Utility Props in NativeBase v3

What we like, love and more

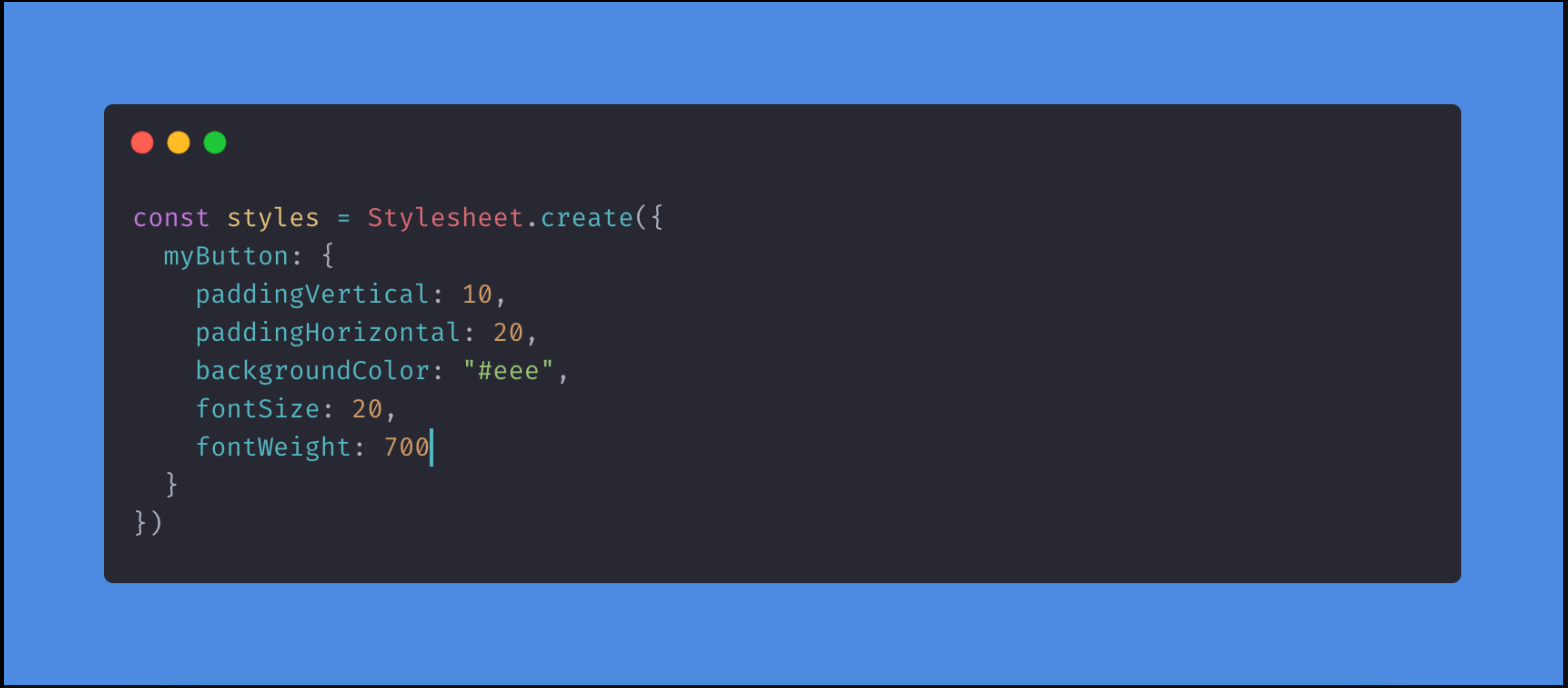
I’m Vipul 👋

- SSE @GeekyAnts.
- My interests are in Developer Experience and Tooling.
- I go by @vipulbhj on the internet.
- And I like sunrises.
- Because they reminds me it’s time to sleep 😛

