PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
ESCOLA POLITÉCNICA E DE ARTES
ENGENHARIA DE COMPUTAÇÃO

PLATAFORMA DE EDIÇÃO/AUTOMAÇÃO PARA TRABALHOS ACADÊMICOS

HIGOR FERREIRA ALVES SANTOS

GOIÂNIA - GO
2024

HIGOR FERREIRA ALVES SANTOS

PLATAFORMA DE EDIÇÃO/AUTOMAÇÃO PARA TRABALHOS ACADÊMICOS

Trabalho de Conclusão de Curso apresentado à Escola Politécnica e de Artes, da Pontifícia Universidade Católica de Goiás, como parte dos requisitos para a obtenção do título de Bacharel em Engenharia de Computação.

Orientador:

Prof. M.E.E. Marcelo Antonio Adad de Araújo

Banca examinadora:

Prof. Me Miriam Sandra Rosa Gusmão
Prof. M.E.E. Carlos Alexandre Ferreira de Lima

GOIÂNIA - GO
2024

HIGOR FERREIRA ALVES SANTOS

PLATAFORMA DE EDIÇÃO/AUTOMAÇÃO PARA TRABALHOS ACADÊMICOS

Trabalho de Conclusão de Curso aprovado em sua forma final pela Escola Politécnica e de Artes, da Pontifícia Universidade Católica de Goiás, para obtenção do título de Bacharel em Engenharia de Computação, em: ____/ ____/ _____

Prof. Me. Ludmillla Reis Pinheiro dos Santos
Coordenadora de Trabalho de Conclusão de Curso

Orientador: Prof. M.E.E.Marcelo Antonio Adad de Araújo

Banca: Prof. Me Miriam Sandra Rosa Gusmão

Banca: Prof. M.E.E. Carlos Alexandre Ferreira de Lima

GOIÂNIA - GO
2024

DEDICATÓRIA

Dedico a Aquele que se assenta inerte no trono de glória, em algum lugar remoto do Universo.

À Pátria amada, vestida em sua pele de cordeiro. Que promove a bondade e a justiça do alto de suas majestosas pirâmides sociais, sustentadas sob os pilares de sal da ambivalência.

À minha família, autora das maiores alegrias e melancolias que a vida pode trazer.

A Deus, à Pátria, à Família!

Ao meu pai: Euripedes Alves dos Santos; minha mãe: Maria Aparecida Ferreira Gomes dos Santos; e minha irmã: Sthefany Ferreira Alves dos Santos.

Em memória do meu avô: Gabriel Ferreira Gomes, falecido durante o processo de escrita deste trabalho.

Em memória de minhas avós: Maria Tavares dos Santos e Jesuína Pereira dos Santos.

Às amizades de qualidade conquistadas, cujo julgo compartilhado torna a caminhada mais leve.

Às criaturas do Vale, cuja matéria de acolhimento e aceitação as vezes é tão boa quanto as extremas opostas criaturas do Caminho.

AGRADECIMETOS

Agradeço a cada um que fez parte da minha jornada e história. Algumas pessoas passam pelas nossas vidas, marcam, e seguem seu caminho.

Minha imensa gratidão ao grupo de jovens da extinta comunidade Viver em Cristo, sob a liderança do Christian Alberto Lopes Clemente. Grupo este composto de excelentes pessoas que compartilharam suas vivências e experiências.

Aos amigos divergentes Matheus Silva Sales e sua esposa Isabella Souza Sales, nossos caminhos se divergem, porém a caminhada conjunta foi de papel extremamente importante em minha jornada, especialmente no que tange ao êxito em vencer os mais obscuros poços da apatia e do transtorno depressivo maior.

Ao meu amigo Ismael Carlos do Nascimento Galvão. Não esquecerei as noites em claro de ligações, onde nos piores momentos ele fez sua presença, de forma cuidadosa, carinhosa e humanitária, ele esteve presente comigo num dos piores momentos de vida que alguém pode passar. Meu coração está contigo.

Aos amigos conquistados nesta instituição: Lais Pereira Felipe; João Victor Banczek Rodrigues; Artur Souza Dias; Dayana Stefany Costa Pamplona; Lucas Silva Queiroz; Gabriel da Costa Diniz; Gabriel Soares Ribeiro; e tantos outros que enriqueceram esta caminhada, fazendo-a valer à pena.

À melhor equipe de desenvolvimento que tenho o prazer de fazer parte: Rogê Batista Gonçalves e Tiago Siqueira Oliveira. Vocês foram compreensíveis comigo em toda a jornada de escrita deste trabalho e não mediram esforços em tornar as coisas mais fáceis para mim. Minha imensa gratidão.

Ao corpo doscente da Pontifícia Universidade Católica de Goiás, especialmente na figura do meu orientador: Marcelo Antonio Adad de Araújo, e da banca: Miriam Sandra Rosa Gusmão e Carlos Alexandre Ferreira de Lima.

RESUMO

Escrever um Trabalho de Conclusão de Curso (TCC) pode ser uma tarefa dolorosa e penosa. Além de se preocupar com o conteúdo em si, os discentes muitas vezes devem se peocupar também com formatação e apresentação visual de seus projetos. As normas brasileiras de formatação reguladas pela Associação Brasileira de Normas Técnicas (ABNT) muitas vezes se tornam um gigantesco desafio para os egressos Brasileiros. Estas normas contém diversas regras e especificidades que devem ser seguidas a risca.

Considerando tais dificuldades, este trabalho consiste em uma implementação na qual é desenvolvida uma plataforma de edição de texto simples, intuitiva e eficiente que possa ser facilmente utilizada pelo público geral. Desta forma, o escritor pode apenas se preocupar com seu conteúdo em si, deixando aspectos como formatação e apresentação a cargo da plataforma.

O produto final consiste numa plataforma com um editor de texto visual que segue as normas da ABNT em conjunto com as normas específicas da Pontificia Universidade Católica de Goiás afim de ajudar os alunos formandos em suas publicações.

ABSTRACT

Writing a final paper or a monograph can be a difficult task. In addition to worrying about the content of the work itself, writers must also focus on the presentation and formatting of their project. Brazilian standards for formatting scientific papers are among the most challenging aspects for Brazilian students. These standards contain many rules and specificities that must be followed strictly.

Considering these difficulties, this work involves the implementation of an application that aims to develop a rich text editor that can be used simply and efficiently by the general public. In this way, the writer only has to worry about the content, and the presentation is automatically formatted by the app.

The final product is a platform with a visual editor that follows the rules of the "Associação Brasileira de Normas Técnicas" (ABNT), the association that regulates formatting norms, in conjunction with rules provided by the Pontifical Catholic University of Goiás (Pontifícia Universidade Católica de Goiás) to help PUC-GO students with their academic publications.

Lista de ilustrações

Lista de tabelas

LISTA DE SIGLAS

ABNT Associação Brasileira de Normas Técnicas. 1, 2, 5, 6, 23, 28

API Application Programming Interface. 18, 19, 22, 26, 28, 39–41, 49, 50, 56, 57

CSS Cascading Style Sheet. 11, 18, 21

DOM Document Object Model. 16, 28, 45, 47, 48, 57

ECMA European Computer Manufacturers Association. 12

IES Instituição de Ensino Superior. 1, 2

M.E.E Mestre em Engenharia Elétrica. 1, 2

NBR Norma Brasileira Regulamentadora. 1

PDF Portable Document Format. 2–5, 12, 23, 78

PUC-GO Pontifícia Universidade Católica de Goiás. 6, 23, 71

SPA Single Page Application. 28

SSR Server Side Rendering. 15

SVG Scalable Vector Graphics. 11

TCC Trabalho de Conclusão de Curso. 2, 5

UFPB Universidade Federal da Paraíba. 2

UI User Interface. 20, 67

UX User eXperience. 20, 44

XML eXtensible Markup Language. 11, 28

LISTA DE ABREVIATURAS

App Application. 32, 33

bash Bourne Again Shell. 29

CERN Conseil Européen pour la Recherche Nucléaire. 10

HTML HyperText Markup Language. 10, 11, 16, 19, 21, 23, 28, 37, 55, 60, 63, 66–68

JS JavaScript. 12, 13, 16, 18, 21, 26, 61, 69

JSON JavaScript Object Notation. 5, 13, 19

JSX JavaScript XML. 16, 33, 37

LaTex Lamport TeX. 5, 6, 23, 26, 27, 59, 61, 62, 64–68, 70, 72–74, 76, 78–80

MathML Mathematical Markup Language. 11

Me Mestre(a). 1, 2

PHP Hypertext Preprocessor. 14, 15

Prof Professor(a). 1, 2

RegEx Regular Expression. 23, 26

RegExp Regular Expression. 23

TS TypeScript. 12, 13

TSX TypeScript XML. 33, 34, 37

UUID Universally Unique Identifier. 72

uuid Universally Unique Identifier. 72, 73

W3C World Wide Web Consortium. 10, 11

Web World Wide Web. 2, 9, 11, 12, 14, 17, 22, 29, 78

XHTML eXtensible HyperText Markup Language. 11

XSS Cross-Site Scripting. 14

Sumário

1 INTRODUÇÃO

Escrever um trabalho científico pode ser uma tarefa desafiadora. (SEVERINO, 2017) destaca a complexidade e o rigor necessários na elaboração de trabalhos científicos, que não apenas envolvem o domínio do conteúdo específico, mas também a aderência às normas técnicas para apresentação formal e formatação correta.

A Associação Brasileira de Normas Técnicas (ABNT) , é a entidade responsável por, dentre outras, fornecer as normas que regulam o processo de criação de trabalhos acadêmicos. A Norma Brasileira Regulamentadora (NBR) Nº 14724, por exemplo: Especifica os princípios gerais para a elaboração de (teses, dissertações e outros), visando sua apresentação à instituição (banca, comissão examinadora de professores, especialistas designados e/ou outros) (ABNT, 2021).

Ademais, ainda com respeito aos trabalhos acadêmicos, não somente a regulamentação da NBR 14724 deve ser observada. Há ainda a NBR 6023 que trata a respeito da elaboração de referências e a NBR 10520, que diz respeito às citações em documentos.

(CASTRO, 2011), adverte que: "Em ciência, não pode haver uma separação entre forma e conteúdo. Trata-se de uma separação fictícia, pois fica se conhecendo o conteúdo pela forma." Ou seja: A forma do trabalho, sua apresentação, sua formatação e todo o seu arranjo gráfico é tão importante quanto seu conteúdo. (MEDEIROS, 2012) vai complementar essa visão, afirmando que a apresentação gráfica "[...] contribui para a consecução de um trabalho capaz de atingir seu objetivo. Monografia realizada sem a preocupação gráfica, em geral, acaba malsucedida."

Em seu artigo, (SILVA; VITORIA, 2014) vão analisar as percepções e dificuldades dos alunos de um curso superior em Tecnologia de Gestão em Recursos Humanos. Dentre suas dificuldades, (dos alunos em questão), é destacada a questão da formatação do trabalho acadêmico. Há também o fato de que as bancas avaliam os trabalhos baseadas em critérios da própria Instituição de Ensino Superior (IES), critérios estes que não estão necessariamente presentes nas normas da ABNT, ou seja, há uma subjetividade presente que não é comum a todas às IES quanto a questão da formatação. Essa subjetividade contribui para a confusão dos alunos, pois a IES avaliará de acordo com aquilo que julga apropriado, o que muitas vezes pode obscurecer o direcionamento do aluno ao redigir/formatar seu trabalho."

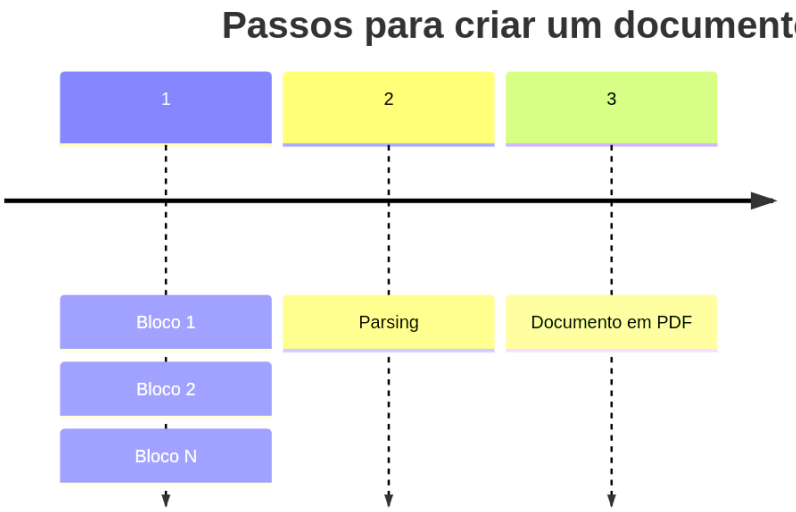
(SANTOS, 2020) em seu Trabalho de Conclusão de Curso (TCC) , também analisa as dificuldades encontradas por egressos, desta vez do curso de Ciências Contábeis da Universidade Federal da Paraíba (UFPB). Em sua pesquisa é destacado que "Quanto a formatação do trabalho com as normas da ABNT, [...], 60% teve alguma dificuldade, inclusive 32% teve muita dificuldade.", ou seja, a formatação do trabalho é um grande desafio presente na vida de boa parte dos estudantes em processo de escrita.

1.1 Objetivo

Levando em consideração os problemas que os alunos de diversas instituições de ensino enfrentam ao elaborar seus respectivos trabalhos (conforme apresentado acima), o objetivo deste instrumento é desenvolver uma plataforma Web de alta interatividade[1] e inteligibilidade[2], de modo que o discente possa se preocupar apenas com o conteúdo. Os detalhes de formatação, de acordo com os padrões da ABNT e da IES, ficarão a cargo da própria plataforma.

A criação de um trabalho de TCC se dará basicamente por três passos básicos: Escrita em blocos; *Parsing*[3] e Documento em PDF . A Figura 1 ilustra esse fluxo na linha do tempo.

Figura 1 – Passo a passo para criar um documento na plataforma



Fonte: Autoria própria

[1] Refere-se à capacidade de um sistema, aplicação ou interface de responder às ações do usuário de maneira eficaz e intuitiva

[2] Refere-se à clareza e compreensibilidade da interface, documentação e feedback fornecidos pelo sistema. Um *software* inteligível facilita o entendimento do usuário sobre como utilizá-lo e quais são os resultados de suas ações.

[3] O termo *Parsing*, (do inglês: análise), será utilizado no sentido de analisar e transformar algo em outra coisa.

O usuário interagirá com a aplicação escrevendo blocos que serão transformados no documento final em PDF . A este processo denominar-se-á *Parsing*. Após este, bastará enviar o download do PDF ao usuário com todo o padrão de formatação. Os trabalhos desenvolvidos nesta plataforma terão então duas versões: A versão de blocos, (sem formatação e interativa); e a versão final já formatada em PDF.

1.2 Fluxo do documento

1.2.1 Escrita em blocos

A escrita se dará de modo em que tudo será considerado um bloco. A escrita em blocos consiste numa abordagem em que o texto vai sendo escrito em seu fluxo natural, porém blocos podem ser adicionados à escrita. Um bloco é um elemento adicionado ao fluxo de trabalho que desempenha um papel que o diferencia dos demais blocos. Por exemplo: Uma imagem pode ser considerada um bloco nesta abordagem, uma vez que não é um texto mas tem o objetivo de fornecer informações visuais. O próprio corpo do texto em si será considerado um bloco, denominado parágrafo. Um título será um bloco textual cujo objetivo será separar sessões do texto coesas. Uma lista será um bloco para enumerar itens e assim por diante. O documento será basicamente uma composição de diversos blocos dispostos de forma a formar uma unidade coesa final, que será o trabalho propiamente dito.

1.2.2 Bloco

Um bloco é uma unidade lógica no documento que desempenha um papel especializado que nenhum outro bloco o faz. Por exemplo: O bloco mais importante da plataforma[4] será o de texto, (denominado bloco parágrafo), pois sem texto, não há trabalho. Sem texto não há tão pouco comunicação que transmita informação de caráter acadêmico-científico.

Semelhante ao bloco de texto, diversos outros blocos adjascentes auxiliarão na construção do documento acadêmico. O bloco de imagem, por exemplo, ajuda a exibir informações de forma ilustrativa e auxilia bastante em exemplos que estão sendo dados em determinado contexto do texto.

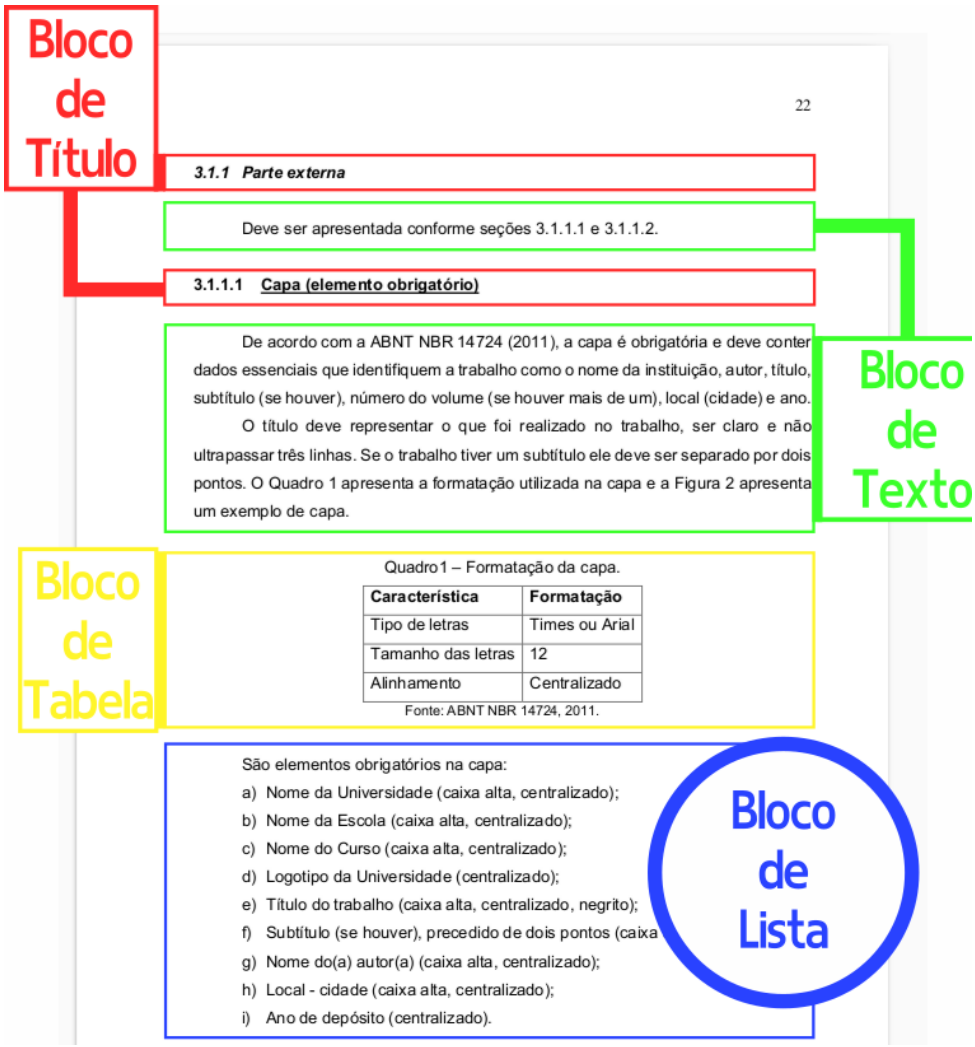
A maior parte dos blocos contará com uma espécie de submenu, (em termos de aplicação), que os permita personalizar. A personalização de blocos é importante para editar configurações e dar autonomia ao usuário em determinar mais precisamente o papel daquele bloco no texto. Por exemplo: Um bloco de título ajuda a separar o texto em unidades coesas. Porém, existem diversos tipos de títulos: Existe o título, o subtítulo, e até o subtítulo do subtítulo.

[4] O termo plataforma será utilizado de forma intercambiável e como sinônimo de aplicativo; sistema web; ou aplicativo da web

O submenu será a configuração que o usuário fará no bloco após escolhê-lo. No caso do título, por exemplo: Após o usuário escolher este bloco, poderá configurar o nível de título desejado. Nível este que varia do 1 ao 5, sendo 2; 3; 4 e 5 espécies de subtítulos. No caso de uma imagem, o submenu funcionará para que possa ser definida a imagem, bem como seu título de sua descrição.

A Figura 2 ilustra a composição de um trabalho com seus respectivos blocos:

Figura 2 – Divisão de blocos em um documento



Fonte: Adaptado de (PUC-GO, 2022)

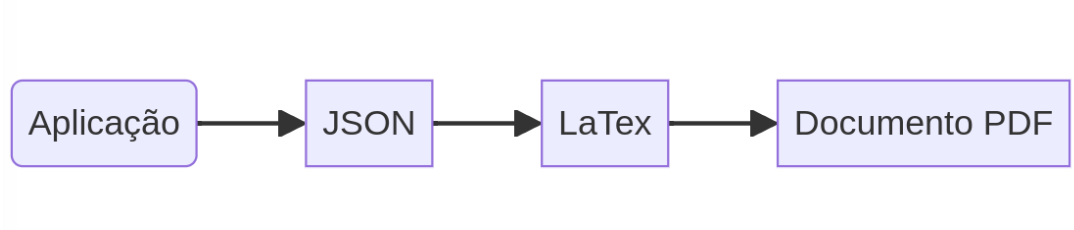
1.2.3 Parsing

O processo de *Parsing* é o processo que acontecerá sempre que o usuário desejar ver o *layout*[5] da versão final de seu trabalho. Ele usa o código intermediário gerado pelos blocos para montar o PDF final.

[5] Do inglês: Disposição, ou esboço. Esta palavra geralmente está associada ao desenho ou visual de algo.

Este processo é, em termos simples, uma espécie de análise a ser aplicada no código gerado pelos blocos da aplicação. A plataforma gerará um código JSON[6] como resultado das interações do usuário, que posteriormente serão convertidos em código LaTex[7]. Só então, finalmente será utilizado um utilitário que converterá o código LaTex em um documento PDF. A Figura 3 ilustra esse processo:

Figura 3 – Etapas do processo de Parsing



Fonte: Autoria própria.

1.3 Ambiente de desenvolvimento

O ambiente de desenvolvimento é de extrema importância para que todas as ferramentas utilizadas possam funcionar em perfeita harmonia em suas respectivas integrações e colaborações. Muitas vezes, problemas de compatibilidade podem afetar o funcionamento das mesmas e impedir que o programa final seja executado corretamente, causando *bugs*[8] e outros imprevistos impeditivos tanto para a correta execução, quanto para a exeperiência de desenvolvimento. A Tabela 1 diz respeito às ferramentas e ao ambiente onde este *software*[9] foi desenvolvido, bem como todas as suas respectivas versões:

[6] Ver (sessão que trata do JSON)
[7] Ver (sessão que trata do LaTex)
[8] Do inglês: Inseto. Esta palavra é muito utilizada no contexto de desenvolvimento de aplicativos para se referir a problemas que afetam o funcionamento dos mesmos
[9] O *software* é o conjunto de instruções dadas a um computador, de modo que ele execute determinada tarefa. Pode-se dizer que o *software* é a parte lógica do sistema computacional. (MACHADO,).

