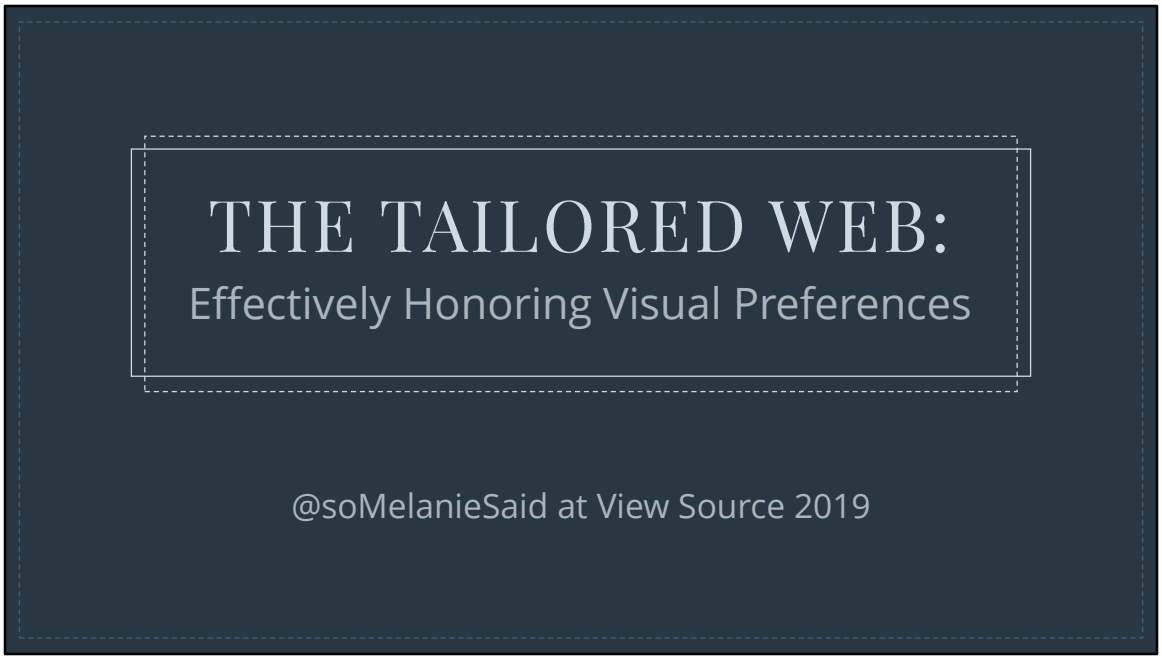
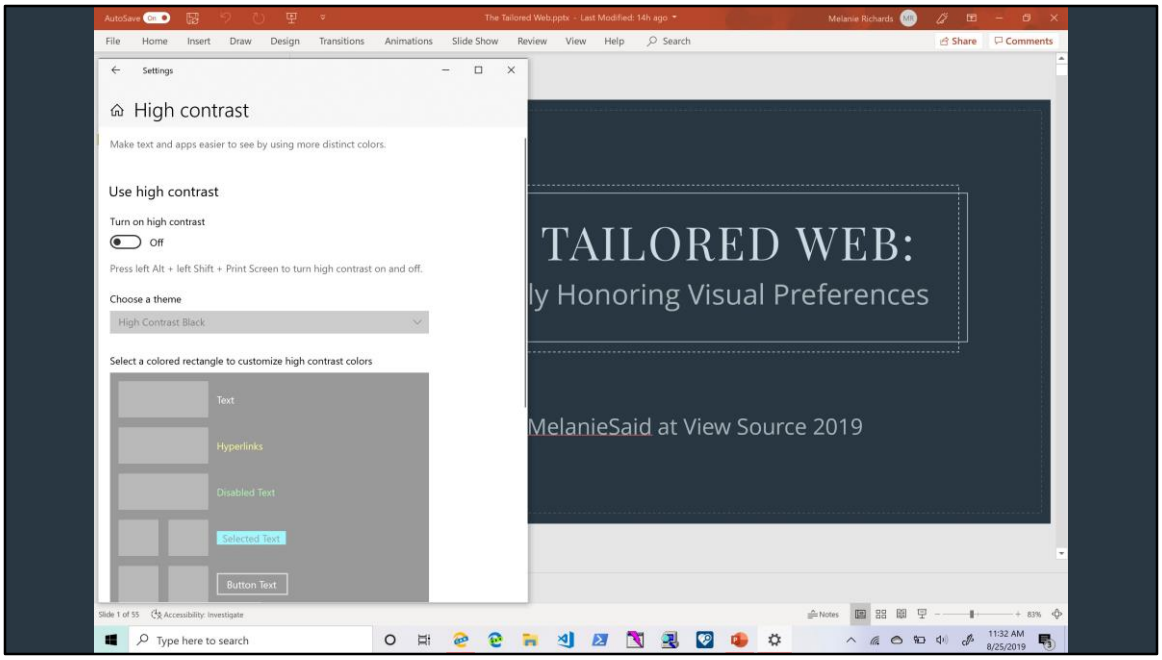
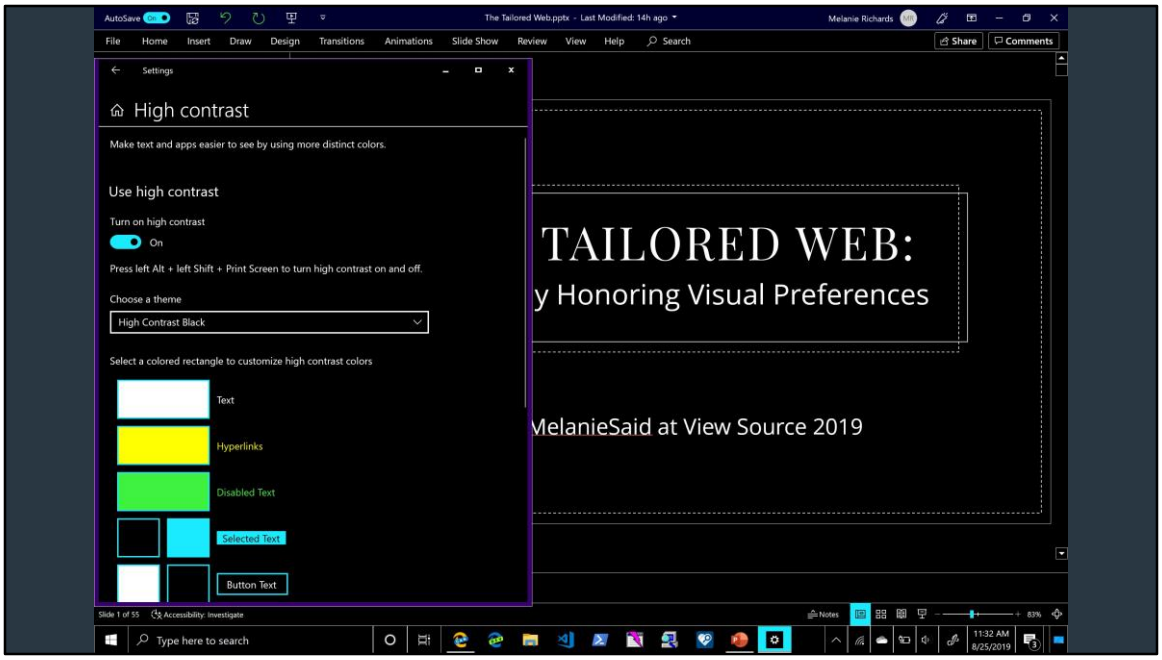


Hi everyone, I'm Melanie, a member of the Microsoft Edge web platform team and participant in W3C web standards communities, including the CSS Working Group.



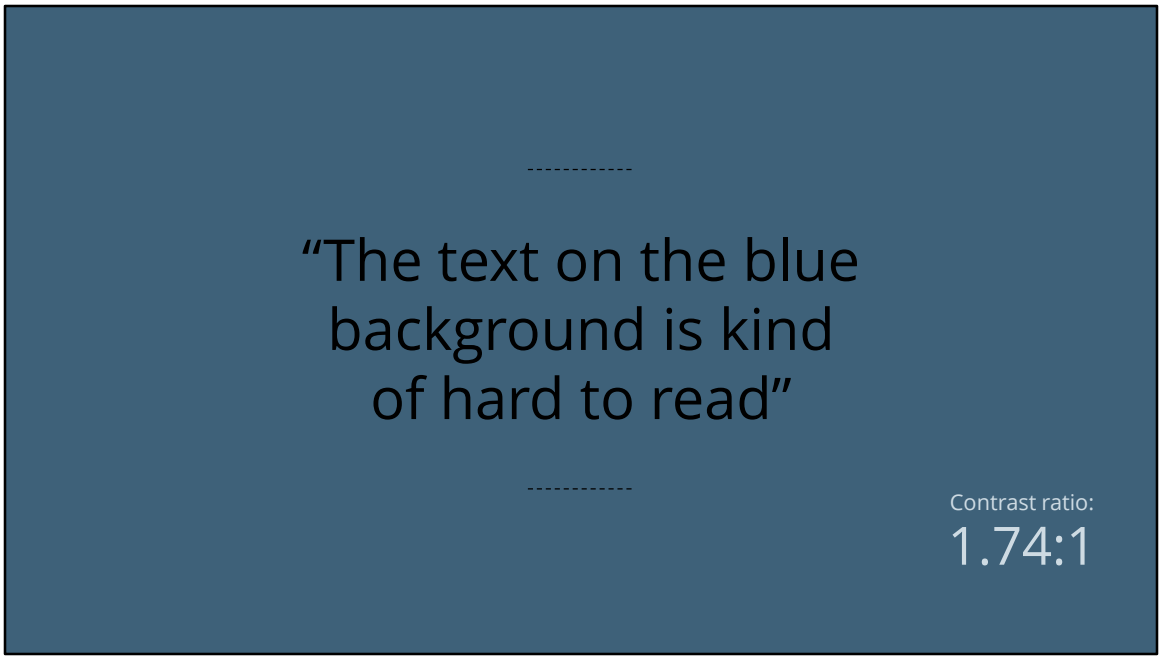
The CSS working group has recently been developing standards for integration with user system settings, such as Windows High Contrast. If you're unfamiliar, High Contrast is an accessibility feature intended to increase text legibility while reducing cognitive load. The feature works by enabling the user to select theme colors for a scoped number of semantic elements.



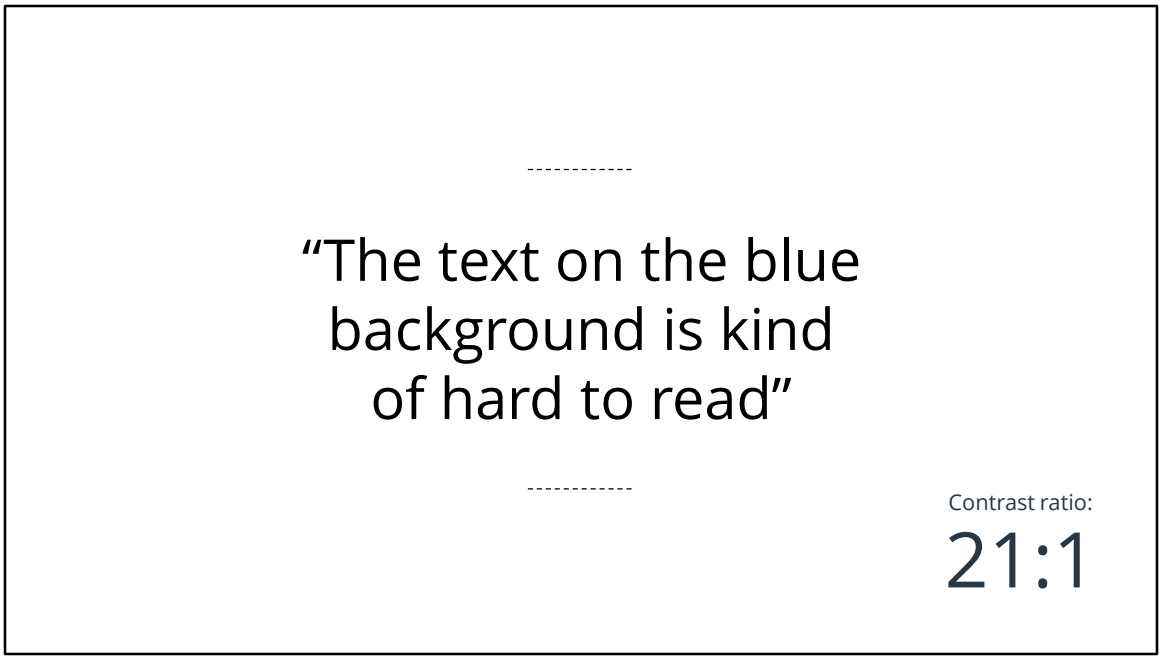
This scheme can then be applied to user interfaces and app content, reducing visual complexity while guaranteeing the user's preferred contrast level. Here's an example of the operating system and some apps rendered in a High Contrast Black theme; if you don't have a Windows machine, I'll share later on in the presentation how you can experiment with this feature.

In any case, like other apps, browsers can render their UI and even web content in the user's provided color scheme. In parallel to bring High Contrast styling techniques to standards, I was part of a user study to determine how well High Contrast in web content is currently working for individuals with low vision.

We had some folks go through websites with and without High Contrast enabled. One individual, while going through a site rendered in its original color scheme, remarked: "The text on the blue background is kind of hard to read."



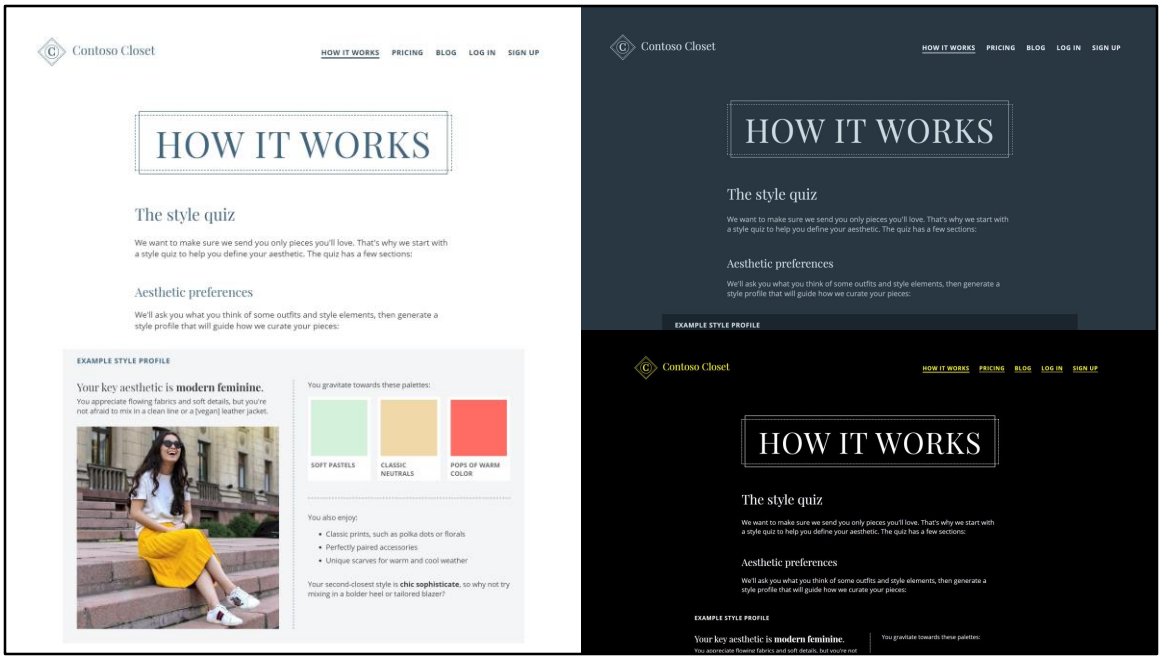
You might think that this was some type of shenanigans where the designer put dark text on a dark background, like some footer text they were trying to make more subtle.



But actually, the participant was referring to black text on a white background. You cannot have a higher foreground-background contrast ratio than this. And still, that person's eyes were picking up on the blue light, and they had trouble reading the text.

We turned on High Contrast with a dark theme, and the participant said "that's much better!" This is the same exact contrast ratio, but in a color scheme that was easier for this particular individual to read.

In web design and development, we put a lot of energy into making some One True Design a pleasant and useful experience often for the greatest number of people. But the truth is that our users bring with them individualized needs for color and contrast, which can change by the day or the hour. Maybe someone had a recent injury that makes it hard to look at certain colors. Or they find that lower text contrast makes it easier for them to focus on words. Or maybe they're trying to finish a major project in the middle of the night without keeping their partner up. How are we supposed to put together one color scheme that simultaneously meets all the many unknown visual needs of our users?