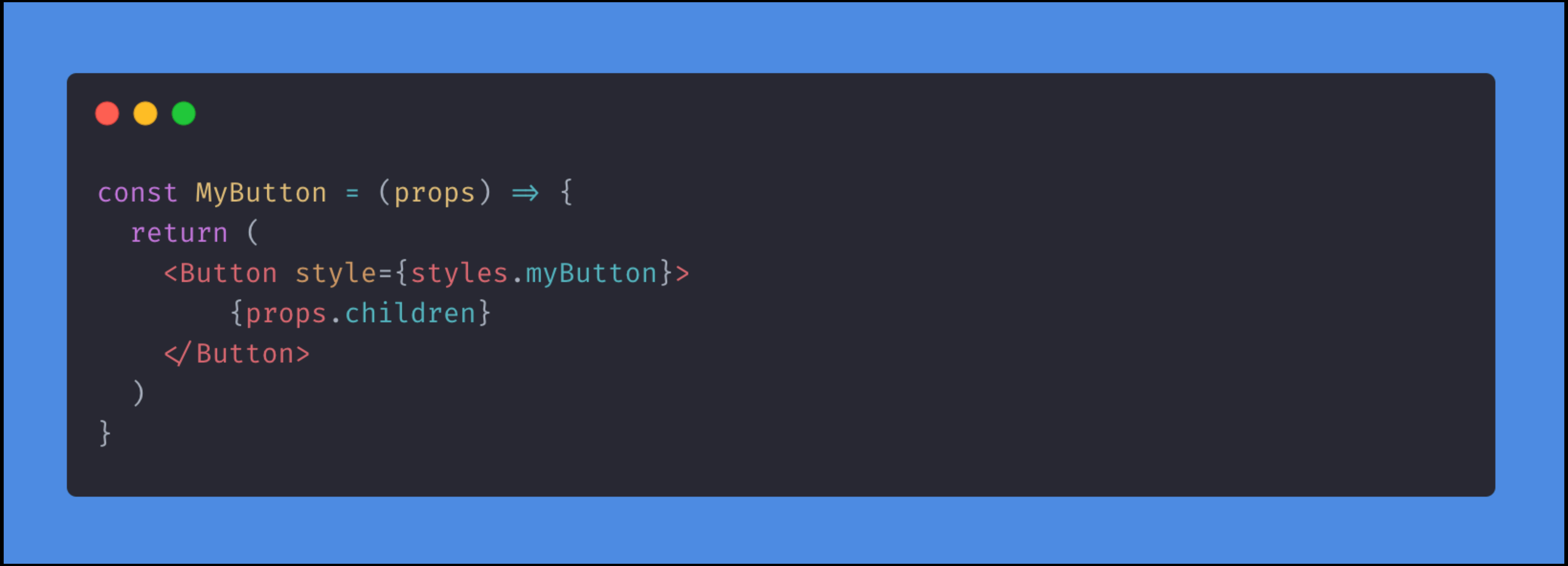
Before we begin

📖 ➡️ “🕗🕢” ➡️ 🧑🏻‍🏫 ➡️ “🎥❌” ➡️

📥 ➡️ 🎙️ ➡️ 📤 ➡️ “IKR, 😞😞😞”



```
const styles = Stylesheet.create({
  myButton: {
    paddingVertical: 10,
    paddingHorizontal: 20,
    backgroundColor: "#eee",
    fontSize: 20,
    fontWeight: 700
  }
})
```



```
const MyButton = (props) => {
  return (
    <Button style={styles.myButton}>
        {props.children}
    </Button>
  )
}
```