1.3.1 Tecnologias do ambiente de desenvolvimento

Atender aos requisitos mínimos de *hardware*[10] e *software* é fundamental para garantir uma experiência de usuário satisfatória e evitar problemas de desempenho ou compatibilidade com o aplicativo da plataforma. A seguir na Tabela 1 enumera-se o ambiente mínimo com seus respectivos *softwares* necessários para rodar o aplicativo da plataforma:

Tabela 1 – Tecnologias do ambiente de desenvolvimento

Tipo	**Tecnologia**
Versionador	Git 2.34.1
AbnTeX2	1.9.7 2018-11-24
Sistema Operacional	Ubuntu 20.04
Gerenciador de pacotes	Npm 10.2.3
Interpretador	*NodeJs* 20.10.0
Utilitário	kpathsea version 6.3.4/dev
Linguagem de Programação	TypeScript 5.3.3
Gerenciador de pacotes	*Yarn* 1.22.19
Utilitário	BibTeX 0.99d (TeX Live 2022/dev/Debian)
Navegador de Internet	Google Chrome 119.0.6045.199
Compilador	pdfTeX 3.141592653-2.6-1.40.22 (TeX Live 2022/dev/Debian)
Utilitário	makeglossaries (Utilitário LaTex)

Fonte: Autoria própria

1.3.2 Tecnologias do projeto

A seguir na Tabela 2 estão listadas as tecnologias exatas do projeto juntamente com suas respectivas versões. As mesmas podem ser baixadas após o download do repositório com qualquer gerenciador de pacotes *NodeJs*, tais como: npm; *yarn*; pnpm e bun. Este trabalho utilizou o *yarn*:

[10] Com *hardware*, compreende-se o equipamento físico de um sistema computacional. Suas unidades Lógicas de Processamento, memórias e unidades de armazenamento são *hardware*. (MACHADO,).

Tabela 2 – Tecnologias do projeto

Nome	**Versão**
antd	^5.15.0
next	^14.1.1
react	^18.2.0
node-latex	^3.1.0
@editorjs/list	^1.9.0
@editorjs/header	^2.8.1
@emotion/react	^11.11.4
@emotion/styled	^11.11.0
@editorjs/editorjs	2.29.1
cheerio	^1.0.0-rc.12

Fonte: Autoria própria

1.3.3 Versionador, (Versão do projeto)

Este trabalho utiliza o Git para controlar suas respectivas versões e evolução. Como o código é dinâmico e está sempre evoluindo, os exemplos fornecidos neste documento estão registrados numa determinada versão específica afim de que não se perca o exemplo dado com código que poderá estar diferente dependendo da data em que o trabalho for lido.

Todos os exemplos em código fornecidos, bem como a correspodência de linhas estão presentes nos detalhes da versão de commit[11] abaixo:

```
1 commit c0cab19296c46355d4bee5cc3e164616ed73fe49
2 Author: Higor Ferreira <hfashigor@hotmail.com>
3 Date:   Sun Jun 9 11:03:02 2024 -0300
4
5     updateDocs: README
```

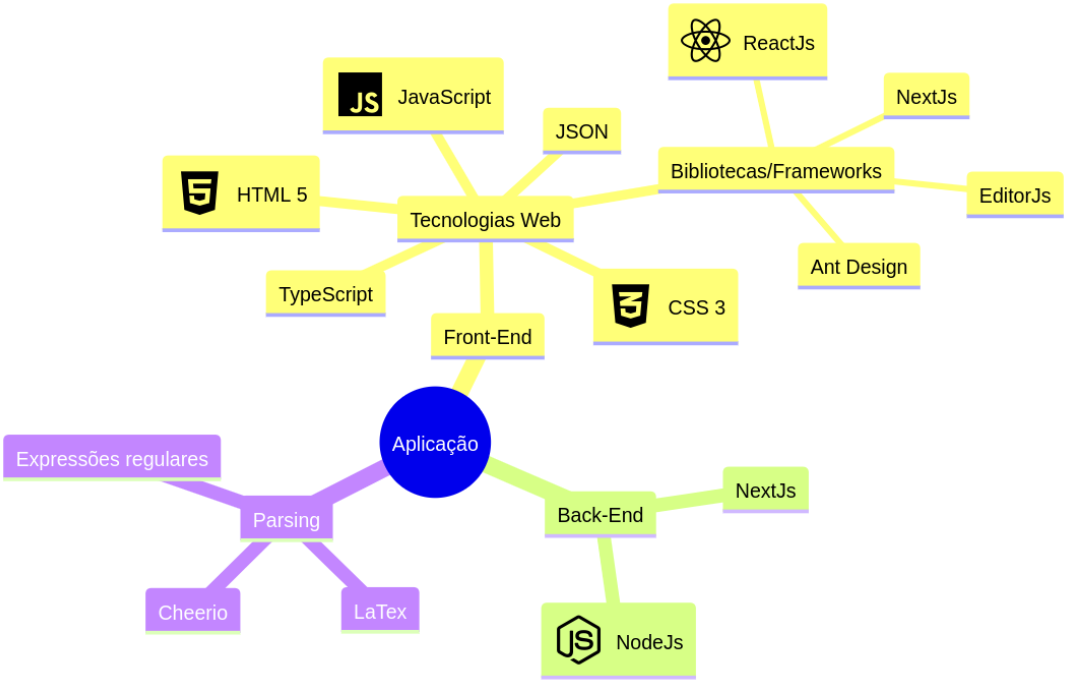
[11] Um commit, (nos termos do git), é como salvar o projeto, guardando uma versão do código naquele momento específico. O commit pode ser comparado a uma foto do projeto, capturando o estado atual e todas as mudanças feitas desde o último commit.

2 FUNDAMENTAÇÃO TEÓRICA

A plataforma será construída sob alguns pilares fundamentais indispensáveis a seu funcionamento. São estes pilares que garantirão o sucesso e o correto funcionamento da aplicação, afim de que todo o objetivo discutido até o presente momento seja atingido.

A Figura 4 mostra em forma de mapa mental todos os principais pilares sobre os quais o aplicativo será contruído. Estes pilares são formados por diversas tecnologias, bibliotecas, *frameworks*[1] e conceitos que deverão trabalhar de forma integrada.

Figura 4 – Pilares da plataforma, (mapa mental)



Fonte: Autoria própria.

Estes pilares estão subdivididos em três grandes subcategorias, a saber: *Front-End*; *Back-End* e *Parsing*. Cada qual com seus respectivos conceitos e tecnologias.

[1] Uma *framework* é como um kit de ferramentas pré-pronto que fornece uma gama de funcionalidades pré-construídas e testadas afim de facilitar o processo de desenvolvimento. (AMAZON; AWS,)

2.1 Do Front-End

O *Front-End* é, basicamente, a "linha de frente". É a parte da aplicação que interagirá diretamente com o usuário. Ao profissional que codifica e desenvolve esta parte do projeto, denomina-se Desenvolvedor *Front-End*. A interface do usuário, que é onde o mesmo realiza suas interações com o sistema, normalmente é desenhada por um *designer*[2] , ficando a cardo do desenvolvedor o papel de adaptar o *design*[3] ao código afim de obter os efeitos desejados. (TOTVS, 2021)

2.1.1 Tecnologias Web

As tecnologias Web desempenham um papel crucial na criação de experiências digitais interativas, permitindo que os usuários se envolvam com o conteúdo de maneira mais dinâmica e significativa. A incorporação da *Internet*[4] na vida diária resultou em mudanças significativas, marcada por um ritmo de evolução e aprimoramento sem precedentes, além da distribuição de conteúdo em massa. Juntamente com essas mudanças, surgiram novas tecnologias, variando de *softwares* a *hardwares*, aprimorando a experiência de navegação na Web (MOLGADO, 2016).

A Internet, que teve origem nos Estados Unidos em 1969, foi inicialmente utilizada por universidades, governos e instituições financeiras antes de se expandir globalmente. No início, a internet era uma via de mão única onde os usuários consumiam informações e se comunicavam de maneira privada. A evolução começou com a introdução de sistemas de busca avançados, destacando-se o lançamento do Google em 1998, que democratizou o acesso à informação. (VITORIANO, 2019).

A grande reviravolta na internet aconteceu em 1999, com o surgimento do *Blogger*, marcando o início da Web 2.0, onde a comunicação tornou-se bidirecional. Os usuários passaram a gerar conteúdo e se relacionar publicamente com marcas, empresas e pessoas por meio de comentários, além de consumir informação. A evolução da tecnologia móvel, em conjunto com o surgimento de redes sociais como *Fotolog*, *MySpace*, *Orkut*, *Facebook*, *YouTube* e *Twitter*, ampliou o conceito de Web 2.0, permitindo o compartilhamento de fotos, vídeos e textos em uma escala maior. (VITORIANO, 2019).

A forma como se interage com a internet também evoluiu ao longo do tempo. Passou-se de sites estáticos para interativos e animados, chegando até aos sites totalmente responsivos[5] e adaptáveis de hoje. Isso foi possível devido ao desenvolvimento de novos gadgets e ao surgimento de novas linguagens de programação. Atualmente,

[2] Profissional que atua com design.
[3] Do inglês: Desenho.
[4] Rede mundial de computadores, (BRASIL, 2014).
[5] A responsividade é a capacidade de uma página da Web em se adaptar a diferentes dispositivos e tamanhos de tela. (MDN, 2023b)

a Web Moderna é composta por várias técnicas, metodologias, linguagens e ferramentas que permitem o desenvolvimento de aplicações conectadas e interativas, oferecendo diversas formas de interação com interfaces digitais. (VITORIANO, 2019).

2.1.1.1 Linguagem de Marcação de Hipertexto, HTML

A Linguagem de Marcação de Hipertexto, do inglês: *HyperText Markup Language* (HTML) foi criada por Tim Berners-Lee enquanto trabalhava na Organização Europeia para a Pesquisa Nuclear (CERN), o laboratório de física de partículas na Suíça, no final dos anos 1980 e início dos anos 1990. O objetivo era criar uma maneira de compartilhar documentos e informações em um ambiente de rede. A primeira versão do HTML tinha apenas 18 elementos de marcação, permitindo a formatação básica de texto e a inclusão de *links*[6], imagens e listas. (W3C, 2023).

O HTML rapidamente ganhou popularidade e passou por várias iterações, cada uma adicionando novos elementos e funcionalidades. O HTML4, lançado em 1997, trouxe uma série de melhorias, incluindo mais controle sobre a aparência das páginas web, a introdução de folhas de estilo em cascata (HTML) e melhor suporte a *scripts*[7]. (W3C, 2023).

Finalmente, o HTML5, lançado oficialmente em 2014 pelo *World Wide Web Consortium* (W3C), trouxe uma série de novas funcionalidades, incluindo suporte nativo para vídeo e áudio; novos elementos semânticos; gráficos e animações; geolocalização; armazenamento local e muito mais. (W3C, 2023).

2.1.1.2 Funcionamento do HTML

O HTML funciona como uma linguagem de marcação, o que significa que ele usa *"tags"*[8] para definir diferentes partes de um documento. Essas tags informam ao navegador como exibir o conteúdo da página. Por exemplo, a *tag* <p> é usada para definir um parágrafo, enquanto a *tag* <h1> é usada para definir um cabeçalho de primeiro nível. (W3C, 2023).

As páginas HTML são estruturadas usando uma combinação de elementos de bloco, (que formam a estrutura principal da página), e elementos *inline*[9], (que formatam o conteúdo dentro desses blocos). Os elementos são aninhados dentro de outros elementos para criar a estrutura hierárquica da página. (W3C, 2023).

[6] Do inglês: Ligação. Também chamado de hiperlink, é uma referência a um documento eletrônico que, quando clicado, leva o usuário para outro recurso ou documento.

[7] Do inglês: Roteiro. Aqui usado no sentido de código fonte, que nada mais são do que um conjunto de instruções que o computador seguirá de modo interpretativo.

[8] Do inglês: Marcação.

[9] Do inglês: Dentro da Linha. São elementos que podem ser escritos sem quebra de linha.

2.1.1.3 HTML versão 5

Com o HTML5, os desenvolvedores podem criar jogos online, reproduzir vídeos e áudios diretamente no navegador, tudo isso sem a necessidade de instalação de plugins externos. Isso resultou em uma melhor experiência geral do usuário, com carregamento mais rápido e maior compatibilidade entre os navegadores. (W3C, 2023).

Além disso, o HTML5 também trouxe recursos avançados de armazenamento local, como o *Web Storage*[10] e o *IndexedDB*[11]. Esses recursos permitem que os sites armazenem dados localmente no navegador do usuário, possibilitando a criação de aplicativos web *offline* e sincronização de dados em tempo real. (W3C, 2023).

Outra contribuição importante do HTML5 é o suporte a tecnologias de geolocalização e acesso aos recursos do dispositivo. Isso permite que os desenvolvedores acessem informações de localização do usuário, câmera, microfone e acelerômetro, abrindo possibilidades para o desenvolvimento de aplicativos web que utilizam esses recursos de forma integrada. (W3C, 2023).

Ao longo de sua história, o HTML tem evoluído constantemente para acompanhar as demandas e os avanços tecnológicos da web. O HTML5 é um marco significativo nessa evolução, trazendo recursos semânticos, multimídia e interativos para a criação de páginas da web modernas. (W3C, 2023).

2.1.1.4 Folhas de Estilo em Cascata, (CSS)

Folhas de Estilo em Cascata, ou *Cascading Style Sheets* (CSS), em tradução livre para o português, é uma linguagem de estilo altamente eficaz e amplamente utilizada. Sua principal função é definir a apresentação de documentos escritos em HTML ou XML. Isso inclui uma série de linguagens baseadas em XML, como SVG, MathML e XHTML. O CSS é responsável por descrever a forma como os elementos são apresentados em diferentes mídias, seja na tela do computador, em papel impresso, por meio de dispositivos de fala ou em outras formas de mídia. (MDN, 2023a).

Considerado uma das principais linguagens da Open[12] Web, o CSS tem uma grande importância na padronização dos navegadores Web. Essa padronização é feita de acordo com as especificações estabelecidas pela W3C, a organização que lidera a Web mundial. O desenvolvimento do CSS é feito em níveis distintos: o CSS1, que hoje é considerado obsoleto; o CSS2.1, que atualmente é uma recomendação; e o CSS3, que está sendo dividido em pequenos módulos e caminha para a sua padronização. (MDN, 2023a).

[10] O termo Web *Storage* pode ser entendido, em tradução livre, como: Armazém da Web
[11] Termo abreviado de "Indexed DataBase", (Base de Dados Indexada).
[12] Do inglês: Aberto. Neste contexto, refere-se ao padrão "Aberto" da Web

2.1.1.5 JavaScript

JavaScript é uma linguagem de programação notavelmente versátil que, apesar de ser comumente conhecida pela sua utilização em páginas Web, vai muito além disso. Frequentemente abreviada para JS, essa linguagem é leve, interpretada e orientada a objetos com funções de primeira classe. Graças à sua flexibilidade, o *JavaScript* se expandiu para uma variedade de ambientes que não são navegadores, incluindo Node.Js[13], Apache CouchDB[14] e Adobe Acrobat[15], demonstrando sua adaptabilidade e eficácia em diversos contextos. (MDN, 2023c).

Com sua estrutura baseada em protótipos, o JavaSript é uma linguagem dinâmica que suporta múltiplos paradigmas de programação. Isso significa que, além de ser orientada a objetos, ela também suporta estilos de programação imperativos e declarativos, como a programação funcional. Essa capacidade de suportar diferentes estilos de programação torna o *JavaScript* uma ferramenta poderosa e flexível para os desenvolvedores. (MDN, 2023c).

O padrão para *JavaScript* é o ECMAScript. Desde 2012, todos os navegadores modernos oferecem suporte completo ao ECMAScript 5.1. Mesmo os navegadores mais antigos fornecem suporte, pelo menos, ao ECMAScript 3. A sexta versão do ECMAScript, oficialmente chamada de ECMAScript 2015 e inicialmente conhecida como ECMAScript 6 ou ES6, foi publicada pela ECMA International em 17 de junho de 2015. Desde então, as especificações do ECMAScript são lançadas anualmente, demonstrando o desenvolvimento contínuo e o avanço dessa linguagem padrão. (MDN, 2023c).

2.1.1.6 TypeScript

O *TypeScript*, as vezes abreviado como TS, é uma linguagem fortemente tipada construída em cima do *JavaScript*, (TYPESCRIPT,). *Typescript* traz uma sintaxe adicional para o *JavaScript* de modo que o mesmo possa suportar checagem de tipos estática. Sem o TS, fica difícil saber com quais tipos de dados estar-se a trabalhar durante o processo de desenvolvimento, pois o *JavaScript* é uma linguagem fracamente tipada. Os parâmetros das funções e variáveis não possuem nenhuma informação, forçando os desenvolvedores a recorrerem a todo momento à documentação ou intuir sobre as tipagens. *Typescript* resolve esse problema, permitindo tipar o código de modo que erros possam ser reportados quando a tipagem estiver incorreta, por exemplo: ao tentar-se passar uma string[16] para uma função que espera um número,

[13] Ver sessão que trata do *Node.Js*

[14] Base de dados que utiliza o JSON nativamente. Veja mais em: https://couchdb.apache.org/#about

[15] *Software* que lê e converte arquivos em formato PDF. Veja mais em: https://www.adobe.com/br/acrobat.html

[16] Do inglês: Corda, barbante ou fio. No contexto de programação, é usado como termo para cadeira de caracteres. O caractere é, na maioria das linguagens de programação, um tipo de dado. E textos

TypeScript lançará um erro. O *JavaScript*, por outro lado, permitirá a execução deste código podendo gerar erros de tempo de execução. (W3C, 2024).

O *TypeScript* possui um compilador, que nada mais é do que um transpilador. Este transpilador é responsável por transformar o código TS em JS. Desta forma, o código *JavaScript* resultante da transpilação pode ser rodado em praticamente qualquer navegador ou ambiente que suporte o *JavaScript*.

2.1.1.7 JavaScript Object Notation, JSON

JavaScript Object Notation, (Notação de Objeto *JavaScript*), popularmente chamado de JSON. É uma sintaxe para a serialização de objetos do *javascript*. Com objetos do *javascript*, compreende-se seus tipos de dados e valores, como: objetos; matrizes; números; *strings*; booleanos; *null*[17] e *undefined*[18]. Apesar de baseado na sintaxe do *JavaScript*, distingue-se desta no sentido da forma de escrita. A serialização é o processo de converter dados estruturados, (ou objetos), em um formato que pode facilmente ser armazenado e transmitido pela rede. O JSON basicamente converte os objetos *JavaScript* em *strings*. Uma caracteristica deste, é que JSON é legível tanto por humanos, quanto por máquinas, (na maioria dos casos). (MDN, 2024a).

A Tabela 3 mostra as principais diferenças entre o *JavaScript* e o JSON.

Tabela 3 – Diferenças entre o JavaScript e o JSON

Tipos e valores JavaScript	**Diferença para o JSON**
Objetos e Arrays	Os nomes das propriedades devem ser *strings* com aspas duplas; as vírgulas à direita são proibidas.
Números	Zeros à esquerda são proibidos; um ponto decimal deve ser seguido por pelo menos um dígito.
Strings	Apenas um conjunto limitado de caracteres pode ser escapado; certos caracteres de controle são proibidos; o separador de linha Unicode (U+2028) e o separador de parágrafo (U+2029) são permitidos; *strings* devem ter aspas duplas.

Fonte: (MDN, 2024a).

são formados por estas cadeias denominadas *strings*,

[17] Do inglês: Nulo. Neste contexto é um valor especial do *JavaScript* para representar a nulidade de um objeto/variável.

[18] Do inglês: Indefinido. Neste contexto é um tipo de dado do *JavaScript* para variáveis indefinidas.

2.1.2 Bibliotecas e Frameworks

2.1.2.1 ReactJs

Figura 5 – Logotipo do React

Fonte: React Dev, disponível em: https://react.dev/

"Biblioteca para interfaces de usuário Web e nativas". O React é uma biblioteca de *JavaScript* criada pelo Facebook para solucionar desafios de manutenção e escalabilidade em suas aplicações. No início de 2011, a equipe de desenvolvedores do Facebook enfrentava dificuldades em lidar com o crescimento da aplicação de anúncios, que estava se tornando cada vez mais complexa e difícil de ser mantida. O aumento no número de membros da equipe e de funcionalidades estava afetando negativamente os processos da empresa. Com tantas atualizações em cascata, a aplicação estava se tornando lenta e difícil de ser atualizada sem falhas. (MORAIS, 2021).

Para resolver esses problemas, Jordan Walke, engenheiro do Facebook, propôs uma solução inovadora. Ele sugeriu levar o XHP, uma versão do PHP, para o navegador usando *JavaScript*. O XHP era uma tecnologia desenvolvida para minimizar ataques de Cross-Site Scripting (XSS) em aplicações Web dinâmicas. No entanto, ele não era capaz de lidar com o grande número de requisições necessárias para esse tipo de aplicação. Com o apoio de sua equipe de gerenciamento, Jordan Walke conduziu um experimento de seis meses para explorar essa ideia. O resultado desse experimento foi o surgimento do ReactJS. (MORAIS, 2021).

O ReactJS revolucionou o desenvolvimento de interfaces de usuário ao introduzir o conceito de componentes reutilizáveis e a abordagem de renderização virtual. Com a utilização de componentes, os desenvolvedores podiam criar e reutilizar peças de interface independentes e isoladas, o que simplificava o desenvolvimento e manutenção do código. Além disso, a renderização virtual permitia atualizações de interface eficientes, otimizando o desempenho da aplicação. O ReactJS foi lançado como um

software de código aberto em 2013, permitindo que desenvolvedores de todo o mundo o utilizassem em seus projetos. (MORAIS, 2021).

Desde então, o React ganhou uma imensa popularidade e se tornou uma das principais ferramentas para o desenvolvimento de interfaces de usuário em aplicações web. Sua abordagem declarativa, que permite descrever como a interface deve ser exibida com base no estado da aplicação, simplifica a construção de interfaces complexas. Além disso, a capacidade de reutilização de componentes economiza tempo e esforço durante o desenvolvimento. O React também influenciou o desenvolvimento do React Native, uma versão da biblioteca voltada para a criação de aplicativos móveis multiplataforma. Com a ajuda de uma grande comunidade de desenvolvedores e empresas, o ecossistema do React continua a evoluir e fornecer soluções inovadoras para o desenvolvimento de interfaces de usuário modernas e eficientes. (MORAIS, 2021).

O React teve seus primeiros sinais em 2010, quando o Facebook introduziu o XHP na sua stack de PHP, permitindo a criação de componentes compostos. Em 2011, Jordan Walke criou o FaxJS, protótipo inicial do React, que foi desenvolvido para resolver os desafios de suporte aos anúncios do Facebook. Em 2012, o Instagram foi adquirido pelo Facebook e expressou interesse em adotar o React. Isso levou o Facebook a dissociar o React da empresa e torná-lo open source. Em 2013, ocorreu o lançamento oficial do React, mas inicialmente enfrentou resistência da comunidade de desenvolvedores. No entanto, uma "turnê do React" foi realizada para conquistar os não adeptos. (MORAIS, 2021).

No ano seguinte, o React começou a ganhar reputação e confiança. O *React Developer Tools*[19] e o *React Hot Reloader*[20] foram lançados, trazendo melhorias no desenvolvimento e na experiência do usuário. Em 2015, o React se estabeleceu como uma tecnologia estável, com empresas como Netflix e Airbnb adotando-o. O Redux, responsável pelo gerenciamento de estado, foi lançado, e o React Native expandiu-se para o desenvolvimento de aplicativos móveis para Android. (MORAIS, 2021).

Atualmente, o React continua evoluindo, com o lançamento de novas funcionalidades e recursos para melhorar o desenvolvimento de aplicações. Iniciativas de SSR *(Server Side Rendering)*[21], e o foco em componentes funcionais são algumas das áreas de desenvolvimento. O React permanece como uma biblioteca consolidada no mercado de *Front-End*, sendo amplamente adotado por grandes empresas em todo o mundo. (MORAIS, 2021).

