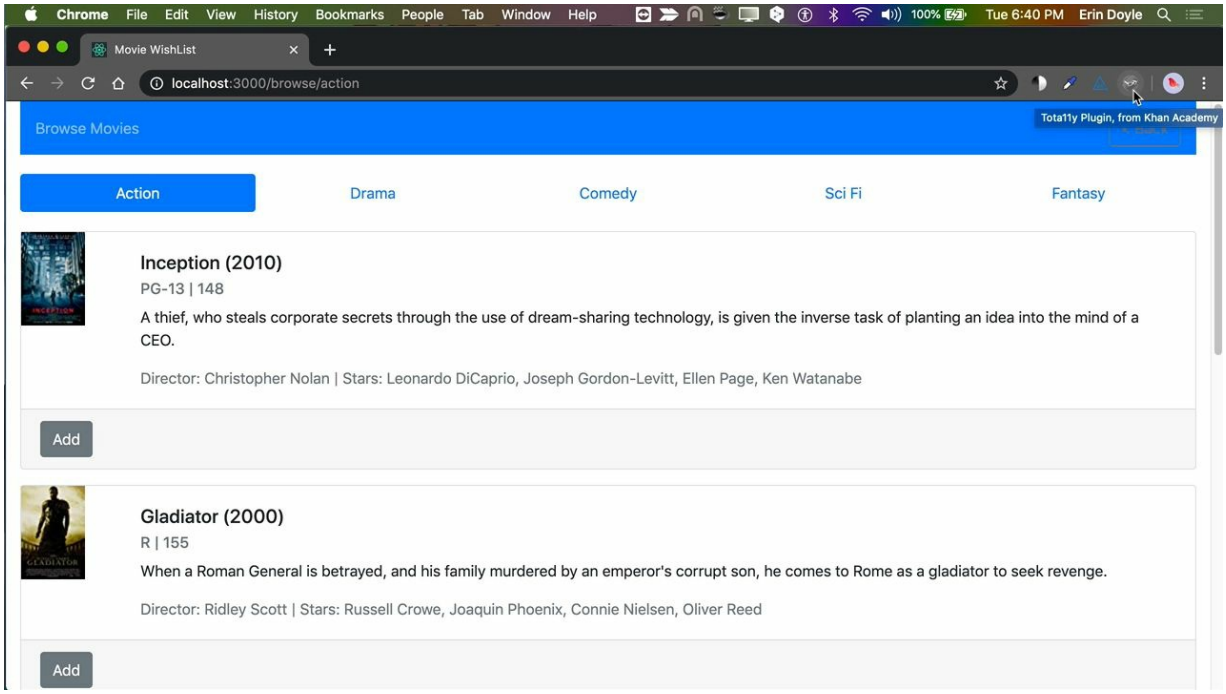
[01:15] Here once again, we've got the severity level. We've got the standard that has been violated for WCAG. We have a bunch of information on how to fix the problem and why it matters, and then finally, links to more resources about this issue.

[01:36] Using `axe` is a great way to learn more about how to solve a plethora of accessibility issues with your website.

Use tota11y to Visualize Accessibility Issues

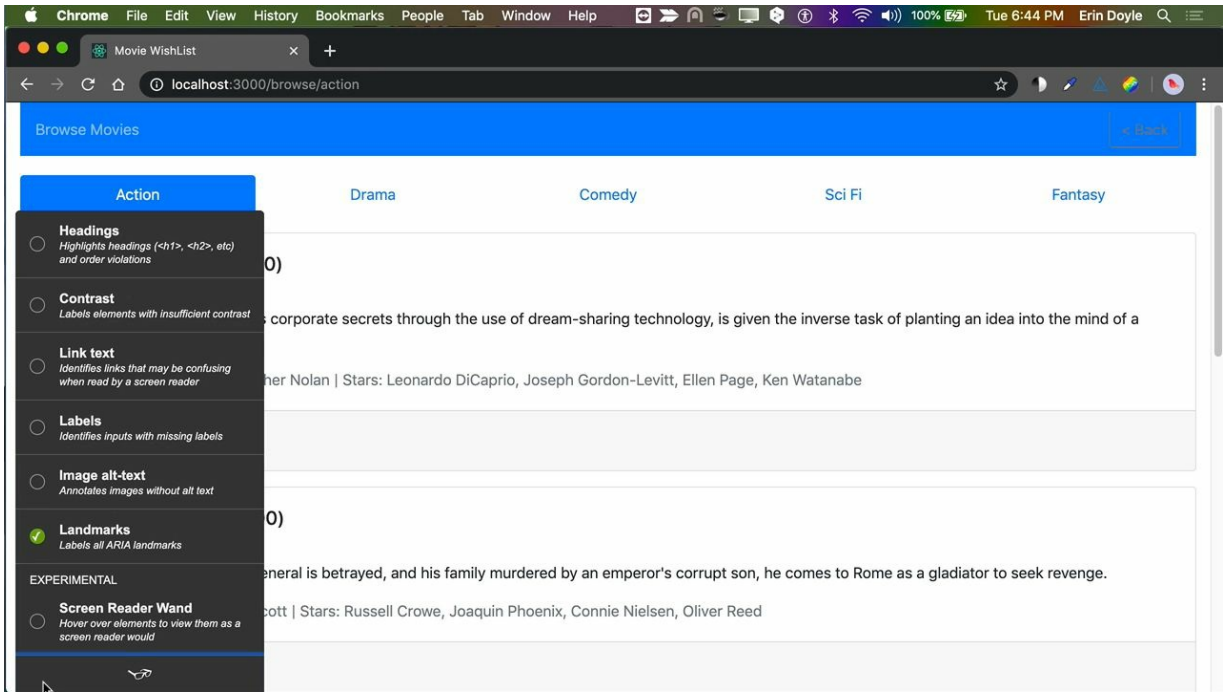
Instructor: [00:01] To install the `tota11y` browser extension you either need to find the plug-in in the Chrome Web store for Chrome (https://chrome.google.com/webstore/detail/tota11y-plugin-from-khan/oedofneiplgibimfkccchnimiadcmhpe?hl=en), or in the browser add-ons for Firefox (https://addons.mozilla.org/en-US/firefox/addon/tota11y-accessibility-toolkit/). Whichever browser you prefer. Now that we've got that installed, let's go ahead and use it.

[00:32] Here's a sample application with some accessibility issues baked in. You can now see the `tota11y` plug-in right here in the browser.



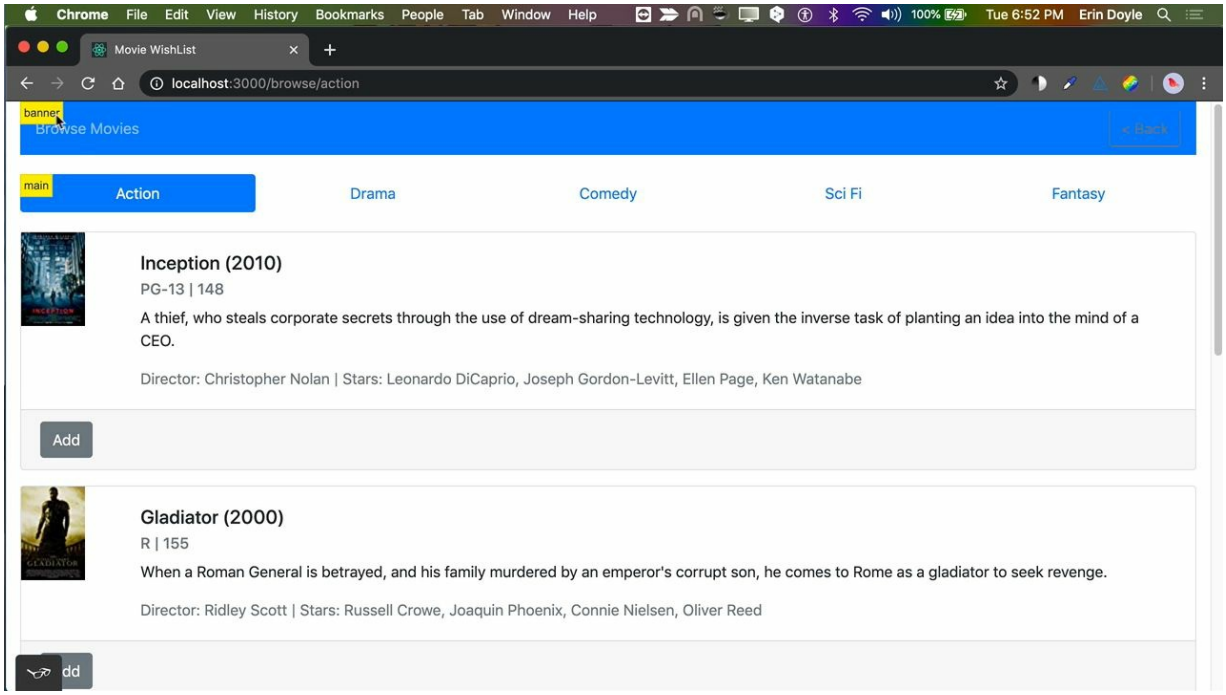
If we click on that, we can turn it on. You should now be able to see the sunglasses tab here on the bottom of the screen.

[00:48] By clicking on that tab, we can open up `tota11y`, and it lists all of the plug-ins available to use.



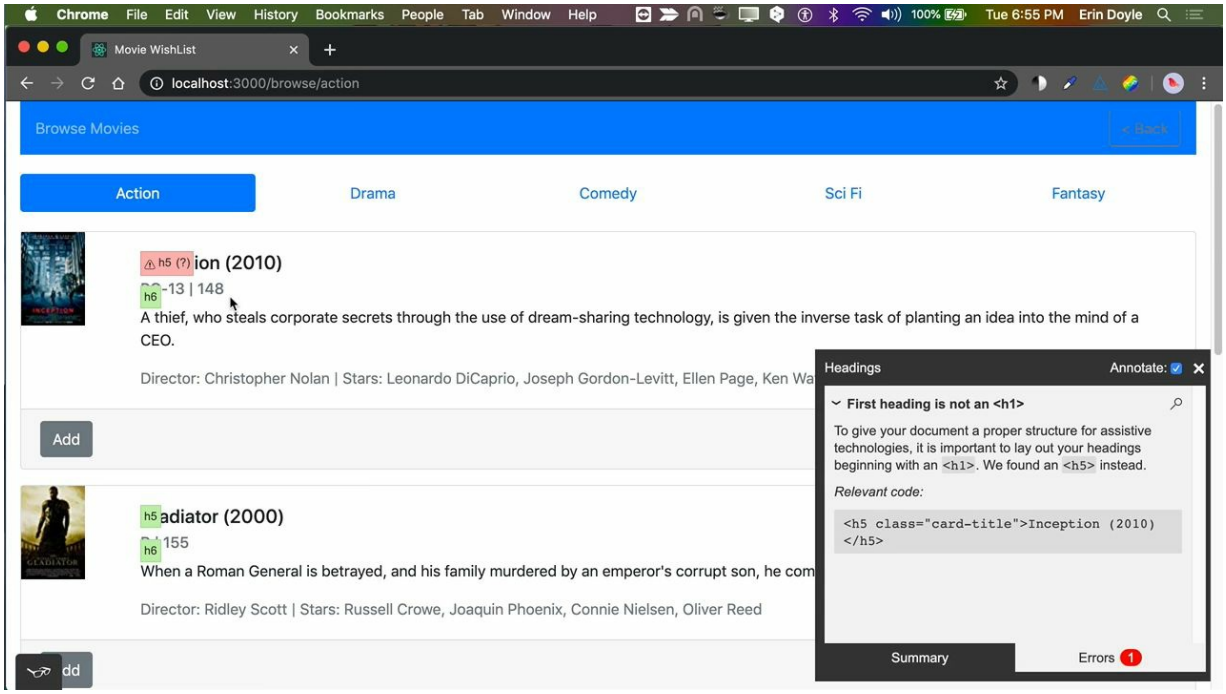
These can be enabled one at a time by clicking on them. If we want to start by looking at what landmark regions we have on the current page, we would enable the `Landmarks` plug in.

[01:11] As you can see, there are no landmarks annotated on this page, because there are no landmark regions, and that's an accessibility issue. Let's see what it looks like when we fix that. Now that we've added landmarks to the page, let's go ahead and enable that plug-in again and see what we've got.

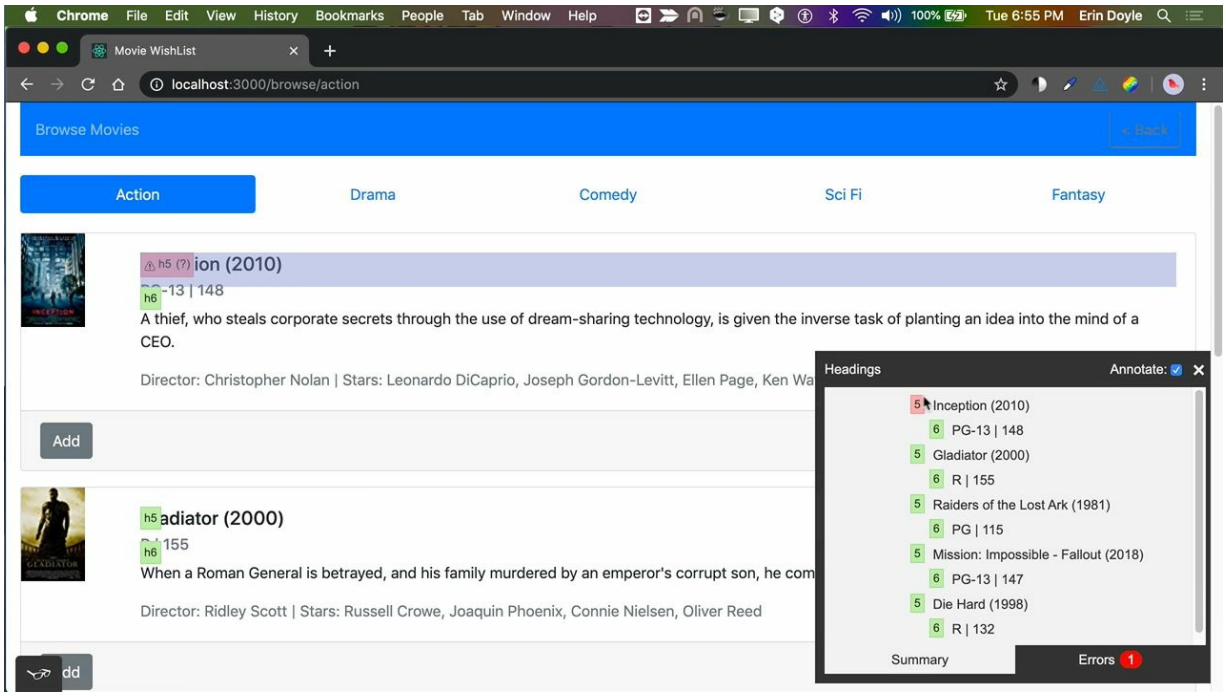


[01:30] Now you can see annotated the various landmark regions on this page. Now let's go ahead and look at headings. This will annotate all of the heading levels on the page, and it will show you when there's an issue with the heading levels, whether there's a missing H1 or the heading levels are not contiguous.

[01:55] Right here we can see in red we've got an issue with this heading level. It tells us what the issue is, it gives us information about it, and it points to the actual source code of that issue.

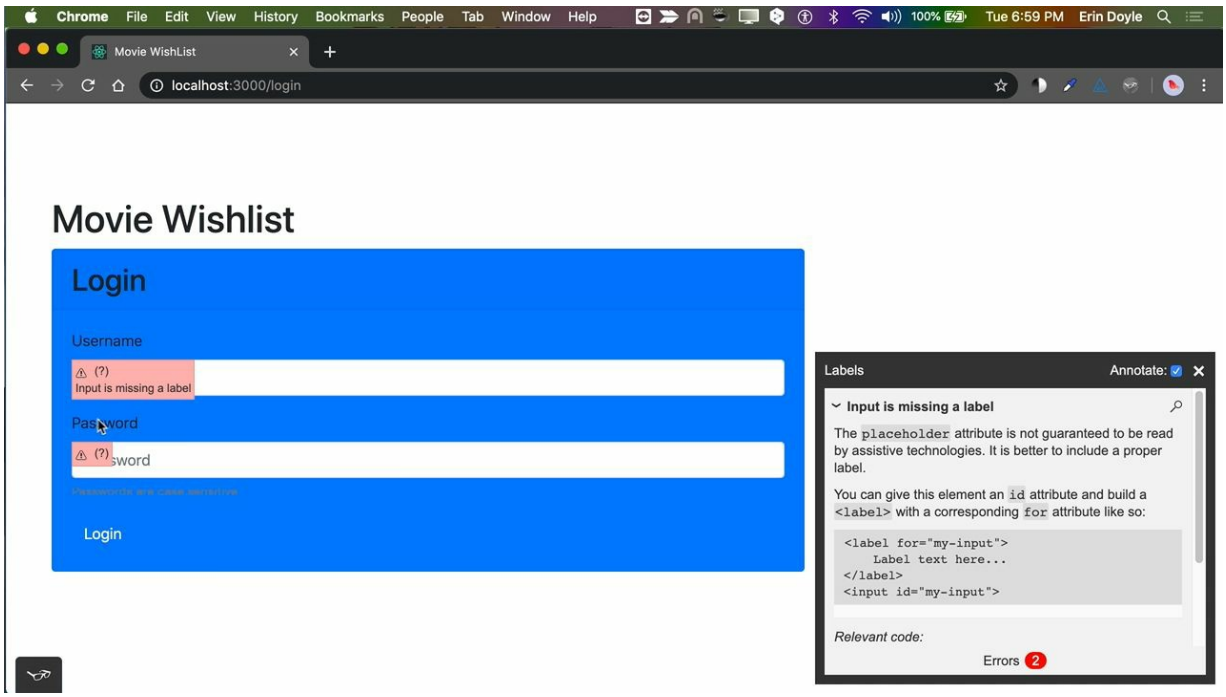


Also, we can look at a summary which will actually give us an outline of the heading levels so we can see if they're contiguous or not, and what order they run in.



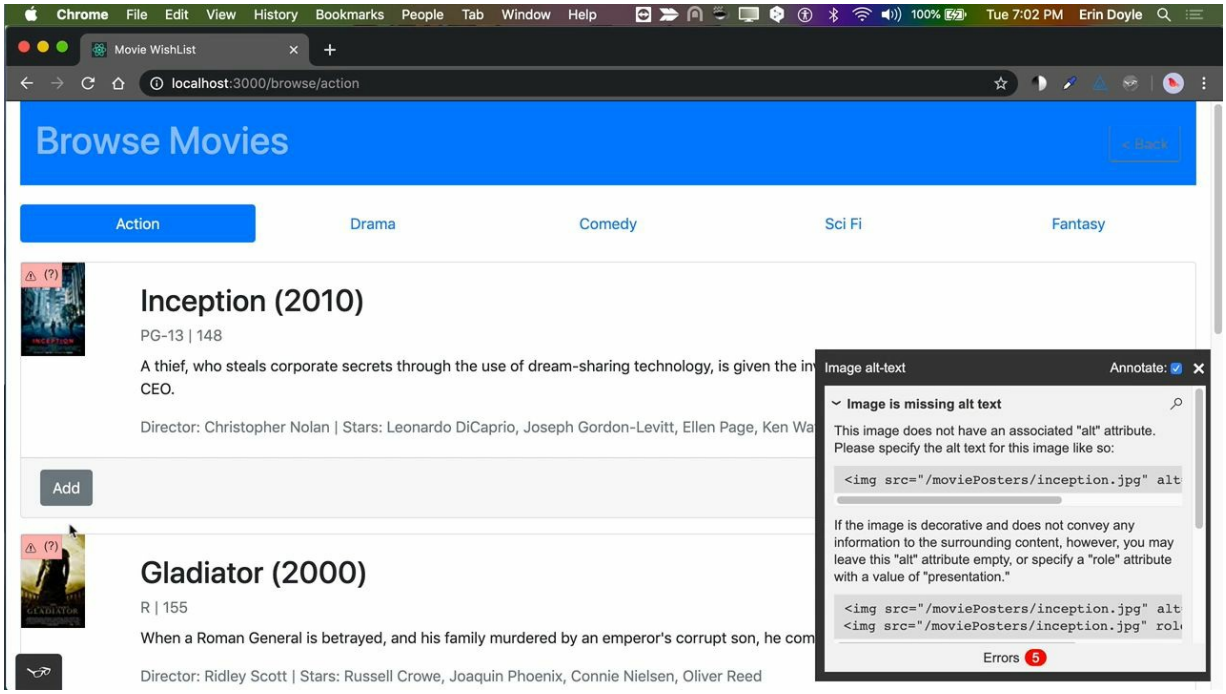
[02:17] If we fix this issue, now reenabling our heading plug in, you can see all of the annotated heading levels are green. You can see the summary is nice and happy. Now let's go to a page that has some input

labels. Let's look at the labels plug-in. This will annotate any missing form labels which we have two right here that are missing.

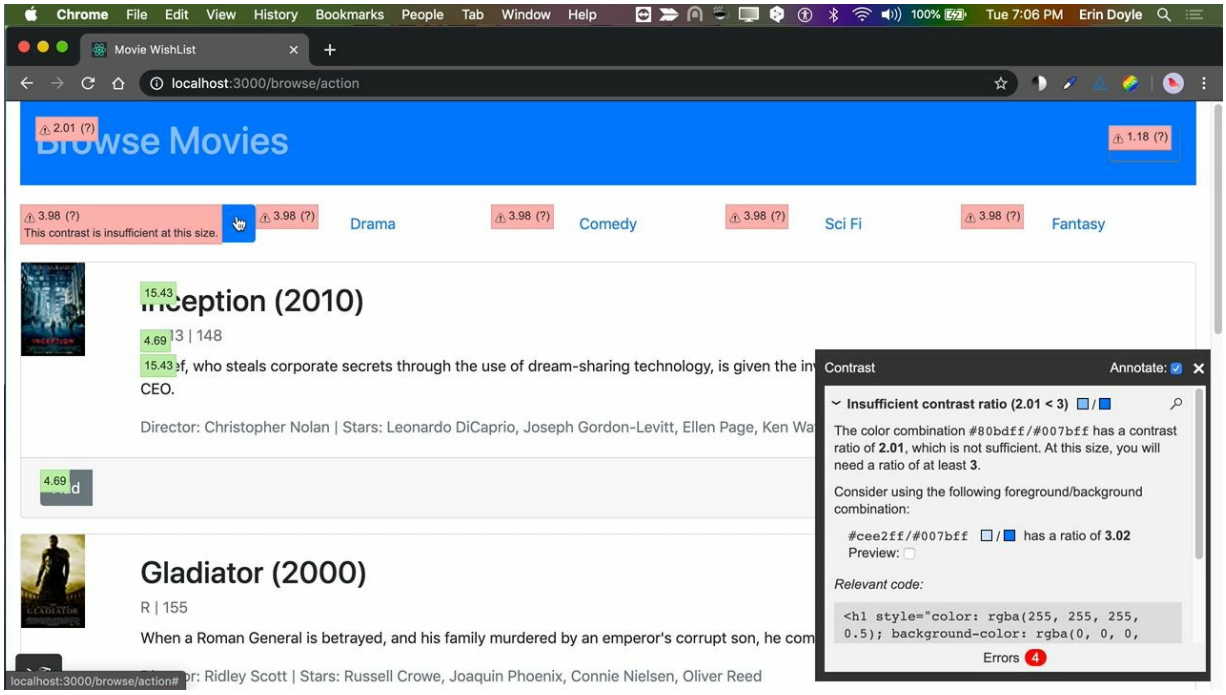


[02:54] Once again, we've got more information about the finding, the actual source code, and what we can do to fix the issue. Let's see what this looks like after it's been fixed. Now there are no annotations, which means there's no missing labels.

[03:23] Now let's go back to a page with some images. Let's run the plug-in that checks for missing image `alt` text. We can see right here that each of these images has been annotated, because they're missing `alt` text. Once again, we've got more information over here pointing to the actual source where the `alt` text that's missing, for each of the elements where it's missing.



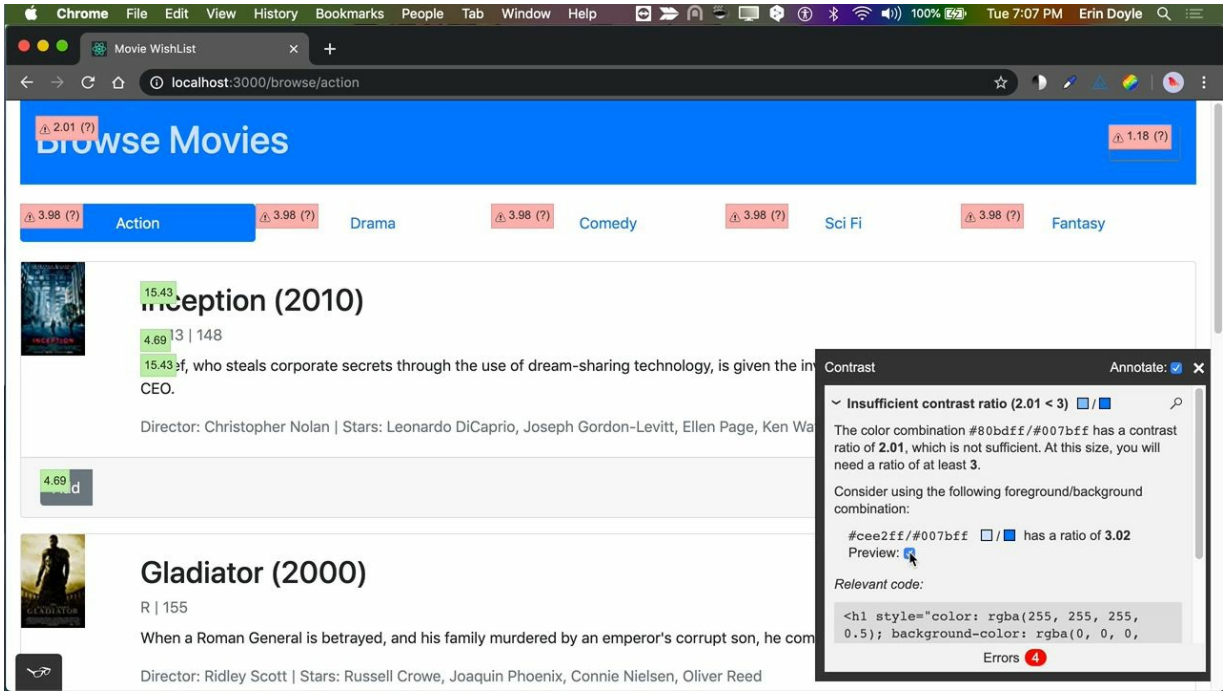
[03:52] If we fix this, and check again, we now see no annotations because all of our images have `alt` text. Now let's see if we have any color contrast issue on this page by enabling the `contrast` plug-in, and we've got a lot of them.



[04:13] You can see all of the elements that have the red annotations showing where there's insufficient contrast ratios between the foreground and background colors. You can also see the areas that have acceptable

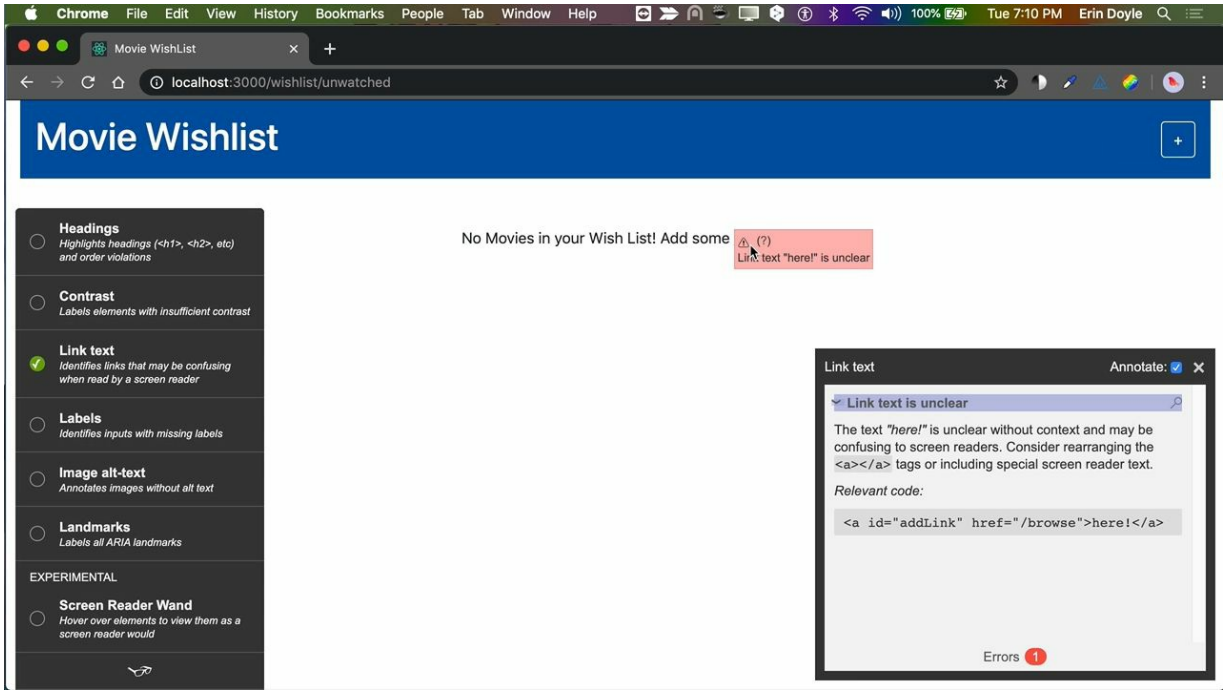
contrast ratios, and those are shown in green.

[04:32] Once again, this information over here lists each element that has an issue. In addition, it lists the contrast ratio, and shows the foreground and background colors. It gives you a suggestion of a foreground and background color combination that would give you an acceptable ratio, and you can actually preview those colors by clicking in the preview checkbox.



[04:58] There you can see we're previewing a change to the `Browse Movies` title foreground color. You can see it's actually a lot easier to read when we try their suggestion. Now let's go ahead and get those colors fixed and see how it looks. Enabling our contrast check again, there we go, all of our annotations are green, we have all acceptable contrast ratios showing.

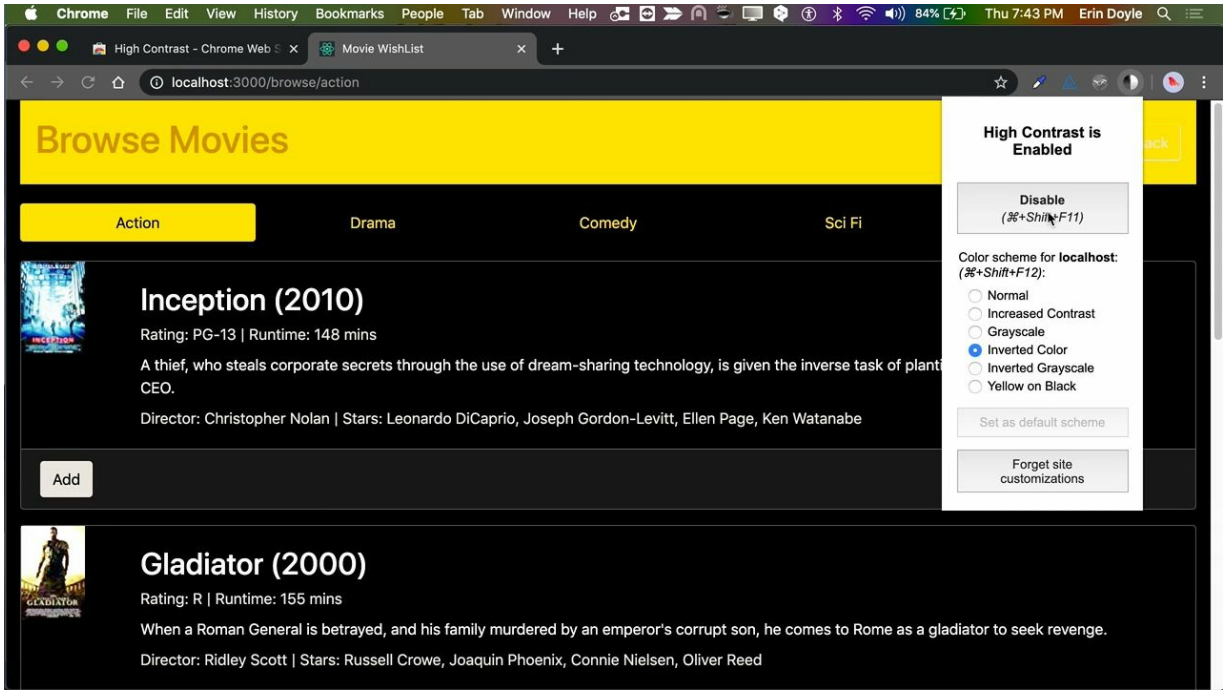
[05:32] Finally, let's check out link text. Let's go to a page where we have some link text and change to the `Link text` plug-in. Right here we have an annotation for this link that just says `here`, and it explains to us here that the text `here`, is unclear without context, and may be confusing to screen readers. It shows you the source of the code that has the issue.