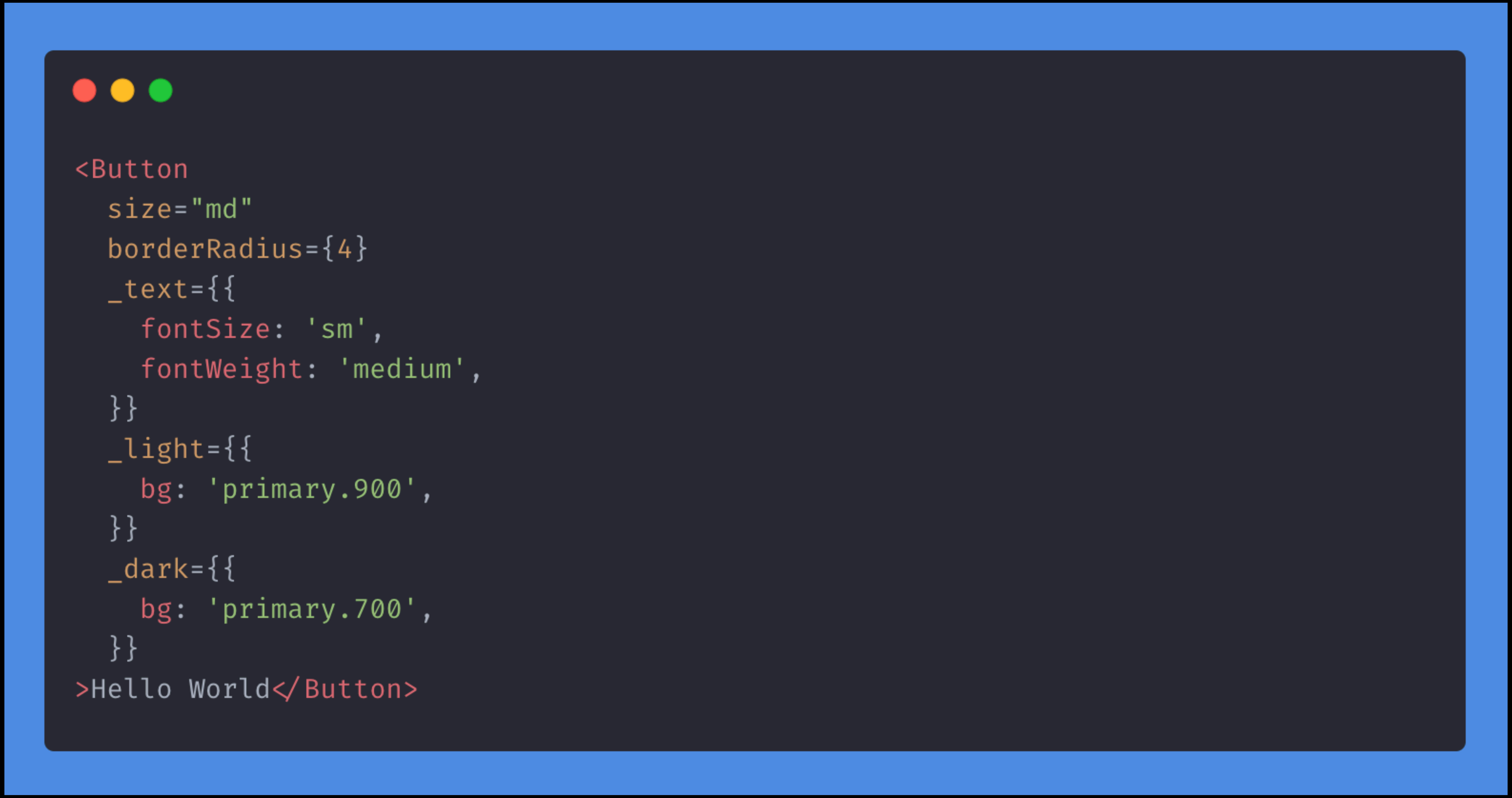
The cost of context switching

Separation of concern

- To write more maintainable code, the general advice is to keep things in isolated modules, which only do one thing.
- React Native Developers commonly interpret this as an advice to keep your markup and styles separate.
- While separating markup and styles work, it might not be the right approach for your use-case because of the penalties context switching brings.

$$\text{Productivity} \propto \frac{1}{\text{Context Switching Effort}}$$

How about utility props ?



```
<Button
  size="md"
  borderRadius={4}
  _text={{
    fontSize: 'sm',
    fontWeight: 'medium',
  }}
  _light={{
    bg: 'primary.900',
  }}
  _dark={{
    bg: 'primary.700',
  }}
>Hello World</Button>
```

