CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

Márcio Welben Montenegro Mariani Alves

DESENVOLVIMENTO DE UM DESIGN SYSTEM SIMPLIFICADO COM REACT, TYPESCRIPT E MATERIAL-UI: METODOLOGIAS E IMPLEMENTAÇÃO

João Pessoa, PB
2024

Márcio Welben Montenegro Mariani Alves

DESENVOLVIMENTO DE UM DESIGN SYSTEM SIMPLIFICADO COM REACT, TYPESCRIPT E MATERIAL-UI: METODOLOGIAS E IMPLEMENTAÇÃO

Trabalho de Conclusão de Curso apresentado ao curso de engenharia da computação do Centro de Informática, da Universidade Federal da Paraíba, como requisito para a obtenção do grau de Bacharel em Engenharia da Computação.

Orientador: Prof. Dr. Carlos Eduardo Coelho Freire Batista

João Pessoa, PB

2024

Catalogação na publicação
Seção de Catalogação e Classificação

A474d Alves, Marcio Welben Montenegro Mariani.
Desenvolvimento de um design system simplificado com react, typescript e material-ui: metodologias e implementação / Marcio Welben Montenegro Mariani Alves. - João Pessoa, 2024.
60 f. : il.

Orientação: Carlos Eduardo Batista.
TCC (Graduação) - UFPB/CI.

1. Design system. 2. ReactJS. 3. TypeScript. 4. Desenvolvimento de interfaces. 5. Desenvolvimento front-end. I. Batista, Carlos Eduardo. II. Título.

UFPB/CI CDU 004.42

Elaborado por Michelle de Kássia Fonseca Barbosa - CRB-738

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

Trabalho de Conclusão de curso de Engenharia da computação intitulado **DESENVOLVIMENTO DE UM DESIGN SYSTEM SIMPLIFICADO COM REACT, TYPESCRIPT E MATERIAL-UI: METODOLOGIAS E IMPLEMENTAÇÃO** de autoria de Márcio Welben Montenegro Mariani Alves, aprovada pela banca examinadora constituída pelos seguintes professores:

Prof. Dr. Carlos Eduardo Coelho Freire Batista
Universidade Federal da Paraíba - UFPB

Profa. Dra. Danielle Rousy Dias Ricarte
Universidade Federal da Paraíba - UFPB

Profa. Dra. Natasha Correia Queiroz Lino
Universidade Federal da Paraíba - UFPB

João Pessoa, 12 de Maio de 2024

Centro de Informática, Universidade Federal da Paraíba
Rua dos Escoteiros, Mangabeira VII, João Pessoa, Paraíba, Brasil CEP: 58038-600
Fone: 55 + (83) 3216 7093

Dedico este trabalho à minha família e, em especial, à minha esposa Tawany Mariani, por seu apoio incondicional e incentivo constante durante todo o curso. Sua paciência e amor foram fundamentais para a realização deste projeto.

AGRADECIMENTOS

Chegar ao final desta jornada acadêmica é um momento de grande realização e alegria, e muitos foram os que contribuíram para que eu pudesse alcançar este marco importante. Gostaria de expressar minha profunda gratidão a todos que me apoiaram e incentivaram ao longo desta caminhada.

Aos meus queridos tio Alex e Valnia, agradeço por terem me acolhido em sua casa durante todo o meu curso, proporcionando-me um ambiente de paz e apoio constante. Sua orientação e carinho foram fundamentais para minha formação.

Meus pais, cujo suporte incondicional e sacrifícios foram cruciais para que eu pudesse trilhar meu caminho acadêmico desde a infância até a graduação, merecem meu mais sincero agradecimento. Seu esforço e dedicação são inspirações que levarei comigo para sempre.

Minha tia Angélica e minha avó Nevolanda, agradeço pelo apoio emocional e incentivo contínuos. Vocês foram fontes de força e coragem, sempre me lembrando da importância de perseverar e acreditar em meus sonhos.

À minha esposa Tawany Mariani, minha eterna companheira e confidente, agradeço por estar ao meu lado em todos os momentos, compartilhando comigo cada desafio e cada vitória. Sua paciência, amor e apoio inabaláveis tornaram possível a conclusão deste ciclo.

Por fim, mas não menos importante, agradeço aos meus professores e orientador, por toda a atenção, ensino e paciência. Suas orientações e conhecimentos foram essenciais para o meu desenvolvimento acadêmico e pessoal. Sou imensamente grato por tudo que aprendi com vocês.

A todos, meu muito obrigado por fazerem parte desta jornada e contribuírem para a realização deste sonho.

RESUMO

Este trabalho aborda a criação de um sistema de design simplificado utilizando ReactJS, TypeScript e Material-UI (MUI), com o objetivo de desenvolver interfaces de usuário consistentes e eficazes. O ReactJS é empregado para criar a camada de interação do usuário, proporcionando uma estrutura modular e reutilizável, enquanto o TypeScript adiciona tipagem estática ao JavaScript, aumentando a robustez e a manutenção do código enquanto que o Material-UI (MUI) oferece componentes prontos e personalizáveis, facilitando a criação de interfaces atraentes e funcionais. A análise da eficácia deste sistema de design foi conduzida por meio de sua aplicação em uma Single Page Application (SPA) desenvolvida com ReactJS e TypeScript. De acordo com a análise realizada neste trabalho, esta abordagem oferece uma coleção padronizada de componentes reutilizáveis que asseguram uma experiência de usuário coesa e uma identidade visual consistente, aspectos cruciais para a percepção positiva da marca. Além disso, promovem uma maior eficiência no desenvolvimento, uma vez que a existência de elementos predefinidos e testados reduz o tempo necessário para lançar produtos no mercado, permitindo que equipes foquem em inovação em vez de desenvolver repetidamente soluções básicas. A manutenção simplificada, outro benefício chave, permite que atualizações em um componente sejam automaticamente aplicadas a todos os projetos que o utilizam, minimizando a possibilidade de erros e inconsistências. Por fim, a escalabilidade é significativamente facilitada, já que o sistema pode se expandir de maneira controlada à medida que a empresa cresce, evitando a complexidade e os custos associados à adaptação de cada novo projeto de forma independente. Portanto, Sistemas de Design representam uma abordagem estratégica superior para organizações que buscam otimizar seus processos de desenvolvimento e manutenção de software.

Palavras-chave: Design System, ReactJS, TypeScript, Desenvolvimento de Interfaces, Desenvolvimento Front-end.

ABSTRACT

This paper discusses the creation of a simplified design system using ReactJS, TypeScript, and Material-UI (MUI), aimed at developing consistent and effective user interfaces. ReactJS is used to build the user interaction layer, offering a modular and reusable structure, while TypeScript enhances JavaScript by adding static typing, thus improving the robustness and maintainability of the code. Material-UI (MUI) provides customizable, ready-to-use components that ease the creation of attractive and functional interfaces. The effectiveness of this design system was evaluated through its implementation in a Single Page Application (SPA) developed using ReactJS and TypeScript. According to the analysis conducted in this study, this approach provides a standardized collection of reusable components that ensure a cohesive user experience and consistent visual identity, crucial for positive brand perception. Moreover, it promotes greater development efficiency, as the presence of predefined and tested elements reduces the time needed to launch products on the market, allowing teams to focus on innovation rather than repeatedly developing basic solutions. Simplified maintenance, another key benefit, enables updates to a component to be automatically applied across all projects that use it, minimizing the likelihood of errors and inconsistencies. Lastly, scalability is significantly facilitated, as the system can expand in a controlled manner as the company grows, avoiding the complexity and costs associated with adapting each new project independently. Thus, Design Systems represent a superior strategic approach for organizations seeking to optimize their software development and maintenance processes.

Keywords: Design System, ReactJS, TypeScript, User Interface Development, Front-end Development.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS

AJAX: *Asynchronous JavaScript and XML*

API: *Application Programming Interface*

CSS: *Cascading Style Sheets*

DOM: *Document Object Model*

HTML: *HyperText Markup Language*

JSX: *JavaScript XML*

MUI: *Material-UI*

SEO: *Search Engine Optimization*

SPA: *Single Page Application*

SUMÁRIO

1 INTRODUÇÃO

O desenvolvimento de interfaces de usuário (UI) tem evoluído significativamente nos últimos anos, particularmente no que diz respeito à consistência e reutilização de componentes visuais (SATZINGER; JACKSON; BURD, 2018). Essa evolução trouxe à tona o conceito de *Design System*, uma coleção de diretrizes, componentes e ferramentas que visa unificar e padronizar a experiência do usuário em uma aplicação ou conjunto de aplicações.

A consistência proporcionada por um *Design System*, que facilita a manutenção e evolução das interfaces e garante uma experiência uniforme ao usuário final, é sua principal vantagem (SHNEIDERMAN; PLAISANT, 2004). Ao fornecer uma "única fonte de verdade", um *Design System* pode ser utilizado por equipes de design e desenvolvimento para alcançar um produto consistente, conforme destacado por Frost (2016).

Neste trabalho, abordamos o desenvolvimento de um *Design System* utilizando as tecnologias ReactJS, TypeScript e Material-UI, explorando metodologias e práticas recomendadas para sua implementação eficaz. A consistência proporcionada por um Design System, que facilita a manutenção e evolução das interfaces e garante uma experiência uniforme ao usuário final, é sua principal vantagem. Utilizando o conceito de "Virtual DOM", o ReactJS otimiza a atualização de componentes na tela, garantindo alta performance mesmo em aplicações complexas (STEFANOV, 2011).

O TypeScript, um superset[1]de JavaScript desenvolvido pela Microsoft, adiciona tipagem estática ao idioma, proporcionando um desenvolvimento mais seguro e robusto. A tipagem estática é crucial para identificar erros em tempo de compilação, proporcionando um desenvolvimento mais seguro e robusto. Além disso, o TypeScript facilita a manutenção e escalabilidade do código, especialmente em projetos grandes (FAIN; MOISEEV, 2020).

Por sua vez, o Material-UI é uma biblioteca de componentes ReactJS que implementa as diretrizes de design do Google Material Design. Esta biblioteca fornece uma ampla variedade de componentes prontos para uso que seguem os princípios de Material Design, permitindo que os desenvolvedores criem interfaces consistentes e modernas de maneira rápida e eficiente (VESSELOV; DAVIS, 2019).

Existem várias etapas essenciais envolvidas na implementação de um Design System com ReactJS, TypeScript e Material-UI. O primeiro passo é definir os princípios que guiarão

[1] O termo "superset" refere-se ao fato de que TypeScript é uma extensão do JavaScript, o que significa que ele incorpora todas as funcionalidades do JavaScript e adiciona novas características.

o *Design System*, incluindo a escolha da paleta de cores, tipografia, espaçamento e outras diretrizes visuais que serão seguidas por todos os componentes (KHOLMATOVA, 2017). Com o ReactJS, é possível criar componentes modulares e reutilizáveis, projetados para serem independentes e facilmente integráveis em diferentes partes da aplicação (STEFANOV, 2011).

A utilização do TypeScript é fundamental para garantir a correta utilização dos componentes, respeitando as tipagens definidas, o que contribui para aumentar a segurança do código e facilitar a manutenção futura (FAIN; MOISEEV, 2020). Além disso, a biblioteca Material-UI facilita a criação de componentes que seguem os padrões de design do Material Design, oferecendo componentes estilizados e prontos para uso, acelerando o desenvolvimento e garantindo consistência visual (VESSELOV; DAVIS, 2019).

Ao longo deste trabalho, exploramos as metodologias e práticas recomendadas para a implementação eficaz de um *Design System* utilizando ReactJS, TypeScript e Material-UI, destacando como essas tecnologias se complementam para criar interfaces de usuário escaláveis, consistentes e eficientes. A utilização do TypeScript é fundamental para garantir a correta utilização dos componentes, respeitando as tipagens definidas, o que contribui para aumentar a segurança do código e facilitar a manutenção futura (SHNEIDERMAN; PLAISANT, 2004; FROST, 2016; STEFANOV, 2011).

1.1 Tema

Existem diversos tipos de sistemas de design, desde sistemas proprietários desenvolvidos por grandes empresas, como o Material Design da Google[2], até soluções personalizadas criadas para atender necessidades específicas de projetos. Cada um visa promover uma base coesa e escalável para o desenvolvimento de interfaces (FROST, 2016).

Este trabalho propõe a criação de um sistema de design simplificado utilizando ReactJS, TypeScript e Material-UI. O objetivo é desenvolver uma base modular de componentes essenciais, garantindo consistência visual e eficiência no desenvolvimento. A implementação e avaliação desse sistema em uma aplicação de página única (SPA) visa demonstrar sua eficácia em termos de consistência, manutenção e escalabilidade (“MUI: The ReactJS component library you always wanted,” [s.d.]).

[2] Disponível em: https://m3.material.io/

1.2 Objetivos Gerais

Desenvolver um sistema de design simplificado utilizando ReactJS, TypeScript e Material-UI (MUI) para criar interfaces de usuário consistentes, eficientes e escaláveis. O sistema será implementado e avaliado em uma *Single Page Application* (SPA), com o intuito de demonstrar sua eficácia em termos de consistência visual, eficiência no desenvolvimento, manutenção do código e escalabilidade. O projeto inclui o desenvolvimento de uma coleção de componentes reutilizáveis, implementação de tipagem estática, documentação via Storybook, aplicação do sistema em uma SPA e a avaliação e documentação detalhada do processo.