[06:00] Let's fix that real quick, checking that again, now we no longer have an annotation because our link text is now nice and clear.

Use a High Contrast Browser Extension to Find Color Contrast Accessibility Issues

Instructor: [00:00] To begin, you'll need to find the high-contrast extension in the Chrome Web Store (https://chrome.google.com/webstore/detail/high-contrast/djcfdncoelnlbldjfhinnjlhdjlikmph) and add it to Chrome. Now that that's installed, let's use it. Here's a sample web application that has some purposeful color contrast issues. We can now see my extension over here in the browser. Clicking on that, we can then click enable.



[00:31] There are a number of different modes we can choose from. `Normal` is, of course, normal. No special high-contrast mode settings are turned on. We've got `Increased Contrast`, `Grayscale`, `Inverted Color`, `Inverted Grayscale`, and `Yellow on Black`. You can go ahead and set a default or, clearly, `Forget set customizations`. We've even got some keyboard shortcuts.

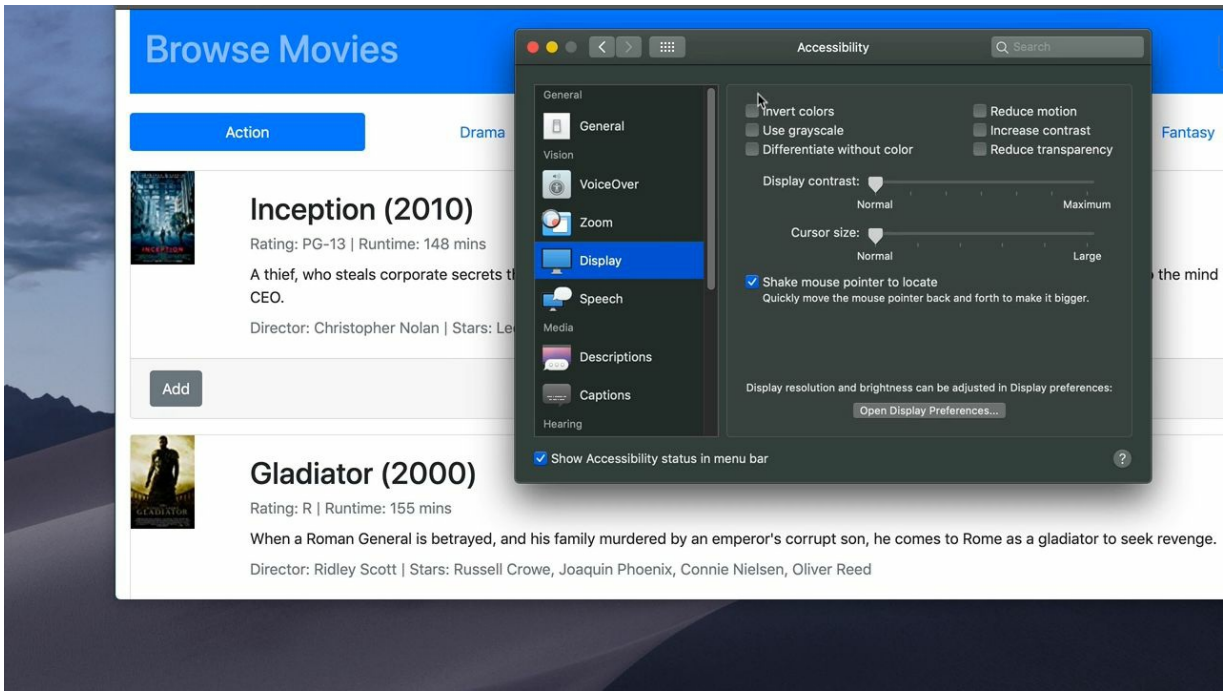
[01:04] It's really helpful to make sure you go through each of these modes for each page of your web application to just ensure that everything is displayed clearly, regardless of the selected mode. Make sure nothing suddenly disappears or becomes difficult to read or see, because any of these modes might be in use by our users.

Change Display Preferences on Mac to Find Color Contrast Accessibility Issues

Instructor: [00:00] Here's a sample web application I've got with some color contrast issues baked in. Now let's modify the macOS accessibility display preferences to see how it impacts the color contrast of the page.

[00:13] We'll go to our `System Preferences`, and then the `Accessibility` preferences. If we want to get back here faster in the future, you can check this box to show accessibility status in the menu bar. Now the icon is right up here and we can get back there quickly.

[00:33] We'll go to the `Display` settings.



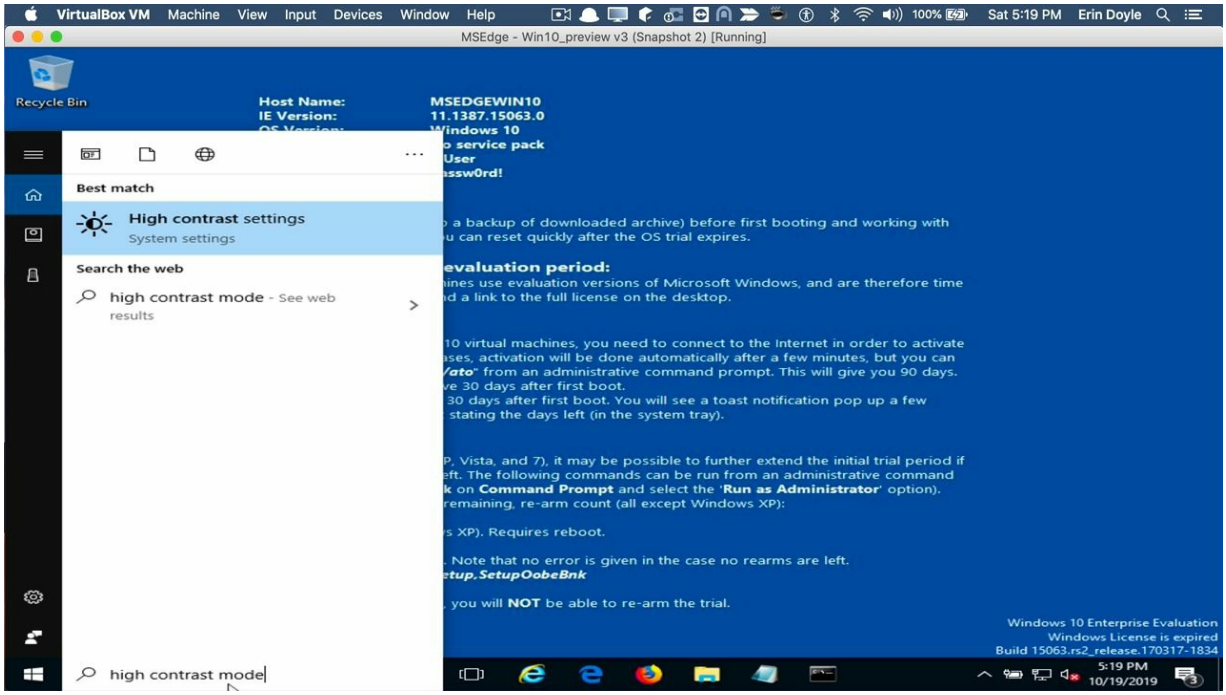
There are a few different settings we can modify. We could `Invert colors`, `Use grayscale`, and we can modify the amount of contrast. We can also see what it looks like when we combine some of these settings.

[01:08] These three are the ones that will have the most impact to your web page. The rest of these settings are primarily for operating system level display. You can see how turning these on and off doesn't really change the way the web page is displayed.

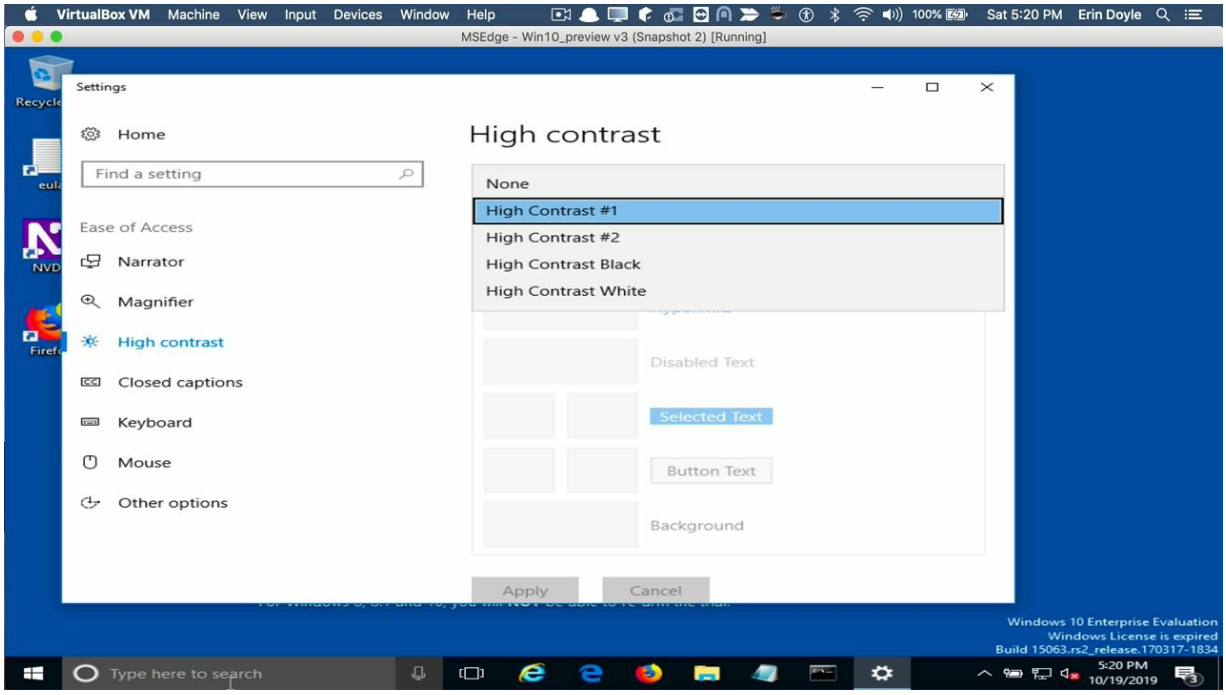
[01:25] It's always a good practice to just make sure that with any combination of these settings, everything on your page is still clear and easy to read.

Enable High Contrast Mode on Windows

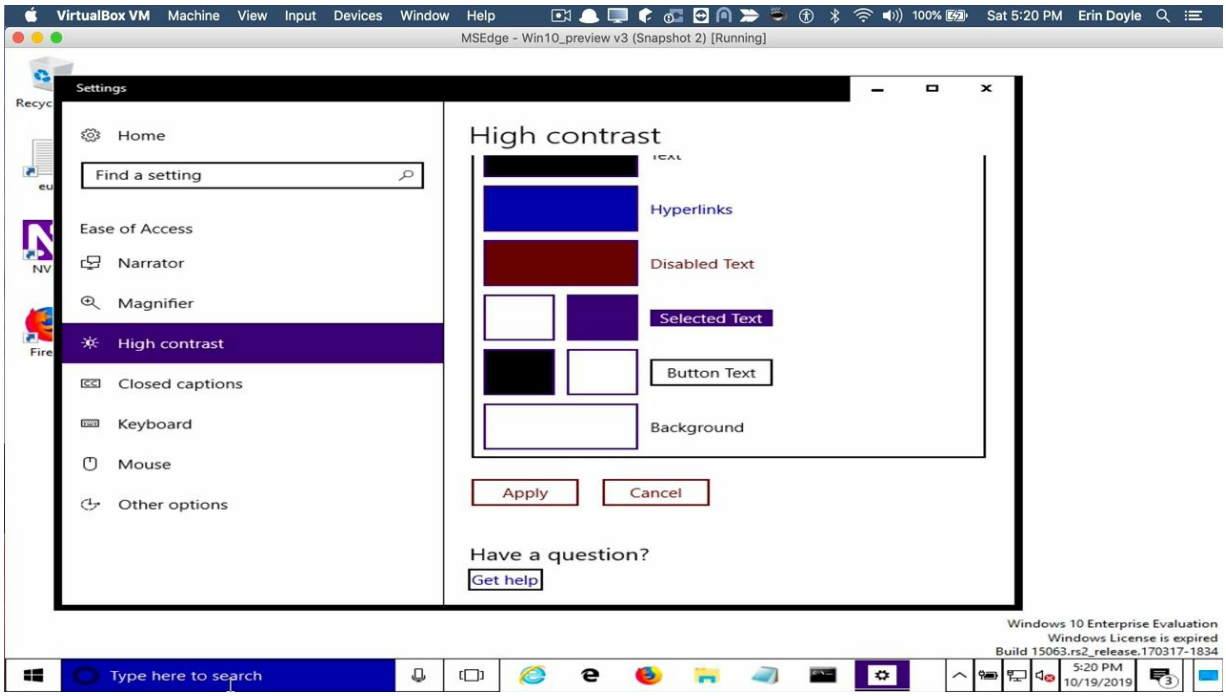
Erin Doyle: [00:00] Here we are in Windows 10, and we want to enable high contrast mode. The fastest way to do so is to go down to the search box and enter `high contrast mode`.



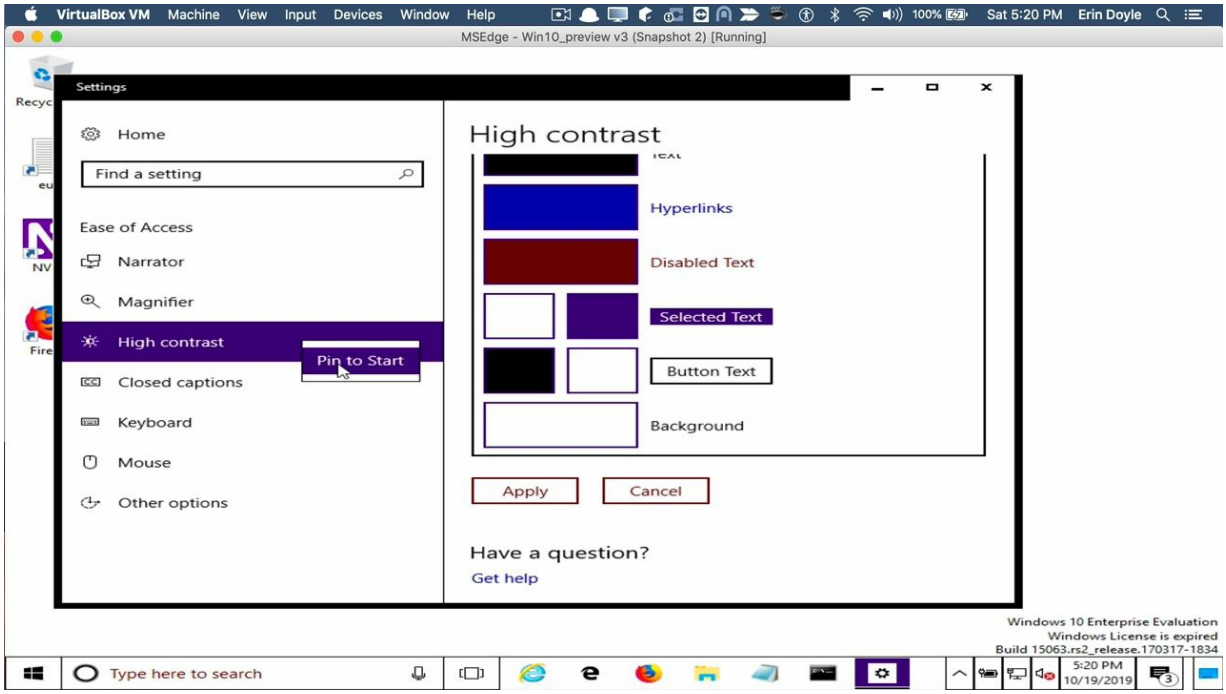
[00:12] Here are the high contrast settings. There are four themes to choose from. There's `High Contrast #1`, `High Contrast #2`, the Black theme, and the White theme.



We're going to go ahead and apply the White theme.



[00:34] One way we can add a shortcut to get back to these settings quickly is by right-clicking on the `High contrast` item in the Ease of Access menu and selecting `Pin to Start`.



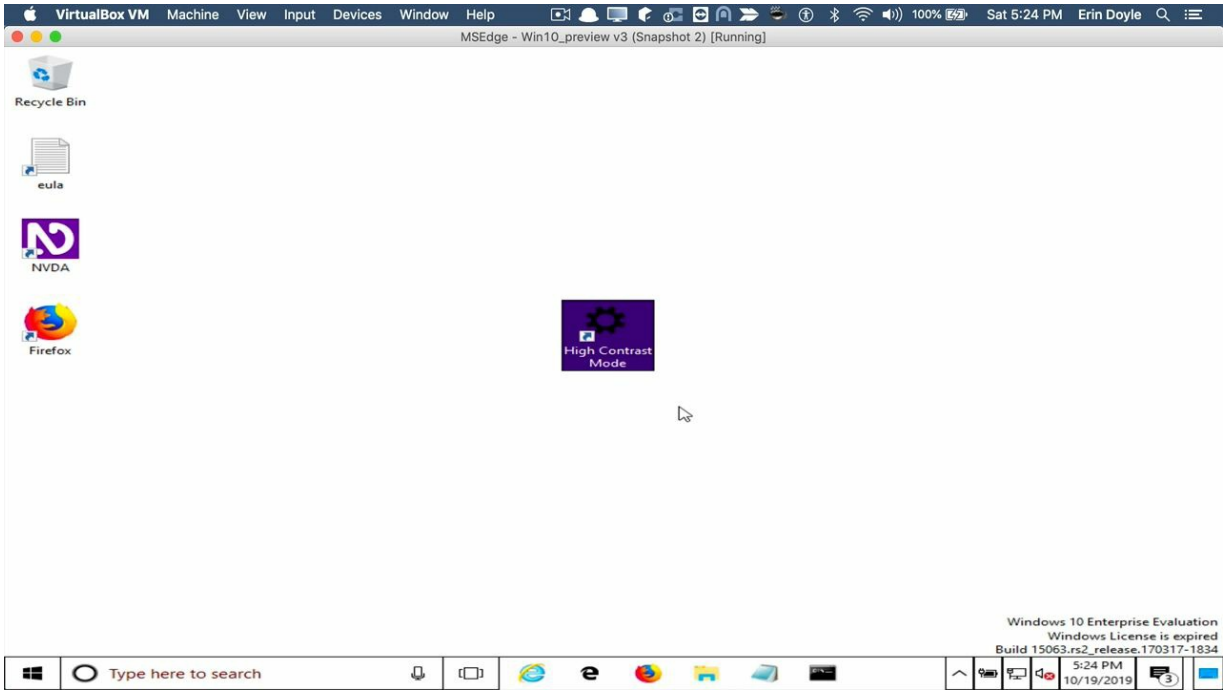
Now, when we go to the Start menu, we can see the high contrast settings.

[00:57] Another way we can get back to these settings quickly is by creating a shortcut. You go to the desktop, select New > Shortcut.

[01:06] For the location we need to enter `ms-settings:easeofaccess-highcontrast`.

We'll give that a name of `High contrast Mode` Hit Finish.

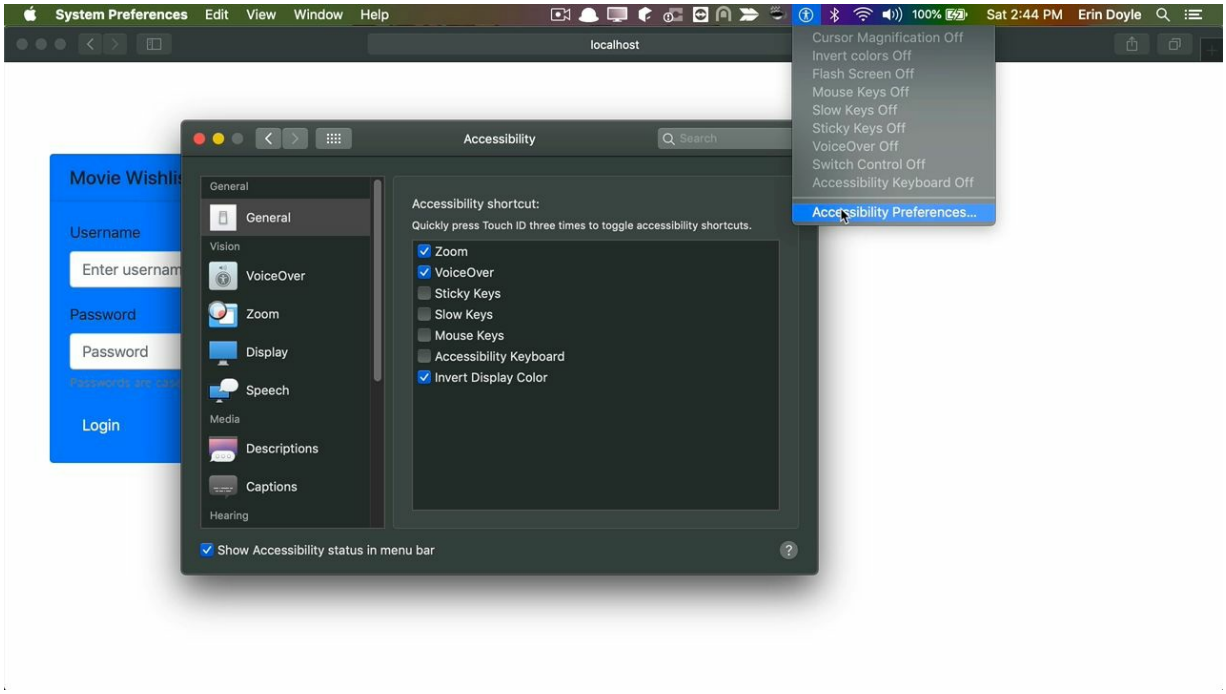
Now I've got a shortcut.



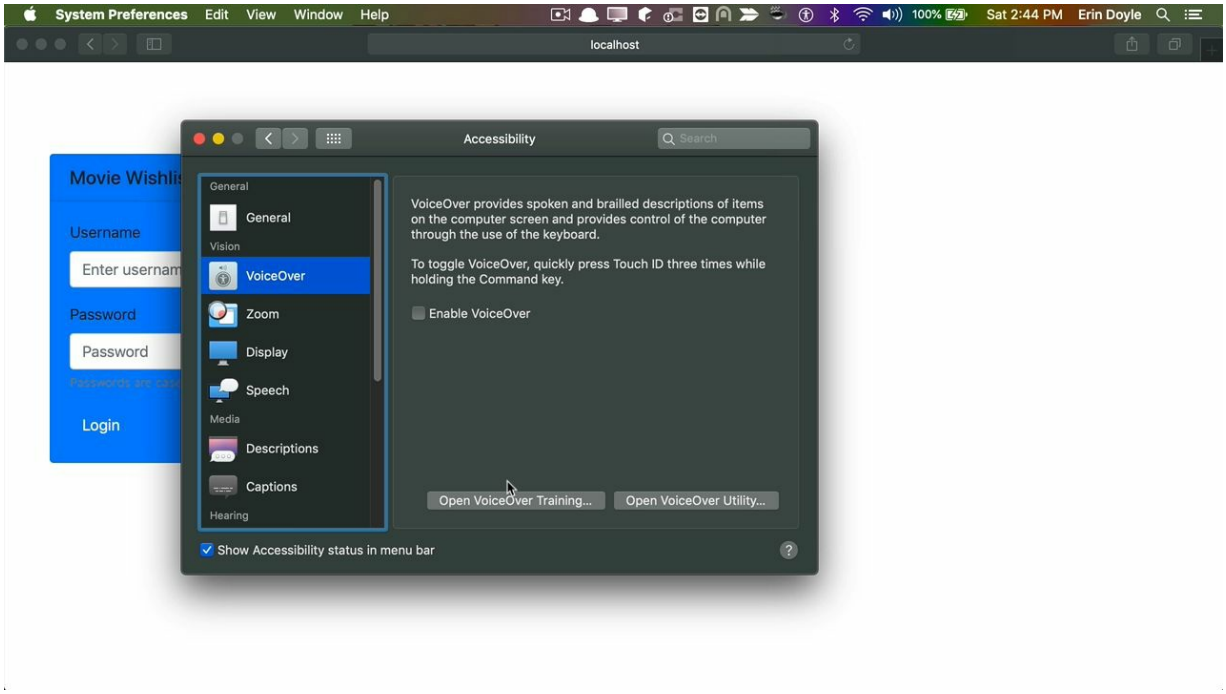
When we double-click it, it brings us right back to the settings.

Access and Customize VoiceOver Settings on MacOS

Instructor: [00:00] To get started with `VoiceOver` on Mac, you want to go to your `System Preferences` and then here on `Accessibility`. One thing you might want to do is clicking this box to show Accessibility status in menu bar. If you click that, you now have this shortcut to get to the preferences if you need to get to them easily later.

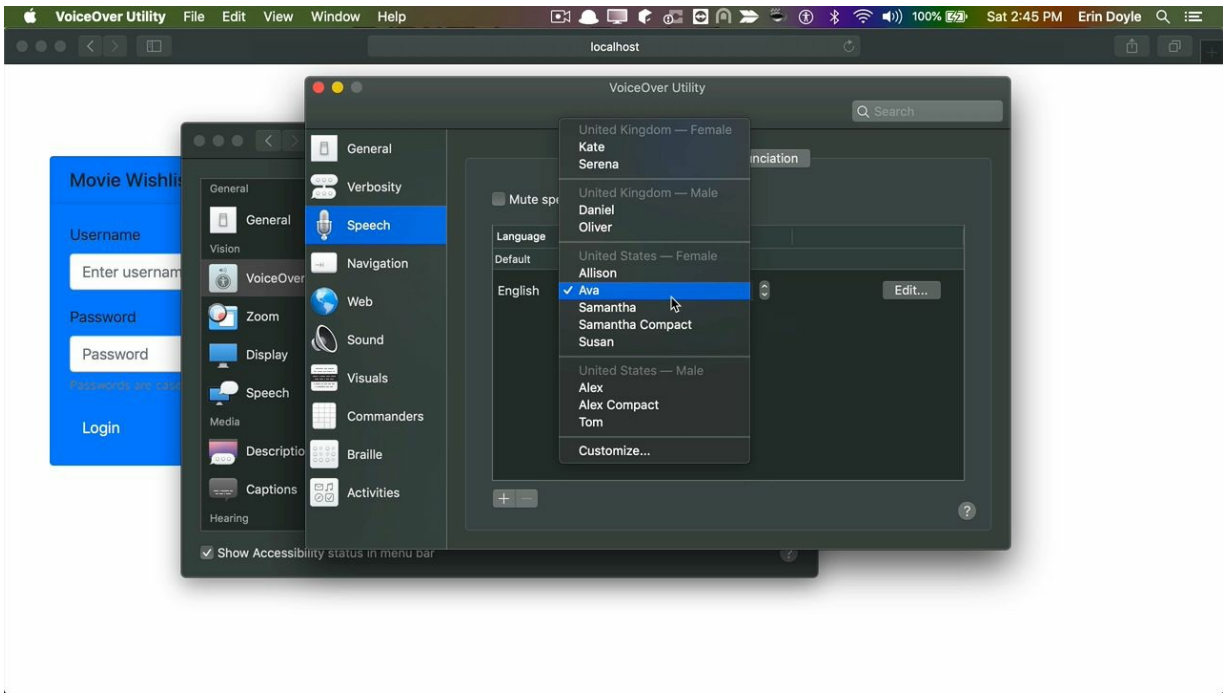


[00:26] Then you go to the `VoiceOver` section. If you're just getting started with `VoiceOver`, I strongly recommend you go ahead and take their training. This is going to walk you through step-by-step how to properly use `VoiceOver`.



[00:40] If you were looking to customize `VoiceOver`, you'll click this `Open VoiceOver Utility`. Some of the things you can do that I found to be really helpful, if you go to `Speech`, you can actually change the

voice. The default that usually comes enabled with `VoiceOver` might not be the most helpful. There are a ton of voices you can choose from.



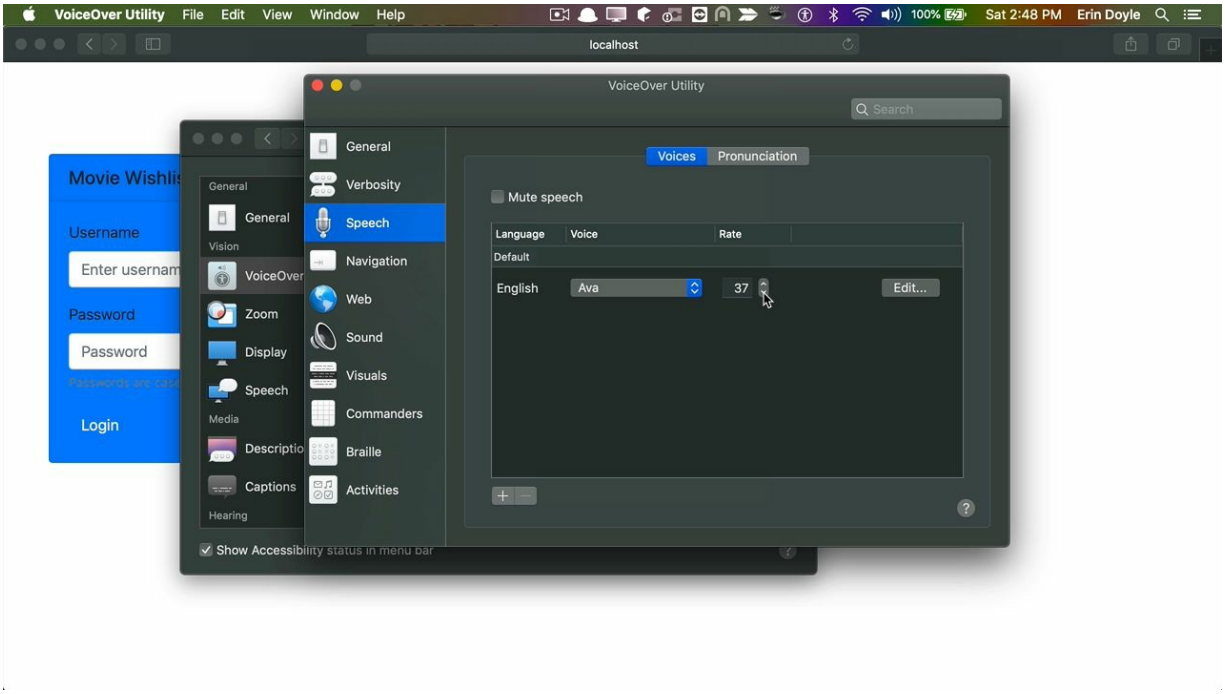
[01:01] There are male and female voices for a whole slew of different regions with different accents. You can go through those and pick one that works for you. You can enable a lot of different options that then you can choose from here in this Context menu.

- Allison: [01:20] Allison, the best way to predict the future is to invent it.
- Ava: [01:24] Ava, the best way to predict the future is to invent it.
- Alex: [01:29] Alex, the best way to predict the future is to invent it.
- Tom: [01:34] Tom, the best way to predict the future is to invent it.

Instructor: [01:38] I'm going to pick `Ava`.

- Ava: [01:39] Ava, the best way to predict the future is to invent it.

Instructor: [01:43] Then you can also change how fast the voice speaks. I found that the default was a little fast for me to keep up with. You may have a different preference. You can change that here with the `Rate` .



[01:53] You can go really slow.

- Ava: [01:55][slow voice] Rate 29, the best way to predict the future is to invent it.

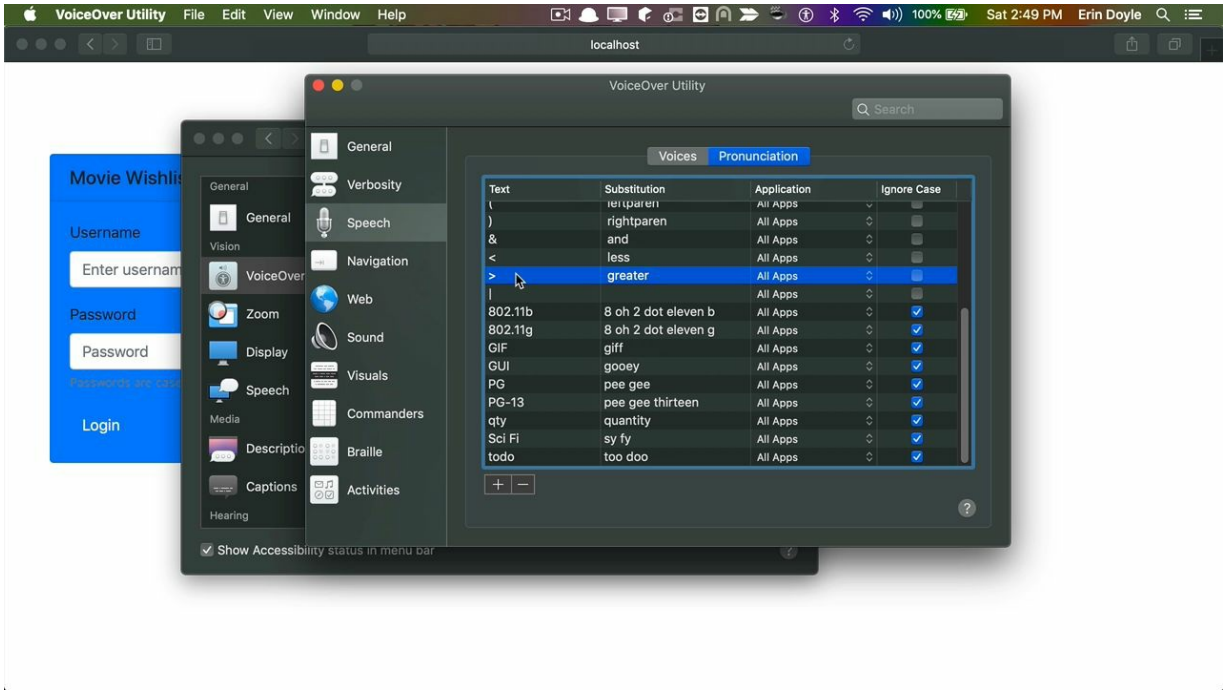
Instructor: [02:01] Or, you could go really fast.

