Design System: Contributing

Styles

BEM

We are using a variant of BEM to manage our stylesheet architecture. When used properly, BEM helps us scale the list of components across our applications without the fear that we are modifying other components unintentionally.

The accepted BEM syntax's at GuideCX are:

- block
- block__element
- block--modifier
- block__element--modifier

For our purposes, the following guidelines should be used:

- a `Block` is the top-level component (`Primary` or `Secondary` buttons, `Card`, `Modal`, etc)
- an `Element` is a piece within the block. Not every component will need these. Good examples are `label` and `required` on the `Input` block
- a `Modifier` is also optional, and allows us to make a variant on either a block or element.
  - Good examples include `error` on an invalid input element, or `loading` on a button block
  - A bad example of a modifier would be `hover` or `disabled`, because css natively provides psuedo-selectors to manage those

Resources:

- http://getbem.com/introduction/
- https://css-tricks.com/bem-101/

Specificity

For a component library, specificity is very important. If the base library has high specificity, it is hard for consumers to make one-off stylistic changes when needed, often leading to the use of `!important` or inline styles. Both of these options are a last-resort.

There are a few ways to build low specifity components, and our use of Tailwind CSS and BEM help significantly. Generally, a component should not have a specificity higher than 20-30, though in very niche cases you may need to go beyond that.

- https://developer.mozilla.org/en-US/docs/Web/CSS/Specificity
- https://css-tricks.com/specifics-on-css-specificity/
- https://specificity.keegan.st/

Atomic CSS

We are not actively using atomic design, but it is a *very helpful* way to think about elements within a system. Thinking atomically will help you identify parts of a mockup or design that should be broken down into smaller components, and will help create better story boundaries and test plans.

- https://bradfrost.com/blog/post/atomic-web-design/

Tailwind CSS

We use Tailwind CSS for the majority of our actual styles. Using the `@apply` directive in our scss files, we can merge class names together into our custom BEM classes. This gives us all of the benefits of Tailwind, while also allowing us to add an architectural layer on top of it to provide some organization.

In cases where tailwind does not offer out-of-the-box support that meets our need, we should consider carefully whether to update our configuration file. If it is not appropriate to update the config, raw CSS can be used (still within the .scss files)

- https://tailwindcss.com/
- https://play.tailwindcss.com/

SCSS

SCSS is our pre-processor of choice for connecting HTML markup with the appropriate Tailwind CSS styles. SCSS was chosen primarily because of it's support for nesting capabilities, but can also benefit from other advanced features like mixins, functions, loops, and math.

We tend to treat SCSS as mostly an adapter layer.

- https://sass-lang.com/guide

Development Environment

Storybook

When you are building new components (or updating existing ones), you will be working in Storybook. It is a development environment & gallery that allows us to build components in isolation, and see how they interact in various settings (viewport size, preferred theme, different props, etc)

The goal is to build a reusable component inside of storybook, and then to implement the completed story in an actual application. Building components in this way has a higher probability of helping you design clean component APIs.

Chromatic

Chromatic is a visual regression testing tool that connects our SDLC pipeline with our storybook gallery. When a pull request is opened, a chromatic build starts and generates side-by-side comparisons of any components that have changed. As we build complex components that are importing each other, this will help us know what scale of changes we are making, and will aid us is catching potential bugs before they get released.

Throughout the process of you building components, you should be working with your team members, designer, and product manager to help you know what tweaks need to be made. Once your component is done and you have opened your PR, you will need to have a designer or PM approve the changes being made (in chromatic) so the github status check is successful and the PR becomes mergeable.

Typescript Path Aliases

As is the case with our other applications, the root of the project is accessible via `~`. This means from anywhere in the app, you can import files based on this base path like `~/components/button/primary` or `~/utils/hooks/useId`

Code Standards

As our applications have grown, the need for well-built components has become more and more apparent. The amount of time spent debugging very complex side effects grows exponentially, and developer experience is dimished.

In an attempt to help combat this, a few baseline standards should be followed when writing components in this directory:

Component props should be used explicitly instead of passed in via the spread operator (`{...props}`)

Spreading props leads to behaviors that are difficult to debug, and prop chains that are difficult to trace. If we are explicit about the props we receive (and the props that we pass in), it becomes significantly easier to reason about our app and know what is happening in our code.