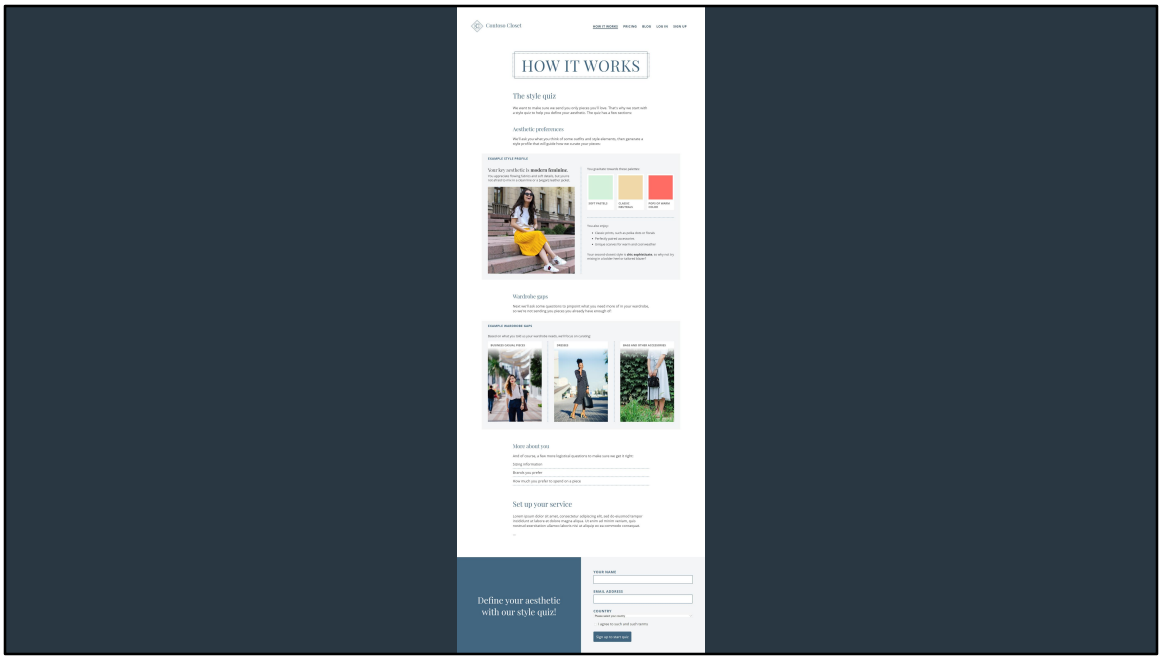


While we should provide reasonable defaults in our base theme, the good news is, we can take a layered approach towards inclusive color and contrast. Emerging media queries and CSS properties enable us to respond to hints as to the user's preferred color scheme, contrast preference, and the presence of forced color modes like High Contrast.

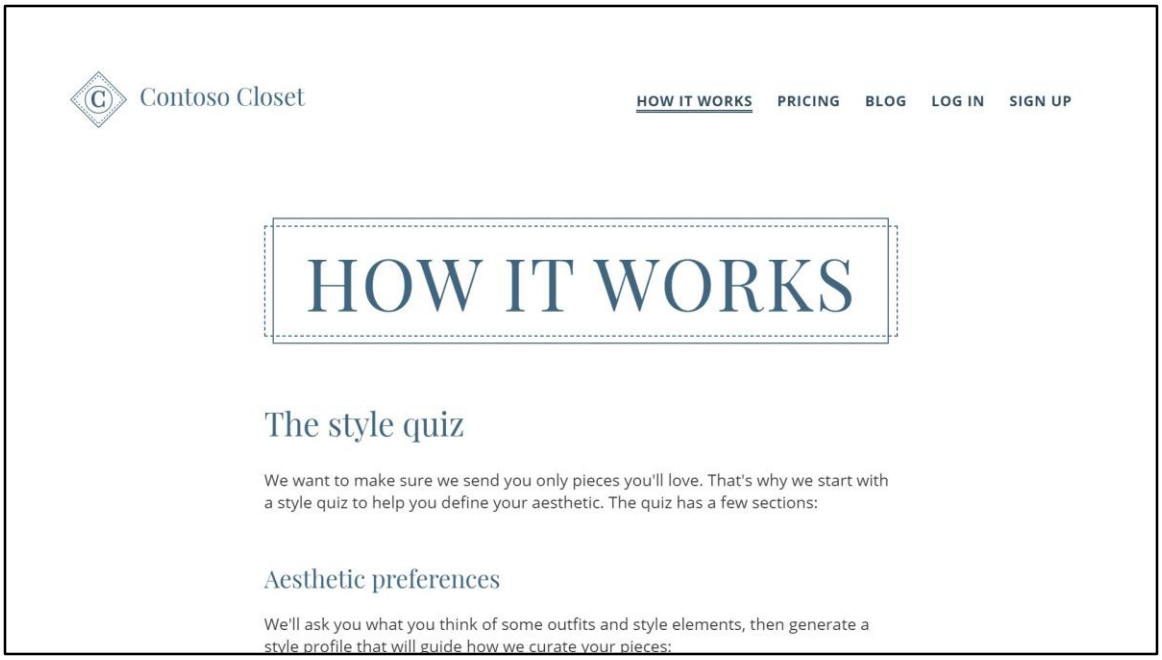
We can tailor our websites to users' visual needs…

Using these new standards, we can efficiently tailor our websites to users' visual needs…

While maintaining a design point-of-view.



So let's suppose that we are building out a marketing page for a service that sends you clothing tailored to your individual preferences.



Zooming in a bit here, our base brand is a light theme with cool accent colors.

EXAMPLE STYLE PROFILE

Your key aesthetic is **modern feminine**.

You appreciate flowing fabrics and soft details, but you're not afraid to mix in a clean line or a [vegan] leather jacket.

You gravitate towards these palettes:

SOFT PASTELS

CLASSIC NEUTRALS

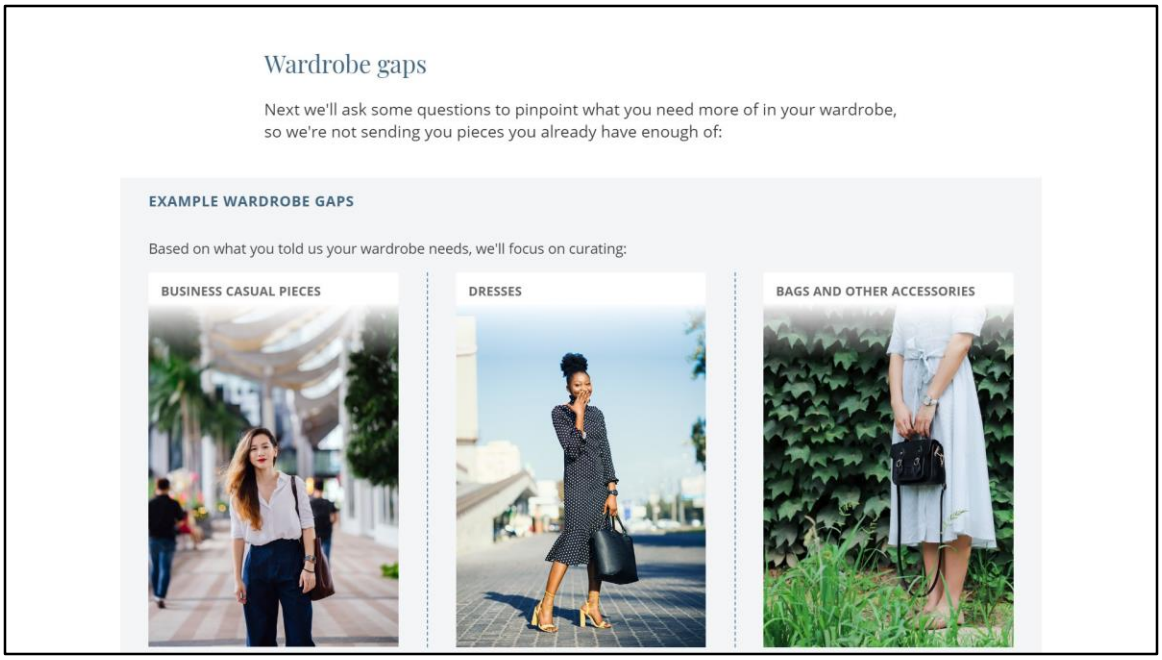
POPS OF WARM COLOR

You also enjoy:

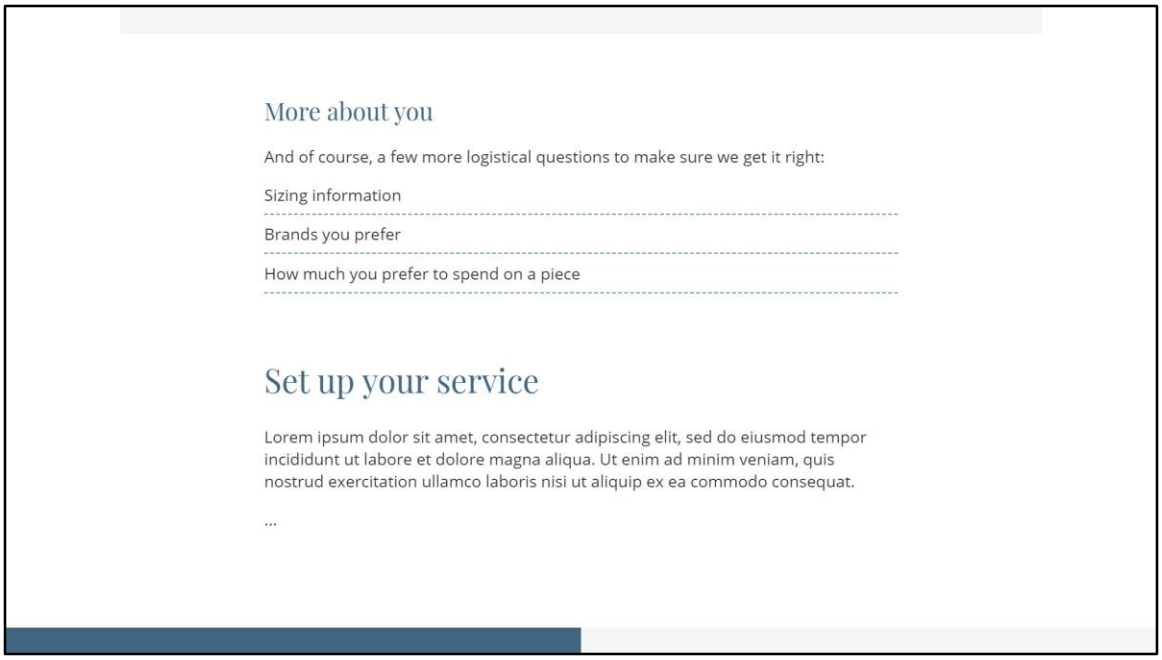
- Classic prints, such as polka dots or florals
- Perfectly paired accessories
- Unique scarves for warm and cool weather

Your second-closest style is **chic sophisticate**, so why not try mixing in a bolder heel or tailored blazer?

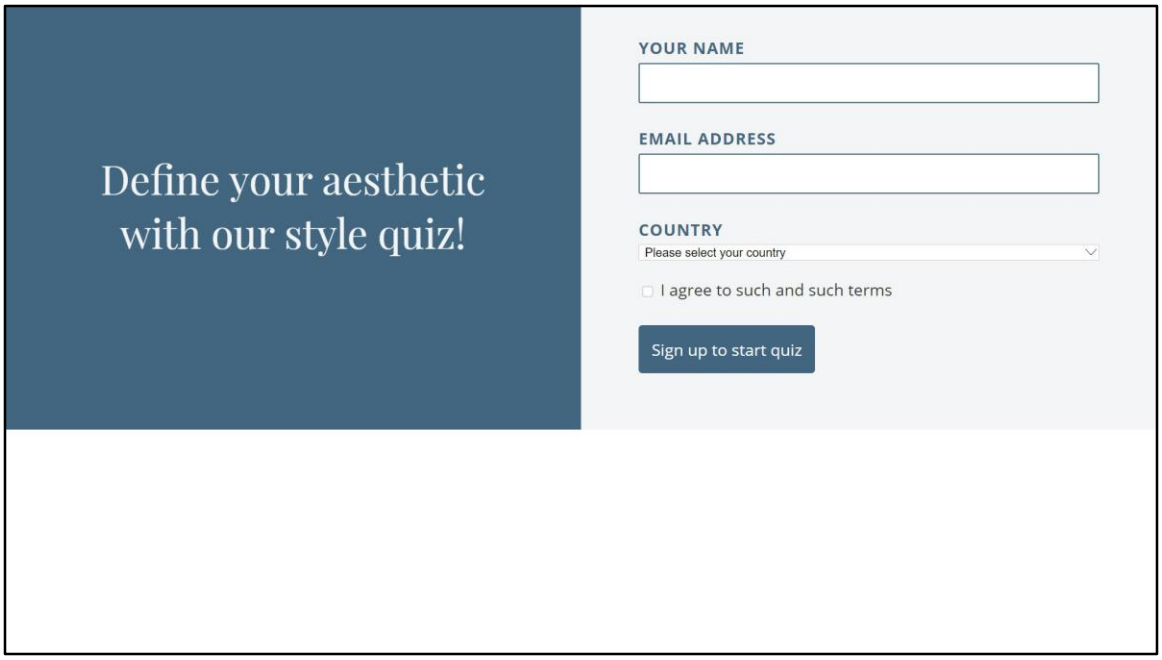
We've got some mixed content here that includes a sample color palette,



And gradients overlaid on top of photography



As well as decorative dashed borders.

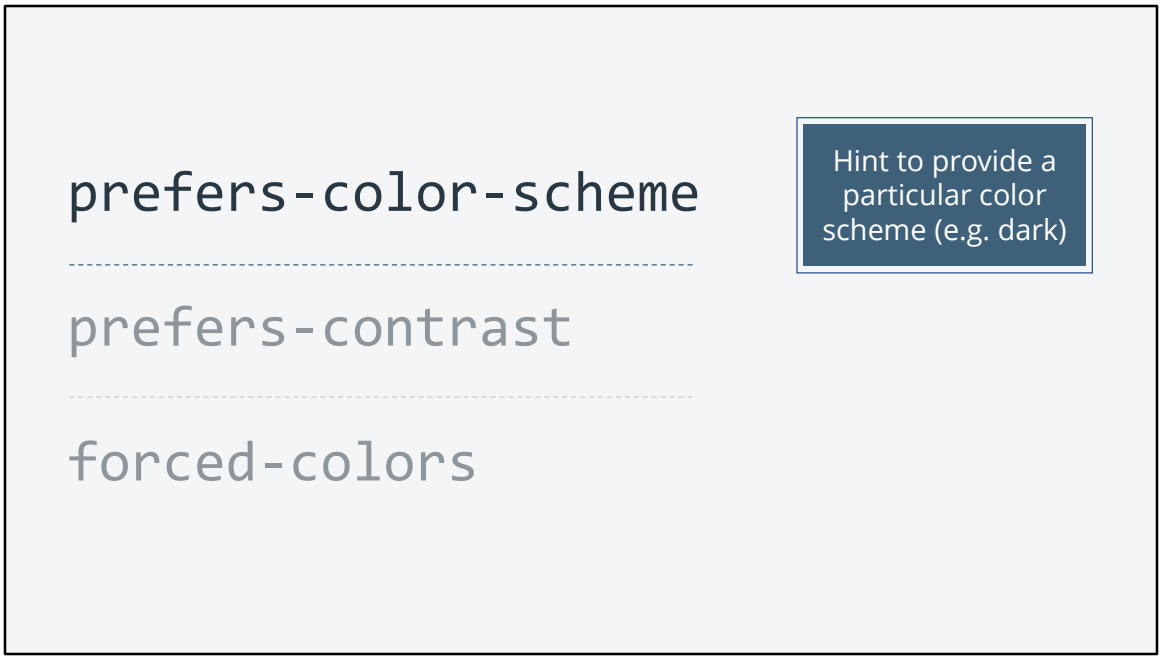


We also have some form fields at the bottom of our page.

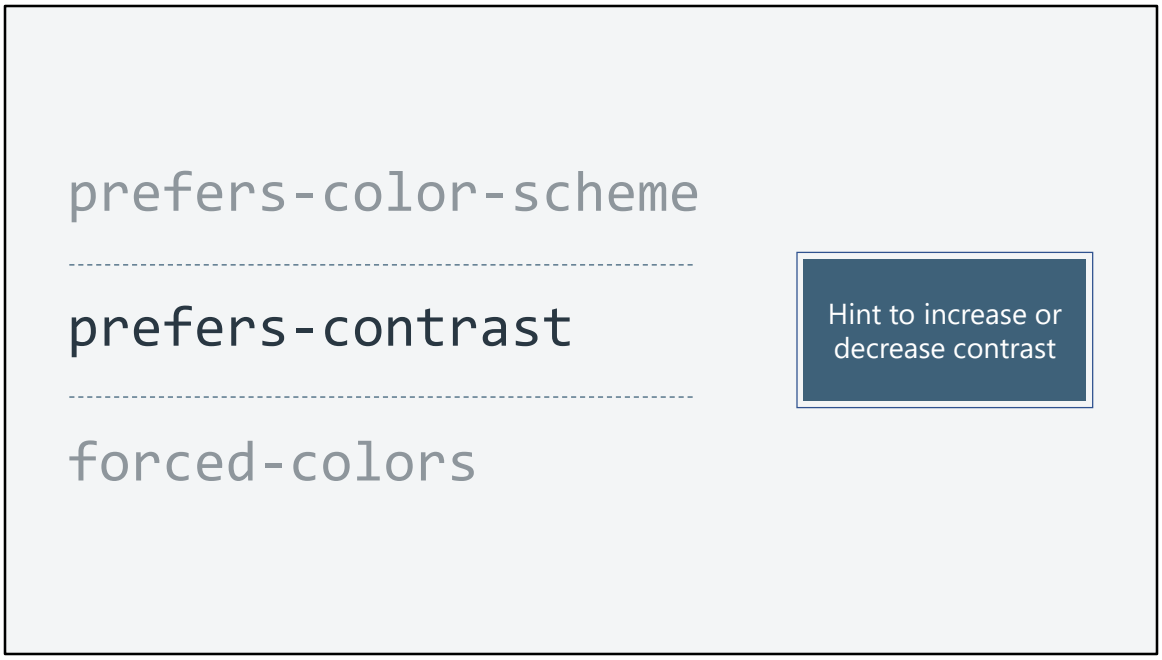
A FEW RELATED MEDIA QUERIES

Let’s start with a brief overview of the media queries we’ll use to create variations on our brand.

