# Building a switch component


- Overview
  - Custom properties
    - Track
    - Thumb
    - Reduced motion
  - Markup
- Layouts
  - .gui-switch
  - Track
  - Thumb
- Styles
  - Touch interaction styles
  - Track
  - Thumb
    - Interaction
  - Thumb position
  - Vertical
  - (RTL) right-to-left
- States
  - Checked
  - Disabled
  - Indeterminate
  - Hover
- JavaScript
  - Draggable thumbs
  - touch-action
  - Pixel value style utilities
  - dragging

- dragEnd
- determineChecked()
- Extra thoughts
- Conclusion
  - Community remixes
  - Resources


---

In this post I want to share thinking on a way to build switch components. Try the demo.

> Demo

If you prefer video, here's a YouTube version of this post:

https://www.youtube.com/watch?v=_KqccADghcA

# Overview

A switch functions similar to a checkbox but explicitly represents boolean on and off states.

This demo uses `<input type="checkbox" role="switch">` for the majority of its functionality, which has the advantage of not needing CSS or JavaScript to be fully functional and accessible. Loading CSS brings support for right-to-left languages, verticality, animation and more. Loading JavaScript makes the switch draggable and tangible.

# Custom properties

The following variables represent the various parts of the switch and their options. As the top-level class, `.gui-switch` contains custom properties used throughout the component children, and entry points for centralized customization.

## Track

The length (`--track-size`), padding, and two colors:

```
.gui-switch {
  --track-size: calc(var(--thumb-size) * 2);
  --track-padding: 2px;
```

```
  --track-inactive: hsl(80 0% 80%);
  --track-active: hsl(80 60% 45%);

  --track-color-inactive: var(--track-inactive);
  --track-color-active: var(--track-active);

  @media (prefers-color-scheme: dark) {
    --track-inactive: hsl(80 0% 35%);
    --track-active: hsl(80 60% 60%);
  }
}
```

## Thumb

The size, background color, and interaction highlight colors:

```
.gui-switch {
  --thumb-size: 2rem;
  --thumb: hsl(0 0% 100%);
  --thumb-highlight: hsl(0 0% 0% / 25%);

  --thumb-color: var(--thumb);
  --thumb-color-highlight: var(--thumb-highlight);

  @media (prefers-color-scheme: dark) {
    --thumb: hsl(0 0% 5%);
    --thumb-highlight: hsl(0 0% 100% / 25%);
  }
}
```

## Reduced motion

To add a clear alias and reduce repetition, a reduced motion preference user media query can be put into a custom property with the PostCSS plugin based on this draft spec in Media Queries 5:

```
@custom-media --motionOK (prefers-reduced-motion: no-preference);
```

# Markup

I chose to wrap my `<input type="checkbox" role="switch">` element with a `<label>`, bundling their relationship to avoid checkbox and label association ambiguity, while giving the user the ability to interact with the label to toggle the input.

*A natural, unstyled label and checkbox.*

```
<label for="switch" class="gui-switch">
  Label text
  <input type="checkbox" role="switch" id="switch">
</label>
```

`<input type="checkbox">` comes prebuilt with an API and state. The browser manages the `checked` property and input events such as `oninput` and `onchanged`.

# Layouts

Flexbox, grid, and custom properties are critical in maintaining the styles of this component. They centralize values, give names to otherwise ambiguous calculations or areas, and enable a small custom property API for easy component customizations.

## `.gui-switch`

The top-level layout for the switch is flexbox. The class `.gui-switch` contains the private and public custom properties the children use to compute their layouts.

*Flexbox DevTools overlaying a horizontal label and switch, showing their layout distribution of space.*

```
.gui-switch {
  display: flex;
  align-items: center;
  gap: 2ch;
  justify-content: space-between;
}
```

Extending and modifying the flexbox layout is like changing any flexbox layout. For example, to put labels above or below a switch, or to change the `flex-direction`:

*Flexbox DevTools overlaying a vertical label and switch.*

```
<label for="light-switch" class="gui-switch" style="flex-direction: column">
  Default
  <input type="checkbox" role="switch" id="light-switch">
</label>
```

# Track

The checkbox input is styled as a switch track by removing its normal `appearance: checkbox` and supplying its own size instead:

*Grid DevTools overlaying the switch track, showing the named grid track areas with the name '*track*'.*

```
.gui-switch > input {
  appearance: none;

  inline-size: var(--track-size);
  block-size: var(--thumb-size);
  padding: var(--track-padding);

  flex-shrink: 0;
  display: grid;
  align-items: center;
  grid: [track] 1fr / [track] 1fr;
}
```

The track also creates a one by one single cell grid track area for a thumb to claim.