- Ava: [02:02][fast voice] Rate 55, the best way to predict the future is to invent it. [slower voice] Rate 43, the best to predict the future is to invent it.

Instructor: [02:15] That's the rate that works for me. The other thing you can do to customize `VoiceOver` further is, if you find there are certain characters, words, combinations of characters that it pronounces strangely, you can go ahead and tell it how to pronounce those things.

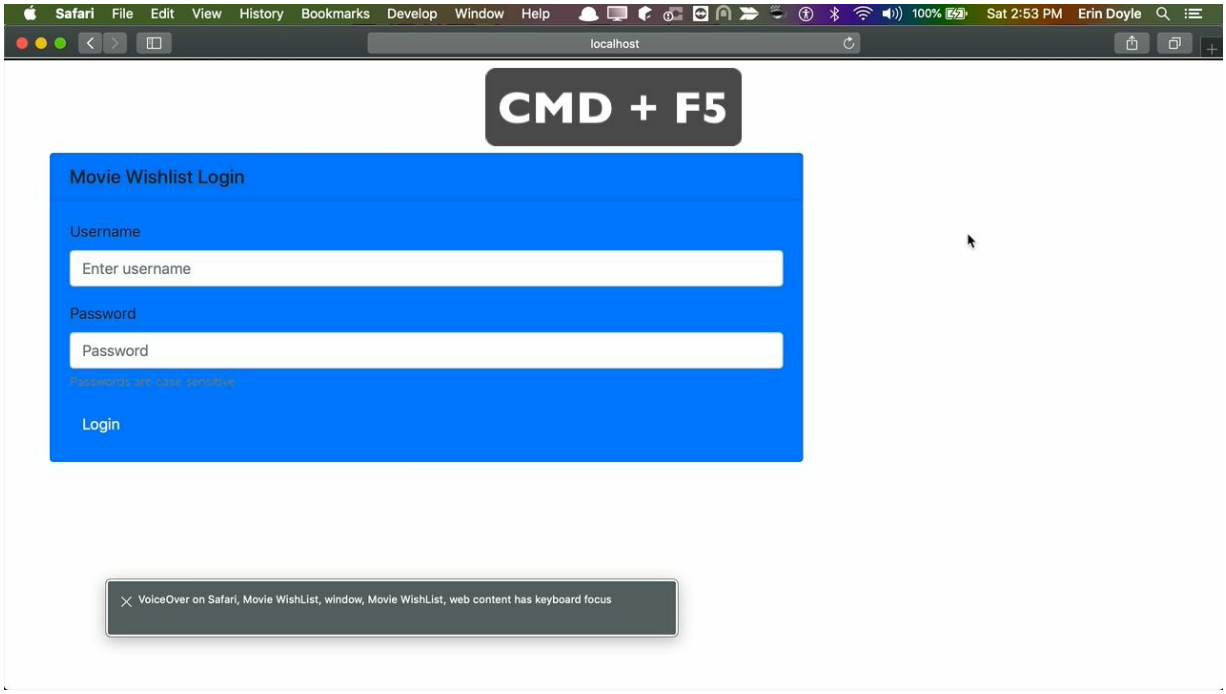
[02:34] Here's some examples of some things, words, either single characters. When you see this character, it should say `less`. For this one, it will say `greater`. Or to acronyms or abbreviations, how they

should be read. You can add however as many of these as you like.



Navigate a Webpage with VoiceOver in Safari

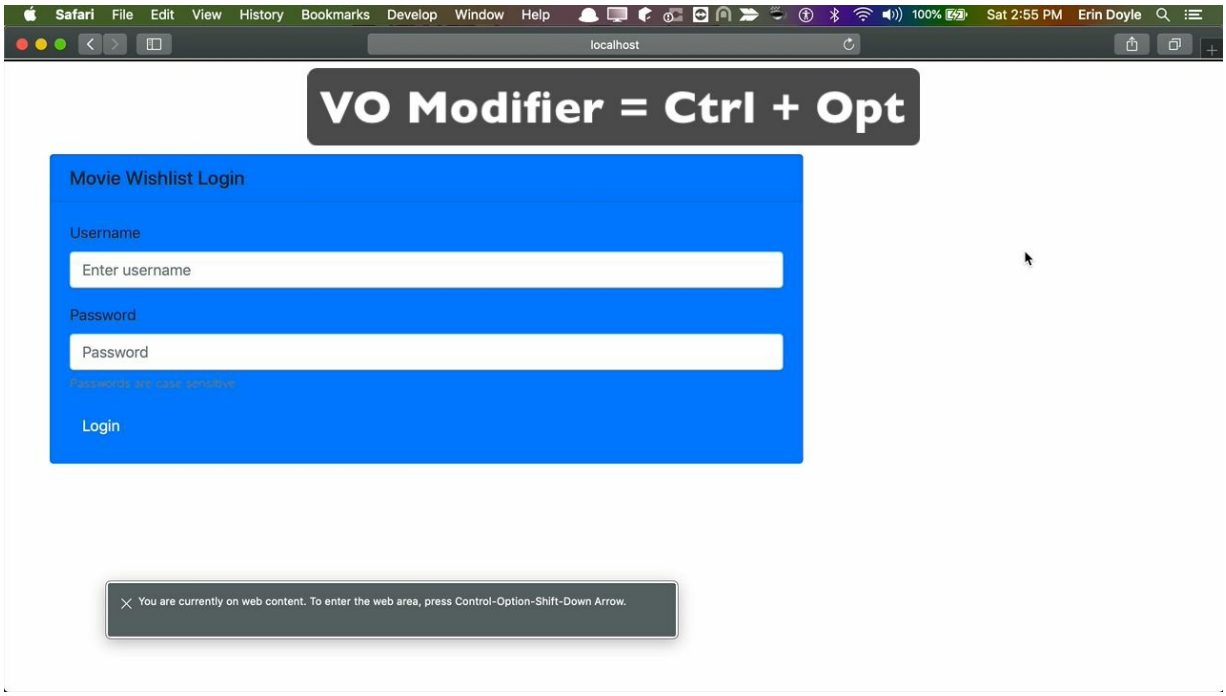
Instructor: [00:01] Here I am in Safari. Safari has been found to be the most popular combination to use with `VoiceOver`, much more so than Chrome and FireFox at this time. We're going to use Safari to test `VoiceOver`. You can enable `VoiceOver` with Command-F5.



- VoiceOver`: [00:22] VoiceOver on Safari, Movie Wishlist, window, Movie Wishlist, web content has keyboard focus. You are currently on web content. To enter the web area, press Control-Option-Shift-Down Arrow.

Instructor: [00:36] Right now, the `VoiceOver` cursor is focused on the entire web window. As it says, we need to hit `Control-Option-Shift-Down Arrow` to enter the web page. If we continued to move the cursor along, we would move through the various options in the Safari window and out into the operating system.

[01:01] If we want to go into the web page and move through the web page, we have to, as it says, `Control-Option-Shift-Down Arrow` into that. We'll be using what it refers to as the `VoiceOver` modifier keys to navigate. By default, that's `Control-Option`.



[01:20] The other default is `Caps Lock`. You can set up whatever combination of keys you want to use for the `VoiceOver` modifiers. We're going to step into the web content.

- `VoiceOver`: [01:33] In movie wish list, web content heading level five, movie wish list login. In heading level five, movie wish list

login.

Instructor: [01:42] As you can see, it reads what it's focused on. It describes what the element is and any sort of details or context around that element. We can just, using again those `VoiceOver` modifier keys and the arrow keys to move through all of the elements on the page.

[02:01] One more thing that's really helpful is sometimes `VoiceOver` can be very verbose. If you need to pause what it's saying, you can use the `Control` key.

[02:10] I am just going to step through the elements of the page using the `VoiceOver` modifier keys plus the Right Arrow key to go forward.

- `VoiceOver`: [02:19] Heading level five, movie wish list login. You're currently on a heading level five username. You're currently on a text element. Enter username. Edit text.
  Password. You're currently on a text element. Password. Password. Secure edit text. Passwords are case sensitive. Login button. You're currently on a button inside of web content. To click this button, press Control-Option-Space to exit this web area. Press Control-Option-Shift-Up Arrow.

Instructor: [02:49] As it describes, if I'm on a button, if I'm on something that could be selected or clicked, then it will inform me that I can do so with `Control-Option-Space`.

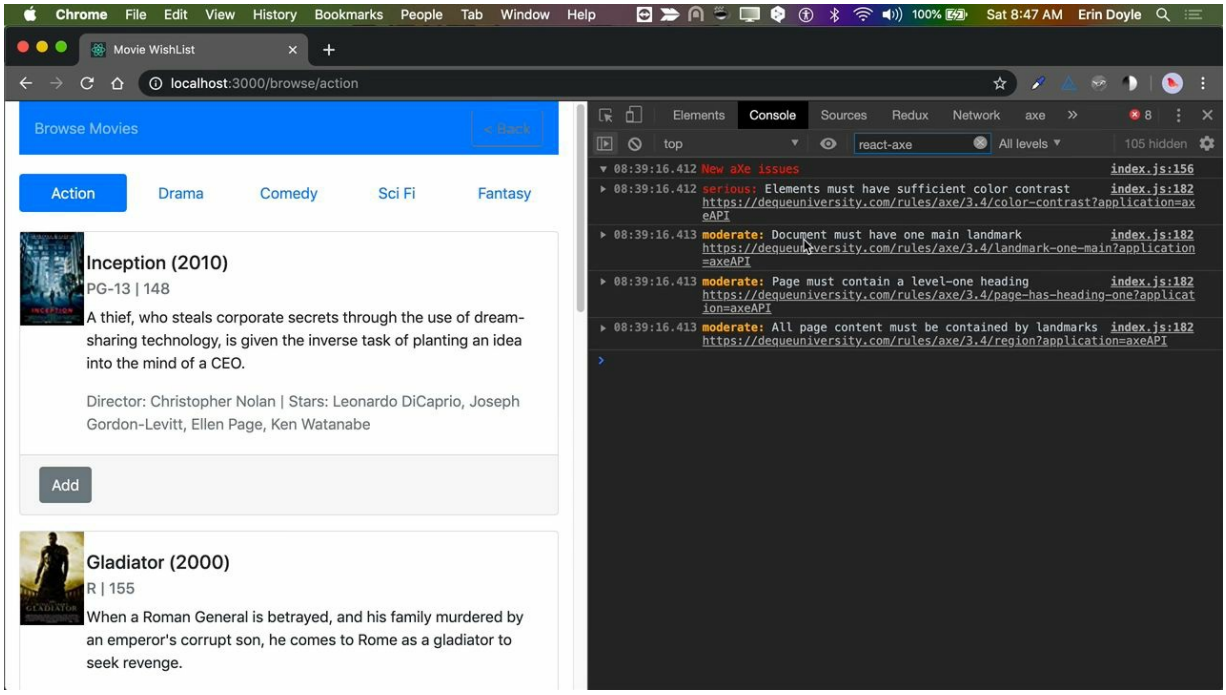
- `VoiceOver`: [03:02] Press login button.

Instructor: [03:06] That's the basics of using `VoiceOver`. I'm going to go ahead and turn it off now, again with `Command-F5`.

- `VoiceOver`: [03:14] `VoiceOver` off.

Test for Landmark Region Accessibility Issues in React

Instructor: [0:00] Here's a sample Web application I've got with some accessibility issues in it. I'm running React Axe, and so, we can see any of the findings being reported to the console here.

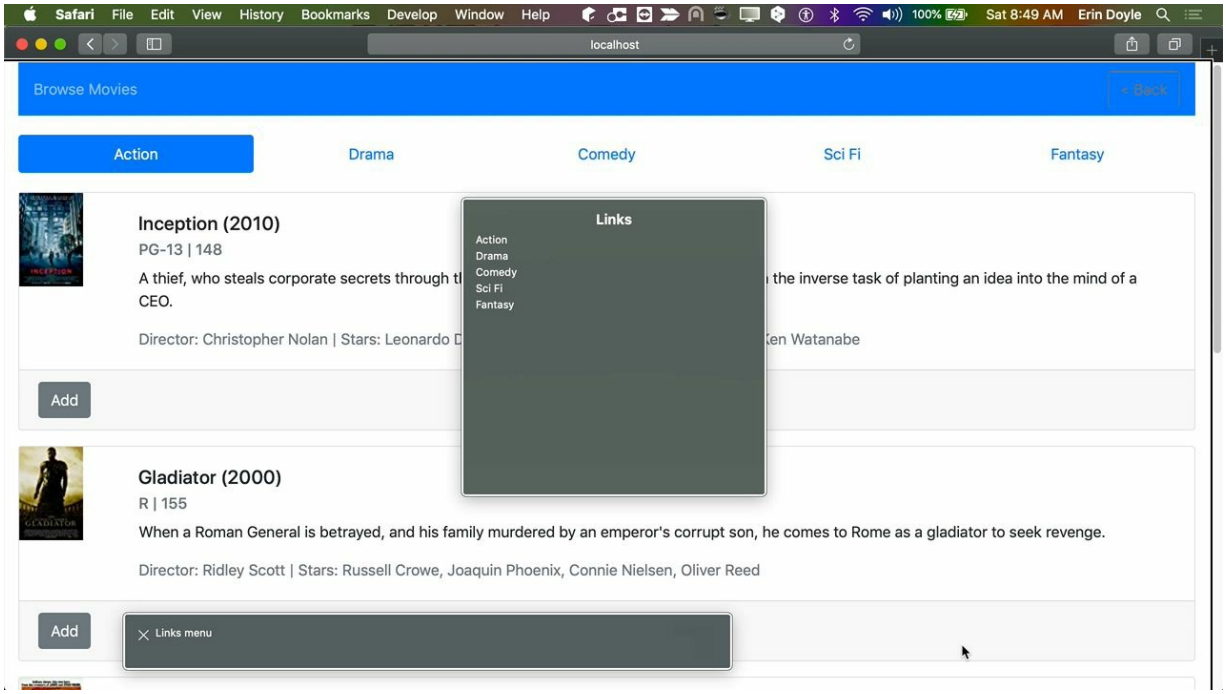


[0:10] We can see that we have a couple of findings concerning `landmark` regions. Our document does not have a `main landmark`, and all page content must be contained by landmarks. For this page, there are no landmarks.

[0:26] If we run `tota11y`, we can also see that when we go to Annotate Landmarks, nothing is annotated. Again, we don't have any landmark regions and that's a problem.

[0:37] One of the impacts, for example, is that the `rotor` with `VoiceOver` is not going to list any of the landmark regions. Users that make use of the `rotor` are not going to be able to easily find the various sections of the page, and we can demonstrate this.

[0:52] Here's the `rotor`.

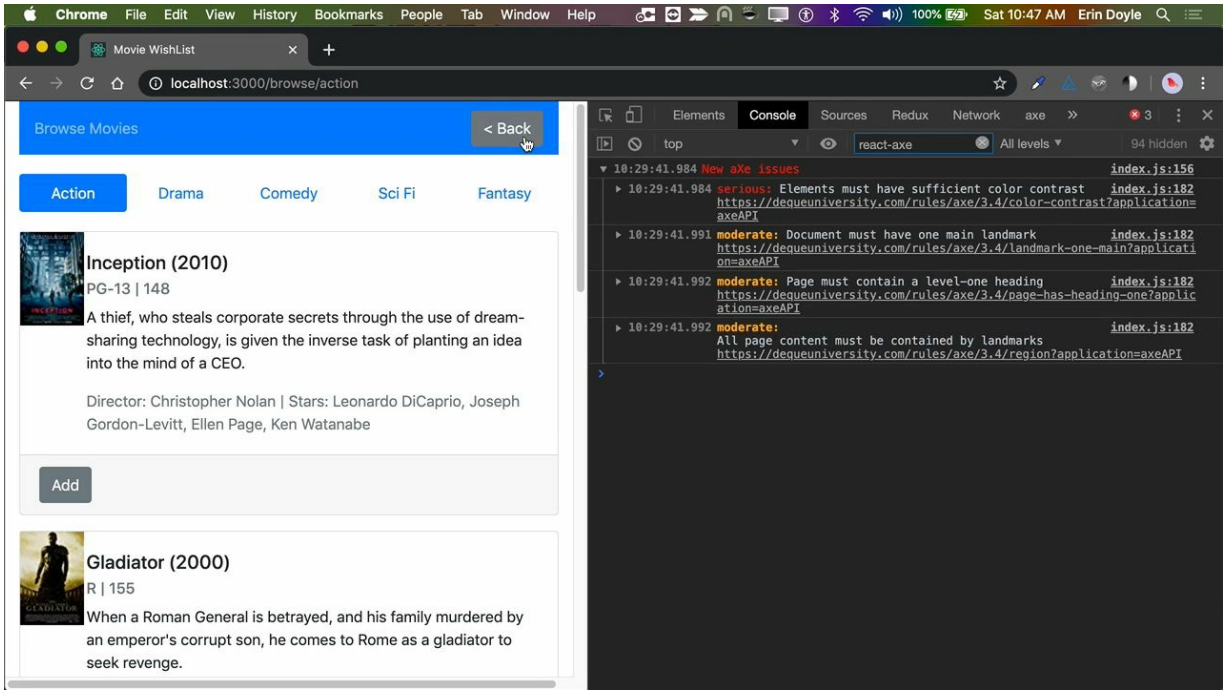


If we go through each of the menus, we should see one for landmark regions. Here, there is no menu for landmark regions because we don't have any, so let's fix that.

Define Landmark Regions of a web page using ARIA roles

Instructor: [0:02] Here is a Web page with some accessibility issues concerning landmark regions. We can see them over here, being reported in the console by `react-axe`. We're missing one `main landmark`, and the page content is not currently contained within landmarks.

[0:18] If we look at the structure of this page, we can figure out which sections should be defined as which landmark regions. Looking here, this banner up here looks like it represents the header. This button right here, which navigates us back to the previous page, should probably have a role of navigation.



[0:38] The rest of this comprises our main section of the page. Then down here is our footer. Let's go ahead and define these landmark regions in our code. Here is my React component that contains all of the layout code making up this page.

MovieBrowser

```
import React from "react";
import PropTypes from "prop-types";
import { NavLink } from "react-router-dom";

import movies from "../movies";

import BrowseList from "./BrowseList";
import getBrowseActions from
"./getBrowseActions";

const MovieBrowser = ({
  history,
  match,
  wishlist,
```

```
  addToWishlist,
  removeFromWishlist
}) => {
  const goToWishlist = () =>
history.push("/wishlist");
  const movieActions =
getBrowseActions(addToWishlist,
removeFromWishlist);
  const moviesInGenre =
movies[match.params.genre];

  return (
    <div>
      <div className="navbar navbar-dark bg-
primary">
        <span className="navbar-text">Browse
Movies</span>

        <button className="btn btn-outline-
secondary" onClick={goToWishlist}>
          {"< Back"}
        </button>
      </div>

      <ul className="nav nav-pills nav-
justified">
        <li className="nav-item">
          <NavLink
            to="/browse/action"
            className="nav-link"
            activeClassName="active"
          >
            Action
          </NavLink>
        </li>
        <li className="nav-item">
```

```
            <NavLink
              to="/browse/drama"
              className="nav-link"
              activeClassName="active"
            >
              Drama
            </NavLink>
          </li>
          <li className="nav-item">
            <NavLink
              to="/browse/comedy"
              className="nav-link"
              activeClassName="active"
            >
              Comedy
            </NavLink>
          </li>
          <li className="nav-item">
            <NavLink
              to="/browse/scifi"
              className="nav-link"
              activeClassName="active"
            >
              Sci Fi
            </NavLink>
          </li>
          <li className="nav-item">
            <NavLink
              to="/browse/fantasy"
              className="nav-link"
              activeClassName="active"
            >
              Fantasy
            </NavLink>
          </li>
        </ul>
```

```
        <div>
          <BrowseList
            movieList={moviesInGenre}
            wishlist={wishlist}
            movieActions={movieActions}
          />
        </div>
        <div className="footer">
          <div>
            <a href="/T&C">Terms &amp;
Conditions</a>
          </div>
          <div>
            <a href="/privacy">Privacy Policy</a>
          </div>
          <div>© Movie Wishlist 2019</div>
        </div>
      </div>
    );
};

MovieBrowser.propTypes = {
  history: PropTypes.object.isRequired,
  match: PropTypes.object.isRequired,
  wishlist: PropTypes.object.isRequired,
  addToWishlist: PropTypes.func.isRequired,
  removeFromWishlist: PropTypes.func.isRequired
};

export default MovieBrowser;
```

[0:58] It's a functional component, so the JSX being returned here is what gets rendered into our page. Looking at our code, this `div` up here is containing that area that we decided was going to be our header.

[1:15] Let's add a `role` of `banner`.

```
<div role="banner" className="navbar navbar-dark
bg-primary">
  <span className="navbar-text">Browse
Movies</span>
  <button className="btn btn-outline-secondary"
onClick={goToWishlist}>
    {"< Back"}
  </button>
</div>
```

We had decided that this `button` should have a `role` of `navigation`, since it functionally navigates us back to the previous page. Let's wrap that in a `div` with `role` navigation.

```
<div role="banner" className="navbar navbar-dark
bg-primary">
  <span className="navbar-text">Browse
Movies</span>
  <div role="navigation">
    <button className="btn btn-outline-
secondary" onClick={goToWishlist}>
      {"< Back"}
    </button>
  </div>
</div>
```

Now, let's add our `div` with `role` `main`, and make sure we wrap the rest of our page content with that `div`.

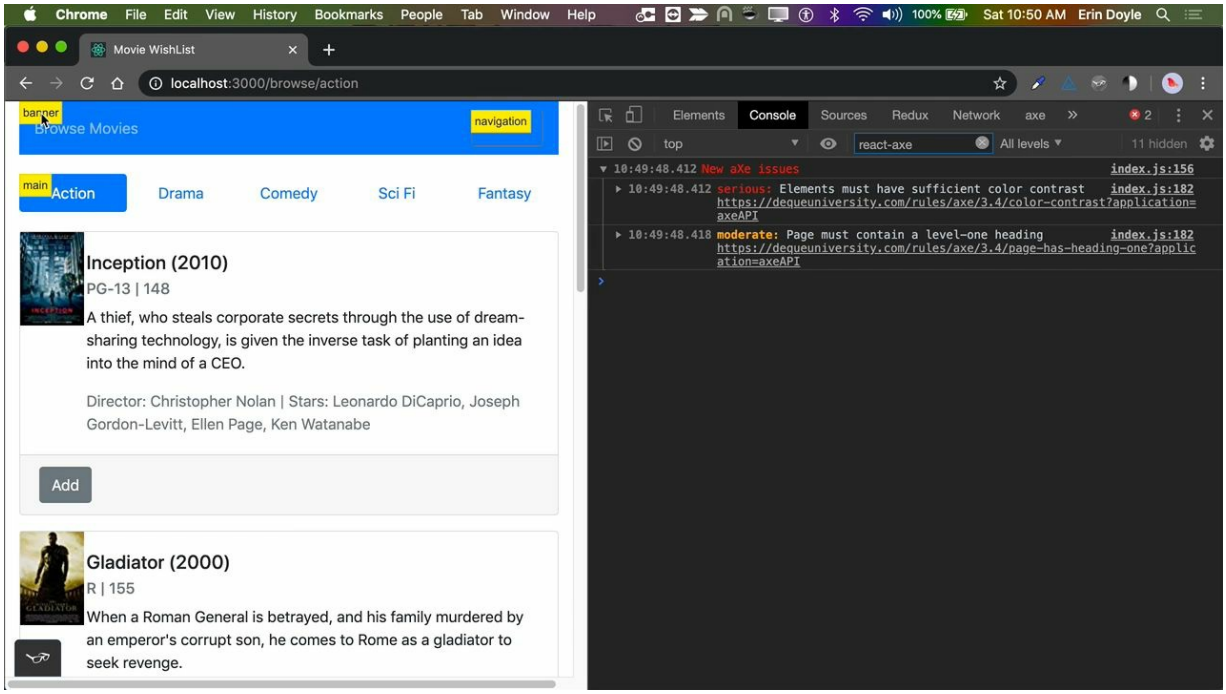
```
<div role="main">
  <ul className="nav nav-pills nav-justified">
    <li className="nav-item">
      <NavLink
        to="/browse/action"
        className="nav-link"
        activeClassName="active"
      >
        Action
      </NavLink>
    </li>
  </ul>
</div>
```

[1:57] Finally, down here is our `footer`. We just need to add a `role`.

```
<div role="contentinfo" className="footer">
  <div>
    <a href="/T&C">Terms &amp;amp; Conditions</a>
  </div>
  <div>
    <a href="/privacy">Privacy Policy</a>
  </div>
  <div>© Movie Wishlist 2019</div>
</div>
```

For the aria landmark `role`, a footer is represented by `contentinfo`. Now, let's see what that looks like. The findings that we were previously seeing being reported by `react-axe` are no longer being reported.

[2:17] If we use `tota11y` to annotate the landmark regions of the page, we should now see annotations, and there they are. We've got our `banner`, our `navigation`, our `main`, and down here, our `contentinfo`.



If we go back over to Safari, run Voice-over, and check the landmarks menu of the rotor...

- Voice-over: [2:41] Voice-over landmarks menu.

Instructor: [2:43] Before we were not even seeing the landmark menu in the rotor. Now, it's showing, and we see all of the landmark regions we expect to. We've got our banner...

- Voice-over: [2:52] `banner`.

Instructor: [2:53] our navigation...

- Voice-over: [2:54] `navigation`.

Instructor: [2:55] our main...

- Voice-over: [2:55] `main`.

Instructor: [2:56] and content information.

- Voice-over: [2:57] content information. Voice-over off.

Define Landmark Regions of a web page using HTML5 sectioning elements

Instructor: [00:00] In a previous lesson, we defined landmark regions using ARIA roles. Now we're going to go ahead and edit those landmark regions to instead use HTML5 sectioning elements. Here we are again in our React component that defines these landmark regions.

MovieBrowser.js

```
import React from "react";
import PropTypes from "prop-types";
import { NavLink } from "react-router-dom";

import movies from "../movies";

import BrowseList from "./BrowseList";
import getBrowseActions from
"./getBrowseActions";

const MovieBrowser = ({
  history,
  match,
  wishlist,
  addToWishlist,
  removeFromWishlist
}) => {
  const goToWishlist = () =>
history.push("/wishlist");
  const movieActions =
```

```
getBrowseActions(addToWishlist,
removeFromWishlist);
  const moviesInGenre =
movies[match.params.genre];

  return (
    <div>
      <div role="banner" className="navbar
navbar-dark bg-primary">
        <span className="navbar-text">Browse
Movies</span>
        <div role="navigation">
          <button className="btn btn-outline-
secondary" onClick={goToWishlist}>
            {"< Back"}
          </button>
        </div>
      </div>

      <div role="main">
        <ul className="nav nav-pills nav-
justified">
          <li className="nav-item">
            <NavLink
              to="/browse/action"
              className="nav-link"
              activeClassName="active"
            >
              Action
            </NavLink>
          </li>
          <li className="nav-item">
            <NavLink
              to="/browse/drama"
              className="nav-link"
              activeClassName="active"
```

```
            >
              Drama
            </NavLink>
          </li>
          <li className="nav-item">
            <NavLink
              to="/browse/comedy"
              className="nav-link"
              activeClassName="active"
            >
              Comedy
            </NavLink>
          </li>
          <li className="nav-item">
            <NavLink
              to="/browse/scifi"
              className="nav-link"
              activeClassName="active"
            >
              Sci Fi
            </NavLink>
          </li>
          <li className="nav-item">
            <NavLink
              to="/browse/fantasy"
              className="nav-link"
              activeClassName="active"
            >
              Fantasy
            </NavLink>
          </li>
        </ul>

        <div>
          <BrowseList
            movieList={moviesInGenre}
```

```
            wishlist={wishlist}
            movieActions={movieActions}
          />
        </div>
      </div>

      <div role="contentinfo"
  className="footer">
        <div>
          <a href="/T&C">Terms &amp;
  Conditions</a>
        </div>
        <div>
          <a href="/privacy">Privacy Policy</a>
        </div>
        <div>© Movie Wishlist 2019</div>
      </div>
    </div>
  );
};

MovieBrowser.propTypes = {
  history: PropTypes.object.isRequired,
  match: PropTypes.object.isRequired,
  wishlist: PropTypes.object.isRequired,
  addToWishlist: PropTypes.func.isRequired,
  removeFromWishlist: PropTypes.func.isRequired
};

export default MovieBrowser;
```

[00:16] Here we have a `div` with `role` of `banner`. That is equivalent to the HTML5 element `header`. Here we have `div` with `role` of `navigation`. That is equivalent to the HTML 5 element `nav`. Here we

have a `div` with `role` of `main`, and that is equivalent to the `main` element. Finally, here is our `div` with `role contentinfo`, and that is equivalent to `footer` element.

```
<div>
  <header className="navbar navbar-dark bg-
primary">
    <span className="navbar-text">Browse
Movies</span>
    <nav>
      <button className="btn btn-outline-
secondary" onClick={goToWishlist}>
        {"< Back"}
      </button>
    </nav>
  </header>

  <main></main>
  <footer></footer>
</div>
```

[00:51] Now that we've changed those, let's go ahead and double check and make sure that we are still passing all of our accessibility audits. `react-axe` is still not reporting any findings concerning `landmark` regions. If we check voiceover in Safari, and run the rotor...

- Voiceover: [01:09] Landmarks menu.

Instructor: [01:10] we still see all of the landmark regions we expect.

- Voiceover: [01:13] banner.

Instructor: [01:14] We've still got our banner...

- Voiceover: [01:15] navigation.

Instructor: [01:16] our navigation.

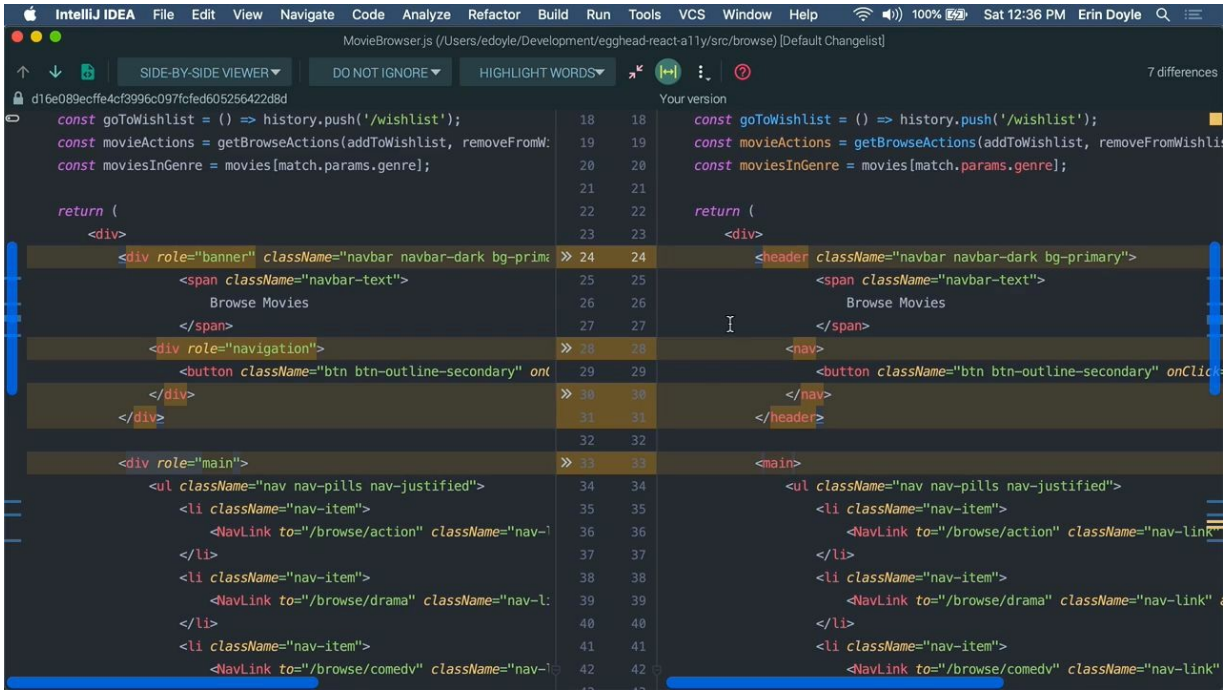
- Voiceover: [01:17] main.