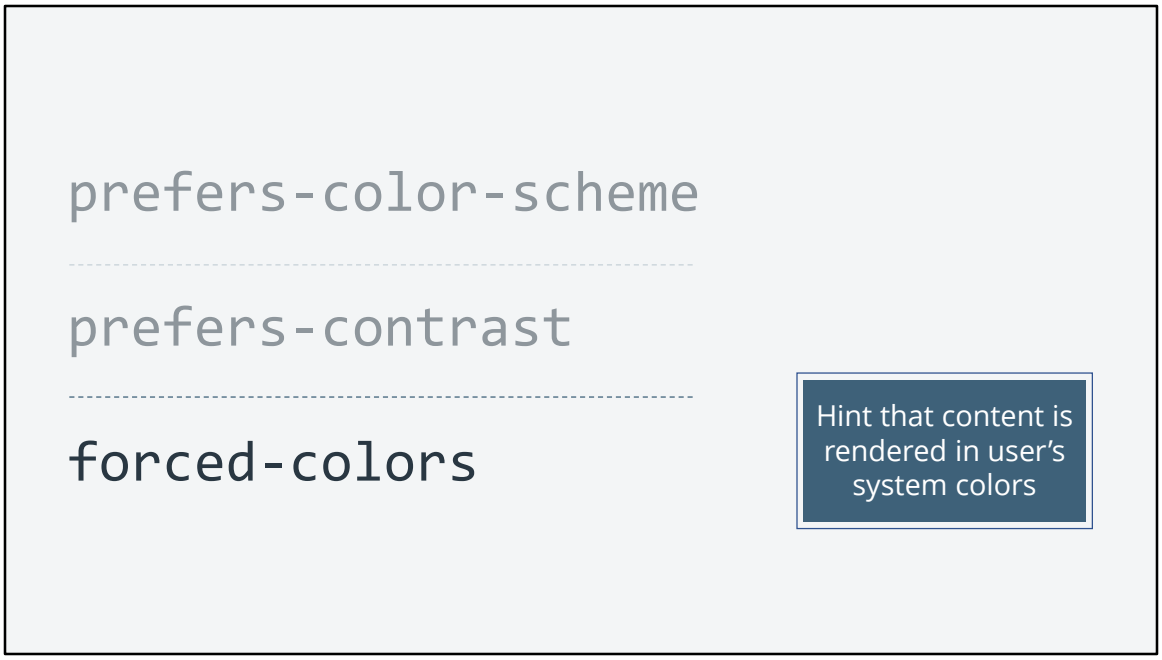
Prefers-color-scheme, prefers-contrast, and forced-colors



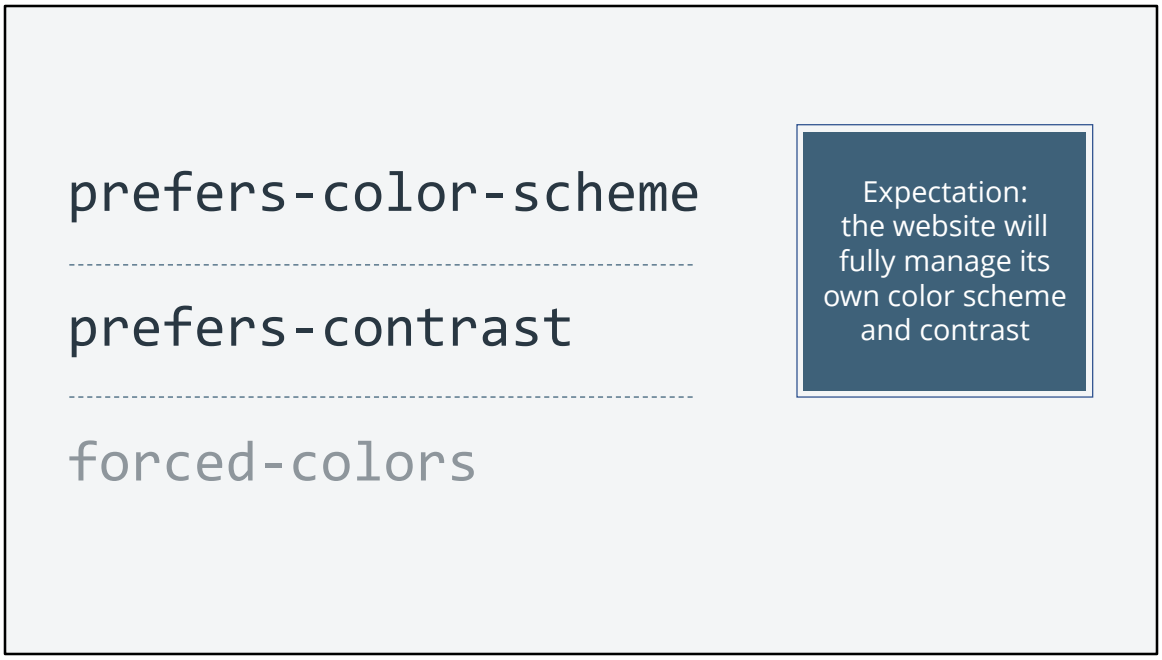
Prefers-color-scheme is a hint to provide a particular color scheme. This is heavily associated with dark mode.



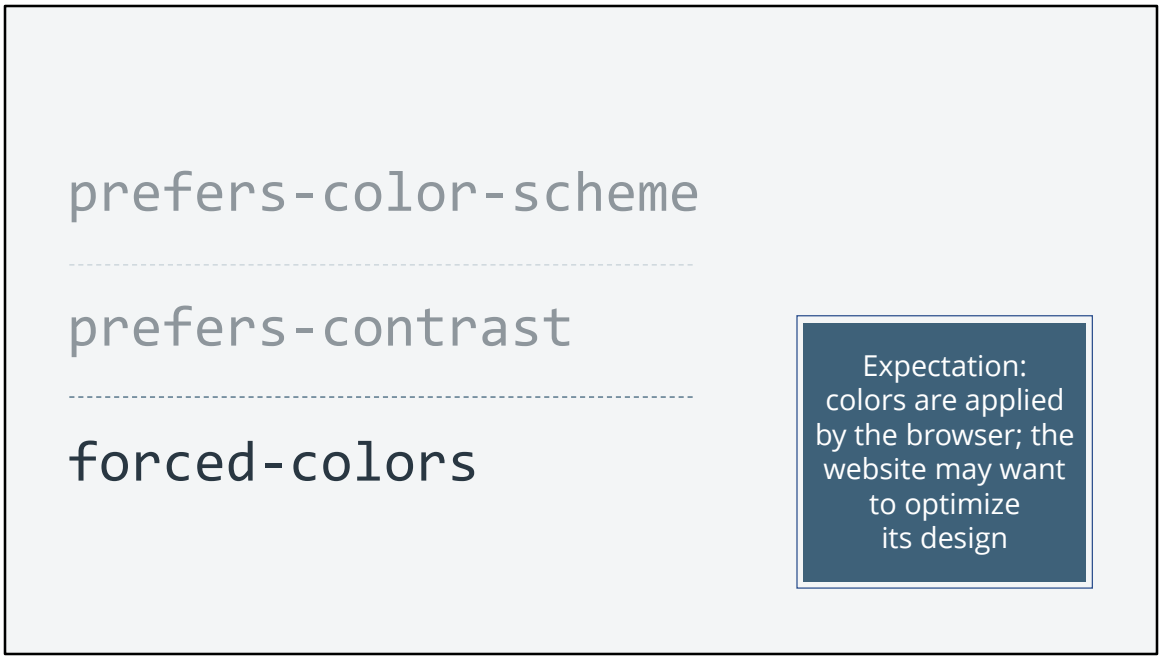
Prefers contrast is a hint to either increase or decrease contrast



And forced colors is a hint that content is rendered by the browser's rendering engine in the user's chosen colors. For example, when High Contrast mode is active.

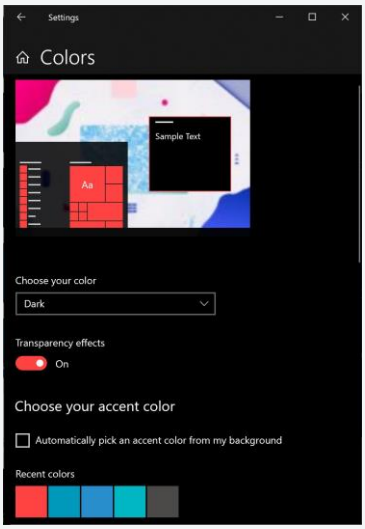


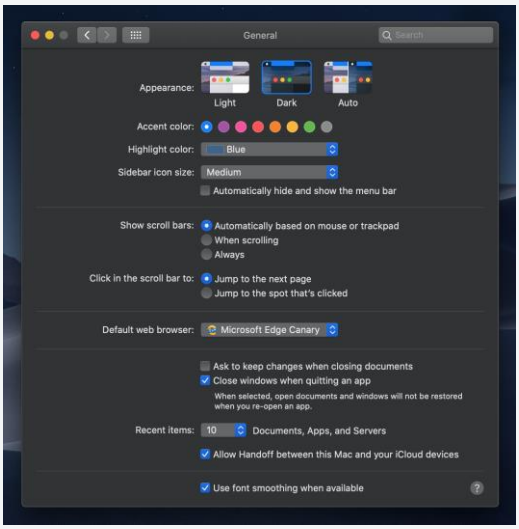
There's a couple different expectations on the web developer. For prefers-color-scheme and prefers-contrast, the expectation is that the website will fully manage its own colors and background-foreground contrast.



In forced color modes, these colors have already been applied, and the website may want to optimize its design.

Let’s start by adding another color scheme.





The new prefers-color-scheme media query allows web developers to pick up this hint from the user's operating system. Their browser could provide its own color scheme settings as well, if it so chooses. Let's suppose that we want to support dark mode on our page.

Custom properties

```
/* In base theme */

:root {
  --bg-color-main: #fff;
  --bg-color-subtle: #F3F5F6;
  --color-brand: #426680;
  …
}
```

One strategy that makes additional schemes way easier is CSS custom properties, known in common parlance as variables. Instead of declaring static color values all over our pattern library, we can set some variables on the :root node, inside of our base theme, that is, outside of any particular media query. So for example, we're setting bg-color-main to white.

Custom properties

```
body {
  background-color: var(--bg-color-main);
}

…
```

Then we access them from elsewhere in our styles. Here we're setting the body background color to bg-color-main, and we'll use the rest of the variables throughout our base styles.

Media query syntax

```
@media (prefers-color-scheme: dark) {}
```

Values:

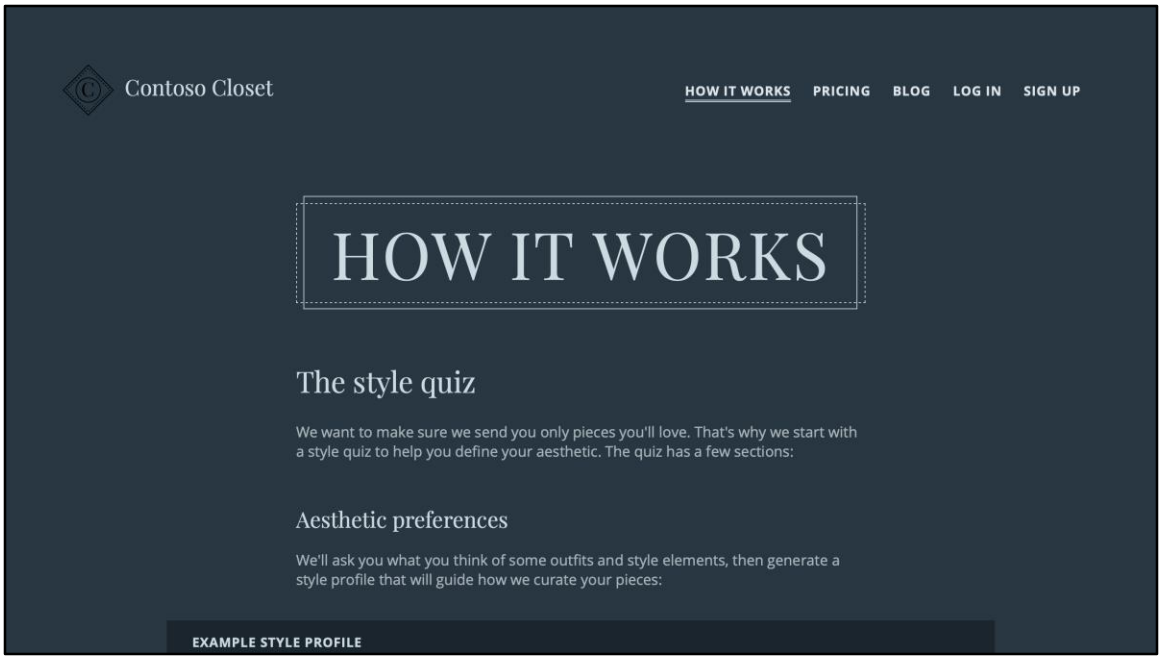
```
no-preference | light | dark
```

To modify the theme for our dark mode, we'll want to query for prefers-color-scheme: dark

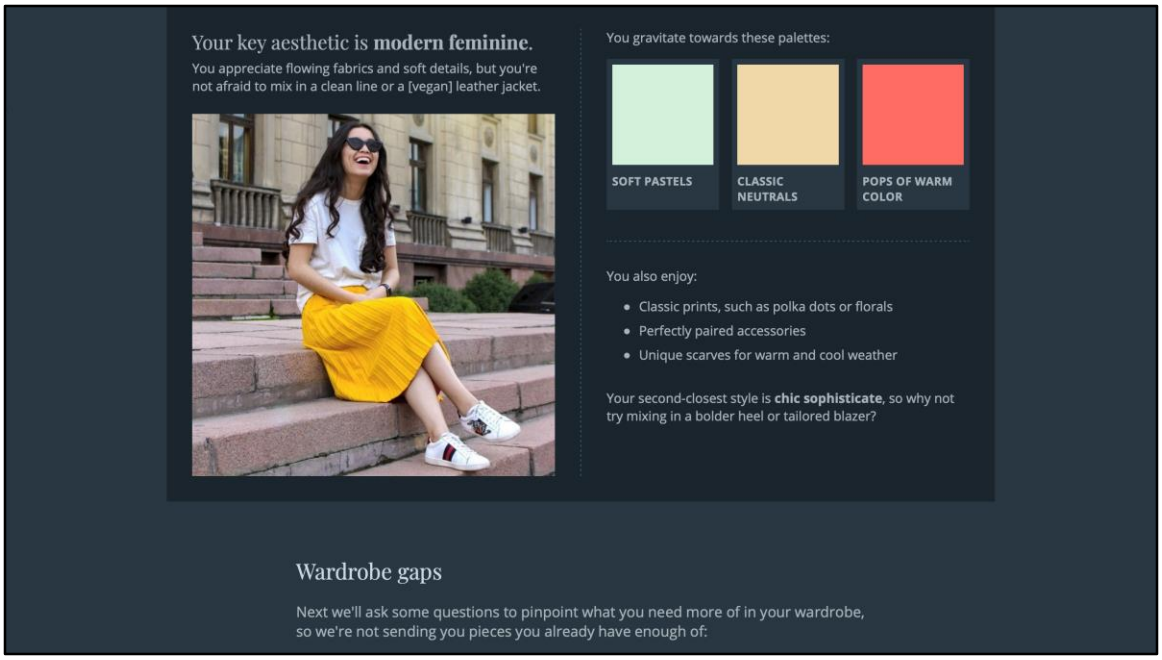
Update custom property values

```
@media (prefers-color-scheme: dark) {
  :root {
     --bg-color-main: #293742;
     --bg-color-subtle: #1b252c;
     --color-brand: #cedbe3;
     …
  }
}
```

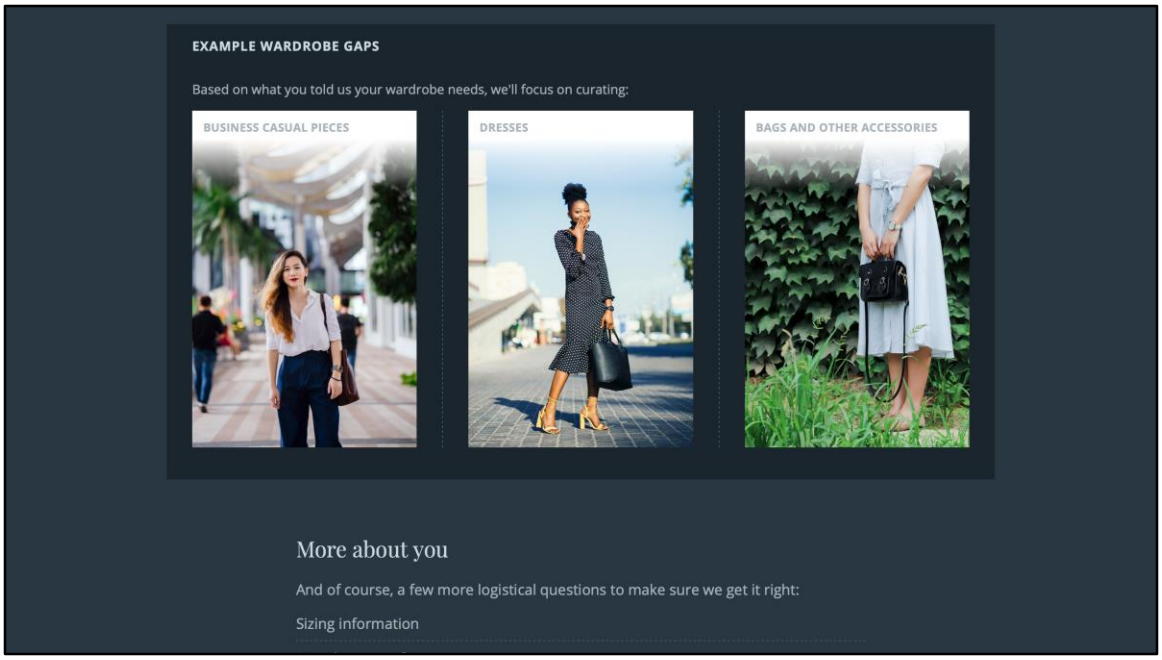
And within that media query, we'll set our dark color scheme. Dark mode often gets associated with shades of grey, but there's nothing that says thou shalt remove all color. The dark blue is really important to our hypothetical brand, so we're going to do this up in shades of blue, with the saturation reduced a bit. We reset the value of those custom properties on the root element, and these will propagate to everywhere that we've used the variables. So for example, bg-color-main is now a dark blue color, and the body background will pick up that change.