What are utility props anyways

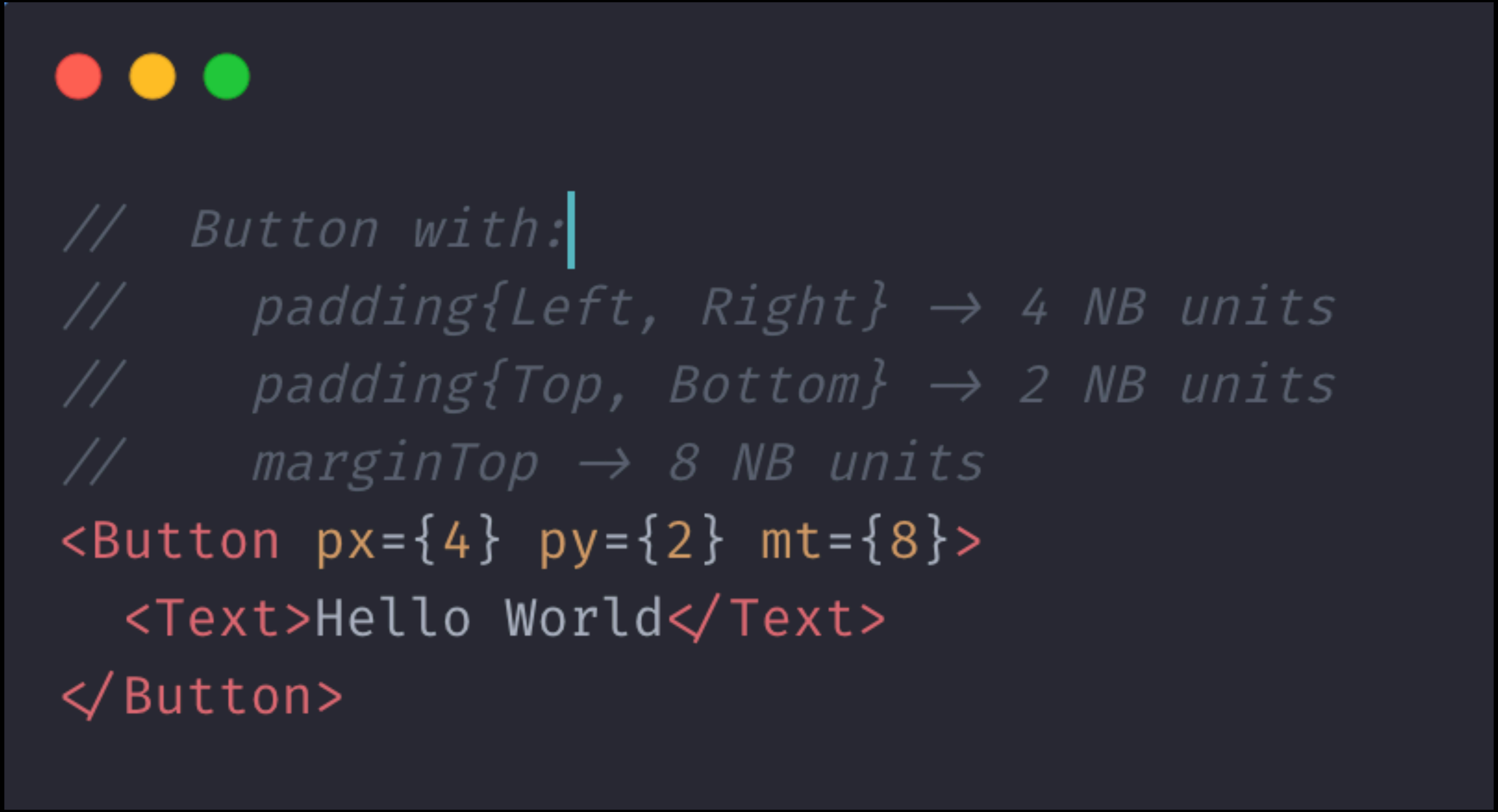
- It’s a declarative way of styling components using a collection of UI props.
- With this, reasoning about certain parts of your code become easier and involves less context switching.
- Utility props are inspired by the concept of utility classes, an approach popularised by libraries like \`tailwind\` but are not utility classes and doesn’t come with the added baggage of users having to remember (sometimes weird) classnames to fit into the ecosystem of using utility classes.

So, why does NativeBase use them anyways ??

Well, I am glad you asked 😉

Why we like utility props

- You are not wasting energy inventing selector names for your components.
- Boosts development speed by avoiding context switching.
- Making changes feel safer since changes are always local and modular, so no mistakenly changing all Buttons in your application, when you tried to change only one.
- Allow us to provide more intuitive and declarative ways to customise your components for different colour modes, platforms, etc.



```
//  Button with:
//     padding{Left, Right} → 4 NB units
//     padding{Top, Bottom} → 2 NB units
//     marginTop → 8 NB units
<Button px={4} py={2} mt={8}>
  <Text>Hello World</Text>
</Button>
```

Why not just use inline styles ?

- Using inline styles, every value is a magic number. With utilities, you’re choosing styles from a predefined design system, which makes it much easier to build visually consistent UI.
- Complex logic is needed on handle cases like “Responsive Design”, “Colour Modes”, etc which pollutes your styles.
- Handling state and event driven styles for things like \`hover\`, \`focus\`, etc become hard to manage and involves logic in your styles