# Thumb

The style `appearance: none` also removes the visual checkmark supplied by the browser. This component uses a pseudo-element and the `:checked` pseudo-class on the input to replace this visual indicator.

The thumb is a pseudo-element child attached to the `input[type="checkbox"]` and stacks on top of the track instead of below it by claiming the grid area `track`:

*DevTools showing the pseudo-element thumb as positioned inside a CSS grid.*

```
.gui-switch > input::before {
  content: "";
  grid-area: track;
  inline-size: var(--thumb-size);
  block-size: var(--thumb-size);
}
```

Not all inputs can have pseudo-elements, learn more here.

# Styles

Custom properties enable a versatile switch component that adapts to color schemes, right-to-left languages and motion preferences.

*A side by side comparison of the light and dark theme for the switch and its states.*

# Touch interaction styles

On mobile, browsers add tap highlights and text selection features to labels and inputs. These negatively affected the style and visual interaction feedback that this switch needed. With a few lines of CSS I can remove those effects and add my own `cursor: pointer` style:

```
.gui-switch {
  cursor: pointer;
  user-select: none;
  -webkit-tap-highlight-color: transparent;
}
```

It's not always advisable to remove those styles, as they can be valuable visual interaction feedback. Be sure to provide custom alternatives if you remove them.

# Track

This element's styles are mostly about its shape and color, which it accesses from the parent `.gui-switch` via the cascade.

*The switch variants with custom track sizes and colors.*

```
.gui-switch > input {
  appearance: none;
  border: none;
  outline-offset: 5px;
  box-sizing: content-box;

  padding: var(--track-padding);
  background: var(--track-color-inactive);
  inline-size: var(--track-size);
  block-size: var(--thumb-size);
  border-radius: var(--track-size);
}
```

A wide variety of customization options for the switch track come from four custom properties. `border: none` is added since `appearance: none` doesn't remove the borders from the checkbox on all browsers.

# Thumb

The thumb element is already on the right `track` but needs circle styles:

```
.gui-switch > input::before {
  background: var(--thumb-color);
  border-radius: 50%;
}
```

*DevTools shown highlighting the circle thumb pseudo-element.*

## Interaction

Use custom properties to prepare for interactions which will show hover highlights and thumb position changes. The user's preference is also checked before transitioning the motion or hover highlight styles.

```
.gui-switch > input::before {
  box-shadow: 0 0 0 var(--highlight-size) var(--thumb-color-highlight);

  @media (--motionOK) { & {
    transition:
      transform var(--thumb-transition-duration) ease,
```

```
      box-shadow .25s ease;
  }}
}
```

# Thumb position

Custom properties provide a single source mechanism for positioning the thumb in the track. At our disposal are the track and thumb sizes which we'll use in calculations to keep the thumb properly offset and between within the track: `0%` and `100%`.

The `input` element owns the position variable `--thumb-position`, and the thumb pseudo element uses it as a `translateX` position:

```
.gui-switch > input {
  --thumb-position: 0%;
}

.gui-switch > input::before {
  transform: translateX(var(--thumb-position));
}
```

We're now free to change `--thumb-position` from CSS and the pseudo-classes provided on checkbox elements. Since we conditionally set `transition: transform var(--thumb-transition-duration) ease` earlier on this element, these changes may animate when changed:

```
/* positioned at the end of the track: track length - 100% (thumb width) */
.gui-switch > input:checked {
  --thumb-position: calc(var(--track-size) - 100%);
}

/* positioned in the center of the track: half the track - half the thumb */
.gui-switch > input:indeterminate {
  --thumb-position: calc(
    (var(--track-size) / 2) - (var(--thumb-size) / 2)
  );
}
```

I thought this decoupled orchestration worked out well. The thumb element is only concerned with one style, a `translateX` position. The input can manage all the complexity and calculations.

This reminds me of reactive state stores, where the store centralizes all the computing, actions and side effects (parent track element). Subscribers (thumb pseudo element) can observe dynamic values without worrying about owning or duplicating logic. One store, lots of subscribers all with the correct values.

