UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

MATHEUS PROCÓPIO DA SILVA

ATINGINDO INTERFACES DE USUÁRIOS CONSISTENTES POR COMPOSIÇÃO E SISTEMAS DE TOKENS

CAMPINA GRANDE - PB

2019

MATHEUS PROCÓPIO DA SILVA

ATINGINDO INTERFACES DE USUÁRIOS CONSISTENTES POR COMPOSIÇÃO E SISTEMAS DE TOKENS

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Professor Dr. Adalberto Cajueiro de Farias.

CAMPINA GRANDE - PB

2019

S586a Silva, Matheus Procópio da.
Atingindo interfaces de usuários consistentes por composição e sistemas de tokens. / Matheus Procópio da Silva. – 2019.

11 f.

Orientador: Prof. Dr. Adalberto Cajueiro de Farias.
Trabalho de Conclusão de Curso - Artigo (Curso de Bacharelado em Ciência da Computação) - Universidade Federal de Campina Grande; Centro de Engenharia Elétrica e Informática.

1. Design de interface. 2. Tokens. 3. Interfaces de usuários. 4. Consistência - design. 5. Biblioteca styled components. 6. Programação funcional. I. Farias, Adalberto Cajueiro de. II. Título.

CDU:004(045)

Elaboração da Ficha Catalográfica:

Johnny Rodrigues Barbosa
Bibliotecário-Documentalista
CRB-15/626

MATHEUS PROCÓPIO DA SILVA

ATINGINDO INTERFACES DE USUÁRIOS CONSISTENTES POR COMPOSIÇÃO E SISTEMAS DE TOKENS

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. Adalberto Cajueiro de Farias
Orientador – UASC/CEEI/UFCG

Professora Dra. Francilene Procópio Garcia
Examinadora – UASC/CEEI/UFCG

Professor Dr. Tiago Lima Massoni
Disciplina TCC– UASC/CEEI/UFCG

Trabalho aprovado em: 25 de novembro 2019.

CAMPINA GRANDE - PB

Atingindo interfaces de usuário consistentes por composição e sistemas de tokens

Trabalho de conclusão de curso

Adalberto Cajueiro de Farias
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil
adalberto@computacao.ufcg.edu.br

Matheus Procópio da Silva
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil
matheus.procopio.silva@ccc.ufcg.edu.br

RESUMO

Definir uma arquitetura de componentes genérica que tenha como objetivo principal atingir uma interface de usuário consistente é uma tarefa difícil pois as especificações de design variam muito. Este documento apresenta duas bibliotecas que permitem compor componentes mais flexíveis e adaptáveis a diferentes cenários, com o fim de facilitar qualquer usuário a construir sua própria arquitetura a partir de suas próprias especificações.

Palavras chave

react, componente, design de interface, biblioteca, npm, tokens, consistência.

1. INTRODUÇÃO

A consistência é um princípio fundamental no design pois transmite a nós humanos, um sentimento de confiabilidade, segurança e proteção. A interface é a linguagem em que nos comunicamos com os usuários, logo sua inconsistência resulta em uma experiência de uso desagradável.

A usabilidade e a capacidade de aprendizagem melhoram quando elementos semelhantes têm aparência e funcionalidade semelhantes. Logo os usuários podem concentrar-se na execução da tarefa desejada, sem a necessidade de reaprender como a interface do produto funciona a cada mudança de página ou contexto.

Uma boa experiência de uso é crucial para o sucesso de muitos negócios, no e commerce, por exemplo podemos ver que: Um site bem projetado pode ter uma taxa de conversão de visitas a pedidos 200% mais alta do que um site mal projetado (FORRESTER, 2016). Considerando o faturamento de R$53,2 bilhões em 2018 com comércio digital no Brasil (EBIT/NILSEN, 2019), é natural a preocupação das companhias com a qualidade de sua interface.