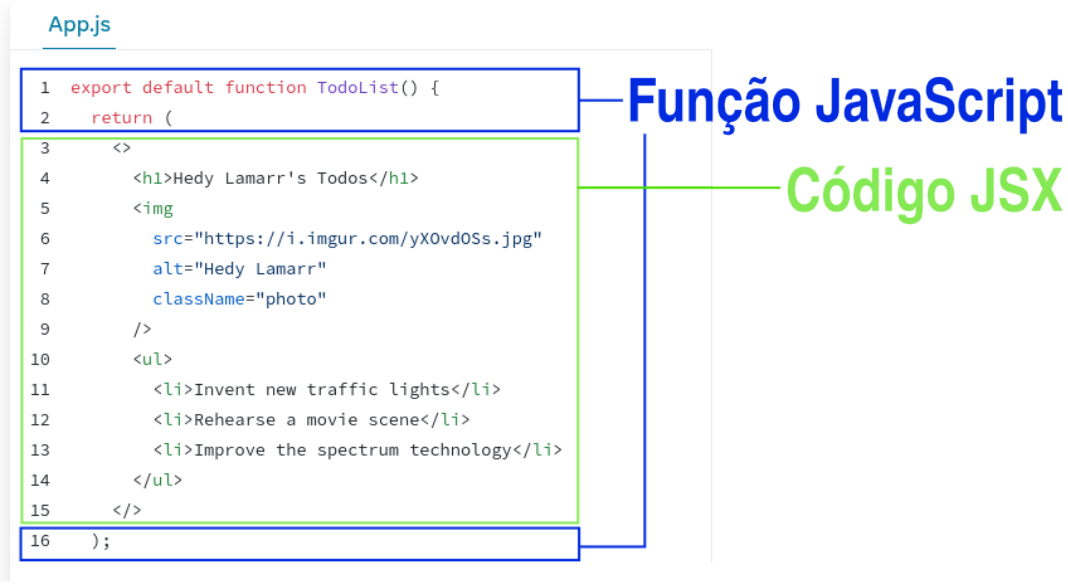
[19] Ferramentas de Desenvolvimento React
[20] Carregamento e recarregamento ultra rápido React
[21] Do inglês: Rederização do Lado do Servidor

2.1.2.1.1 JavaScript XML

O *JavaScript* XML, (JSX), é uma extensão de sintaxe para o *JavaScript* que permite escrever código de marcação, (como o HTML), dentro do *JavaScript*. Os componentes React nada mais são do que funções ou classes chamadas a partir do código JS que atualizam o HTML através da *Document Object Model*[22], (DOM). O JSX facilita esse trabalho. Pois ao invés de chamar funcões ou instanciar classes no código *JavaScript*, pode-se escrever a marcação diretamente no mesmo, como se o HTML estivesse dentro do JS. (REACTJS, 2024).

A Figura 6 mostra um exemplo de código em JSX. É exportada uma função denominada *TodoList*, que nada mais é do que um componente React. Esta função retorna um determinado valor, que é o código JSX propriamente dito. Este código será renderizado no lugar onde esta função for chamada.

Figura 6 – Exemplo de código JSX

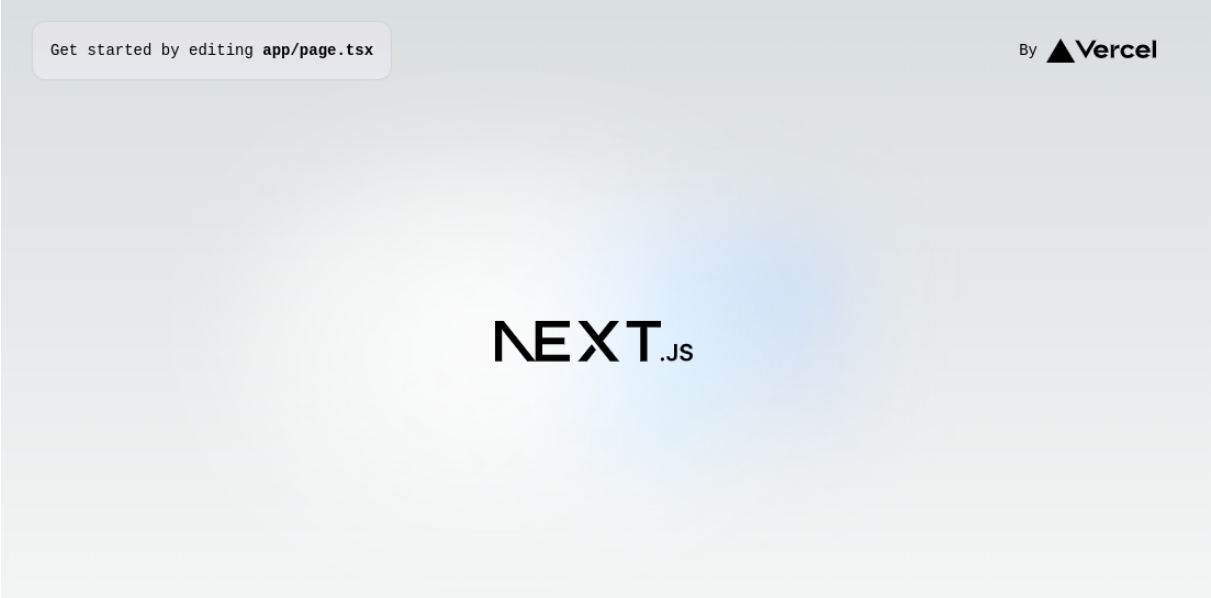


Fonte: Adaptado de: ReactJs. Disponível em: <https://pt-br.react.dev/learn/writing-markup-with-jsx>

[22] Do inglês: Modelo de Documento de Objeto. O DOM é utilizado pelo *JavaScript* para manipular o documento HTML exibido em tela. (PIMENTEL, 2022)

2.1.2.2 NextJs

Figura 7 – Logotipo do NextJs



Fonte: Next.Js, disponível em: https://nextjs-template.vercel.app/

O NextJs é uma *framework* em ReactJs voltada à construção de aplicações Web tando na parte do *Front-End* quanto no *Back-End*. Com NextJs, utiliza-se os componentes em React para construir as interfaces de usuário, com o NextJs provendo recursos adicionais e otimizações. (NEXTJS, 2024).

NextJs também se encarrega de todas as configurações necessárias do React, como o processo de enpacotamento[23], compilação e etc... Permitindo ao desenvolvedor apenas focar no desenvolvimento da aplicação em si. (NEXTJS, 2024).

Devido à natureza desta *framework*, O NextJs é um pilar que aparece tanto no *Back-End* quanto no *Front-End*. Estas duas frentes serão abordadas com a utilização desta ferramenta, aproveitando ao máximo os recursos fornecidos pela mesma. Os principais recursos oferecidos pelo NextJs são:

- Roteamento: O NextJs provê um roteamento, (que é basicamente a navegação por páginas dentro do app), baseado no sistema de arquivos do Sistema Operacional. Os arquivos em pastas do projeto são mapeados para links, que fornecem os componentes de servidor com suporte a layouts[24], rotas aninhadas, estados de carregamento, manipulação de erros, entre outros...

[23] Em inglês: Bundling. Um Bundle para a Web, por exemplo, junta todos os códigos e recursos em um pacote otimizado para ser distribuído.

[24] *Layouts* são como *templates* que são comuns às páginas roteadas. Ajudam no processo de reaproveitamento de componentes pois eles podem ser extendidos às páginas, que herdam características destes *layouts*.

- Renderização: NextJs fornece renderização do lado do cliente e do lado do servidor com componentes de cliente, e componentes de servidor.
- Busca de dados: Há um processo de busca de dados simplificado com o uso de *async/await*[25] nos componentes de servidor, além de uma API[26] expandida para a memorização das requisições, *caching*[27] de dados e revalidação.
- Estilização: Suporte para os métodos preferidos de estilização. Com inclusão de: Módulos CSS, *Tailwind* CSS, e CSS-in-JS.
- Otimizações: Otimizações de scripts, imagens e fontes são também fornecidos para aprimorar o núcleo do aplicativo e a experiência de usuário.
- *TypeScript*: Suporte total ao *TypeScript*, com uma melhor checagem de tipos e compilação eficiente.

(NEXTJS, 2024).

[25] Recurso do *JavaScript* para lidar com a execuçao de código assíncrono.

[26] Do inglês: Interface de Programação de Aplicações. É uma forma na qual dois ou mais aplicativos ou componentes de de computador se comunicam entre si. É uma interface de *software* que oferece um serviço para outras partes do mesmo ou de outros *softwares*. (REDDY, 2011).

[27] O processo de caching é o ato de armazenar informações que são acessadas frequentemente de maneira que seu acesso se torne mais rápido. Neste contexto, o resultado de uma requisição pode ser armazenado em cache para que não seja necessário consultar o servidor novamente quando a mesma informação for requisitada.

2.1.2.3 EditorJs

Figura 8 – Logotipo do EditorJs

Fonte: Editor.Js, disponível em: https://editorjs.io/

"Editor livre em blocos com saída universal em JSON". O EditorJs é um rico editor de texto em blocos que oferece uma experiência de edição intuitiva e versátil. Tudo o que é feito no EditorJs no fim é transformado em um arquivo JSON ao invés de um documento de marcação em HTML. Essa abordagem deixa o processo mais simples para os desenvolvedores no sentido de projetarem suas próprias integrações. Assim, o EditorJs pode ser aplicado a diversas plataformas. (EDITORJS,).

São recursos do EditorJs:

- Dados de saída limpos
- API baseada em plugins
- Código aberto

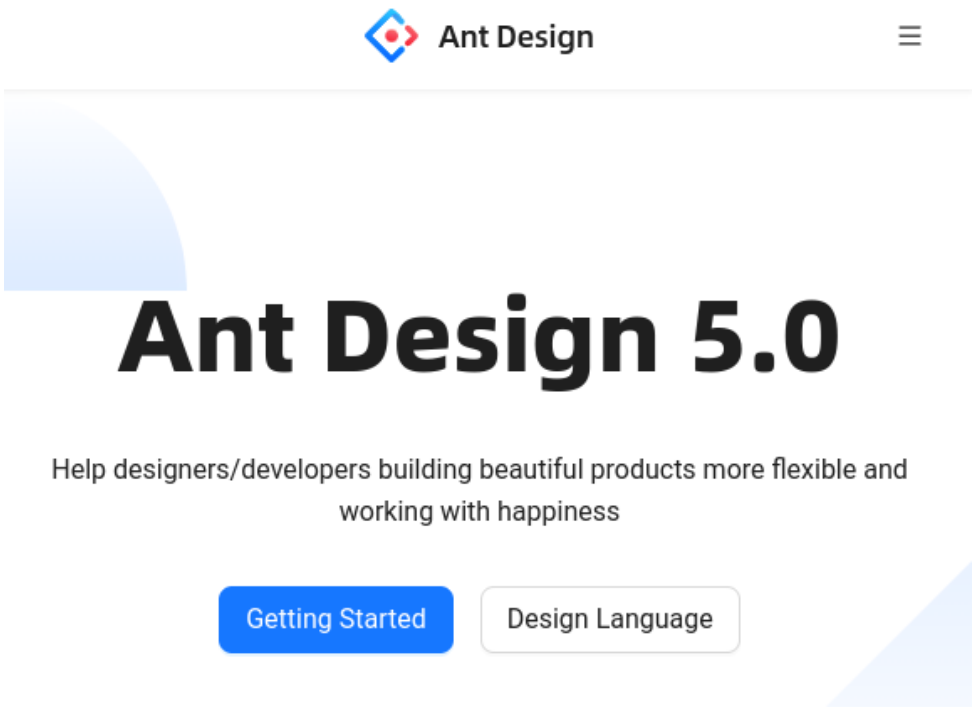
O espaço de trabalho do EditorJs consiste em blocos separados, como: Parágrafos; títulos; listas; etc... Cada um deles independentes entre si, com muitos outros recursos como: Copiar e colar; seleção de vários blocos; e entre outros que funcionam de forma familiar a outras ferramentas. (EDITORJS,).

O conceito chave do EditorJS é sua API, na qual todas as unides funcionais do editor são providas através de plugins externos que fazem uso da mesma. Assim, o núcleo do EditorJs fica sendo mais abstrato e poderoso, de modo que o desenvolvedor

possa implementar diversos desafios com a criação de seus próprios plugins. (EDITORJS,).

2.1.2.4 AntDesign

Figura 9 – AntDesign



Fonte: AntDesign, Disponível em: <https://ant.design>.

"Ajudando designers e desenvolvedores a construir belos produtos de forma flexível, trabalhando com alegria.". O AntDesign é uma biblioteca de UI, *User Interface*[28]. Esta biblioteca é uma grande auxiliadora na hora de produzir aplicativos com um design agradável sem que o desenvolvedor gaste muito tempo estilizando os componentes da aplicação.

Totalmente integrado ao ReactJs, o AntDesign já traz consigo uma vasta gama de componentes reutilizáveis que podem ser "encaixados" à construção das telas do app. O AntDesign não impacta somente a UI, mas também a *User Experience*[29], UX, uma vez que seus componentes são construídos em cima do React. Deixando-os altamente reativos e já com seus devidos comportamentos padrão. Ficado a cargo do desenvolvedor apenas personalizar o que se deseja fazer. A Figura 10 mostra um exemplo de implementação de um componente em AntDesign.

[28] Do inglês: Interface de Usuário
[29] Do inglês: Experiância de Usuário

Figura 10 – Exemplo de componente em AntDesign

Fonte: Adaptado de: AndDesign. Disponível em: <https://ant.design/components/switch>

Nota-se que com apenas uma linha tem-se um componente completo com um visual totalmente moderno. Algo que com apenas HTML, CSS e JS, gastaria algumas dezenas de linhas de código. Tanto ReactJs quanto AntDesign significam economia em termos de tempo, código e simplicidade de projeto.

2.2 Do Back-End

Back-end se relaciona com o que está nos bastidores das aplicações desenvolvidas na programação. Ou seja, tudo que dá estrutura e apoio às ações do usuário da máquina é chamado de *back-end*. Quando se acessa um site, por exemplo, por trás de toda sua apresentação amigável esteticamente, há uma comunicação das informações trocadas entre banco de dados e navegador. Portanto, atrás da interface gráfica do realizador, o *back-end* está sempre agindo. (TOTVS, 2020).

2.2.1 NodeJs

Figura 11 – Logotipo do NodeJs

Fonte: Adaptado de: NodeJs. Disponível em: <https://nodejs.org/en>

"Rode *JavaScript* em todo lugar". O *NodeJs* é um ambiente de *runtime*[30] que permite ao desenvolvedor criar servidores, aplicativos Web, ferramentas e linha de comando, entre outros... É a principal tecnologia deste projeto e o que permitirá a execução de todas as outras na sequência.

Antes do *NodeJs*, o *JavaScript* era uma linguagem que rodava puramente em *Browsers*[31] como uma forma de adicionar interações às páginas da internet. Com o *NodeJs*, o *JavaScript* passou do ambiente dos *Browsers* ao ambiente dos Sistemas Operacionais. Abstraindo APIs dos mesmos. Hoje, com NojeJs, por exemplo, pode-se acessar a API do sistema de arquivos do Sistema Operacional. Algo que há um tempo atrás só se fazia com linguagens de baixo nível como o C++.

A plataforma desenvolvida neste trabalho será um servidor que distribuirá o Sistema Web que é a plataforma em si. O NextJs, tecnologia escolhida para este fim, é um servidor capaz de processar uma parte das interfaces do sistema estaticamente, e entregá-las ao cliente juntamente com os scripts de suas partes dinâmicas. Desta forma, ganha-se segurança no processamento *back-end* ao mesmo tempo que não perde-se em termos de interatividade com o usuário. Há também diversos ganhos

[30] Do inglês: Tempo de execução. Um runtime é basicamente um interpretador capaz de executar um script.

[31] Do inglês: Navegadores. Aqui usado no sentido de navegador da Web, ou seja, o aplicativo no qual acessa-se páginas na internet.

de performance e simplificação de código, uma vez que *back-end* e *front-end* serão concentrados no mesmo lugar, não precisando separá-los em projetos diferentes.

2.3 O processo de Parsing

O processo de *Parsing* é uma das partes mais vitais deste projeto. Sem ele, não é possível obter o documento final formatado de acordo com as normas postas da ABNT e da PUC-GO. Pode-se dizer que o *Parsing* é o código núcleo da aplicação, pois todas as outras partes, como edição em blocos e navegação, por exemplo, serão feitas com o auxílio de bibliotecas e *frameworks*. O *Parsing*, por sua vez, será escrito puramete em *TypeScript* para processar as saídas do EditorJs.

Seu tratamento utilizar-se-á de uma combinação de expressões regulares para tratar os caracteres especiais de código LaTex, manipulação de código HTML produzido pelos plugins do editor utilizando-se o *Cheerio*. E por fim, a compilação de código LaTex utilizando-se o utilitário pdflatex para gerar o PDF.

2.3.1 Expressões regulares

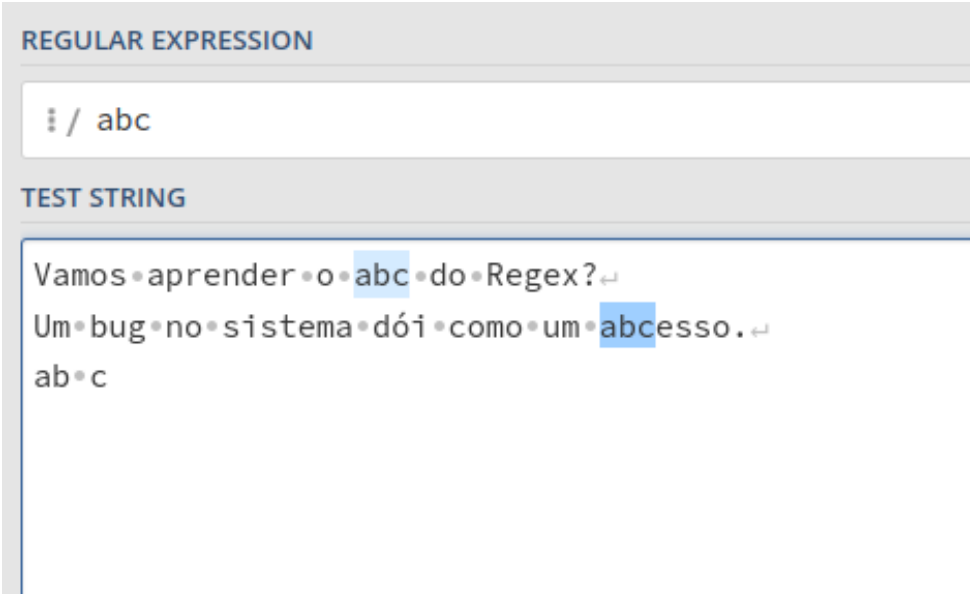
As expressões regulares, (as vezes denominadas RegEx ou RegExp), podem ser descrevidas basicamente como uma sequência de caracteres que descrevem um padrão de busca em um texto. (MAGALHãES, 2022).

Uma expressão regular pode ser composta por uma cadeia simples de caracteres, como **abc**; ou uma cadeia composta pela combinação de caracteres simples e caracteres especiais. Aos caracteres especiais, denominam-se metacaracteres. Os metacaracteres são usados quando a busca no texto requer algo mais do que uma simples correspondência direta. (MDN, 2024b).

2.3.1.1 Expressão regular simples

No exemplo anterior, a expressão simples **abc** irá corresponder a uma letra **"a"**, seguida de um **"b"**, seguida de um **"c"** no texto ao qual se está avaliando. Estas letras terão de estar juntas e nesta exata ordem. Esta expressão irá encontrar correspondências nas *strings* "Vamos aprender o abc do Regex?" e "Um bug no sistema dói como um abcesso.". Nestes dois casos, houve uma correspondência da substring **abc**, mas a *string* "ab c" não irá obter correspondência, pois aqui a exata substring **abc** não está contida. Observe as correspondências na Figura 12. (MDN, 2024b).

Figura 12 – Correspondêcias do Regex abc

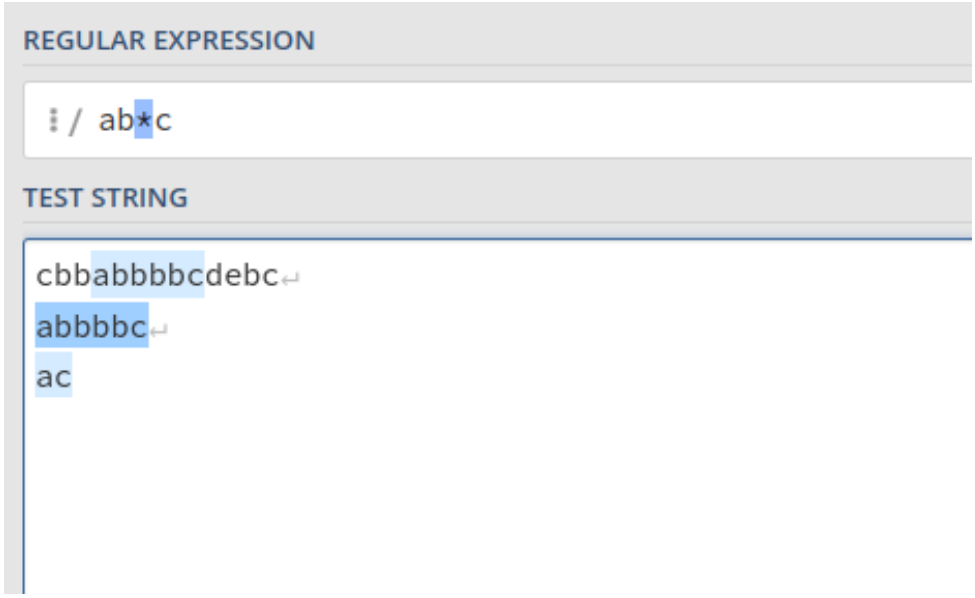


Fonte: Autoria própria

2.3.1.2 Expressão regular com metacaractere

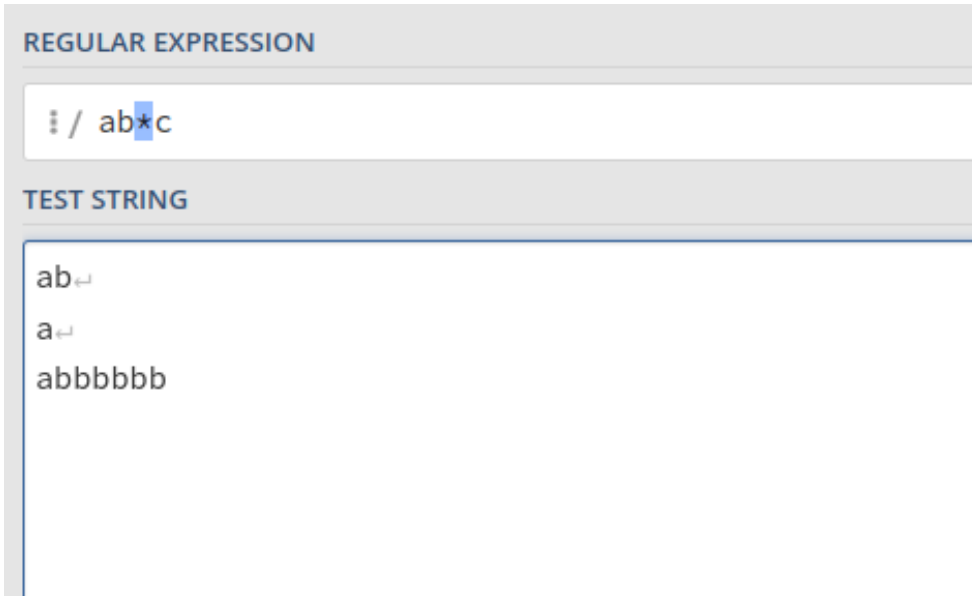
Como mencionado anteriormente, quando se quer algo que é mais do que uma simples correspondência direta, utilizam-se metacaracteres. O caractere * é um metacaractere de expressão regular. Por exemplo: **ab*c** utiliza o * como metacaractere. A expressão **ab*c** deve ser lida como: Corresponda ao caractere **a,** seguido por zero ou mais caracteres **b**s, seguido por um caractere **c.** Usar * após o **b** significa: zero ou mais ocorrências do item anterior, no caso **b.** Como observa-se na Figura 13 as *strings* "cbbabbbbcdebc", "abbbbc" e "ac" encontrarão correspondências. As *strings* "ab", "a" e "abbbbbb", por outro lado, não encontrarão correspondências, conforme a Figura 14. (MDN, 2024b).