1.4 Estrutura do Trabalho

Este trabalho está organizado em cinco seções. A primeira seção apresenta uma introdução ao tema, definindo a problemática do desenvolvimento de interfaces de usuário consistentes e eficientes, além dos objetivos gerais e específicos do projeto. Na segunda seção, são discutidos os conceitos teóricos fundamentais, incluindo uma revisão bibliográfica sobre ReactJS, TypeScript, Material-UI (“MUI: The ReactJS component library you always wanted,” [s.d.]), estabelecendo o embasamento teórico necessário. A terceira seção detalha o desenvolvimento do *Design System*, abordando a criação dos componentes essenciais, a integração com TypeScript, e sua aplicação em uma Single Page Application (SPA). A quarta seção descreve os métodos de avaliação do *Design System*, focando na análise da consistência visual, eficiência no desenvolvimento, manutenção do código e escalabilidade e por fim, a quinta seção apresenta as conclusões do estudo, propondo melhorias e sugerindo direções para trabalhos futuros.

2 REFERENCIAL TEÓRICO

Nesta seção, serão discutidos os fundamentos teóricos que embasam o desenvolvimento do *Design System* proposto. Inicialmente, serão apresentados os conceitos e as vantagens de utilizar *Design Systems* no desenvolvimento de interfaces de usuário. Em seguida, serão detalhadas as tecnologias específicas escolhidas para este projeto: ReactJS, TypeScript, Material-UI (“MUI: The ReactJS component library you always wanted,” [s.d.]).

A revisão teórica buscará estabelecer uma base sólida para a compreensão e aplicação prática dessas tecnologias no contexto de um *Design System* simplificado.

2.1 Desenvolvimento Web

O desenvolvimento web refere-se à construção e manutenção de websites, desde páginas estáticas simples até aplicações web complexas. HTML (HyperText Markup Language) é a linguagem padrão usada para estruturar o conteúdo na web. Criado por Tim Berners-Lee em 1991, o HTML permite definir a estrutura e o layout de uma página web através de uma série de elementos combinados representados por tags[3] (DUCKETT, 2014).

Facilitando a acessibilidade e a indexação por motores de busca, a semântica do HTML, que descreve o significado do conteúdo da página, é uma característica fundamental. Tags como <*header*>, <*article*>, <*section*> e <*footer*> ajudam a estruturar o conteúdo de maneira lógica e intuitiva, melhorando a experiência do usuário e a otimização para motores de busca (ROBSON; FREEMAN, 2012).

Complementado por CSS (Cascading Style Sheets) para estilização e JavaScript para interatividade, o HTML forma a tríade essencial do desenvolvimento front-end (MARCOTTE, 2011). A evolução do HTML, especialmente com o HTML5 lançado em 2014, trouxe novas APIs[4] e elementos que facilitam a criação de conteúdos multimídia e aplicações web mais interativas sem a necessidade de plugins externos (PILGRIM, 2010).

Frameworks e bibliotecas modernas de desenvolvimento web, como ReactJS, Angular e Vue.js, utilizam HTML como base para criar componentes reutilizáveis e aplicações de página única (SPA), proporcionando uma estrutura mais modular e eficiente para o desenvolvimento de interfaces de usuário (SATZINGER; JACKSON; BURD, 2018).

Apesar de suas muitas vantagens, o desenvolvimento web com HTML enfrenta desafios como a compatibilidade entre navegadores e a segurança das aplicações, que requerem práticas de desenvolvimento seguras para proteger os dados dos usuários (SHNEIDERMAN; PLAISANT, 2004).

---

[3] Elementos de marcação utilizados no HyperText Markup Language (HTML) para estruturar e formatar conteúdo em páginas da web. Cada tag HTML é representada por um nome envolvido em colchetes angulares (< >) e geralmente vem em pares, com uma tag de abertura (<tag>) e uma tag de fechamento (</tag>)

[4] Conjuntos de definições e protocolos que permitem que diferentes sistemas de software se comuniquem entre si.

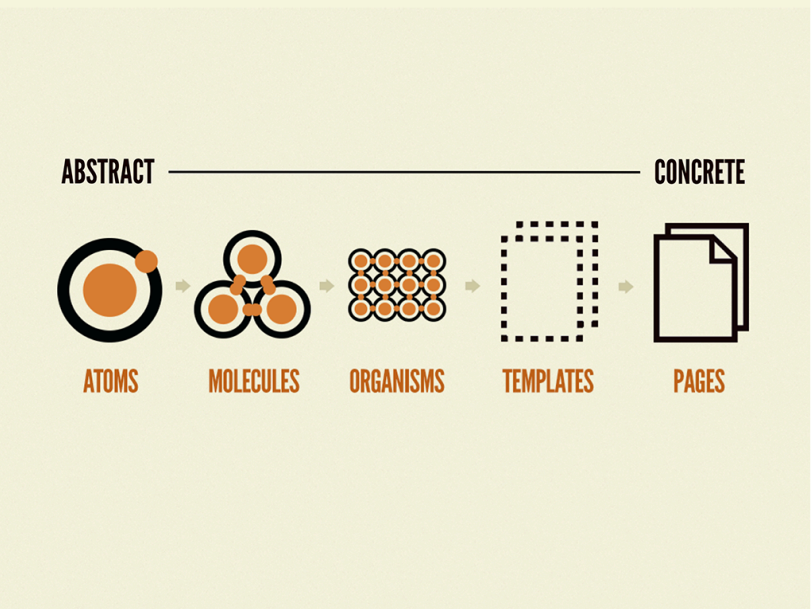
Em conclusão, o HTML, servindo como a espinha dorsal da estrutura e do conteúdo das páginas web, continua sendo essencial no desenvolvimento web. Sua combinação com CSS e JavaScript permite a criação de interfaces de usuário ricas e interativas, enquanto as boas práticas garantem acessibilidade, performance e segurança (ROBBINS, 2018; MARCOTTE, 2011).

2.2 Atomic Design

Atomic Design é uma metodologia de design de interfaces proposta por Brad Frost, que visa criar sistemas de design consistentes e escaláveis. Esta metodologia é baseada no conceito de decomposição de interfaces em componentes menores e reutilizáveis, similar à maneira como os átomos se combinam para formar moléculas e, eventualmente, organismos maiores. Este processo de decomposição é fundamental para entender a essência do Atomic Design e como ele se aplica ao desenvolvimento de interfaces modernas.

A metodologia do Atomic Design (Figura 1) categoriza as interfaces de usuário em cinco níveis distintos, nomeadamente: átomos, moléculas, organismos, templates e páginas. Cada nível representa um grau de complexidade crescente, onde os componentes de níveis inferiores são combinados para formar interfaces completas. Os átomos, que são os elementos básicos da interface e incluem componentes como botões, inputs e ícones, não podem ser decompostos em partes menores. Já as moléculas são combinações de átomos que formam elementos de interface mais complexos, como um campo de formulário composto por um input, um label e um botão. Por sua vez, os organismos são combinações de moléculas e átomos que formam seções distintas da interface, tais como um cabeçalho ou um rodapé. Os templates, que são estruturas de página, organizam os organismos em layouts e fornecem um contexto para a interação dos componentes (FROST, 2013; GONZALEZ et al., 2022).

Figura 1 — Os 5 estágios do Atomic Design



Fonte: FROST (2016)

O Atomic Design, proposto por Brad Frost em 2013, surgiu como uma resposta à crescente complexidade do design de interfaces e à necessidade de sistemas de design escaláveis e de fácil manutenção. A metodologia permite que designers e desenvolvedores trabalhem de maneira mais colaborativa e eficiente, garantindo que todos os componentes de uma interface sejam consistentes e reutilizáveis. Esta abordagem promove a criação de componentes uniformes e reutilizáveis, garantindo uma experiência de usuário consistente em todas as partes da aplicação (FROST, 2016). Além disso, facilita a adição de novos componentes e funcionalidades ao sistema sem a necessidade de redesenhar toda a interface, simplificando a manutenção e permitindo que os componentes sejam atualizados de forma independente e reutilizados em diferentes contextos.

O Atomic Design estabelece uma linguagem comum entre designers e desenvolvedores, facilitando a comunicação e a colaboração entre as equipes. A abordagem modular permite o desenvolvimento paralelo de diferentes partes da interface, acelerando o processo de desenvolvimento. O Atomic Design, ao decompor a interface em componentes menores e reutilizáveis, não só facilita a manutenção e a colaboração, mas também melhora a eficiência no desenvolvimento de produtos digitais (GONZALEZ et al., 2022). Esta metodologia promove um desenvolvimento mais ágil e organizado, garantindo que os sistemas de design sejam robustos, consistentes e escaláveis (FROST, 2016).

A implementação do Atomic Design, que representa uma abordagem inovadora e eficiente para o design de interfaces de usuário. Esta metodologia, que promove um

desenvolvimento mais ágil e organizado, garante que os sistemas de design sejam robustos, consistentes e escaláveis (FROST, 2016; GONZALEZ et al., 2022).

2.3 Javascript, Typescript e ReactJS

No desenvolvimento web moderno, o JavaScript destaca-se como uma linguagem essencial para criar interfaces dinâmicas e interativas, facilitando a implementação de comportamentos que tornam as aplicações web mais responsivas. Sua capacidade de execução direta nos navegadores, sem a necessidade de compilação prévia, é uma característica valorizada, conforme McFarland (2014) descreve ao discutir a manipulação do Document Object Model (DOM) e a utilização de AJAX para comunicações assíncronas com servidores. A flexibilidade do JavaScript em suportar diversos estilos de programação, incluindo orientação a objetos e programação funcional, contribui significativamente para essa versatilidade.

Contudo, o JavaScript enfrenta desafios, especialmente em projetos de grande escala, devido à sua tipagem dinâmica que pode introduzir erros difíceis de detectar (CROCKFORD, 2008). Para mitigar esses problemas, a Microsoft desenvolveu o TypeScript, um superset de JavaScript que adiciona tipagem estática opcional e outras funcionalidades avançadas, lançado em 2012. Esta inovação visa melhorar a robustez e a manutenção do código, permitindo a detecção de erros em tempo de compilação (FAIN; MOISEEV, 2020). O TypeScript também suporta os mais recentes recursos do JavaScript, incluindo módulos, classes e async/await, além de permitir a transpilação para versões mais antigas do JavaScript, garantindo compatibilidade com uma ampla gama de navegadores (MICROSOFT CORPORATION, [s.d.]).

A integração do TypeScript com ferramentas modernas de desenvolvimento aprimora ainda mais a eficiência e a produtividade do processo de desenvolvimento. Esta combinação de JavaScript e TypeScript é especialmente benéfica em projetos que empregam frameworks e bibliotecas modernas como ReactJS, Angular e Vue.js. O TypeScript melhora a robustez e a manutenção do código ao definir tipagens para componentes e garantir seu uso correto, contribuindo para o desenvolvimento de aplicações web escaláveis, seguras e eficientes. Portanto, estas tecnologias, ao serem utilizadas em conjunto, representam um avanço significativo para o desenvolvimento web, realçando a importância dessas ferramentas na melhoria contínua dos processos de desenvolvimento de software (OVCHINNIKOVA, 2022).

Complementando as capacidades do JavaScript e do TypeScript no desenvolvimento web, o ReactJS se destaca como uma biblioteca JavaScript de código aberto criada pelo Facebook e lançada em 2013. Desenvolvido para resolver problemas de desempenho em grandes aplicações web, como o Facebook Ads, o ReactJS otimiza a manipulação do Document Object Model (DOM)[5] (FLANAGAN,2020; STEFANOV, 2021). O DOM é uma representação hierárquica da estrutura HTML de um documento, permitindo a interação dinâmica com o conteúdo da página.

Uma das principais características do ReactJS é a componentização, que facilita a criação de componentes reutilizáveis e modulares. Essa abordagem simplifica a manutenção e a escalabilidade do código, permitindo aos desenvolvedores dividir a interface em partes menores e independentes, promovendo a reutilização de código e a organização do projeto (FEDOSEJEV, 2023; ANTHONY; NATHANIEL; ARI, 2017).

O ReactJS utiliza o Virtual DOM, uma cópia leve do DOM real, que permite atualizar apenas os elementos que mudaram, em vez de redesenhar toda a árvore DOM. Essa técnica resulta em um desempenho superior, essencial para aplicações que requerem alta interatividade e eficiência (REACTJS, 2021; FLANAGAN, 2020).

Outro aspecto importante do ReactJS é o uso do JSX, uma extensão de sintaxe que permite escrever HTML dentro do JavaScript. Isso torna o código mais intuitivo e fácil de entender, aproximando-se da estrutura dos componentes da interface de usuário (REACTJS, 2021). A familiaridade com HTML facilita a adoção do ReactJS por desenvolvedores web (DUCKETT, 2014).

O ecossistema do ReactJS inclui ferramentas poderosas como Redux, para gerenciamento de estado, e ReactJS Router, para navegação entre páginas em aplicações de página única (SPA). O Redux centraliza o estado da aplicação, tornando o gerenciamento de dados mais previsível e fácil de analisar. Já o ReactJS Router permite a navegação entre diferentes vistas da aplicação, mantendo a experiência de usuário fluida e responsiva (FREEMAN, 2019; ANTHONY; NATHANIEL; ARI, 2017).

Apesar de suas muitas vantagens, o ReactJS apresenta algumas desvantagens. A curva de aprendizado pode ser íngreme para iniciantes, especialmente devido ao uso de conceitos como JSX e a necessidade de compreender o gerenciamento de estado (REACTJS, 2021; ANTHONY; NATHANIEL; ARI, 2017). Além disso, a rápida evolução da biblioteca pode

[5] Estrutura hierárquica que representa a estrutura de documentos HTML e XML, permitindo a manipulação de conteúdo e estilo por scripts.