Podemos dividir a UX de aplicações web e mobile em duas partes: guia de estilo e componentes de código. A primeira, é uma representação visual construída por designers contendo como o produto deve ser apresentado e compreendido. Os componentes são blocos de código reutilizáveis que implementam as especificações. Um guia inconsistente ou confuso pode ocasionar um delay no desenvolvimento de melhorias incrementais pois impede que designers e engenheiros conversem de forma clara e sucinta, atrasando a tomada de decisões.

Como temos dois problemas em nossa mão, podemos dividir para conquistar e assumir que os designers são competentes e entregaram um guia de estilo ideal. A partir disso, como podemos estruturar o frontend para corresponder ao design esperado mantendo a qualidade de código?

2. SOLUÇÃO

A fim de resolver os problemas de consistência foi desenvolvida uma biblioteca typescript chamada *styled-tokens* que permite a geração de tokens capazes de se adaptar à mudanças, mesmo que drásticas, ao design. O resultado final pode ser encontrado no repositório do github *matheusps/styled-tokens*.

Com o objetivo de tornar a *styled-tokens* mais acessível, foi desenvolvida uma outra biblioteca de nome *adapt-ui* que provém tokens built-in e componentes prontos para recebê-los. O resultado final pode ser encontrado no repositório do github *matheusps/adapt-ui*.

Vale lembrar que ambas ferramentas estão disponíveis no npm através dos pacotes de mesmo nome, podendo ser utilizadas por qualquer aplicação que tenha *react, react-dom* e *styled-components* como dependências.

2.1 Estilizando componentes utilizando CSS

CSS é uma tecnologia peculiar. O básico pode ser aprendido em um período extremamente curto, mas podem levar anos até a descoberta de uma boa arquitetura para a organização de estilos. Isto se deve parte às limitações da própria linguagem como esquema de variáveis diminuto, ausência de loops e funções, impossibilidade de herdar ou compor novas classes a partir das existentes, suporte limitado à criação de módulos, etc. Pré-processadores como SASS e LESS adicionam tais recursos, mesmo que para isso necessitem realizar a aplicação de um arcabouço consideravelmente grande no projeto.

A anarquia de nomenclatura e organização podem ser resolvidos adotando metodologias como o BEM, que - apesar de útil - é totalmente opcional e não pode ser aplicado a nível do idioma ou checado por ferramentas de forma simples e efetiva. Além que apresentam uma curva de aprendizado para todos os integrantes de um time.

Com a popularização da organização de apps frontend por meio de componentes, a fim de evitar conflitos de escopo de estilos, passou-se a criar um arquivo css para cada correspondente em javascript. Isto é negativo se pensarmos no número excessivo de arquivos que todo o projeto passa a ter. O exemplo abaixo, ilustra este cenário

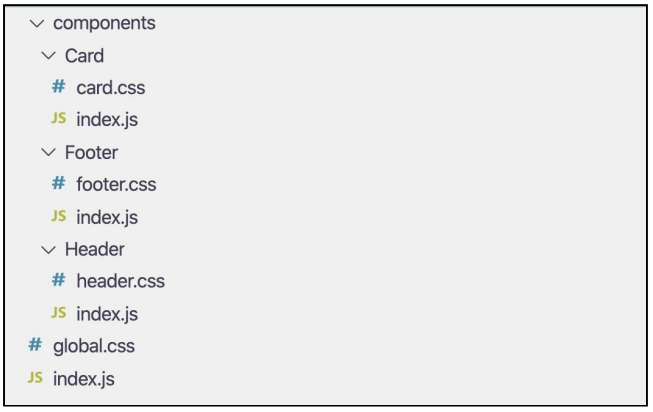

Figura 1: Árvore de componentes com CSS.

2.2 Styled Components e Programação funcional

A biblioteca *styled-components* apresenta uma alternativa para modelar estilos em componentes React, contemplando uma ótima experiência desenvolvimento. Ela permite definir a estilização dos componentes sem a necessidade de criar arquivos, assim a árvore da figura 1 se tornaria:

Figura 2: Árvore de componentes com styled-components.