Figura 13 – Correspondências do Regex ab*c, com metacaractere



Fonte: Autoria própria

Figura 14 – Não correspondências do Regex ab*c, com metacaractere



Fonte: Autoria própria

Uma expressão regular é uma combinação de caracteres, alguns deles, como o metacaractere quantificador *, visto anteriormente, são caracteres especiais. Estes metacaracteres são simbolos especiais que definem como a é interpretada. (MAGALHãES, 2022).

2.3.1.3 Expressões regulares em JavaScript

Em *JavaScript*, as expressões regulares podem ser escritas diretamente dentro do código, desde que postas entre barras //. Por exemplo: /abc/ é uma expressão

regular válida em JS. O tipo de dados das expressões regulares é o *object*[32], assim como *Array*[33] ou *Set*[34]. As expressões regulares no JS são usadas com dois objetos principais, a saber RegEx e *String*. A API de *strings* do JS fornece uma gama de métodos nos quais se pode utilizar juntamente com RegEx. A Tabela 4 fornece uma descrição destes métodos:

Tabela 4 – Métodos de string que podem ser usados com regex

Método	**Descrição**
match	O método match retorna o resultado da correspodência de um dado (MDN, 2024b) à *string* ao qual está sendo aplicado. Aplicação em código: "Meu abc".match(/abc/);
matchAll	Faz o mesmo que match. Porém, ao contrário de match, que traz apenas a primeira correspondência, matchAll traz todas as correspondências encontradas.
replace	O método replace de uma dada *string*, retorna uma nova *string* que substitui a correspondência de uma expressão regular por alguma *string* de substituição.
replaceAll	Um caminho diferente para fazer o mesmo que replace, porém para todas as ocorrências.
search	Faz uma busca pelo padrão na *string* e retorna o índice da primeira ocorrência.
split	Divide a *string* em uma lista ordenada de substrings em que o critério de divisão é a expressão regular fornecida.

Fonte: (MDN, 2024b).

O objeto RegEx também fornece dois métodos em que se usam as expressão regulares, a saber: exec() e test(). (MDN, 2024b).

2.3.2 Lamport Tex, LaTex

O LaTex é um sistema de preparação de documentos e processamento de texto, desenvolvido na década de 1980 pelo norte-americano Leslie Lamport baseado no sistema tipográfico TeX, desenvolvido por Donald Knuth. (FILHO; NOGUEIRA, 2009).

[32] Do inglês: Objeto

[33] Do inglês: Variedade ou matriz. No contexto de *JavaScript* é um dado estruturado, composto de uma sequencia de outros objetos ou dados puros.

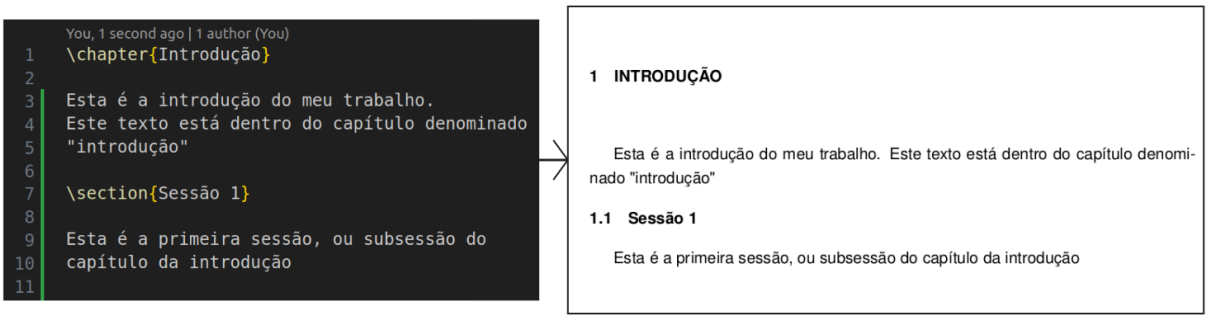
[34] Do inglês: Conjunto. Assim como Array, é um dado estruturado que agrupa outros objetos, porém de forma não ordenada e não indexada.

> O LaTex é utilizado para criar documentos dos mais variados tipos de publicação, como artigos, teses, dissertações, livros, cartas, relatórios ou qualquer outro tipo de documento. Possui um alto grau de exatidão e precisão na diagramação do conteúdo do documento e alta qualidade na formatação automática do documento. O LaTex é uma ampliação do original sistema de tipografia TEX. Tornou-se um padrão para produção de documentos científicos. (MADEIRA, 2020).

O código escrito em LaTex se utiliza de convenções de marcação (*tagging*) para controlar o modo como o texto é exibido, por exemplo: negrito, itálico, citação, referências cruzadas e etc... Sua utilidade está na produção de um texto entendível tanto por humanos, quanto pela máquina que o compilará e formatará. Deste modo, o autor pode ser distanciado da apresentação visual da informação, se preocupando apenas com o conteúdo escrito. (WIKIPEDIA, 2024).

Observe como no exemplo da Figura 15 o texto simples possui marcadores que demarcam sua exibição no arquivo final. O marcador \chapter demarca o capítulo, ao passo que \section uma subsessão, (ou sessão), do capítulo. Uma das grandes utilidades é que tanto a enumeração já é feita automaticamente, quanto sua presença no sumário já é automaticamente preenchida.

Figura 15 – Exemplo de compilação de código LaTex



Fonte: Autoria própria

2.3.2.1 AbnTex2

> O abnTeX2, evolução do abnTeX (ABsurd Norms for TeX), é uma suíte para LaTeX que atende os requisitos das normas da ABNT (Associação Brasileira de Normas Técnicas) para elaboração de documentos técnicos e científicos brasileiros, como artigos científicos, relatórios técnicos, trabalhos acadêmicos como teses, dissertações, projetos de pesquisa e outros documentos do gênero. A suíte abnTeX2 é composta por uma classe, por pacotes de citação e de formatação de estilos bibliográficos, por exemplos, modelos de documentos e por uma ampla documentação. (ARAUJO, 2012)

2.3.3 Cheerio

Cheerio é uma biblioteca útil para processar linguagem de marcação. Diferente do *Browser*, que renderiza uma página HTML e implementa a estilização, *Cheerio* analisa a linguagem de marcação tornando-a em dado estruturado. Deste modo, tem-se a possibilidade de percorrer e manipular a estrutura de dados resultante do código HTML. Especificamente, o *Cheerio* não renderiza visualmente, aplica CSS, carrega recursos externos ou executa *JavaScript*, o que é comum em uma aplicação de página única (SPA, na sigla em inglês). Isso torna o *Cheerio* muito mais rápido do que outras soluções.

2.3.3.1 Recursos

- Sintaxe familiar: O *Cheerio* implementa um subconjunto do core do *jQuery*. O *Cheerio* elimina todas as inconsistências do DOM e as peculiaridades dos navegadores da biblioteca *jQuery*, revelando sua API verdadeiramente esplêndida.
- Velocidade impressionante: O *Cheerio* trabalha com um modelo de DOM muito simples e consistente. Como resultado, a análise, manipulação e renderização são incrivelmente eficientes.
- Incrivelmente flexível: O *Cheerio* utiliza o analisador parse5 e pode opcionalmente usar o tolerante htmlparser2 de @FB55. O Cheerio pode analisar praticamente qualquer documento HTML ou XML.

Fonte: (CHEERIO, 2022)

3 DESENVOLVIMENTO

A base da aplicação se dará por meio de um servidor onde serão realizadas todas as operações. Este distribuirá o conteúdo estático e dinâmico para a interação com o usuário, além também de fornecer rotas e funções para o processamento do conteúdo gerado pelo usuário. Por se tratar de uma aplicação em seu estágio inicial de concepção, este projeto não se preocupará com autenticação e autorização de usuários. Consistirá apenas de um servidor simples a ser rodado localmente a fim de se desenvolver suas funcionalidades principais.

3.1 O servidor NextJs

O servidor da aplicação será construído em cima do *NextJs*, a framewok *React* para a Web. Iniciar o projeto é uma tarefa simples. Basta apenas navegar para algum diretório onde se deseja criar o projeto e digitar o seguinte comando em bash[1]:

```
1 npx create-next-app@latest
```

Vale ressaltar que é necessário ter o *NodeJs* na versão 18.17 ou superior para iniciar o projeto em *NextJs*. Após rodar este comando, o prompt fará uma série de perguntas para a configuração do mesmo, observe o exemplo abaixo:

```
1 What is your project named? my-app
2 Would you like to use TypeScript? No / Yes
3 Would you like to use ESLint? No / Yes
4 Would you like to use Tailwind CSS? No / Yes
5 Would you like to use ‘src/‘ directory? No / Yes
6 Would you like to use App Router? (recommended) No / Yes
7 Would you like to customize the default import alias (@/*)? No / Yes
8 What import alias would you like configured? @/*
```

Os itens a serem configurados são:

1. Nome do projeto

[1] Bash (*Bourne Again Shell*): Interface de linha de comando do linux

2. Será feito o uso de TypeScript?
3. Será feito o uso de ESlint?
4. Será feito o uso de Tailwind CSS?
5. Será usado o diretório ‘src‘ como diretório padrão de código?
6. Será utilizado o App Router (Roteador de App)?
7. Deseja customizar o apelido padrão de importação (@/*)?
8. Qual apelido de importação gostaria de configurar?

A tabela Tabela 5 mostra as opções escolhidas para este projeto:

Tabela 5 – Configurações do servidor NextJs

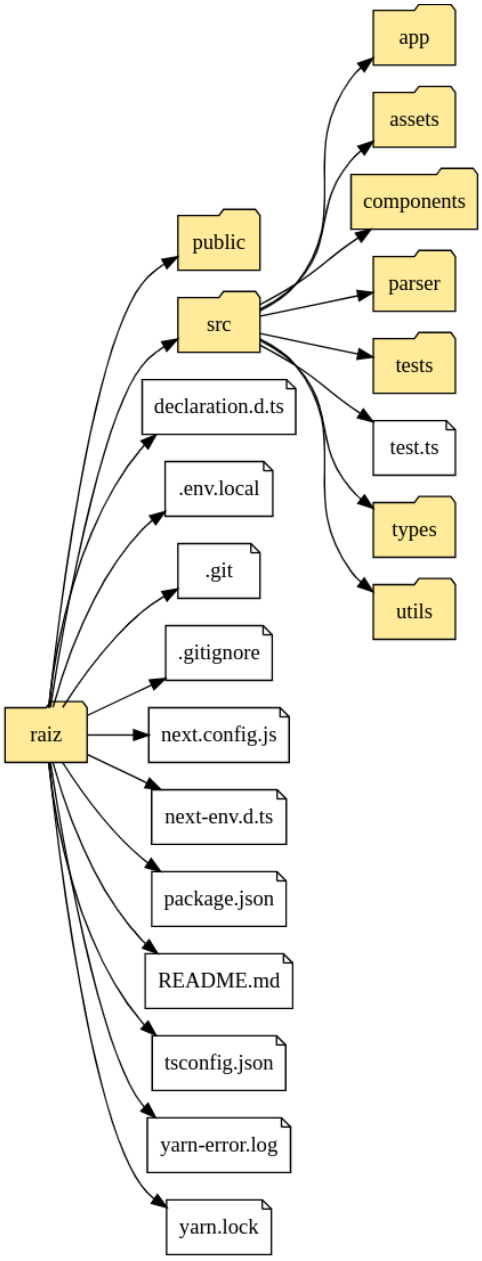
Pergunta	**Resposta**
What is your project named? my-app	editor2
Would you like to use TypeScript? No / Yes	Yes
Would you like to use ESLint? No / Yes	No
Would you like to use Tailwind CSS? No / Yes	No
Would you like to use ‘src/‘ directory? No / Yes	Yes
Would you like to use App Router? (recommended) No / Yes	Yes
Would you like to customize the default import alias (@/*)? No / Yes	Yes
What import alias would you like configured? @/*	@/*

Os comandos a seguir instalam todas as dependências necessárias do projeto:

```
yarn add @editorjs/editorjs @editorjs/header @editorjs/list
yarn add -D @types/editorjs__header @types/node @types/react
yarn add @emotion/react @emotion/styled antd cheerio
yarn add -D @types/react-dom @types/uuid typescript
yarn add node-latex react-icons uuid
```

Após isso o utilitário de criação da aplicação *NextJs* irá criar uma estrutura de pastas e arquivos de projeto totalmente configurado e pronto para ser programado. A Figura 16 mostra a estrutura de pastas base do projeto, com todos os seus respectivos arquivos de configuração, roteamento e componentes básicos do *react*.

Figura 16 – Estrutura de pastas e arquivos básica do projeto



Fonte: Autoria própria

Tabela 6 – Estrutura de pastas básica do projeto e suas atribuições

Arquivo/Pasta	**Descrição**
public	Pasta pública para distribuição de arquivos estáticos.
src	Esta pasta é praticamente o código fonte da aplicação onde todas as operações acontecem, tais quais: Edição com seus respectivos plugins; parser dos arquivos de saída e tudo mais.
declaration.d.ts	Arquivo de declarações. Aqui é anotado algumas tipagens para serem usadas globalmente durante o processo de desenvolvimento.
.env.local	Arquivo de variáveis de ambiente para serem usadas como teste durante o tempo de desenvolvimento.
.git	Pasta de controle do *git*.
.gitignore	Arquivos a serem ignorados pela ferramenta de versionamento.
next.config.js	Arquivo de configurações do *NextJs*.
next-env.d.ts	
package.json	Arquivo que define que o projeto é um projeto *NodeJs*. Aqui está toda a informação sobre dependências do projeto, que são todos os pacotes usados de terceiros. Também possui definição de scripts úteis para serem utilizados no processo de desenvolvimento.
README.md	Documentação de apresentação do projeto.
tsconfig.json	Configurações do *TypeScript*.
yarn-error.log	Erros do gerenciador de pacotes *yarn*
yarn.lock	Arquivo de lock de dependências do gerenciador de pacotes *yarn*.

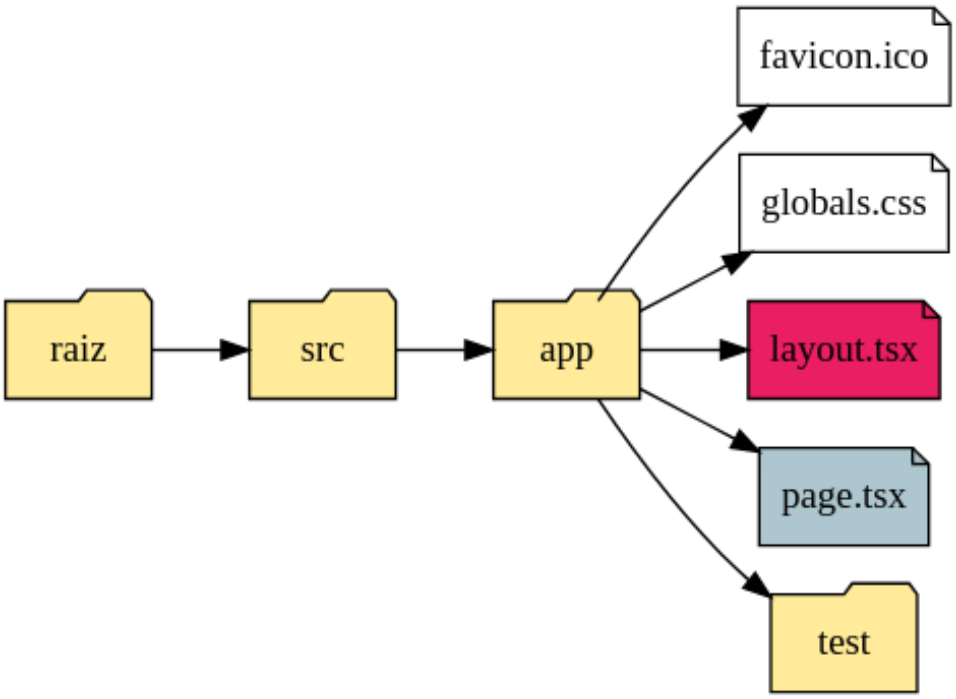
Fonte: Autoria própria

3.1.1 Roteamento (App Router)

No ato de configuração do servidor, conforme mostrado na Tabela 5, na pergunta: *"Would you like to use App Router? (recommended) No / Yes"*, foi escolhida a opção sim, *"yes"*. Isto significa que no código fonte do projeto haverá uma pasta chamada App onde serão arquivados os roteamentos, as páginas, *layouts*, páginas de erro, *loading*, entre outros.

A Figura 17 mostra a estrutura da pasta App.

Figura 17 – Estrutura da pasta de roteamento (App Router)



Fonte: Autoria própria

Observe os arquivos destacados em azul e rosa, respectivamente page.tsx e layout.tsx. A extensão TSX significa o mesmo que JSX, com a difereça de que é um arquivo em *TypeScript* ao invés de *JavaScript*.

Os arquivos *page* e layout são a primeira página da aplicação. Por estarem no nível do diretório App (dentro da pasta app) esta página será mapeada para a rota / na navegação. Isto significa que quando o servidor estiver rodando e o usuário acessar o endereço do serviço, o código contido em *page*.tsx renderizará o conteúdo da página para o usuário.

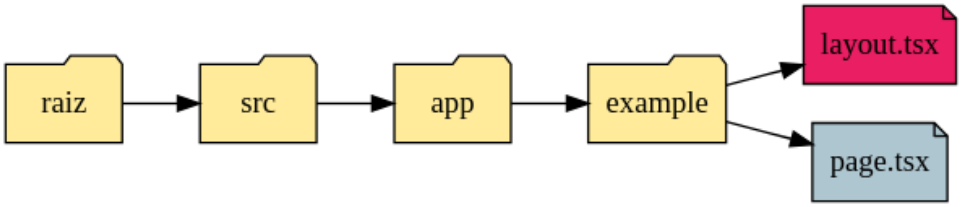
O arquivo layout.tsx serve como um *template* para a página. Pode-se pensá-lo como uma espécie de casca da aplicação. A página herda o que está em *layout*, de modo que o que será renderizado no *Browser* será o *template*, acrescido da *page*.

Por estar no diretório raiz de app, todas as rotas de páginas da aplicação herdarão o que está em *layout*.tsx. Arquivos de *layout* também podem ser escritos em rotas que não sejam a raiz.

3.1.1.1 Exemplo de criação de uma rota

Para criar uma rota para uma nova página no *NextJs*, basta que uma pasta no diretótio app seja criada. Esta pasta deve conter pelo menos um arquivo page.tsx ou route.ts. Observe a Figura 18.

Figura 18 – Estrutura da pasta de roteamento (App Router)



Fonte: Autoria própria

Ao criar a pasta example com o arquivo page.tsx. O servidor mapeará para a rota /example no *Browser*, renderizando o componente exportado em *page*. Observe a seguir o código em layout.tsx:

```
1  import { PropsWithChildren } from "react";
2
3  export default function Layout(
4      { children }: PropsWithChildren
5  ){
6      return <section>
7          <h1>Eu sou o Layout</h1>
8          { children }
9      </section>
10 }
```

Observe que o arquivo exporta uma função denominada *Layout*, que retorna um código TSX a ser renderizado na página. Na linha 4 a função recebe um objeto como parâmetro que é anotado por PropsWithChildren. Dentro deste objeto há a propriedade *children*[2], que contém outros componentes *React* a serem renderizados. No caso, o *NextJs* injetará dentro de *children* o código que é exportado por page.tsx. A linha 8 é a posição onde este componente será renderizado.

Observe abaixo o código de page.tsx:

```
1  export default function Page(){
2      return <div>
3          Olá mundo!
4          Eu sou uma page.
5      </div>
6  }
```

No código *page* apenas é exportada uma função (componente) *Page* que retorna uma frase simples: "Olá mundo! Eu sou uma *page*".

[2] Do inglês: Filhos.

3.1.1.2 Executando o exemplo

Para executar o exemplo e ver como ele se comporta no navegador do usuário, basta a partir do diretório do projeto rodar o seguinte comando:

```
yarn dev
```

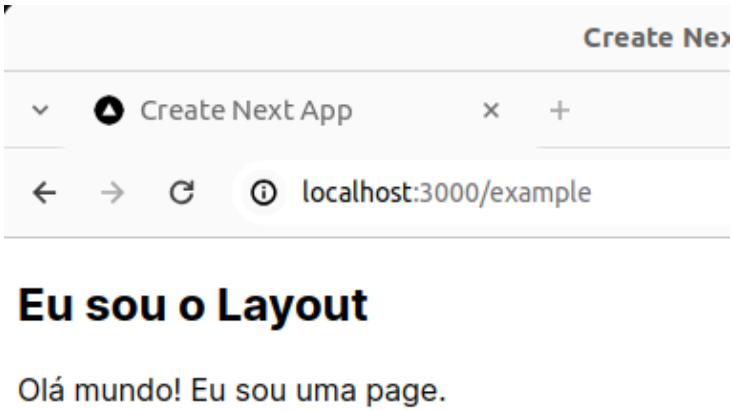
Após rodar o comando o servidor iniciará localmente. A saída do terminal mostrará a porta do serviço a ser acessada afim de renderizar a página:

```
yarn run v1.22.19
$ next dev
    Next.js 14.1.1
    - Local:        http://localhost:3000
    - Environments: .env.local

    Ready in 2.7s
```

Ao acessar o endereço http://localhost:3000 na rota /example, a página é renderizada conforme a Figura 19:

Figura 19 – Exemplo de roteamento



Fonte: Autoria própria

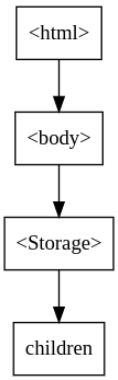
3.1.2 Página principal

Neste primeiro momento, o editor em que o usuário interagirá será renderizado na página principal. Conforme dito anteriormente e ilustrado na Figura 17, a página principal está contida ao nível da pasta app, juntamente com seu *layout*.

3.1.2.1 Layout

O arquivo de *layout* é o *template* base no qual todas as outras páginas herdarão. Observe na Figura 20 a sub árvore de renderização do *layout*:

Figura 20 – Sub árvore de renderização do layout principal



Fonte: Autoria própria

Tem-se uma estrutura básica com a *tag* html, (a *tag* raiz do documento), logo após o *body*, que diz respeito à área de renderização do documento. *Storage* é um componente personalizado em *React* que será discutido mais adiante, ele serve basicamente para armazenar conteúdos no navegador do usuário. Logo em seguida há o último componente, (ou nó folha), que consiste no *children*. Neste contexto, *children* pode ser qualquer coisa a depender da rota de página ao qual se está acessando.