Instructor: [01:18] our main.

- Voiceover: [01:18] Content information.

Instructor: [01:20] and content information. We should prefer HTML5 element over ARIA attributes when there is redundant functionality provided by each. That's because HTML5 provides much more semantic elements.

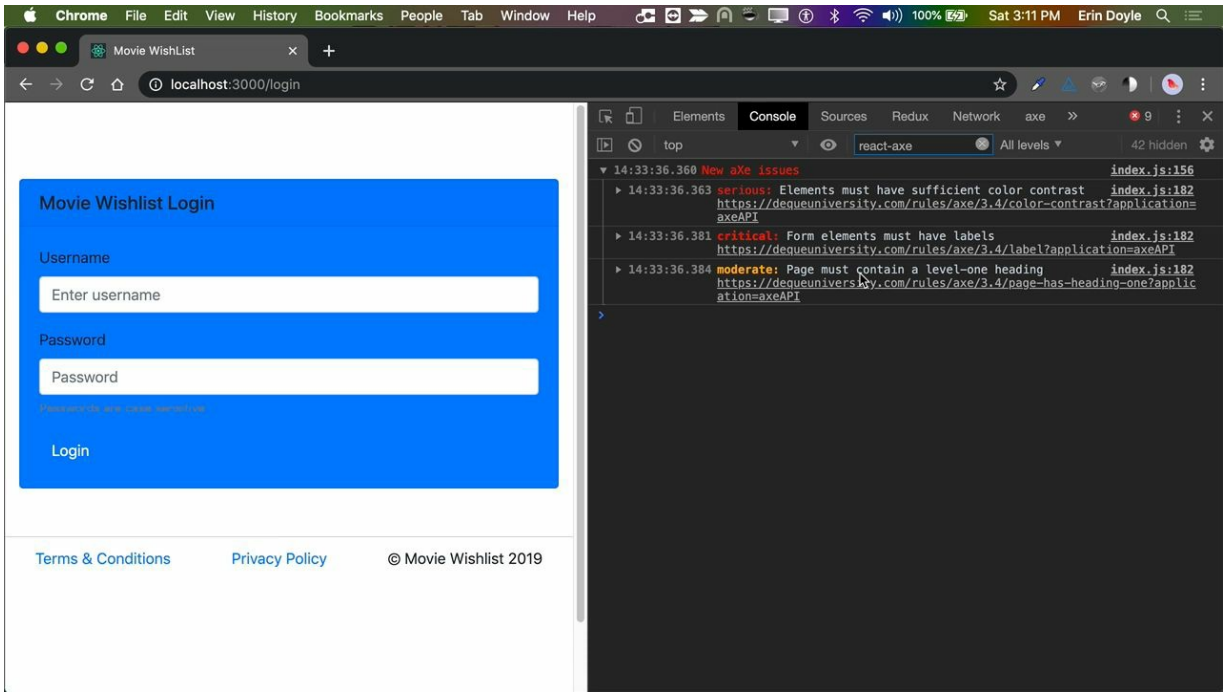
[01:36] If we look at a diff comparing these two, with our ARIA roles here on the left and our HTML5 elements here on the right, you can see that the HTML5 elements just provide much more clean and easier to read code.



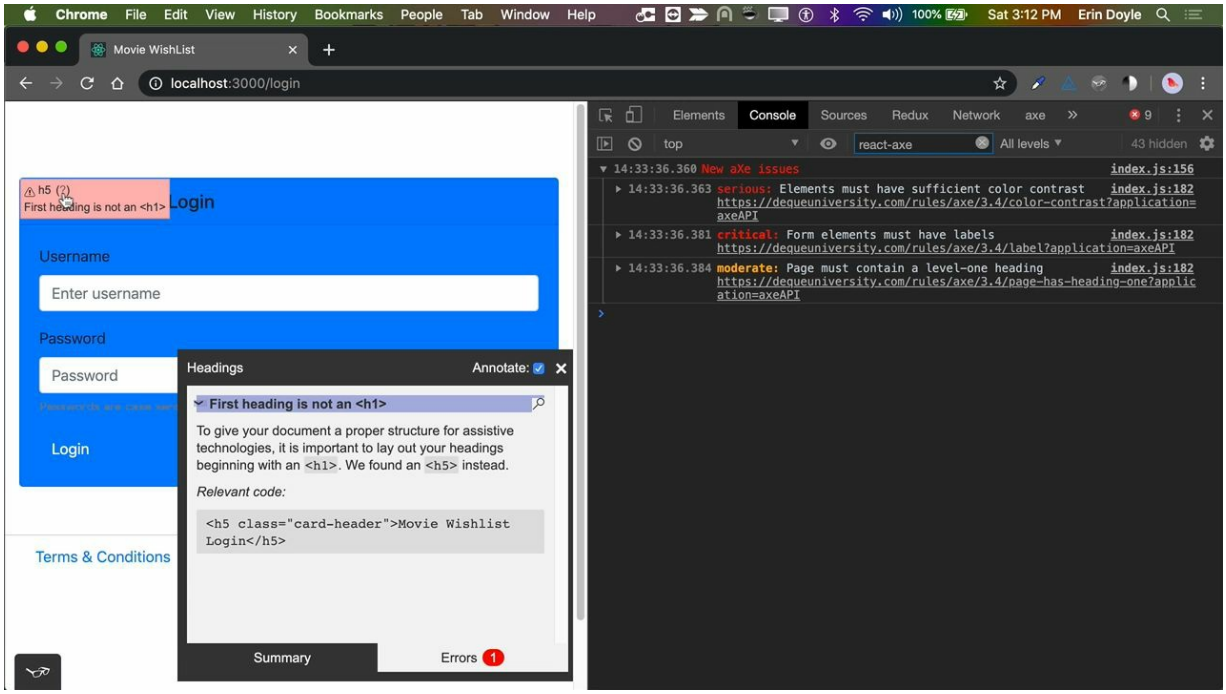
[01:50] Instead of having to scroll through all of these divs with roles, we've got elements that are very explanatory.

Test for Heading Level Accessibility Issues

Instructor: [00:00] Here I have a web page that has some accessibility issues concerning heading levels. Over here, I've got `react-axe` reporting any findings in the console. We can see we have a finding right here -- `page must contain a level one heading`. That means that this page does not have an `h1` element in it.

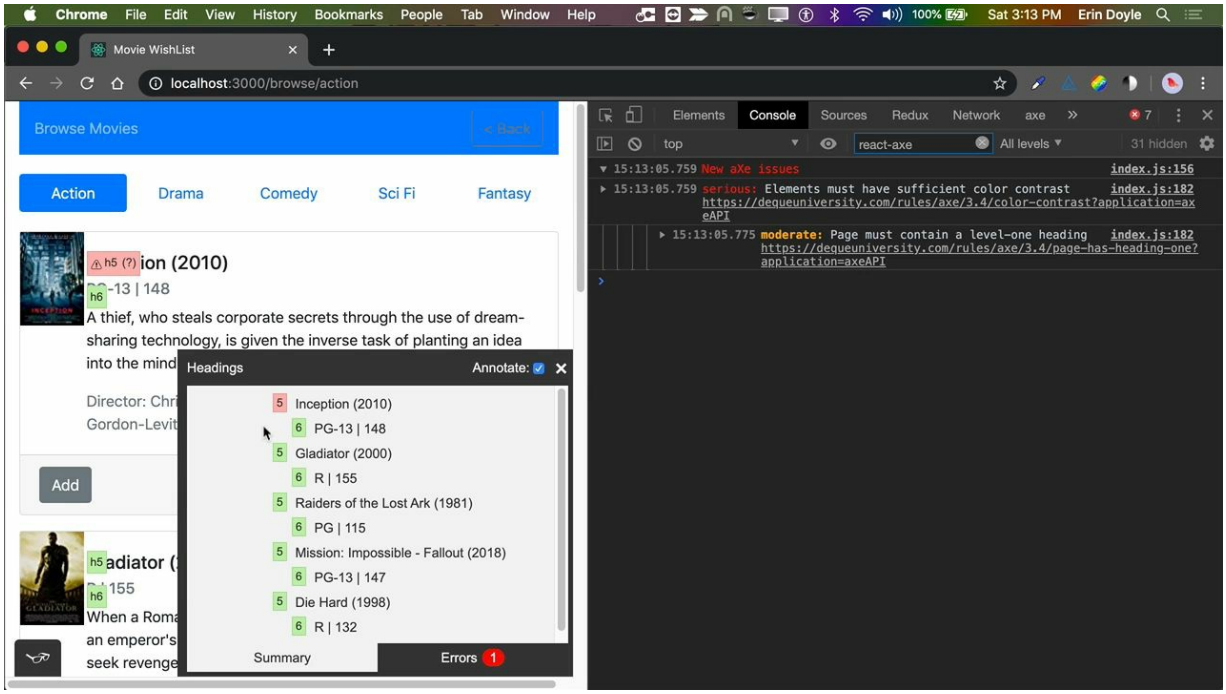


[00:18] We can also use `tota11y` to annotate the heading levels on the page. This will show us in red any levels that have an issue. This is telling us that the only heading level we have is actually an `h5`, not an `h1`.



[00:34] All pages should have at least one `h1` heading level, and they should start with each one and be contiguous from that point on. The fact that we don't have an `h1` and the only heading level that we do have is an `h5` is definitely an issue.

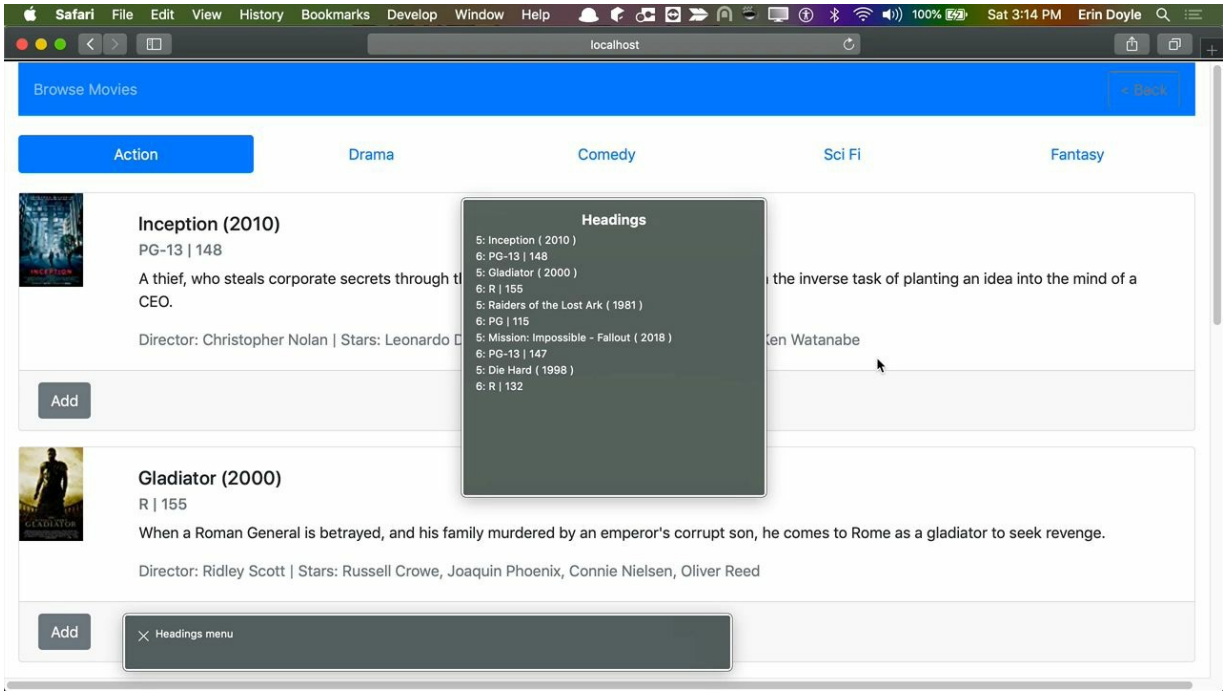
[00:53] Here's another page we can look at to see other issues with heading levels. Here `react-axe` is telling us the same thing -- `page must contain a level one heading`. If we use `tota11y` to annotate the heading levels here, we do not have an `h1` heading level, we start with `h5`, and we go from there.



Let's see what this looks like in VoiceOver's rotor.

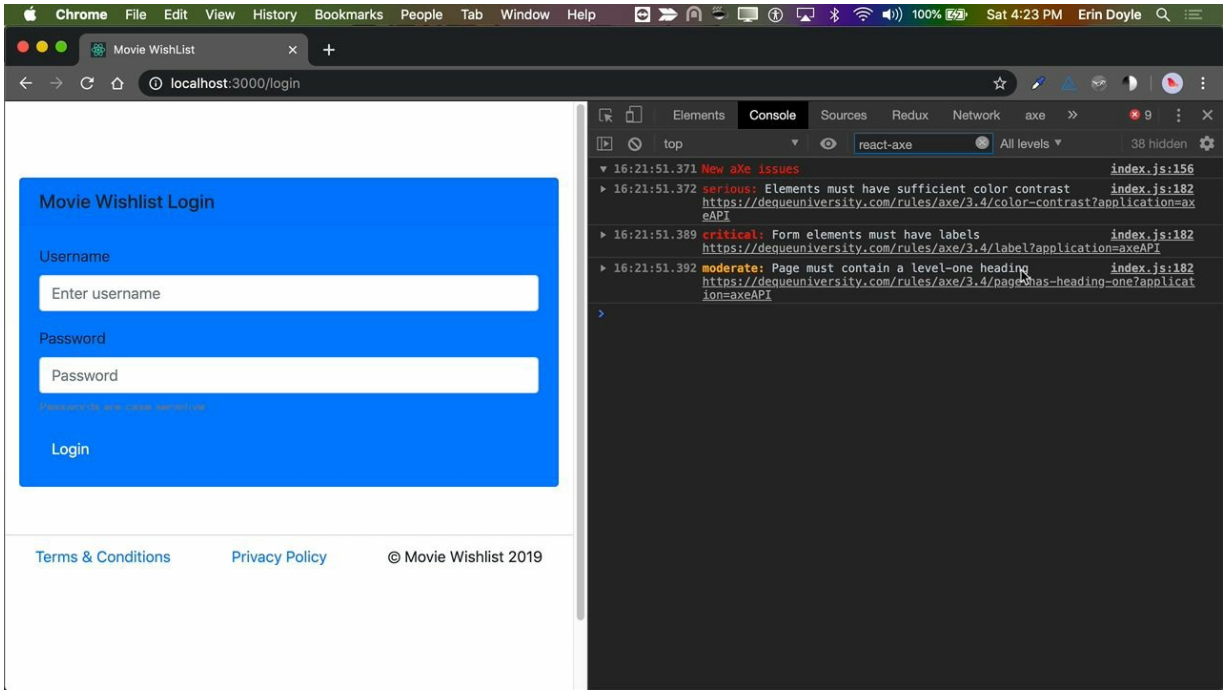
- VoiceOver: 01:24 Headings menu.

Instructor: [01:26] Here's the headings menu, and the same thing as we saw in `totally`. We start with a level 5 and we have only 5's and 6's for this entire page.



Correctly Define Heading Levels of a Web Page

Instructor: [00:00] Here's a web page that has some accessibility issues with heading levels. Here on the side, I've got `react-axe`, which is logging any findings to the console. We've got this one here, `Page must contain a level-one heading`



[00:13] If run `tota11y` and annotate headings, we can see that the very first heading level is an `h5`. We need that to be an `h1`. Let's look at the code. Here's that `h5` that we saw being reported by `tota11y`.

Login.js

```
return (
  <div className="login row align-items-center">
      <div className="col-12 col-md-9 col-xl-8
py-md-3 pl-md-5">
          <header>
              <h5 className="card-header">Movie
Wishlist Login</h5>
          </header>
    ...
)
```

[00:26] If we consider the structure of the page and the meaning we want to convey, with this being the first page of our web application, and that web application being the movie wishlist, really, we could consider `Movie Wishlist` to be our top-level heading. It's the title of the application.

[00:45] We could consider `Login` to be a sub-level to that, as it's the title of the page. What we can do here, instead of making this whole thing an `h1`, let's separate these two heading levels out from each other. Let's move the header up here above the card.

```
<div className="login row align-items-center">
  <div className="col-12 col-md-9 col-xl-8 py-
md-3 pl-md-5">
    <header>Movie Wishlist</header>
  </div>
</div>
```

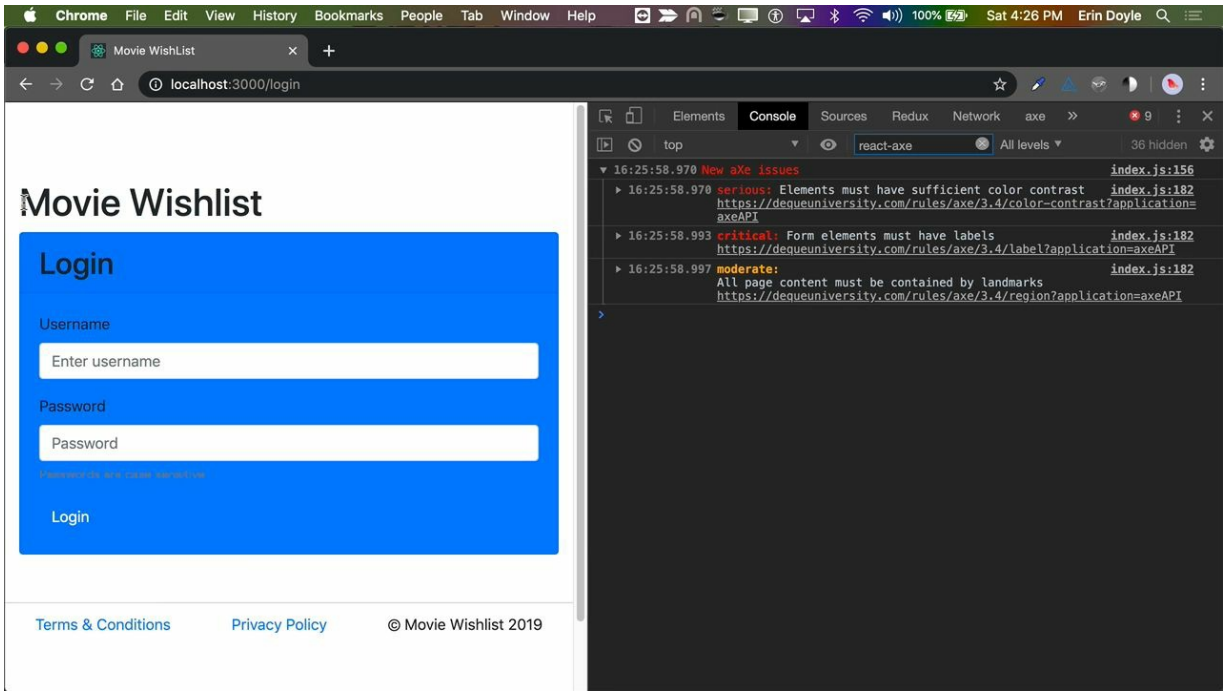
[01:01] We're going to put the application title here in the `h1`. That's Movie Wishlist. Now, we can make our `Login` title an `h2`.

```
<div className="login row align-items-center">
  <div className="col-12 col-md-9 col-xl-8 py-
md-3 pl-md-5">
    <header>
      <h1>Movie Wishlist</h1>
    </header>

    <div className="card bg-primary">
      <h2 className="card-header">Login</h2>
    </div>
  </div>
</div>
```

Let's see what this looks like. The finding that `react-axe` was reporting about missing a heading level one is now gone.

[01:20] We can now see the way we've reorganized the page a little bit. Here is our application title, and here's our page title.



I think that this makes more sense visually and from the perspective of the assistive technologies, because the heading level is meant to convey the structure of the page.

[01:36] Let's look at another page with heading level issues. Again, `react-axe` is showing us that this page does not have a level one heading. `tota11y` shows us that the first heading level is an `h5`. Let's look at the code for this.

MovieBrowswer.js

```
<header className="navbar navbar-dark bg-
primary">
  <span className="navbar-text">Browse
Movies</span>
  <nav>
    <button className="btn btn-outline-
secondary" onClick={goToWishlist}>
      {"< Back"}
    </button>
  </nav>
</header>
```

[01:56] Looking at the code in our `header`, we don't even have a heading level within the `header`. Once again, the heading levels are meant to convey the structure of the page. Really, the page title should be an `h1`. Let's go ahead and add that.

```
<header className="navbar navbar-dark bg-
primary">
  <span className="navbar-text">
    <h1>Browse Movies</h1>
  </span>
  <nav>
    <button className="btn btn-outline-
secondary" onClick={goToWishlist}>
      {"< Back"}
    </button>
  </nav>
</header>
```

[02:12] Looking through the rest of the page, the `h5` being reported are within our movie component. Here's that `h5`, and here's the `h6`.

Movie.js

```
<h5 className="card-title">{name} ({year})</h5>
<h6 className="card-subtitle mb-2 text-muted">
{rating} | {runtime} { genre ? `| ${genre}` :
null }</h6>
```

If we look at what information these two heading levels contain, the `h5` contains the `name` and `year` of the movie, which we can see right here.

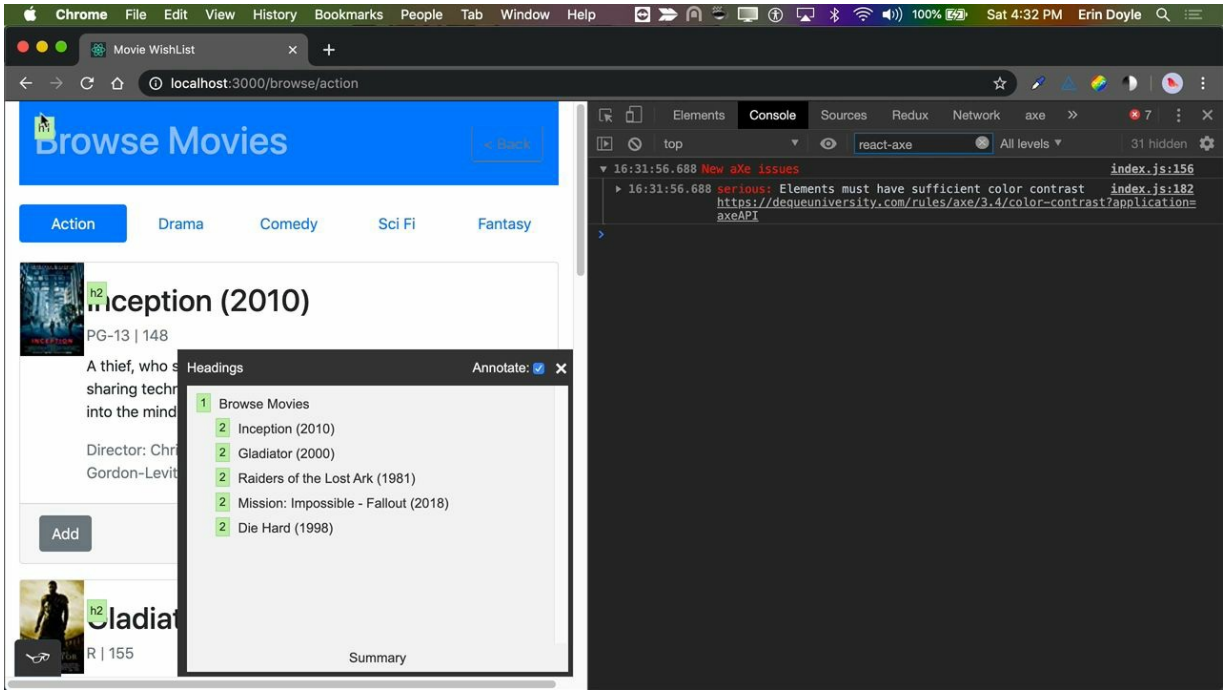
[02:32] It makes sense that that's a heading level, because we have a section for each movie. Let's make that an `h2`, as that is the next contiguous heading level after our `h1`. Now, let's look at what's in the `h6`. We've got the `rating`, the `runtime`, and sometimes the `genre`.

[02:53] That's this line right here, and that's not really heading level. That's just information. Heading level was used originally here to achieve a certain font size, which is not what heading levels are for. They're to convey the structure of the page.

[03:06] Instead, let's just make this a `p` tag.

```
<h2 className="card-title">{name} ({year})</h2>
<p className="card-subtitle mb-2 text-muted">
{rating} | {runtime} { genre ? `| ${genre}` :
null }</p>
```

Now, let's see what that looks like. Once again, our `react-axe` findings about heading levels are gone. If we run `tota11y` and annotate the heading levels, we can see that they now start with an `h1`, continue with `h2`, and everything is green and appropriate.



[03:27] Finally, if we go to voiceover in Safari and look at the rotor...

- Voiceover: [03:32] Headings menu.

Instructor: [03:33] We now can see...

- Voiceover: [03:34] Heading level one. Browse movies.

Instructor: [03:36] each of the heading levels...

- Voiceover: [03:38] Heading level two, four items.

Instructor: [03:39] is appropriate.

- Voiceover: [03:40] Heading level two, four items. Heading level two, four items. Heading level two, four items. Heading level two, four items. Die Hard.

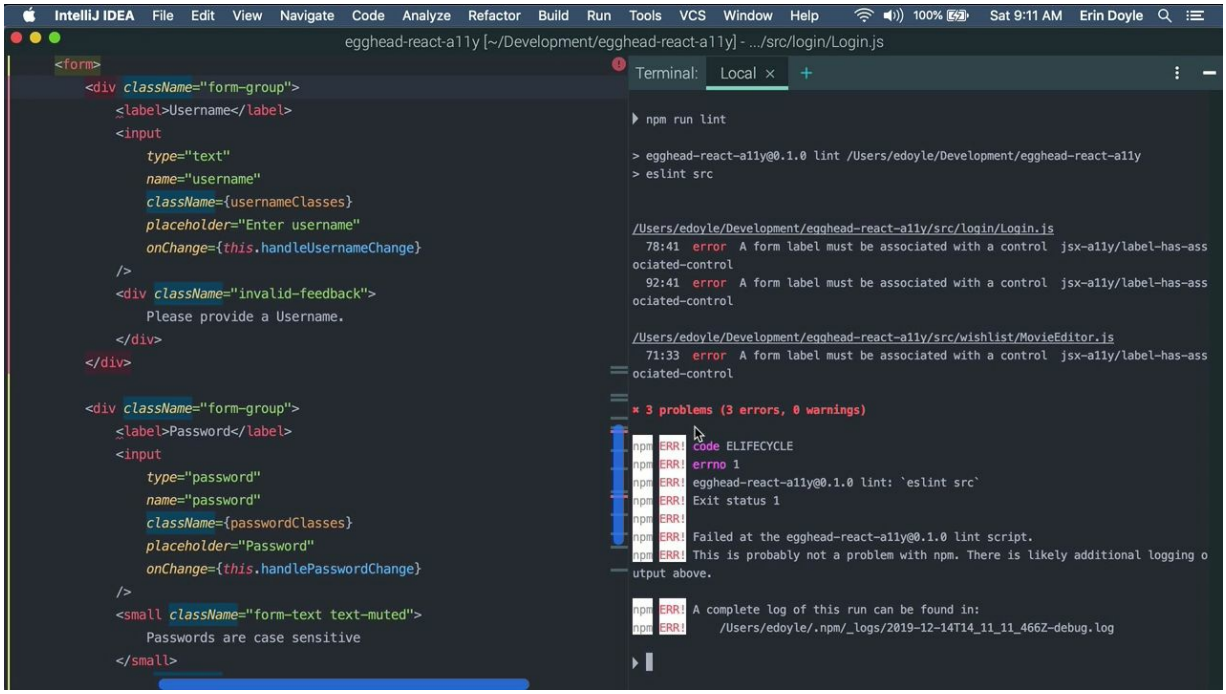
Test for Form Control Label Accessibility Issues

Instructor: [0:00] When it comes to testing your application for issues concerning accessible labels, we can start right in our terminal by running ESLint.

```
npm run lint
```

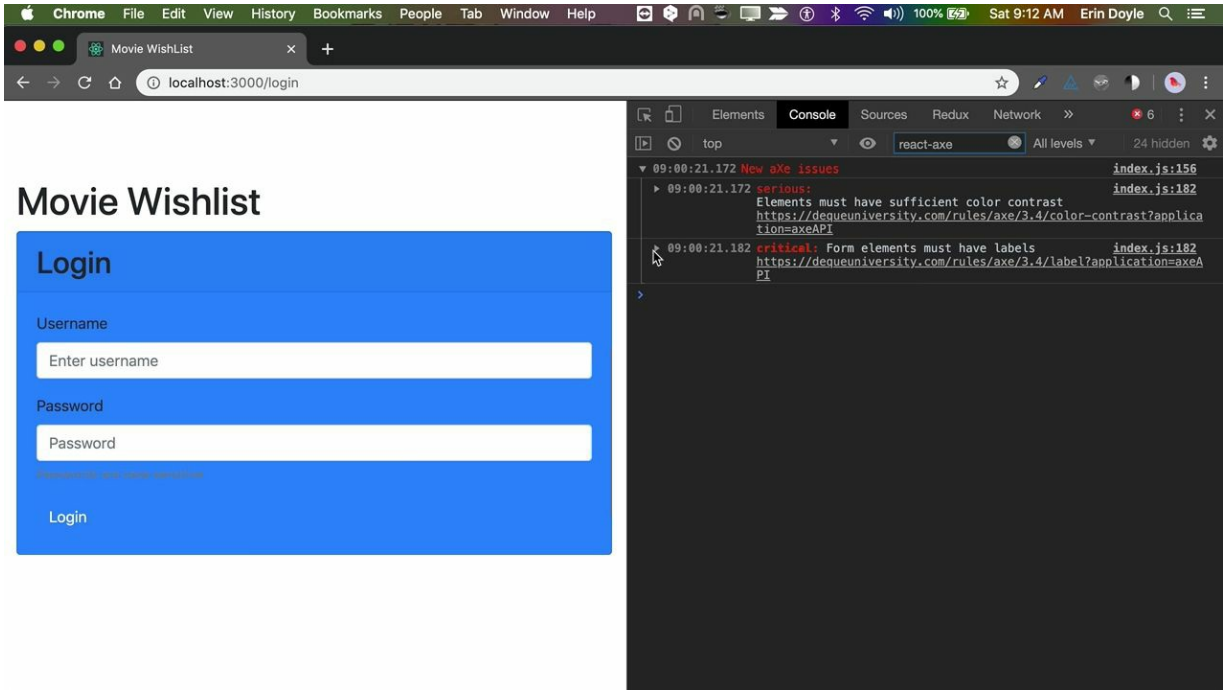
If you've got the `eslint-plugin-jsx-a11y` installed, then we will get some findings when we are missing appropriate labels.

[0:18] Right here, we've got a couple findings on one form and here we have a finding on another form where we are missing labels associated with their form controls.



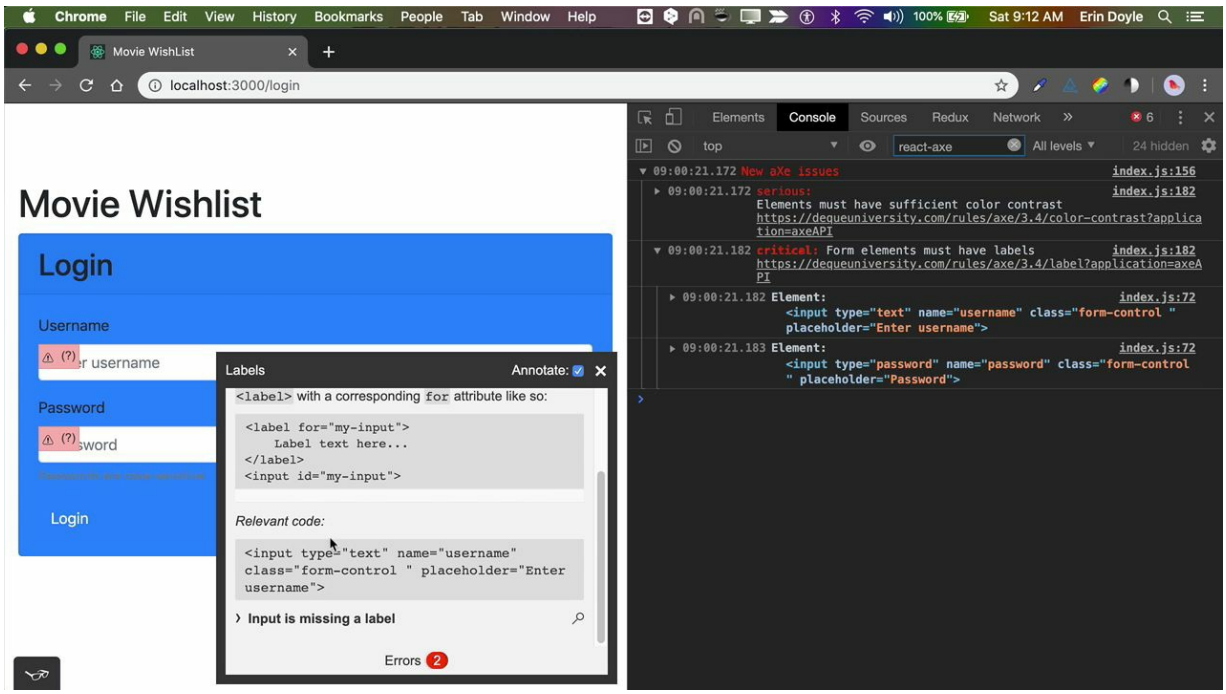
If you're using an IDE that's integrated with ESLint, then you can see right in the code if you have any of these findings.