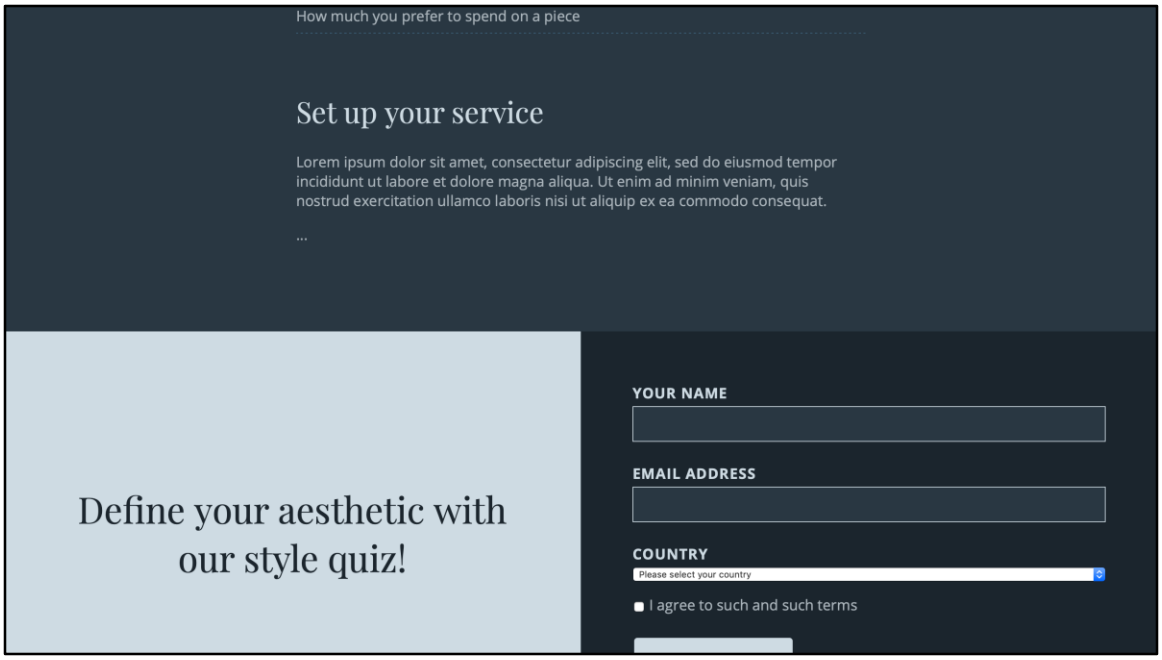
And, now our page is dark blue!



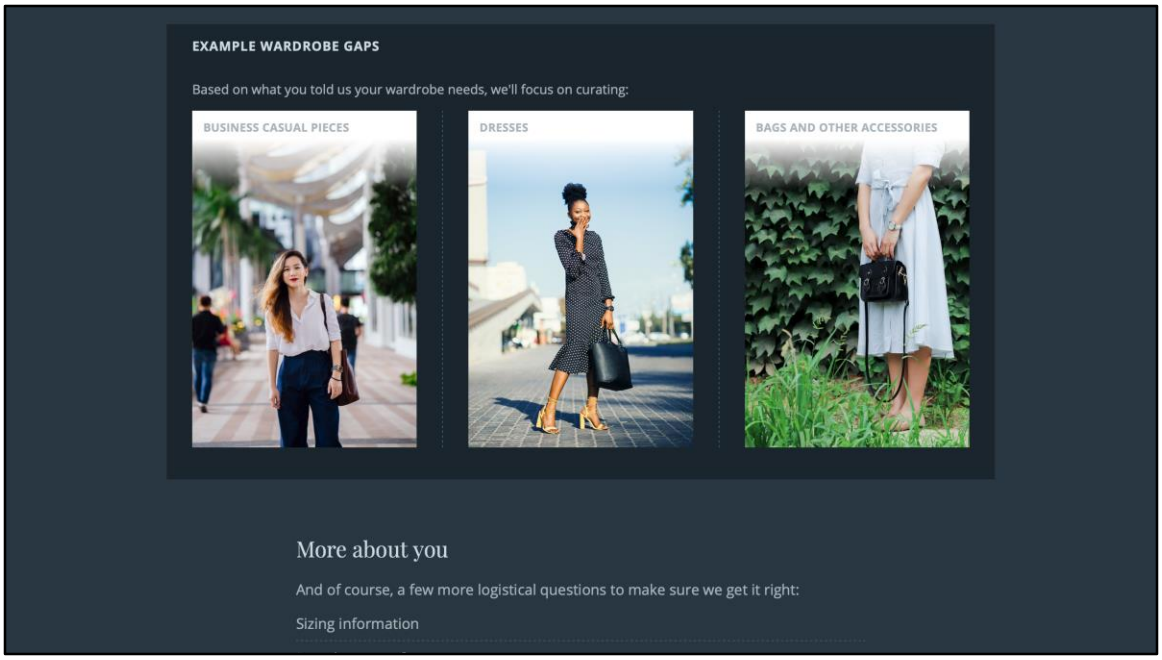
By updating our color variables…



…we've got most of this page styled



In a matter of 10 minutes.



A couple little things need to be adjusted. For example we should update this gradient to a darker color…

EXAMPLE WARDROBE GAPS

Based on what you told us your wardrobe needs, we'll focus on curating:

BUSINESS CASUAL PIECES

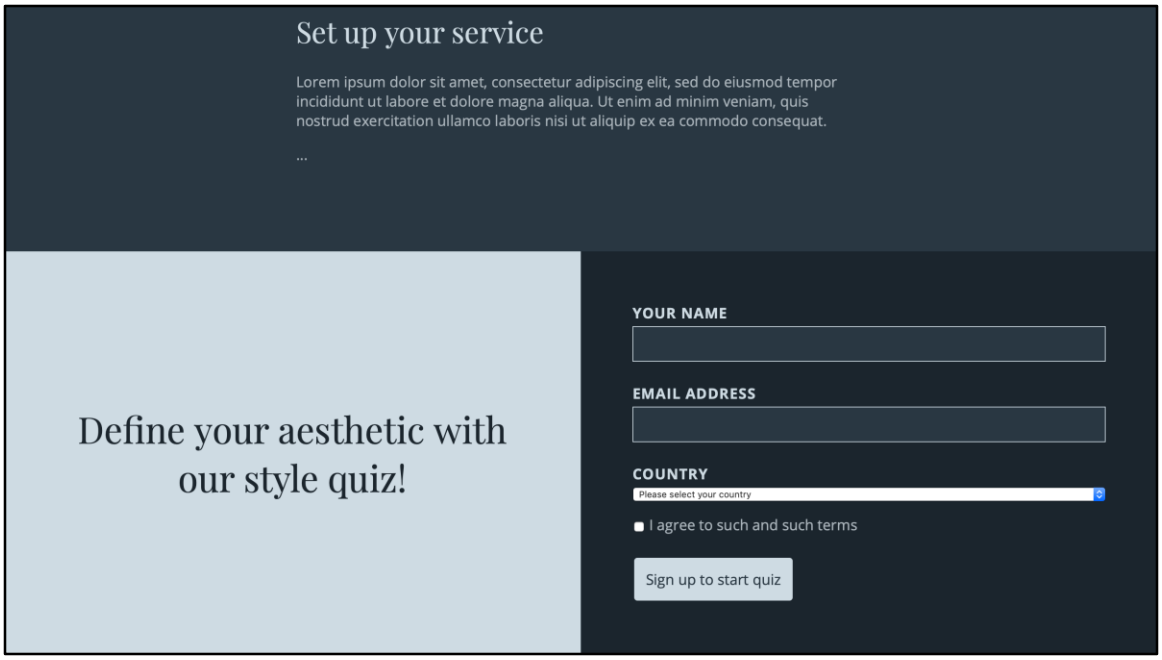
DRESSES

BAGS AND OTHER ACCESSORIES

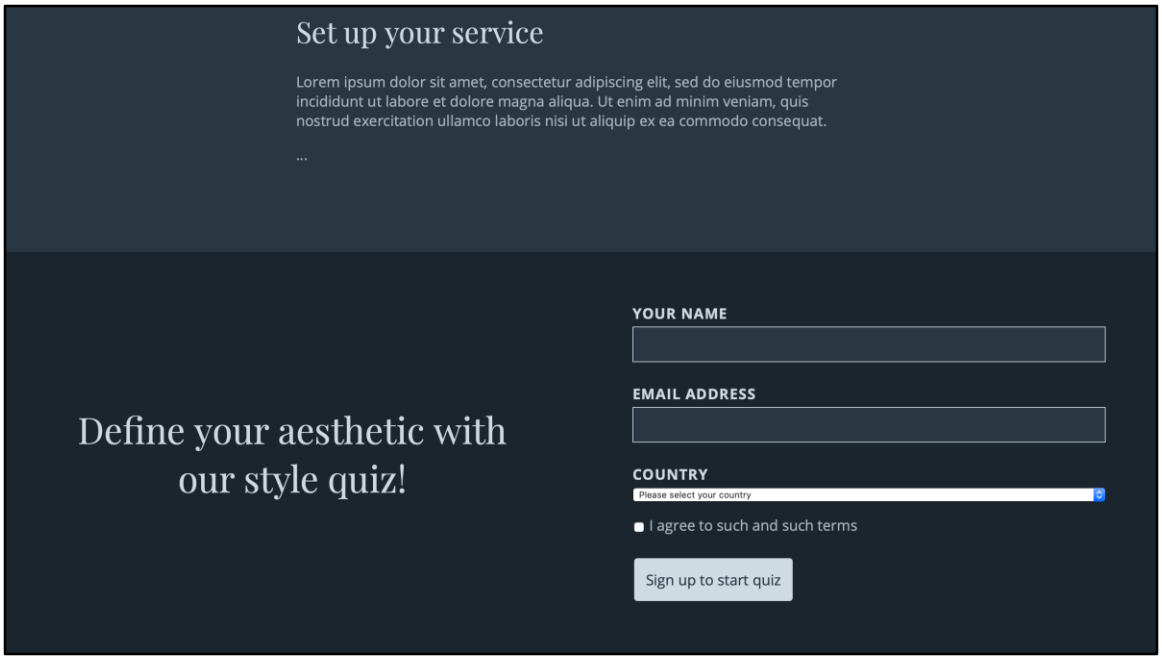
More about you

And of course, a few more logistical questions to make sure we get it right:

Sizing information



And this rectangle is too bright after a simple color swap.



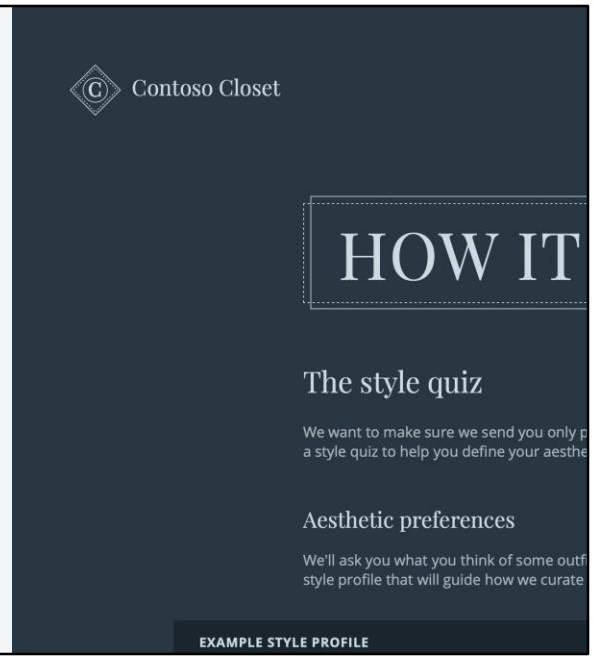
So we can update styles like this manually.

currentColor

```
/* In base theme */

.header__brand-icon * {
  stroke:
  currentColor;
}

.header__c-glyph {
  fill: currentColor;
}
```



Our logo doesn't look so great in this mode either. What we can do is set the strokes and fills on this SVG to currentColor so that it will essentially inherit the color of the parent link. We'll do this in our base theme instead of inside the media query, which means that any color mode could pick up this style and apply the correct color.

srcset media conditions

```
<picture>
  <source srcset="dark.jpg"
  media="(prefers-color-scheme: dark)">
  <img src="light.jpg" />
</picture>
```

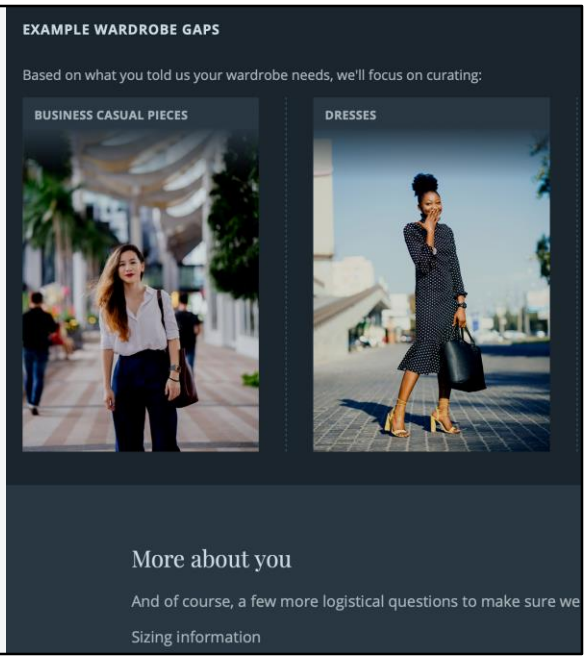
Big bright images might detract from our new dark scheme, so we could choose to swap out imagery using srcset media conditions.

Filters

```
@media (prefers-color-scheme: dark) {
…

img {
  filter:
  brightness(.8)
  contrast(1.2);
}

…
```

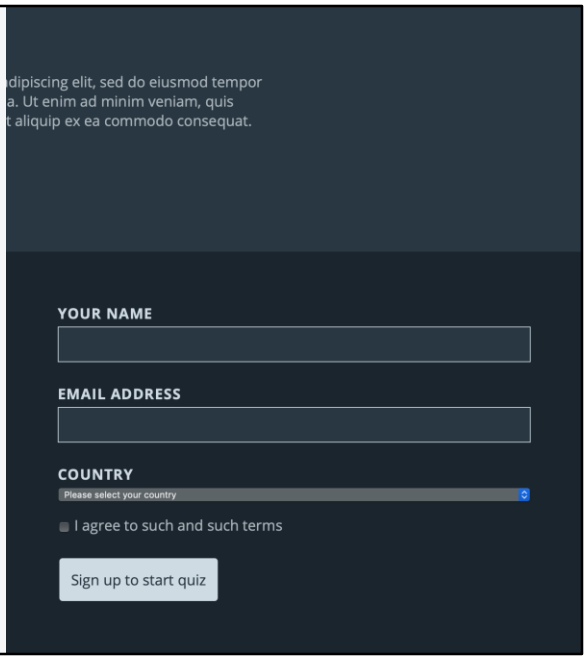


But in our case, the original imagery is important, so we want to tone it down just a bit. We'll place a filter on images to reduce the brightness and increase the contrast a bit, to prevent the colors from getting too muddy. We do this by setting the CSS filter property to brightness(.8) contrast(1.2).

color-scheme

```
/* In base theme */

:root {
  color-scheme:
  light dark;
}
```



Some of our native form controls are still rendered in light-mode versions. To fix this, we can set the color scheme property to “light dark”, which tells the browser which scheme it can render controls or scrollbars in. We set this on the root element so that it will propagate to the entire page, but we could set it on individual elements if we wanted to. This allows for use cases where you might not want to support additional color schemes in part of a document.