# Vertical

Supporting was done with a modifier class `-vertical` which adds a rotation with CSS transforms to the `input` element.

A 3D rotated element does not change the overall height of the component though, which can throw off block layout. Account for this using the `--track-size` and `--track-padding` variables. Calculate the minimum amount of space required for a vertical button to flow in layout as expected:

```
.gui-switch.-vertical {
  min-block-size: calc(var(--track-size) + calc(var(--track-padding) * 2));

  & > input {
    transform: rotate(-90deg);
  }
}
```

# (RTL) right-to-left

A CSS friend, Elad Schecter, and I prototyped together a slide out side menu using CSS transforms that handled right-to-left languages by flipping a single variable. We did this because there are no logical property transforms in CSS, and there may never be. Elad had the great idea of using a custom property value to invert percentages, to allow single location management of our own custom logic for logical transforms. I used this same technique in this switch and I think it worked out great:

```
.gui-switch {
  --isLTR: 1;

  &:dir(rtl) {
    --isLTR: -1;
  }
}
```

A custom property called `--isLTR` initially holds a value of `1`, meaning it's `true` since our layout is left-to-right by default. Then, using the CSS pseudo class `:dir()`, the value is set to `-1` when the component is within a right-to-left layout.

Put `--isLTR` into action by using it within a `calc()` inside of a transform:

```
.gui-switch.-vertical > input {
  transform: rotate(-90deg);
  transform: rotate(calc(90deg * var(--isLTR) * -1));
}
```

Now the rotation of the vertical switch accounts for the opposite side position required by the right-to-left layout.

The `translateX` transforms on the thumb pseudo-element also need updated to account for the opposite side requirement:

```
.gui-switch > input:checked {
  --thumb-position: calc(var(--track-size) - 100%);
  --thumb-position: calc((var(--track-size) - 100%) * var(--isLTR));
}

.gui-switch > input:indeterminate {
  --thumb-position: calc(
    (var(--track-size) / 2) - (var(--thumb-size) / 2)
  );
  --thumb-position: calc(
   ((var(--track-size) / 2) - (var(--thumb-size) / 2))
    * var(--isLTR)
  );
}
```

While this approach won't work to solve all needs regarding a concept like logical CSS transforms, it does offer some DRY principles for many use cases.

# States

Using the built in `input[type="checkbox"]` wouldn't be complete without handling the various states it can be in: `:checked`, `:disabled`, `:indeterminate` and `:hover`. `:focus` was intentionally left alone, with an adjustment only made to its offset; the focus ring looked great on Firefox and Safari:

*A screenshot of focus ring focused on a switch in Firefox and Safari.*

Chromium will be catching up to the rounded focus ring styles seen in Firefox and Safari. Follow along here.

# Checked

```
<label for="switch-checked" class="gui-switch">
  Default
  <input type="checkbox" role="switch" id="switch-checked" checked="true">
</label>
```

This state represents the `on` state. In this state, the input "track" background is set to the active color and the thumb position is set to "the end".

```
.gui-switch > input:checked {
  background: var(--track-color-active);
  --thumb-position: calc((var(--track-size) - 100%) * var(--isLTR));
}
```

# Disabled

```
<label for="switch-disabled" class="gui-switch">
  Default
  <input type="checkbox" role="switch" id="switch-disabled" disabled="true">
</label>
```

A `:disabled` button not only visually looks different, but also should make the element immutable.Interaction immutability is free from the browser, but the visual states need styles due to the use of `appearance: none`.

```
.gui-switch > input:disabled {
  cursor: not-allowed;
  --thumb-color: transparent;

  &::before {
    cursor: not-allowed;
    box-shadow: inset 0 0 0 2px hsl(0 0% 100% / 50%);
```

```
    @media (prefers-color-scheme: dark) { & {
      box-shadow: inset 0 0 0 2px hsl(0 0% 0% / 50%);
    }}
  }
}
```

*The dark styled switch in disabled, checked, and unchecked states.*

This state is tricky since it needs dark and light themes with both disabled and checked states. I stylistically chose minimal styles for these states to ease the maintenance burden of the combinations of styles.