Observe abaixo o código de *layout*:

```
1  import type { Metadata } from 'next'
2  import { Inter } from 'next/font/google'
3
4  import Storage from '@/components/Storage';
5
6  const inter = Inter({ subsets: ['latin'] })
7
8  export const metadata: Metadata = {
9      title: 'Create Next App',
10     description: 'Generated by create next app',
11 }
12
13 export default function RootLayout({
14     children,
15 }: {
16     children: React.ReactNode
17 }) {
```

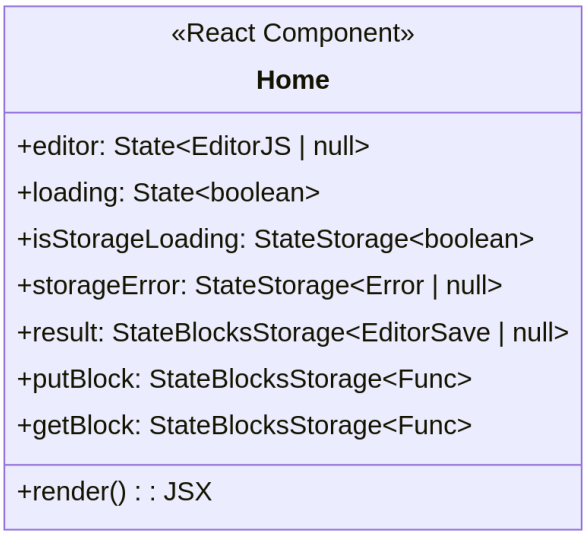
```
18      return (
19          <html lang="pt-BR">
20              <body className={inter.className}>
21                  <Storage>
22                      {children}
23                  </Storage>
24              </body>
25          </html>
26      )
27  }
```

Na linha 13 há a exportação do componente em si. O código TSX é retornado a partir da linha 18. Note que aqui não há o uso da *tag* head, padrão comum do HTML. Isso se dá pois o gerenciamento das configurações desta *tag* fica a cargo do *NextJs*. Nas linhas 8 à 11 há a exportação de uma constante denominada metadata. Nela há a chave title que o Next utilizará para renderizar o título da página, (que em um html normal seria configurado dentro de head).

3.1.2.2 Page

O arquivo page.tsx exporta o componente *React* denominado *Home*, que é a tela inicial propriamente dita. Embora os componentes em *React* sejam definidos como funções, os mesmos serão representados em forma de diagrama de classes para fins didáticos. Toda representação de componente terá um método render() que retorna um código JSX. Este método não existe no componente em si, mas sim na implementação do *React*. Pode-se pensar o render() como a função que renderiza o que é retornado pela "função componente" *Home*. A Figura 21 ilustra o componente em forma de diagrama de classe com seus estados e utilitários.

Figura 21 – Componente da página Home



Fonte: Autoria própria

O campo editor é o mais importante de todos, pois é nele que estará armazanada a referência à classe EditorJs em forma de estado. É importante salientar que todo estado vem acompanhado com uma função de atualização. Sempre que este estado é atualizado, a função de renderização é chamada novamente, remotando o componente e consequentemente todas as referências aos seus estados.

Note que o campo editor pode ser uma instância de EditorJs ou um valor nulo. Isso se dá pois inicialmente este estado começa nulo até que a classe seja carregada. Observe na Figura 22 como o componente *Home* será renderizado:

Figura 22 – Sub árvore de renderização da página principal

Fonte: Autoria própria

Percebe-se que o componente Container possuí dois filhos: Um texto ("Carregando editor...") e o componente Editor. Observe a partir da linha 87 no código abaixo como

a renderização do texto "Carregando editor..." está condicionada ao estado *loading*.

```
82 [...]
83 return (
84     <main>
85         <Container>
86             {
87                 loading
88                     ? 'Carregando editor...'
89                     : null
90             }
91             <Editor
92 [...]
```

O estado *loading* inicialmente começa com o valor *true*, o que faz com que o texto seja renderizado. O componente Editor também estará em tela, mas como faz uso da classe EditorJs, só aparecerá quando a mesma estiver pronta.

Note na linha 121 do código abaixo que a propriedade *onReady*[3], que pertence ao componente Editor, recebe uma função com o parâmetro { editor }.

```
120 [...]
121 onReady={ ({ editor }) => {
122     setLoading(false);
123     setEditor(editor);
124 } }
125 [...]
```

A propriedade onReady é um evento que chama a função passada quando a classe EditorJs está pronta e totalmente carregada. Note que a função passada chama dois atualizadores de estado: *setLoading* e setEditor. Estes atualizadores setam[4] os estados *loading* como *false*, e editor com o valor { editor } presente no parâmetro da função. O resultado disso é que a partir deste momento o texto ("Carregando editor...") não aparecerá mais em tela; o editor estará pronto para ser usado; e o estado editor possui a referência à classe com toda a API do EditorJs.

A Tabela 7 lista cada um dos estados do componente *Home*, bem como cada uma de suas atribuições:

[3] Do inglês: Quando pronto.

[4] Neologismo derivado do verbo inglês 'To Set' que significa 'Definir', (INFORMAL, 2018).

Tabela 7 – Propriedades do componente de tela Home

Propriedade	**Atualizador**	**Descrição**
editor	*setEditor*	Armazena a referência à classe do EditorJs
loading	*setLoading*	Estado de quando a tela está carregando
isStorageLoading	Automático	Quando a API da *Storage* está carregando
storageError	Automático	Caso haja algum erro com a *Storage*
result	Automático	Blocos guardados na *Storage*, (caso Hajam)
putBlock	Automático	Função para guardar blocos na *Storage*
getBlock	Automático	Função para pegar blocos da *Storage*

Fonte: Autoria própria

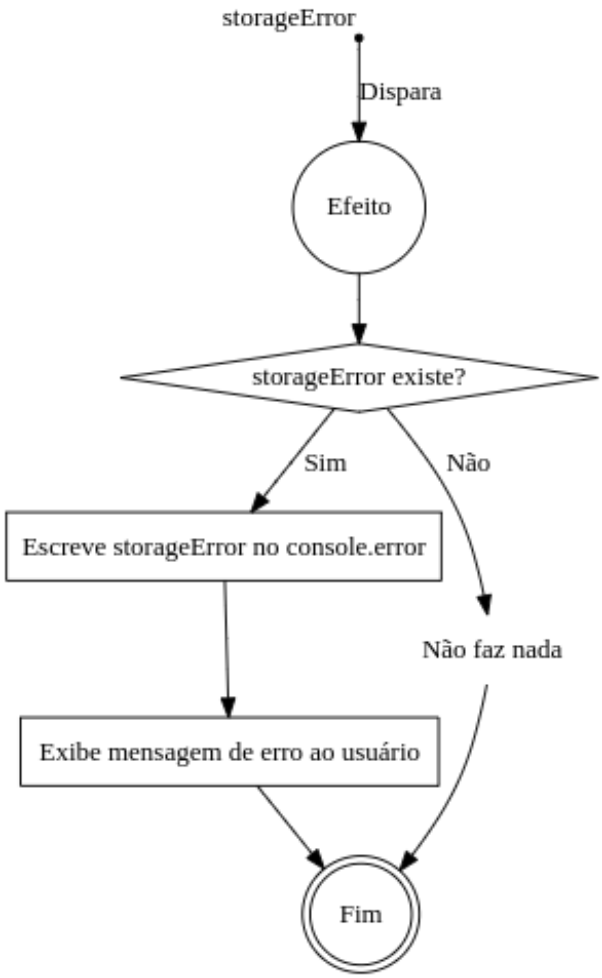
3.1.2.3 Disparo de efeitos

Os estados podem alterar o comportamento da tela, bem como seu fluxo de renderização de modo a enriquecer a experiência de usuário. A mudança de um estado também pode disparar um efeito. Efeito este que adiciona alguma ação ou modificação de outros estados. Observe o código abaixo:

```
75 [...]
76 useEffect(() => {
77     if(storageError){
78         console.error('Storage error', storageError);
79         message.error('Erro ao carregar storage');
80     }
81 }, [ storageError ]);
82 [...]
```

O código acima adiciona uma função que monitora o estado denominado *storageError*. Na primeira renderização da página, ou sempre que este estado muda, a função é disparada. Tudo que a função faz é checar se há algum erro presente em *storageError*. Caso afirmativo, o erro é transcrito na saída de erro do console através da linha 78, e uma mensagem é exibida ao usuário na linha 79. A Figura 23 ilustra este comportamento de forma visual:

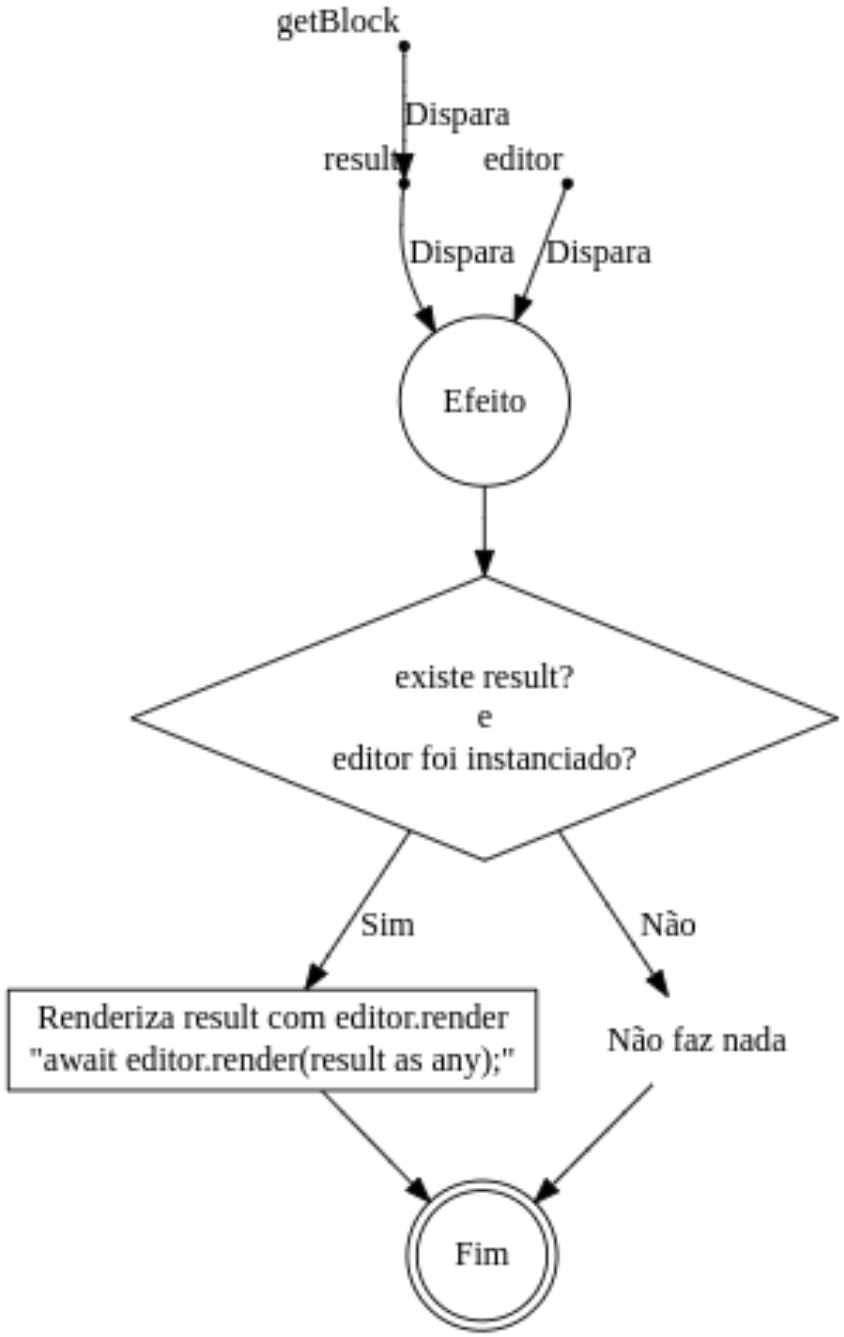
Figura 23 – Disparo de efeito de storageError



Fonte: Autoria própria

Diversos efeitos podem ser adicionados ao longo do componente, e cada efeito pode monitorar um ou mais estados. Observe na Figura 24 como os estados result e editor se utilizam de um efeito para checar se existem blocos salvos para então carregá-los através da API contida em editor:

Figura 24 – Disparo de efeito de result e editor



Fonte: Autoria própria

Note como getBlock, que é uma função provida pelos estados do BlockStorage, pode disparar uma mudança no estado result. Este é exatamente o fluxo desejado. A Figura 25 mostra como a aplicação fica aguardando a *Storage* carregar, e assim que ela carrega, getBlock é chamada, disparando o result, que porventura seguirá o fluxo na Figura 24.

Figura 25 – Disparo de efeito isStorageLoading

Fonte: Autoria própria

Por fim, a função putBlock é utilizada por um evento disparado pelo componente Editor. Observe como a propriedade *onChange*[5] recebe uma função que recupera o estado atual do editor, e em seguida repassa-o para putBlock.

Os detalhes de implementação da *Storage* se encontram no repositório através do diretório: /src/components/*Storage*

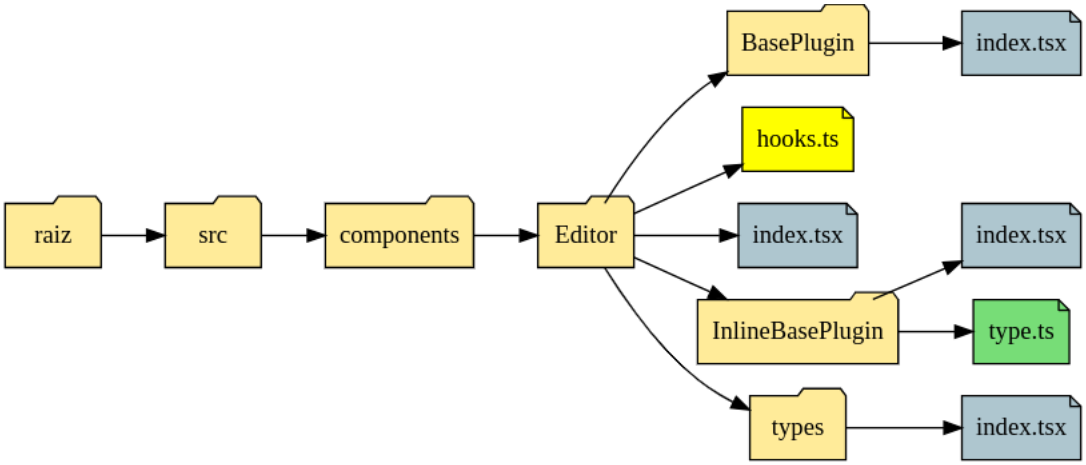
```
116 [...]
117 onChange={async (api, event) => {
118     const blocks = await api.saver.save();
119     putBlock(blocks as EditorSave);
120 }}
121 [...]
```

[5] Do inglês: Quando mudar

3.2 Editor

O Editor é o principal componente da aplicação em termos de interatividade com o usuário e UX. Através dele o usuário poderá inserir os blocos a comporem o seu trabalho acadêmico. Veja na Figura 26 a localização da estrutura de pastas no projeto.

Figura 26 – Estrutura de pastas do componente Editor

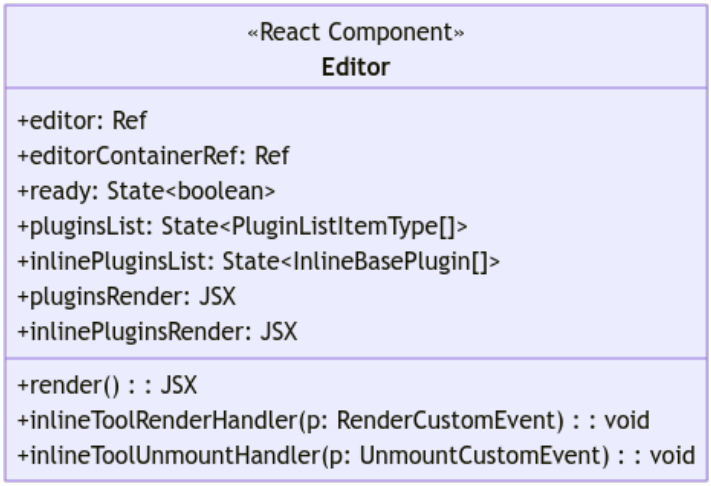


Fonte: Autoria própria

3.2.1 Provider

O componente do Editor propriamente dito, aqui denominado Provider, é exportado pelo arquivo Editor/index.tsx. É Chamado de Provider pois em seu contexto são colocadas funções que poderão ser utilizadas no decorrer de toda a aplicação, com a condição de que os componentes utilizadores sejam filhos de Editor. A Figura 27 contém a descrição do componente:

Figura 27 – Componente React - Editor

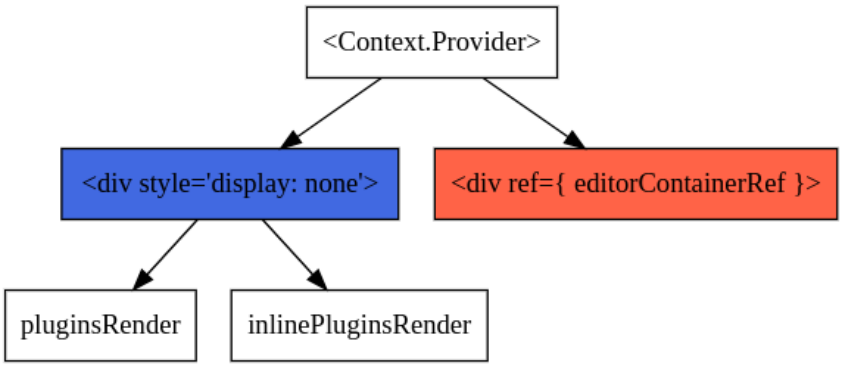


Fonte: Autoria própria

Os dois principais atributos deste componente são: editor e editorContainerRef, ambos anotados como Ref. Desta vez, estes atributos não são estados, mas sim referências. A referência, ao contrário do estado não possui função de atualização. São objetos especiais que podem guardar referências para objetos no DOM ou outros objetos *JavaScript*. Isso acontece pois editor guardará a instância do EditorJs, ao passo que editorContainerRef referenciará uma div no DOM. Esta div será utilizada por editor para inputar a ferramenta de edição de texto.

Observe na Figura 28 que o Context.Provider renderiza dois elmentos div irmãos. O azul, que conterá os plugins nomais e in-line. E o vermelho que é a div na qual se define o atributo ref com editorContainerRef.

Figura 28 – Sub árvore de renderização do componente Editor

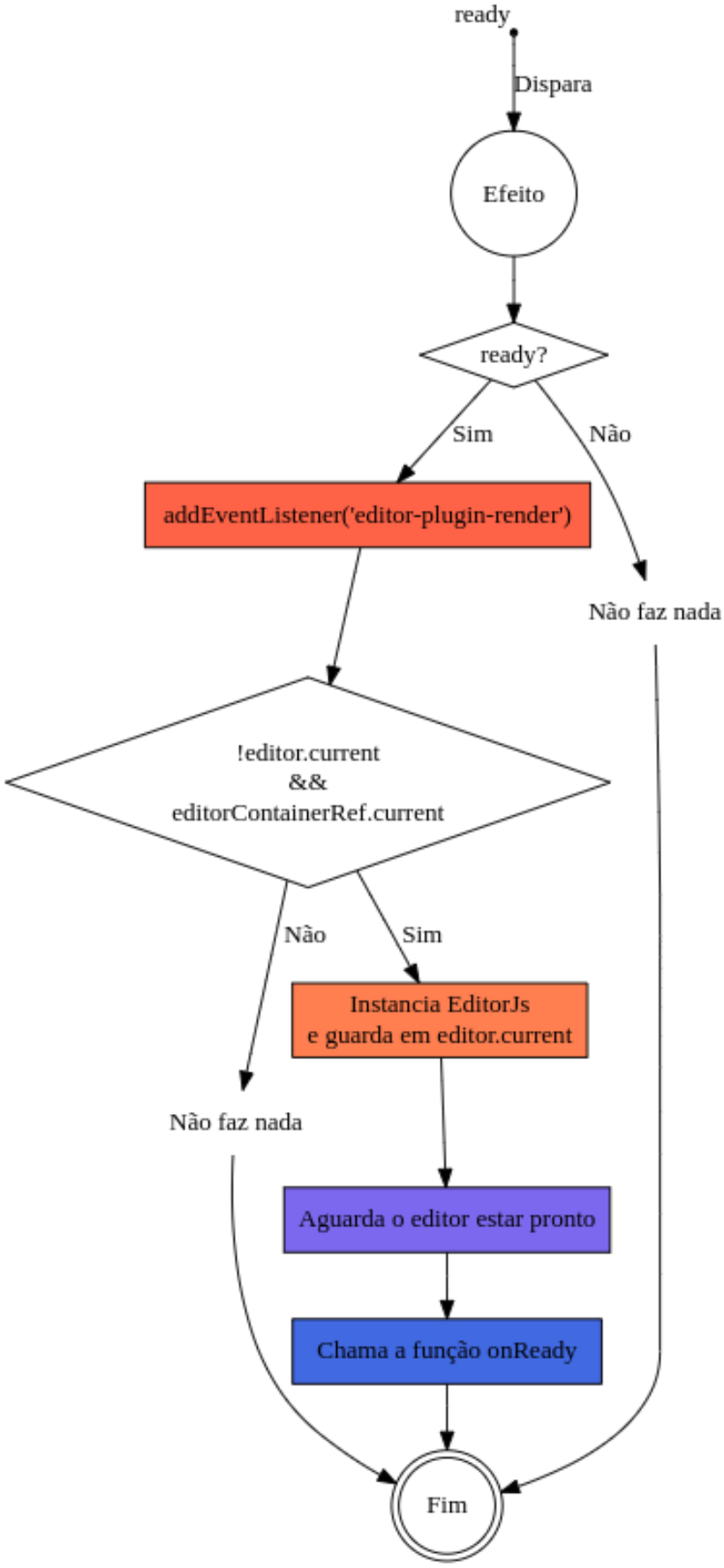


Fonte: Autoria própria

Observe que a div azul não é exibida em tela através da propriedade style='display: none'. Isso acontece porque os plugins listados nela serão transportados para dentro da div vermelha por meio de um conceito chamado *React Portals*. Virtualmente, para o React, os plugins estarão na div azul. Porém praticamente, eles estarão na verdade dentro da div vermelha num lugar a ser definido pelo EditorJs, que define isso automaticamente. Este recurso é usado para se injetar os componentes do React dentro do editor sem quebrar sua árvore lógica. Mais a frente será demonstrado que a classe EditorJs é quem é responsável por chamar e decidir a localização de cada componente através da instanciação de classes.

A Figura 29 ilustra o processo simplificado de quando o estado *ready* é disparado. *Ready* é disparado quando acontece a primeira renderização da página, por meio de um efeito que não monitora nenhum estado em particular. A tarefa deste efeito é apenas setar o estado *ready* para *true*. Isto funciona pois o efeito que não monitora nenhum estado é executado apenas uma vez, quando o componente é renderizado.