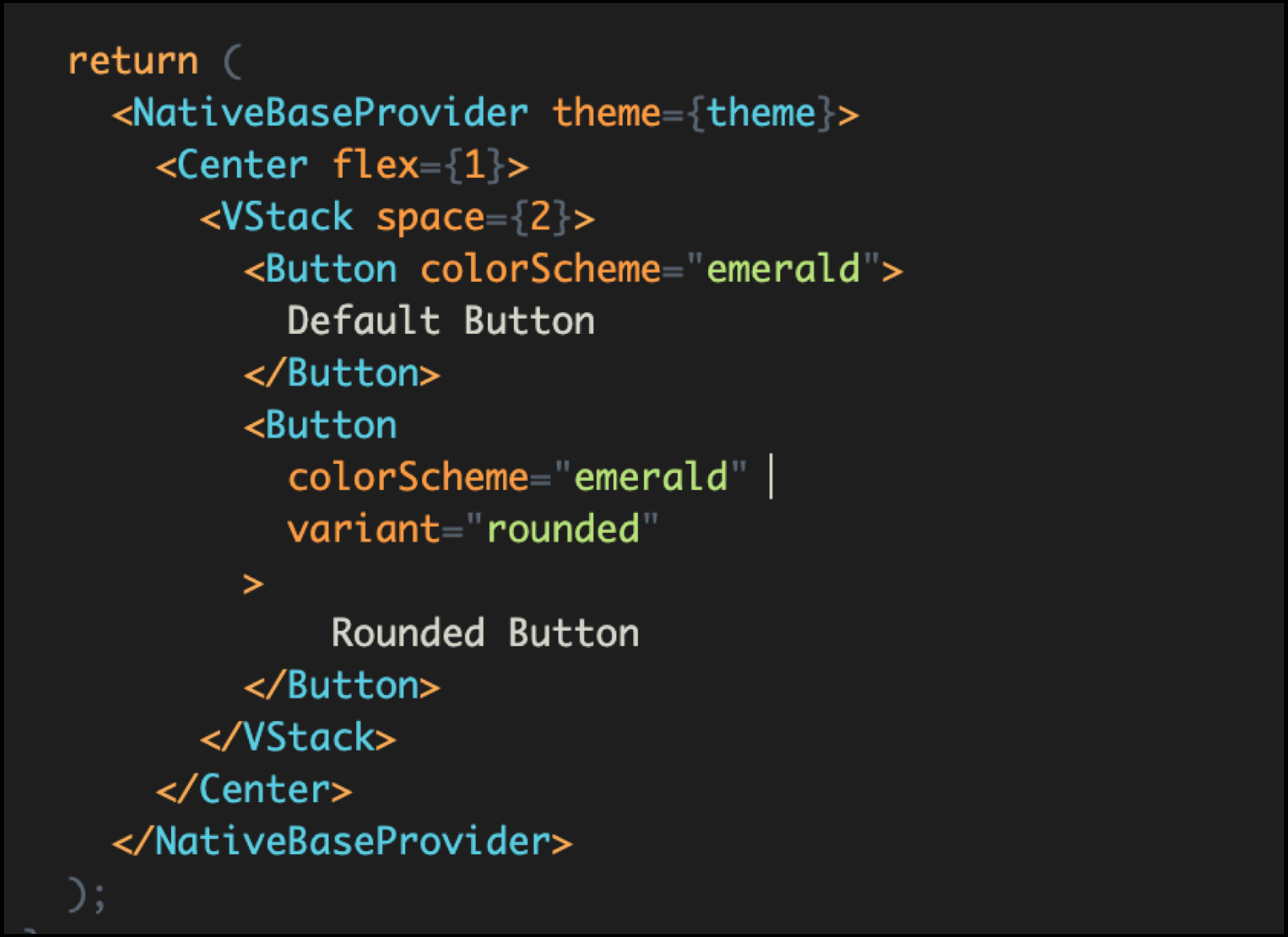
Are there any Maintainability concerns ?

- The way NativeBase handles Maintainability is by introducing the notion of variants. A variant represents a component which inherits most of the base styles but different in a small set of properties.
- A very common example of this would be an application having various types of `Alert` representing “success”, “failure”, “info”, etc. All of them have a standard base style but a few properties that are different in each.