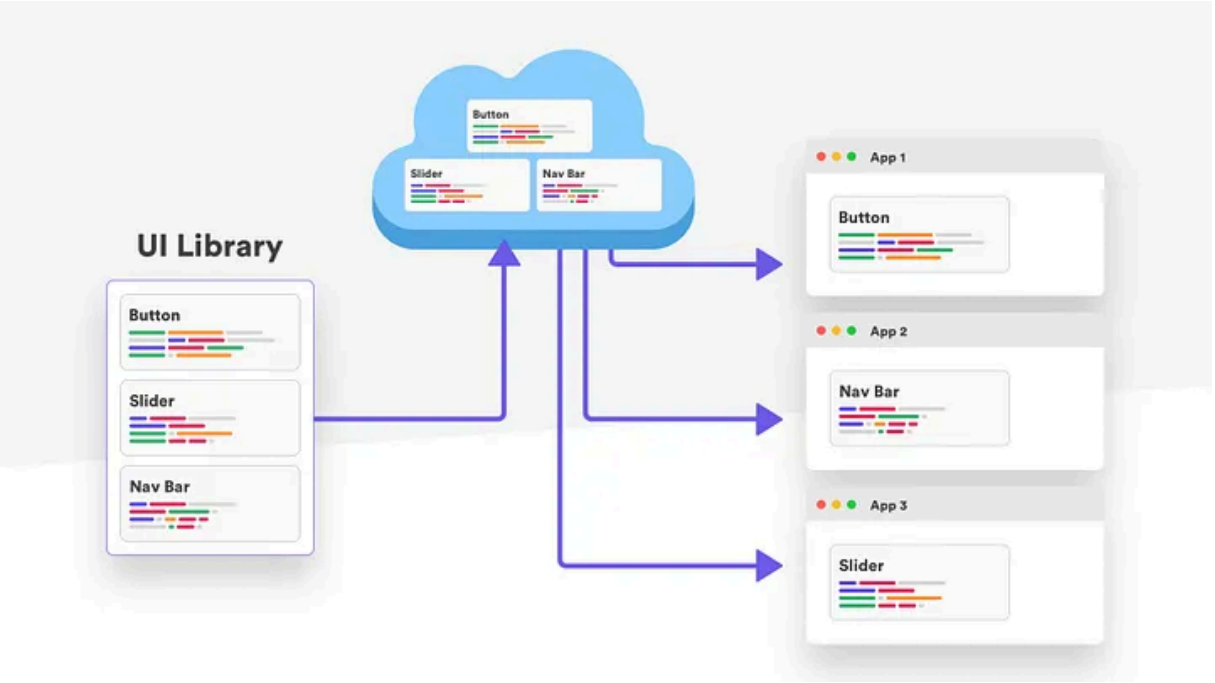
levar a documentação desatualizada, exigindo adaptação constante às novas versões (FREEMAN, 2019).

Em conclusão, o uso do ReactJS no desenvolvimento de sistemas de design simplificados proporciona uma abordagem moderna e eficiente para a criação de interfaces de usuário escaláveis e manuteníveis. A combinação de suas características, como componentização, Virtual DOM e ecossistema robusto, torna o ReactJS uma escolha ideal para este projeto. O ReactJS não apenas melhora a experiência do desenvolvedor, mas também proporciona uma performance superior e um código mais organizado (FLANAGAN, 2020; FREEMAN, 2019; REACTJS, 2021).

2.5 *Design System* e Material-UI (MUI)

A consistência visual e a eficiência na criação de interfaces de usuário são essenciais no desenvolvimento web moderno. Um *Design System* é uma coleção de diretrizes, componentes e ferramentas que visam unificar e padronizar a experiência do usuário em uma aplicação ou conjunto de aplicações, promovendo a reutilização de componentes e facilitando a manutenção e evolução das interfaces (SATZINGER; JACKSON; BURD, 2018) como exemplificado na Figura 2.

Figura 2 — Visão simplificada de um Design System.

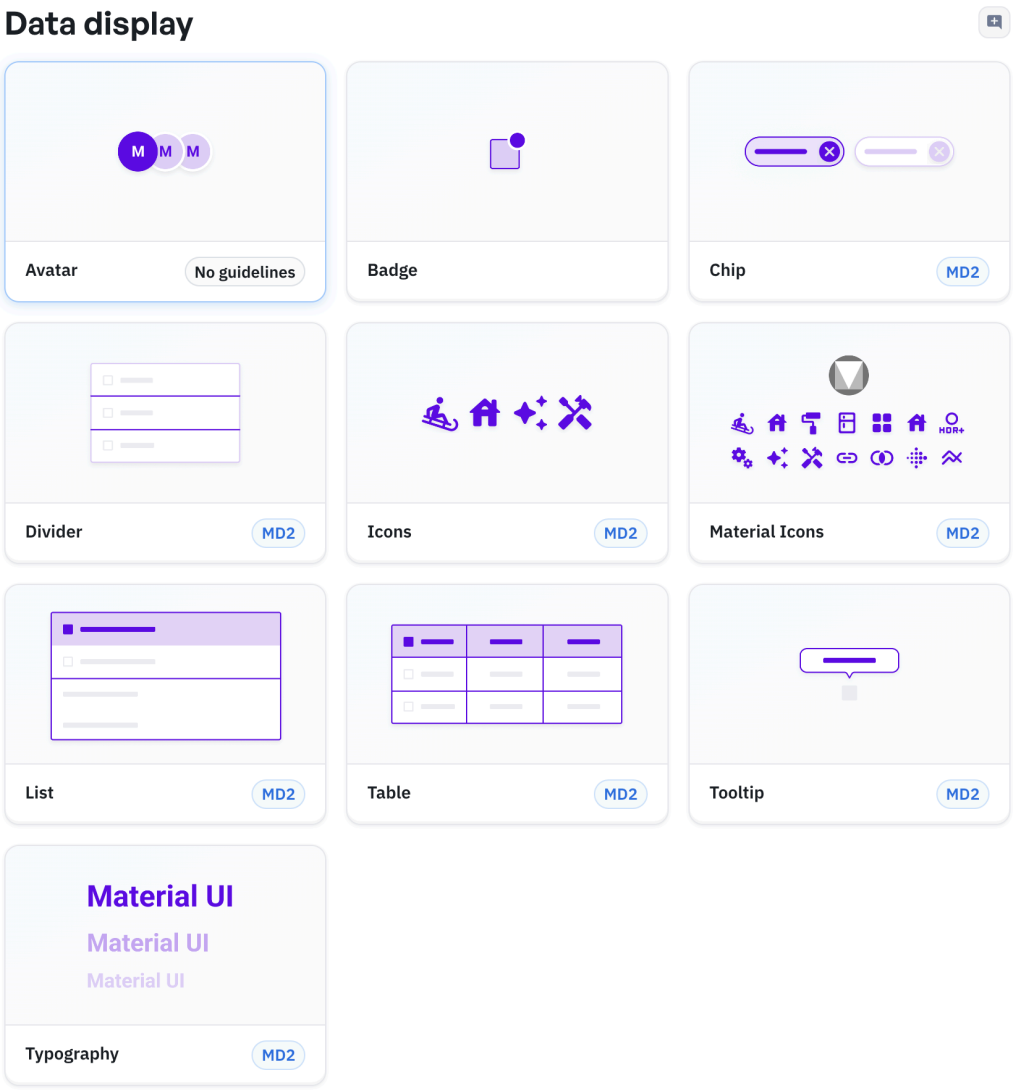


Fonte: Disponível em:
https://medium.com/loftbr/creating-a-design-system-with-monorepo-bc18e055fb3c

Implementando as diretrizes do Google Material Design, o Material-UI (MUI) é uma biblioteca de componentes ReactJS. O MUI, que oferece uma ampla gama de componentes prontos para uso que seguem os princípios de Material Design, permite que os desenvolvedores criem interfaces visualmente atraentes e consistentes de maneira rápida (VESSELOV; DAVIS, 2019).

A capacidade do MUI de fornecer componentes estilizados e prontos para uso, como botões, menus e diálogos, é uma de suas principais vantagens. Esses componentes são altamente customizáveis, permitindo ajustes para atender às necessidades específicas de projetos (Figura 3). A biblioteca suporta temas, permitindo personalização global de cores, tipografia e espaçamento, mantendo a identidade visual da marca (MUI, 2021).

Figura 3 — Componentes estilizados do MUI.



Fonte: (“MUI: The ReactJS component library you always wanted,” [s.d.])

Além disso, o MUI se integra facilmente com outras ferramentas e bibliotecas modernas de desenvolvimento web, como TypeScript, o que adiciona tipagem estática aos componentes, garante seu uso correto e facilita a manutenção do código (FAIN; MOISEEV, 2020). Em conclusão, o Material-UI, que proporciona consistência visual, eficiência no desenvolvimento e flexibilidade para personalização, é uma ferramenta poderosa para a criação de Design Systems. A experiência do usuário e do desenvolvedor é melhorada pela combinação de MUI com ReactJS e TypeScript, que permite o desenvolvimento de aplicações robustas e bem estruturadas (VESSELOV; DAVIS, 2019).

2.6 Vite

Vite é uma ferramenta de build inovadora, desenvolvida por Evan You, criador do Vue.js, que se destaca pela velocidade e facilidade de configuração no desenvolvimento web. Utilizando módulos ES nativos, Vite permite atualizações em tempo real e um recarregamento rápido de módulos durante o desenvolvimento, evitando a necessidade de empacotar o código, o que acelera significativamente o processo de desenvolvimento (“Vite”, [s.d.]).

A ferramenta também se integra perfeitamente com Rollup para otimizações de produção, assegurando que o código seja compactado e otimizado para ambientes de produção. Vite suporta uma ampla gama de frameworks e bibliotecas de JavaScript através de um sistema de plugins, permitindo uma fácil integração com várias tecnologias e adição de funcionalidades customizadas(“Vite”, [s.d.]).

Além disso, Vite melhora a experiência do desenvolvedor com funcionalidades como Hot Module Replacement (HMR), que permite que alterações no código sejam visualizadas instantaneamente sem perder o estado da aplicação, otimizando o desenvolvimento de interfaces de usuário (“Vite”, [s.d.]).

2.7 Linter

Linters são ferramentas essenciais para garantir a qualidade do código. Eles analisam o código-fonte em busca de erros, inconsistências estilísticas, problemas de segurança, e outras potenciais vulnerabilidades. Utilizados amplamente em ambientes de desenvolvimento, linters como o ESLint para JavaScript não apenas ajudam a manter o código limpo, consistente e seguro, mas também facilitam a manutenção a longo prazo. Essas ferramentas são cruciais para garantir que as práticas de codificação estejam alinhadas com os padrões estabelecidos, aumentando assim a confiabilidade e a segurança do software desenvolvido (SONARSOURCE, [s.d.]).

2.8 Emotion

Emotion é uma biblioteca CSS-in-JS amplamente utilizada para estilizar componentes em aplicações React. Ela permite a escrita de estilos dinâmicos usando JavaScript, oferecendo flexibilidade tanto na sintaxe tradicional CSS quanto em objetos JavaScript. A integração direta com temas facilita a criação de componentes consistentes e reutilizáveis, que podem adaptar-se dinamicamente às mudanças de tema.

Uma das principais vantagens do Emotion é seu suporte robusto para a renderização no lado do servidor (SSR), o que assegura que os estilos sejam aplicados corretamente durante a renderização inicial, melhorando o desempenho e a otimização para motores de busca. Além disso, Emotion oferece manipulação eficiente de propriedades, aninhamento de seletores e suporte a media queries, tornando-o ideal para criar layouts responsivos..

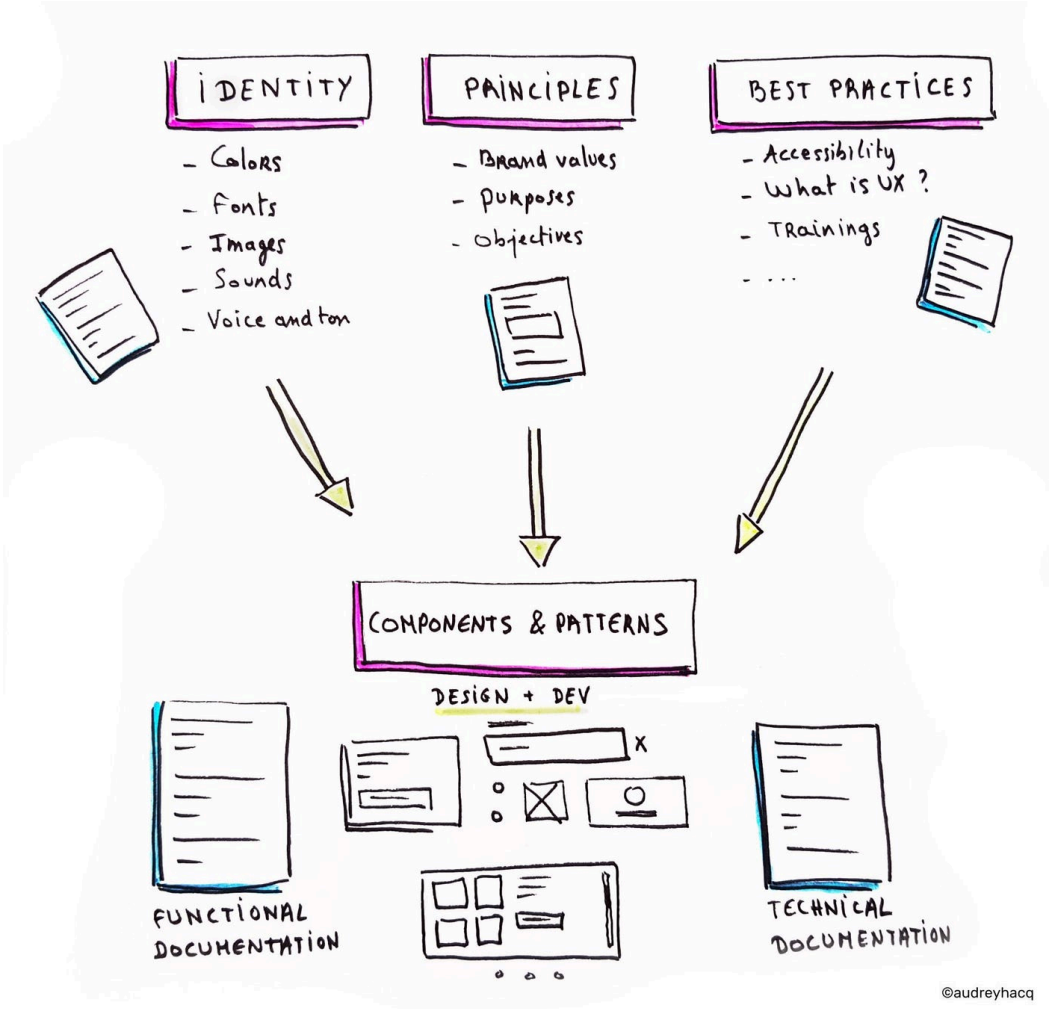
2.9 Figma

O Figma é uma ferramenta de design colaborativa baseada na nuvem, amplamente utilizada para a criação de interfaces de usuário, temas e componentes de design. Nesta pesquisa, o Figma desempenha um papel crucial ao permitir a criação e a prototipação de temas e componentes que serão implementados no Design System. Sua capacidade de facilitar a colaboração em tempo real entre designers e desenvolvedores é essencial para garantir a consistência e a eficiência no desenvolvimento de interfaces.