A performance é garantida através do controle de quais os componentes são renderizados na página atual e injeta os estilos automaticamente, logo o usuário carrega a menor quantidade de necessária. E mantém escopo de estilo gerando nomes de classe restritos para cada componente, logo a preocupação com duplicação e sobreposição é eliminada.

A biblioteca aceita funções como argumentos para criar estilos dinâmicos com base em nas propriedades. O exemplo abaixo expressa como a propriedade *padding* pode ser definida de forma dinâmica:

```
import styled from 'styled-components'

const Section = styled.section`
  padding: ${props => props.padding};
`

//Usage
<Section padding="2rem" />
```

Figura 3: Propriedade padding com styled-components.

Podemos realizar um refactoring no código acima, extraindo a função para uma de nome significativo:

```
import styled from 'styled-components'

const padding = props => ({
  padding: props.padding
})

const Section = styled.section`
  ${padding};
`

//Usage
<Section padding="2rem" />
```

Figura 4: Refactoring da figura 3.

Nota-se que o problema pode ser resolvido com composição de funções. Com um bom uso de programação funcional, podemos criar facilmente uma função de alta ordem (curry) que retorna uma função de estilo para qualquer propriedade, da seguinte forma:

```
import styled from 'styled-components'

const styledProp = prop => props => ({
  [prop]: props.[prop]
})

const padding = styledProp('padding')
const margin = styledProp('margin')

const spacing = [padding, margin]

const Section = styled.section`
  ${spacing};
`

//Usage
<Section padding="2rem" margin="2rem" />
```

Figura 5: Curry retornando funções de estilo.

O exemplo acima (figura 5) ilustra o *curry styledProp,* que retorna uma nova função de estilo para cada propriedade passada como primeiro parâmetro. Podemos facilmente criar um sistema de espaçamentos (*spacing*) através da combinação das propriedades *margin* e *padding*. Um ponto importante a ser observado é que *styled* automaticamente realiza um flatMap das funções passadas, isso implica que podemos fornecer arrays de qualquer dimensão e ainda receberemos um resultado correto.

2.3 Sistema de Tokens

Conseguimos construir uma função de alta ordem capaz de retornar uma função de estilo para qualquer propriedade passada, mas com esta solução ficamos limitados apenas aos nomes padrões para as propriedades como *width*, *backgroundColor*, *boxShadow* e *maxHeight* que são deveras verborrágicas. Além disso, não existe forma de pré-definir um conjunto de valores para cada propriedade com o fim de garantir a consistência visual dos nossos componentes.

Com o fim de solucionar estes problemas que surgiu a biblioteca *styled-tokens*. Cada token é uma lista de funções de estilo que é construído a partir de uma representação intermediária, sendo este trabalho realizado pela função *createToken*. Essa representação pode ser expressada pela interface a seguir:

```ts
interface DraftToken {
  values: {
    [valueKey: string]: number | string
  }
  propName: {
    [propKey: string]: string
  }
}
```

Figura 6: Representação intermediária de um token

O atributo *propName* refere-se em sua chave quais propriedades o token contemplará, e em seu valor qual a nomenclatura da propriedade no componente. Já *values* representa o conjunto de valores que cada propriedade contida em *propName* pode assumir. Para melhor entendimento vamos dizer que temos três valores possíveis *sm, md e lg* para as props *height, width, maxHeight e maxWidth*:

```ts
import styled from 'styled-components'
import { createToken } from 'styled-tokens'

const draftSize = {
  values: {
    sm: '0.5rem',
    md: '1rem',
    lg: '2rem',
  },
  propName: {
    width: 'w',
    height: 'h',
    maxWidth: 'mw',
    maxHeight: 'mh',
  },
}

const sizeToken = createToken(draftSize)

const Section = styled.section`
  ${sizeToken}
`

function Usage(){
  return (
    <Section w="sm" h="sm" mw="lg" mh="md">
      <h1>Test</h1>
    </Section>
  )
}
```

Figura 7: Criação e uso do token size.