Figura 29 – Simplificação do Efeito ready no componente Editor



Fonte: Autoria própria

3.2.1.1 Event Listener

O quadro vermelho na Figura 29 corresponde a adição de um escutador de evento personalizado denominado *editor-plugin-render*. Isto é importante pois este evento recebe a criação dos plugins que são disparados de dentro da classe EditorJs. Observe o código abaixo:

```
136 [...]
137 document.addEventListener('editor-plugin-render', e => {
138     // console.log('editor-plugin-render');
139     // console.log({ e });
140     setPluginsList(prev => [
141         ...prev,
142         {
143             excluded: false,
144             // @ts-ignore
145             plugin: e.detail.context
146         }
147     ]);
148 });
149 [...]
```

Na linha 137 o escutador é adicionado ao objeto DOM. Este evento monitora qualquer renderização de plugin que é despachada, seja ela da onde for. Quando acontece um despache a função é chamada, de modo que na linha 140 tem-se a adição do plugin na lista de plugins. Note que o objeto adicionado possui uma propriedade excluded que controla quando o plugin sai de cena.

Quando o estado pluginsList é atualizado por setPluginsList, a lista de plugins é memorizada em pluginsRender, que é então renderizado na linha 213 do código abaixo:

```
208 [...]
209     return <Context.Provider value={{
210         editor: editor.current
211     }}>
212         <div style={{ display: 'none' }}>
213             { pluginsRender }
214             { inlinePluginsRender }
215         </div>
216         <div ref={ editorContainerRef } ></div>
```

```
217     </Context.Provider>
218 }
```

3.2.1.2 Instância do EditorJs

Os quadros laranja, violeta e azul da Figura 29 correspondem a instanciação do EditorJs e suas configurações. Quando a referência editor ainda não foi atribuida mas o editorContainerRef já está renderizado em tela conforme a linha 173, a linha 175 se encarrega de instanciar a classe EditorJS.

Observe a importancia da linha 177 em que a referência editorContainerRef é repassada para a propriedade holder. O EditorJS utiliza essa propriedade para colocar a ferramenta de edição na DOM através da referência ao elemento repassado.

A propriedade tools recebe o register na linha 178, que nada mais é do que a lista de plugins repassada quando o componente vai ser utilizado. na linha 190 há uma espécie de trava que aguarda até que o editor esteja pronto antes de chamar a função onReady na linha 192. Note que onReady só é chamada caso a mesma tenha sido repassada. Ao chamar, a instância do editor é repassada na linha 193.

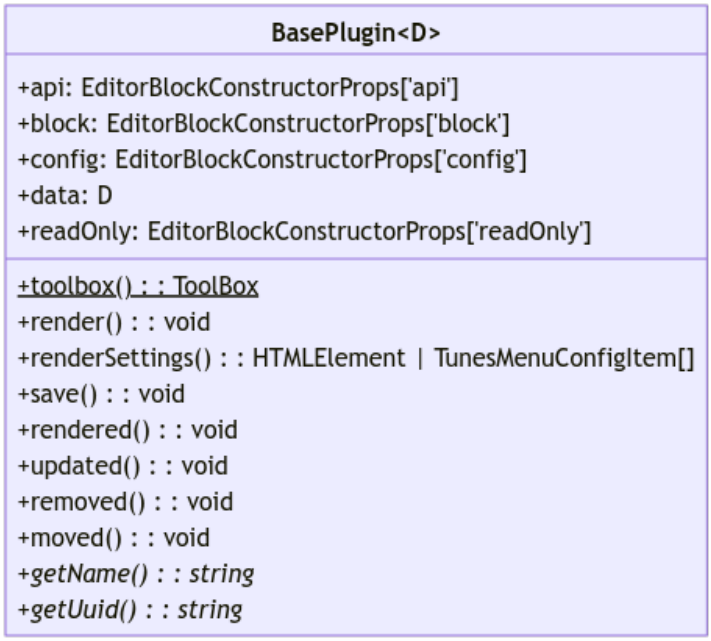
```
172 [...]
173 if(!editor.current && editorContainerRef.current){
174     // console.log("Comming to assing editorjs...");
175     editor.current = new EditorJS({
176         ...config,
177         holder: editorContainerRef.current,
178         tools: Object.keys(register).reduce((prev, key) => {
179             const { component, ...restOfProps } = register[key];
180             return {
181                 ...prev,
182                 [key]: restOfProps
183             };
184         }, {}),
185         onChange: (api, event) => {
186             onChange && onChange(api, event);
187         },
188     });
189 
190     await editor.current.isReady;
191 
192     onReady && onReady({
193         editor: editor.current
```

```
194     })
195 }
196 [...]
```

3.2.2 BasePlugin

A classe BasePlugin será a classe na qual todos os plugins irão derivar. Ela implementa os métodos chamados pelo EditorJS quando um plugin customizado é chamado pela API do EditorJS.

Figura 30 – Classe - BasePlugin



Fonte: Autoria própria

A tabela Tabela 8 mostra a descrição dos métodos da classe e suas atribuições.

Tabela 8 – Métodos da classe BasePlugin

Método	**Descrição**
toolbox()	Método estático que retorna o ícone do plugin a ser renderizado no menu
render()	Método que renderiza o plugin na ferramenta de edição
renderSettings()	Método que renderiza o menu personalizado do plugin
save()	Método que retorna o estado do componente quando saver da API é chamado
redered()	Método disparado quando o plugin é renderizado
updated()	Método disparado quando o plugin é atualizado
removed()	Método disparado quando o plugin é removido
moved()	Método disparado quando o plugin é movido
getName()	Método abstrado que retorna o nome do Plugin. Toda implementação de plugin deve implementar essa classe obrigatoriamente
getUuid()	Método abstrado que retorna um identificador único ao instanciar o plugin. Toda implementação de plugin deve implementar essa classe obrigatoriamente

O método mais importante da classe base é o render(). É ele quem coloca o plugin em tela quando o usuário escolhe um plugin a partir do menu de plugins. Observe no código abaixo o funcionamento da classe:

```
50 [...]
51 public render(){
52     const wrapper = document.createElement('div');
53     wrapper.id = this.pluginId;
54
55     const ev = new CustomEvent<{ context: BasePlugin }>(
56         'editor-plugin-render', {
57             detail: {
58                 context: this
59             }
60         }
61     );
62     setTimeout(() => document.dispatchEvent(ev), 20);
63
64     return wrapper;
```

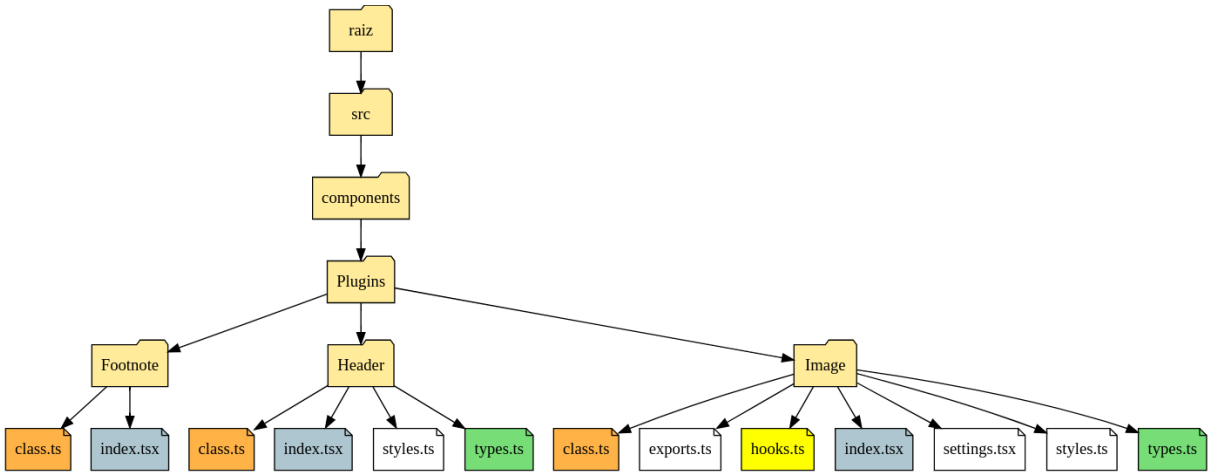
```
65 }
66 [...]
```

Observe que o método render cria um wrapper a partir de um elemento div na linha 52. Na linha 53 é atribuído um id a partir da chamada do método getUuid(). Na linha 55 a 61 é criado um evento customizado que é despachado 20 milissegundos após o retorno do wrapper. O wrapper é o que o EditorJS renderiza na ferramenta. O despache do evento juntamente com a referência à instância da classe é então capturada pelo escutador discutido na Figura 29 da sessão Provider. Assim, o plugin do react vai para a lista de plugins ao mesmo tempo que é renderizado de forma integrada ao EditorJS. (Falar do código do portal)

3.2.3 Plugins

Plugins são o principal objetivo em se usar o EditorJS, pois cada plugin é o que se traduz nos blocos da escrita em blocos conceituada no capítulo 1. A Figura 31 ilustra a estrutura de pastas dos plugins que foram desenvolvidos:

Figura 31 – Estrutura de pastas dos plugins



Fonte: Autoria própria

Observe que as condições mínimas para a existência de um plugin é possuir os arquivos index.tsx e class.ts. O arquivo class.ts é a classe propriamente dita que é derivada de BasePlugin, que consequentemente há de se implementar os dois métodos abstratos obrigatórios.

O arquivo index.tsx é o componente React interativo. É neste arquivo em que se programa a aparência e comportamentos do mesmo.

3.2.3.1 Header

O plugin de Header, (até o momento de escrita deste trabalho), pode ser considerado um dos mais simples que foi desenvolvido. Observe abaixo a implementação do arquivo class.tsx:

```
16 [...]
17 export default class HeaderClass extends BasePlugin<DataType> {
18     text: string = ""
19     public setLevel:
20         Dispatch<SetStateAction<HeaderLevelsType>> | null = null
21
22     static get conversionConfig() {
23         return {
24             export: 'text',
25             import: 'text',
26         };
27     }
28 [...]
```

Na linha 17, é realizada a exportação da classe HeaderClass, que estende a classe BasePlugin. Observe que nas linhas 18 e 19, são definidas uma propriedade text (que armazenará o texto do Header) e a função setLevel (que é uma função de atualização de estado do React).

No caso do plugin Header, é importante ter uma referência à função de atualização do nível (level) do Header, pois é nesta classe que o menu, onde o usuário selecionará o nível do título, é definido, conforme o código abaixo:

```
35 [...]
36 public renderSettings(): TunesMenuConfigItem[] {
37     return ([1,2,3,4,5] as HeaderLevelsType[]).map(lv => ({
38         title: `Nível ${lv}`,
39         // @ts-ignore
40         onActivate: () => {
41             console.log({ setLevel: this.setLevel });
42             this.setLevel && this.setLevel(lv);
43         },
44         closeOnActivate: true,
45         isActive: lv === this?.pluginData?.level,
46 [...]
```

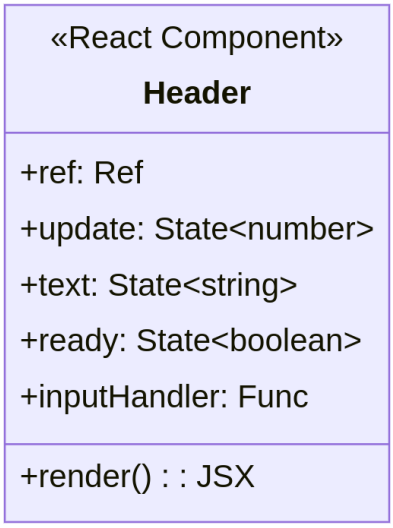
Observe como a função de atualização de estado é chamada a partir de onActivate na linha 40. Isso garante a comunicação do menu com o componente React, permitindo ao usuário selecionar o nível de título a partir do menu.

No código abaixo tem-se a implementação das duas funções abstratas obrigatórias getName() e getUuid(). Observe que em getName retorna-se o nome header do plugin, ao passo que em getUuid retorna-se um uuid versão 4 para criar um identificador único para o plugin.

```
51 [...]
52     getName(): string {
53         return 'header';
54     }
55
56     getUuid(): string {
57         return uuidv4();
58     }
59 }
```

3.2.3.1.1 Plugin React

Figura 32 – Plugin: Componente React

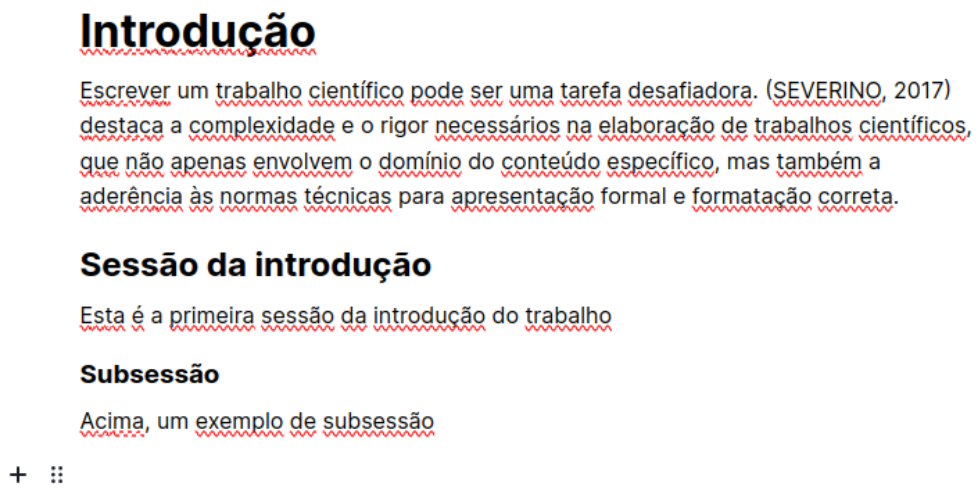


Fonte: Autoria própria

3.2.3.1.2 Aparência

A Figura 33 mostra como o plugin de Header se parece na interface do editor. Também é possível observar seus subtítulos juntamente blocos de parágrafo.

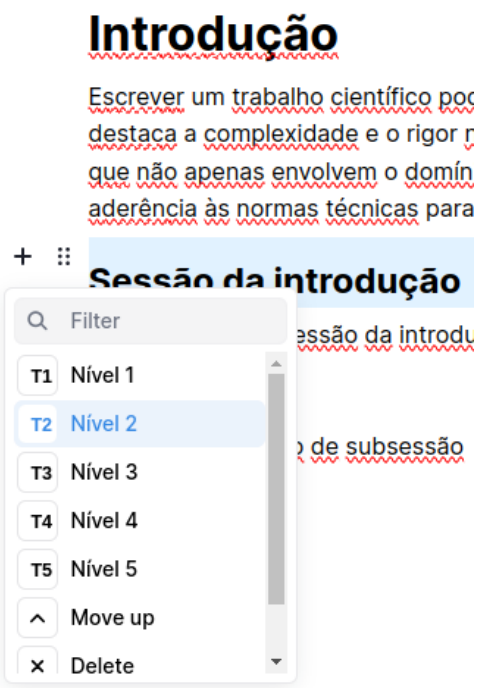
Figura 33 – Plugin de Header na interface gráfica



Fonte: Autoria própria

O nível do título, (Capítulo, sessão e subsessões), é escolhido a partir do submenu do próprio bloco conforme a Figura 34:

Figura 34 – Submenu do Header na interface gráfica



Fonte: Autoria própria

3.2.3.2 Paragraph

Não há uma construção de um plugin paragraph pois se utiliza o plugin padrão já pronto fornecido pelo EditorJS.

3.2.3.3 Image

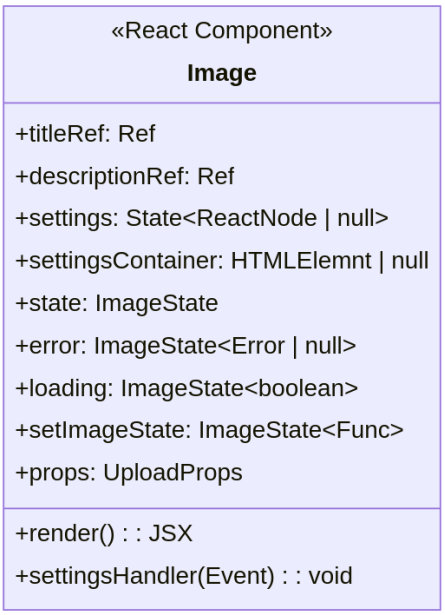
O plugin de imagem é um dos mais importantes e complexos do projeto. É com ele que pode-se trazer ilustrações de imagens para o trabalho escrito, de forma a enriquecer a monografia, seja ela qual for.

O plugin de imagem possui casos de uso interessantes, como: Trazer imagens a partir de links ou de arquivos do sistema; Atribuir um título à imagem; Atribuir uma descrição; Definir a porcentagem de ocupação na folha do documento; e etc...

3.2.3.3.1 O plugin

Observe na Figura 35 a modelagem do plugin de imagem:

Figura 35 – Plugin Image



Fonte: Autoria própria

Tem-se primeiramente, as referências a titleRef e descriptionRef, que são os elementos HTML que receberão o título e descrição, respectivamente. Note também que há um estado denominado settings juntamente com settingsContainer, que guarda um elmento HTML. Isto se dá devido a uma particularidade do componente Header, pois o mesmo tem um menu personalizado que se difere do padrão do EditorJs. Este menu é um componente *React* que é guardado no estado settings, que por ventura é renderizado através do portal no elemento indicado por settingsContainer.

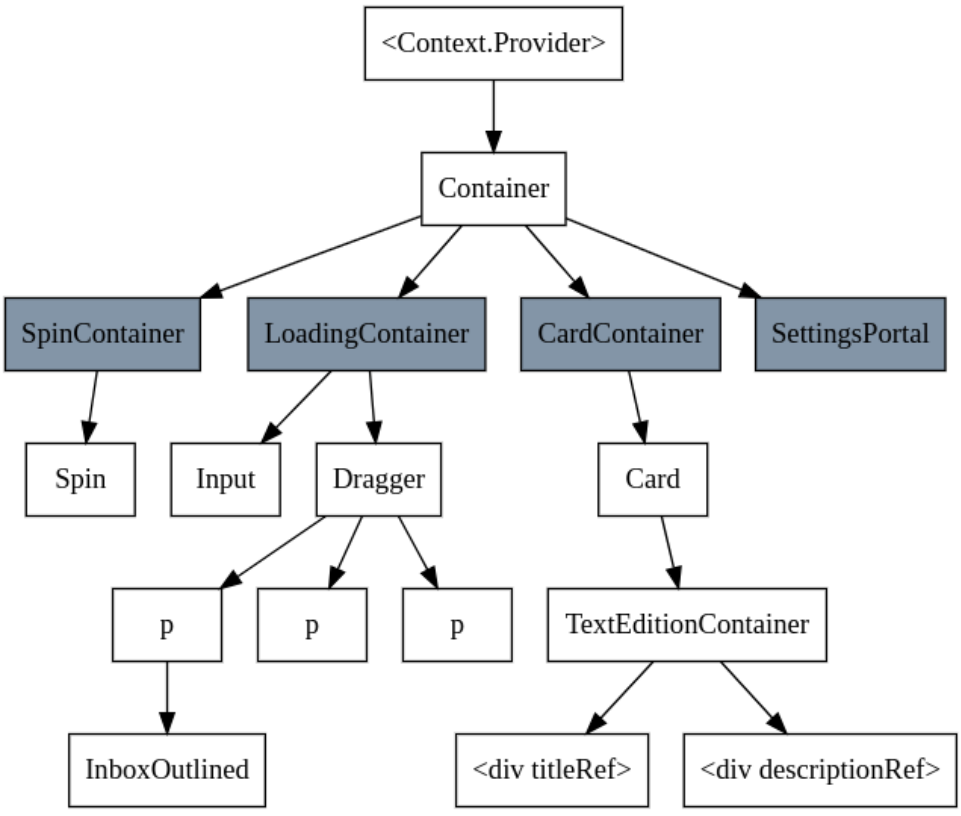
Observe que *state*; *error*; *loading* e setImageState fazem parte de um estado especial denominado ImageState. Isto se dá devido a complexiade do plugin de imagem, que possui um hook personalizado em um código separado, de modo a ajudar a gerenciar os casos de uso do plugin.

O método settingsHandler é o responsável por controlar o estado settings, de modo que é este quem define quando o menu aparecerá em tela. O controlador do menu é o EditorJs, que através da chamada de um evento em sua API define quando tal estado será definido. Desta forma pode-se controlar a renderização de um menu personalizado no EditorJs.

3.2.3.3.2 Renderização do componente imagem

A Figura 36 demonstra a sub-árvore de renderização do plugin de imagem:

Figura 36 – Sub-árvore de renderização do plugin Image



Fonte: Autoria própria

Observe que há quatro componentes demarcados em azul. Estes componentes são componentes de renderização condicional, pois o plugin de imagem altera o seu visual diversas vezes a depender das ações de usuário.

Observe primeiramente o *SpinContainer*. Ele é renderizado sempre que o estado *loading* é verdadeiro, ao passo que seus outros três irmãos não. O SpinContainer

nada mais é do que um receptáculo que possui um componente Spin, que é um indicador animado com papel de mostrar ao usuário que algo está sendo carregado.

LoadingContainer é renderizado quando não há nenhuma imagem atribuída ao componente e é controlado pelo estado *state*. Possui como filhos um componete Input e Dragger. O componente Input é onde pode-se colar um link para uma imagem, ao passo que Dragger é onde se pode arrastar algum arquivo de imagem a partir do Sistema Operacional. O Dragger também possui três elementos de parágrafos que são textos que orientam o usuário em como o mesmo pode fazer uso do componente. A Figura 37 mostra como é a aparência desta parte na interface gráfica.

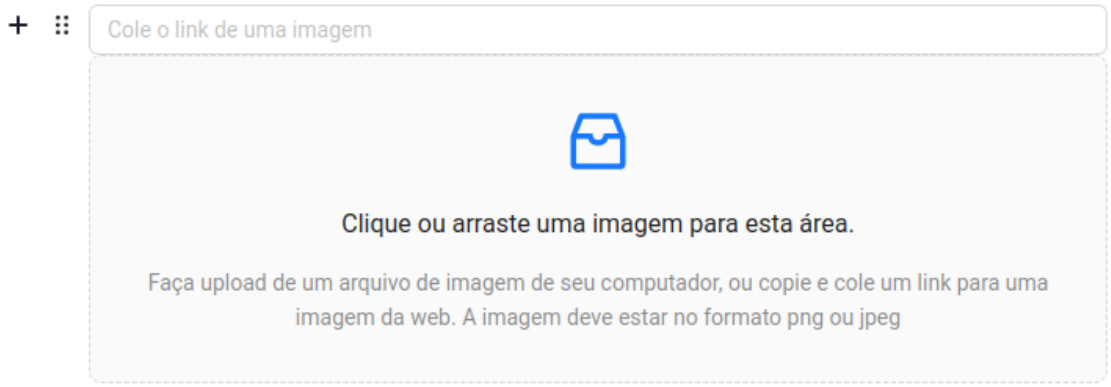
CardContainer é rederizado quando o usuário atribui alguma imagem ao compoente e também é controlado pelo estado *state*. Possui um componente Card que vai abrigar a imagem propriamente dita, além também de uma área textual denominada *TextEditionContainer* onde o usuário pode definir o título e a descrição da imagem. A Figura 38 demonstra a aparência de um componente imagem com uma imagem atribuída.

Por fim o *SettingsPortal* renderizará uma aparente quebra de árvore no DOM do menu do componente. A condição para tal renderização é a chamada do menu a partir da API do EditorJs. Desta forma, tem-se um componente totalmente integrado ao seu menu que responde às ações do mesmo e vice versa. A Figura 39 mostra a renderização deste menu personalizado.