3 PROCESSO DE DESENVOLVIMENTO

Nesta seção, iremos detalhar as etapas necessárias para o estudo e implementação do *Design System* proposto. Cada subcategoria abordará um aspecto específico do desenvolvimento, desde a criação de componentes básicos até a integração final e testes de consistência, ilustradas na Figura 4.

Figura 4 — Processo de construção e desenvolvimento de um *Design System.*



Fonte:Disponível em:
https://ihfernando.com.br/blog/design-system-design-tokens-style-dictionary

3.1 Estrutura do Projeto

Para garantir a organização, escalabilidade e manutenção do código, a estrutura de arquivos de um projeto de Design System é fundamental. Iremos explicar como se organiza a estrutura de arquivos do projeto que foi elaborada com base em boas práticas de desenvolvimento de software e visa promover a reutilização e a manutenibilidade, como podemos observar na Figura 5. Inicialmente, o arquivo HTML base da aplicação está contido no diretório public/. Especificamente, o arquivo index.html é a porta de entrada para a aplicação, onde o React será montado (DERKS, 2022).

Além da organização física dos arquivos, o uso do Git no projeto é crucial para a gestão de versões do código-fonte. O Git permitirá que a equipe acompanhe todas as mudanças feitas nos arquivos, mantenha um registro histórico completo e gerencie diferentes

ramificações de desenvolvimento de forma eficaz. A utilização dessa ferramenta de controle de versão distribuída facilita a colaboração entre os desenvolvedores, permitindo que alterações sejam integradas e revisadas de maneira controlada e transparente. Assim, o Git não apenas suporta a organização técnica do projeto, mas também é essencial para a escalabilidade e manutenção do sistema de design ao longo do tempo, garantindo que todos os componentes e implementações estejam devidamente sincronizados e documentados.

O diretório *src/*, para o qual seguimos, é o principal do código-fonte e está organizado de maneira modular. Este diretório inclui subdiretórios como *assets/*, *components/*, *themes/* e *utils/*. A manutenção de arquivos estáticos, como imagens e ícones, no diretório *assets/* auxilia na organização e facilita o acesso, conforme evidenciado por Fain (2020).

Dentro do diretório *components/*, cada componente possui seu próprio subdiretório, contendo a implementação (.tsx), a estilização (.styles.ts) e os testes (.test.tsx). Essa separação não só facilita a manutenção, mas também a compreensão do código, conforme observado por Fain (2020) e Flanagan (2020). Exemplos de componentes fundamentais do *Design System* incluem *Button/*, *Input/*, que seguem a filosofia de componentização do React, promovendo a reutilização e a modularidade (DERKS, 2022).

Contendo definições de temas globais e estilos, o diretório *themes/* permite a personalização e a consistência visual em toda a aplicação. Vesselov e Davis (2019) destacam a importância deste diretório, que inclui arquivos como defaultTheme.ts, que define o tema padrão. Enquanto isso, o diretório *providers/* se encarrega dos Providers utilizados na aplicação onde cada provider ficará alocado em sua respectiva pasta. Os providers são um padrão utilizado para passar dados e funcionalidades através da árvore de componentes sem precisar usar props explicitamente em cada nível. Este padrão é implementado principalmente por meio do contexto de ReactJS, que permite que os dados sejam disponibilizados para todos os componentes, independentemente de sua posição na árvore de componentes., em nosso caso, temos o ThemeProvider que possibilita encapsular todos os componentes com os nossos temas e definições específicas.

Adicionalmente, o diretório *utils/* é utilizado para funções utilitárias e helpers que são reutilizáveis em diferentes partes do projeto (FAIN, 2020). Arquivos principais, como App.tsx e index.tsx, são usados como o componente raiz da aplicação e o ponto de entrada do React, respectivamente. O arquivo vite-env.d.ts contém declarações de tipos específicas do Vite, essencial para o desenvolvimento abordando os tipos.

Além dos diretórios de código-fonte, há uma série de arquivos de configuração essenciais para o projeto. O .gitignore define arquivos e diretórios que devem ser ignorados

pelo Git, ajudando a manter o repositório limpo e organizado. O arquivo package.json contém as dependências do projeto e scripts de build. O tsconfig.json inclui as configurações do TypeScript, garantindo a tipagem estática e melhorando a robustez do código (MICROSOFT CORPORATION, [s.d.]). O vite.config.ts define as configurações do Vite, e o README.md fornece a documentação do projeto, essencial para informações e instruções sobre o projeto (KHOLMATOVA, 2017).

Figura 5 — Representação da organização das pastas do projeto.

Fonte: Elaborada pelo Autor

Esta estrutura modular facilita escalabilidade, manutenção e colaboração entre desenvolvedores, seguindo boas práticas de organização de projetos, essenciais para garantir eficiência e produtividade no desenvolvimento de software (DERKS, 2022; FAIN, 2020).

3.2 Tecnologias

Em um contexto de Single Page Application (SPA), cada uma das tecnoloiga mencionadas na seção 2 (Referencial Teórico) deste trabalho, desempenha um papel específico no desenvolvimento e design desta aplicação.

- Typescript: Utilizado para fornecer a tipagem estática para nosso projeto, garantindo mais segurança e confiabilidade durante o desenvolvimento.
- ReactJS: Biblioteca principal utilizada para construir a interface dos usuários em uma estrutura baseada em Componentes.
- Vite: Uma ferramenta de build utilizada para garantir a inicialização do projeto bem como fornecer funcionalidades adicionais como carregamento rápido das modificações;
- Linter: Analisa o código para encontrar problemas, inconsistências de estilo ou potenciais bugs.
- Emotion: Nos permite a utilização do Javascript para a construção dos nossos estilos personalizados de CSS.
- Figma: Utilizado como base de documentação durante a elaboração e planejamento do Design System, bem como referência visual durante a implementação.

3.3 Configuração do Ambiente

Alguns passos essenciais devem ser seguidos para configurar o ambiente de desenvolvimento. Primeiramente, é necessário garantir que o Node.js[6] e o npm (Node Package Manager) estejam instalados na máquina do desenvolvedor. Essas ferramentas são fundamentais para a execução de JavaScript no servidor e para a instalação e gestão de dependências do projeto, respectivamente.

Após a instalação do Node.js e do npm, o Vite será utilizado para inicializar um novo projeto. Pelo terminal o processo de criação de um novo projeto pelo Vite guia o usuário pela configuração inicial, incluindo a escolha do framework e do template de TypeScript. Durante este processo, o Vite configura automaticamente as dependências necessárias e prepara o ambiente para o desenvolvimento (GONZALEZ et al., 2022).

[6] Disponível em: https://nodejs.org/

É essencial instalar todas as dependências do projeto após a sua criação. Este passo garante que todas as bibliotecas e ferramentas necessárias estejam disponíveis para uso, evitando problemas durante o desenvolvimento. A instalação das dependências é realizada utilizando o comando npm install, que permite instalar e configurar todos os pacotes listados no arquivo package.json (GONZALEZ et al., 2022).

O próximo passo, após a instalação das dependências, é configurar alguns arquivos essenciais para o funcionamento do projeto. Estes arquivos incluem configurações específicas do TypeScript, como o tsconfig.json, e do Vite, que garantem que o ambiente esteja pronto para o desenvolvimento. Configurar corretamente esses arquivos é fundamental para aproveitar ao máximo os benefícios dessas tecnologias, como a tipagem estática do TypeScript e a rapidez do Vite.

Por fim, inicia-se o servidor de desenvolvimento e verifica se o projeto está funcionando corretamente. O servidor de desenvolvimento permite que o desenvolvedor visualize as mudanças em tempo real no navegador, facilitando o processo de desenvolvimento iterativo.

3.2 Desenvolvimento dos Componentes

Nesta seção, detalharemos o processo de desenvolvimento dos componentes do *Design System*. Para esta implementação, foram selecionados dez componentes fundamentais do MUI que são frequentemente utilizados em aplicações modernas. A seleção desses componentes visa garantir uma ampla cobertura das necessidades de interface, promovendo consistência e eficiência no desenvolvimento.

Os componentes selecionados pertencem a diferentes grupos de elementos, sendo eles:

- Entradas: Button, Select, Radio Group
- Exibição de dados: List, Chip
- *Feedback*: Alert, Snackbar
- Surface: App Bar
- Navegação: Bottom Navigation, Link, Menu, Pagination

Além disso alguns outros componentes padrão do MUI serão utilizados sem modificação, como este *Design System* é feito utilizando como base o MUI, além dos componentes e temas customizados implementados neste trabalho, ainda temos disponíveis as

funcionalidades básicas do Material-UI para servirem de suporte, como *Stack*[7], *Grid*[8], *Container*[9], entre outros.

3.2.1 Definição dos Temas

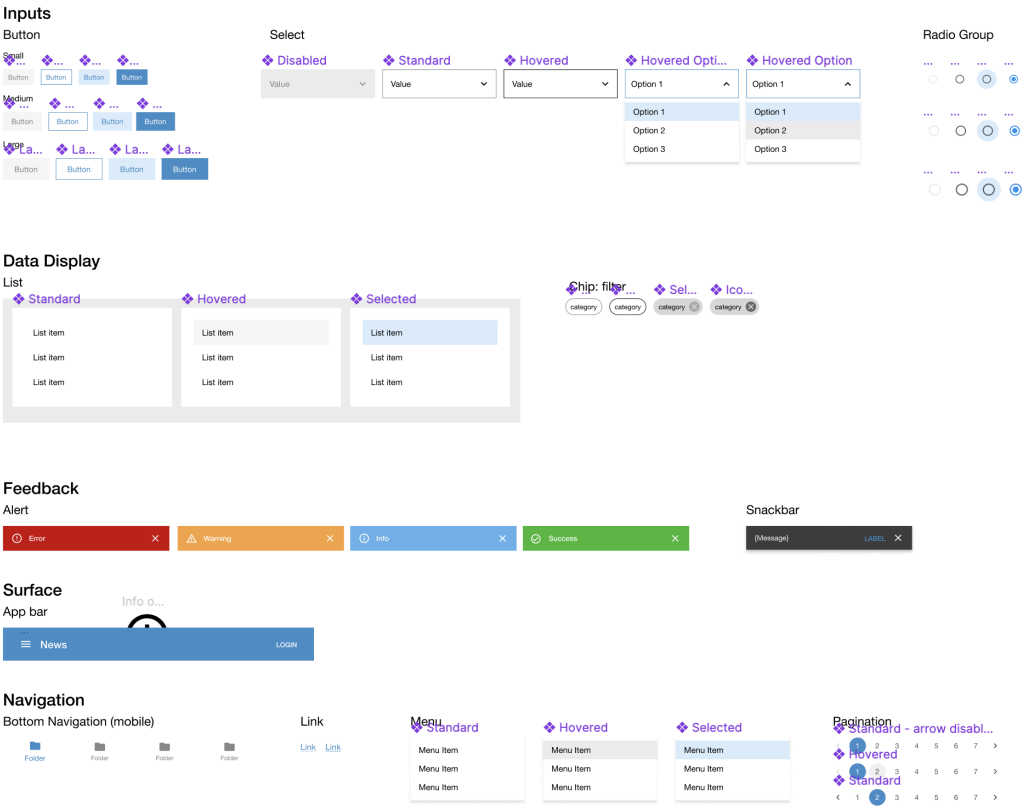
A definição do tema é uma etapa crucial na criação de um *Design System*, pois assegura a consistência visual e funcionalidade das interfaces de usuário. Serão especificadas paletas de cores, tipografia e outros elementos estilísticos essenciais para garantir a consistência visual em toda a aplicação. A escolha de temas coesos e bem definidos é crucial para a criação de uma experiência de usuário harmoniosa, a Figura 6 demonstra a estrutura de componentes final aplicada aos nossos componentes. Transferindo para o figma as informações definidas para o tema em termos de padrão de cores, tipografia e componentização, foi criada uma galeria de componentes para ser utilizada.

[7] Stack é um componente que facilita o layout de seus filhos em uma única direção, vertical ou horizontal, com espaçamento e alinhamento flexíveis.

[8] Grid é um sistema de layout bidimensional que permite a criação de layouts complexos e responsivos através de um sistema de colunas e linhas, similar ao Grid do CSS

[9] Container é um componente que centraliza o conteúdo horizontalmente dentro de sua área máxima permitida, proporcionando um alinhamento e preenchimento consistentes.