Observando a figura 7, a representação intermediária é compreendida entre as linhas 4 e 16, expressa pela constante *draftSize.* Como um exemplo, ao visualizarmos a linha 11 podemos entender que a propriedade *width* será apelidada de *w*, que ao receber as strings *sm, md e lg* assumirá o valor de *0.5rem, 1rem e 2rem* respectivamente. A função *Usage* (linhas 24-30) demonstra que o mesmo se aplica para todos os outros atributos de *propName* (linhas 10-15).

2.4 Sistema de Variações

Mesmo os tokens são capazes de sanar sozinhos as inconsistências de estilo entre componentes. Por essa razão a biblioteca *styled-tokens* também fornece o suporte à variações, que define um conjunto de estilos que serão aplicadas à um componente dado o valor de certa propriedade. Ilustrando por exemplo, vamos supor um guia de estilo com os seguintes botões:

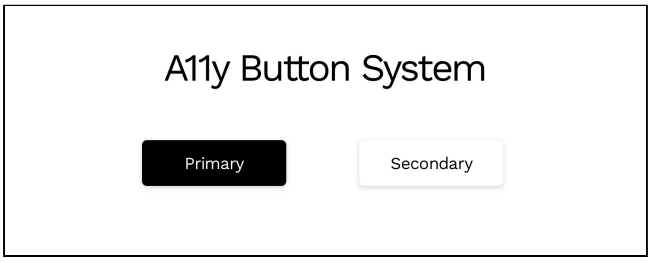

Figura 8: Sistema exemplo de botões.

Observa-se que a fonte, espaçamento, arredondamento de bordas e sombreamento se mantém o mesmo em ambos os botões. Assumindo um sistema de tokens previamente construído, teríamos um componente Button como:

```ts
import { spacing, font, border, shadow } from 'token-system'

const Button = styled.div`
  ${spacing}
  ${font}
  ${border}
  ${shadow}
`

Button.defaultProps = {
  horizontalPadding: 'medium',
  verticalPadding: 'small',
  fontSize: 'medium',
  roundedCorners: 'medium',
  shadow: 'small'
}
```

Figura 9: Componente Button

Apenas as cores de *texto* e *fundo* são alteradas, logo podemos apontar duas variações: *primary* e

secondary. A representação intermediária é definida por:

```
const variation = {
  primary: {
    backgroundColor: 'black',
    color: 'white',
  },
  secondary: {
    backgroundColor: 'white',
    color: 'black',
  },
}
```

Figura 10: Representação intermediária de uma variação

Podemos compor um novo componente de nome *ButtonWithTheme* que utiliza como base o Button previamente definido, tendo como adicional a possibilidade de receber diferentes temas à partir de uma propriedade expressada por *theme*. Novas variações podem ser geradas através de uma chamada à função *createVariation* que tem como parâmetro um objeto contendo duas propriedades: variation (representação intermediária) e name (string que representa o nome que a propriedade assumirá no componente). A seguir, temos um exemplo de uso:

```
import { createVariation } from 'styled-tokens'

const theme = createVariation({ variation, name: 'theme'})

const ButtonWithTheme = styled(Button)`
  ${theme}
`

// usage:
<>
  <ButtonWithTheme theme="primary">Primary</ButtonWithTheme>
  <ButtonWithTheme theme="secondary">Secondary</ButtonWithTheme>
</>
```

