More Vue + Visual Design Study + Visual Design Libraries

Recitation 7, 6.1040, 10-23-2025

Amber Horvath

More Vue

Vue Tutorial

https://vuejs.org/tutorial

Vue Tutorial Highlights

- Rendering content dynamically
  - Reactive Rendering, step 2 of tutorial
- Binding values to HTML elements
  - Attribute Bindings, step 3 of tutorial
- Event Handling
  - Event Listeners, step 4 of tutorial
- Conditional Rendering
  - Step 5 of tutorial
- Working with Parent-Child Components
  - Components, step 11 of tutorial
- Sending Data from Parent to Child
  - Props, step 12 of tutorial
- Sending Data from Child to Parent
  - Step 13 of tutorial

Declarative Rendering

"…using a template syntax that extends HTML, we can describe how the HTML should look based on JavaScript state. When the **state changes**, the **HTML updates automatically**."

```
import { ref } from 'vue'

const message = ref('Hello World!')

console.log(message.value) // "Hello World!"
message.value = 'Changed'
```

https://vuejs.org/tutorial/#step-2

Declarative Rendering - `reactive`

```
import { reactive } from 'vue'

const counter = reactive({
  count: 0
})

console.log(counter.count) // 0
counter.count++
```

`reactive` **used for objects,** `ref` **for primitives**

App.vue +

```vue
<script setup>
import { ref, reactive } from 'vue'

// component logic
// declare some reactive state here.
const counter = reactive({
  count: 0 // note this is an object
})

const message = ref('Hello World!')
</script>

<template>
   <h1>{{ message }}</h1>
   <p>Count is: {{ counter.count }}</p>
</template>
```

Show Error

Auto Save

PREVIEW

Hello World!

Count is: 0

Attribute Bindings - `v-bind`

Binding values to HTML elements

Allows you to reference dynamic, runtime-defined values in your HTML

Variable name defined elsewhere in code

```
<div v-bind:id="dynamicId"></div>
```

```
<div :id="dynamicId"></div>
```

Shorthand for `v-bind` is just using a colon before the attribute you want to bind to a variable

App.vue +

```
<script setup>
import { ref } from 'vue'

const titleClass = ref('title')
</script>

<template>
  <h1 >Make me red</h1> <!-- add dynamic class binding here -->
</template>

<style>
.title {
  color: red;
}
</style>
```

Show Error

Auto Save

PREVIEW

Make me red

Event Listeners - v-on (or @)

```
<button v-on:click="increment">{{ count }}</button>
```

Can replace “v-on:” with @click

```
<script setup>
import { ref } from 'vue'

const count = ref(0)

function increment() {
  // update component state
  count.value++
}
</script>
```

App.vue +

```vue
<template>
<div :style="{ backgroundColor: currentColor }"> <!-- :style is shorthand for setting style -->
    <p>Current Color: {{ currentColor }}</p>
    <button
      @click="generateColor"
    >
      New Color
    </button>
  </div>
</template>

<script setup lang="ts">
import { ref } from 'vue'

const currentColor = ref('rgb(0, 0, 0)') // note the ref is a string

function generateColor() {
  const r = Math.floor(Math.random() * 256)
  const g = Math.floor(Math.random() * 256)
  const b = Math.floor(Math.random() * 256)
  currentColor.value = `rgb(${r}, ${g}, ${b})`
}
</script>
```

Show Error

Auto Save

PREVIEW

Current Color: rgb(235, 41, 68)

New Color

Conditional Rendering - `v-if` and `v-else`

```
<h1 v-if="awesome">Vue is awesome!</h1>
```

```
<h1 v-else>Hmmmmm…</h1>
```

Just like in code, `v-if` and `v-else` can logically follow from one another.

Reference variables to determine which of the connected blocks will render

App.vue +

```vue
<script setup>
import { ref } from 'vue'

const awesome = ref(true)

function toggle() {
  // ... how can we conditionally render this?
}
</script>

<template>
  <button @click="toggle">Toggle</button>
  <h1>Vue is awesome!</h1>
  <h1>Oh no 😢</h1>
</template>
```

Show Error

Auto Save

PREVIEW

Toggle

Vue is awesome!

Oh no 😢

Working with Multiple Components

```
<ParentComp />
 <ChildComp />
</ParentComp>
```

Just like regular HTML in the DOM, follows a hierarchical and nested tree structure.

App.vue ChildComp.vue × Comp.vue × +

```
1  <script setup>
2  import ChildComp from './ChildComp.vue'
3  import Comp from './Comp.vue'
4  </script>
5
6  <template>
7    <!-- render child component -->
8    <Co
9  </template>
```

Show Error

Auto Save

(8:2) Element is missing end tag.

PREVIEW

<

Sending Data from Parent to Child with Props

Parent.vue

```
<ChildComp :msg="greeting" />
```

Child.vue

```
<script setup>
const props = defineProps({
  msg: String
})
</script>
```

“Props” are a special type of data that the parent defines and sends to the child to access. Parent renders the child and passes the data.

Sending Data from Child to Parent with Emits

Parent.vue

```
<script setup>
  const childMsg = ref('No child msg yet’)
</script>
…
<ChildComp
   @response=“(msg) => childMsg = msg“
/>
```

Child.vue

```
<script setup>
// declare emitted events
const emit = defineEmits(['response'])

// emit with argument
emit('response', 'hello from child')
</script>
```

Child elements can define emits which can be called programmatically to send data back to the parent component.

App.vue ChildComp.vue × +

```
1  <script setup>
2  import { ref } from 'vue'
3  import ChildComp from './ChildComp.vue'
4
5  const childMsg = ref('No child msg yet')
6  </script>
7
8  <template>
9    <ChildComp @response="(msg) => childMsg = msg"/>
10   <p>{{ childMsg }}</p>
11 </template>
```

Show Error

Auto Save

PREVIEW

Child component

Click me

hello from child

Questions?