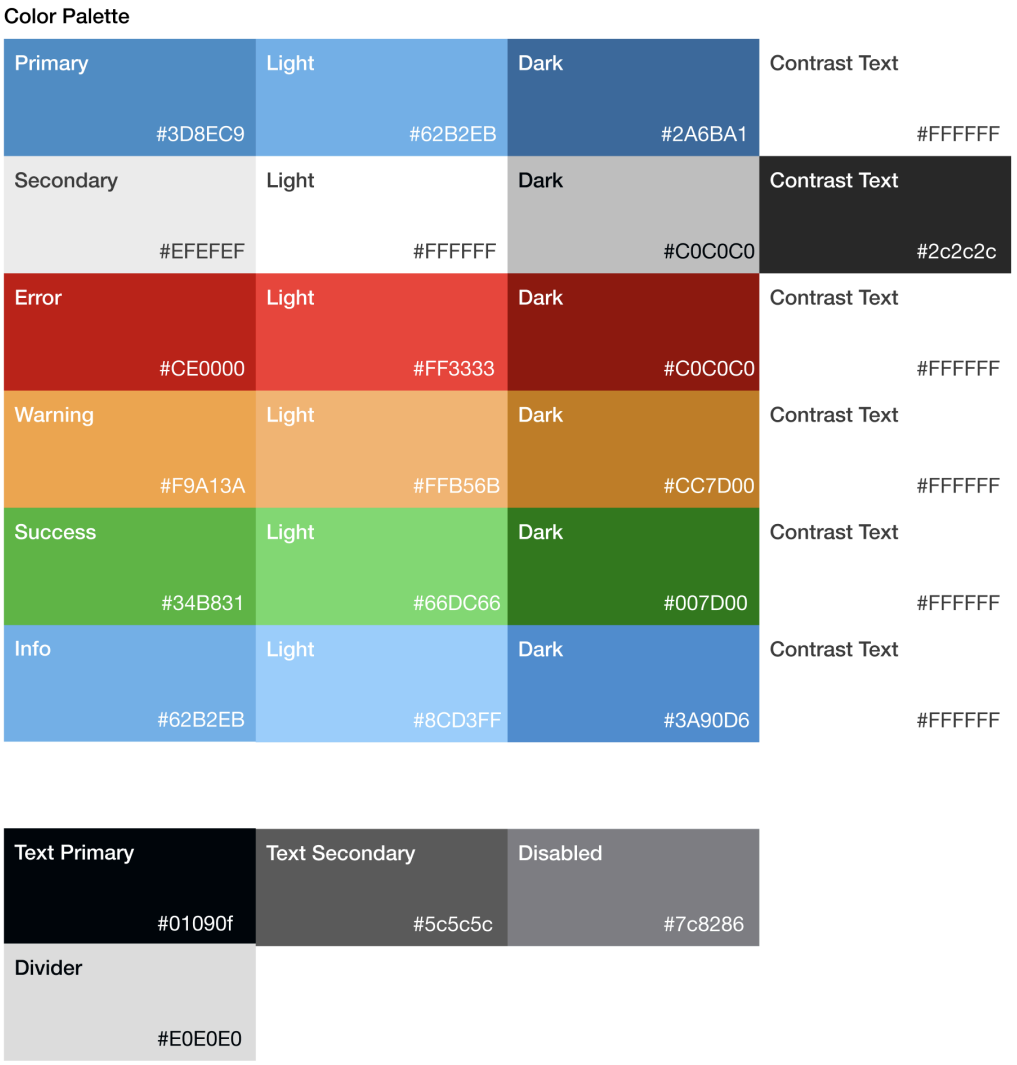
Figura 6 — Visão do tema para os componentes do *Design System*



Fonte: Elaborada pelo Autor.

As cores selecionadas para este *Design System* foram escolhidas para proporcionar uma experiência visual harmoniosa e acessível. A paleta de cores inclui tons de azul para os estados padrão e selecionado dos botões e elementos de navegação, a Figura 7 descreve as cores e variações utilizadas no contexto deste projeto. Cores neutras como cinza e branco são utilizadas para estados desabilitados e de fundo. Para feedback, cores distintas são usadas para alertas (vermelho para erro, laranja para aviso, azul para informação e verde para sucesso), garantindo que os usuários possam identificar rapidamente o tipo de mensagem exibida. A utilização estratégica das cores visa não apenas a estética, mas também a funcionalidade, promovendo a usabilidade e a acessibilidade.

Figura 7 — Padrão de cores do tema



Fonte: Elaborada pelo Autor.

A tipografia selecionada para o *Design System* é fundamental para a legibilidade e a consistência visual, como representado na Figura 8. Optou-se por uma fonte sans-serif moderna e limpa, que proporciona clareza em diferentes tamanhos e resoluções de tela. Exemplos incluem fontes com maior peso e tamanho para cabeçalhos, fontes leves e regulares para texto de corpo, e fontes de tamanho intermediário para botões e labels. A escolha da tipografia segue as recomendações de design de interfaces contendo elementos como:

- H1 a H6: Usados para títulos e subtítulos, do mais importante (h1) ao menos importante (h6). Cada um tem um tamanho de fonte e peso decrescentes, ajudando a criar uma distinção clara na hierarquia da informação..

- Subtitle1 e Subtitle2: Usados para subtítulos que são menos proeminentes que os títulos principais. subtitle1 é geralmente maior e mais enfático, enquanto subtitle2 é mais sutil.
- Body1 e Bodyd2: Usados para o texto principal dentro de um componente ou página. body1 é frequentemente utilizado para conteúdo mais longo, enquanto body2 pode ser usado para conteúdo complementar ou informações adicionais em tamanhos um pouco menores.
- Caption: Usado para texto descritivo pequeno, como legendas sob imagens ou textos explicativos em pequenos componentes. É útil para adicionar contexto sem chamar muita atenção.
- Button: Especificamente estilizado para ser usado em botões. Geralmente é em caixa alta, com espaçamento entre letras ajustado para aumentar a legibilidade em botões pequenos.
- Overline: Usado para texto acima de cabeçalhos ou seções, muitas vezes em caixa alta, para indicar uma categoria ou uma etiqueta.

Figura 8 — Tipografia do tema.

Typography

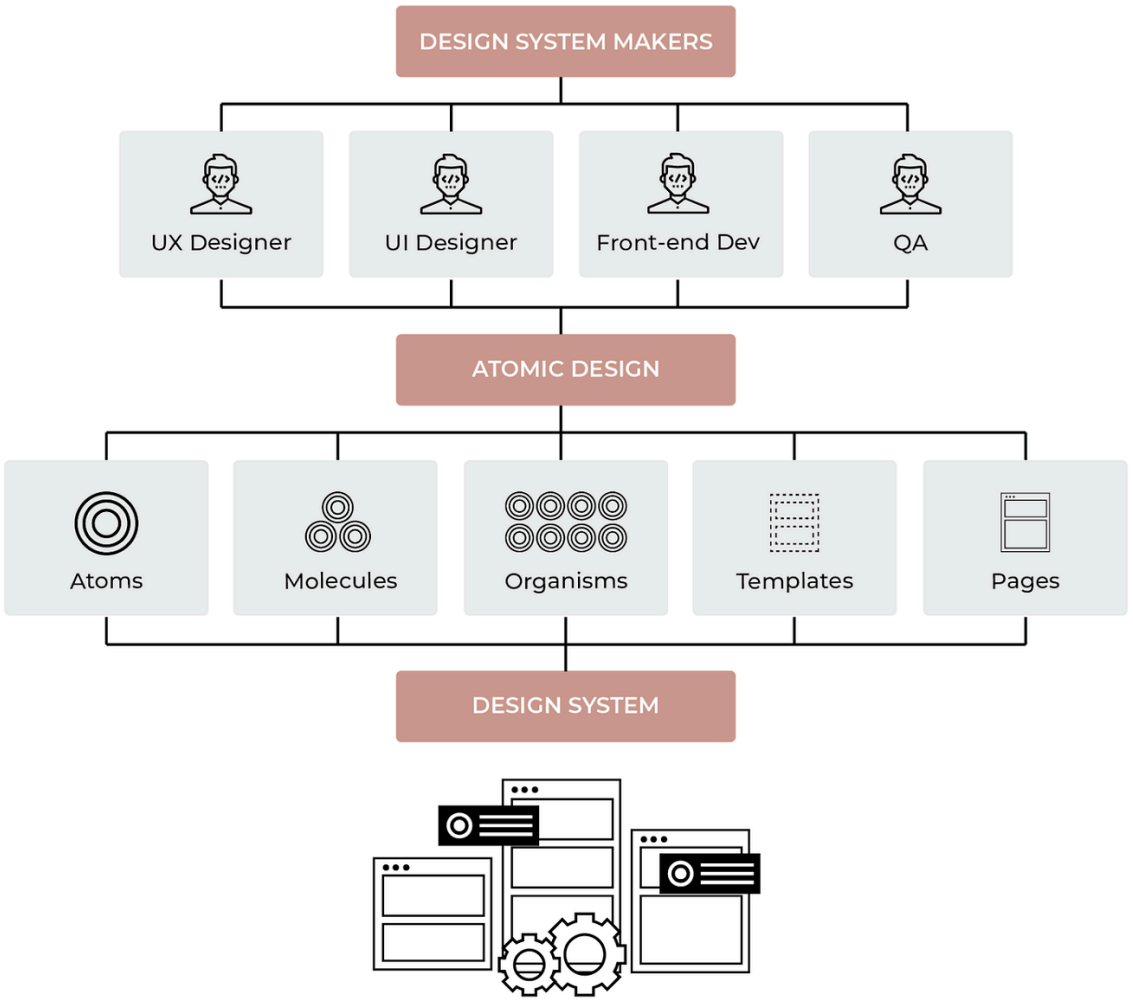
Scale Category	Typeface	Weight	Size	Case	Letter spacing
H1	Helvetica Neue	Light	96	Sentence	-1.5
H2	Roboto	Light	60	Sentence	-0.5
H3	Roboto	Regular	48	Sentence	0
H4	Roboto	Regular	34	Sentence	0.25
H5	Roboto	Regular	24	Sentence	0
H6	Roboto	Medium	20	Sentence	0.15
Subtitle 1	Roboto	Regular	16	Sentence	0.15
Subtitle 2	Roboto	Medium	14	Sentence	0.1
Body 1	Roboto	Regular	16	Sentence	0.5
Body 2	Roboto	Regular	14	Sentence	0.25
BUTTON	Roboto	Medium	14	All caps	1.25
Caption	Roboto	Regular	12	Sentence	0.4
OVERLINE	Roboto	Regular	10	All caps	1.5

Fonte: Elaborada pelo Autor.

3.2.2 Documentação dos Componentes

No desenvolvimento de software, a documentação é fundamental, pois oferece clareza e facilita a manutenção e integração de novos membros na equipe. Documentar componentes assegura que desenvolvedores, designers e outros stakeholders, como os designers, desenvolvedores e QAs, como vemos na Figura 9, compreendam o funcionamento e a aplicação de cada parte do sistema, promovendo uma comunicação eficiente e melhor colaboração entre as equipes (KRYSIK, 2023; “The Crucial Role of Documentation in Software Development,” [s.d.]).

Figura 9 — Atores e processos de um *Design System*.



Fonte: Disponível em:
https://medium.com/ipnet-growth-partner/design-system-o-que-e-438773dd811

O Storybook será utilizado para documentar os componentes deste projeto. O Storybook é uma ferramenta reconhecida que permite visualizar e testar componentes de UI de forma isolada, sendo particularmente eficaz para *Design Systems*. Ele possibilita a criação de "stories", representações dos diferentes estados dos componentes, que servem tanto como documentação interativa quanto como exemplos práticos de uso (AKHTAR, 2023).

Facilitando a documentação automatizada, que inclui tabelas interativas de propriedades, exemplos de uso e diretrizes visuais, o uso do Storybook é essencial para a correta utilização e consistência dos componentes em toda a aplicação (“The Crucial Role of Documentation in Software Development,” [s.d.]). Empresas renomadas, como Shopify[10] e IBM[11], utilizam o Storybook para documentar seus *Design Systems*, evidenciando sua eficácia e flexibilidade (KRYSIK, 2023).

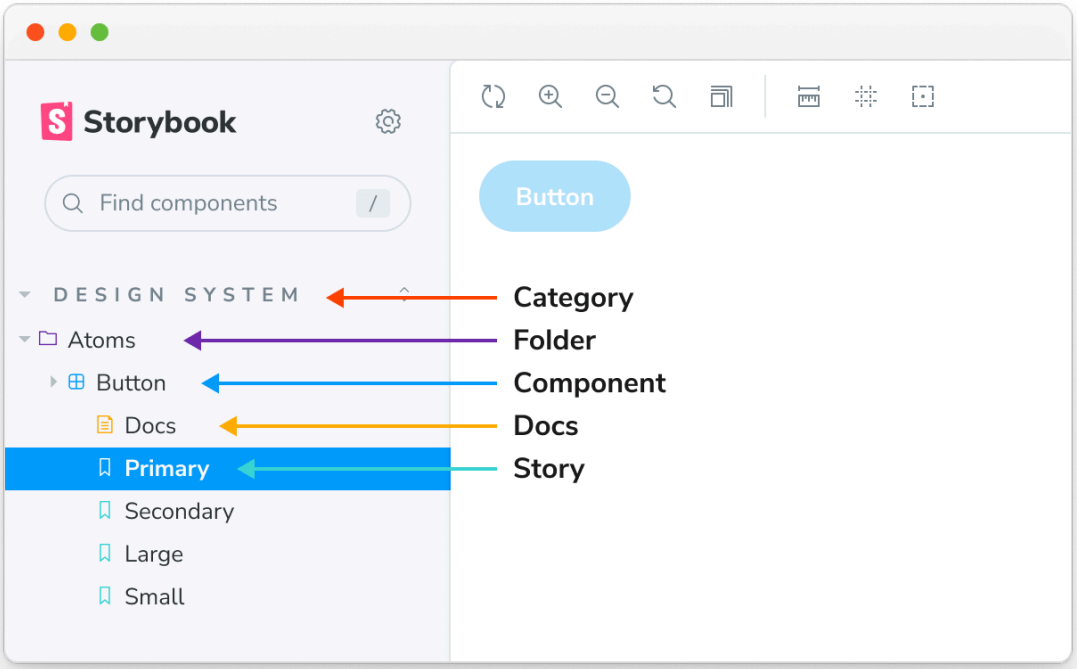
Portanto, a adoção do Storybook neste projeto assegurará que todos os stakeholders compreendam e utilizem os componentes de maneira eficaz, promovendo um desenvolvimento mais ágil e colaborativo.