Figura 11: Exemplo de uso da função createVariation

De forma similar, se botões pudessem ter diferentes tamanhos e formas deveríamos alterar os tokens de maneira coordenada para garantir consistência.Toda vez que tal regra acontece, devemos utilizar variações e não tokens para representar tais propriedades. Então agora, vamos criar *size e shape* para representar as variantes de tamanho e forma respectivamente, como na figura 12:

```
const size = createVariation({ /*...*/, name: 'size' })
const shape = createVariation({ /*...*/, name: 'shape' })

// theme: primary, secondary
// size: small, medium, large
const ButtonWithThemeSize = styled(ButtonWithTheme)`
  ${size}
`

// theme: primary, secondary
// size: small, medium, large
// shape: rounded, circle, squared, pill
const ButtonWithThemeSizeShape = styled(ButtonWithSize)`
  ${shape}
`

// usage:
<>
  <ButtonWithThemeSize theme="primary" size="small">
    Primary Small Button
  </ButtonWithThemeSize>
  <ButtonWithThemeSizeShape theme="secondary" shape="pill" size="medium">
    Secondary Medium Pill Button
  </ButtonWithThemeSizeShape>
</>
```

Figura 12: Composição de variações

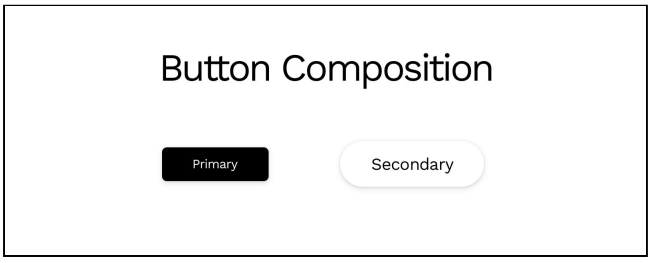

Figura 13: Resultado visual da figura 12.

Embora gerar diversos componentes a partir de composição resulta em uma ótima reusabilidade, visto que podemos utilizar cada um desses de maneira separada e com versatilidade, geramos um problema de inconsistência pois existe a possibilidade de ter botões de mesmo objetivo apresentados de formas distintas no projeto. Além do claro overhead no sistema por conter diversos componentes minúsculos para um mesmo fim, podendo confundir o desenvolvedor sobre qual deles utilizar. Isso pode ser corrigido definido todas as variações de uma só vez, assim podemos refatorar o componente Button da seguinte forma:

```
import styled from 'styled-components'
import { createVariation } from 'styled-tokens'
import { theme, size, shape } from './variations'
import { shadow } from './tokens'

const Button = styled.button`
  ${theme}
  ${size}
  ${shape}
  ${shadow}
`

Button.defaultProps = {
  theme: 'primary',
  size: 'secondary',
  shape: 'rounded',
  shadow: 'small'
}
```

Figura 14: Versão final de Button utilizando variações.

Um outro ponto a ser observado é que diferente dos Tokens, variações não são elementares pois atribuem valores mais de uma propriedade por vez. A fim de garantir que tenhamos variações objetivas e reutilizáveis, é importante definirmos o menor número de propriedades possível, e que estas tenham alta correlação com o nome da variante.

2.5 Adapt-UI

A styled-tokens é uma biblioteca muito objetiva e abstrata. Provê uma forma simples para criar sistemas de tokens e variações, mas não apresenta nenhum destes pronto para uso. Com objetivo de facilitar o acesso à suas funcionalidades surgiu *adapt-ui*, uma biblioteca que contém um sistema de tokens bem definido e componentes estilizados capazes de recebê-los. Vale lembrar que os componentes são muito elementares e flexíveis, com o fim de aumentar o nível de seu reúso.
Os tokens, componentes e hooks disponíveis até o momento estão listados nas tabelas 1 e 2 logo abaixo:

Token, Variação	Descrição
colors	Define uma alternativa de cores para bordas, fundo e texto que são mais acessíveis (a11y) em relação às padrão do browser.
display	Propriedade *display* do css.
elevation	Valores default para sombreamento, renomeando a prop *box-shadow*
flexbox	Controle de todas as funções do alinhamento *flexbox* do css.
float	Propriedade *float* do css.
position	Propriedade *position* do css.
scroll	Variação para definir scrolls no eixo x, y ou ambos.
size	Valores default em porcentagens e rems para as propriedades *height*, *maxHeight*, *minHeight*, *width*, *maxWidth*, *minWidth*.
spacing	Valores default para as propriedades *margin* e *padding*.
text	Permite o controle das propriedades textuais *align*, *transform* e *decoration*.