# Indeterminate

An often forgotten state is `:indeterminate`, where a checkbox is neither checked or unchecked. This is a fun state, it's inviting and unassuming. A good reminder that boolean states can have sneaky in between states.

It is tricky to set a checkbox to indeterminate, only JavaScript can set it:

```
<label for="switch-indeterminate" class="gui-switch">
  Indeterminate
  <input type="checkbox" role="switch" id="switch-indeterminate">
  <script>document.getElementById('switch-indeterminate').indeterminate = true</script>
</label>
```

*The indeterminate state which has the track thumb in the middle, to indicate undecided.*

Since the state, to me, is unassuming and inviting, it felt appropriate to put the switch thumb position in the middle:

```
.gui-switch > input:indeterminate {
  --thumb-position: calc(
    calc(calc(var(--track-size) / 2) - calc(var(--thumb-size) / 2))
    * var(--isLTR)
  );
}
```

# Hover

Hover interactions should provide visual support for connected UI and also provide direction towards interactive UI. This switch highlights the thumb with a semi-transparent ring when the label or the input are hovered. This hover animation then provides direction towards the interactive thumb element.

The "highlight" effect is done with `box-shadow`. On hover, of an non-disabled input, increase the size of `--highlight-size`. If the user is OK with motion, we transition the `box-shadow` and see it grow, if they're not ok with motion, the highlight appears instantly:

```
.gui-switch > input::before {
  box-shadow: 0 0 0 var(--highlight-size) var(--thumb-color-highlight);

  @media (--motionOK) { & {
    transition:
      transform var(--thumb-transition-duration) ease,
      box-shadow .25s ease;
  }}
}

.gui-switch > input:not(:disabled):hover::before {
  --highlight-size: .5rem;
}
```

# JavaScript

This script is optional

To me, a switch interface can feel uncanny in its attempt to emulate a physical interface, especially this kind with a circle inside a track. iOS got this right with their switch, you can drag them side to side, and it's very satisfying to have the option. Conversely, a UI element can feel inactive if a drag gesture is attempted and nothing happens.

# Draggable thumbs

The thumb pseudo-element receives its position from the `.gui-switch > input` scoped `var(--thumb-position)`, JavaScript can supply an inline style value on the input to dynamically update the thumb position making it appear to follow the pointer gesture. When the pointer is released, remove the inline styles and determine if the drag was closer to off or on by using the custom property `--thumb-position`. This is the backbone of the solution; pointer events conditionally tracking pointer positions to modify CSS custom properties.

Since the component was already 100% functional before this script is showing up, it does take quite a bit of work to maintain the existing behavior, like clicking a label to toggle the input. Our JavaScript shouldn't add features at the expense of existing features.

## `touch-action`

Dragging is a gesture, a custom one, which makes it a great candidate for `touch-action` benefits. In the case of this switch, a horizontal gesture should be handled by our script, or a vertical gesture captured for the vertical switch variant. With `touch-action` we can tell the browser what gestures to handle on this element, so a script can handle a gesture without competition.

The following CSS instructs the browser that when a pointer gesture starts from within this switch track, handle vertical gestures, do nothing with horizontal ones:

```
.gui-switch > input {
  touch-action: pan-y;
}
```

The desired result is a horizontal gesture that doesn't also pan or scroll the page. A pointer can vertically scroll start from within the input and scroll the page, but horizontal ones are custom handled.

# Pixel value style utilities

On setup and during drag, various computed number values will need to be grabbed from elements. The following JavaScript functions return computed pixel values given a CSS property. It's used in the setup script like this `getStyle(checkbox, 'padding-left')`.

```
const getStyle = (element, prop) => {
  return parseInt(window.getComputedStyle(element).getPropertyValue(prop));
}

const getPseudoStyle = (element, prop) => {
  return parseInt(window.getComputedStyle(element, ':before').getPropertyValue(prop));
}

export {
  getStyle,
  getPseudoStyle,
}
```

Notice how `window.getComputedStyle()` accepts a second argument, a target pseudo element. Pretty neat that JavaScript can read so many values from elements, even from pseudo elements.

These functions use `parseInt()` which is making an assumption that you are querying a value that returns a pixel value. This means that you cannot use these functions with `getStyle(element, "display")`, for example.