A Figura 10 ilustra a estrutura hierárquica utilizada no Storybook, permitindo a organização do conteúdo em diversas categorias. Dentro desta estrutura, a categoria "Design System" é seguida por diretórios que representam diferentes níveis de componentização, como "Atoms", que abriga a documentação de componentes atômicos. O arquivo "Docs" dentro de cada pasta fornece uma documentação abrangente do componente, incluindo visualizações, descrições textuais e variações possíveis. Paralelamente, os "Stories" detalham cada variação específica do componente. No exemplo mencionado do botão, identificam-se quatro variações distintas, cada uma com seu próprio "Story". Um documento "Docs" mais genérico engloba e contextualiza todas essas variações, facilitando o entendimento e a aplicabilidade dos componentes dentro do sistema.

Figura 10 — Estrutura e Hierarquia do Storybook.

[10] Shopify é uma plataforma de comércio eletrônico que permite a criação e gestão de lojas virtuais.

[11] IBM (International Business Machines Corporation) é uma empresa multinacional de tecnologia e consultoria, fundada em 1911.



Fonte: Disponível em:
https://storybook.js.org/docs/writing-stories/naming-components-and-hierarchy

3.2.3 Criação de Componentes Básicos

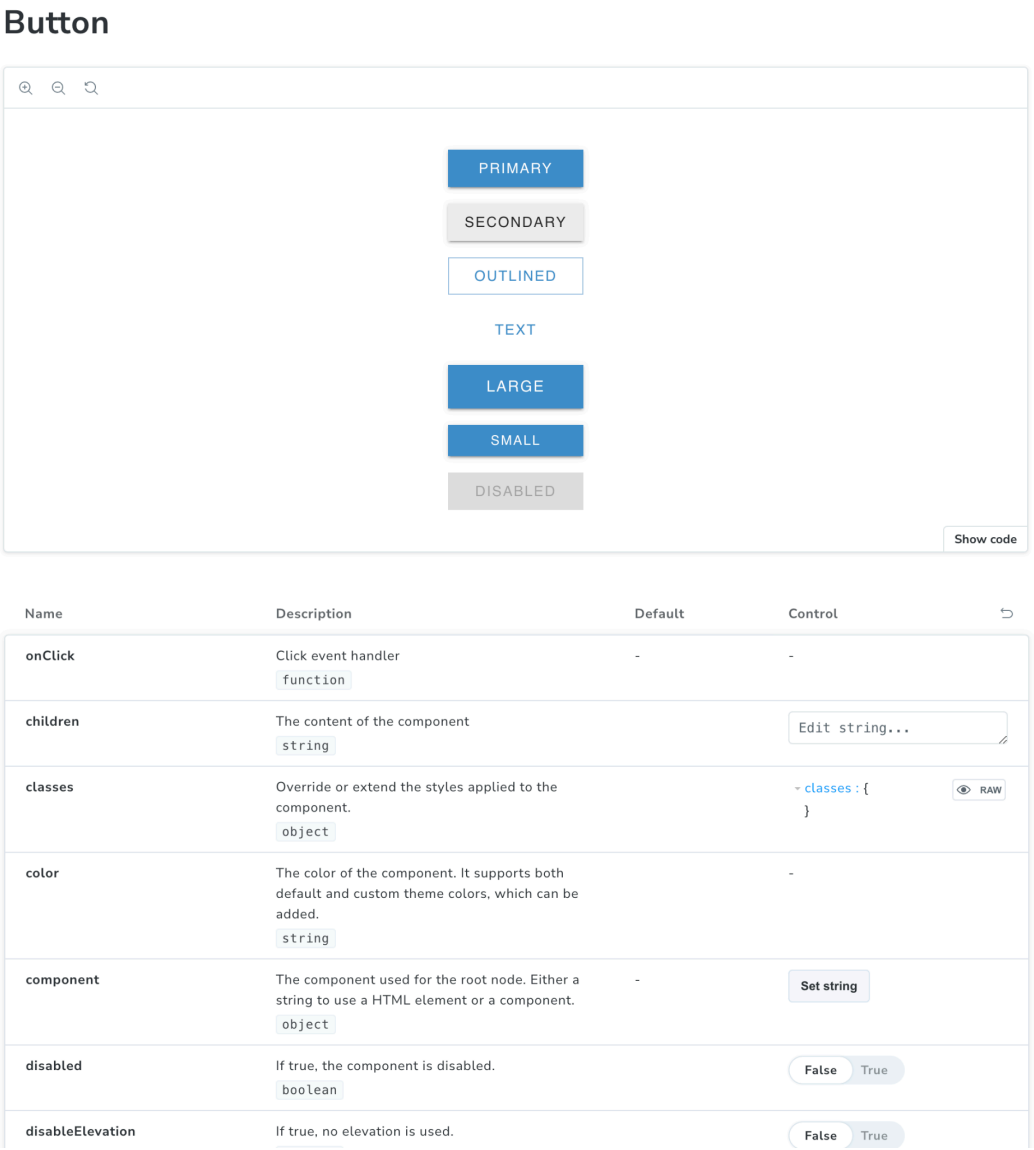
Para a implementação dos componentes, o primeiro passo foi criar um tema customizado do Material-UI (MUI) para um tema padrão, denominado *defaultTheme*. Este tema incluiu a tipografia e a paleta de cores especificadas anteriormente, proporcionando uma consistência visual em toda a aplicação.

A definição do tema envolveu a configuração das cores primárias, secundárias, de erro, de aviso, de sucesso e de informação, bem como suas variantes claras e escuras como visto na seção de Definição dos Temas deste trabalho. Adicionalmente, ajustamos a tipografia para assegurar legibilidade e coerência visual em diferentes tamanhos e resoluções de tela. Este tema customizado serve como base para todos os componentes, garantindo uma aparência uniforme e profissional.

Com o tema devidamente configurado, avançamos para a implementação dos componentes. Cada componente foi documentado no Storybook com seus diferentes estados e

variações, facilitando o desenvolvimento iterativo e a colaboração entre equipes. A Figura 11 ilustra como ficou definido nosso componente *Button* com suas variações.

Figura 11 — Documentação para o componente *Button*.



Fonte: Elaborada pelo Autor.

A documentação dos componentes no Storybook incluiu detalhes abrangentes sobre propriedades, parâmetros, valores padrão e exemplos de uso. Por exemplo, o componente de botão foi documentado com várias stories demonstrando suas variações, como diferentes tamanhos (pequeno, médio, grande), estados (padrão, desabilitado, selecionado) e tipos (contido, de texto, de contorno). Esta abordagem não só proporciona uma documentação clara

e acessível, mas também permite que os desenvolvedores visualizem e interajam com os componentes em tempo real, ajustando propriedades e observando os efeitos imediatos.

A implementação cuidadosa e a documentação detalhada dos componentes garantem que o *Design System* seja robusto, escalável e fácil de manter. A criação de um tema customizado do MUI, aliado à documentação no Storybook, estabelece uma base sólida para o desenvolvimento de interfaces de usuário consistentes, acessíveis e visualmente atraentes.

3.3 Integração em uma SPA

A implementação do Design System em um projeto SPA utilizando ReactJS será abordada nesta seção. Serão analisados aspectos relevantes da integração e validados em comparação com o desenvolvimento tradicional. Esta análise permitirá avaliar a eficácia e as vantagens do *Design System* em termos de consistência, reutilização de componentes e manutenção do código, em um ambiente de desenvolvimento real.

Inicialmente, um sistema de controle de versões, como o Git[12], foi utilizado para desenvolver e versionar o Design System. Este processo permitiu manter um histórico detalhado de todas as modificações e evoluções do *Design System*, facilitando o gerenciamento e a colaboração entre os membros da equipe. Após a finalização da fase de desenvolvimento, o *Design System* foi empacotado, criando um package que poderia ser facilmente distribuído e integrado a outros projetos.

O processo de criação do package envolveu a configuração de um manifesto de package, geralmente um arquivo package.json, onde foram especificadas as dependências, scripts e outras configurações necessárias para o funcionamento do *Design System*. Este arquivo serviu como ponto central de controle, garantindo que todas as especificações e requisitos fossem claramente definidos.

Um repositório Git foi utilizado para disponibilizar o package. No projeto SPA, a integração do *Design System* foi realizada adicionando o repositório Git como uma dependência no arquivo de manifesto do projeto. Isso possibilitou que todas as funcionalidades e estilos definidos no *Design System* fossem incorporados na SPA de forma direta e modular.

[12] Git é um sistema de controle de versão distribuído usado para rastrear mudanças no código-fonte durante o desenvolvimento de software.

Uma vez importado, o *Design System* foi utilizado nos componentes da SPA. A aplicação do tema customizado e a utilização dos componentes predefinidos garantiram uma aparência uniforme e uma experiência de usuário consistente. A integração permitiu ainda que quaisquer atualizações ou melhorias no *Design System* fossem refletidas automaticamente na SPA, promovendo uma manutenção mais simples e eficiente.

Em contrapartida, a ausência de um *Design System* pode levar a uma fragmentação substancial no processo de desenvolvimento. Projetos sem um sistema de design unificado tendem a apresentar uma variedade de estilos e componentes redundantes, o que resulta em uma perda significativa de tempo e recursos, à medida que equipes diferentes desenvolvem soluções similares de forma isolada. Além disso, a falta de padrões coesos pode causar inconsistências visuais entre produtos, prejudicando a identidade e a coesão da marca.

A eficiência operacional promovida pelo uso de um *Design System* via Git não apenas acelera o processo de desenvolvimento, mas também facilita a manutenção e a escalabilidade dos sistemas. Esta abordagem reduz o retrabalho e melhora a manutenção do código ao longo do tempo. Em comparação, a ausência de um sistema de design estruturado exige mais esforços de manutenção e aumenta a probabilidade de erros, o que pode resultar em custos mais elevados e prolongar o ciclo de desenvolvimento.

Portanto, a adoção de um *Design System* integrado via Git é uma prática recomendada para organizações que visam otimizar seus processos de desenvolvimento e garantir uma consistência visual robusta em seus produtos, não sendo tão relevante no caso do desenvolvimento de aplicações individuais com temas e componentes únicos. A padronização de componentes e estilos não só beneficia a experiência do usuário, mas também contribui para uma gestão mais eficaz e eficiente dos recursos de desenvolvimento.

4 ANÁLISE DOS RESULTADOS

4.1 Análise de Consistência Visual

Para garantir que todos os componentes e estilos sejam aplicados uniformemente, a implementação de testes de consistência visual envolve várias etapas fundamentais. Inicialmente, configura-se o ambiente de teste instalando ferramentas apropriadas como o Storybook para documentar e visualizar componentes, e bibliotecas de teste visual como Chromatic[13] ou Percy[14], que capturam e comparam capturas de tela dos componentes.

Após a configuração do Storybook, capturas de tela de referência para cada componente e suas variações são geradas, criando um baseline visual. Em seguida, o Design System é importado no projeto SPA como um package, utilizando um repositório Git. Essa importação permite que todos os componentes e estilos do Design System sejam aplicados no contexto da SPA, garantindo uma implementação modular e eficiente.

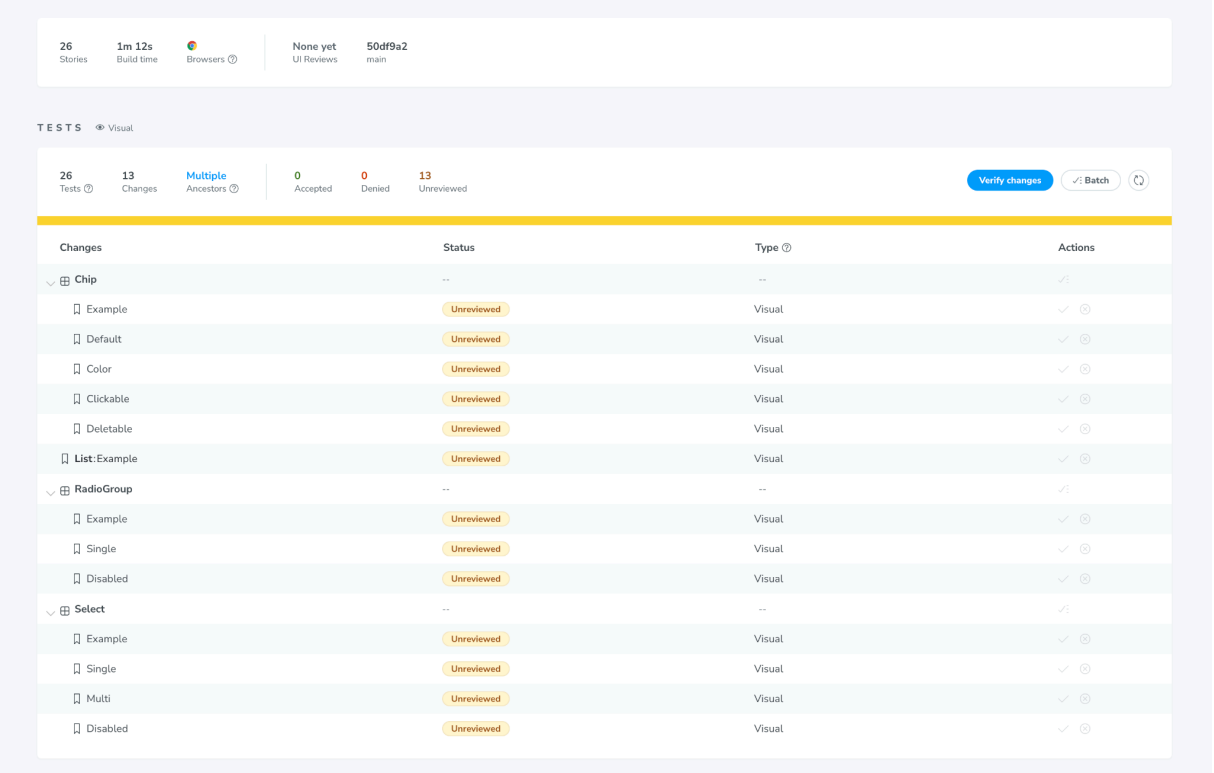
Após a devida configuração do Design System, o Storybook será configurado novamente, mas dessa vez no SPA. Precisamos novamente criar as mesas *stories* criadas no diretório do DS. Dessa forma teremos como usar um storybook como referência para podermos analisar o outro.