Visual Design Study

What is a Visual Design Study?

A visual design study explores how visual choices—like color, layout, typography, and imagery—affect how people **see, understand, and feel** about what they're looking at.

Why do one?

The goal is to train your *designer's eye.* It's a way of putting into context the core visual design elements we just discussed by seeing what works and what doesn't.

Visual Design Study of Color

Conducting a Visual Design Study - What You'll Gain

1. **Visual Literacy** – Learn to see like a designer by analyzing how typography, color, and layout work together.

2. **Design Vocabulary -** Build language to describe what you find effective ("high contrast palette," "tall x-height," "strong alignment," etc.).

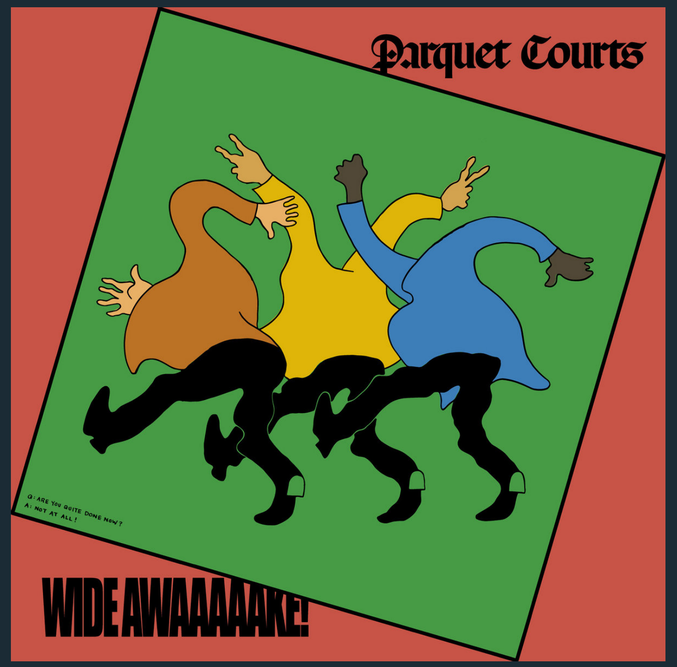
3. **Creative Reference Bank** - Collect visual inspiration to draw from in your own work.

4. **Intentional Design Choices** - Rather than designing by instinct, you'll be able to justify why you picked certain colors or typefaces.

5. **Pattern Recognition** - Spot recurring design strategies across different media and styles.

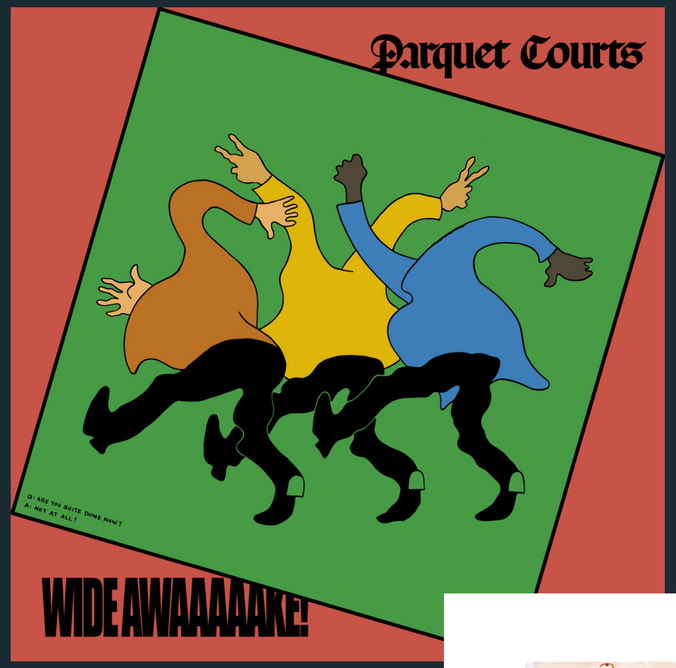
How to do a Visual Design Study

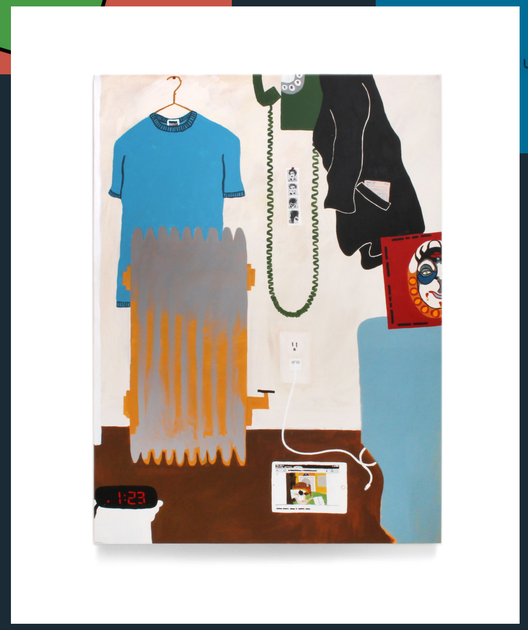
1. **Collect inspirational materials** — e.g., from posters, games, music, or any other visual media or material