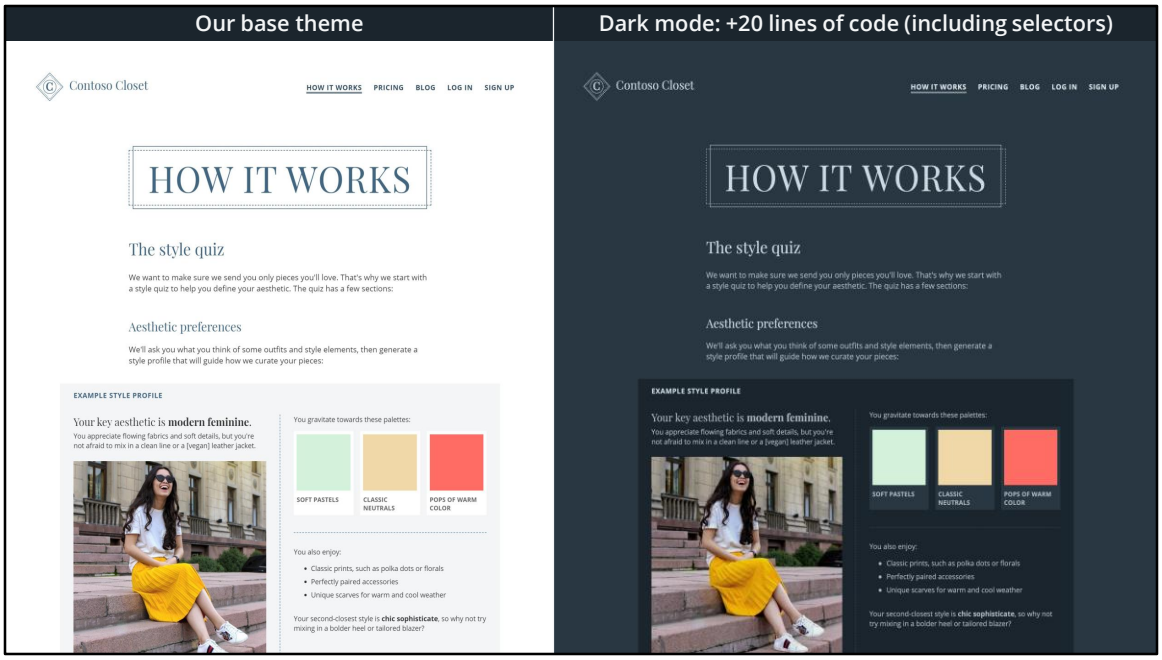
Let's get <meta>

Hints to the rendering engine how a document can be rendered, before any CSS loads:

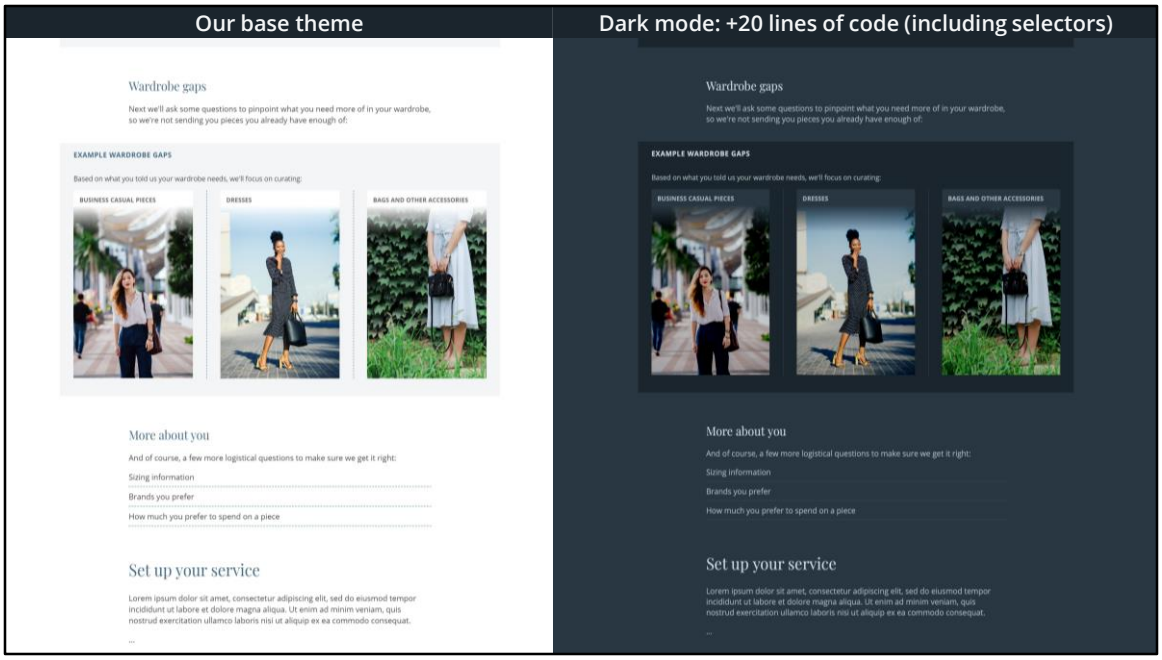
```
<meta name="color-scheme"
      value="light dark">
```

aka.ms/color-scheme-meta

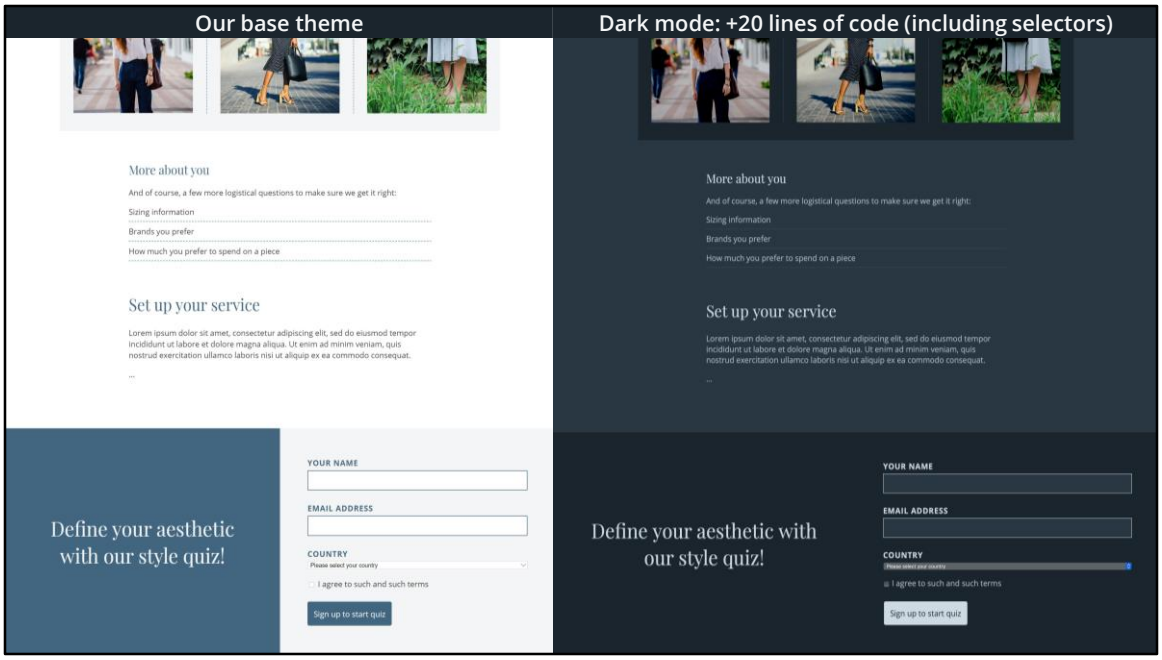
We should also pair this with a meta tag that gives the rendering engine an early hint about how to render a document. This can help prevent a flash of white before our CSS loads. This exact syntax is still being discussed in standards, so you can find the latest specification development at aka.ms/color-scheme-meta



With 20 lines of code, which includes CSS selectors and redefinition of color variables, we've now implemented a complete dark theme on top of our base styles, available for anyone who finds dark mode more restful on the eyes.



With 20 lines of code, which includes CSS selectors and redefinition of color variables, we've now implemented a complete dark theme on top of our base styles, available for anyone who finds dark mode more restful on the eyes.



With x lines of code, which includes CSS selectors and redefinition of color variables, we've now implemented a complete dark theme on top of our base styles, available for anyone who finds dark mode more restful on the eyes.

Additional considerations: a site-level user override?

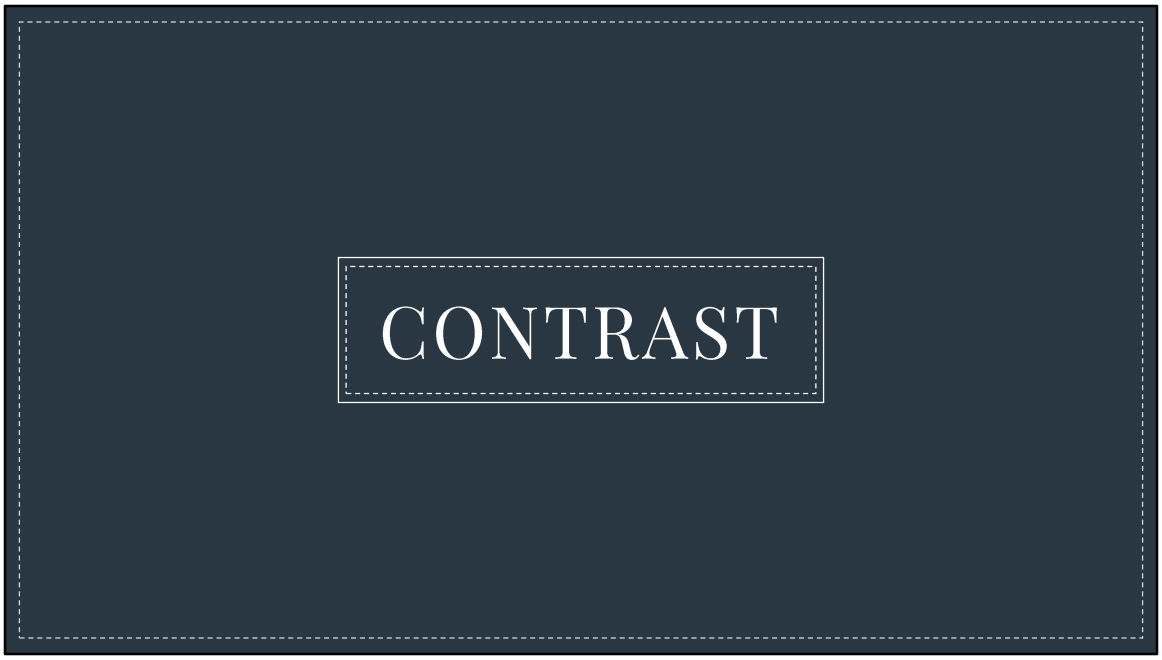
```
@media (prefers-color-scheme: dark) {
  :root:not(.user-theme--light) {
    /* Your dark theme */
  }
}
```

If you'd like to go the extra mile, you might also consider a site-level user override. It's always possible that the user prefers dark color scheme for the rest of their ecosystem, but actually prefers to view your site in a lighter color scheme. For these users, you could provide a color scheme switch that toggles a classname connected to your dark mode styles.

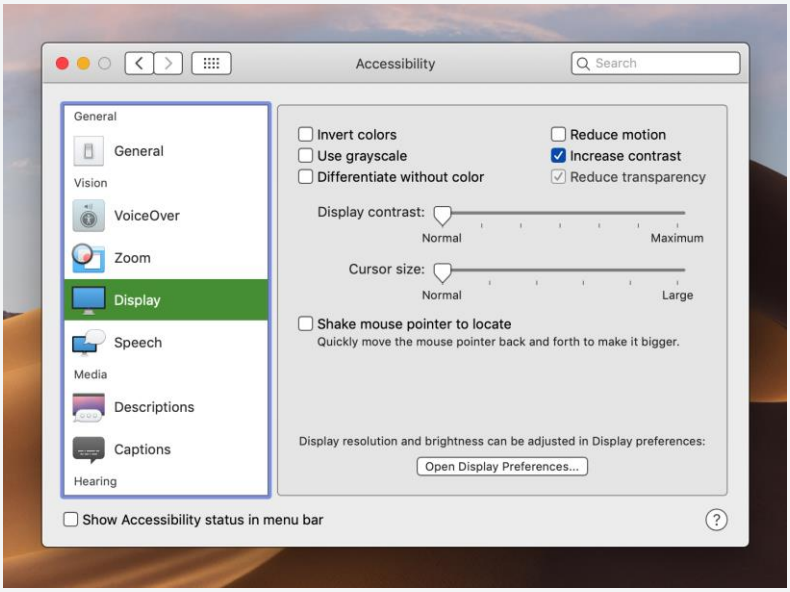
Browser support for color schemes (desktop)

`prefers-color-scheme` MQ	
`color-scheme` property	
`color-scheme <meta>` tag name and values (initial syntax)	

There's good support today for prefers-color-scheme across modern browsers. The color-scheme property and current meta tag syntax are supported in Safari 12.1+.



Now that we've got our dark mode up and running, let's talk about querying for contrast.



This media query integrates with user settings—on the OS or otherwise—as to the contrast between foregrounds and backgrounds in the UI. For example, on MacOS, “increase contrast” in light mode makes text and borders on the operating system darker, to stand out more from the background.

Current (proposed) syntax

```
@media (prefers-contrast: high) {}
```

Values:

```
no-preference | high | low
```

This standard is a little further out than the others, as its syntax is still under discussion. Currently the available values are no-preference, high, and low.

Do those values align with reality?

`reduce` | `no-preference` | `increase` | `high` ?

Numeric values?

But some folks wondered if this really aligned with the way current OS-level contrast settings work. Should we modify the keywords? Change to numeric values? For this reason, we won't dwell too much on this media query today, as the guidance might change.

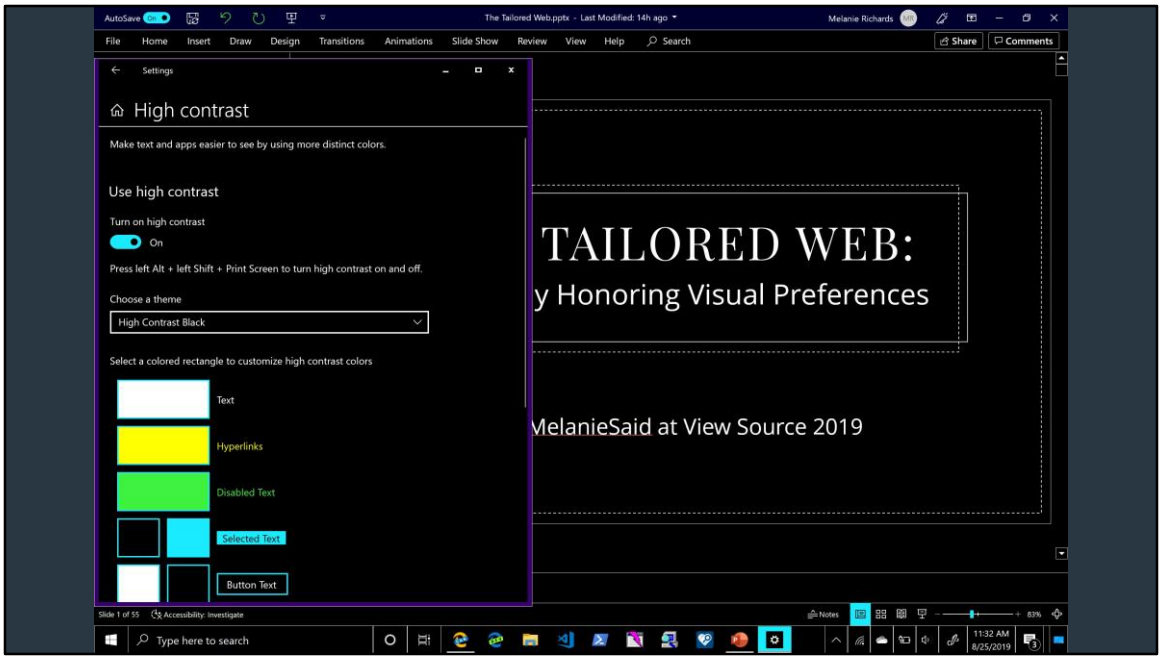
Focus on adjusting

- Text
- Non-decorative icons / imagery
- Backgrounds and decorations on interactive elements
- Other design elements important to understanding relationships

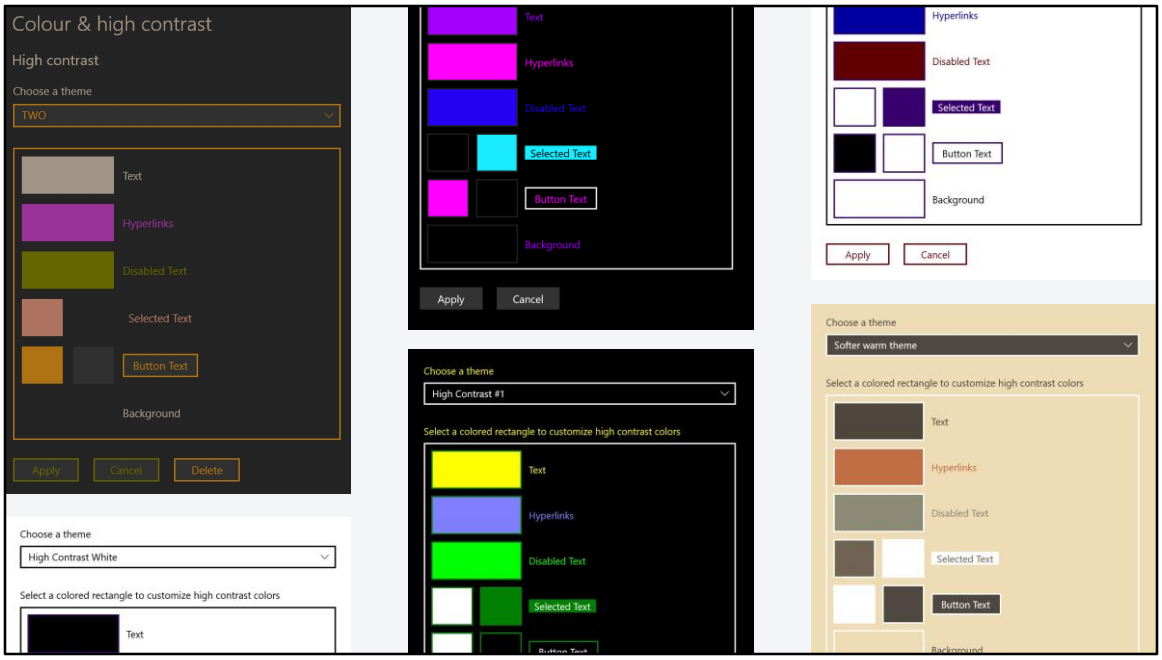
But when it's available in the future, web developers should focus on adjusting text, non-decorative icons or imagery, backgrounds and decorations on interactive elements, and other design elements important to understanding relationships.