`Bad:`

```
// we don't know what can or cannot be passed in
const Button = ({ children, ...props }) => {
  return (
    <button {...props}>{children}</button>
  )
}
```

`Good:`

```
// we are confident that only the props we have listed can be used. typescript aids us in the parent component
const Button = ({
  children,
  onClick,
  disabled = false
}) => {
  return (
    <button
      onClick={onClick}
      disabled={disabled}
    >
      {children}
    </button>
  )
}
```

Be explicit about which default props are supported

When declaring typescript interfaces, it is preferred to support a select few set of native html attributes rather than everything blindly. This is easy enough to do since we are not spreading props, either.

`Bad:`

```
// we are accepting every html attribute
interface Props extends React.HTMLAttributes<HTMLElement> {...}

const Button: React.FC<Props> = (...) => {...}
```

`Good:`

```
// we are only allowing className and onClick, and are omitting other element
attributes like onDblClick
type SupportedDefaultProps = Pick<
  React.HTMLAttributes<HTMLElement>,
  'className' | 'onClick'
>

interface Props extends SupportedDefaultProps {...}

const Button: React.FC<Props> = (...) => {...}
```

Parent components should be able to easily override styles

One of the major responsibilities of a component library is to be extensible. The components in this folder do not know where they are going to be rendered, so they must support the ability to be modified on a case-by-case basis (though they should provide a reasonable default style that matches the most common use-case).

If a component does not accept the `className` prop, that prop has too low of a specifity, or the default styles have too high of a specifity, it will be too difficult for the parent to modify the styles.

As another example, if the styles for a component are too nested or complex, or are not organized into BEM standards, this rule is broken.

`Bad:`

```
// these are too specific
.Primary {
  &__bg {
    @apply bg-blue;
  }

  &__rounded {
    @apply border rounded;
  }

  ...
}
```

```
// The class names are overridden by the ids
const Button = ({ className }) => {
  return (
    <button id="Primary" className={className} />
```

```
  )
}
```

`Good:`

```
// The default styles have a low specifity, allowing the parent to easily override
const Button = ({ className }) => {
  return (
    <button className={['Primary', className].join(' ')} />
  )
}
```

Styles should be done in a SCSS file using BEM, Atomic CSS, and Tailwind CSS

There are a lot of reasons this is the way we have structured our application. We have cleaner markup, better separation between functional code and presentational code, and we have industry-proven methods for scaling our design system.

`Bad:`

```
// markup is muddied by so many classes
<div class="relative flex min-h-screen flex-col justify-center overflow-hidden bg-
gray-50 py-6 sm:py-12">
  <img src="/img/beams.jpg" alt="" class="absolute top-1/2 left-1/2 max-w-none -
translate-x-1/2 -translate-y-1/2" width="1308" />
  <div class="absolute inset-0 bg-[url(/img/grid.svg)] bg-center [mask-image:linear-
gradient(180deg,white,rgba(255,255,255,0))]"></div>
  <div class="relative bg-white px-6 pt-10 pb-8 shadow-xl ring-1 ring-gray-900/5
sm:mx-auto sm:max-w-lg sm:rounded-lg sm:px-10">
    <div class="mx-auto max-w-md">
    </div>
  </div>
</div>
```

```
.Button {
  // not using BEM, class is too specific
  &.bg {
    @apply bg-blue;
  }

  // not using tailwind
  &__rounded {
    border: 1px solid red;
    border-radius: 8px;
  }

  // not using psuedo selectors
  &--hover {
    @apply bg-blue-300;
  }
}
```

`Good:`

```
// easy to see functional code, markup not messy
const Button = ({ onClick }) => {
  return (
    <button className="PrimaryButton" onClick={onClick} />
  )
}
```

```
.PrimaryButton {
  @apply ...

  // good use of BEM at every level
  &__icon {
    @apply ...

    &--left {
      // consistently using tailwind classes
      @apply ...
    }

    &--right {
      @apply ...
    }
  }

  // using psuedo selectors where possible
  &:hover {
    @apply ...
  }
}
```

```
.PrimaryButton {
  // using tailwind's psuedo selectors is okay, too
  @apply hover:bg-blue
}
```

Do not use JS when CSS will suffice

Many developers would be surprised by how powerful CSS is. As we are trying to use the right tool for the job, we need to stop using javascript to solve problems that are easily solveable in css.