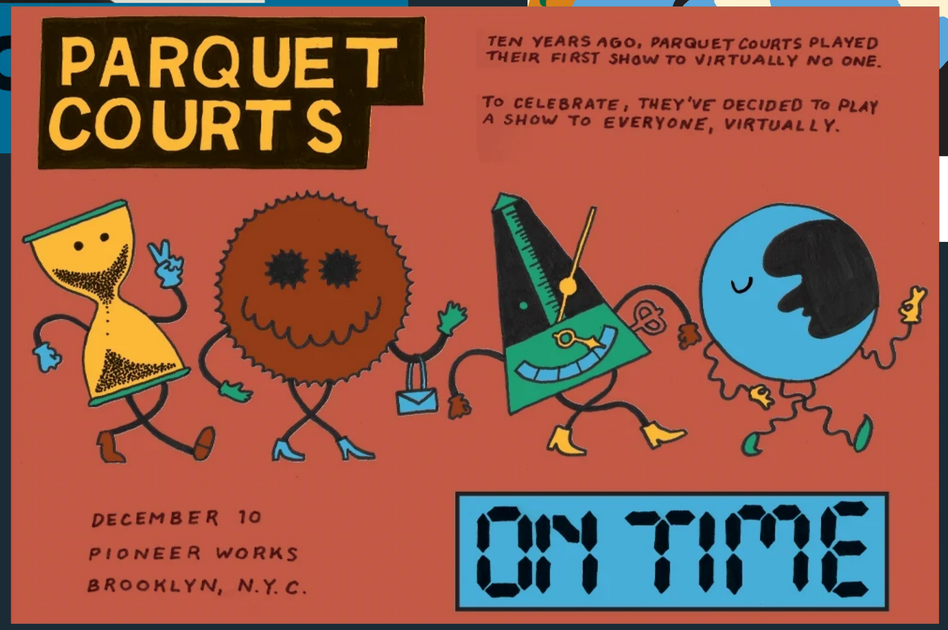


https://a-savage.com/portfolio

2. Organize into a collage

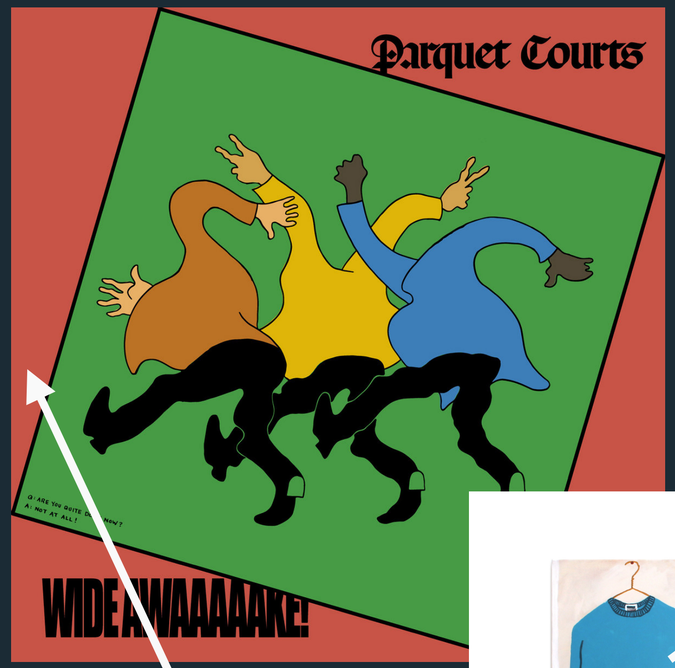




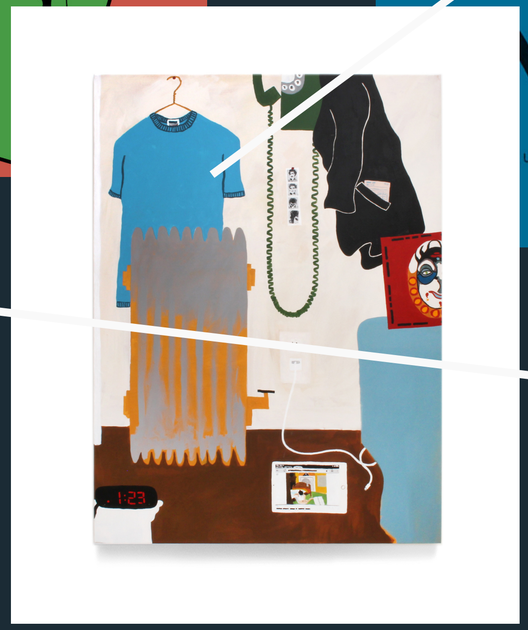


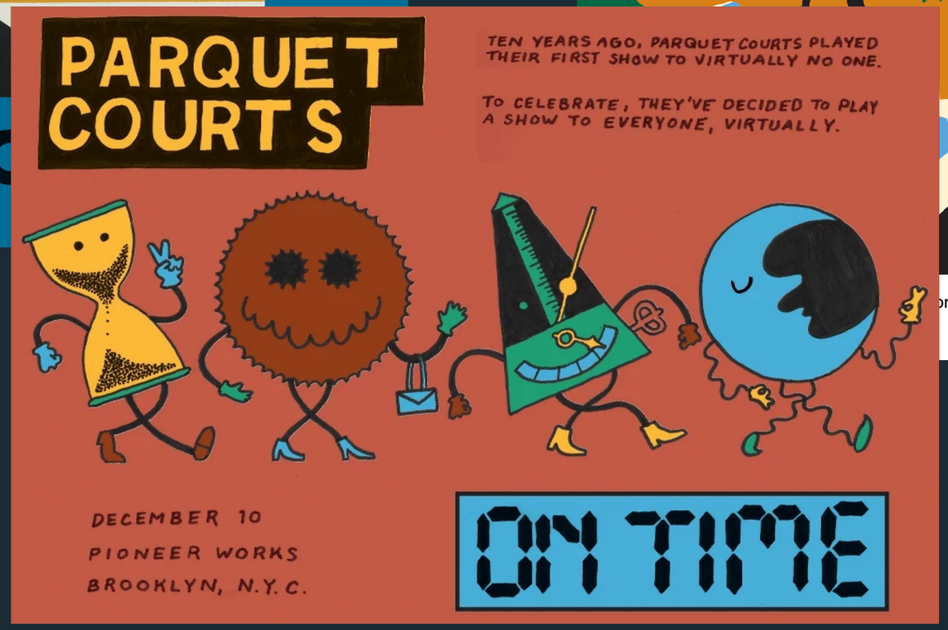
2. **Annotate! - for colors, can annotate dominant colors, relationships and palette choices**