## `dragging`

This is a core moment for the drag logic and there are a few things with noting from the function event handler:

```
const dragging = event => {
  if (!state.activethumb) return

  let {thumbsize, bounds, padding} = switches.get(state.activethumb.parentElement)
  let directionality = getStyle(state.activethumb, '--isLTR')

  let track = (directionality === -1)
    ? (state.activethumb.clientWidth * -1) + thumbsize + padding
    : 0

  let pos = Math.round(event.offsetX - thumbsize / 2)

  if (pos < bounds.lower) pos = 0
  if (pos > bounds.upper) pos = bounds.upper

  state.activethumb.style.setProperty('--thumb-position', `${track + pos}px`)
}
```

The script hero is `state.activethumb`, the little circle this script is positioning along with a pointer. The `switches` object is a `Map()` where the keys are `.gui-switch`'s and the values are cached bounds and sizes that keep the script efficient. Right-to-left is handled using the same custom property that CSS is `--isLTR`, and is able to use it to invert logic and continue supporting RTL. The `event.offsetX` is valuable as well, as it contains a delta value useful for positioning the thumb.

```
state.activethumb.style.setProperty('--thumb-position', `${track + pos}px`)
```

This final line of CSS sets the custom property used by the thumb element. This value assignment would otherwise transition over time, but a previous pointer event has temporarily set `--thumb-transition-duration` to `0s`, removing what would have been a sluggish interaction.

## `dragEnd`

In order for the user to be allowed to drag far outside the switch and let go, a global window event needed registered:

```
window.addEventListener('pointerup', event => {
  if (!state.activethumb) return

  dragEnd(event)
})
```

I think it's very important that a user has freedom to drag loosely and have the interface be smart enough to account for it. It didn't take much to handle it with this switch, but it did need careful consideration during the development process.

```
const dragEnd = event => {
  if (!state.activethumb) return

  state.activethumb.checked = determineChecked()

  if (state.activethumb.indeterminate)
    state.activethumb.indeterminate = false

  state.activethumb.style.removeProperty('--thumb-transition-duration')
  state.activethumb.style.removeProperty('--thumb-position')
  state.activethumb.removeEventListener('pointermove', dragging)
  state.activethumb = null

  padRelease()
}
```

Interaction with the element has completed, time to set the input checked property and remove all the gesture events. The checkbox is changed with `state.activethumb.checked = determineChecked()`.

## `determineChecked()`

This function, called by `dragEnd`, determines where the thumb current lies within the bounds of its track and returns true if it is equal to or over halfway along the track:

```
const determineChecked = () => {
  let {bounds} = switches.get(state.activethumb.parentElement)

  let curpos =
    Math.abs(
      parseInt(
        state.activethumb.style.getPropertyValue('--thumb-position')))

  if (!curpos) {
    curpos = state.activethumb.checked
      ? bounds.lower
      : bounds.upper
  }

  return curpos >= bounds.middle
}
```

# Extra thoughts

The drag gesture incurred a bit of code debt due to the initial HTML structure chosen, mostly notably wrapping the input in a label. The label, being a parent element, would receive click interactions after the input. At the end of the `dragEnd` event, you may have noticed `padRelease()` as an odd sounding function.

```
const padRelease = () => {
  state.recentlyDragged = true

  setTimeout(_ => {
    state.recentlyDragged = false
  }, 300)
}
```

This is to account for the label getting this later click, as it would uncheck, or check, the interaction a user performed.

If I was to do this again, I *might* consider adjusting DOM with JavaScript during the UX upgrade, as to create an element that handles label clicks itself and doesn't fight with built-in behavior.

This kind of JavaScript is my least favorite to write, I don't want to manage conditional event bubbling:

```
const preventBubbles = event => {
  if (state.recentlyDragged)
    event.preventDefault() && event.stopPropagation()
}
```

# Conclusion

This teeny switch component ended up being the most work of all GUI Challenges so far! Now that you know how I did it, how would you? □□

Let's diversify our approaches and learn all the ways to build on the web. Create a demo, tweet me links, and I'll add it to the community remixes section below!

# Community remixes

- @KonstantinRouda with a custom element: demo and code.
- @jhvanderschee with a button: Codepen.

# Resources

Find the `.gui-switch` source code on GitHub.

---

□□□ #3
□ □□□ □□□ 2026-10-31 21:41:31 UTC
□ cocobook-sync □□□ 2026-11-04 17:36:41 UTC