FORCED COLORS

For some folks, a slight adjustment to contrast or an art-directed color scheme doesn't quite meet their needs for legibility. Instead, they might prefer or require UI rendered in exactly the colors of their choosing. This is where forced color modes come in.



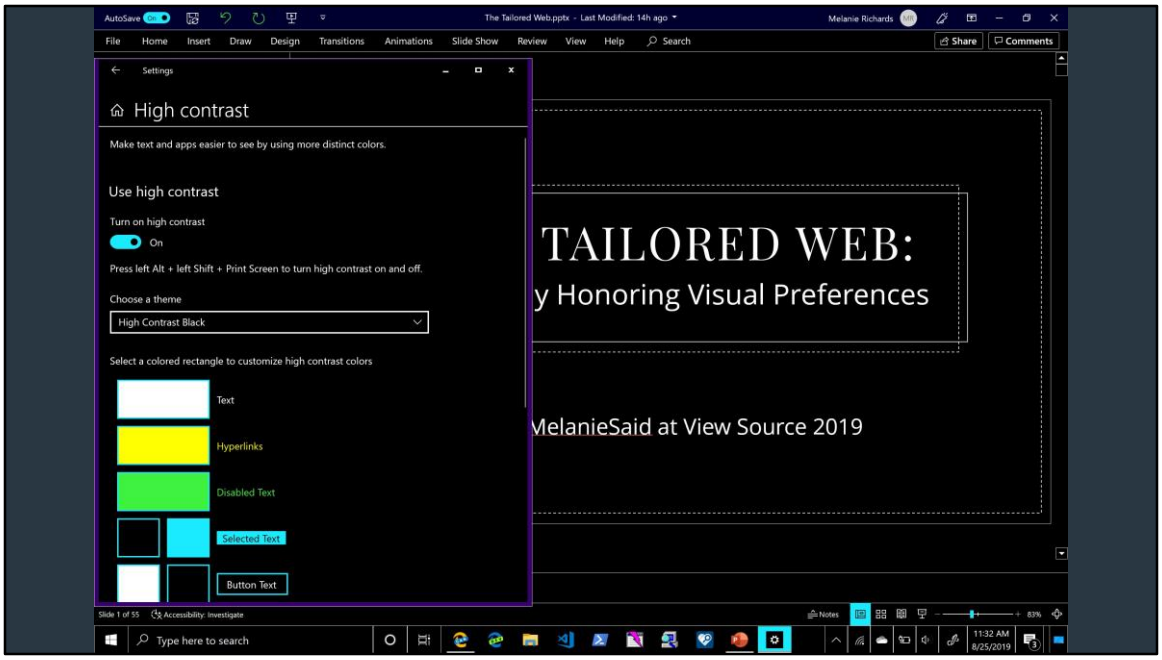
Standards around forced colors are heavily influenced by the Windows High Contrast feature shown earlier, as well as prior art used to style for this mode in IE and Microsoft Edge.



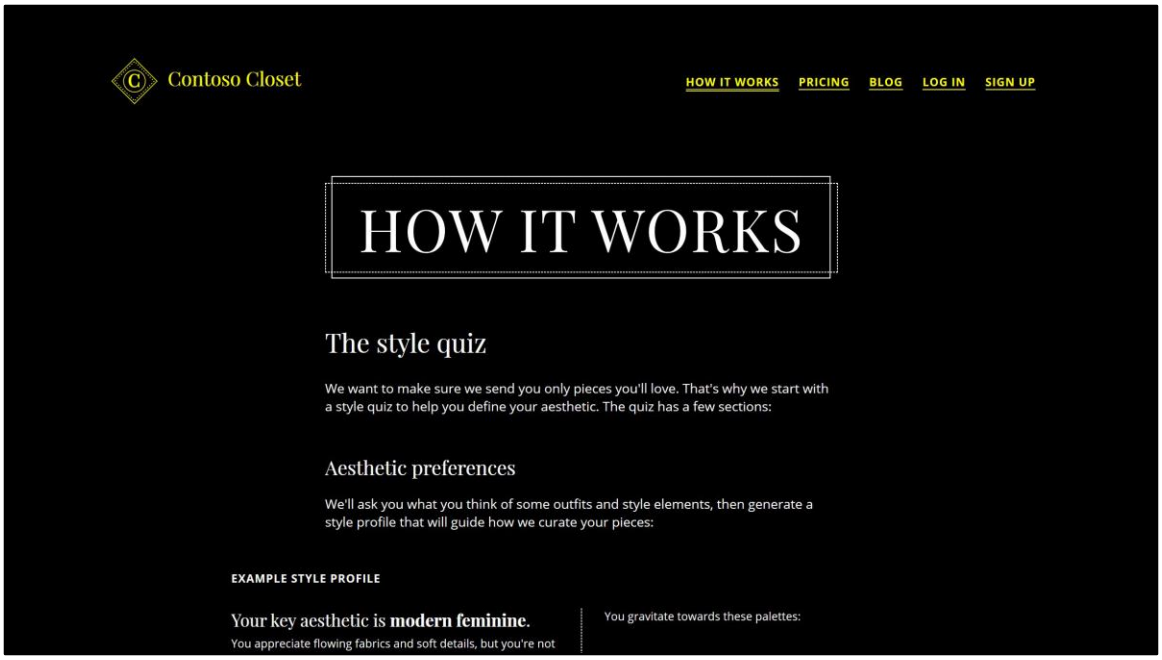
It should be noted that Windows High Contrast is somewhat of a misnomer. While the in-box themes are certainly the most popular and guarantee high contrast, users can adjust these colors to anything they want. This can be helpful if someone needs colors that are a bit more differentiated from each other, or if they actually need UI to render with SOFTER contrast. No matter their reason for using High Contrast, this is far and away one of the most popular Ease of Access features on Windows—last time we checked, used on 4% of active devices. That represents millions of people!

How browsers apply forced colors to web content

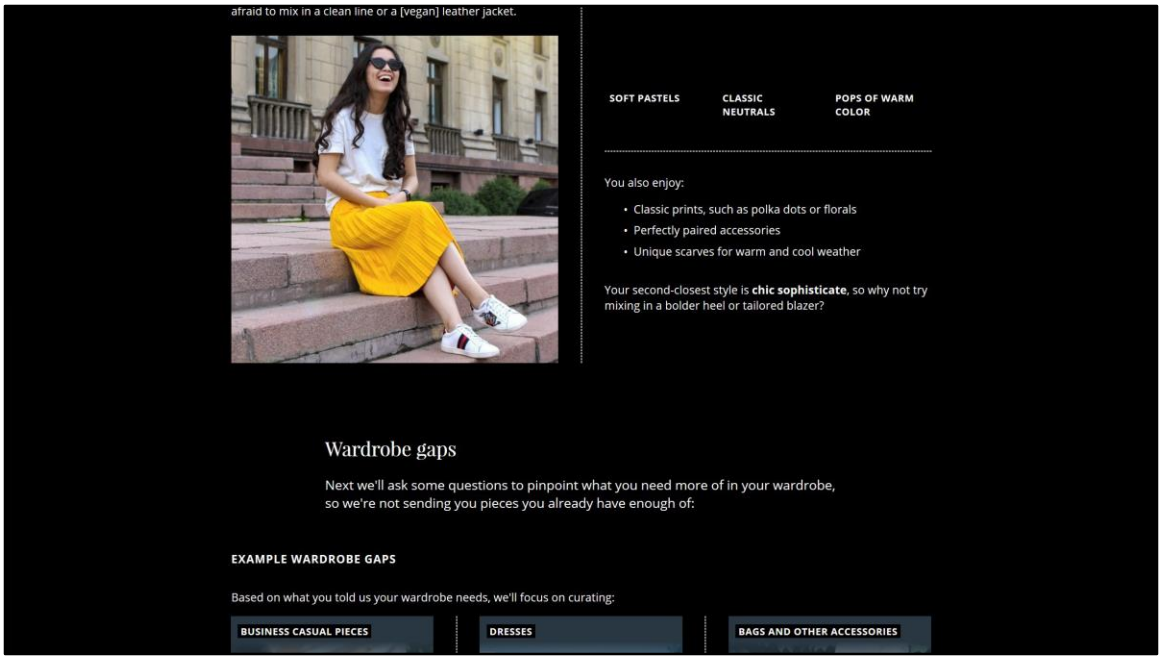
The OS setting may provide the information needed to implement forced color modes, but it's up to individual apps to apply these colors in a semantic fashion to their UI. This means that the browser rendering engine is directly responsible for applying the user's colors to web content. It also means that the browser can provide its own forced color modes separately from the OS; for example, Firefox currently offers some in-browser forced color settings. Let's take a look at our example page in a forced color mode with the new standards.



We've currently set the machine to a High Contrast Black theme…



And without doing any work on the web development end, you can see that our page automatically picked up the colors. The rendering engine has applied these semantically: regular text gets the text color, while links are rendered in their own color from the palette.



These colors are shifted throughout the page…



More about you

And of course, a few more logistical questions to make sure we get it right:

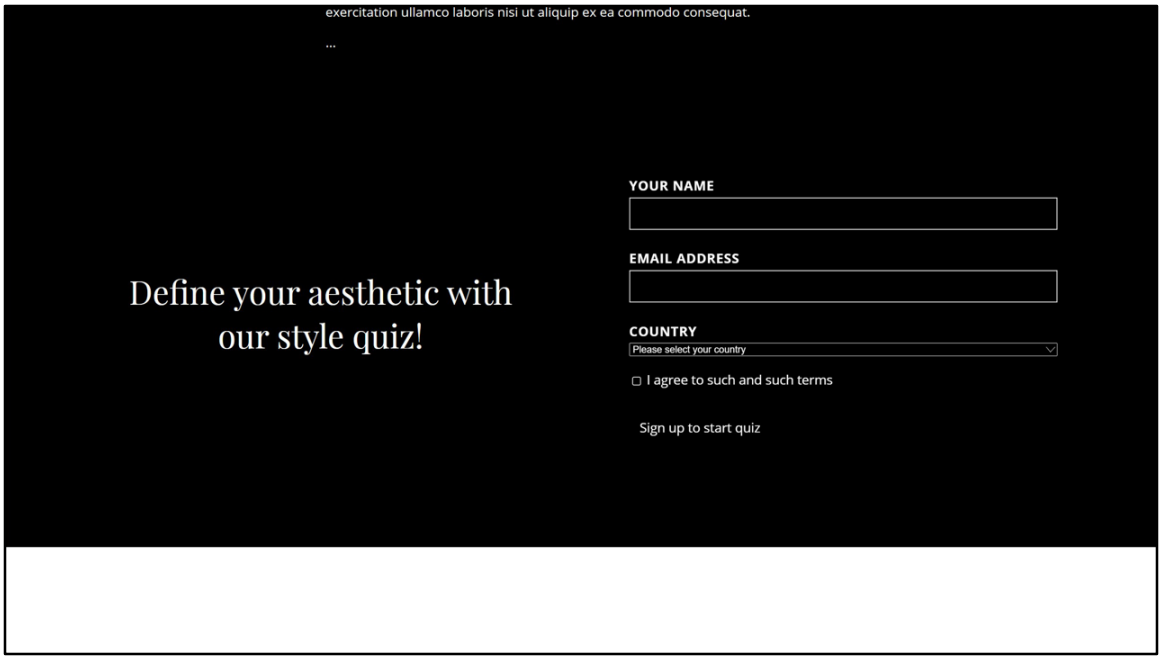
Sizing information

Brands you prefer

How much you prefer to spend on a piece

Set up your service

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud



…including native form controls.

How this works under the hood

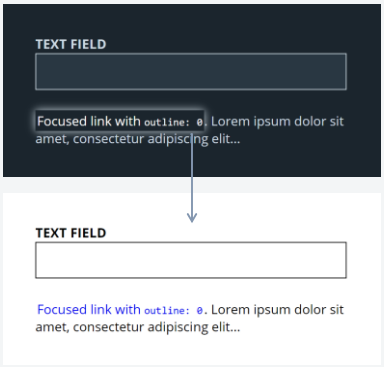
Let’s explore how this works under the hood.

The rendering engine reverts some properties:

Various properties related to color (`border-color`, `background-color`, `etc.`)

`box-shadow`, `text-shadow`

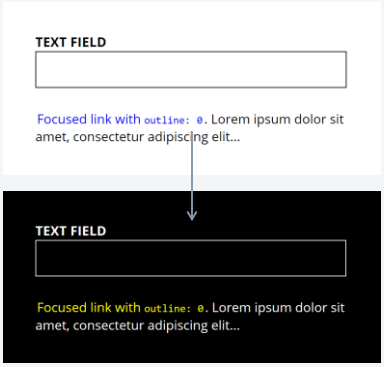
`background-image` (only on user interactive controls, excluding `buttons`)



The rendering engine first reverts the values of some CSS properties. This rolls back styles for properties related to color, as well as properties that could interfere with legibility. For example, box-shadow and text-shadow can make edges harder to discern, or can overlap over other elements.

The rendering engine applies the user's colors:

These styles are specified in one of the browser's "user agent" stylesheets



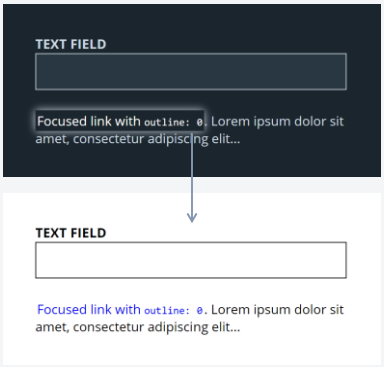
Then, the rendering engine supplies its own user agent stylesheet, where various elements are themed with the user's system colors.

Notes on reverted property values

Various properties related to color (`border-color`, `background-color`, `etc.`)

`box-shadow`, `text-shadow`

`background-image` (only on user interactive controls, excluding `butttons`)



Returning to our reverted properties, what's important to note here as web developers is that if you use a reverted property to denote state, such as box-shadow, you may need to make some adjustments when your content is rendered in a forced color mode.

Notes on reverted property values

Various properties related to color (`border-color`, `background-color`, `etc.`)

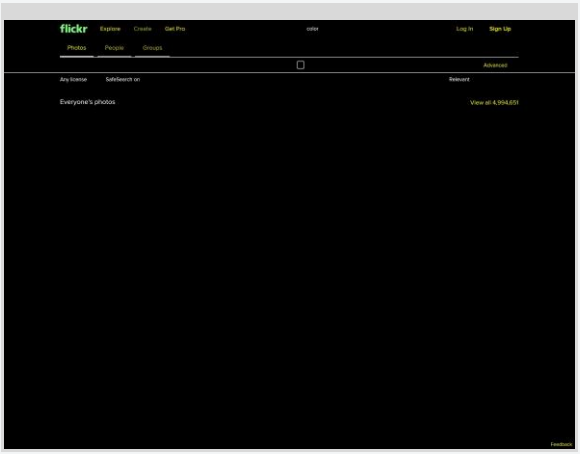
`box-shadow`, `text-shadow`

`background-image` (only on user interactive controls, excluding `butttons`)

Another interesting point to call out is that background-image is only reverted in user interactive controls, excluding buttons.

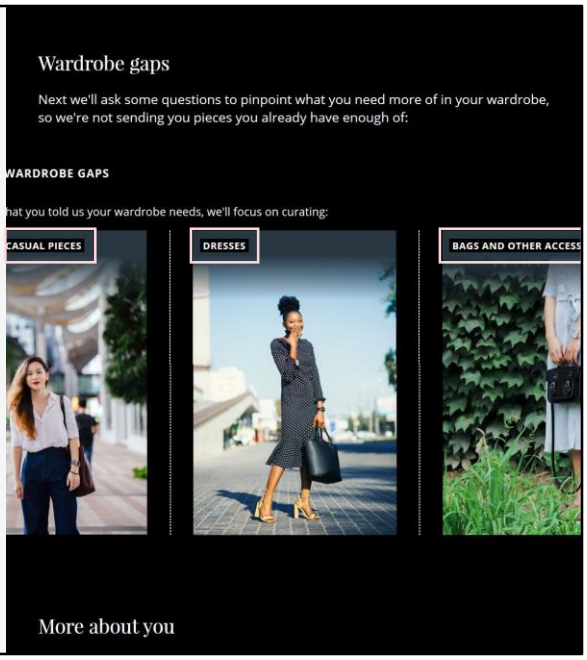
Why not globally revert `background-image`?

Many developers use this property to display "content" images



Why not globally revert background-image? In historical implementations that remove background-images, this causes issues when CSS is used to display content images. Check out this feed of photos, for example. Let the record show, there are no images visible in this screenshot. Web developers have taken this approach for various different reasons—in this case it was a historical performance trick—so we need to make sure sure people can view this content.