Cool blue with tertiary greens and yellows evokes urban nature

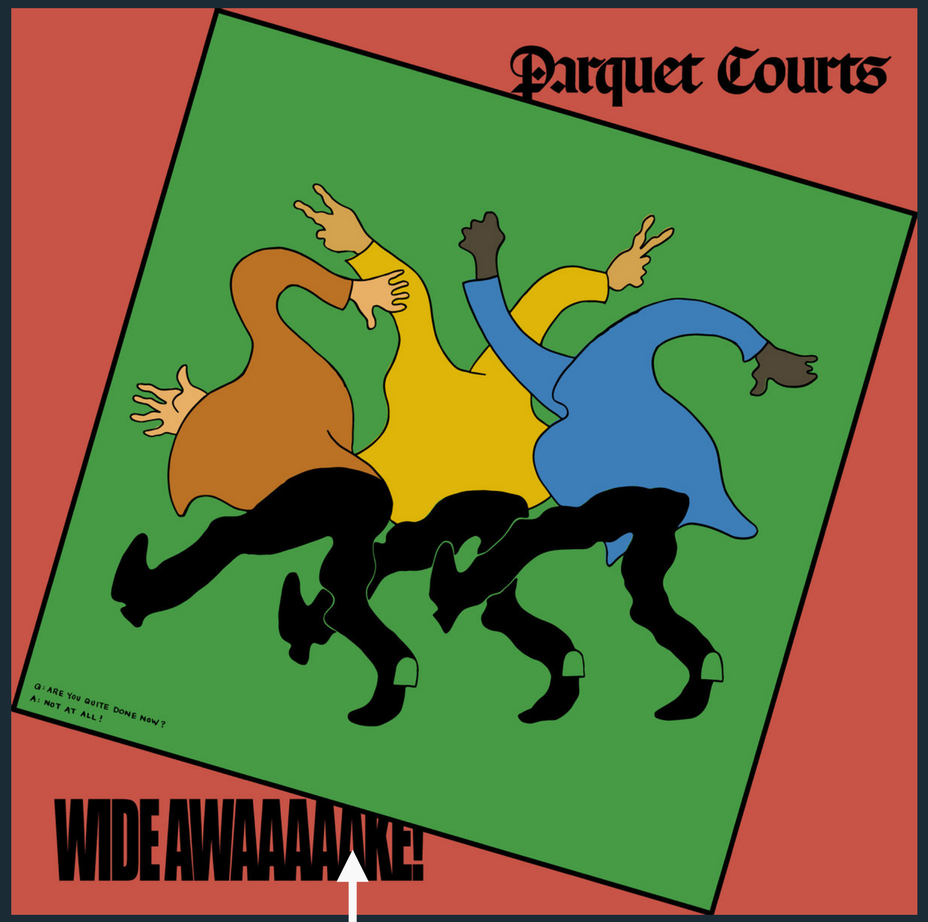


Square



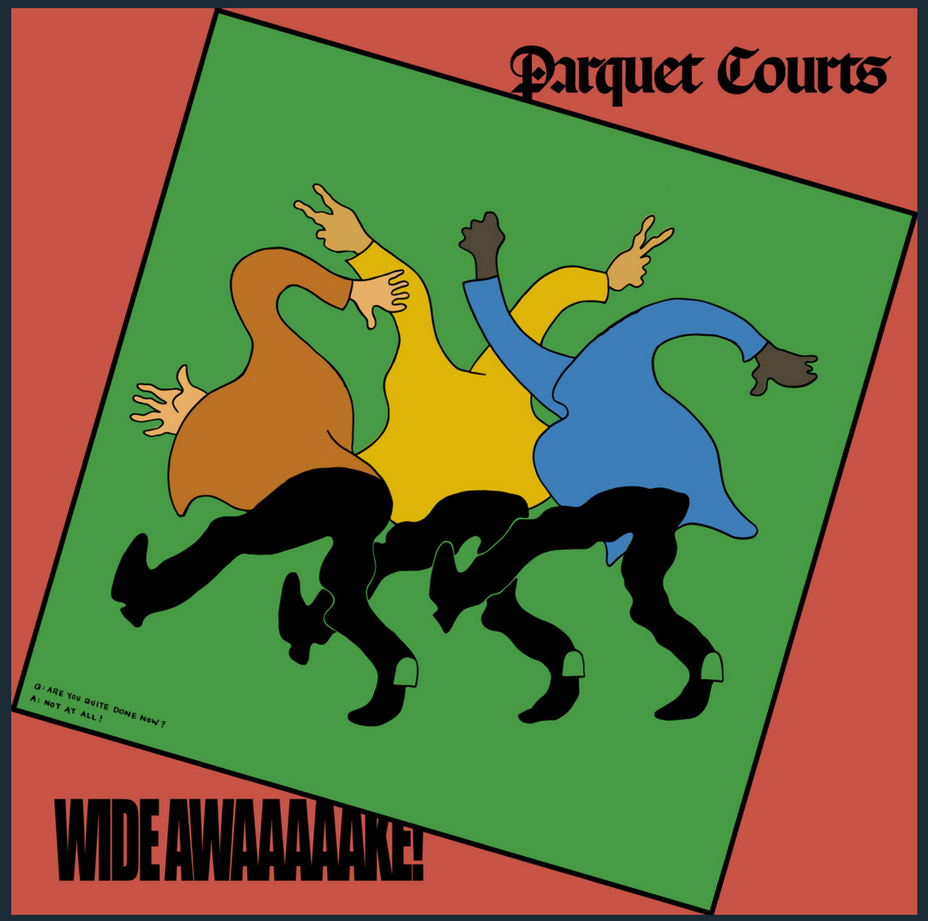


2. **Annotate! - for typography, consider typefaces, sizing, placing, content**



All caps! Text close together! Conveys excitement, fits with the content (WIDE AWAKE!)