[0:35] I'm running the same application right now with `React aXe`. It also will show me if we have any findings with labels.



It will highlight each input that is missing an associated label. If we run Totally, we can annotate for missing labels.

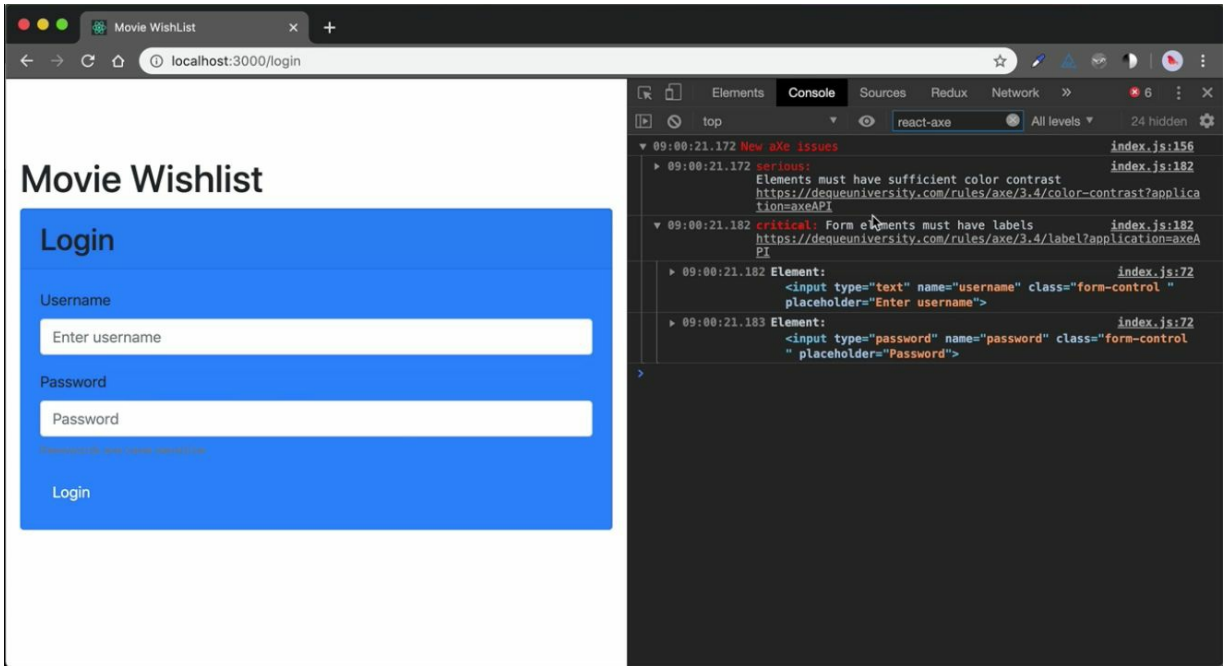
[0:55] Right here, those same two form inputs are annotated. It shows me the source code right here as well.



Our auditing tools make it really easy to find issues concerning form labels.

Ensure Form Controls have Accessible Labels

Instructor: [0:00] Here I have a form that is missing accessible labels for each of the form inputs. We can see this here from our `React aXe` finding here in the console.



Let's look at the code for this. Here's the code for that form.

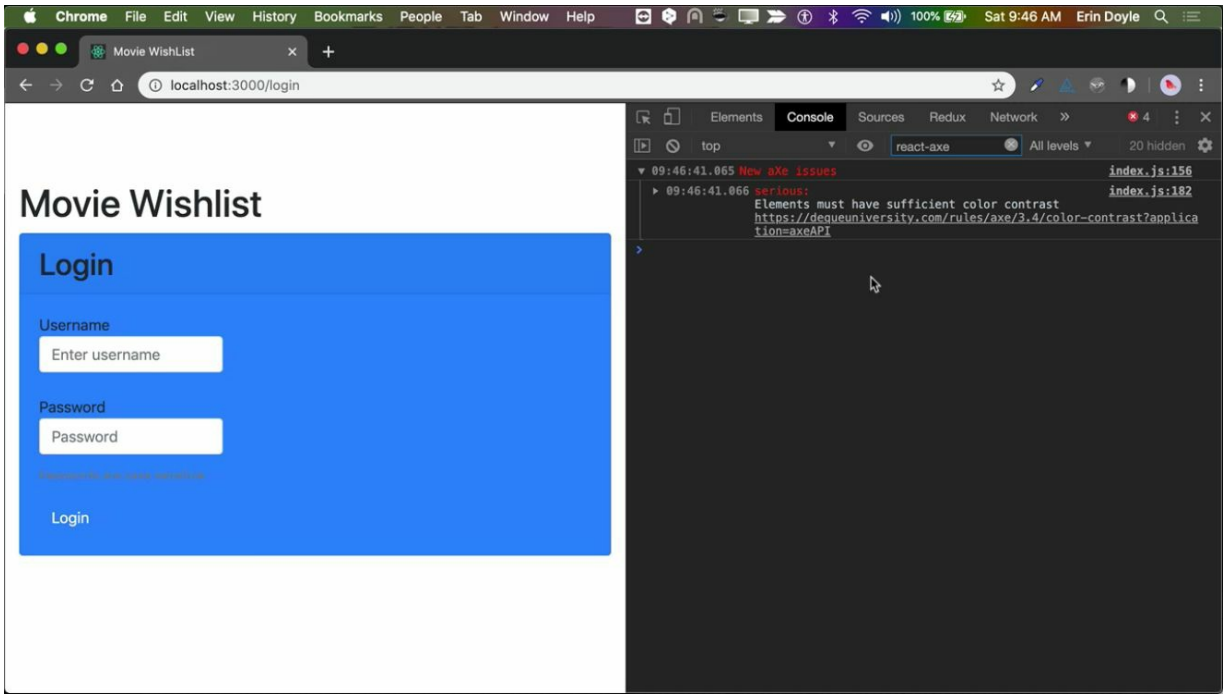
[0:15] I at least already have visual labels using label elements for each of the inputs that they should be labeling. The only thing left to do is to associate the label with the input to which it applies. We've got two options for doing that. If possible, we can wrap the input with the label.

```
<label
  >Password
  <input
    type="password"
    name="password"
    className="passwordClasses"
    placeholder="Password"
    onChange="{this.handlePasswordChange}"
  />
</label>
```

[0:42] If we check that out by running `ESLint` real quick, let's see if those findings have been resolved. We still have one in a different file, but the findings that we had originally for this file have now been resolved. We

can also see that over here with `React aXe` where we're no longer seeing that finding.

[1:01] However, my display has been impacted.



Now that I'm wrapping these inputs with the label elements, it's restricting the width of these inputs. I can probably fix that with CSS, but there are lots of reasons why perhaps I don't want to wrap my input element. Or maybe the label is located a little somewhere else in the structure of my page.

[1:23] If we don't want to nest our inputs within our labels, we can associate them by adding an ID to our input and adding an htmlFor attribute to the label that references that ID of the input.

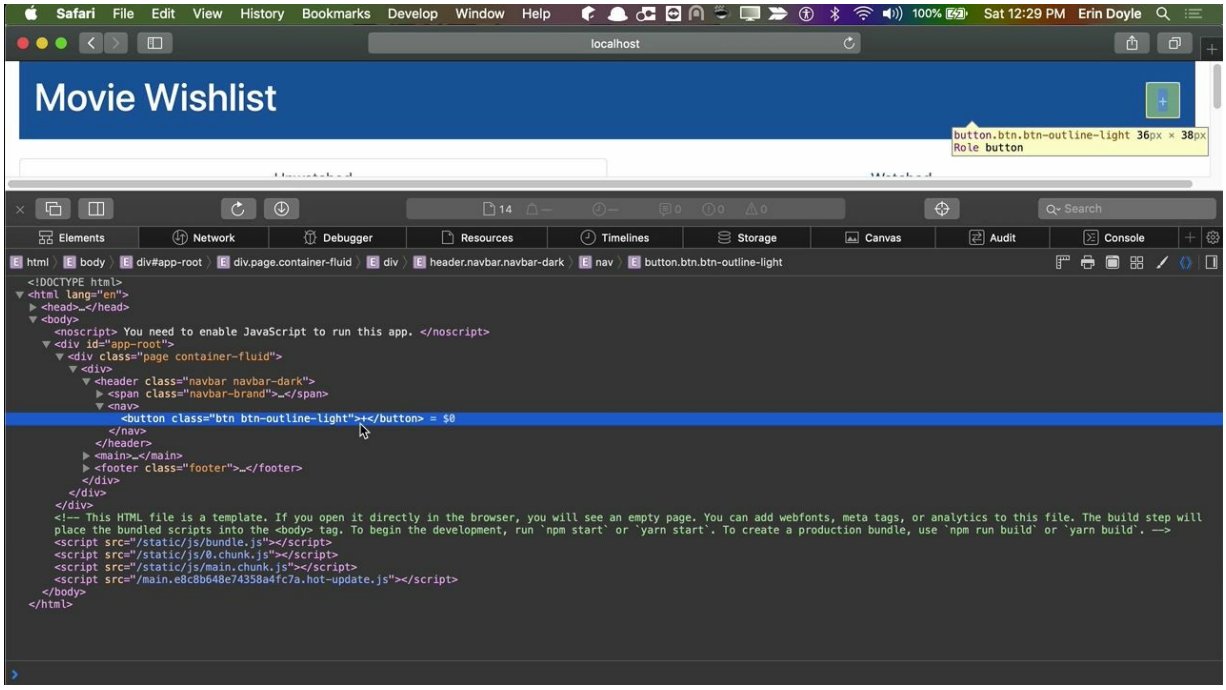
```
<label htmlFor="password-input">Password</label>
<input
  id="password-input"
  type="password"
  name="password"
  className="passwordClasses"
  placeholder="Password"
  onChange="{this.handlePasswordChange}"
/>
```

[1:41] One thing to note. In plain old HTML, this attribute would be for with a lower case F (`htmlfor`). Because we're using React, instead we use the attribute htmlFor in camelCase. Let's do the same thing here for the username input.

[2:03] If we run `ESLint` again, we still are no longer seeing any findings for the login.js file. If we run `React aXe` again, we no longer see a finding here either. Now my inputs are now back to their full width as they were originally designed. Just to double-check, let's look at Totally and annotate any missing labels. We don't have any annotations because we are no longer missing any labels.

Add Accessible Labels to Elements Whose Labels are Not Clear Enough

Instructor: [0:00] Here I have an application that has some known issues with accessible labels. The problem we have here is that we have a button on each of these pages. You can see in the source for this button that its text is just a plus sign.



[0:15] Those of us who can see are very used to seeing buttons that have a plus sign to indicate that you're going to add something. Over here with this button, we have this chevron pointing back. Again, for those of us who can see, we're used to seeing this to indicate we're going back to the previous page.

[0:37] The problem with these buttons is while they work well for those of us who can see and we can visually understand their meaning, that meaning is not conveyed as clearly to those using screen readers. Let's demonstrate that. I'm in Safari. I'm going to go ahead and run voiceover.

Voiceover: [0:54] Plus button.

Instructor: [0:56] As you can hear, the screen reader reads this as "plus button." That's not very descriptive for those people who can't see the button and probably don't know what plus button indicates or what to expect the functionality to be from that button. If we go to the browse page, let's hear what this button sounds like.

Voiceover: [1:15] Less back button.

Instructor: [1:17] This button is read as "less back" because the screen reader is reading this character as a less sign. Combining the two, it says, "Less back." That definitely is not going to make sense to those using a screen reader. If they hear "less back button," that is not conveying to them the functionality that we're going to go back to the previous page.

[1:39] Both of these buttons need accessible labels because their own text is not descriptive enough to convey what functionality we can expect these buttons to have. One more test real quick. Let's see what's shown in the rotor for these buttons.

Voiceover: [1:52] Less back button.

Instructor: [1:53] Same thing, less back button. On our previous page...

Voiceover: [1:56] Form controls menu. Plus button.

Instructor: [1:58] Plus button. Whether navigating to the button or using the rotor, it is not clear to users what this button is going to do. Let's fix this.

Voiceover: [2:07] Voiceover off.

Instructor: [2:08] Here's that component that we were just looking, that had the plus button. As we could see from looking at the source previously, the text for the button is just a plus sign. We want to give this button a label that says that we're adding a movie when you click this button.

[2:23] We can do that by adding an aria-label attribute. We can be as verbose as we want with this. This is what's going to be read to a screen reader user when they focus on this button. We want to tell them what this button is. Really what it is is a button to add a movie.

```
<nav>
  <button
    className="btn btn-outline-light"
    aria-label="Add a Movie"
    onClick={goToBrowse}
  >
    +
  </button>
</nav>
```

[2:38] Now if we look at the component for the browse page button, here it is right here. There is our less back text. Let's give this an aria-label saying, "Go back to wish list" because that's what's going to happen if you click that button. Now let's hear how that sounds when read by the screen reader.

```
<nav>
  <button
    className="btn btn-outline-light"
    aria-label="Back to Wish List"
    onClick={goToWishlist}
  >
    {'< Back'}
  </button>
</nav>
```

Voiceover: [2:58] Add a movie button.

Instructor: [2:59] Now the button says, "Add a movie." That is a lot more useful. Let's check that out with the rotor.

Voiceover: [3:04] Form controls menu. Add a movie button.

Instructor: [3:06] There it is again. If we're navigating through the rotor and we're going through the buttons and we hear "Add a movie," that's very clear now what we can expect that button to do. Let's check out the button on the browse page.

Voiceover: [3:17] Go back to wish list button.

Instructor: [3:19] Now that says, "Go back to wish list button." Very helpful.

Voiceover: [3:22] Go back to wish list button.

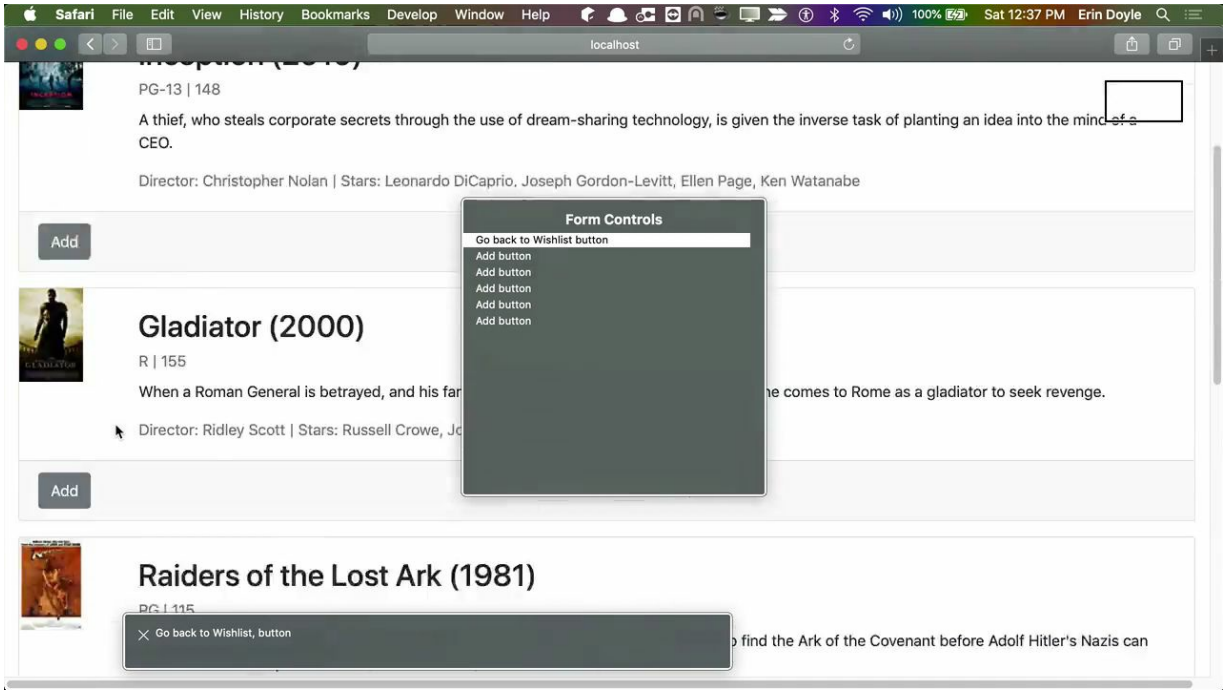
Instructor: [3:24] Those buttons are now dramatically more useful to a screen reader than they were before just by adding an aria-label attribute.

Add Accessible Labels to Provide Elements with More Context

Instructor: [0:00] Here I have an application that has some known issues with accessible labels. Here I have Safari open. If I open the rotor in voiceover, we can demonstrate the issue.

Voiceover: [0:10] Form controls menu.

Instructor: [0:12] If you look at this page, we have a number of buttons, one for each movie that allows you to add that movie to your wish list.



However, in the rotor you just see the same text over and over again. If we were to go through this menu...

Voiceover: [0:29] Add button. Add button. Add button. Add button. Add button.

Instructor: [0:33] we would just hear "Add button" over and over again. There's no additional context telling us what are we adding. We don't even know that this button, if we click it, is going to add a specific movie. We certainly don't know which of those movies it's going to add. This is completely unusable for a screen reader user at this point. Again, this is something we can easily fix.

Voiceover: [0:54] Voiceover off.

Instructor: [0:56] Here's the component that generates that button. When the movie is already in the wish list, the button is going to say remove. When it's not, it's going to say add. That gets passed into this movie toolbar button component. Let's look at that.

[1:10] We're already passing down that button text, which is either going to be add or remove. We also have our movie title available here. We can make use of that.

[1:18] Let's give this a more accessible label using `aria-label`. We can use that button text. It's going to be add or remove, depending, and then the name of the movie.

```
// src/primitives/MovieToolbarButton.js
return (
  <button
    className="btn btn-secondary"
    aria-label={ariaLabel}
    onClick={clickHandler}
  >
    {buttonText}
  </button>
)
```

Let's go ahead and try that out. If I run the rotor, here are all our buttons.

Voiceover: [1:36] Form controls menu. Add "Inception" button.

Instructor: [1:38] Add Inception.

Voiceover: [1:38] Add "Gladiator" button.

Instructor: [1:40] Add Gladiator.

Voiceover: [1:41] Add "Raiders of the Lost Ark" button. Add "Mission Impossible - Fallout" button. Add "Die Hard" button.

Instructor: [1:46] Now we can clearly hear which movie each of these buttons is going to add. Just by adding that `aria-label` attribute, we've dramatically increased the accessibility of this page.

Add an Accessible Label to an Element from the Text of Other Elements

Instructor: [00:00] Here is a page that has a problem we need to fix concerning accessible labels. I'm in Safari. I'm going to go ahead and run VoiceOver so we can hear the problem. When this page loads and there are no movies in my wish list, this message is shown, stating that "You don't have any movies in your wish list. Here is a link to go add some."

[00:20] That takes you to the page where you can go pick what movies you want to add to your wish list. When this text is read by the screen reader...

Screen Reader: [00:29] No movies in your wish list. You are currently on a text element. Visited link. Add some. You are currently on a link. To click this link, press control-option-space.

Instructor: [00:40] The way that's read feels disconnected to me. I feel like it would be a better experience if those two sentences were read together as a group. It would say "No movies in your wish list. Add some" instead of the first sentence and then the second sentence with the link read separately in a disconnected way.

[00:57] We can group these two elements together and give them a label that makes sure that they are read together. Let's see how to do that.

Screen Reader: [01:04] VoiceOver off.

Instructor: [01:05] Here's the component where we have this message. This is the JSX that's returned when there are no movies yet in the wish list. It's just a p tag. Within it, we have a link. We can reorganize this to be in a group. We can surround this with a div.

[01:25] Then changing this from a `p` tag, instead we're going to make this a span. We've got two elements. We've got our span. We've got our link. They are grouped within this div. If we give each of these an ID, then now we can add an `aria-labelledby` that will reference both of these elements.

[01:48] `aria-labelledby` can take a list of IDs delimited by a space. If we give it the noMovies ID and the addLink ID, it should now read the text from both of these elements to create the label for this div group. Let's hear what this sounds like.

```
<div aria-labelledby="noMoviesText addLink">
  <span id="noMoviesText">
    No Movies in your Wish List!
    <Link id="addLink" to="/browse">
      Add some!
    </Link>
  </span>
</div>
```

Screen Reader: [02:08] No movies in your wish list. Add some group. You are currently in a group.

Instructor: [02:13] Now these two elements are read as being a group. You heard the full two sentences read together as one statement instead of two separate statements where I had to move the screen reader cursor from one to the next and they sounded disconnected. Now they're read together. It makes a lot more sense.

Add Accessible Descriptions to Elements

Instructor: [0:00] Here, I have a form that has some issues when it comes to how screen readers announce things like help text on the fields, as well as any errors associated with each of the fields. Let's run VoiceOver and demonstrate with those issues sound like.

[0:18] If I use the screen reader to navigate down to the password field that has this help text that informs us that passwords are case-sensitive, let's hear how that's read.

Announcer: [0:30] Password, secure edit text. You are currently on a text field.

Instructor: [0:34] When the cursor focuses on the password field, we don't hear that message about passwords being case-sensitive. We don't actually hear that until we move the cursor out of the field.

Announcer: [0:46] Passwords are case-sensitive. You are currently on a text element.

Instructor: [0:50] By then, we've already presumably entered our password, and we're moving onto the login button. We really need a way to associate that descriptive, instructional information with the password field.

[1:03] Additionally, if we run our form validation...

Announcer: [1:06] Login button.

Instructor: [1:07] and trigger those errors with each of the fields, if I go back to the username field, for instance...

Announcer: [1:13] Username, edit text, enter username. You are currently on a text field.

Instructor: [1:18] Similarly, while I'm in the username field, I don't know that there's something wrong with it that I need to fix. If I go to the password field...

Announcer: [1:26] Password, secure edit text.

Instructor: [1:29] Once again, while I'm in the field, I don't know that there's a problem that I need to fix until I move the cursor out of the field.

Announcer: [1:35] Passwords are case-sensitive. Please provide a password.

Instructor: [1:39] All of this important information that we need to give the user while they're in the field, not after they've moved out of it, and then have to go back and change or fix something. Let's see how we do this.

[1:51] Here's my form input component, and that's being used for each of these inputs in this login form. You can see that it receives helper text and error text, but neither of them are required. Here is our input, and if we have helper text, we include it. If we have error text, we include it.

[2:14] The first thing we need to do is we need to give each of these an ID. We need to make this conditional, since we don't always have either of these values. Let's create a helper ID, and it's conditional on helper text.

[2:31] We have it. Then we want create an ID. We can use the name of the field for this in a `template literal`, and we just want to return an empty string.

```
// src/primitives/FormInput.js
const helperId = helperText ? `${name}-helper` :
''
```

Then we give the containing element this ID.

```
// src/primitives/FormInput.js
{
  helperText && (
    <small id={helperId} className="form-text
text-muted helper-text">
      {helperText}
    </small>
  )
}
```

We'll do something similar for the error text, if we have error text.

[2:52] We're going to go one further than that and verify of the field is actually invalid. We don't want to reference the error message if the field doesn't have an error. Due to the way this conditional is set, if we have error text, we're going to include this div in the DOM.

[3:07] Based on our styling, we're going to visually hide it, so it will exist, but we don't want to associate our input with it if we don't actually have an error, if the field is not actually invalid. Because we have this `isValid` prop, we can reference that.

[3:23] If we have the error text, and if it's not valid, then we will set the ID. We'll use another template literal with the name of the field and just append that with error. Otherwise, same thing, empty string.

```
// src/primitives/FormInput.js
const errorId = errorText && !isValid ? `${id}-
error` : ''
```

Now that each of these elements has an ID, we can reference them.

```
// src/primitives/FormInput.js
{
  errorText && (
    <div id={errorId} className="invalid-
feedback">
      {errorText}
    </div>
  )
}
```

[3:43] Again, we're looking to add a description association to the input, and we can do that with the `aria-describedby`. Similar to how `aria-labeledby` works, it takes a list of IDs delimited by a space. We'll add our helper ID and our error ID.

```
// src/primitives/FormInput.js
<input
  id={id}
  type={type}
  name={name}
  className={inputClasses}
  onChange={onChange}
  aria-describedby={`${helperId} ${errorId}`}
/>
```

[4:03] Now, let's hear how this is working with the screen reader. Now, let's hear what is read when we get to the password field.

Announcer: [4:09] Password, secure edit text. Passwords are case-sensitive.

Instructor: [4:13] Now, it's telling us. It's reading this helper text, "Passwords are case-sensitive," as soon as we focus on the field, instead of waiting until after we move out of it. Let's go ahead and trigger the form validation and find out how our field errors are read.

Announcer: [4:28] Username, edit text. Please provide a username.

Instructor: [4:32] Now, we're hearing that we've missed providing a username while we in the username field. Let's hear what password sounds like.

Announcer: [4:38] Password, secure edit text. Passwords are case-sensitive. Please provide a password.

Instructor: [4:44] We're hearing both the helper text and the error text when we get to the password field, which is exactly what we want. If we supply that value and try to submit the form again, now that error text is no longer being displayed.

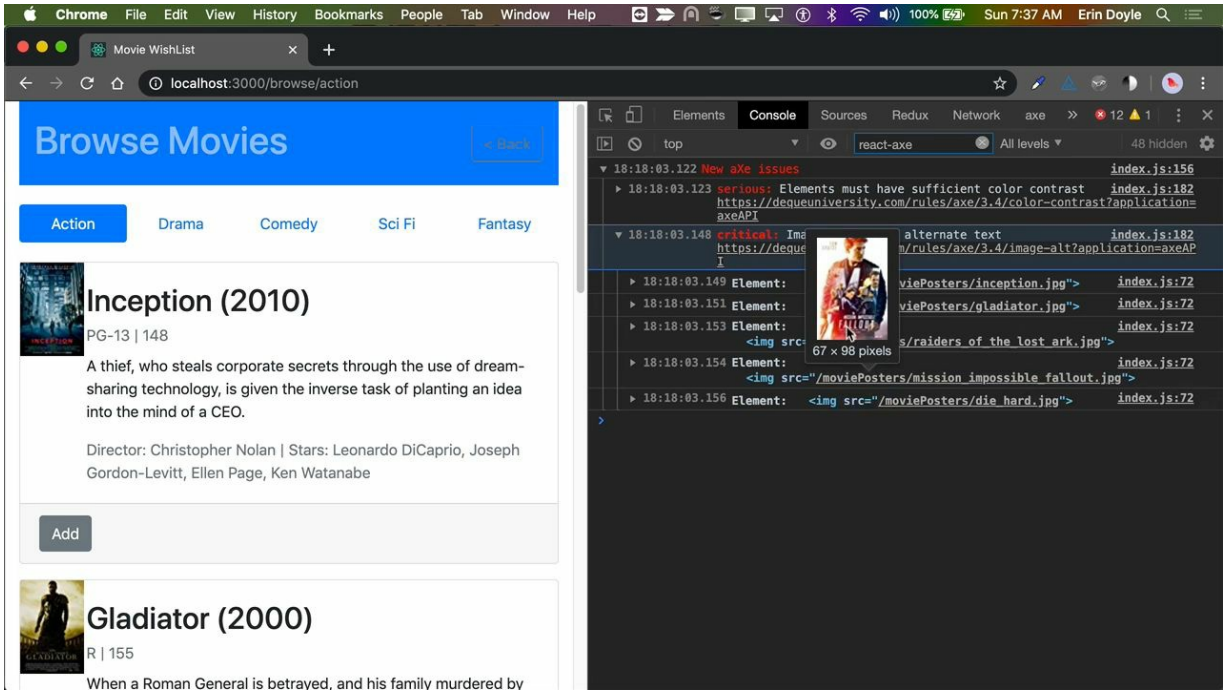
[4:58] Let's make sure that we also don't hear it when we focus on the password field.

Announcer: [5:02] Password, secure edit text. Passwords are case-sensitive. You are currently on a text field.

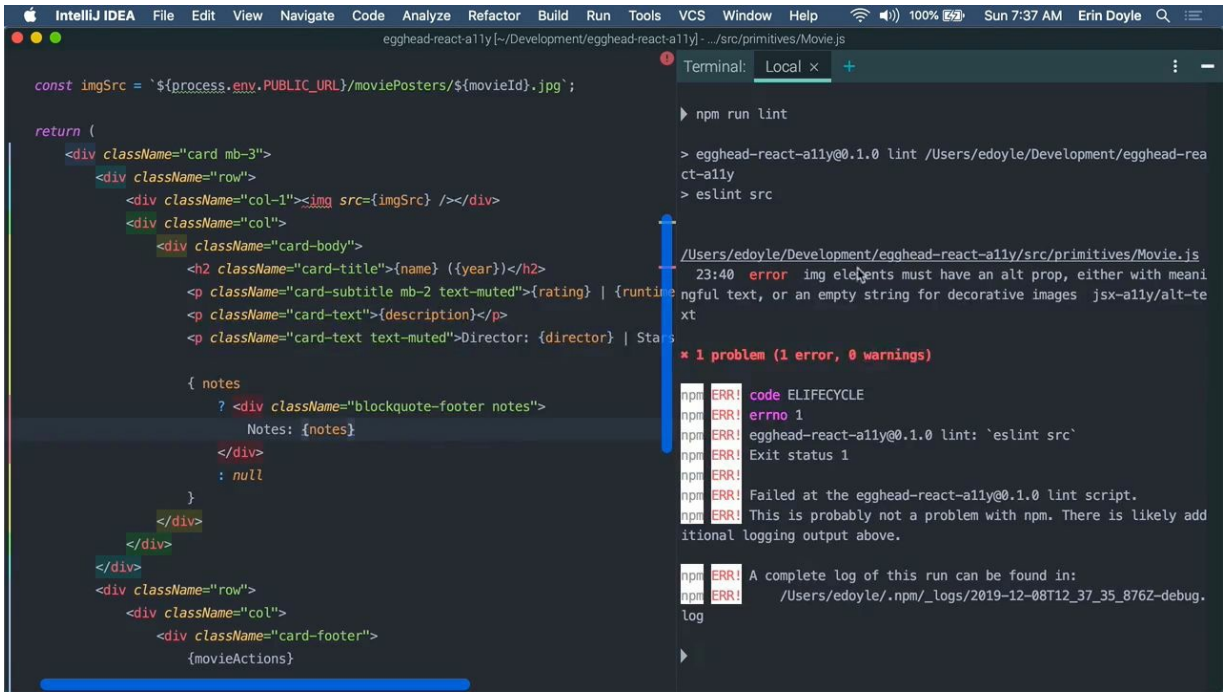
Instructor: [5:08] Again, we are only hearing the helper text. We're no longer hearing that error message, because it's not displaying.

Test for Image Alternative Text Accessibility Issues

Instructor: [00:00] Here we have a web page with a number of images on it. I'm running `react-axe`. Over here, we can see being logged the console. `Images must have alternate text`. It lists each of these images. All of these are missing alternative text.



[00:21] Additionally, we can run `eslint`, which has installed the `eslint-plugin-js-a11y`. If we run that, this will also report that we are missing an `alt` prop on our `image` elements.



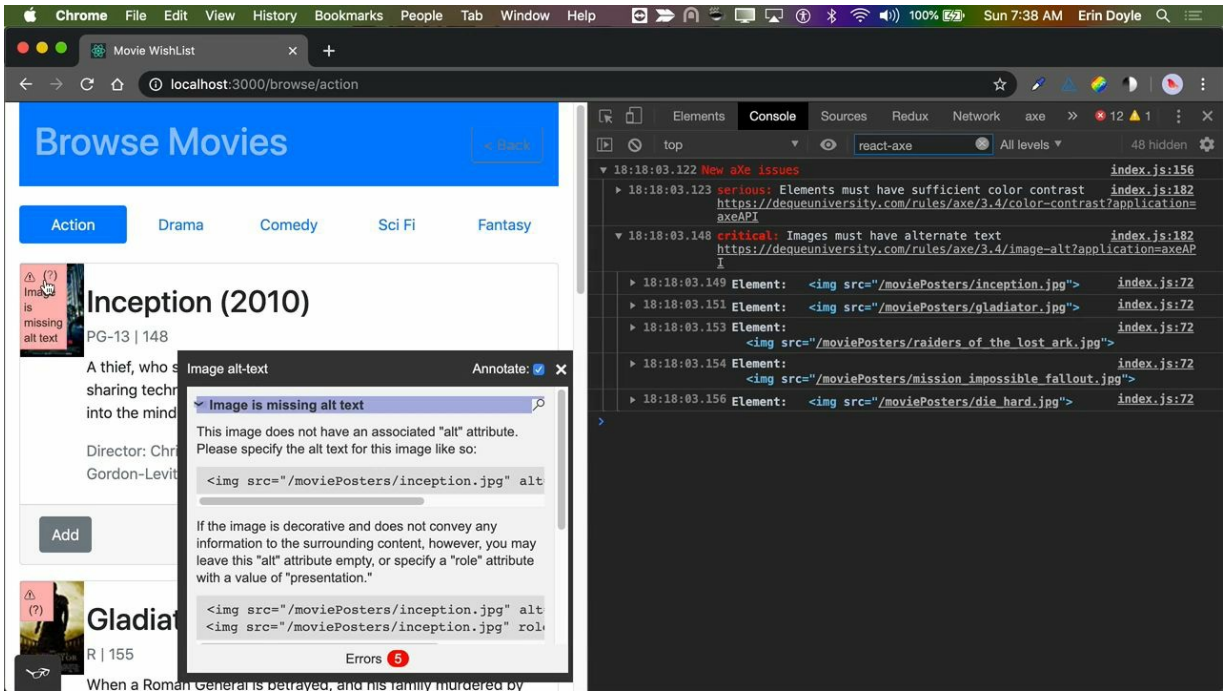
Over here is the responsible code.