Text backplates guarantee legibility

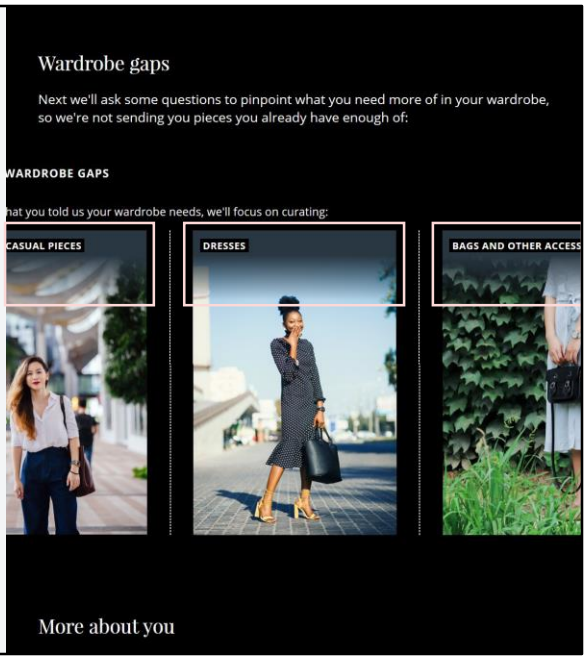


So, how do we show background images and guarantee legibility of text layered on top of these images? The solution we came up with in EdgeHTML, and which is now part of the standard, is to render text backplates behind runs of text in forced color modes. No matter what the text is layered on top of, users will be able to read it in their preferred color scheme.

Future standards may unlock the ability to tweak backplate styling; for example, adding the ability to change the border radius.

Color schemes in forced color modes

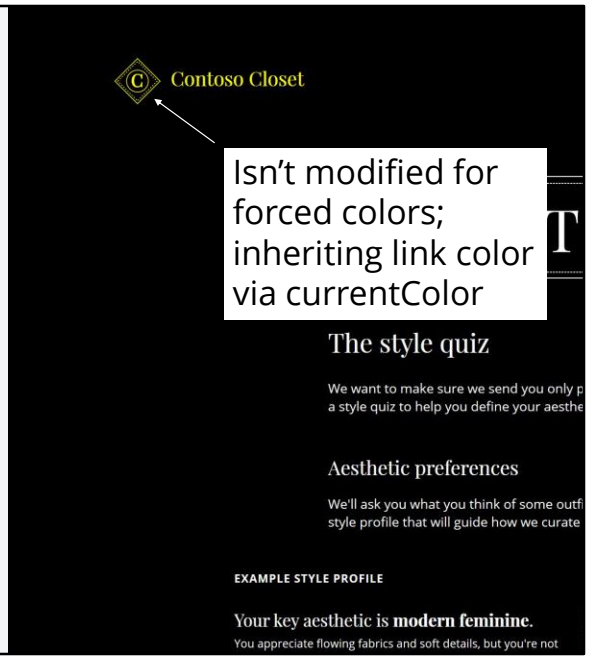
`prefers-color-scheme` evaluates appropriately based on canvas color (page background)



A new detail in standard implementations is that prefers-color-scheme also evaluates properly in forced color modes. The rendering engine uses the Canvas color, or page background color, to determine if the color scheme is light or dark. It then evaluates prefers-color-scheme media queries accordingly. Here, our gradients are rendering in the styles we've set from within prefers-color-scheme: dark.

SVGs are not modified…mostly

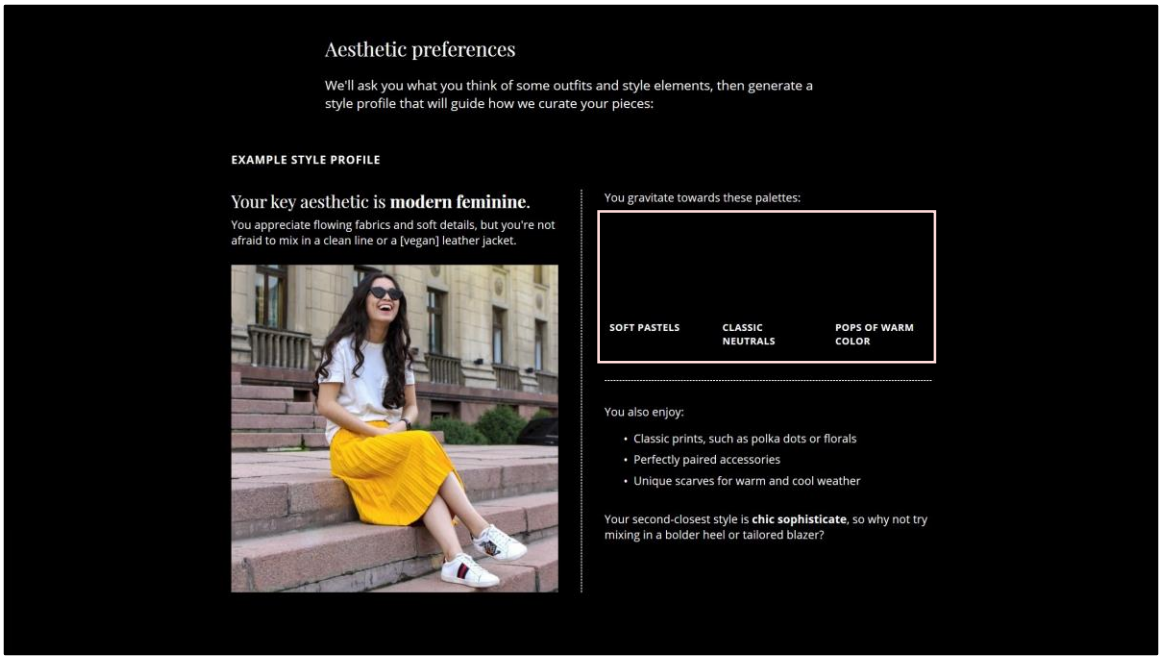
Only SVG `<text>` and `<foreignObject>` have forced colors applied for legibility



SVGs can contain complex content, and so as rendering engines we can't really guess at how these should be rendered. The only parts of SVG that have forced colors applied are <text> elements, and <foreignObject>s (which embed HTML elements within SVG). Funnily enough, this is actually why our logo SVG is picking up the right system link color. We previously set the strokes and fills to currentColor, which tells the logo to inherit the color from the parent link, or in this case, the system link color!

Web devs can apply content-specific knowledge to tailor the forced colors experience.

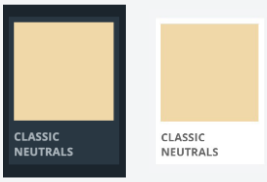
Rendering engines can get most of the way there by applying reasonable defaults semantically to web content. But the people who work on individual sites know the needs of their content best. Web devs can apply content-specific knowledge to tailor the forced colors experience.



Alright so! You may have noticed that when we flipped into a forced color mode, we lost some detail in the palette of this style profile. That's because we used background-color to supply the color swatches. We could switch to using images, or we could manage this component ourselves.

Component requirements

1. Retain the color swatch
2. Prevent swatch rendering on top of any given background color (muddy?)
3. Preserve the user's color scheme for text legibility



Let's review our requirements in forced color modes for this component.

We want to:

1. Retain the color swatch.
2. Prevent the swatch from rendering on top of any old background color the user may have chosen, in case it looks a little muddy. So we've decided we always want to render that on top of either black or white.
3. And preserve the user's color scheme for text legibility.

Prior art: media query syntax

```
@media (-ms-high-contrast: active) {}
```

Values:

`none` (IE and early MS Edge) | `active` | `black-on-white` | `white-on-black`

Let’s start with our media query. In browser versions built before these new standards, we’d query for –ms-high-contrast: active

Standard media query syntax

```
@media (-ms-high-contrast: active),
       (forced-colors: active) {}
```

Values:

```
none | active
```

This has been renamed in web standards for forced-colors, and the value we're querying for is "active". The CSS Working Group landed on this name to future-proof the media query, so that web developers can efficiently style for any OS or browser feature which applies forced colors to web content.

Prior art: opt out of forced colors

```
@media (-ms-high-contrast: active) {
  .prefs__palette {
    -ms-high-contrast-adjust: none;
  }
  …
```

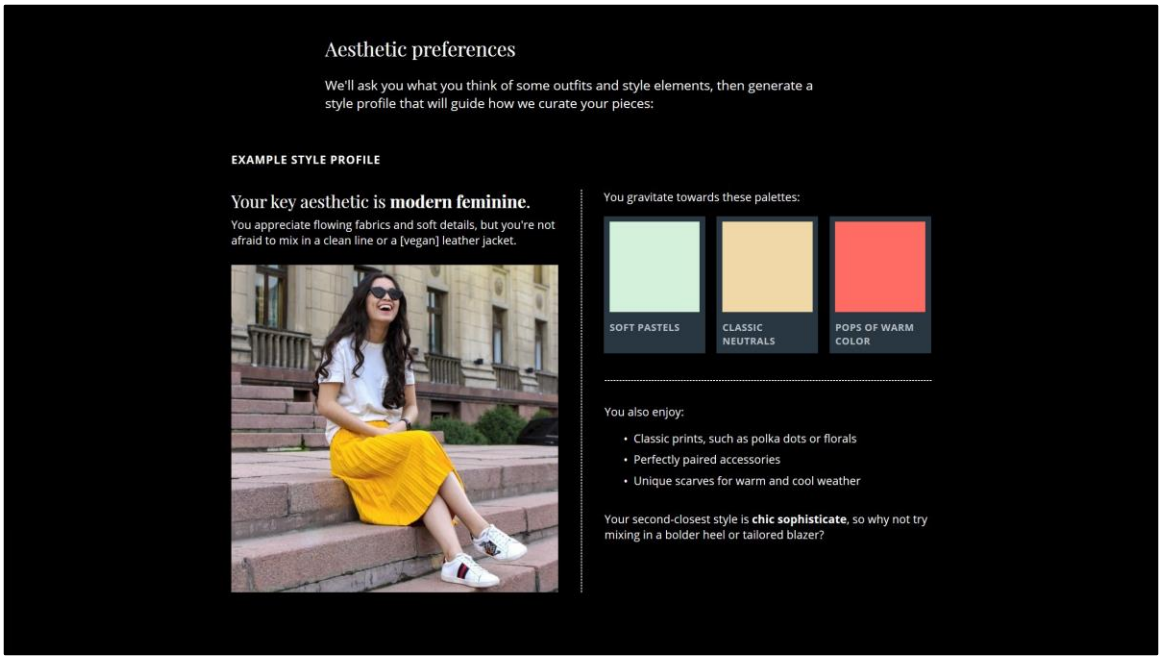
After setting up our media query, we'll want to unset the forced colors from the element. This strategy comes from the –ms-high-contrast-adjust CSS property.

Opt out of forced colors

```
@media (-ms-high-contrast: active),
(forced-colors: active) {

  .prefs__palette {
    -ms-high-contrast-adjust: none;
    forced-color-adjust: none;
  }
```

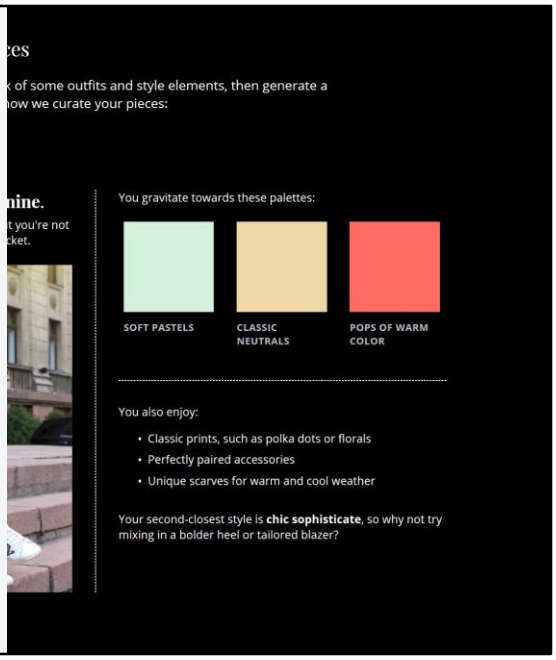
In the new standards, forced-color-adjust set to “none” will remove the colors from the component and its children, including all text backplates.



Our palette now looks as it did in dark mode.

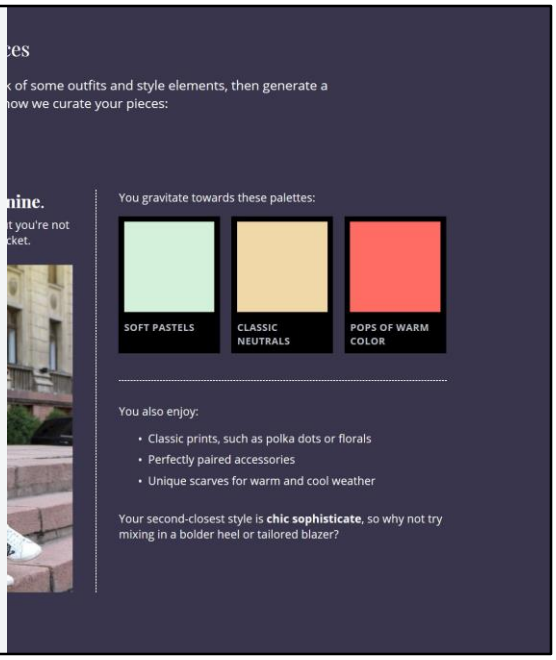
Be careful with static colors

```
.prefs__palette {
  forced-color-adjust: none;
  background-color: #000;
}
```



Next, we want to render the background of this component in black. We're doing this because we don't want the swatch to render on a random color, but we should NOT try and use static colors to match a user's theme. This reason why is less obvious when we have a black theme.

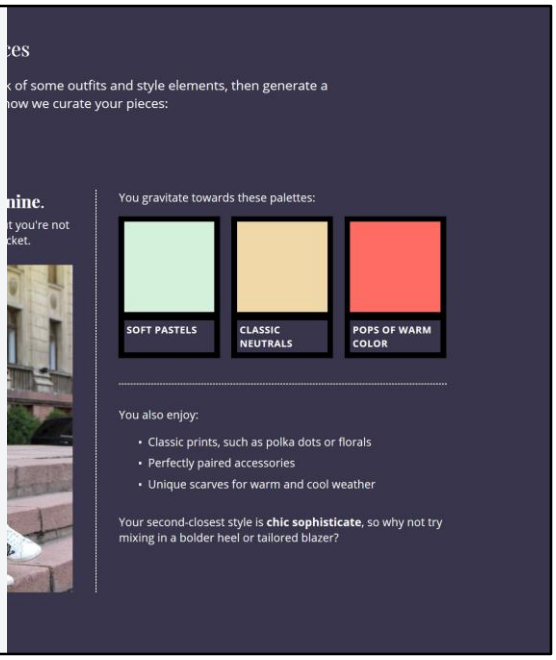
Static color values are unaware of the user scheme



But flip this into, say, a purple user color scheme, and you can see that our static color isn't aware of the user scheme. We want to make sure that the text is in the user's colors, so in order to do that…

Use appropriate system colors

```
.c-label {
  display: block;
  padding: .25em;
  background-color: Canvas;
  color: Text;
}
```



…we're going to use system colors to guarantee legibility. We'll create our own text backplate by setting the background-color to Canvas (page background) and the color to Text.

CSS system colors relevant to Windows High Contrast

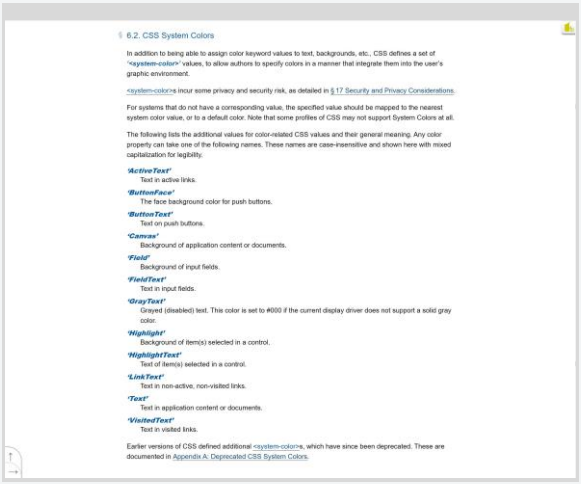
High Contrast Theme Color	CSS System Color
Text	`Text`
Hyperlinks	`LinkText` (replaces `-ms-hotlight`)
Disabled Text	`GrayText`
Selected Text	`HighlightText` (foreground), `Highlight` (background)
Button Text	`ButtonText` (foreground), `ButtonFace` (background)
Background	`Canvas`

There are a few CSS system colors that are relevant when Windows High Contrast is active. Besides Text and Canvas, the refreshed standard list includes LinkText for links, GrayText for disabled text, and additional keyword values for selections and buttons.

System colors are an abstraction that make forced color styling much more efficient. You don't need to manage static colors for each individual visitor who's using a forced color mode. Instead, you can set your styles with these colors just once, and trust that they'll be rendered in whatever colors the user has set.

Full list of system colors on CSS Color Editor's Draft

aka.ms/
system-colors-draft



§ 6.2. CSS System Colors

In addition to being able to assign color keyword values to text, backgrounds, etc., CSS defines a set of '<system-color>' values, to allow authors to specify colors in a manner that integrate them into the user's graphic environment.