2. **Annotate! - for typography, consider typefaces, sizing, placing, content**



Meanwhile, blocked and spaced, disjointed, broken up across the cover (“human performance”) — conveys a feeling of loss

3. **Reflect - synthesize your notes and what you've learned**

Color: The covers use bold complementary colors to create a striking, energetic tone—fitting for a punk band. By shifting from red-based palettes to cooler blues, greens, and yellows, the designer softens the mood, evoking a quieter, urban atmosphere.

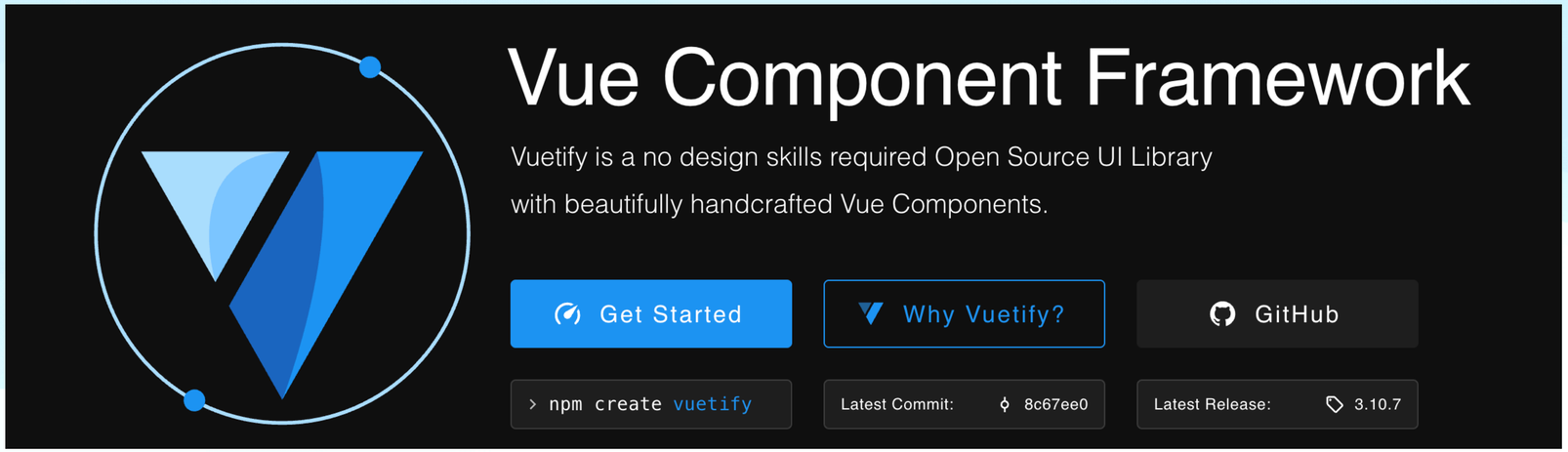
Typography: *Wide Awake!* uses tight, all-caps lettering that amplifies energy and chaos, matching its vivid imagery, while *Human Performance* spaces and fragments its text to convey melancholy and emotional distance.

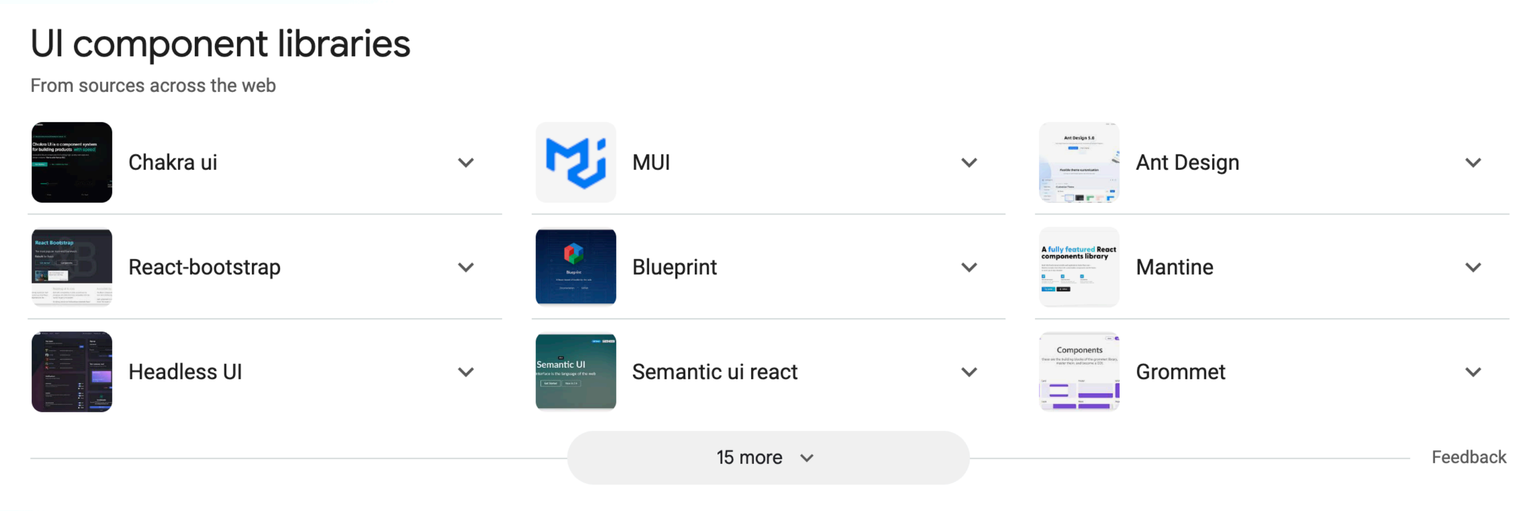
So, we know what makes a good visual design now…

… but how do we actually implement that in code?