Tabela 1: Tokens definidos pelo adapt-ui

Componente, Hook	Descrição
Box	Componente elementar o qual todos os outros são compostos. Define um bloco capaz de representar qualquer elemento do DOM.
Clickable	Capaz de representar qualquer elemento clicável, que é por padrão um *button.*
Collapsible	Envolve itens que podem ser escondidos por alguma ação.
useCollapsible	Hook que controla o estados de Collapsible e Collapsible Toggle
CollapsibleToggle	Elemento capaz de ativar ou desativar a presença de elementos filhos de Collapse.
Spinner	Spinner animado altamente customizável.
Flexible	Box que automaticamente tem o token **flexbox** aplicado.
Scrollable	Box que automaticamente tem a variação **scroll** aplicada.
Heading	Texto para títulos e automaticamente utiliza o

	token **text**.
Paragraph	Texto para parágrafos e seções. Automaticamente utiliza o token **text**.

Tabela 2: Componentes e hooks do adapt-ui

2.6 Tecnologias Relacionadas

Para poder compilar todo o código e exportá-lo como biblioteca, tanto para o *adapt-ui* quando *styled-tokens*, foi utilizado o module bundler *rollup*. Utilizar bundlers é muito importante principalmente por causa do *tree-shake*, que importa apenas o código utilizado de cada dependência, desta forma mesmo utilizando de outras bibliotecas - seu código completo será adicionado na build final.

Para realizar o showcase de componentes foi utilizado o storybook, que é uma ferramenta que permite o desenvolvimento de componentes React, Vue e Angular de forma isolada. Isso é importante pois já temos um endereço de documentação iterativa e um ambiente de testes de uma só vez.

Em ambas as bibliotecas foi utilizado typescript, que pode ser visto como um superset de javascript, e fornece uma série de vantagens. As características determinantes que levaram à esta decisão são:

1. O código torna-se mais fácil de ler, entender e debugar.
2. Provê diversas ferramentas para IDEs que aumentam a produtividade.
3. Facilidade no rastreio dos erros no código de maneira eficiente, através da checagem estática de tipos, resultando em uma economia de tempo desenvolvimento
4. Sistema de tipagem poderosos como interfaces, types, generics, union types, permitem estender as verificações de tipo ao usuário da biblioteca.

3. Bibliotecas em uso

Como estudo de caso foi criado um sistema de tokens a partir do *styled-tokens* - o *aurum-system*. Por meio de uma função geradora, realiza a criação de tokens que utilizando como base a proporção áurea, que pode ser aproximada por fibonacci. Todos os detalhes estão documentados no repositório *matheusps/aurum-system.*

Outro estudo a ser observado é o adapt-ui, que também usa os styled-tokens para ajudar a manter sua consistência. Como foi desenvolvido utilizando o storybook temos um ótimo showcase para seus componentes, disponível no endereço: https://adapt-ui.netlify.com/.

Figura 15: Storybook do adapt-ui

4. LIÇÕES APRENDIDAS

Este bloco apresenta algumas lições aprendidas e desafios encontrados durante o desenvolvimento das bibliotecas *adapt-ui* e *styled-tokens*.

4.1 Generalizar facilita o desenvolvimento de componentes

Esse é de longe o conselho que eu mais gostaria de ter quando comecei a desenvolver o *adapt-ui*. Ao criar componentes que serão utilizados em múltiplos contextos, é melhor mantê-los amplos e abertos a vários usos e interpretações. De início desejava que os componentes fossem limitados e muito opinativos por assumir que meus usuários gostariam que todas as decisões de design fossem completamente abstraídas deles.