3.2.3.3.3 Aparência do plugin de imagem

Abaixo é descutida as diversas aparências do plugin de imagem a depender de seus respectivos estados e ações do usuário. Inicialmente são renderizadas as instruções para que o usuário possa atribuir uma imagem ao componente. O desenho de tal tentou ser o mais simples e objetivo possível, de modo que o usuário não necessite de nenhum manual para entender o que está acontecendo.

Figura 37 – Visual do plugin de imagem



Fonte: Autoria própria

Quando uma imagem é atribuida, como no caso abaixo, a imagem será exibida em tela através do editor com um título e uma descrição genérica. Caso o usuário tenha importando a imagem da internet através de um link, a descrição padrão automaticamente conterá a referência à sua fonte original. Note que os campos de título e descrição são editáveis.

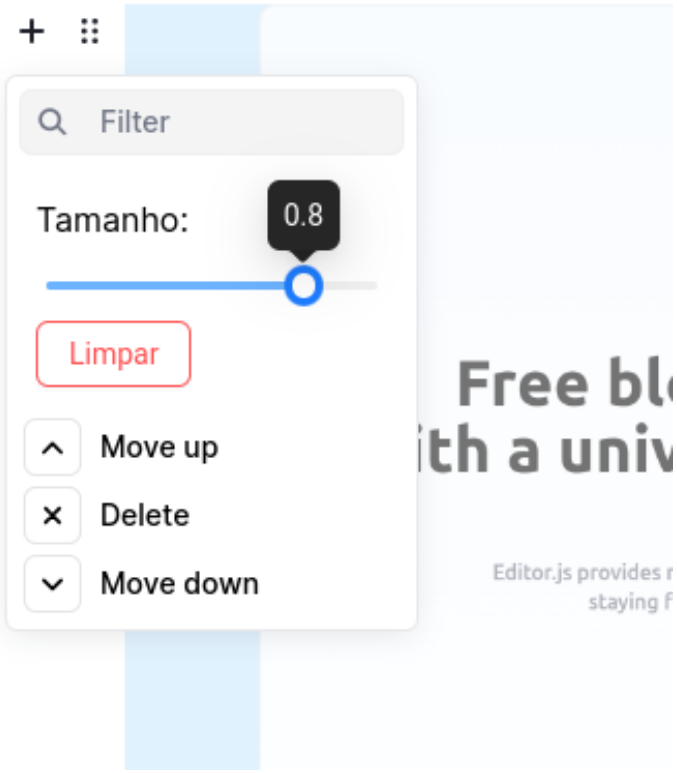
Figura 38 – Imagem atrelada no plugin

Fonte: Autoria própria

O menu de personalização do componente de imagem possui configurações adicionais. Pode-se alterar a porcentagem de ocupação da imagem no documento, bem

como desatribuir a imagem corrente através do botão limpar. Caso o usuário desatribuia, o estado retorna para o exibido na Figura 37.

Figura 39 – Sub-menu do plugin de imagem



Fonte: Autoria própria

3.2.3.4 List

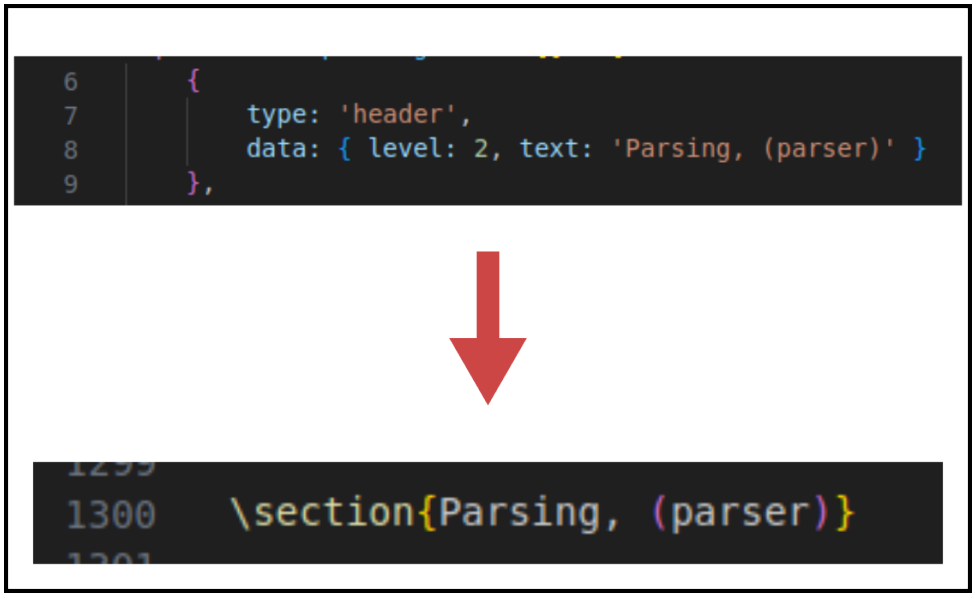
O plugin List que será utilizado será o padrão fornecido pelo EditorJS.

3.3 Parsing, (parser)

O processo de *Parsing* transformará cada bloco provido pela saída do EditorJs em um trecho de código em LaTex. Observe na Figura 40 e Figura 41 os exemplos de *parsing* aplicados a um objeto de *Header*[6] e *Paragraph*[7], respectivamente.

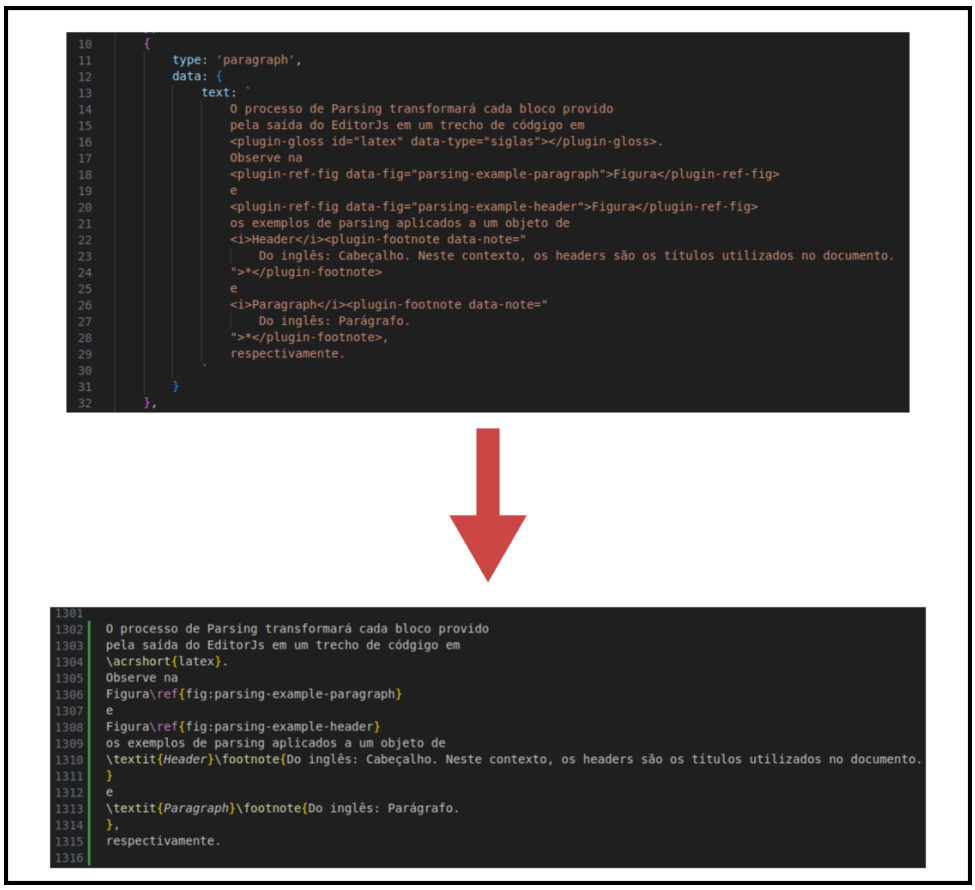
[6] Do inglês: Cabeçalho. Neste contexto, os headers são os títulos utilizados no documento.
[7] Do inglês: Parágrafo.

Figura 40 – Parsing de um bloco Header



Fonte: Autoria própria

Figura 41 – Parsing de um bloco Paragraph



Fonte: Autoria própria

No exemplo do bloco paragraph, é notável o processo de processamento de HTML acontecendo. Note que o conteúdo textual do bloco não é apenas texto simples. Há

quatro tags de marcação personalizadas que se refletem em comandos especiais LaTex, a saber:

1. plugin-gloss
2. plugin-ref-fig
3. i
4. plugin-footnote

A tabela Tabela 9 mostra o mapeamento destas tags, que são plugins *in-line*[8] do EditorJs. As Tags possuem conteúdo e atributo, cada qual se refletindo a uma particularidade do código LaTex a depender de sua natureza.

Tabela 9 – Mapeamento de tags em código LaTex

Tag	**Conteúdo**	**Atributo**	**Valor do Atributo**	**LaTex**
plugin-gloss	Não	id	latex	\acrshort{latex}
plugin-ref-fig	Figura	data-fig	parsing-example-paragraph	\ref{fig:parsing-example-paragraph}
plugin-ref-fig	Figura	data-fig	parsing-example-header	\ref{fig:parsing-example-header}
i	Header	Não	Não	\textit{Header}
plugin-footnote	Não	data-note	<texto da nota>	\footnote{<texto da nota>}
i	Paragraph	Não	Não	\textit{Paragraph}
plugin-footnote	Não	data-note	Do inglês: Parágrafo.	\footnote{Do inglês: Parágrafo.}

3.3.1 Visão geral

O parser da aplicação não depende de nenhuma biblioteca ou *framework* de terceiros, (com excessão do *processHTML* que utiliza a biblioteca *cheerio*). O processo de *parsing* é feito apenas utilizando-se de recursos nativos da linguagem *TypeScript*, (consequentemente JS). A Figura 42 mostra a estrutura de pastas do módulo de parse, com três grandes grupos:

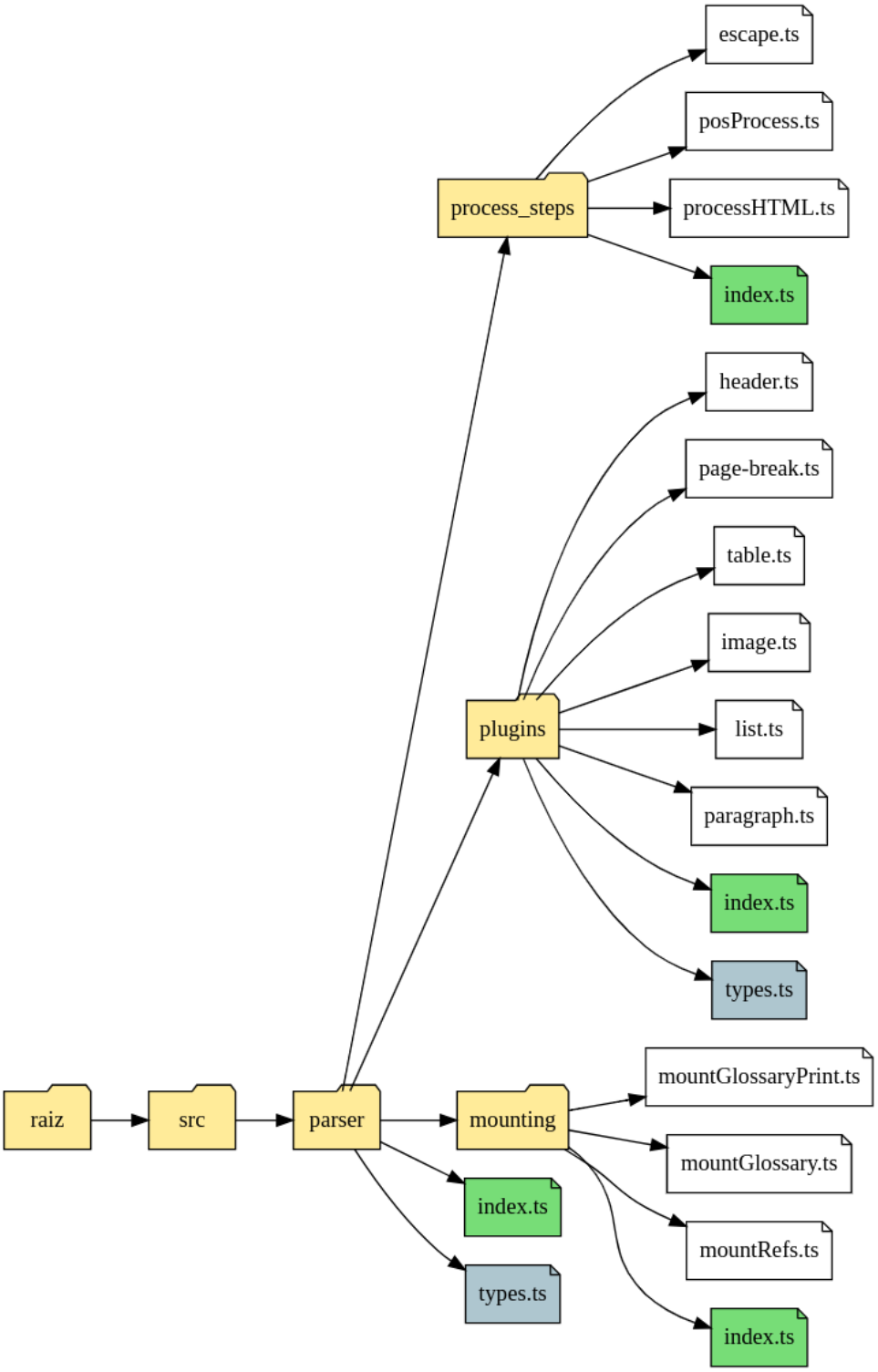
- process_steps

[8] Do inglês: Em linha.

- plugins
- mounting

O process_steps diz respeito a funções utilitárias do processamento de texto. É uma das mais importantes partes pois fornece funções que serão utilizadas a todo momento em outras partes do *parsing*. Provê segurança pois nele reside as funções de escape de caracteres especiais e entre outros. O plugins fornece o processamento respectivo de cada plugin da aplicação, bem como o modo em que cada um deverá ser convertido em código LaTex. O mounting diz respeito a montagens de partes dinâmicas do documento LaTex.

Figura 42 – Estrutura de pastas do módulo de parse



Fonte: Autoria própria

3.3.2 Estapas de processamento

O principal plugin da aplicação a ser processado será o de paragraph. A informação principal deste plugin é tão somente o texto de marcação HTML. Devido a este fato, este plugin será o que mais utilizará as funções providadas pelo process_steps.

A Figura 43 ilustra um ciclo de processamento de texto de marcação:

Figura 43 – Um ciclo de processamento de texto

Fonte: Autoria própria

3.3.2.1 Escape de caracteres

O escape de processamento é a primeira etapa do processamento de texto. Necessária para evitar o primeiro problema de transformar o texto de marcação em código LaTex, que é o problema dos caracteres especiais.

O código LaTex, (como dito na fundamentação teórica), é um código legível tanto para seres humanos quanto por máquinas. Devido a isto, existem alguns caracteres especiais de controle que ajudam a definir como determinado conteúdo aparecerá no documento. Por exemplo: O caractere \ é um dos mais importantes caracteres do LaTex, pois ele define uma gama de comandos, como por exemplo o \chapter{**texto**}, que define **texto** como sendo um capítulo no documento. Mas e se o usuário digitar no documento um caractere \? Quais problemas isso poderá gerar?

Caso o usuário digite um \ no documento e este caractere simplesmente seja transcrito no documento LaTex, gerará uma série de efeitos indesejados, desde comandos inexistentes como: \esse-comando-não-existe, como o uso acidental de comandos do próprio LaTex, como: \chapter; \section ou \subsection. Para evitar isto qualquer ocorrência de \ no texto será substituída por \textbackslash, que é um comando LaTex responsável por imprimir uma \ no corpo do texto. A Tabela 10 mostra as ocorrências de todos os caracteres que devem ser substituídos por algum comando especial no LaTex que possua o mesmo significado textual:

Tabela 10 – Mapeamento de escape de caracteres para código LaTex

Caractere	**Substituição**
\	\textbackslash
#	\#
$	\$$
%	\%
^	\textasciicircum
&	\&
_	_
{	\{
}	\}
[	{[}
]	{]}
~	\textasciitilde

Fonte: Adaptado de: (MADEIRA, 2020)

Todas as ocorrências destes caracteres devem ser devidadmente substituídas afim de evitar qualquer problema de interpretação do compilador LaTex.

3.3.2.1.1 Código do escape.ts

Abaixo tem-se a aplicação do processamento de escape de caracteres em *typescript*. a função escapeCharacters recebe uma *string* na linha 1, e das linhas 3 a 14 faz uma sucessão de novas atribuições desta nova *string*. As atribuições consistem de uma nova *string* que, através da função replace, substituem as expressões regulares pela nova *string*, que seguem de acordo com a Tabela 10. Ao final, na linha 16, a nova *string* é retornada.

```
1 export function escapeCharacters(str: string){
2     let newStr = str;
3     newStr = newStr.replace(/\\/gm, '\\textbackslash ');
4     newStr = newStr.replace(/#/gm, '\\#');
5     newStr = newStr.replace(/\$/gm, '\\$$');
6     newStr = newStr.replace(/%/gm, '\\%');
7     newStr = newStr.replace(/\^/gm, '\\textasciicircum ');
8     newStr = newStr.replace(/&/gm, '\\&');
```

```
9     newStr = newStr.replace(/_/gm, '\\_');
10    newStr = newStr.replace(/\{/gm, '\\{');
11    newStr = newStr.replace(/\}/gm, '\\}');
12    newStr = newStr.replace(/\[/gm, '{[}');
13    newStr = newStr.replace(/\]/gm, '{]}');
14    newStr = newStr.replace(/~/gm, '\\textasciitilde ');
15
16    return newStr;
17 }
```

3.3.2.2 Processamento de HTML

O processamento de HTML é a segunda etapa do processo de *parsing*. É nele que os plugins in-line customizados são transformados em comandos LaTex. A Figura 44 inlustra como os plugins de glossário e referência de figuras se comportam no código LaTex através dos quadrados coloridos:

Figura 44 – Conversão de plugins em código latex

```
text: `
    O processamento de
    <plugin-gloss id="html"></plugin-gloss>
    é a segunda etapa do processo de parsing. É
    nele que os plugins in-line customizados são transformados
    comandos
    <plugin-gloss id="latex"></plugin-gloss>.
    A figura
    <plugin-ref-fig data-fig="html-processing-example">Figura</plugin-ref-fig>
    inlustra como os plugins de glossário e referência de figuras
    se comportam no código
    <plugin-gloss id="latex"></plugin-gloss>:
`

1533
1534    O processamento de
1535    \acrshort{html}
1536    é a segunda etapa do processo de parsing. É
1537    nele que os plugins in-line customizados são transformados
1538    comandos
1539    \acrshort{latex}.
1540    A figura
1541    Figura\ref{fig:html-processing-example}
1542    inlustra como os plugins de glossário e referência de figuras
1543    se comportam no código
1544    \acrshort{latex}:
1545
```

Fonte: Autoria própria

No caso do plugin-gloss, há também um objeto de glossário que contém todas as definições de siglas e abreviaturas. Este objeto é transformado e montado em declarações LaTex para serem usadas ao longo do documento. Para mais detalhes deste processo, verifique a sessão de montagem

A Tabela 11 mapeia como cada plugin se comportará no código LaTex:

Tabela 11 – Mapeamento de pugins para código latex

Tag plugin	**Atributo(s)**	**LaTex**
plugin-gloss	id	\acrshort{<id>}
plugin-ref	id	\cite{<id>}
plugin-footnote	data-note	\footnote{<data-note>}
plugin-ref-fig	data-fig	Figura\ref{fig:<data-fig>}
plugin-ref-table	data-table	Tabela\ref{fig:<data-table>}
br	Não	\\
strong	Não	\textbf{<conteúdo>}
i	Não	\textit{<conteúdo>}

Observe que no processamento de HTML o conteúdo da maioria das tags é irrelevante, com excessão das tags strong e i. Isso se dá pois na maioria das vezes a informação de interesse é guardada nos atributos, ficando o conteúdo servindo apenas para que se exiba o plugin de forma confortável no texto que o usuário está digitando na UI. As tags strong e i, são respectivamente, o negrito e itálico. São padrões do HTML e aqui seus conteúdos são aproveitados para dentro dos comandos equivalentes em LaTex.

3.3.2.2.1 Código do processamento HTML

Para processar os plugins está sendo utilizada a biblioteca *cheerio*. A *string* a ser analisada, recebida na linha 3, passa por uma checagem de tags presentes. Por exemplo: Observe a linha 5. Sempre que houver qualquer ocorrência de plugin-ref, esta é substituída através do comando replaceWith pelo texto: \cite{${$(node).attr('id')}}. Observe que a parte $(node).attr('id') é o que recupera o atributo "id" da *tag*.

```
1 import * as cheerio from 'cheerio';
2
3 export function processHTML(text: string): string{
4     const $ = cheerio.load(text);
5     $('plugin-ref').replaceWith((_, node) => {
6         return `\\cite{${$(node).attr('id')}}`;
7     }); [...]
```

Por fim, na linha 40 é retornado o resultado do processamento em HTML. Note que ele é recuperado a partir do nó *body* do documento virtual, e então, a função text() é chamada para recuperar a representação textual do que foi processado.

```
39 [...]
40     return $('body').text();
41 }
```

O código completo do *processHTML*.ts juntamente com todos os plugins pode ser encontrado no repositório do projeto.

3.3.2.3 Pós processamento

Após feito o processo de escape e processamento de HTML, ainda podem sobrar alguns caracteres remanescentes problemáticos à correta interpretação do código LaTex. Por exemplo: Os caracteres < e > são escapados no HTML como as entidades lt e gt, respectivamente. Eles por sua vez, ao passar pelo escape de caracteres, possuem o caractere & escapado como \&. Seu código final acaba resultando em \< e \>, que tem de novamente ser convertido em < e >. Observe o código abaixo:

```
1  export function posProcess(str: string): string{
2      let new_str = str.replace(/\\</gm, '<');
3      new_str = new_str.replace(/\\>/gm, '>');
4      new_str = new_str.replace(/\\"/gm, '"');
5      new_str = new_str.replace(
6          /"(?!\w|\}|\)|\\|~|\.|,|\{|\(|\[)/gm,
7          '"~'
8      );
9      new_str = new_str.replace(
10         /(\\textbackslash)((\s?\n)|(\s{2,}))/gm,
11         '$1~'
12     );
13     return new_str;
14 }
```

3.3.3 Plugins

O *parsing* dos plugins é algo essencial na montagem do documento ao qual estar-se a escrever. São os plugins que formam cada pequena unidade de bloco que comporá um documento que muitas vezes se tornará gigantesco.

3.3.3.1 Tipagem

Figura 45 – Caminho da pasta types dos plugins no parser

Fonte: Autoria própria

O arquivo de tipagem é onde é concentrada todas as anotações de tipo de cada bloco. O documento todo é um array de objetos, no qual cada um desses objetos corresponde a um bloco do documento. É através de cada tipo de bloco que pode-se por exemplo, discriminar qual tipo de bloco está sendo tratado no momento do processamento. Observe por exemplo a anotação de tipo do ParagraphBlock:

```
37 [...]
38 export interface ParagraphBlock {
39     type: 'paragraph'
40     id: string
41     data: {
42         text: string
43     }
44 }
45 [...]
```

Todo bloco é composto basicamente por estas três propriedades: type; id e data. Sendo que a propriedade data variará sua forma a depender do bloco que está sendo definido. Este código nunca é executado ou incluido na versão de execução final em JS, (por se tratar de uma anotação de tipo). Mas é de extrema utilidade quando se trata de guiar o desenvolvedor na hora de instanciar um objeto do tipo que é definido pelo interface[9] ParagraphBlock.