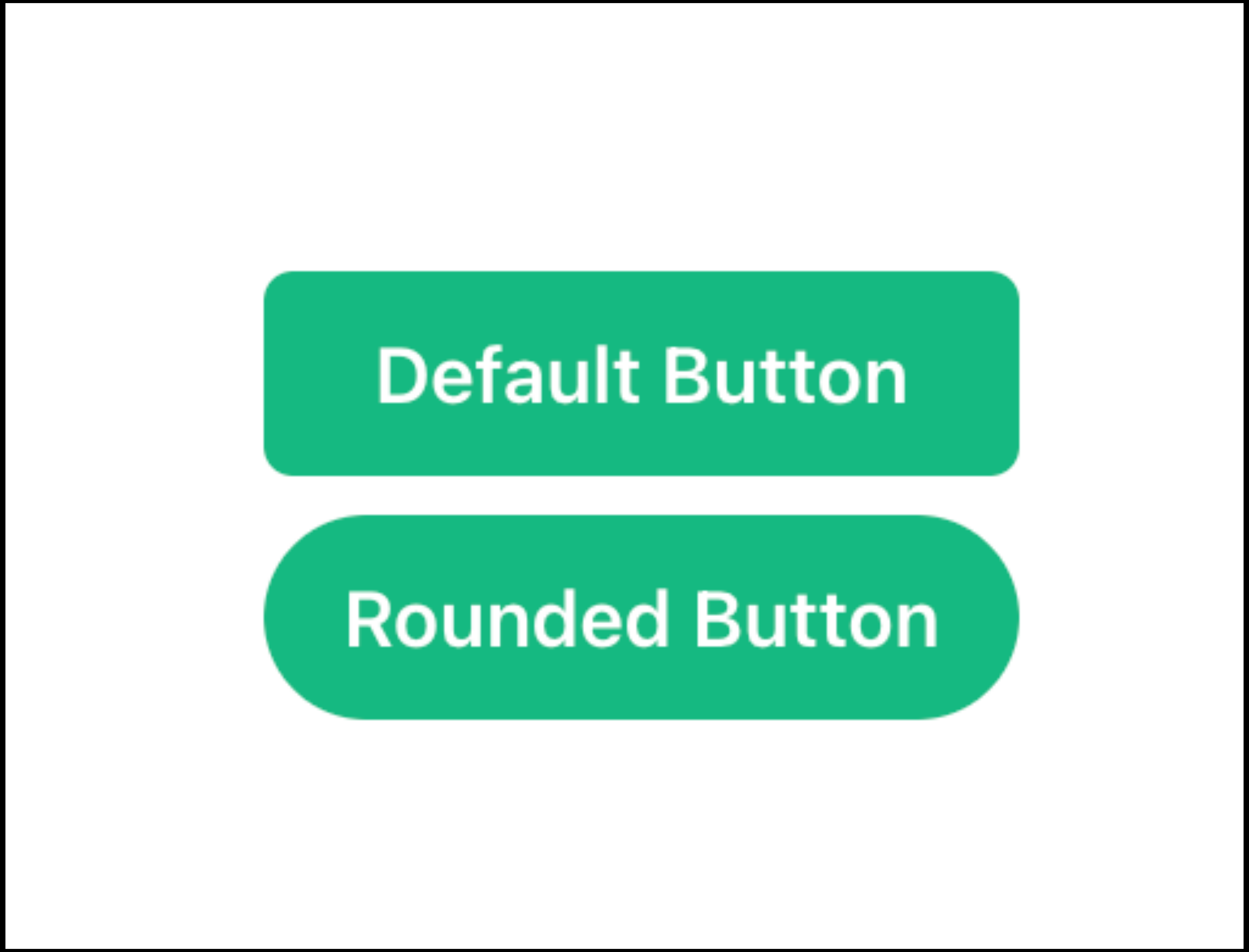
- NativeBase allows users to create variants for any component that might have a similar use-case as the one mentioned previously. This allows you to write more manageable code and keep your application markup clean.
- You can learn more about variants on https://docs.nativebase.io/customizing-components

```
import React from 'react';
import {
  NativeBaseProvider,
  Button,
  extendTheme,
  Center,
  VStack,
} from 'native-base';

export default function () {
  const theme = extendTheme({
    components: {
      Button: {
        variants: {
          rounded: ({ colorScheme }: any) => {
            return {
              bg: `${colorScheme}.500`,
              rounded: 'full',
            };
          },
        },
      },
    },
  });
```



```
return (
  <NativeBaseProvider theme={theme}>
    <Center flex={1}>
      <VStack space={2}>
        <Button colorScheme="emerald">
          Default Button
        </Button>
        <Button
          colorScheme="emerald"
          variant="rounded"
        >
            Rounded Button
        </Button>
      </VStack>
    </Center>
  </NativeBaseProvider>
);
```


Default Button
Rounded Button

And there is much more you can explore on the NativeBase docs website at https://docs.nativebase.io

Thank You