Front-End Design Libraries

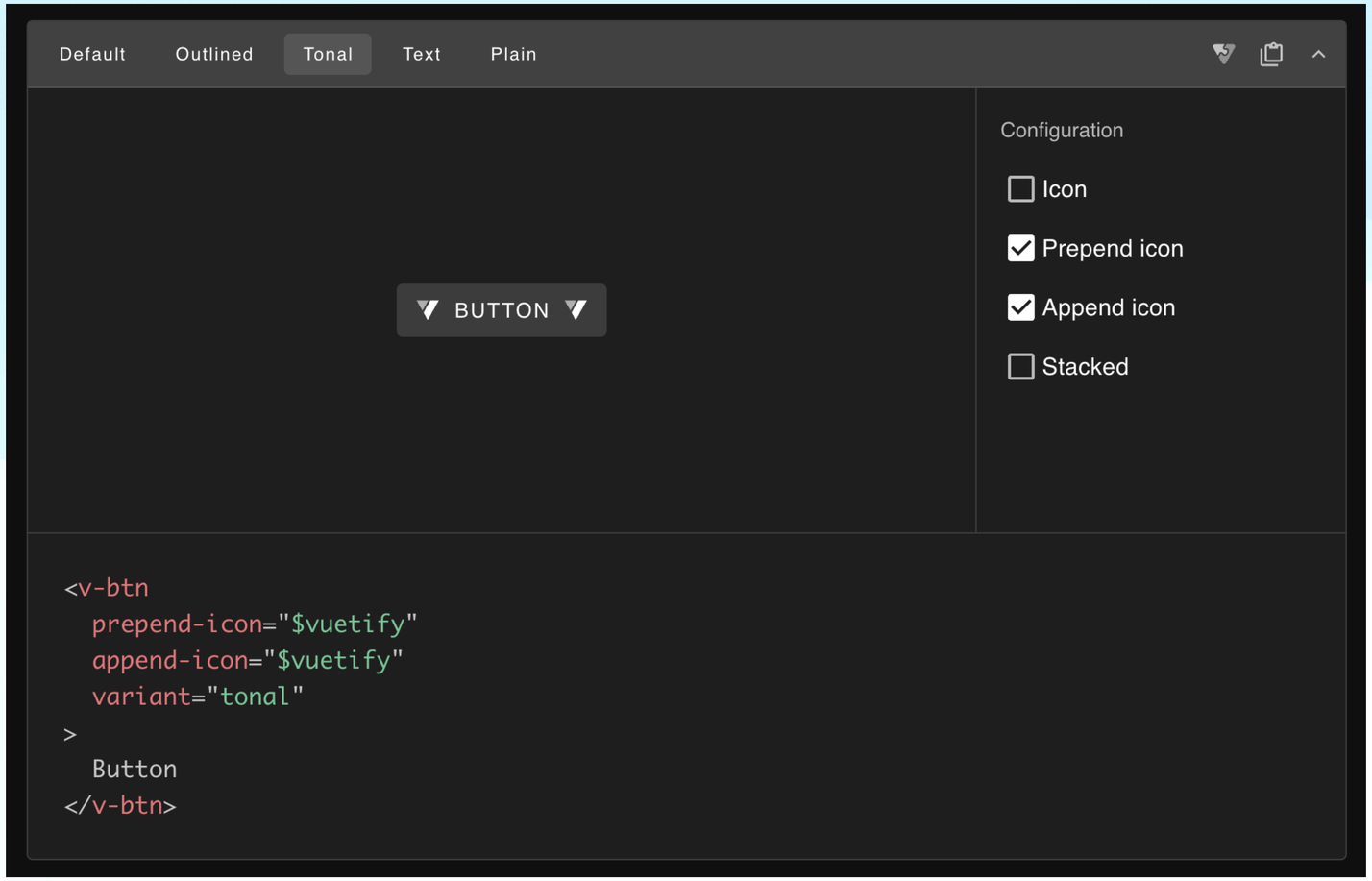
https://vuetifyjs.com/en/





Front-End Design Libraries

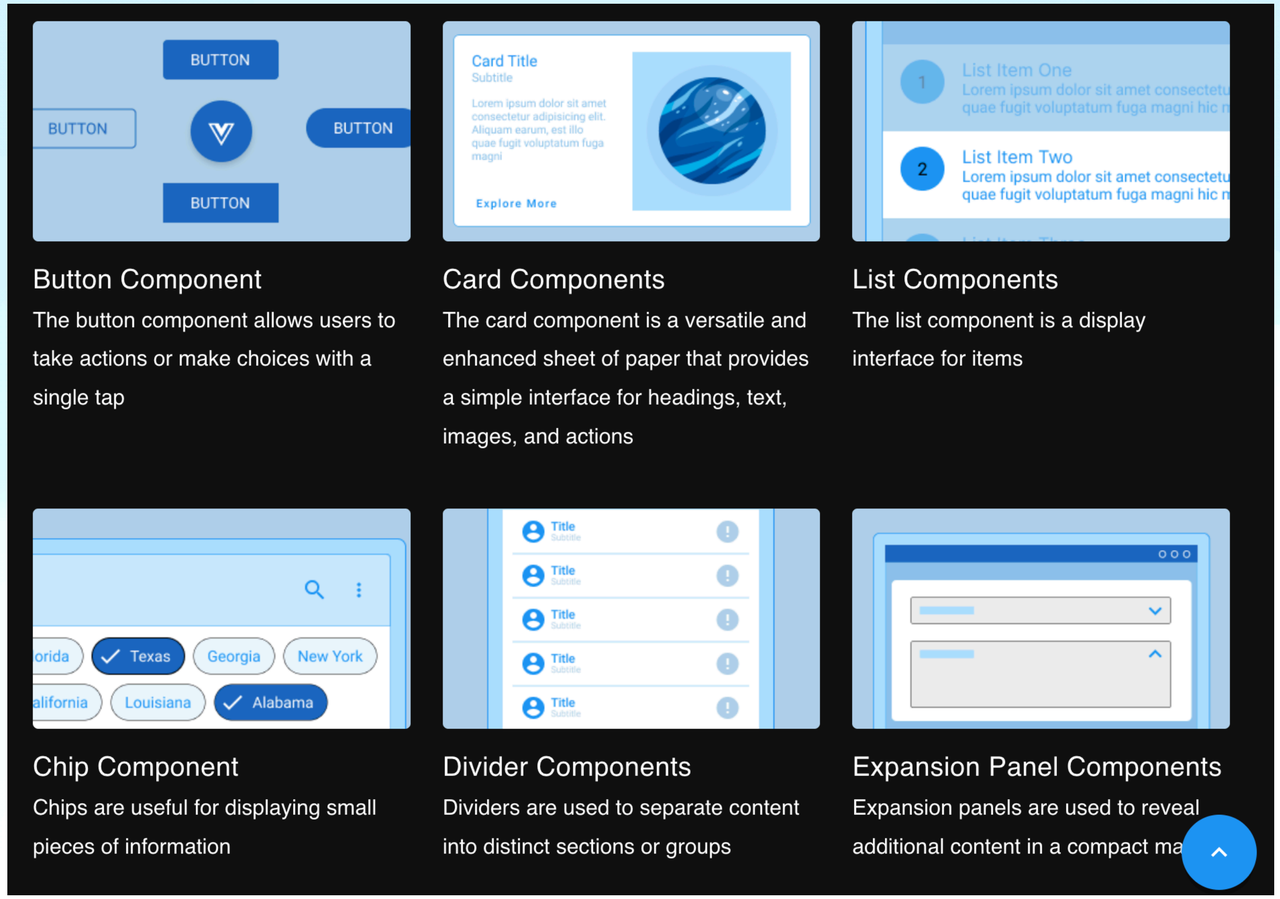
Pros



Have a lot of easy-to-use styles and components out of the box

Front-End Design Libraries

Pros



Button Component

The button component allows users to take actions or make choices with a single tap

Card Components

The card component is a versatile and enhanced sheet of paper that provides a simple interface for headings, text, images, and actions

List Components

The list component is a display interface for items

Chip Component

Chips are useful for displaying small pieces of information

Divider Components