Observe como o interface do HeaderBlock se diferencia do ParagraphBlock em sua anotação de tipo:

```
5 [...]
6 export interface HeaderBlock {
7     type: 'header'
8     id: string
9     data: {
```

[9] Neste contexto, está-se utilizando o termo interface em seu original inglês, conforme a documentação do *TypeScript*. Devido a este fato, interface (não confundir com interface de usuário) será tratado com pronome masculino.

```
10          level: 1 | 2 | 3 | 4 | 5
11          text: string
12      }
13  }
14  [...]
```

No HeaderBlock, a propriedade data é composta por outras duas propriedades: text e level. Ao refletir sobre isso, percebe-se que essas duas propriedades são essenciais para definir o que é um título em um documento. O level (nível) indica o nível hierárquico do título, enquanto o text (texto) representa o conteúdo exibido como título. Perceba que há uma limitação na definição do nível do título, com valores variando apenas de 1 a 5. Cada um desses valores corresponde respectivamente às sessões: primária, secundária, terciária, quaternária e quinária.

Observe na Tabela 12 a relação do level do título com o código LaTex correspondente:

Tabela 12 – Relação do level do título com o código latex

Level	**Código LaTex**
1	\chapter{<conteúdo>}
2	\section{<conteúdo>}
3	\subsection{<conteúdo>}
4	\subsubsection{<conteúdo>}
5	\subsubsubsection{<conteúdo>}

3.3.3.2 Paragraph, (parágrafo)

O bloco de parágrafo parece de longe um dos blocos mais simples de todos. Ele simplesmente retorna o texto do bloco recebido, fazendo-o passar pelas etapas de processamento conforme já descrito na Figura 43. Observe na linha 7 como o parâmetro block é anotado com o tipo ParagraphBlock. É neste caso que entra a utilidade do *TypeScript*. Pois além de prevenir erros não deixando que se passe um bloco incorreto para ser processado por essa função, tem-se a condição de saber exatamente o que tem dentro da propriedade data na linha 11. Neste caso, apenas passa-se o texto para o processamento.

```
1  import { escapeCharacters } from '@/parser/process_steps/escape';
2  import { processHTML } from '@/parser/process_steps/processHTML';
3  import { ParagraphBlock } from '@/parser/types';
4  import { posProcess } from '@/parser/process_steps/posProcess';
```

```
5
6
7  export function getParagraph(block: ParagraphBlock){
8      return posProcess(
9          processHTML(
10             escapeCharacters(
11                 block.data.text
12             )
13         )
14     );
15 }
```

3.3.3.3 Header, (cabeçalhos)

O parse do bloco de Header já distoa-se um pouco do bloco de paragraph. Aqui é notável a aplicação do exposto na Tabela 12 onde o cada level se traduz em um comando diferente no latex. Através da clausula switch, o level é testado afim de retornar as particularidades de cada tipo de título. Observe que o level 4, implementado na linha 13, possui um comando a mais de \underline para que o título apareça como sublinhado. Este comando a mais foi implementado para atender as particularidades exidas pelo documento da PUC-GO em: (PUC-GO, 2022).

```
1  import { escapeCharacters } from '@/parser/process_steps/escape';
2  import { HeaderBlock } from '@/parser/types';
3
4  export function getHeader(block: HeaderBlock){
5      switch(block.data.level){
6          case 1:
7              return `\\chapter{${escapeCharacters(block.data.text)}}`
8          case 2:
9              return `\\section{${escapeCharacters(block.data.text)}}`
10         case 3:
11             return `\\subsection{${escapeCharacters(block.data.text)}}`
12         case 4:
13             return `\\subsubsection{\\underline{${
14                 escapeCharacters(block.data.text)
15             }}}`
16         case 5:
17             return `\\subsubsubsection{${
18                 escapeCharacters(block.data.text)
```

```
19              }}‘
20      }
21  }
```

3.3.3.4 Image, (imagens)

O parse do plugin de imagem é um dos mais complexos em termos de comandos LaTex. Ele retorna toda uma estrutura de modo que o compilador de documentos LaTex possa renderizar a imagem corretamente e seja capaz de referenciá-la ao longo do texto.

Nas linhas 8 a 12 tem-se a desconstrução das propriedades da imagem que estão presentes na propriedade data do bloco. Estas propriades são: uuid[10]; title; width; description e fileType.

```
1  import { escapeCharacters } from '@/parser/process_steps/escape';
2  import { posProcess } from '@/parser/process_steps/posProcess';
3  import { processHTML } from '@/parser/process_steps/processHTML';
4  import { ImageBlock } from '@/parser/types';
5
6  export function getImage(block: ImageBlock){
7      const {
8          uuid,
9          title,
10         width,
11         description,
12         fileType,
13     } = block.data;
14 [...]
```

Na linha 20 tem-se a inclusão do título da imagem. Note que o título somente precisa passar pelo processo de escape de caracteres, uma vez que no mesmo não haverá plugins no corpo do texto. Nas linhas 25 à 33 há a inclusão da descrição da imagem. Note que diferente do título, a descrição passa por todo o ciclo de processamento, pois na descrição o usuário pode incluir referências e outras coisas que resultam na presença de tags de plugin no corpo do texto.

As linhas 21 a 23 tratam do tamanho que a imagem vai ocupar na tela, bem como a referência ao arquivo de imagem que será renderizado. As propriedades width, uuid, e fileType são utilizadas neste processo. Cada imagem é guardada na pasta

[10] UUID, *Universally Unique Identifier* (Identificador Universalmente Único).

interna images com o nome do uuid que o *software* define, bem como sua extensão de arquivo.

A linha 24 utiliza o uuid para definir a label da imagem. Desta forma, a imagem a que se trata o bloco torna-se referenciável no texto do documento.

```
13 [...]
14
15
16 // H Prevents figure to be placed incorrectly
17 return `
18     \\begin{figure}[H]
19         \\centering
20         \\caption{${escapeCharacters(title)}}
21         \\includegraphics[width=${
22             width.toFixed(1)
23         }\\textwidth]{./images/${uuid}.${fileType}}
24         \\label{fig:${uuid}} \\\\
25         \\textnormal{\\fontsize{10pt}{12pt}${
26             posProcess(
27                 processHTML(
28                     escapeCharacters(
29                         description
30                     )
31                 )
32             )
33         }}
34     \\end{figure}
35 `.trim().replace(/^\s{8}/gm, '');
36 }
```

3.3.3.5 List, (listas)

Existem dois tipos de lista, as enumeradas e as não enumeradas, respectivamente enumerate e itemize no código LaTex. A linha 9 utiliza a propriedade type para definir qual tipo de lista será utilizada. Por fim, a lista é composta de uma iteração sobre as *strings* do array presente na propriedade list. Este array, após ajustado com a função map, é transformado em uma *string* unindo seus itens por uma quebra de linha com o auxílio da função join na linha 11. Observe também o processamento textual presente na linha 11. Isso se dá pela liberdade do usuário em utilizar plugins nos campos de lista.

```
import { escapeCharacters } from "@/parser/process_steps/escape";
import { processHTML } from "@/parser/process_steps/processHTML";
import { ListBlock } from "@/parser/types";
import { posProcess } from "@/parser/process_steps";

export function getList(list: ListBlock){
    const { data: { type, list: __list } } = list;
    return `
    \r\\begin{${ type === 'bullet' ? 'itemize' : 'enumerate' }}
        ${__list.map(el => `\r\t\\item ${
            posProcess(processHTML(escapeCharacters(el))) }`).join('')
        }
    \r\\end{${ type === 'bullet' ? 'itemize' : 'enumerate' }}
    `.trim()
}
```

3.3.3.6 Page Break, (quebra de página)

O bloco de PageBreak é o bloco de parser mais simples de todos. Embora receba como parâmetro um bloco do tipo PageBreakBlock, nem uso do mesmo é feito. O PageBreak apenas retorna um comando em LaTex que limpa a página e faz com que o conteúdo após o mesmo seja escrito na próxima página.

```
import { PageBreakBlock } from '@/parser/types';

export function pageBreak(block: PageBreakBlock){
    return '\\clearpage';
}
```

3.3.4 Montagem

O processo de montagem é o processo no qual algumas partes do código LaTex são montadas separadamente para fins de organização. Após montadas, cada uma destas partes são incluídas no código principal através do comando \input.

3.3.4.1 Glossário

Cada item de glossário, seja ele uma abreviação ou sigla, deve ser definido no documento antes que se possa fazer uso do mesmo. Observe abaixo o exemplo de uma definição de sigla em LaTex:

```
\newacronym[type=sigla]{abnt}{ABNT}{
    Associação Brasileira de Normas Técnicas
}
```

Além de montados, cada clossário deve ser exibido com o comando \printglossary. Veja o código abaixo:

```
\printglossary[type=sigla,title=LISTA DE SIGLAS]
\clearpage

\printglossary[type=abreviacao,title=LISTA DE ABREVIATURAS]
\clearpage
```

Isto garante que os devidos glossários sejam exibidos com folga de uma página, com seus respectivos títulos.

A função mountGlossary exportada em mountGlossary.ts contém sua implementação, que faz todo o processamento dos items de glossário a partir de um GlossaryObjectType:

```
[...]
export function mountGlossary(glossary: GlossaryObjectType){
    const header = `
        \\newglossary*{abreviacao}{Lista de abreviaturas}
        \\newglossary*{sigla}{Lista de siglas}
        \\newglossary*{simbolo}{Lista de símbolos}
    `.trim().replace(/^\s{8}/gm, '');

    const gloss_arr = Object.keys(glossary).map(
        key => ({ ...glossary[key], key })
    );

    const acronyms = gloss_arr
        .filter(({ type }) => type === 'sigla')
        .sort(({ label: a }, { label: b }) => a.localeCompare(b))
        .map(({ short, label, type, key }) => {
            return `\\newacronym[type=${type}]{${key}}{${
                escapeCharacters(short)
            }}{${
                escapeCharacters(label)
```

```
23              }}‘
24          })
25          .join('\n');
```

Note que todos os objetos de glossário vem misturados no parâmetro glossary da função. Para montar as siglas da linha 15, por exemplo, é feita uma filtragem no tipo de glossário, uma ordenação através da função sort, e por fim, a conversão em código LaTex através da função map na linha 18. Por fim, o array de siglas é unido através do caractere de quebra de linha por meio da função join em 25. O mesmo processo acontece com a lista de abreviações e de termos.

Por fim, basta unir tudo em uma *string* conforme mostrado na linha 65 e 66 abaixo. As linhas 68 e 69 foram uma tentativa de customizar o estilo do glossário e seu título. Por hora, esta parte está omitida do trabalho. A linha 67 adiciona o comando \makeglossaries, que diz ao LaTex para montar o glossário.

```
64 [...]
65     let str = ‘${header}\n\n${acronyms}\n‘;
66     str = str.concat(‘${abbreviations}\n${symbols}\n\n‘);
67     str = str.concat(‘\\makeglossaries‘);
68     // str = str.concat('\n\n').concat(custom_style);
69     // str = str.concat('\n\n').concat(custom_title);
70
71
72     return str;
73 }
```

Após todo o processamento seu resultado é guardado em um arquivo denominado makeGlossaries.tex a ser importado posteriormente pelo código fonte contruído no processo de *Parsing*.

3.3.4.2 Referências

A parte de referênciação é uma das automações mais úteis do projeto, pois adiciona referências cruzadas ao projeto facilmente navegável na versão digital do mesmo.

Observe a função mountRefs exportada a partir do arquivo mountRefs.ts:

```
1 import { RefsObjectType } from '@/parser/types';
2
3 export function mountRefs(refs: RefsObjectType){
4     return Object.keys(refs).map(key => {
5         const ref  = refs[key];
```

```
6           const { type, ...restRefs } = ref;
7           if(type){
8               return `
9                   @${type}{${key},
10                      ${(() => {
11                          return Object.keys(restRefs).map(ref_key => {
12                              switch(type){
13                                  case 'misc':
14                                  case 'book':
15                                  case 'article':
16                                      // @ts-ignore
17                                      const value = restRefs[ref_key];
18                                      if(ref_key === 'author')
19                                          return `${ref_key}={${
20                                              (value as string[])
21                                                  .join(' and ')
22                                      }}`
23                                      else if(ref_key === 'edition')
24                                          return `${ref_key}={${
25                                              (value as number)
26                                                  .toString()
27                                                  .concat('st')
28                                      }}`
29                                      else
30                                          return `${ref_key}={${value}}`
31                              }
32                          }).join(',\n    ')
33                  })()}
34              }
35          `.trim().replace(/^\s{16}/gm, '');
36      }
37
38      return '';
39  }).join('\n\n');
40  }
```

4 CONSIDERAÇÕES FINAIS

Este instrumento visou durante seu processo de elaboração o desenvolvimento da plataforma de modo que a mesma seja uma ferramenta útil e de facil compreensão para o mais diverso público. Desta forma, o usuário da ferramenta precisa se preocupar apenas com seu conteúdo defendido, de modo que a forma de sua apresentação é um encargo da ferramenta.

Esta concepção inicial constitui-se numa proposta de mudança de paradigma na forma e no fluxo com que se escreve um trabalho acadêmico, fornecendo uma alternativa mais amigável às formas tradicionais de escrita utilizando-se o *Microsoft Word*, *Libre Office* ou o próprio LaTex em sua forma pura.

4.1 O que foi desenvolvido

Foi desenvolvido um programa aplicativo, no modelo cliente-servidor, que serve um sistema Web para o usuário final. O sistema consiste em um rico editor de texto baseado em blocos em que, através de plugins, monta-se todo um complexo trabalho de monografia constituído de pequenos blocos simples.

O processo de *parsing* se encontra totalmente funcional para todos os seguintes blocos:

- Code, (código)
- DirectCite, (citação direta)
- Header, (cabeçalhos, títulos)
- Image, (imagens)
- List, (listas)
- PageBreak, (quebra de página)
- Paragraph, (parágrafo)
- Table, (tabelas)

Deste modo, qualquer bloco saída do editor já é devidamente transpilado para seu equivalente código LaTex e consequentemente compilado no PDF final.

Em se tratando da interface do editor com o usuário, nem todos os blocos plugins foram desenvolvidos conforme a proposta inicial deste instrumento, (vide "O que deu errado" e "Trabalhos futuros"). Os seguintes plugins encontram funcionais no editor:

- Paragraph, (parágrafo)
- Header, (cabeçalho)
- Image, (Imagem)

Este trabalho não foi desenvolvido utilizando-se o aplicativo em sua interface gráfica, mas utilizou todos os recursos providos pelo processo de *parsing*. O código de saída do editor foi escrito manualmente e em seguida compilado. O processo de compilação pode ser repetido rodando o comando abaixo a partir da pasta raiz do projeto no GitHub:

```
1 yarn tex-make
```

4.2 O que deu errado

Devido a duração do processo de pesquisa e do prazo determinado para a finalização deste instrumento, alguns plugins não foram desenvolvidos ou foram desenvolvidos parcialmente. Os plugins in-line, que consistem na atribuição de glossários e referências bibliográficas, não foram desenvolvidos para a interface gráfica, porém possuem seus equivalentes em termos de *parsing*.

4.3 Performance

Todo este trabalho foi desenvolvido utilizando-se do processo de *parsing* desenvolvido pelo mesmo, ao passo que paralelamente foi desenvolvida a interface. O código LaTex resultante gerou um arquivo texto de aproximadamente 176kb, que foi compilado sem demais problemas e com um tempo de menos de 60 segundos. Além também de mais de 2000kb em imagens.

4.4 Trabalhos futuros

Há muitas coisas novas que podem ser implementadas e desenvolvidas a partir da concepção inicial deste trabalho. Plugins novos podem ser desenvolvidos e ajustes de interface e melhorias de experiência de usuário podem ser implementados.

4.4.1 Plugins inline

Os plugin inline são de grande importância para uma boa experiência de usuário e escrita prazerosa. Através deles podem-se atribuir glossários, referências cruzadas a tabelas e figuras, e referências bibliográficas. Seus respectivos códigos de *parsing* encontram-se integrados ao projeto, porém seus equivalentes em termos de interface e experiência de usuário ainda não foram desenvolvidos.

4.4.2 Tabelas

As tabelas são plugins úteis na exbição e organização de determinados tipos de informação. Seu *parsing* encontra-se no projeto poŕém ainda não há o equivalente plugin integrado ao editor.

4.4.3 Matemática

O LaTex é bastante versátil em termos de equações matemáticas. Neste sentido, pode-se desenvolver um plugin em que o usuário possa digitar as equações na sintaxe matemática do LaTex ou ainda uma interface low code em que o mesmo possa escrevê-las clicando em ícones na tela.

4.4.4 Gráficos e diagramas

Uma grande utilidade poderia ser a integração com bibliotecas e *frameworks* como Mermaid e Graphviz. Estas bibliotecas são de facil compreensão e muito úteis na hora de criar qualquer tipo de gráfico ou diagrama.

4.5 Finalização

Levando-se em consideração todas as funcionalidades desenvolvidas, (principalmente no que diz respeito à questão de *parsing*), o sistema funcionou de forma coerente trazendo bons resultados, se apresentando como uma solução sub-ótima, levando-se em consideração a concepção inicial.

REFERÊNCIAS

ABNT. *Quem somos? ABNT - Associação Brasileira de Normas Técnicas*. 2021. [Online; acessado em 15-Maio-2023]. Disponível em: <http://www.abnt.org.br/institucional/sobre>.

AMAZON; AWS. *O que é uma framework em programação e engenharia?* [Online: acessado em 25-Maio-2024]. Disponível em: <https://aws.amazon.com/pt/what-is/framework/>.

ARAUJO, L. C. Abntex2. Abntex, 2012. [Online: acessado em 06-Junho-2024]. Disponível em: <https://www.abntex.net.br/>.

BRASIL. Lei nº 12.965, de 23 de abril de 2014. dispõe sobre os direitos civis na internet. *Diário Oficial da União*, 2014.

CASTRO, C. M. *Como redigir e apresentar um trabalho científico*. [S.l.]: São Paulo: Pearson Prentice Hall, 2011.

CHEERIO. NPM, 2022. [Online: acessado em 06-Junho-2024]. Disponível em: <https://www.npmjs.com/package/cheerio>.

EDITORJS. Base concepts. [Online: acessado em 30-Maio-2024]. Disponível em: <https://editorjs.io/base-concepts/>.

FILHO, N. F. D.; NOGUEIRA, M. T. Latex: Benefícios, mitos e temores. *Revista SBC Horizontes*, Sociedade Brasileira de Computação (SBC), 2009.

INFORMAL, D. Setar. Dicionário inFormal (SP), 2018. [Online: acessado em 04-Junho-2024]. Disponível em: <https://www.dicionarioinformal.com.br/significado/setar/17876/>.

MACHADO, E. Hardware e software. Enciclopédia Significados. Disponível em: <https://www.significados.com.br/hardware-e-software/s>.

MADEIRA, D. *Tutorial básico de LATEX*. [S.l.: s.n.], 2020.

MAGALHãES, C. Regex: o guia essencial das expressões regulares. *Blog DP6*, Medium, 2022. [Online: acessado em 01-Junho-2024]. Disponível em: <https://blog.dp6.com.br/regex-o-guia-essencial-das-express%C3%B5es-regulares-2fc1df38a481>.

MDN. Css. Mozilla Developer Network, 2023. [Online: acessado em 16-Maio-2023]. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/CSS>.

MDN. Design responsivo. Mozilla Developer Network, 2023. [Online: acessado em 28-Maio-2024]. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Learn/CSS/CSS_layout/Responsive_Design>.

MDN. Javascript. Mozilla Developer Network, 2023. [Online: acessado em 16-Maio-2023]. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>.

MDN. Json. Mozilla Developer Network, 2024. [Online: acessado em 30-Maio-2024]. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/JSON>.

MDN. Regular expressions. *Web Docs*, Mozilla Developer Network, 2024. [Online: acessado em 01-Junho-2024]. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_expressions>.

MEDEIROS, J. B. *Redação científica: a prática de fichamentos, resumos, resenhas*. 11st. ed. [S.l.]: São Paulo: Atlas, 2012.

MOLGADO, V. *A Evolução da Web: linha do tempo interativa da História da internet.* 2016. [Online: acessado em 15-Maio-2023]. Disponível em: <https://labvis.eba.ufrj.br/a-evolucao-da-web-linha-do-tempo-interativa-da-historia-da-internet/>.

MORAIS, P. A história do react! Medium, 2021. [Online: acessado em 29-Maio-2024]. Disponível em: <https://medium.com/@ppternunes/a-hist%C3%B3ria-do-react-ba346c416fe1>.

NEXTJS. Introduction. 2024. [Online: acessado em 29-Maio-2024]. Disponível em: <https://nextjs.org/docs>.

PIMENTEL, E. O que é dom? Alura, 2022. [Online: acessado em 01-Junho-2024]. Disponível em: <https://www.alura.com.br/artigos/o-que-e-o-dom>.

PUC-GO. Coordenação de tcc. manual para elaboração de trabalho de conclusão de curso. Pontifícia Universidade Católica de Goiás, 2022.

REACTJS. Writing markup with jsx. 2024. [Online: acessado em 01-Junho-2024]. Disponível em: <https://pt-br.react.dev/learn/writing-markup-with-jsx>.

REDDY, M. *API Design for C++*. [S.l.]: Elsevier Science, 2011. ISSN 9780123850041.

SANTOS, I. R. As dificuldades na construção do trabalho de conclusão de curso: Percepção de estudantes egressos do curso de ciências contábeis. 2020. 69. trabalho de conclusão de curso (graduação) - ciências contábeis. Universidade Federal da Paraíba, João Pessoa, 2020.

SEVERINO, A. J. *Metodologia do trabalho científico*. 24st. ed. [S.l.]: Cortez, 2017.

SILVA, M. O. da; VITORIA, M. I. C. A experiência de escrita no trabalho de conclusão de curso: Percepções de alunos de um curso superior de tecnologia em gestão em recursos humanos. CAMINE: Caminhos da Educação, Franca, 2014. ISSN 2175-4217.

TOTVS. *O que é back-end e qual seu papel na programação?* 2020. [Online: acessado em 01-Junho-2024]. Disponível em: <https://www.totvs.com/blog/developers/back-end/>.

TOTVS. *Front end: O que é, como funciona e qual a importância*. 2021. [Online: acessado em 25-Maio-2024]. Disponível em: <https://www.totvs.com/blog/developers/front-end/>.

TYPESCRIPT. *TypeScript*. [Online: acessado em 29-Maio-2024]. Disponível em: <https://www.typescriptlang.org/>.

VITORIANO, D. *O que é a Web Moderna*. 2019. Medium. [Online: acessado em 15-Maio-2023]. Disponível em: <https://blog.danvitoriano.com.br/o-que-%C3%A9-a-web-moderna-b01e4df9a565>.

W3C. Html. World Wide Web Consortium, 2023. [Online: acessado em 16-Maio-2023]. Disponível em: <https://html.spec.whatwg.org/multipage>.

W3C. Typescript introduction. W3SCHOOLS, 2024. [Online: acessado em 29-Maio-2024]. Disponível em: <https://www.w3schools.com/typescript/typescript_intro.php>.

WIKIPEDIA. Latex. Wikipédia, 2024. [Online: acessado em 06-Junho-2024]. Disponível em: <https://pt.wikipedia.org/wiki/LaTeX>.