Um bom exemplo foi o desenvolvimento do componente *Button*. Eu tinha receio que os usuários criassem botões que em um contexto geral seriam considerados estranhos. Então, de início o pensei apenas como um bloco que contém apenas um texto ao centro e onde os espaçamentos horizontais são duas vezes maiores que verticais, além disso atribuí por padrão os cursores *pointer* e *not-allowed* para os estados *normal* e *disabled* respectivamente. Dias depois, foi necessário um botão que pudesse ter ícones à esquerda e direita - logo adaptei o componente existente para estes casos. Situações similares foram acontecendo com bordas, cor de fundo, texto, etc, até um momento em que me ocorreu que os usuários deveriam passar o que dejar com o mínimo de interferência possível, ou seja, no final acabei apagando toda a lógica pré-escrita e frustrado por minhas próprias restrições agressivas.

Pode-se concluir que tentar prever o futuro na tentativa de adivinhar todos os cenários possíveis de

bom e mau uso de um componente apenas dificultou o meu desenvolvimento com preocupações que podem nunca ocorrer, e inclusive levou à imposição de restrições equivocadas que precisaram ser removidas. Logo, o melhor a ser feito é iniciar o desenvolvimento com um componente bem flexível, genérico e bem documentado e restrições vão sendo aplicadas à medida que sejam absolutamente necessárias.

4.2 Componentes Experimentais tornam a biblioteca melhor

No meu contexto foi incluída a tag *Experimental* ao início do nome do componente, para que você saiba que ainda não está pronta para ser usada em qualquer aplicativo que seja. No começo, tratei a biblioteca como um pequeno filho, não queria que ninguém a visualizasse de outra forma que não a perfeição absoluta. Isso foi um tremendo erro, pois impedia que eu escrevesse componentes hipotéticos a fim de combiná-los para chegar à um mais objetivo. Pensar no storybook como um estúdio de arte e não um museu tornou tudo mais fácil, prático e acabei chegando à componentes mais ricos. Manter um pequeno caos organizado faz bem para a criatividade, inclusive em um caso de trabalho em equipe, permite que todos possam opinar nas etapas experimentais, gerando componentes melhores e mais úteis ao fim do processo.

4.3 Trabalhos Futuros

Como qualquer ferramenta ou biblioteca trás uma curva de aprendizado, um dos objetivos é diminuí-la através da melhora da documentação existente incluindo mais exemplos interativos com diferentes casos de uso, para que os usuários consigam compreender melhor e encaixá-la no seu contexto.

Aumentar a confiabilidade dos componentes e sistema de tokens com testes visuais e de unidade. Isso é necessário para que atualizações sejam realizadas com mais confiança e garantia de não quebrar componentes antigos para os usuários. Os componentes já são desenvolvidos em isolamento, mas isto não dispensa a adição de testes visuais para que a consistência de design seja mantida mediante qualquer tipo de alteração.

Especificamente falando do styled-tokens, as variações deveriam ser criadas a partir de tokens para garantir consistência e otimizar o reúso juntamente com a adição de tokens reativos à diferentes temas, e resoluções de tela.

5. REFERÊNCIAS

[1] The Six Steps For Justifying Better UX link: https://www.forrester.com/report/The+Six+Steps+For+Justifying+Better+UX/-/E-RES117708#

[2] E-COMMERCE FATURA R$53,2 BILHÕES EM 2018, ALTA DE 12% link: https://www.nielsen.com/br/pt/insights/article/2019/e-commerce-fatura-53-bilhoes-em-2018-alta-de-12-por-cento

[3] Styled components documentation link: https://www.styled-components.com/docs

[4] The Bottom Line: Why Good UX Design Means Better Business. Link: https://www.forbes.com/sites/forbesagencycouncil/2017/03/23/the-bottom-line-why-good-ux-design-means-better-business/#6922ac592396

[5] StoryBook documentation. Link: https://storybook.js.org/