Dividers are used to separate content into distinct sections or groups

Expansion Panel Components

Expansion panels are used to reveal additional content in a compact ma

Reusable components ensure a consistent visual design

Front-End Design Libraries

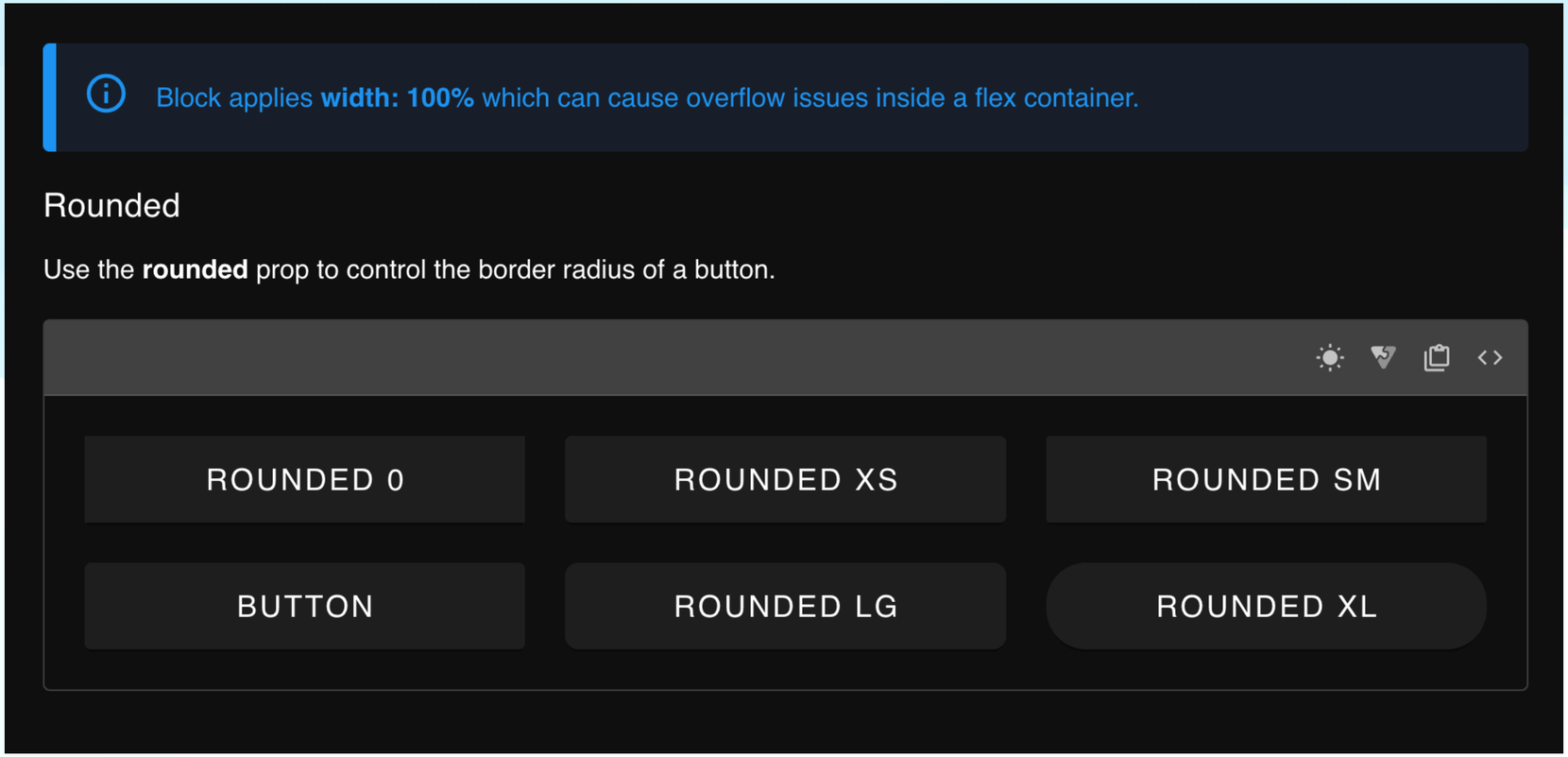
Pros

```
<template>
  <v-responsive class="border rounded" max-height="300">
    <v-app>
      <v-app-bar title="App bar"></v-app-bar>

      <v-navigation-drawer>
        <v-list>
          <v-list-item title="Navigation drawer"></v-list-item>
        </v-list>
      </v-navigation-drawer>

      <v-main>
        <v-container>
          <h1>Main Content</h1>
        </v-container>
      </v-main>
    </v-app>
  </v-responsive>
</template>
```