Movie.js

```
 <div className="card mb-3">
    <div className="row">
        <div className="col-1"><img src={imgSrc}
/></div>
```

For IDEs that integrate with `eslint`, you'll actually see in the code the eslint finding about missing `alt` text.

[00:49] Going back to the browser, we can also use `tota11y` to annotate any missing `alt` text on images. As we can see, each of these images is annotated.



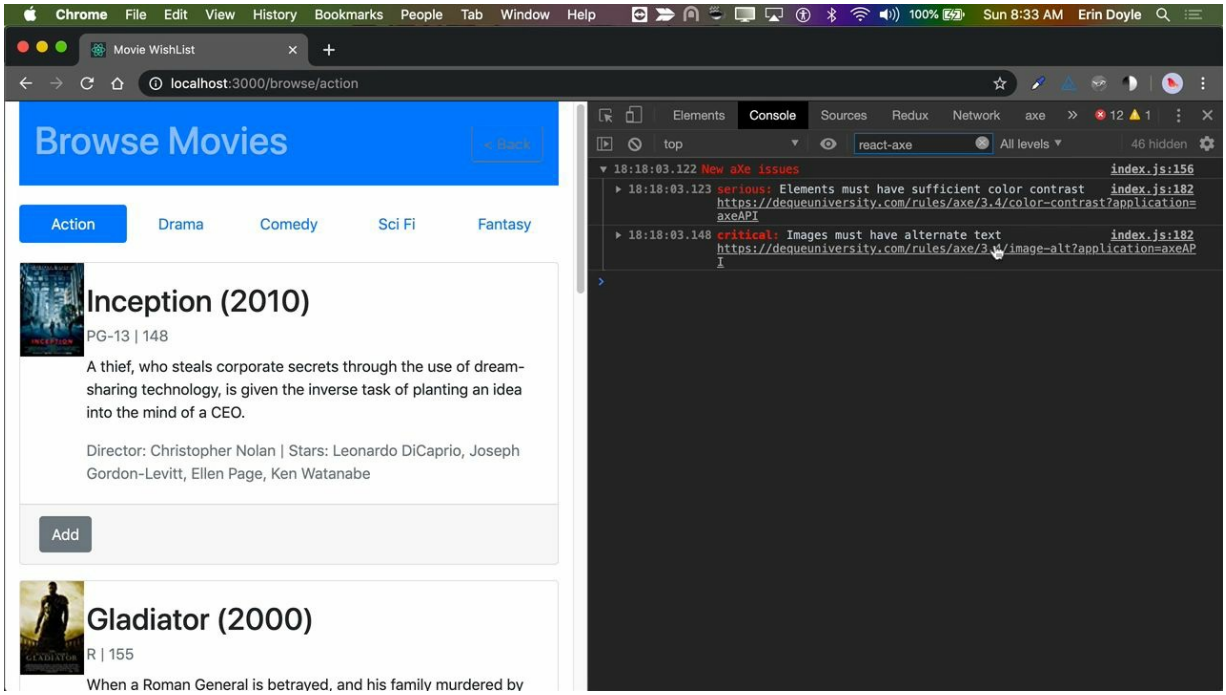
[01:03] Finally, if we go to Safari and run VoiceOver, we can hear what is read for each of the images.

- Automated Voice: [01:11] Diener. current visit inception JPG image. Gladiator JPG image. underscore under the underscore lost underscore arc jpg image.

Instructor: [01:24] What it's doing is, it's reading the filename. Because we don't supply the `alt` attribute providing alternative text, the screen reader will fall back to reading the filename, which is really not helpful. This is why we need to make sure we're providing alternative text for our images.

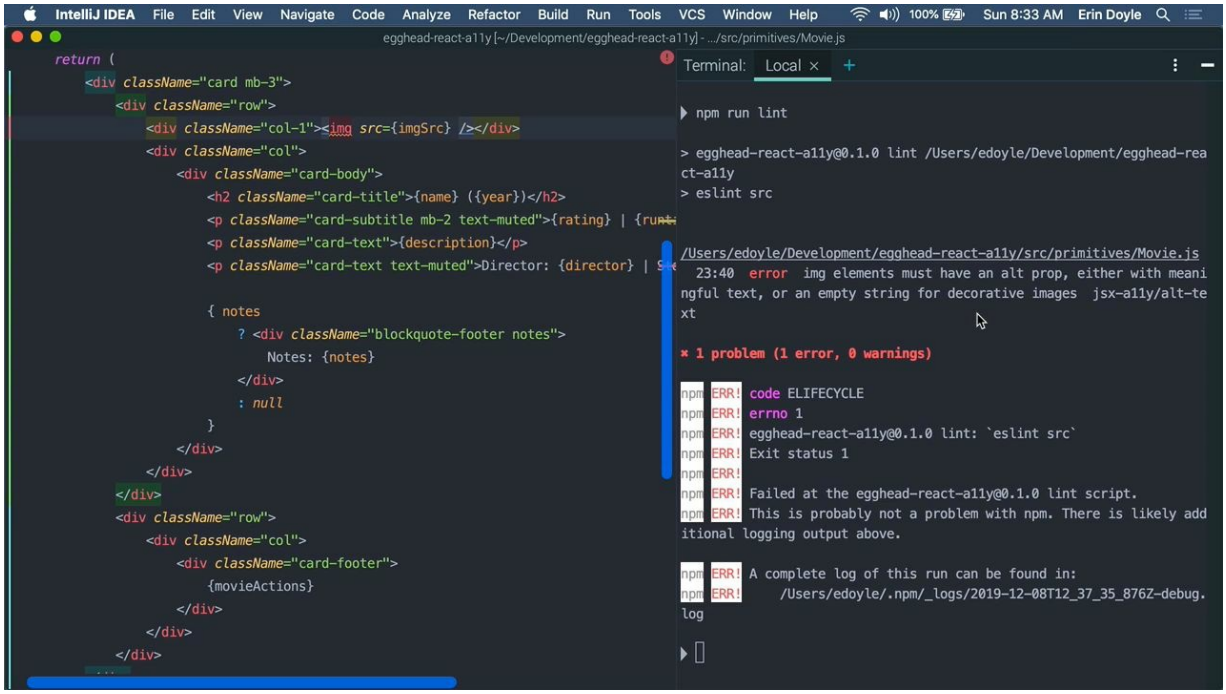
Define Images with Appropriate Text Alternatives

Instructor: [0:00] Here I have a web page with some images that, as you can over here from `react-axe`, are missing alternative text.



Here's the responsible code.

As you can see over here, I've run `eslint`. It's showing me the same finding, with `image` elements missing an `alt` prop.



[0:20] This is the `image` that's missing the `alt` prop.

Movie.js

```
<div className="card mb-3">
    <div className="row">
        <div className="col-1"><img src={imgSrc}
/></div>
```

It corresponds to each of these movie poster images in my application. If we look at this component, we've got some information up here about the movie. This is my `Movie` component.

```
const Movie = ({ movieId, movie, movieActions })
=> {
    const {
        name,
        year,
        description,
        director,
        stars,
        rating,
        runtime,
        genre,
        notes
    } = movie;
```

[0:39] If we want to provide some meaningful alternative text for the `image`, which is of the movie poster, we should be able to put something together for that. If we provide an `alt` attribute, we can use the `name` of the movie in a template literal and combine that with movie poster.

```
 <div className="card mb-3">
    <div className="row">
        <div className="col-1"><img src={imgSrc}
alt="{`${name} Movie Poster`}" /></div>
```

[1:02] We run `eslint` again. That finding is now gone. If we look at the browser, `react-axe` is no longer reporting that finding as well. If we go to Safari and run VoiceOver and listen to what that sounds like.

- Screen Reader: [1:24] "Inception" movie poster image. "Gladiator" movie poster image. "Raiders of the Lost Ark" movie poster image.

Instructor: [1:32] Now the name of the movie is read as the movie poster image. That's definitely more useful than what we had before.

[1:40] If we consider the functionality that these movie poster images is conveying to the user, they really don't provide any additional information that we're not already conveying through the title and all of the information and description of the movie that's already being read by the screen reader.

[1:58] Really, these are just decorative images. They're not functional in any way. What we could do instead of providing the name of the movie, movie poster as the alternative text, we could just provide an empty string. What this does is actually removes it from the assistive technology's tree. It won't be announced by the screen reader.

```
 <div className="card mb-3">
    <div className="row">
        <div className="col-1"><img src={imgSrc}
alt="" /></div>
```

[2:20] Let's check real quick and make sure eslint still approves. It does. `react-axe` is still not reporting any findings. Let's see what that sounds like when read by the screen reader.

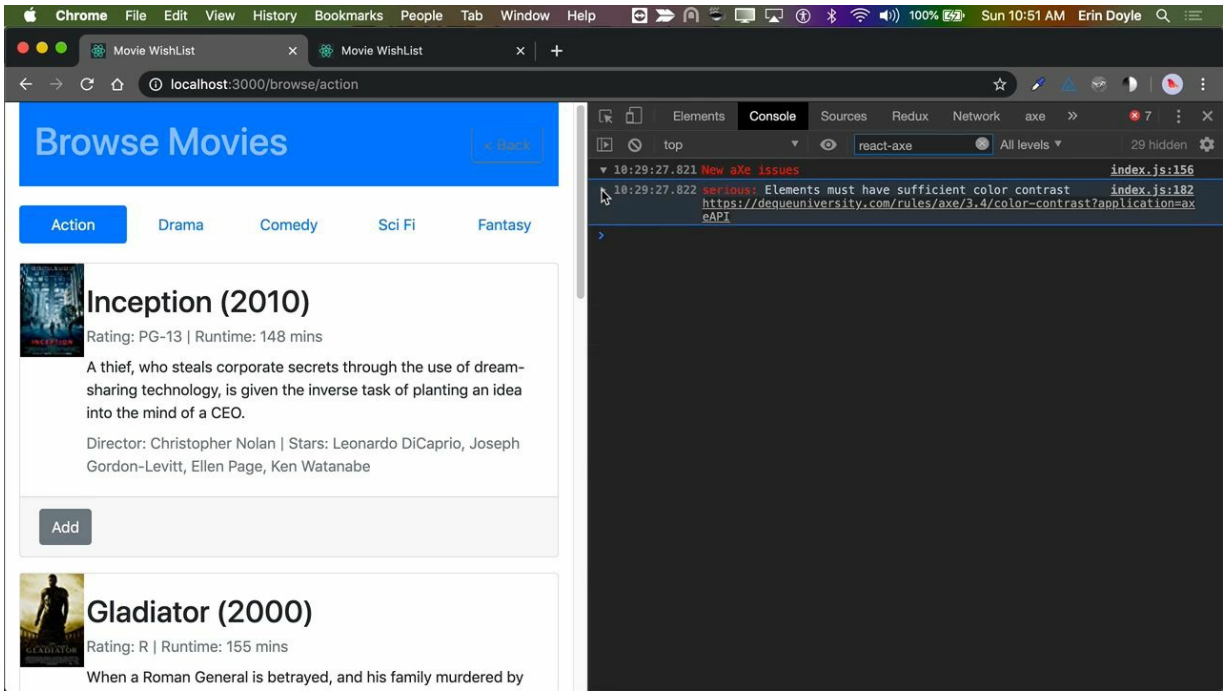
- Screen Reader: [2:43] Navigate. Heading. Add Inception. Heading level. Add Gladiator. Heading Level. Add Raiders. Heading Level. Add Mission.

Instructor: [2:52] The screen reader actually skips focusing on each `image` and does not announce its presence. When you're providing alternative text on your images, you'll have to consider what function does this `image` provide, does it need to be announced to the screen reader, or is it just decorative.

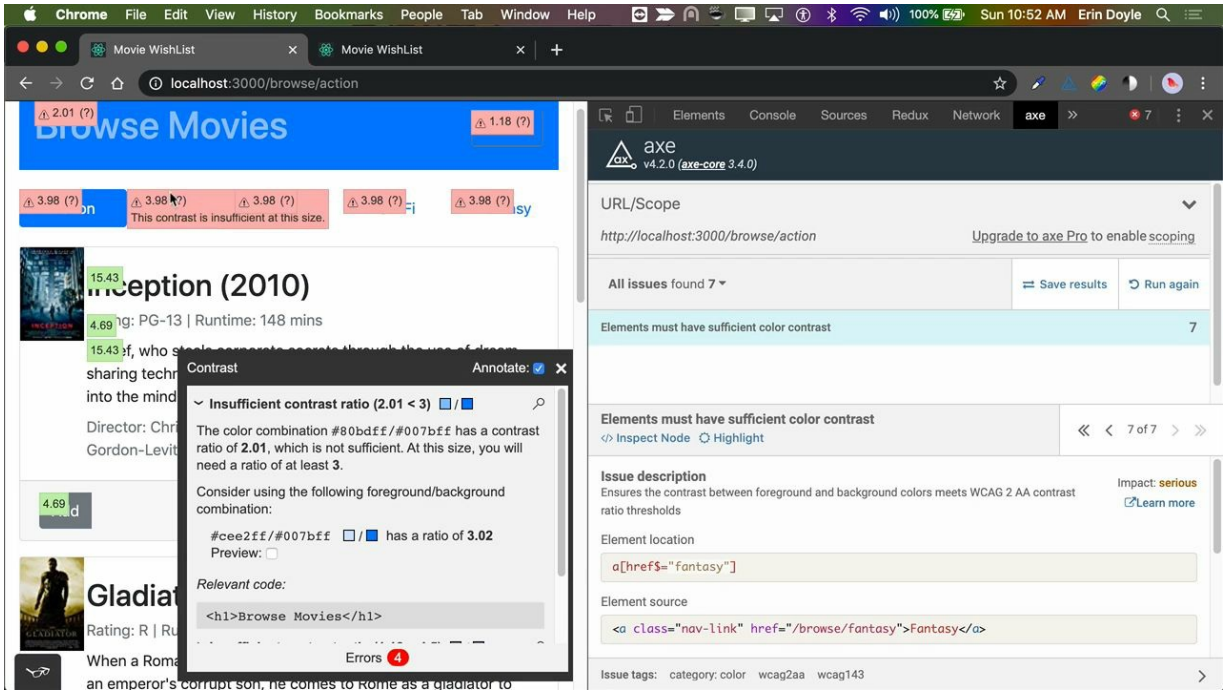
[3:08] It's only useful for those who are sighted and therefore would just be noisy or distracting if read by a screen reader.

Test for Insufficient Color Contrast Accessibility Issues

Instructor: [00:00] Here is a web application that has a number of color contrast issues in it. I'm running `react-axe`, and we can see we have this finding -- `Elements must have sufficient color contrast`.



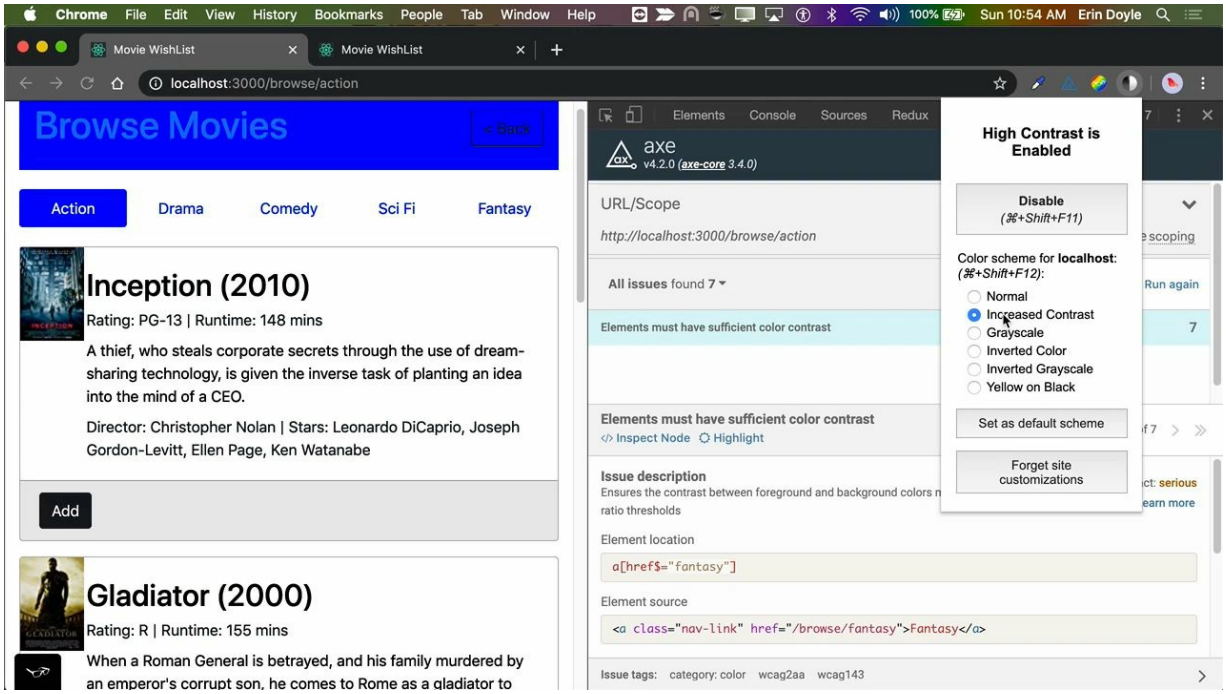
[00:12] If I expand that, it lists out each element that does not have a sufficient color contrast ratio. If I use the `axe` browser extension, I've got the same finding, and I can highlight each element. If we use `tota11y`, we can annotate the contrast ratios, and all of these in red show an insufficient contrast ratio.



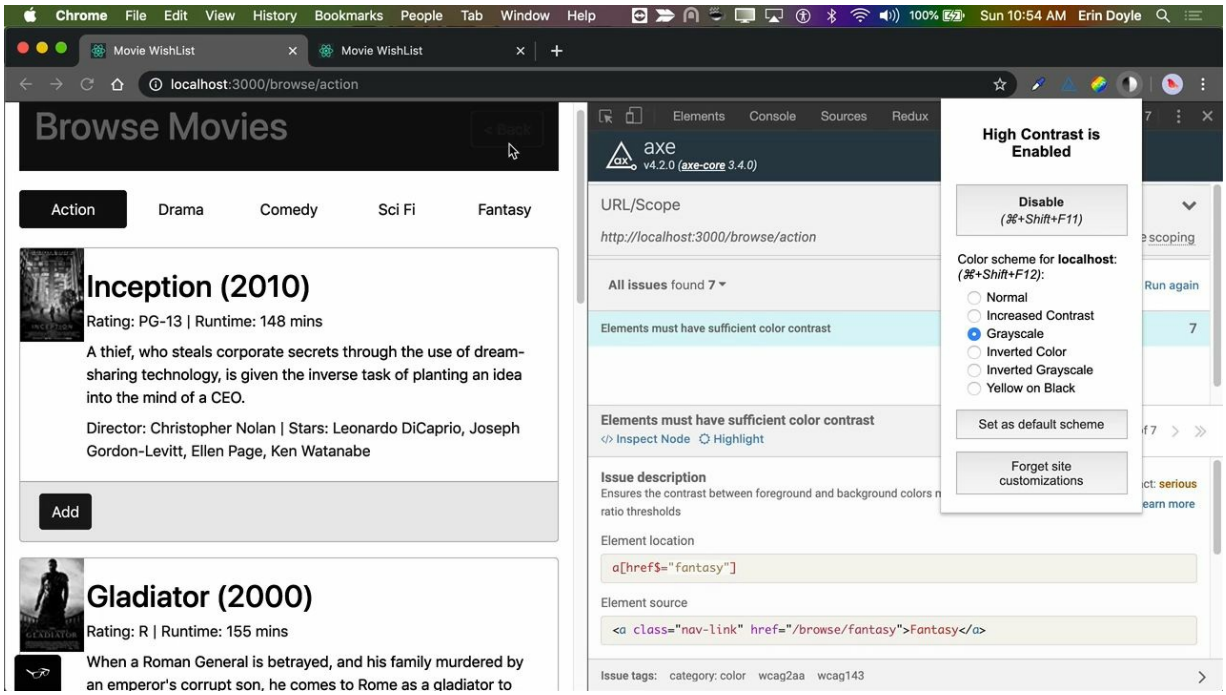
[00:41] Another cool feature about `tota11y` is that it lists the hex values of the foreground and background colors for each element that has an insufficient ratio. It provides a suggestion of the foreground and background color that would create a sufficient ratio. You can actually preview it by checking this box.

[01:08] In addition to using these various auditing tools, it's also really important to use the various experience tools, the tools that our users are using in order to facilitate their accessibility. A lot of users with visual impairments may be using high contrast tools.

[01:24] Here we have a high contrast browser extension installed. I'm going to enable that and check my web page with each of the modes to see how it displays.

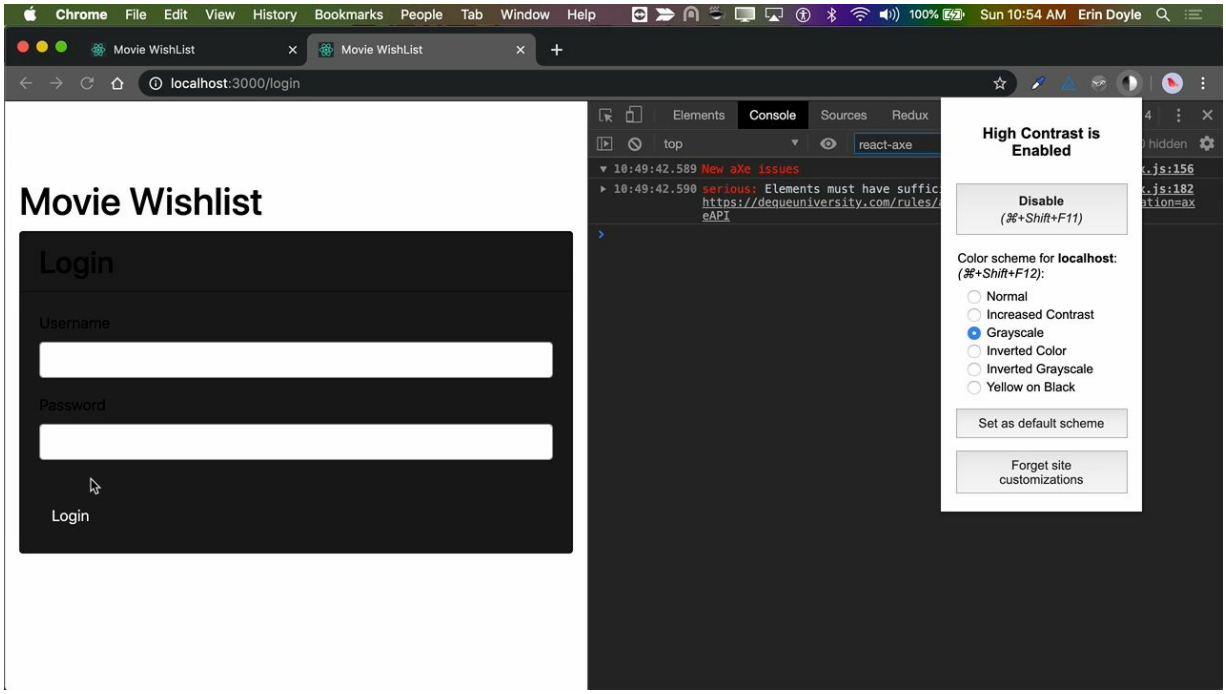


On this mode, for instance, this button becomes virtually invisible.

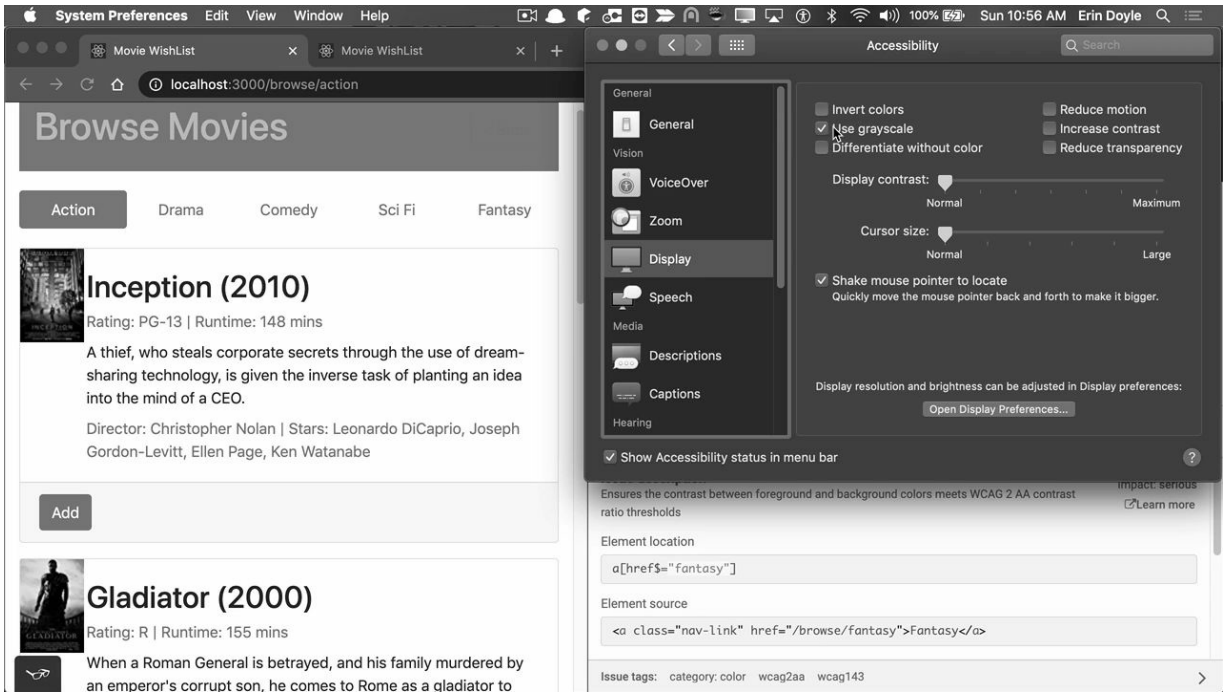


Here, it's slightly more visible, but almost impossible to read. The same with this one.

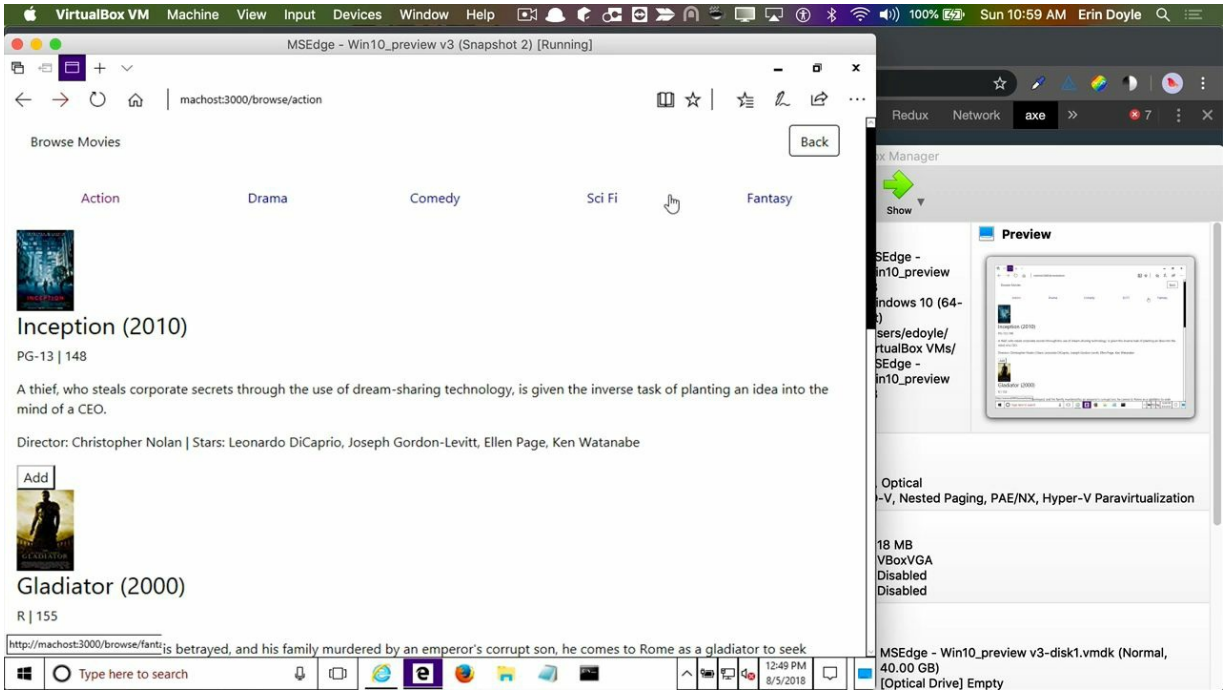
[01:53] If we look at another page in our web application, we find here that text becomes almost impossible to read.



If I go to the accessibility display preferences here on my Mac, and modify each of these, I can also see how my pages display. This one absolutely makes this button invisible.



[02:51] If we look at this on an IBM running Windows, in Windows high contrast mode, everything looks pretty decent on this page. However, we can see that it is not clear which of these tabs is selected. It's pretty much invisible.

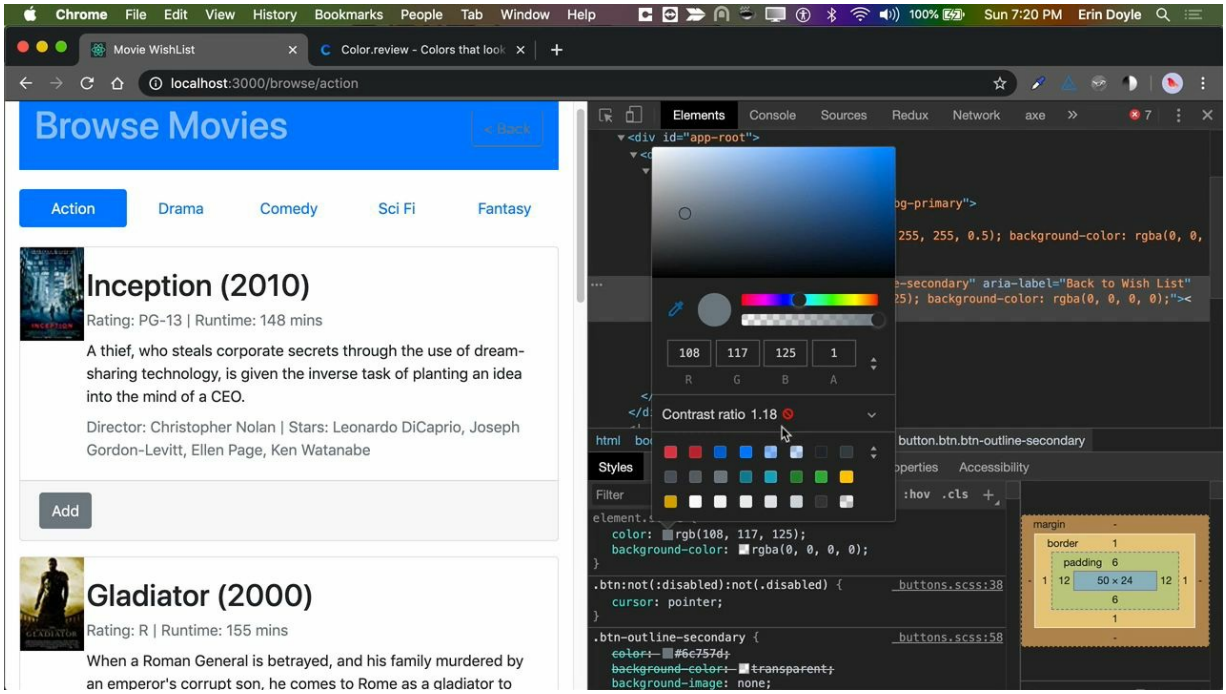


[03:16] By testing with each of these different high contrast tools, we get a full picture of all of the different modes and operating system settings that our users might possibly be using out there. While one might look OK, another one might make something completely hard or impossible to see.