<system-color>s incur some privacy and security risk, as detailed in § 17 Security and Privacy Considerations.

For systems that do not have a corresponding value, the specified value should be mapped to the nearest system color value, or to a default color. Note that some profiles of CSS may not support System Colors at all.

The following lists the additional values for color-related CSS values and their general meaning. Any color property can take one of the following names. These names are case-insensitive and shown here with mixed capitalization for legibility.

'ActiveText'
Text in active links.

'ButtonFace'
The face background color for push buttons.

'ButtonText'
Text on push buttons.

'Canvas'
Background of application content or documents.

'Field'
Background of input fields.

'FieldText'
Text in input fields.

'GrayText'
Grayed (disabled) text. This color is set to #000 if the current display driver does not support a solid gray color.

'Highlight'
Background of item(s) selected in a control.

'HighlightText'
Text of item(s) selected in a control.

'LinkText'
Text in non-active, non-visited links.

'Text'
Text in application content or documents.

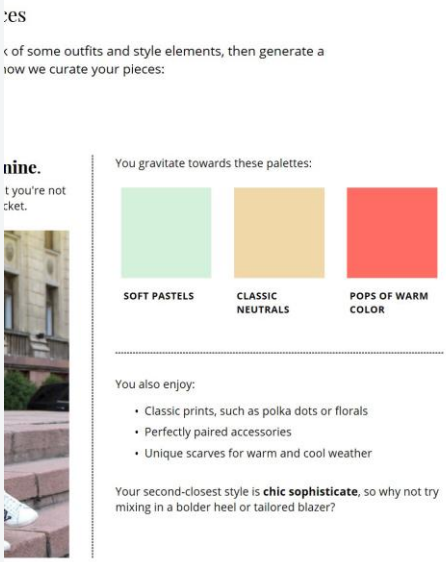
'VisitedText'
Text in visited links.

Earlier versions of CSS defined additional <system-color>s, which have since been deprecated. These are documented in Appendix A: Deprecated CSS System Colors.

You can find the full list of current standardized system colors on the CSS Color draft specification, using the short link aka.ms/system-colors-draft. You'll notice that there's more colors on this list—not every operating system will be able to support every standardized system color, so it's important to test whenever you use these values.

Optimize for dark / light forced colors

```
@media (forced-colors: active)
and (prefers-color-scheme: light){
  .prefs__palette {
    background: #fff;
  }
}
```

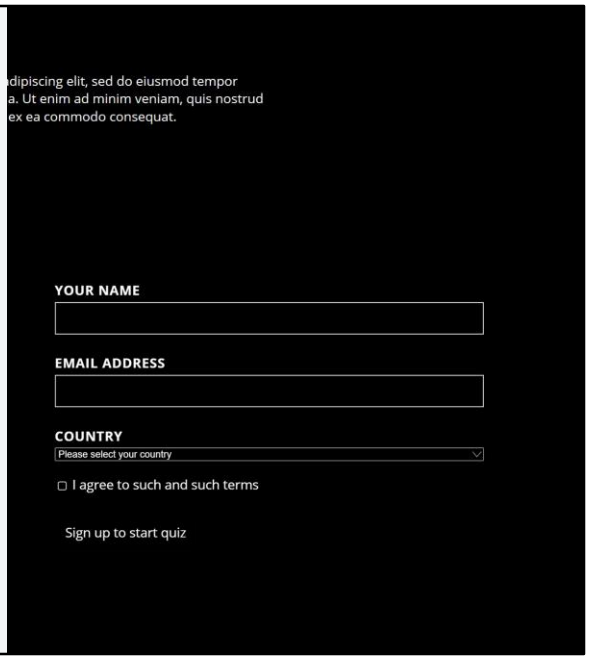


Let's go back to our component. When the user is in a light forced color scheme, they still get the correct colors for the text, because we used system colors. But our static black background for the component is a bit much, and we want it to always be white in lighter schemes. So to adjust this, we'll take advantage of the fact that prefers-color-scheme evaluates intelligently in forced color modes. We'll query for forced-colors: active and prefers-color-scheme: light and update our styles accordingly.

Instead of removing styles…

```
/* In base theme */

button {
  border: 0;
}
```

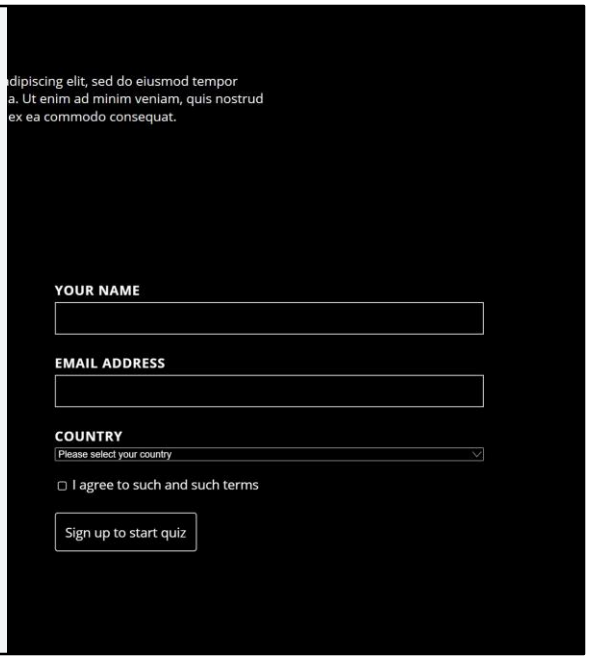


Elsewhere on our page, we should be careful about using shorthands to remove styles. For example, in this forced color scheme, the button and page background both happen to be black. So setting border: 0 in our base theme means that the user can't see the bounds of this button when in this forced color mode.

…consider transparency

```
/* In base theme */

button {
  border: 1px solid
  transparent;
}
```

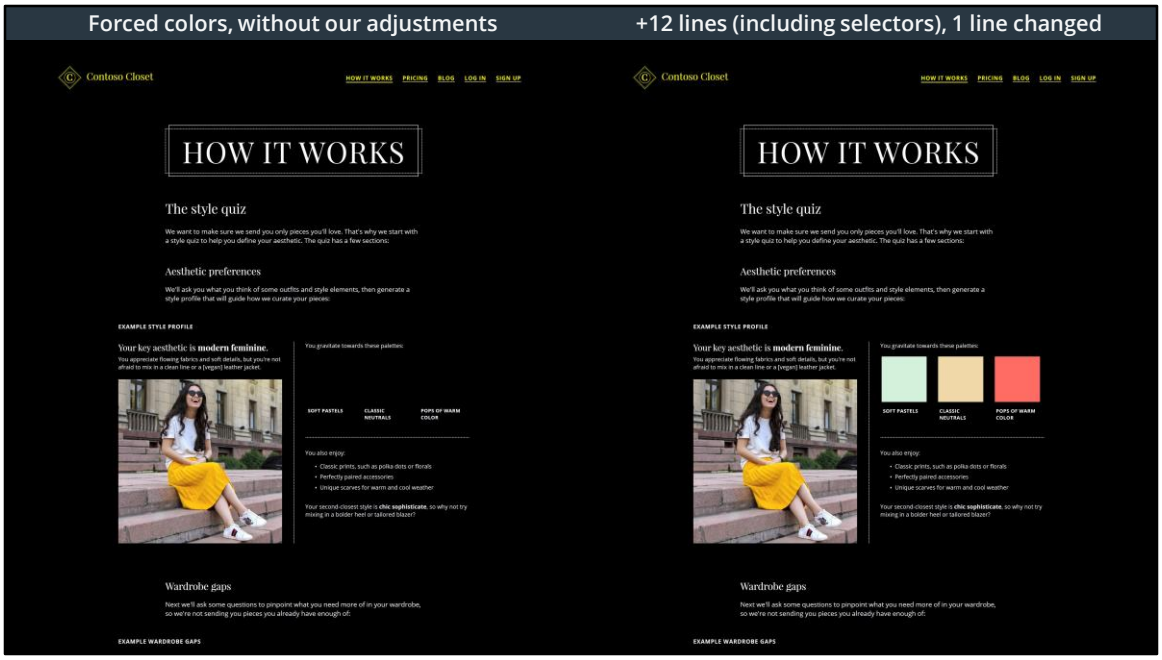


Change this to border: 1px solid transparent, and the border is rendered in forced color modes while invisible in the base theme.

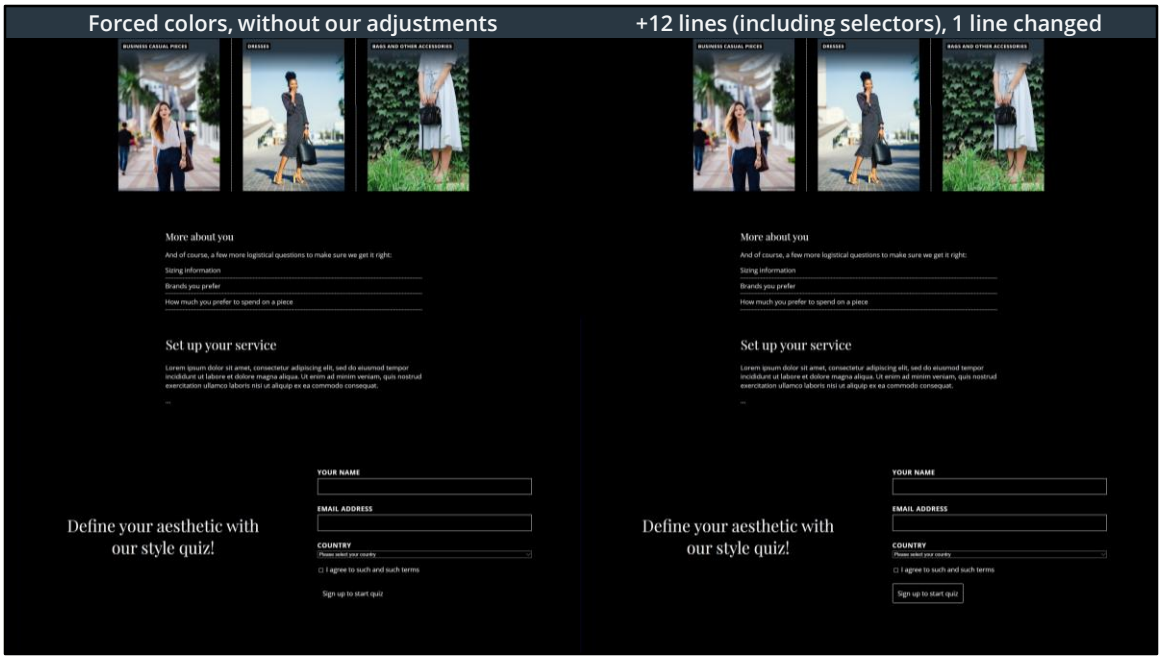
Additional considerations: styling based on ARIA

```
[role="link"] {
  color: LinkText;
}
```

In additional considerations, browser rendering engines aren't really in the business of styling defaults based on ARIA roles and properties. The expectation is that if you use ARIA to build out a custom control, you are responsible for maintaining keyboard accessibility and styling as well. This applies in forced color modes as it does in default contexts.



Unless you're creating a lot of your own semantic components with various states, it's likely that tailoring your site for forced colors involves fairly minimal work. I just threw a lot of information at you, but in order to optimize this design, we added 12 lines of code (including selectors and excluding the old prefixed syntax).



We also changed one more line of CSS to account for our button element.

Browser support for forced colors (desktop)

Vendor-prefixed styling

New standards
(designed in 2019)

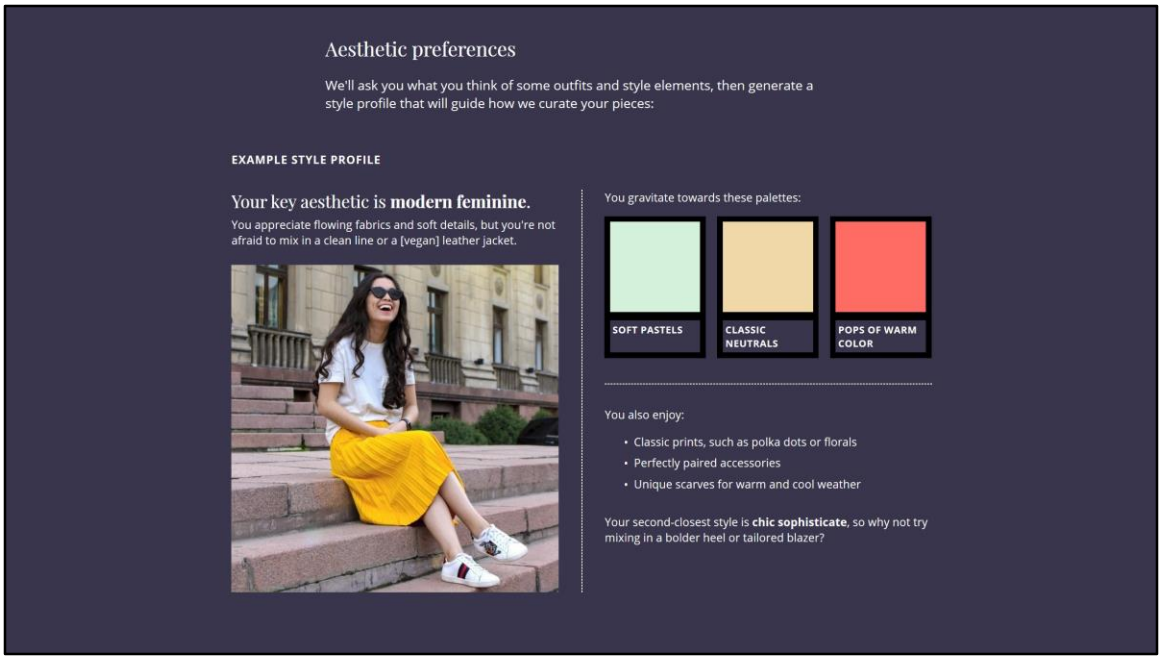
In development behind a flag
in the Chromium project

As far as browser support for forced-color modes goes, the vendor-prefixed styling techniques are supported in IE and in-market versions of Microsoft Edge, running on EdgeHTML. The new standards are pretty new—developed in 2019—so support is in development behind a flag in the Chromium project. should be ready for folks to start playing with soon in a couple browsers.

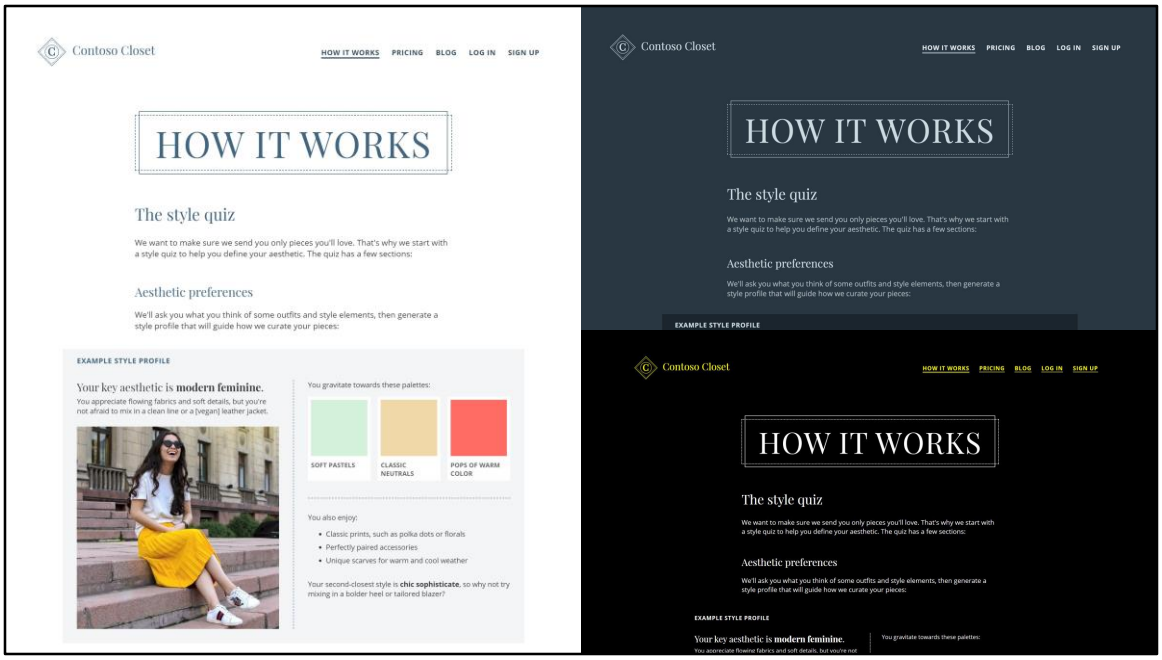
Testing without a Windows machine?

Virtual Machines:
aka.ms/web-vms

If you don’t have a Windows machine, you can use a free Virtual Machine from aka.ms/web-vms to try out forced colors on Windows. In the future, cross-platform browsers could also explore providing dev tools to “spoof” forced color modes, so that you don’t need a particular OS or to push your whole machine into forced colors. Show of hands, who would be interested in such a feature?



I hope you do get the chance to try out your content in forced color modes. Because while browser rendering engines can get pretty far with reasonable defaults, you are the expert on your content and can work on top of the platform to deliver a truly well-tailored experience.



By using new hints on color and contrast preferences, you can meet your users and their highly-individualized needs where they are. Whether someone finds that dark mode reduces eye strain, or needs a very specific color palette to stay productive, you have the power and the content-specific knowledge to give them truly delightful experience.

With these new media queries and properties in hand, let’s design a more inclusive and accessible web together.