Makes resulting code more portable and usable for yourself and collaborators

Front-End Design Libraries

Cons



Customization can be a headache

Front-End Design Libraries

Cons

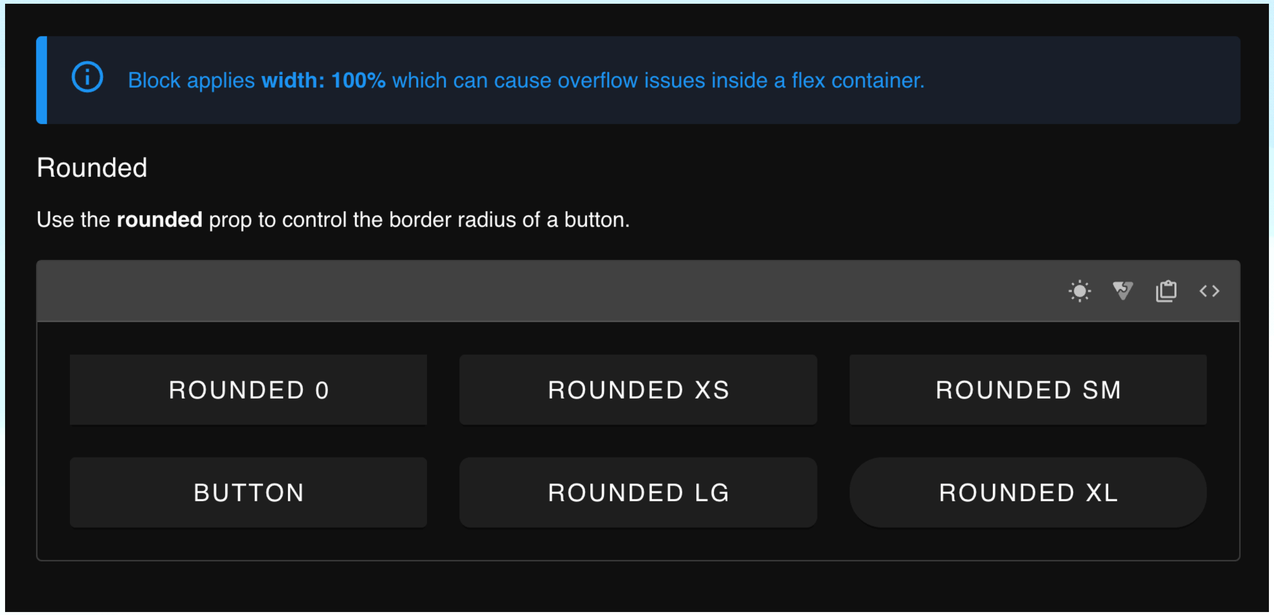
"I don't want my buttons to be gray"

Button container color	-	md.sys.color.primary
Button container color - toggle (unselected)	-	md.sys.color.surface-container
Button container color - toggle (selected)	-	md.sys.color.primary
Button shadow color	-	md.sys.color.shadow

"Okay, cool, we can configure the primary theme color"

Front-End Design Libraries

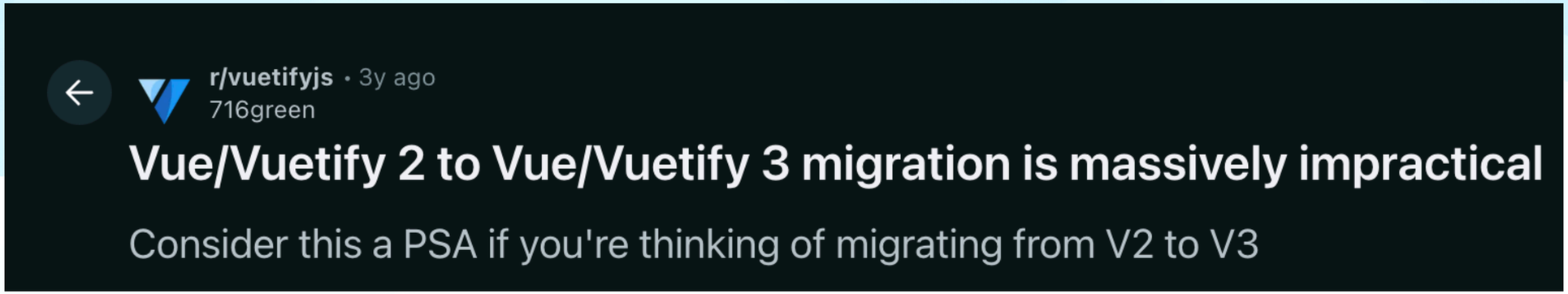
Cons



“I want my button to *not* use 100% width in my flex box … how do I do that?”

Front-End Design Libraries

Cons



Can go outdated or be incompatible with your front-end framework