Use Sufficient Color Contrast in Web Page Design

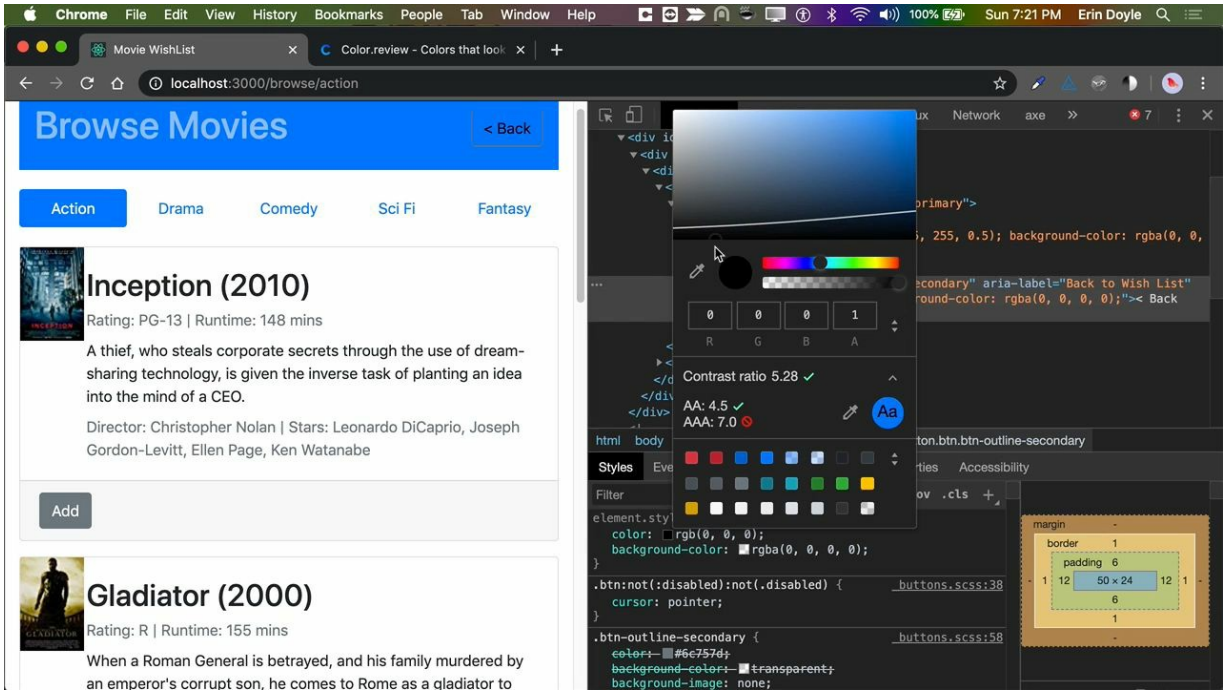
Instructor: [00:00] Here's my web page with known color-contrast issues. We've got some tools that can help assign sufficient color-contrast ratios. First, in Chrome DevTools we can inspect an element that has insufficient color contrast. Here is my `button`.

[00:18] If I look at the color style and click on that, right here Chrome tells us that the contrast ratio's insufficient with this icon.



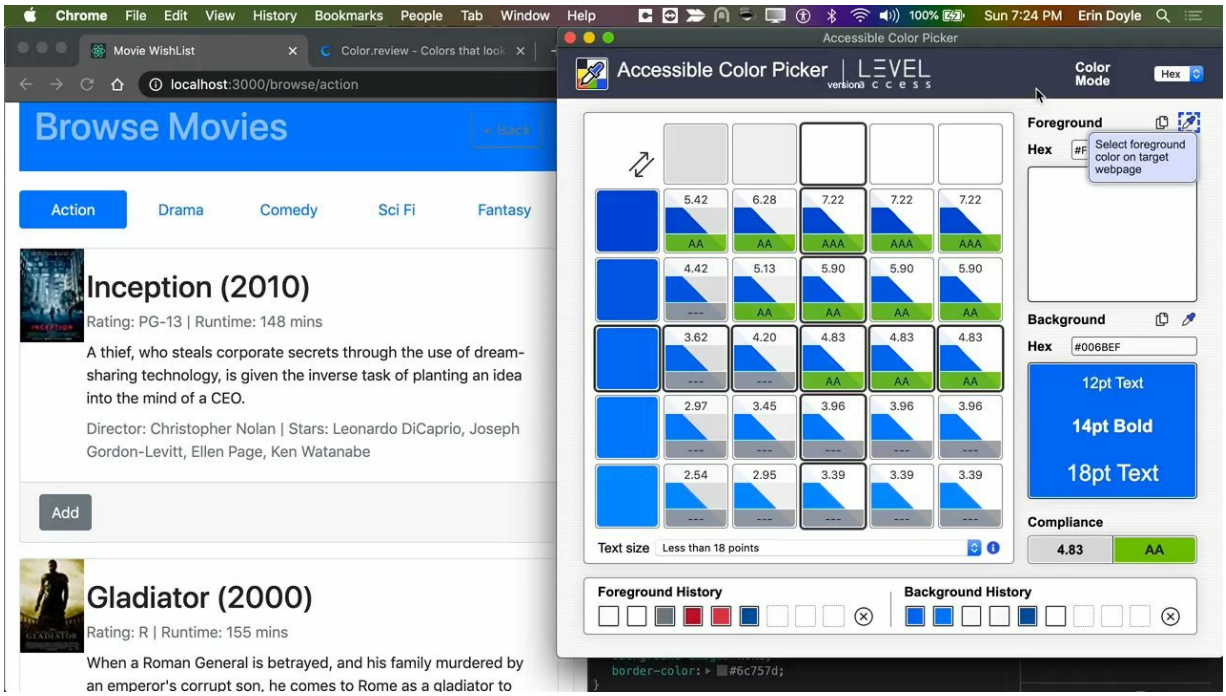
It tells us what the ratio is. We need that to be 4.5. If we expand this section, it lists the acceptable ratios for the AA and AAA levels of the WCAG standard.

[00:43] Here's a preview of the foreground against the background color. If we look at the color swatch here, there's this white line. That represents the WCAG AA level of acceptance. If the color is below this line, it should meet AA standards. If the color was closer to AAA, we would see an additional line for AAA.



[01:07] As we could see, if we move our color under the line, there's really no way we can possibly meet AAA. We can only meet AA based on the combination of foreground and background colors. If we can't meet the ratio we're looking to meet, we may have to change the background color as well as the foreground color to find a ratio that's sufficient.

[01:32] Another tool that's helpful is the Level Access browser extension, Accessible Color Picker. Again, this is available for Chrome.

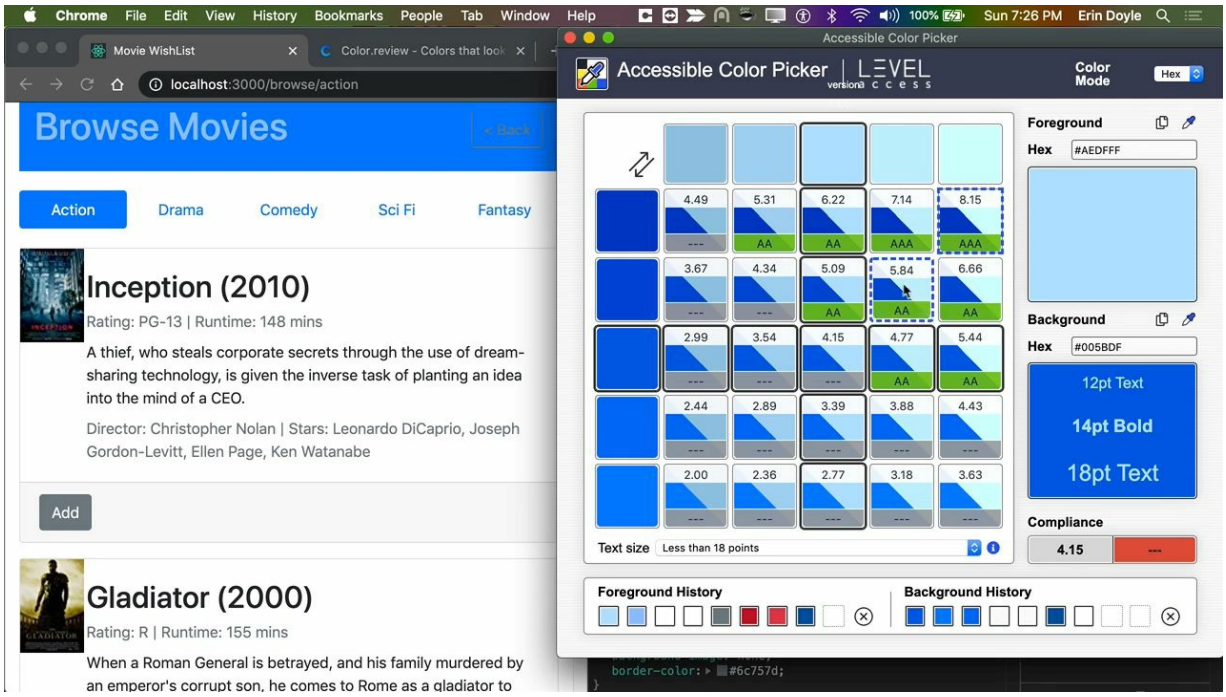


You can select your foreground color and your background color. It shows the contrast ratio here and that it is not meeting WCAG AA or AAA compliance levels.

[01:57] It gives you a preview here of the foreground against the background colors as well as our hex values. In this chart we can see right in the middle is our current foreground-and-background color combination. As we go right, the foreground color gets lighter.

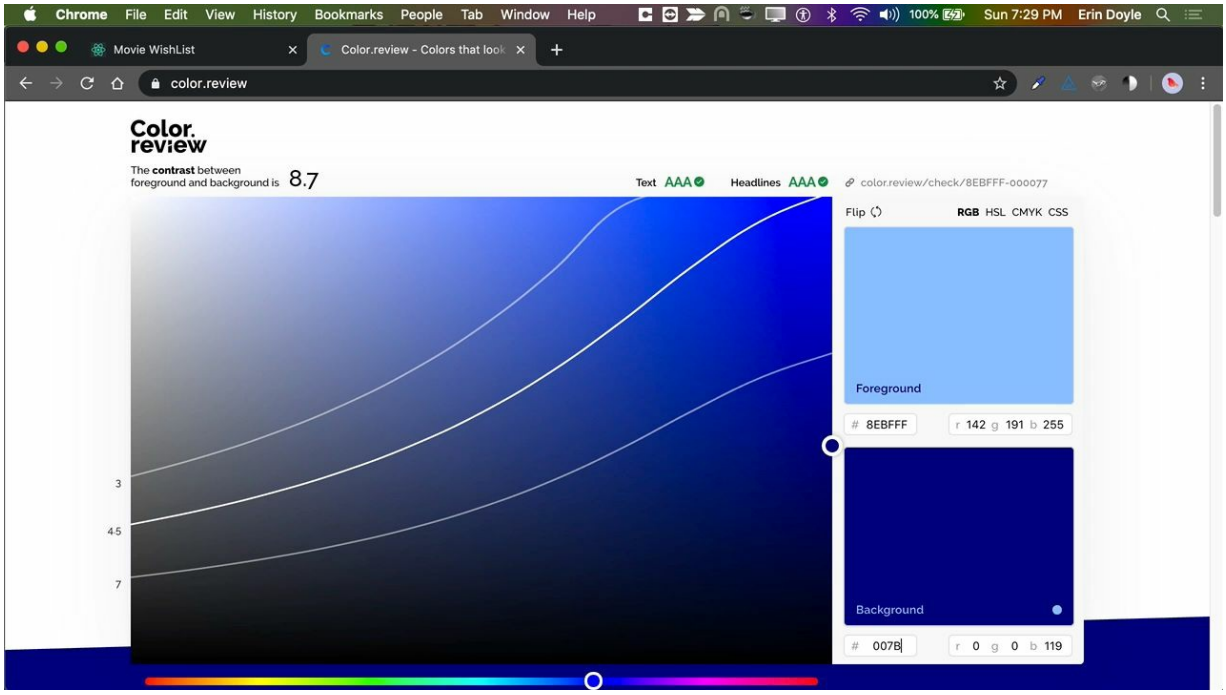
[02:11] As we go left, the foreground color gets darker. As we go up, the background color gets darker. As we go down, the background color gets lighter. You have all of these examples of ways you can slightly modify each of the colors and how that impacts the contrast ratio. We can see that, based on this combination, none of these modifications get us to the acceptable contrast ratio.

[02:34] By selecting that color, that then puts that combination here in the middle. Now we can see suggestions that meet AA and AAA standards.



This is a great tool for helping us find what alterations we can make to those colors to get us to the acceptable contrast ratio. Finally, here's another site - (https://color.review (https://color.review)) that's really

helpful for this.

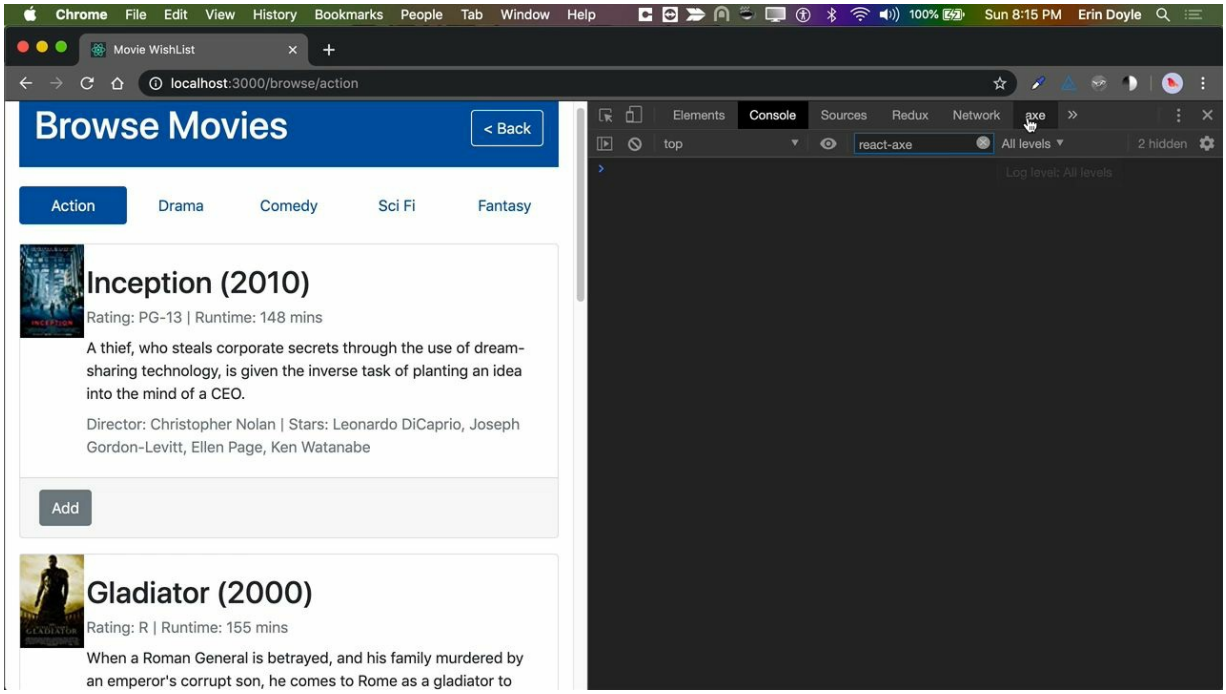


[02:58] If we enter the hex values for our foreground and background colors, we can see them previewed here. Similar to what we saw on Chrome DevTools when we inspected the element, we've got these three lines. We can pick either the foreground or background color to change.

[03:19] If we can get this foreground color down to the first line, it passes AA standards. If we can get it past the third line, it now passes AAA standards as well. Again, a great tool to help us find the color combination that will allow us to pass the WCAG standards.

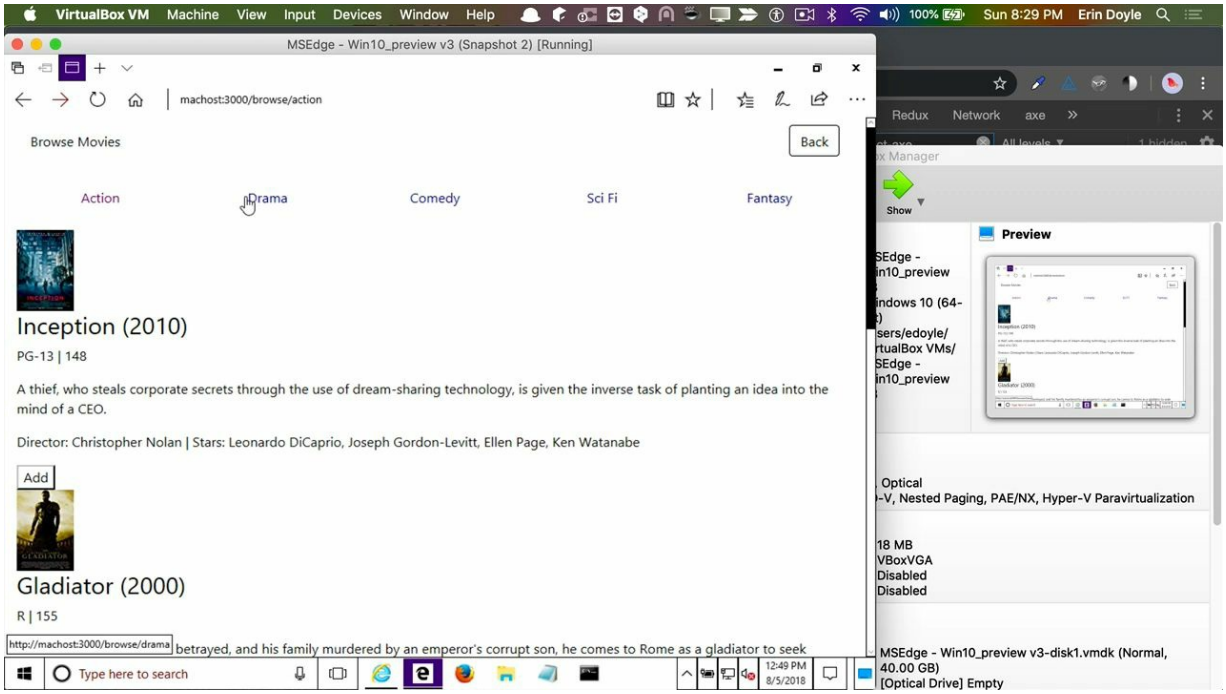
Use More than Color Alone to Convey Information in a Web Page

Instructor: [00:01] Here's a Web page that previously had color contrast issues, but they've all been resolved. We can see our console is clear of any findings from `react-axe`.



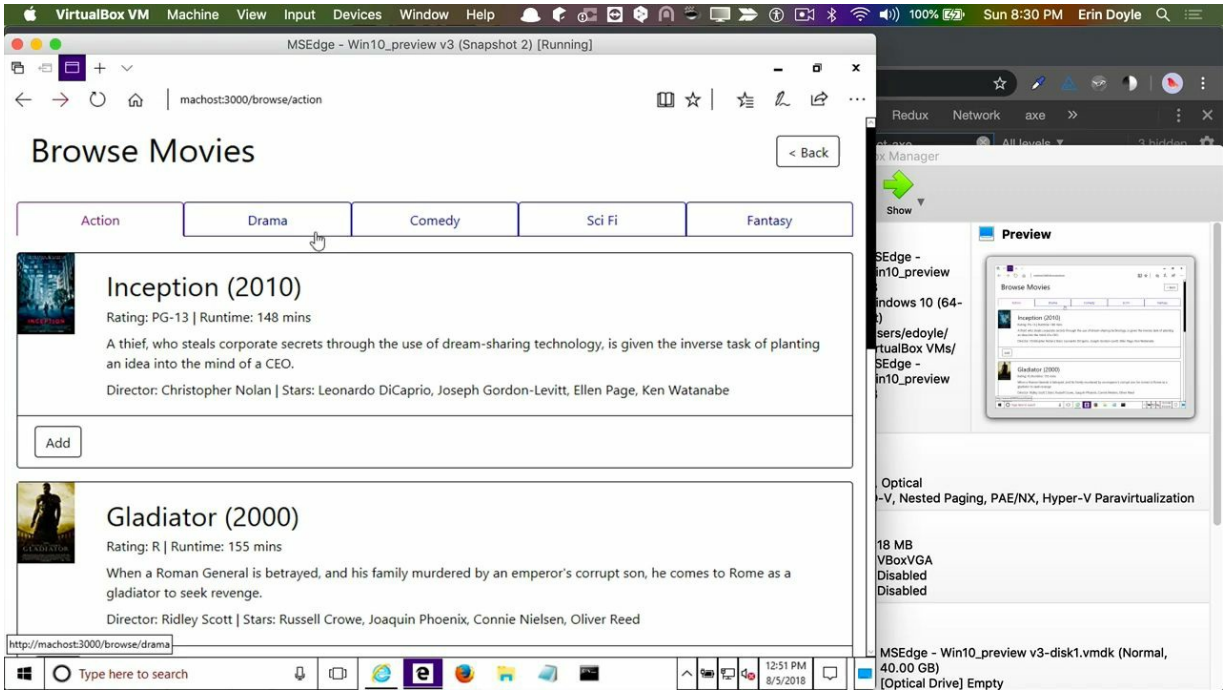
[00:12] If we run axe again, we now have no issues with color contrast. Everything is crisp and clear. If we use our Chrome High Contrast browser extension and check each of the modes, everything is still easy to read and see in all of the modes. Notice the tabs when we select a tab, we can tell which one is selected.

[00:40] Let's see what this looks like in Windows high contrast mode. We can see here that we still can't tell which tab is selected. I can click on each of these tabs and nothing changes visually to indicate which one has been selected, so we still need to fix that.

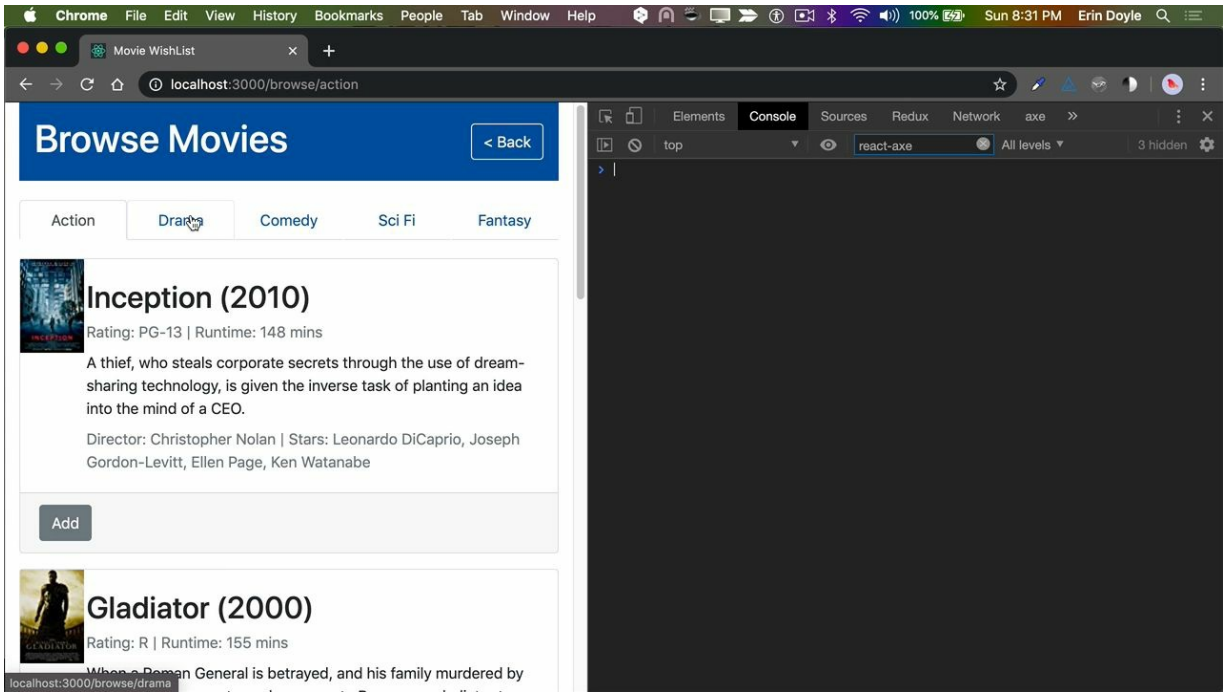


[01:09] The problem here is that our design is trying to use color alone to convey meaning. With certain high-contrast tools, like the one we're looking at right now, color is removed, or for certain users who have issues with color and they can't see those differences, we need to alter our design to convey the meaning without relying on color alone, so let's go fix that.

[01:34] Now we've changed our design so that instead of using a pill design for each tab, we're using a tab design. Now you can clearly see which tab is selected and which is not.



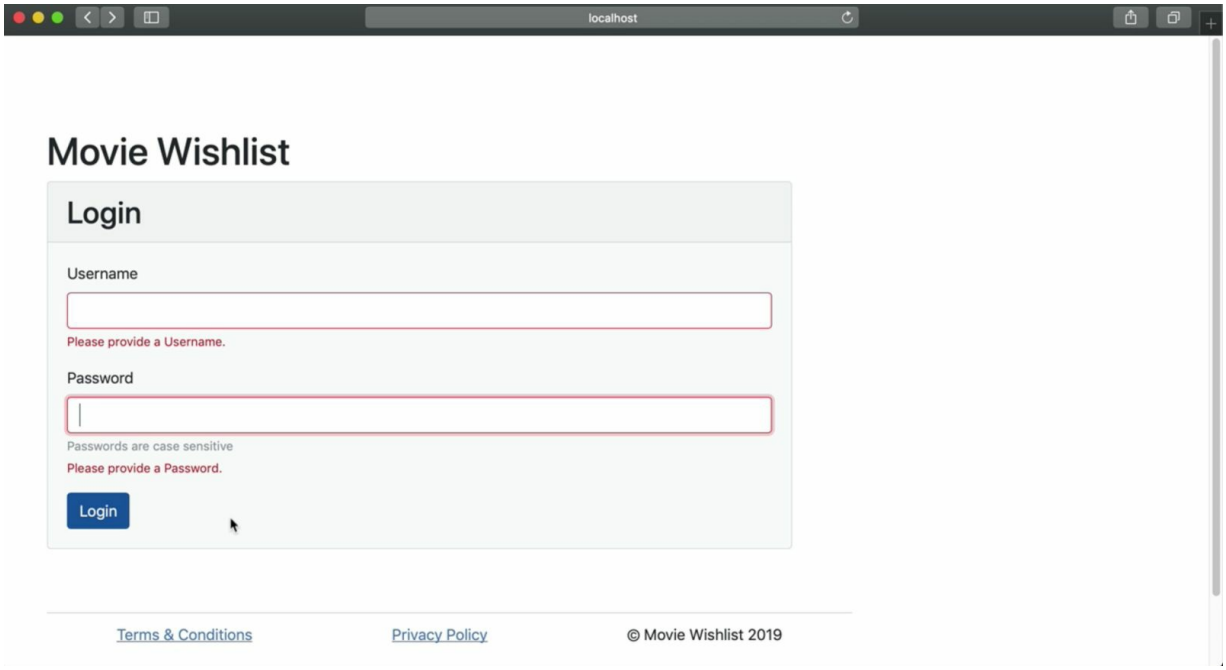
[01:48] If we go back to Chrome on the Mac, we can see that this new design works just fine here as well. Everything is very clear. You can tell the difference between the selected tab and the hover state.



[02:04] If we use our High Contrast browser extension and check each mode, each of these works just as well. This is a solution that works in all high-contrast modes on all operating systems and browsers.

Define a Live Region to Ensure Dynamic Changes are Announced by Assistive Technologies

Instructor: [0:00] Here, I have a form that has an issue concerning accessibility. This form has validation. Each of these fields is required. If we try to submit the form without filling in either of these fields, we'll get an error message letting us know that we need to provide a value for each field.

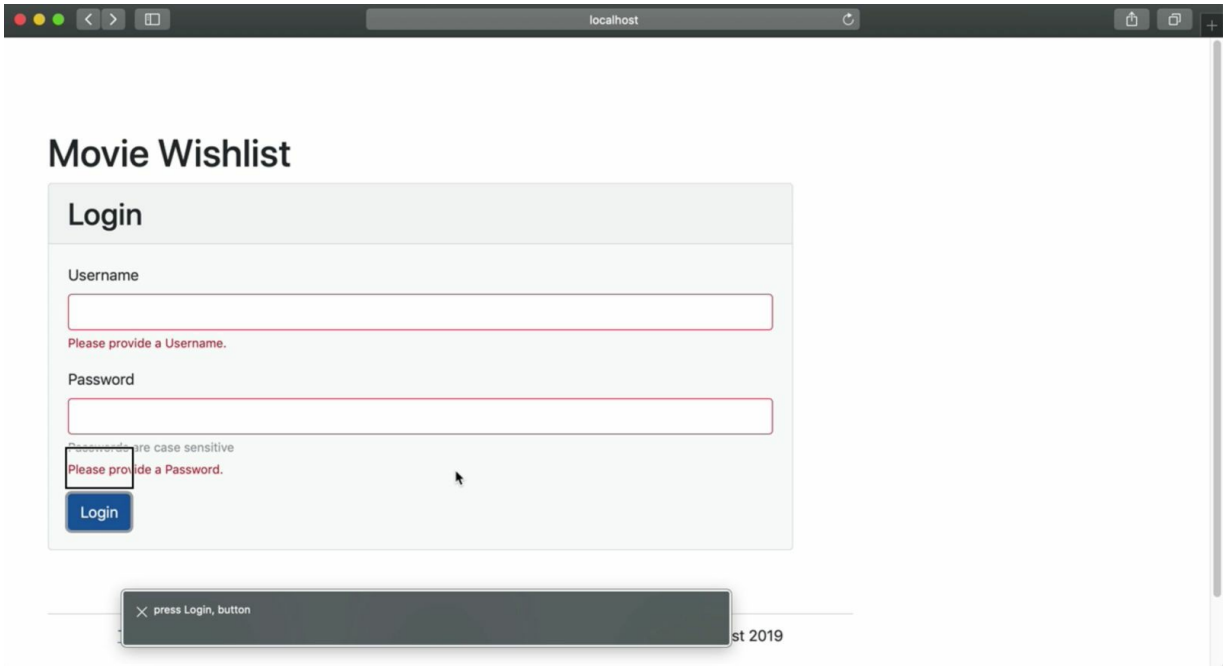


[0:16] The problem with this is that these error messages are not read to assistive technology users, unless they're actually focused on the field at the time, but our form validation isn't triggered until we click the login button.

[0:29] At that point in time, we're focused on the button, so the user's not going to know to go back to each field to hear the error message. They're not going to know that anything was wrong, and they're not going to know why the form isn't submitting, and they're not moving to the next page.

[0:43] Let's demonstrate that. I'm in Safari, and I'm going to use the VoiceOver screen reader. If I move all the way through my form without filling in any of the fields, and I press the login button...

Announcer: [0:53] Press login button.



Instructor: [0:55] nothing was read. We weren't notified in any way that there were any problems with either of our fields. Nothing is telling us that we're still on the login form or there's anything more we need to do.

[1:06] There's a way we can fix this. Let's take a look.

Announcer: [1:08] VoiceOver off.

Instructor: [1:10] Here is our `FormInput` component. Here's our input, and here is our `errorText`. We can add an attribute to this containing element that has the `errorText` in it that notifies assistive technologies that, if any changes are made to the content of this element, they should be announce, even without the user having to be focused on this element.

FormInput.js

```
const FormInput = ({ ... }) => {
  ...
  return (
    <div className="form-group">
      <label htmlFor={id}>{label}</label>
      <input
        id={id}
        type={type}
        name={name}
        className={inputClasses}
        onChange={onChange}
        aria-required={isRequired}
        aria-invalid={!isValid}
        aria-describedby={`${helperId}
${errorId}`}
      />
      { helperText &&
        <small id={helperId} className="form-
text text-muted helper-text">
          {helperText}
        </small>
      }
      { errorText &&
        <div
          id={errorId}
          className="invalid-feedback"
        >
          {errorText}
        </div>
      }
    </div>
  );
};
```

[1:36] That's called `aria-live`. `aria-live` has three possible values. It could be off, it could be `polite`, or it can be `assertive`. The difference between `polite` and `assertive` is that `polite` will wait until the screen reader is done reading whatever it currently is reading before it will go ahead and announce the change to this element.

[1:56] aria `assertive` will interrupt whatever the screen reader is currently reading to announce the change. Additionally, with `aria-live`, there are two other attributes that are useful. There's `aria-atomic`, and its values are either `true` or `false`.

```
<div
  id={errorId}
  className="invalid-feedback"
  aria-live="asertive"
  aria-atomic="true"
>
  {errorText}
</div>
```

[2:10] It defaults to true, and this defines whether the screen reader should always present this element as a whole anytime there are any changes made to it, even if only part of the content's changed. True means it will always present the entire contents of the element marked as `aria-live`.

[2:27] Finally, there's `aria-relevant`. This can take a list of values that include `additions`, `removals`, and `text`, or all. `additions` means that, if any nodes are added to the content of the `aria-live` element. Right now, we just have text.

```
<div
  id={errorId}
  className="invalid-feedback"
  aria-live="asertive"
  aria-atomic="true"
  aria-relevant="additions text removals"
>
  {errorText}
</div>
```

[2:46] Imagine we had some child elements in here within the div. If any of them were added, then by specific additions in `aria-relevant`, those additions would be announced. Similarly, with `removals`, if any elements are removed from the `aria-live` region, those would be announced.