Durante a fase de execução dos testes, a ferramenta de teste visual captura novas imagens dos componentes no ambiente da SPA e as compara com as capturas de tela de referência. Qualquer discrepância visual é destacada, permitindo a identificação de inconsistências. Os resultados dos testes são então analisados para determinar se as discrepâncias são intencionais ou indicam erros de implementação.

Figura 12 — Relatório visual gerado pelo Chromatic.

[13] Chromatic é uma ferramenta de testes visuais para Storybook que captura e compara capturas de tela de componentes, detectando alterações visuais ao nível de pixel e facilitando a revisão de mudanças diretamente no painel do Storybook.

[14] Percy é uma ferramenta de testes visuais que também captura e compara capturas de tela de componentes, destacando diferenças visuais e integrando-se com diversas plataformas e sistemas de CI/CD, facilitando a detecção de regressões visuais em mudanças de código.



Fonte: Elaborada pelo Autor

Como constatado na Figura 12, onde temos o resultado gerado pelo Chromatic validando o nosso SPA em relação ao referencial do nosso Design System. onde diversas divergências foram identificadas entre as capturas de tela atuais dos componentes e as referências previamente estabelecidas durante a análise de consistência visual, gerando uma lista de componentes com pequenas diferenças visuais em alguns textos ou bordas. No entanto, é importante salientar que muitas dessas diferenças são ruídos visuais que não afetam a usabilidade ou o visual definido do componente em comparação com a referência. Tais ruídos podem ser o resultado de pequenas variações no ambiente de teste, tais como diferentes resoluções de tela, variações de renderização entre navegadores, ou até mesmo pequenas alterações no *subpixel anti-aliasing*[15]. Por exemplo, as variações detectadas no componente Select, como demonstrado na Figura 13, que incluem diferenças mínimas na renderização de bordas e sombras, são imperceptíveis ao usuário final e não comprometem a funcionalidade ou a estética pretendida dos componentes. Essas pequenas discrepâncias são comuns em testes visuais automatizados e devem ser cuidadosamente revisadas pela equipe de design, onde são avaliados os aspectos relevantes de cada componente e como eles divergem.

[15] Subpixel anti-aliasing é uma técnica de renderização de fontes e gráficos que melhora a nitidez e a suavidade das bordas em displays digitais.

para garantir que não representem problemas reais.

Figura 13 — Detalhamento de alterações visuais pelo Chromatic.

Fonte: Elaborada pelo Autor

Portanto, embora a análise inicial possa identificar várias divergências, a revisão cuidadosa permite distinguir entre ruídos visuais e mudanças significativas, garantindo assim que o *Design System* mantenha sua integridade visual e funcional ao longo do desenvolvimento.

4.2 Análise de Desenvolvimento e Manutenibilidade

Para avaliar o desenvolvimento e a manutenibilidade de um DS em comparação com um projeto tradicional em Single Page Application (SPA), utilizamos o Rollup Plugin Visualizer[16] para medir métricas de tamanho de bundle e tempo de execução das tarefas. Foram criadas duas páginas distintas: uma utilizando várias variações dos componentes criados no *Design System*, e outra utilizando os mesmos componentes, porém criados diretamente no projeto ao invés de importados como um pacote.

O uso do Rollup Plugin Visualizer permitiu obter dados detalhados sobre o tamanho dos *bundles*[17] e o tempo de execução das tarefas. Com a configuração adequada, foi possível medir a eficiência de cada abordagem.

Figura 14 — Métricas de *bundle* com desenvolvimento tradicional.

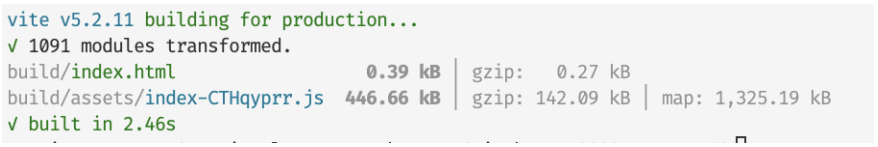
```
vite v5.2.11 building for production...
✓ 1114 modules transformed.
build/index.html                  0.39 kB │ gzip:   0.27 kB
build/assets/index-dM7qJRC5.js  372.43 kB │ gzip: 116.21 kB │ map: 1,431.97 kB
✓ built in 2.38s
```

Fonte: Elaborada pelo Autor

[16] Disponível em: https://www.npmjs.com/package/webpack-bundle-analyzer

[17] Um bundle é um arquivo que agrupa e compacta múltiplos arquivos JavaScript (e possivelmente outros tipos de arquivos) em um único arquivo

Figura 15 — Métricas de *bundle* utilizando o *Design System.*



Fonte: Elaborada pelo Autor

No desenvolvimento tradicional, o bundle teve um tamanho de 372.43 kB (gzip: 116.21 kB) e um tempo de execução de 2.38 segundos. O tamanho do source map foi de 1,431.97 kB. Em comparação, utilizando o *Design System*, o tamanho do *bundle* foi maior, alcançando 446.66 kB (gzip: 142.09 kB), e o tempo de execução foi de 2.46 segundos. No entanto, o tamanho do source map foi menor, com 1,325.19 kB, ilustrados nas Figuras 14 e 15. Essas diferenças indicam que o uso de um *Design System*, apesar de aumentar inicialmente o tamanho do bundle devido ao overhead introduzido pelo pacote do *Design System*, resulta em uma menor complexidade e quantidade de código customizado.

Existem diversas razões pelas quais a manutenibilidade de um Design System se destaca. Isso é particularmente útil para projetos múltiplos que compartilham o mesmo tema ou conjunto de componentes. Além disso, o *Design System* geralmente inclui ferramentas próprias de testes e validações, como testes unitários, visuais e de acessibilidade, que são executados separadamente do projeto principal. Isso assegura uma maior robustez e segurança nas alterações.

Outro aspecto crucial é a compatibilidade retroativa. Componentes podem ser marcados como obsoletos (*deprecated*) e substituídos por novas implementações, mantendo a retrocompatibilidade com versões anteriores até que os projetos possam ser atualizados. Em contraste com isso, em projetos desenvolvidos de forma tradicional, cada mudança visual ou funcional precisa ser replicada em múltiplos locais, o que aumenta o risco de inconsistências e erros. A falta de um ponto central de verdade leva a um maior esforço de manutenção.

Além disso, a ausência de um sistema centralizado para testes e validações em projetos tradicionais pode resultar em uma menor cobertura de testes e maior propensão a bugs não detectados até a fase de produção. Atualizações e melhorias precisam ser aplicadas manualmente em cada projeto individualmente, o que pode ser trabalhoso e suscetível a falhas.

Em conclusão, apesar do maior tamanho inicial do bundle, nota-se que o Design System proporciona uma abordagem mais sustentável e eficiente para projetos de larga escala. A centralização das alterações, o suporte a testes automatizados e a capacidade de manter a

compatibilidade retroativa são aspectos que contribuem significativamente para a robustez e a sustentabilidade do código a longo prazo, destacando-se como a escolha ideal para ambientes de desenvolvimento colaborativos e dinâmicos.

4.3 Análise de Escalabilidade

As vantagens e desafios de cada abordagem são destacados na análise de escalabilidade entre o uso de um *Design System* e o desenvolvimento tradicional de SPA. O *Design System* oferece uma estrutura centralizada e reutilizável de componentes, promovendo consistência visual e funcionalidade em múltiplos projetos. Em contrapartida, o desenvolvimento tradicional pode levar à redundância de código e a uma manutenção mais complexa.

Ao utilizar um *Design System*, a reutilização de componentes é significativamente maximizada, o que resulta em uma experiência de usuário coesa e em uma redução considerável do tempo de desenvolvimento e da introdução de erros. Sistemas de Design fornecem uma coleção de elementos funcionais reutilizáveis, como componentes e padrões de interação, garantindo consistência e eficiência em todos os produtos digitais de uma empresa (“What are Design Systems?”, 2024). A manutenção centralizada permite a aplicação de atualizações e correções de forma uniforme, propagando mudanças automaticamente para todos os projetos que utilizam o sistema, o que pode reduzir o tempo de manutenção de forma significativa (CHAVARRIA, [s.d.]).

A escalabilidade é limitada pela redundância de código e pela manutenção dispersa em projetos tradicionais, nos quais podemos ter diversas aplicações que compartilham recursos em comum mas não possuem uma base única de compartilhamento, onde os componentes são desenvolvidos individualmente para cada aplicação. Cada projeto deve ser atualizado separadamente, aumentando o risco de inconsistências e erros. A redundância de código pode resultar em maiores tempos de desenvolvimento e maiores probabilidades de erros (CHAVARRIA, [s.d.]).

No projeto desenvolvido neste trabalho, por se tratar de uma implementação simples em SPA, não se consegue ver com clareza a vantagem de escalabilidade de um Design System. No entanto, ao criar um outro projeto em separado, onde há a necessidade de utilizar os mesmos componentes, começa-se a notar a redundância de código e a dificuldade de manutenção. Um exemplo claro da necessidade e vantagem de um Design System é ao desenvolver uma aplicação com dois pontos de entrada ligados a projetos diferentes, como

uma área de usuário logado e uma área de administrador. Ambas estão em projetos distintos, mantidas por diferentes equipes, mas compartilham o mesmo tema e *design*. Neste cenário, a centralização dos componentes no *Design System* facilita a manutenção e atualização, garantindo que ambas as áreas permaneçam consistentes e alinhadas com as diretrizes de design estabelecidas.

Em conclusão, especialmente em ambientes colaborativos com múltiplos desenvolvedores e equipes, a escalabilidade do *Design System* supera as desvantagens do aumento inicial do tamanho do *bundle*. A abordagem centralizada e estruturada promove uma manutenção eficiente e uma consistência visual aprimorada, destacando-se como a escolha ideal para projetos de larga escala.

5 CONCLUSÃO

Este trabalho se propôs a desenvolver e implementar um *Design System* simplificado utilizando as tecnologias ReactJS, TypeScript e Material-UI (MUI). O objetivo principal foi avaliar a eficácia dessa abordagem em termos de consistência visual, eficiência de desenvolvimento, manutenção do código e escalabilidade, comparando-a com métodos tradicionais de desenvolvimento.

Através da aplicação do *Design System* em uma *Single Page Application* (SPA), foram realizados testes que evidenciaram diversas vantagens desta abordagem. Em termos de consistência visual, o *Design System* se mostrou superior ao garantir que todos os componentes seguissem as mesmas diretrizes de design, proporcionando uma experiência de usuário uniforme e coesa. Este fator é crucial em grandes projetos onde múltiplas equipes podem estar trabalhando simultaneamente, evitando divergências visuais e funcionais.

Em relação à eficiência no desenvolvimento, o *Design System* permitiu a reutilização de componentes bem definidos e documentados. Esta prática não só reduziu o tempo de desenvolvimento como também minimizou a probabilidade de erros. A centralização das alterações em um único sistema facilitou a aplicação de atualizações e correções, propagando-as automaticamente para todos os projetos que utilizam o sistema.

No que diz respeito à manutenção do código, a abordagem tradicional apresenta desafios significativos. A redundância de código e a necessidade de atualizar cada projeto individualmente aumentam o risco de inconsistências e erros. Por outro lado, a manutenção centralizada no Design System facilita a aplicação de mudanças e melhorias, assegurando que todas as aplicações mantenham a uniformidade e qualidade necessárias.

A análise de escalabilidade demonstrou que, embora o uso de um *Design System* possa aumentar inicialmente o tamanho do *bundle* devido ao overhead do pacote, a complexidade do código customizado é reduzida. Em projetos grandes, a centralização das alterações e a reutilização de componentes resultam em um código mais robusto e sustentável a longo prazo, facilitando a manutenção e a atualização. A eficiência ganha em manutenção e a facilidade de atualização superam as desvantagens do aumento inicial do *bundle*, especialmente em ambientes colaborativos com múltiplos desenvolvedores e equipes.

Para aprimorar ainda mais o *Design System*, sugere-se a inclusão de práticas avançadas de acessibilidade, garantindo que os componentes sejam utilizáveis por pessoas com diferentes necessidades. A implementação de ferramentas automatizadas de verificação de acessibilidade pode assegurar que todos os componentes atendam aos padrões

estabelecidos. Além disso, a otimização contínua do bundle através de técnicas avançadas de *tree-shaking*[18] e *code-splitting*[19] pode reduzir ainda mais o tamanho dos arquivos finais, melhorando a performance das aplicações. Estas melhorias e trabalhos futuros podem ser trabalhadas como:

- Inclusão de novos componentes e padrões de design para cobrir uma gama mais ampla de casos de uso.
- Testar a integração do *Design System* com outras bibliotecas de front-end para verificar a compatibilidade e versatilidade.
- Implementar práticas avançadas de acessibilidade e realizar testes com usuários reais para garantir que o *Design System* seja inclusivo.
- Explorar técnicas avançadas de otimização de *bundle*, como *tree-shaking* e *code-splitting*, para melhorar a performance das aplicações.
- Aplicar o *Design System* em projetos reais e documentar os resultados para validar sua eficácia em diferentes contextos.
- Desenvolver um conjunto abrangente de testes automatizados para garantir a robustez e a confiabilidade dos componentes do *Design System*.