`Bad:`

```
const Button = (...) => {
  return (
    <button
      onMouseEnter={handleMouseEnter}
      onMouseLeave={handleMouseLeave}
      className={hovering ? 'hover' : ''}
    />
```

```
  )
}
```

`Good:`

```
const Button = (...) => {
  return (
    <button className="Button" />
  )
}
```

```
.Button {
  &:hover {
    @apply ...
  }
}
```

Prefer semantic HTML over generic tags

One of the major benefits of using semantic tags is that we gain a huge lead for SEO, and users with various assistance tools (like screen readers) have an easier time navigating our apps. We should avoid using generic divs or spans when there are other, more semantic tags available

`Bad:`

```
const Button = (...) => {
  return (
    <div onClick={...} />
  )
}
```

`Good:`

```
const Button = (...) => {
  return (
    <button onClick={...} />
  )
}
```

CSS class names are joined in an array

When joining class names in an array instead of passing them all in as strings, we gain a few benefits. To name one, when we are inspecting the DOM, it is easier to find what we are looking for. We don't have to mentally filter out a bunch of `false` or `undefined` class names (or auto-generated css module names).

`Bad:`

```
const Button = ({ className }) => {
  return (
    <button className={`${className} Primary ${bool ? 'something' : 'other'}`} />
  )
}
```

`Good:`

```
const Button = ({ className }) => {
  return (
    <button
      className={[
        'Primary',
        className,
        bool ? 'something' : 'other'
      ].join(' ')}
    />
  )
}
```

Global CSS classes should be prefixed so we don't confuse them with tailwind class names

In some cases, it makes sense to make global classes (or scss mixins) for tailwind classes that are frequently used together. In these cases, it is acceptable to add or update a global stylesheet. However, when doing this, we should prefix class names with `GCX-` to avoid overwriting existing class names.

`Bad:`

```
// ~/styles/reset.scss
.bg-red-500 {
  background: white;
}
```

`Good:`

```
// ~/styles/reset.scss

// we can be confident that we are not overwriting a tailwind class name here
.GCX-button {
  @apply ...
}
```

Components should not be aware of GuideCX data

The following sections on various `Design Philosophies` will hopefully illustrate why components in this application should be presentational only. They should accept generic data, and render it, but should not fetch their own data, or be aware of domain-specific language or models.

This is easier in practice than it may seem at first, and the more you build components this way, the better you will get at it.

`Bad:`

```
// this component knows way too much about the data it is rendering
const Table = ({ users }) => {
  return (
```

```
    <table>
      {users.map(user => {
        if (user.type === 'Project Manager') {
          return (...)
        }
        ...
      })}
    </table>
  )
}
```

`Good:`

```
// the table is generic, so will work in multiple applications and on multiple pages
const Table = ({ columns, records }) => {
  return (
    <table>
      {records.map(record => {
        return <TableRow columns={columns} record={record}>
      })}
    </table>
  )
}
```

Design Philosphies

There are many different ways to build a component library. Many large organizations operate with different paradigms, and have created successful tools. We have chosen a few primary patterns to build our components with, but not every pattern is the right tool for the job, so every developer needs to be cognisant of which pattern to use and when.

Base/Variant

The base/variant pattern is a great option for components like inputs that generally share the same basic functionality, but have stylistic, or slight functional differences. For example, a base input class may have an interface like this:

```
interface BaseInput {
  type?: string;
  value?: string;
  onChange(event: any): void;
  required?: boolean;
  placeholder?: string;
}
```

that may lend itself to a currency variant or an icon variant:

```
interface CurrencyInput extends BaseInput {
  locale?: string;
  precision?: int;
}
```

```
const CurrencyInput = (...) => {
  return (
    <BaseInput ... />
  )
}

interface IconInput extends BaseInput {
  Icon?: React.Element;
  position?: 'left' | 'right'
}

const IconInput = (...) => {
  return (
    <BaseInput ... />
  )
}
```

However, the base/variant approach is probably not a good solution for a Card component where there will not be many variants being created

- https://blog.bitsrc.io/design-systems-react-buttons-with-the-base-variant-pattern-c56a3b394aaf
- https://spotify.design/article/reimagining-design-systems-at-spotify

Presentational vs Container Components

The principles of Presentational components should influence everything in this application, and has been discussed in previous sections. A good, reusable component should be domain-agnostic.

- https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0

Compound Components

Compound components are a great option for complex components that dynamically place content inside of them, or contain fragmented components that need to pass data around in a clean, self-contained way. A good example may be a Table component that has `Table.Header`, `Table.Row`, and `Table.Filters` children. These components likely need to interact with each other, and could be rendered in different ways, so using compound components allow us to manage complex interactions between components without making the parent component(s) messy with state.

- https://kentcdodds.com/blog/compound-components-with-react-hooks