[3:02] Finally, `text`. If any text contents of the `aria-live` element are changed, those are announced. By default, `aria-relevant` is always `additions` and `text`. These (`aria-atomic` and `aria-relevant`) are implicit if we have an element marked as an `aria-live` region, so we don't need to specify them if we're using the default values.

[3:22] Finally, one important rule of defining a live region is that the element needs to already be in the DOM before its content changes, and it needs to already be defined as a live region. That's in order for the assistive technology in the browser to be aware of it, know it exists, and be able to watch it for changes.

[3:42] If we dynamically add this entire div element, and or we add the attribute defining it as a live region after the initial render, then it's not always likely to work for all assistive technologies equally. We've already got this conditional checking for if we have `errorText`.

[3:59] The way we define that here in our form, we always pass in the `errorText`. Whether the field is valid or not, we're going to provide that value. It should always exist in the DOM, whether it's valid or not. I have some CSS that will hide this element visually if the form is valid, but we want to actually register changes to the content of the element.

[4:24] We don't want the `errorText` to exist in the element until we actually have an error, so that we can make sure our `aria-live` picks up that content change and reads it. Let's add a conditional here. Only if not valid should we return the `errorText`. Otherwise, empty string.

```
<div
  id={errorId}
  className="invalid-feedback"
  aria-live="asertive"
  aria-atomic="true"
  aria-relevant="additions text removals"
>
  {!isValid ? errorText : ''}
</div>
```

[4:42] Now we need to decide, should this value be `assertive` or `polite`? Let's try this out with assertive and hear how it works. If I go ahead and move down to the login button and submit the form without filling out any of these fields, let's hear what it says.

Announcer: [4:56] Please provide a password. You are currently on a button. To click this button, press control-option-space.

Instructor: [5:02] What happened is interesting. When I clicked the login button, it ran the form validation, and we presented our errors, "Please provide a password," was read immediately, and it interrupt the screen

reader from finishing reading about letting us know that we are currently on a button.

[5:16] That was good, but we never heard, "Please provide a username." What the screen reader was doing was it actually interrupting itself. Before it even had a chance to read, "Please provide a username," it interrupted itself to say, "Please provide a password."

[5:30] We only heard the last live region on the page. `assertive` seems to be a little too aggressive in this scenario. There may be cases where we have one error message. Maybe it's a page-level error or alert of some sort, and that might make sense to be `assertive`.

[5:48] Something like this, where we might have multiple at the same time, it seems like `assertive` is going to interrupt itself so much that we don't hear all of the messages. Let's hear what that sounds like if we change it to `polite`.

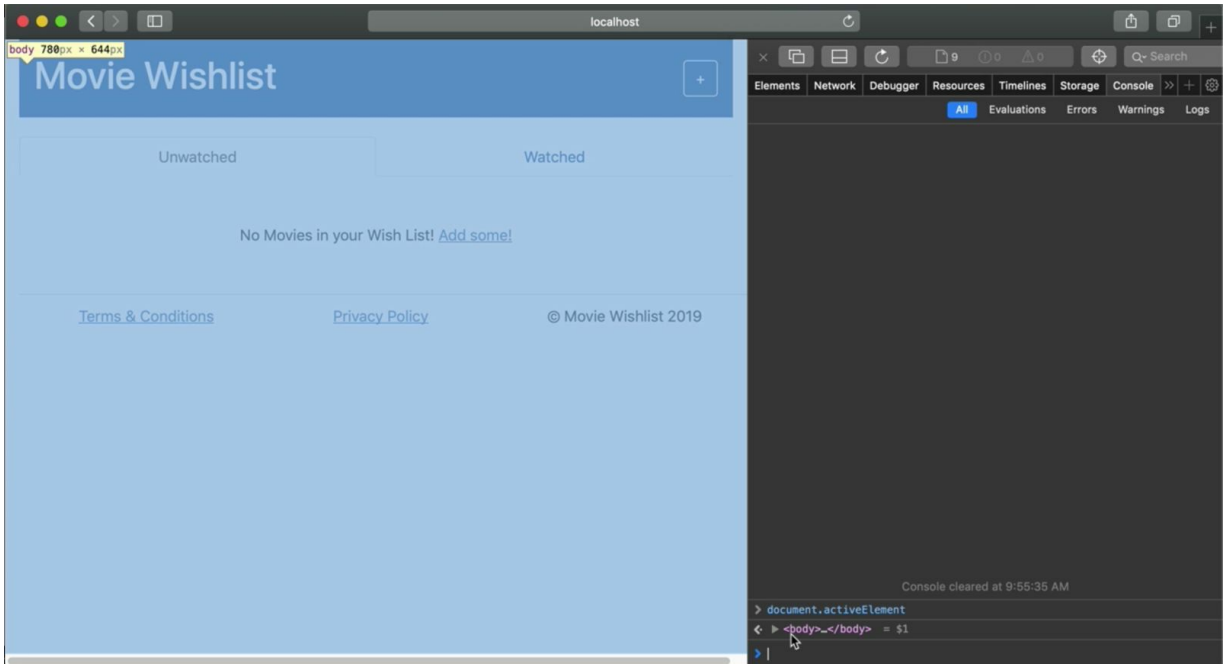
Announcer: [6:04] Press login button. Please provide a username. Please provide a password.

Instructor: [6:09] This time, it announced both changes, "Please provide a username, please provide a password." We heard both of those read, and that's exactly what we want. Now, the user can tell that they have some issues with the form that they need to go fix.

Appropriately Set the Focus on Each Page Load of a Web Application

Instructor: [00:00] Here, I have a web application where I'm not deliberately setting or managing the focus as the user navigates from one page to the next. Let me demonstrate. As a keyboard user, when I move to the next page, and that page loads, let's see where the focus is set.

[00:16] We can see right here that it goes to the body, because we are not explicitly telling it where to go. If I move to another page, it's the same thing.

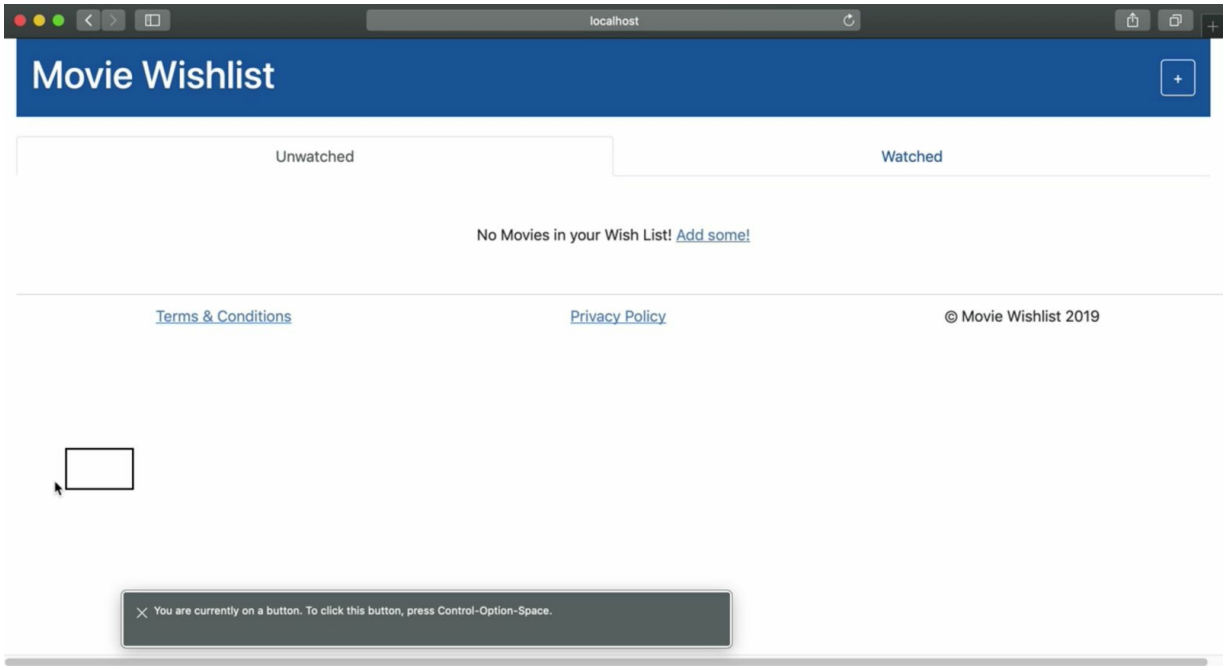


That happens for every single time we navigate from one page to the next. We lose the focus. Let's listen to how this impacts our screen readers.

[00:37] If I run VoiceOver...

Announcer: [00:40] Login button.

Instructor: [00:41] Now, pay attention to where this focus indicator goes when we route to the next page, and listen to what the screen reader reads. If you notice where this focus indicator is (currently on the login button), it's still located over where the login button was on the previous page.



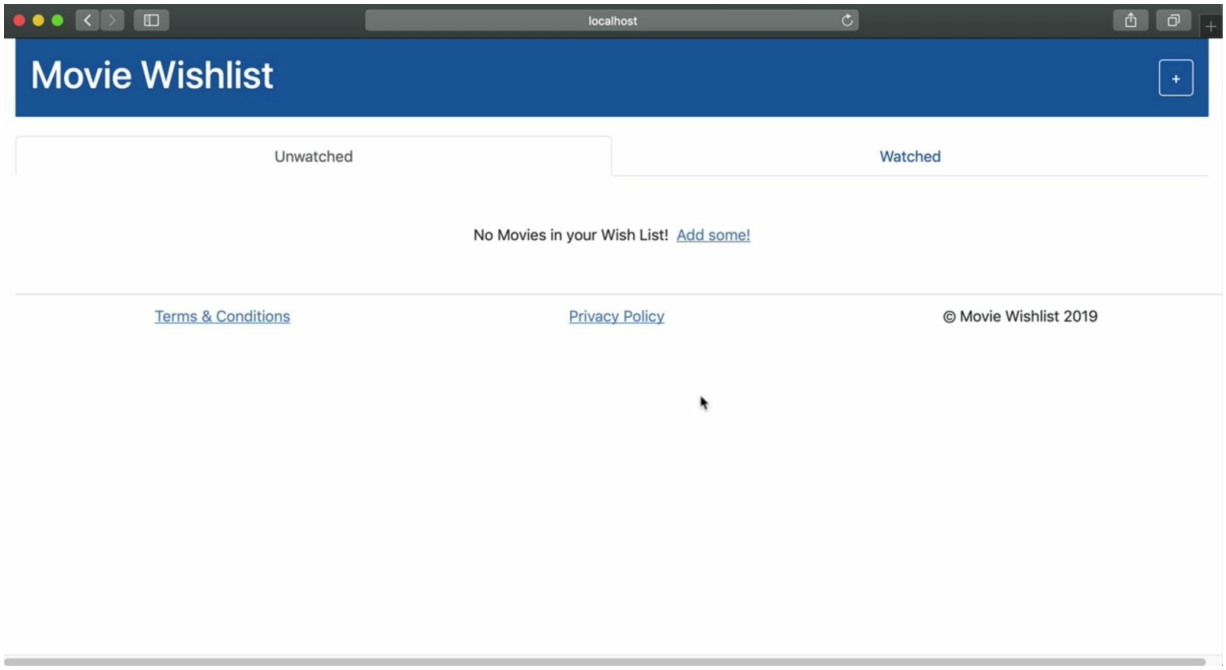
[00:57] What's announced is that you are currently on a button, but it doesn't tell us what button we're on. If I actually go ahead and press this button...

Announcer: [01:05] Press add a movie button.

Instructor: [01:07] presumably, my focus was actually up here on the add a movie button that was here in the header on the previous page. Somehow, the focus went there. We weren't told that that's where it was, and now, I'm on another page altogether. The user can become very lost very quickly this way.

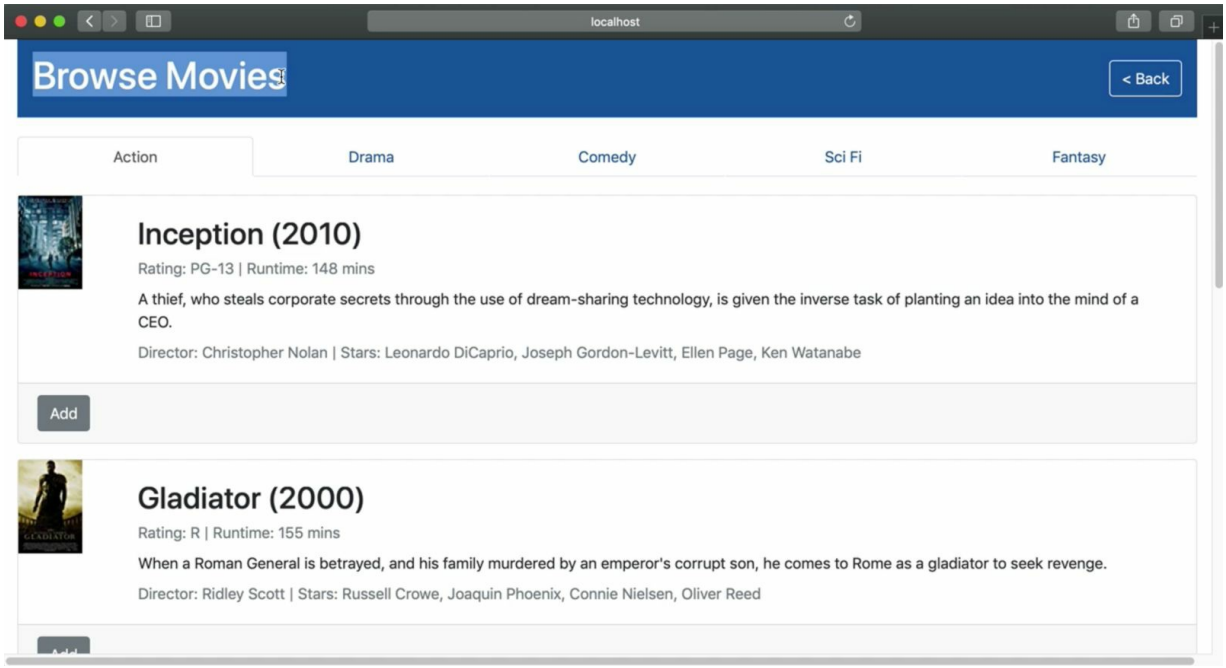
Instructor: [01:26] Let's consider what functionality we want to happen when it comes to setting the focus on each page upon navigation.

[01:34] We want the focus to start somewhere near the top of the page in a place that is logical to the user, so they are not lost somewhere down in the bottom or middle, and they can very easily get to whatever makes the most logical sense for them to need to do first in a linear order on the page.

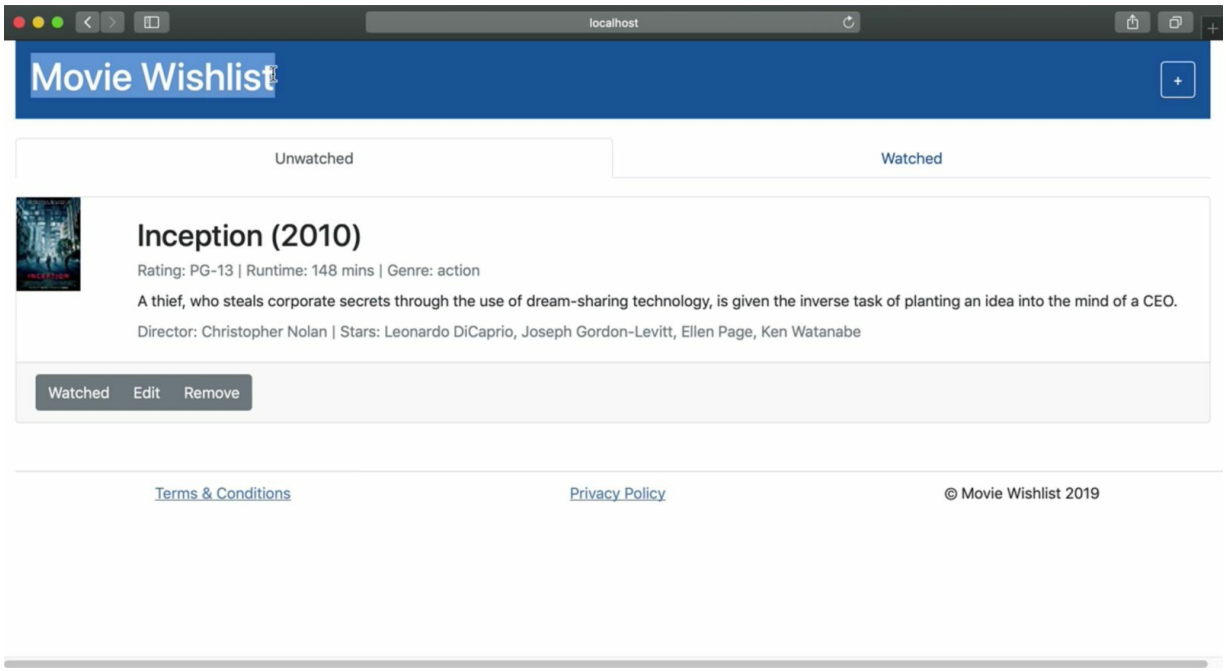


[01:52] This is the first page we come to after we've logged in. A good, logical place to start would be to move the focus right here on the element that says `No Movies in your Wish List! Add some!`, to go ahead and announce that the user has no movies in their wishlist, and that they need to add some.

[02:06] That should be the first thing the user hears. Once they've clicked to add some, and they move to the browse page, we should let them know that they're on the browse page. We should set the focus to this header.



[02:17] From here, they can very quickly get to if they need to go back or move down to pick a genre and use the rest of the page. Finally, if they do have at least one movie in their wishlist, when they come back to the wishlist page, similarly to the browse page, we should set up focus up here to this header and let them know that they're on the wishlist.



[02:37] This is what we're going to do. Let's see how to do that. Here's our `MovieWishlist` component, and all the way we down here, we have this section that we display when we have no movies in our wishlist.

What we're going to want to do is we're going to want to set focus on this `Link`, because it is a focusable element.

```
<main>
{ hasMovies
  ? ( // Show WishList
    <Fragment>
      <TabList tabList={tabList} />

      <div>
        <WishList
          movieList={wishlist}
          watched={match.params.status ===
'watched'}
          movieActions={movieActions}
        />
      </div>

      <MovieEditor
        key={movieInEditing.name}
        movie={movieInEditing}
        updateMovie={this.handleUpdateMovie}
        isOpen={showEditor}
      />
    </Fragment>
  )

  : ( // No movies yet in the WishList
      // This link!
    <div aria-labelledby="noMoviesText addLink"
className="no-movies-container">
      <span id="noMoviesText">
        No Movies in your Wish List!  

        <Link id="addLink" to="/browse"
```

```
            aria-label="Add some movies to your
wishlist now!"
          >
            Add some Movies!
          </Link>
        </span>
      </div>
    )
}
</main>
```

[02:54] In order for us to do that, we're going to need to set a ref to this element, so that we can reference it later and actually call the focus function on it. Up here in my constructor, I'm going to add a member variable that will hold the reference to the link element, `this.addSomeMoviesLink`.

[03:10] Because I'm using React, greater than version 16.3, and I'm using a class component, I'm going to use `React.createRef`. Now, we need to pass this reference to our `Link` element.

```
class MovieWishlist extends Component {
  constructor(props) {
    super(props);

    this.state = {
      showEditor: false,
      movieIdInEdit: null
    };

    this.addSomeMoviesLink = React.createRef();

    this.handleShowEditor =
this.handleShowEditor.bind(this);
    this.handleHideEditor =
this.handleHideEditor.bind(this);
    this.handleUpdateMovie =
this.handleUpdateMovie.bind(this);
  }

  ...
}
```

One quick thing to note is I'm using a component from a third-party library. React router to be specific.

[03:31] I can't just pass a ref prop, necessarily, because this is not an HTML element. It's a higher-level component, and if this component is not expecting a ref prop, it's not going to do anything with it. Specifically, I've looked at the docs, and I know that this link component will take a prop called innerRef that it will actually pass down to the ref attribute on the underlying a tag.

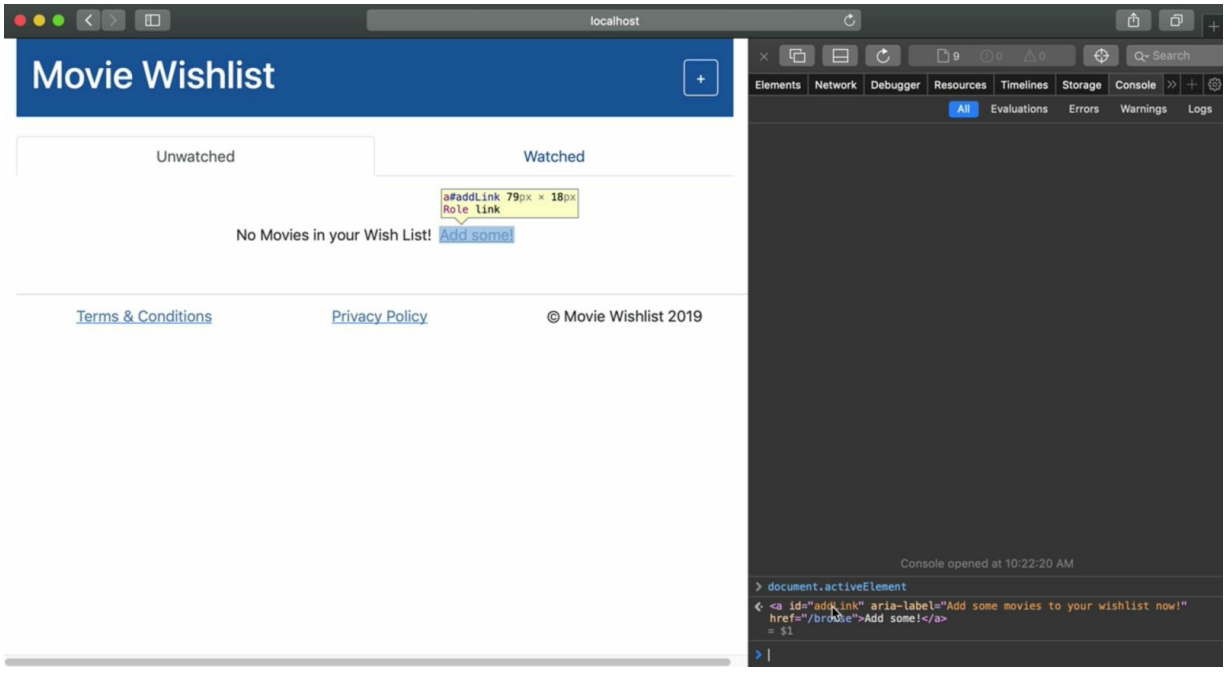
```
: ( // No movies yet in the WishList
  <div aria-labelledby="noMoviesText addLink"
classNamе="no-movies-container">
    <span id="noMoviesText">
      No Movies in your Wish List!  
      <Link id="addLink" to="/browse"
        aria-label="Add some movies to your
wishlist now!"
        // innerRef is the prop used by react-
router's Link to forward the ref attribute
        // to the underlying <a> element
        innerRef={this.addSomeMoviesLink}>Add
some Movies!</Link>
    </span>
  </div>
)
```

[03:56] We're going to pass it our variable that's holding the ref that we created. Now, we have a reference to this element. Finally, we need to add a `componentDidMount` life cycle event function. This is so we can make sure our focus gets set when the component mounts, when the page loads.

[04:13] First, we need to make sure that our ref has already been set and exists. Now that we know that our variable exists, and it's actually been set with a reference to an element, we can call focus on it. Now, let's try that out.

```
componentDidMount() {
    if (this.addSomeMoviesLink &&
this.addSomeMoviesLink.current) {
      this.addSomeMoviesLink.current.focus();
    }
  }
```

[04:33] Here we are back on our login page. Once I hit login and we route to the wishlist, we can now visually see the focus indicator is here on the link. Let's just double-check that real quick by inspecting the active element, and yep, it's on our link, exactly where we wanted it.



[04:55] Now, let's look at how to set the focus on the page heading when we get to browse movies or when we go back to movie wishlist. Here we are in our `Header` component, and this is shared by both the wishlist and browse pages.

primitives/Header.js

```
const Header = ({ title, buttonText,
buttonLabel, handleButtonClick, doFocus }) => {
  return (
    <header>
      <nav className="navbar navbar-dark">
        <span className="navbar-brand">
          <h1>{title}</h1>
        </span>
        <button
          className="btn btn btn-outline-light"
          onClick={handleButtonClick}
          aria-label={buttonLabel}
        >
          {buttonText}
        </button>
      </nav>
    </header>
  );
};
```

[05:18] Similarly, we're going to need to set a variable with a ref. Because I'm using greater than React 16.8, and I'm in a functional component, versus a class component, I can call a `useRef` hook. Now, I need to pass that reference to my h1 element, which is where I want to set the focus.

```
const Header = ({ title, buttonText,
buttonLabel, handleButtonClick, doFocus }) => {
  const headerRef = useRef(null);

  return (
    <header>
      <nav className="navbar navbar-dark">
        <span className="navbar-brand">
          <h1 ref={headerRef} tabIndex={0}>
{title}</h1>
        </span>
        <button
          className="btn btn btn-outline-light"
          onClick={handleButtonClick}
          aria-label={buttonLabel}
        >
          {buttonText}
        </button>
      </nav>
    </header>
  );
};
```

[05:38] Now, an additional thing we're going to need to do is we need to make this h1 focusable. By default, h1 elements are not focusable, because they're not interactive. You can't take action on them. In order to make this h1 focusable, we need to give it a tab index of zero.

[05:54] Note that React uses tab index with a capital I and camelcase, versus all lowercase. You'll see some red squigglies here and note that I now have an ESLint finding, because it has this rule that only interactive elements -- elements that you can take actions on -- should be focusable.

[06:11] I disagree with this, and I believe that the user experience of

setting the focus at the page title when routing from one page to the next is a better user experience than setting the focus on whatever the first focusable element may be.

[06:26] I feel like this is a rule we need to make an exception for, so I'm just going to suppress that.

```
return (
  <header>
    <nav className="navbar navbar-dark">
      <span className="navbar-brand">
        {/* eslint-disable-next-line jsx-
a11y/no-noninteractive-tabindex */}
        <h1 ref={headerRef} tabIndex={0}>{title}
</h1>
      </span>
      <button
        className="btn btn btn-outline-light"
        onClick={handleButtonClick}
        aria-label={buttonLabel}
      >
        {buttonText}
      </button>
    </nav>
  </header>
);
```

Now, that we have our reference to the h1, we need to similarly set it when the component mounts. Since I'm in a functional component, I'm going to do that with a `useEffect` hook.

[06:45] Same as what we had in the movie wishlist component, we want to make sure that our reference exists and that it's been set, and when it is, we can call focus on it.

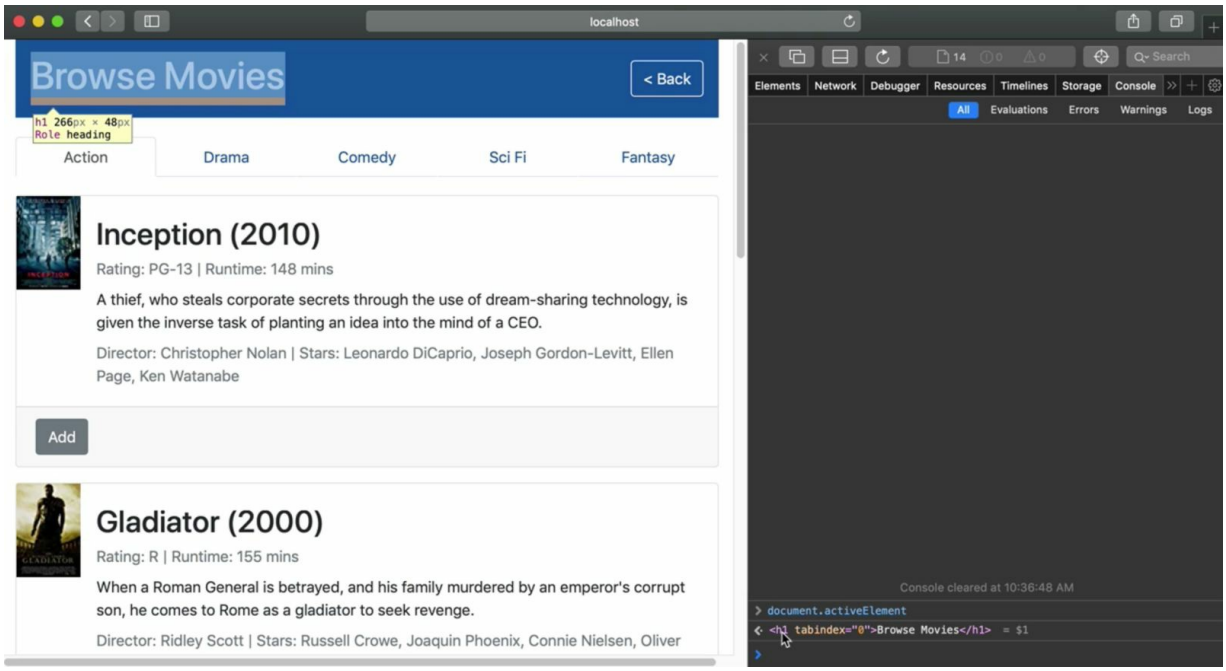
```
const Header = ({ title, buttonText,
buttonLabel, handleButtonClick}) => {
  const headerRef = useRef(null);

  useEffect(() => {
    if (headerRef && headerRef.current) {
      headerRef.current.focus();
    }
  });

  ...
}
```

Let's test that out now. Here we are in our wishlist.

[06:59] We're going to click the link to add some movies, and we can see here our focus indicator has moved to this h1, our page title. Let's check here, and we can confirm that the focus has been set on our h1.



Now, we've got one last thing to do.

[07:17] We want to set focus on the page title when we go back to the wishlist, but only when there are movies in our wishlist. When there are not movies in our wishlist, we want to go to the link. We have to make sure that that's conditional.

[07:30] If we add a prop to this `Header` component called `doFocus`, and it's a Boolean, and we'll give it a default of false, so that not everybody has to supply it if they don't want to. We're going to add that to our checking.

```
Header.defaultProps = {
    buttonText: '',
    buttonLabel: null,
    doFocus: false
};

Header.propTypes = {
  title: PropTypes.string.isRequired,
  buttonText: PropTypes.string,
  buttonLabel: PropTypes.string,
  handleButtonClick: PropTypes.func.isRequired,
  doFocus: PropTypes.bool
};
```

[07:45] Before we actually set focus, let's make sure that we explicitly want to. Now that this is being used within our `useEffect`, we need to make sure that it's part of our dependencies. Finally, we need to make sure that the movie wishlist and movie browse components are passing this in appropriately.

```
useEffect(() => {
    if (doFocus && headerRef &&
headerRef.current) {
      headerRef.current.focus();
    }
}, [doFocus]);
```

[08:04] For our `MovieBrowser` component, we want to always focus. This is just going to always be true.

browse/MovieBrowser.js

```
const MovieBrowser = ({
    history,
    match,
    wishlist,
    addToWishlist,
    removeFromWishlist
}) => {

  ...

  return (
    <div>
      <Header
        title="Browse Movies"
        buttonText="< Back"
        buttonLabel="Back to Wish List"
        handleButtonClick={goToWishlist}
        doFocus
      />

    ...
  )
}
```

Then our wishlist, we want this to be conditional on whether we have movies or not. Here in our JSX is where we decide whether to show the movies in wishlist or our message about not having any.

```
return (
  <div>
    <Header
      title="Movie Wishlist"
      buttonText="+" buttonLabel="Add a Movie"
      handleButtonClick={goToBrowse}
      doFocus={hasMovies} // only focus the
header if we have movies, otherwise focus the
Add some movies! link
    />

    ...
)
```

[08:25] Here in our header, we can use that variable that tells us whether we have movies or not. If we have movies, we want to focus on the header. One last clean-up item. We probably shouldn't even show our tabs of whether the movies or unwatched if we don't have any movies. Let's just move that into here into the `hasMovies` conditional.

[08:43] Now, let's test it all out. If we go back to the beginning, to our login page, and let's use the screen reader, let's watch where the focus goes now when I push the login button.

Announcer: [08:52] Login button, visited link, add some movies to your wishlist now. No movies in your wishlist.

Instructor: [08:58] There we go. We initially focus on the link, as we had previously, and that's the first thing that's read to the user. If we go ahead and click the link...

Announcer: [09:08] Heading level one, browse movies, navigation. You are currently on a heading level one.

Instructor: [09:14] Once this page loads, we focus directly on that page title, and that is the first thing we hear. It's very clear to us what page we have navigated to. Now, if we add a movie...

Announcer: [09:24] Add "Inception."

Instructor: [09:25] and then navigate back to our wishlist...

Announcer: [09:27] Back to wishlist button. Heading level one, movie wishlist, navigation. You are currently on a heading level one.

Instructor: [09:35] Since we have movies in our wishlist, we focus on the page title, just as designed.