Por fim, ficam claramente evidentes as vantagens da utilização de um *Design System*. Ao analisar os benefícios em termos de consistência visual, eficiência no desenvolvimento, manutenção de código e escalabilidade, fica evidente que esta abordagem moderna supera os métodos tradicionais de desenvolvimento. Assim, este estudo não apenas reforça a relevância dos *Design Systems* na prática contemporânea de desenvolvimento web, mas também estabelece uma base sólida para futuras investigações e inovações no campo.

[18] Tree-shaking é uma técnica de otimização que elimina código morto (ou não utilizado) durante o processo de build, resultando em um bundle menor e mais eficiente.

[19] Code-splitting é a prática de dividir o código da aplicação em vários pequenos bundles que podem ser carregados sob demanda.

REFERÊNCIAS

STEFANOV, S. **JavaScript Patterns.** 1. ed. Heidelberg, Germany: O'Reilly, 2011.

FAIN, Y.; MOISEEV, A. **TypeScript quickly.** New York, NY: Manning Publications, 2020.

VESSELOV, S.; DAVIS, T. **Building Design Systems: Unify User Experiences Through a Shared Design Language.** [s.l: s.n.].

SATZINGER, J. W.; JACKSON, R. B.; BURD, S. D. **Systems analysis and design in a changing world**. 8. ed. Taipei, Taiwan: Cengage Learning, 2018.

SHNEIDERMAN, B.; PLAISANT, C. **Designing the user interface: Strategies for effective human-computer interaction: International edition**. 4. ed. Upper Saddle River, NJ: Pearson, 2004.

KHOLMATOVA, A. **Design Systems: A Practical Guide to Creating Design Languages for Digital Products.** [s.l.] Smashing Magazine, 2017.

DUCKETT, J. **HTML and CSS: Design and Build Websites.** 1. ed. Nashville, TN: John Wiley & Sons, 2014.

ROBSON, E.; FREEMAN, E. **Head first HTML and CSS: A learner's guide to creating standards-based web pages.** 2. ed. Sebastopol, CA: O'Reilly Media, 2012.

MARCOTTE, E. **Responsive Web Design.** [s.l.] A Book Apart, 2011.

PILGRIM, M. **HTML5: Up and Running.** 1. ed. Sebastopol, CA: O'Reilly Media, 2010.

ROBBINS, J. **Learning Web Design: A Beginner's Guide to HTML, CSS, JavaScript, and Web Graphics.** Sebastopol, CA: O'Reilly Media, 2018.

CROCKFORD, D. **JavaScript: The Good Parts**. Sebastopol, CA: O'Reilly Media, 2008.

FLANAGAN, D. **JavaScript - The Definitive Guide.** Sebastopol, CA: O'Reilly Media, 2020.

OVCHINNIKOVA, N. **Typescript vs. Javascript: The Key Differences You Should Know in 2023.** Flatlogic, 11 Feb. 2022. Disponível em: <https://flatlogic.com/blog/typescript-vs-javascript-the-key-differences-you-should-know-in-2020/>. Acesso em: 10 may. 2024

MCFARLAND, D. S. **JavaScript & jQuery: The Missing Manual 3e.** Sebastopol, CA: O'Reilly Media, 2014.

MICROSOFT CORPORATION. **The TypeScript Handbook.** [s.l: s.n.]. Disponível em: <https://www.typescriptlang.org/docs/handbook/intro.html>.

FROST, B. **Atomic Design.** [s.l.] Brad Frost, 2016.

GOOGLE. **Material Design.** Disponível em: <https://material.io/design>. Acesso em: 10 may. 2024.

MUI: The ReactJS component library you always wanted. Disponível em: <https://mui.com>. Acesso em: 10 may. 2024.

ReactJS. Disponível em: <https://reactjs.org/.>. Acesso em: 10 may. 2024.

STEFANOV, S. **React: Up & running: Building web applications.** 2. ed. Sebastopol, CA: O'Reilly Media, 2021.

FEDOSEJEV, A. **React.js Essentials.** Birmingham, England: Packt Publishing, 2023.

FREEMAN, A. **Pro React 16.** 1. ed. Berlin, Germany: APress, 2019.

ANTHONY, A.; NATHANIEL, M.; ARI, L. **Fullstack react: The complete guide to reactjs and friends.** [s.l.] Fullstack.IO, 2017.

DERKS, R. **React Projects: Build advanced cross-platform projects with React and React Native to become a professional developer.** 2. ed. Birmingham, England: Packt Publishing, 2022.

SONARSOURCE. **What is a Linter? Lint Code Definition & Guide.** SonarSource, [s.d.]. Disponível em: <https://www.sonarsource.com/learn/linter/>. Acesso em: 10 may. 2024

KRYSIK, A. **Documenting Software in Development Projects: Types, Best Practices, and Tools.** Stratoflow, 7 Aug. 2023. Disponível em: <https://stratoflow.com/documenting-software/>. Acesso em: 10 may. 2024

The Crucial Role of Documentation in Software Development. Mach One Digital, [s.d.]. Disponível em: <https://www.machonedigital.com/blog/the-crucial-role-of-documentation-in-software-development>. Acesso em: 10 may. 2024

AKHTAR, H. **Storybook for React.** BrowserStack, 9 Aug. 2023. Disponível em: <https://www.browserstack.com/guide/storybook-for-react>. Acesso em: 10 may. 2024

STORYBOOK. **How to document components.** [s.l: s.n.]. Disponível em: <https://storybook.js.org/docs/writing-docs>.

CHAVARRIA, L. **Reusability, scalability and consistency in design and code**. Disponível em: <https://www.keen.design/en/design-blog/design-systems>. Acesso em: 10 may. 2024.

What are Design Systems? — updated 2024. Disponível em: <https://www.interaction-design.org/literature/topics/design-systems>. Acesso em: 10 may. 2024.

Vite. Disponível em: <https://vitejs.dev/guide/>. Acesso em: 16 maio. 2024a

APÊNDICE A — IMPLEMENTAÇÃO DO TEMA CUSTOMIZADO PARA O MUI

```
import { ThemeOptions, createTheme } from "@mui/material";

export const breakpoints = {
  values: {
    xs: 0,
    sm: 600,
    md: 960,
    lg: 1280,
    xl: 1920,
  },
};

const theme: ThemeOptions = {
  breakpoints,
  shape: {
    borderRadius: 0,
  },
  palette,
  typography,
};

export const defaultTheme = createTheme(theme);

export const palette = {
  text: {
    primary: "#01090f",
    secondary: "#5c5c5c",
    disabled: "#7c8286",
  },
  primary: {
    main: "#3D8EC9", // Cor principal
    light: "#62B2EB", // Variante clara
    dark: "#2A6BA1", // Variante escura
    contrastText: "#ffffff", // Texto em contraste
  },
  secondary: {
    main: "#EFEFEF", // Cor principal
```

```
    light: "#FFFFFF", // Variante clara
    dark: "#C0C0C0", // Variante escura
    contrastText: "#2c2c2c", // Texto em contraste
  },
  error: {
    main: "#CE0000", // Cor principal
    light: "#FF3333", // Variante clara
    dark: "#990000", // Variante escura
    contrastText: "#ffffff", // Texto em contraste
  },
  warning: {
    main: "#F9A13A", // Cor principal
    light: "#FFB56B", // Variante clara
    dark: "#CC7D00", // Variante escura
    contrastText: "#ffffff", // Texto em contraste
  },
  success: {
    main: "#34B831", // Cor principal
    light: "#66DC66", // Variante clara
    dark: "#007D00", // Variante escura
    contrastText: "#ffffff", // Texto em contraste
  },
  info: {
    main: "#62B2EB", // Cor principal
    light: "#8CD3FF", // Variante clara
    dark: "#3A90D6", // Variante escura
    contrastText: "#ffffff", // Texto em contraste
  },
  divider: "#E0E0E0", // Cor do divisor
} satisfies ThemeOptions["palette"];


import { ThemeOptions } from "@mui/material";

export const typography = {
  fontFamily: ["Helvetica", "sans-serif"].join(","),
  h1: {
    fontSize: 96,
    letterSpacing: "-1.5px",
```

```
  fontWeight: "600",
},
h2: {
  fontSize: 60,
  letterSpacing: "-0.5px",
  fontWeight: "400",
},
h3: {
  fontSize: 48,
  letterSpacing: "0px",
  fontWeight: "600",
},
h4: {
  fontSize: 34,
  letterSpacing: "0.25px",
  fontWeight: "600",
},
h5: {
  fontSize: 24,
  letterSpacing: "0px",
},
h6: {
  fontSize: 20,
  letterSpacing: "0.15px",
  fontWeight: "600",
},
subtitle1: {
  fontSize: 16,
  letterSpacing: "0.15px",
  fontWeight: "400",
},
subtitle2: {
  fontSize: 14,
  letterSpacing: "0.1px",
  fontWeight: "600",
},
body1: {
  fontSize: 16,
  letterSpacing: "0.5px",
```

```
  },
  body2: {
    fontSize: 14,
    letterSpacing: "0.25px",
  },
  button: {
    fontSize: 14,
    letterSpacing: "1.25px",
    textTransform: "uppercase",
  },
  caption: {
    fontSize: 12,
    letterSpacing: "0.4px",
  },
} satisfies ThemeOptions["typography"];
```

APÊNDICE B — EXEMPLO IMPLEMENTAÇÃO DA DOCUMENTAÇÃO DO COMPONENTE BUTTON NO STORYBOOK

```
import type { Meta, StoryObj } from "@storybook/react";
import { fn } from "@storybook/test";
import Button from "./Button";
import { Stack } from "@mui/material";

const meta = {
  title: "Components/Button",
  component: Button,
  parameters: {
    layout: "centered",
  },
  tags: ["autodocs"],
  argTypes: {
    onClick: {
      type: "function",
      action: "clicked",
      description: "Click event handler",
    },
    children: {
      control: "text",
      description: "The content of the component",
      defaultValue: "Button",
    },
    classes: {
      control: "object",
      description: "Override or extend the styles applied to the component.",
      defaultValue: {},
    },
    color: {
      control: {
        type: "radio",
        options: [
          "primary",
          "secondary",
          "success",
          "error",
```

```
      "info",
      "warning",
    ],
    labels: {
      primary: "Primary",
      secondary: "Secondary",
      success: "Success",
      error: "Error",
      info: "Info",
      warning: "Warning",
    },
  },
  description:
    "The color of the component. It supports both default and custom theme colors, which can be added.",
  defaultValue: "primary",
},
component: {
  control: "text",
  description:
    "The component used for the root node. Either a string to use a HTML element or a component.",
  defaultValue: undefined,
},
disabled: {
  control: "boolean",
  description: "If true, the component is disabled.",
  defaultValue: false,
},
disableElevation: {
  control: "boolean",
  description: "If true, no elevation is used.",
  defaultValue: false,
},
disableFocusRipple: {
  control: "boolean",
  description: "If true, the keyboard focus ripple is disabled",
  defaultValue: false,
},
disableRipple: {
  control: "boolean",
```

```
  description: "If true, the ripple effect is disabled.",
  defaultValue: false,
},
endIcon: {
  control: "text",
  description: "Element placed after the children.",
  defaultValue: undefined,
},
fullWidth: {
  control: "boolean",
  description:
    "If true, the button will take up the full width of its container.",
  defaultValue: false,
},
href: {
  control: "text",
  description: "The URL to link to when the button is clicked.",
  defaultValue: undefined,
},
size: {
  control: {
    type: "radio",
    options: ["small", "medium", "large"],
  },
  description: "The size of the component.",
  defaultValue: "medium",
},
startIcon: {
  control: "text",
  description: "Element placed before the children.",
  defaultValue: undefined,
},
sx: {
  control: "object",
  description:
    "The system prop that allows defining system overrides as well as additional CSS styles.",
  defaultValue: {},
},
variant: {
```

```
    control: {
      type: "radio",
      options: ["text", "outlined", "contained"],
    },
    description: "The variant to use.",
    defaultValue: "contained",
  },
 },
 args: {
  onClick: fn(),
  children: "",
  classes: {},
  color: "primary",
  component: undefined,
  disabled: false,
  disableElevation: false,
  disableFocusRipple: false,
  disableRipple: false,
  endIcon: undefined,
  fullWidth: false,
  href: undefined,
  size: "medium",
  startIcon: undefined,
  sx: {},
  variant: "contained",
 },
} satisfies Meta<typeof Button>;

export default meta;
type Story = StoryObj<typeof meta>;

export const Example: Story = {
 render: (props) => (
  <Stack spacing={2}>
   <Button {...props} color="primary">
    Primary
   </Button>
   <Button {...props} color="secondary">
    Secondary
```

```
    </Button>
    <Button {...props} color="primary" size="large">
      Large
    </Button>
    <Button {...props} color="primary" size="small">
      Small
    </Button>
    <Button {...props} color="primary" disabled>
      Disabled
    </Button>
   </Stack>
  ),
};

export const Primary: Story = {
  render: (props) => <Button {...props}>Primary</Button>,
};

export const Secondary: Story = {
  args: {
    color: "secondary",
  },
  render: (props) => <Button {...props}>Secondary</Button>,
};

export const Large: Story = {
  args: {
    size: "large",
  },
  render: (props) => <Button {...props}>Large</Button>,
};

export const Small: Story = {
  args: {
    size: "small",
  },
  render: (props) => <Button {...props}>Small</Button>,
};
```

```
export const Disabled: Story = {
  args: {
    disabled: true,
  },
  render: (props) => <Button {...props}>Disabled</Button>,
};

export const Loading: Story = {
  args: {
    size: "small",
  },
};
```