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

Fanny Batista Vieira

Docs-Checker: Uma ferramenta de análise de documentações escritas a partir de geradores de sites estáticos em markdown.

CAMPINA GRANDE - PB

2021

Fanny Batista Vieira

Docs-Checker: Uma ferramenta de análise de documentações escritas a partir de geradores de sites estáticos em markdown.

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

Orientador: Professor Dr. Matheus Gaudêncio.

CAMPINA GRANDE - PB

2021

V665d Vieira, Fanny Batista.

Docs-Checker: uma ferramenta de análise de documentações escritas a partir de geradores de sites estáticos em markdown. / Fanny Batista Vieira. – 2021.

12 f.

Orientador: Professor Dr. Matheus Gaudêncio do Rêgo.

Trabalho de Conclusão de Curso - Artigo (Curso de Bacharelado em Ciência da Computação) - Universidade Federal de Campina Grande; Centro de Engenharia Elétrica e Informática.

1. Docs-Checker . 2. Análise de documentação escrita. 3. Geradores de sites estáticos - markdown. 4. Estrutura de documentação de APIs - conformidade. 5. Static site generators - markdown. I. Rêgo, Matheus Gaudêncio do. II. Título.

CDU:004(045)

Elaboração da Ficha Catalográfica:

Johnny Rodrigues Barbosa
Bibliotecário-Documentalista
CRB-15/626

Fanny Batista Vieira

Docs-Checker: Uma ferramenta de análise de documentações escritas a partir de geradores de sites estáticos em markdown.

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

BANCA EXAMINADORA:

Professor Dr. Matheus Gaudêncio

Orientador – UASC/CEEI/UFCG

Professora Dr. Carlos Wilson

Examinador – UASC/CEEI/UFCG

Professor Tiago Lima Massoni

Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em 25 de Maio de 2021.

CAMPINA GRANDE - PB

ABSTRACT

Documentations are crucial artifacts in software development. The documentation explains the purpose of projects, unifies information and in general, avoids possible doubts from interested parties. For the documentation to be effective, its structure must be simple, friendly and direct, however, ensuring that these requirements are met is a difficult job, mainly because it is currently a process carried out manually. In this work, we will create a tool to analyze the conformity of API documentation structures with a specification, taking into account those written using static site generators (SSGs) in markdown. The system establishes rules that the documentation must meet and thus automatically assesses the quality of the documentation. The user is able to customize these rules according to his needs. The performance of the tool was evaluated and the data obtained revealed that the solution has a linear complexity behavior, taking approximately 300 milliseconds for small documents and 1 minute for large ones. In addition, when evaluating the tool in popular documentation such as React and Typescript, we obtained a success rate of more than 80%, proving that the solution is able to guarantee the conformity of the documentation structure.

DocsChecker: Uma ferramenta de análise de documentações escritas a partir de geradores de sites estáticos em markdown

Trabalho de Conclusão de Curso

Fanny Batista Vieira, Matheus Gaudencio do Rêgo
fanny.vieira@ccc.ufcg.edu.br, matheusgr@computacao.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba

RESUMO

Documentações são artefatos cruciais no desenvolvimento de software. A documentação explica o propósito de projetos, unifica informações e de modo geral, evita possíveis dúvidas das partes interessadas. Para que a documentação seja efetiva, sua estrutura deve ser simples, amigável e direta, no entanto, garantir que esses requisitos sejam atendidos é um trabalho difícil, principalmente, porque atualmente é um processo realizado manualmente. Neste trabalho, criaremos uma ferramenta para analisar a conformidade de estruturas de documentação de APIs com uma especificação, se atendo àquelas escritas por meio de static site generators (SSGs) em *markdown*. O sistema estabelece regras que a documentação deve atender e assim automaticamente avalia a qualidade da documentação. O usuário consegue customizar essas regras, de acordo com suas necessidades. O desempenho da ferramenta foi avaliado e os dados obtidos revelaram que a solução possui um comportamento de complexidade linear, levando aproximadamente 300 milissegundos para documentações de pequeno porte e 1 minuto para aquelas de grande porte. Além disso, ao avaliar a ferramenta em documentações populares como é o caso de React e Typescript obtivemos uma taxa de sucesso superior a 80%, comprovando que a solução consegue garantir a conformidade da estrutura de documentações.

PALAVRAS-CHAVE

Documentação, Markdown, Static Site Generators.

1. INTRODUÇÃO

O desenvolvimento de websites segue, de forma geral, um processo bem definido. Inicia com a criação de um HTML com o conteúdo, o uso de CSS, para determinar a aparência, e, por fim, o uso do JavaScript, para controlar as interações do usuário. À medida que a indústria de desenvolvimento web foi evoluindo, encontrou-se novas maneiras de tornar o processo de criação de websites mais eficiente, e assim, surgiram os Static Site Generators (SSGs - geradores de websites estáticos).

A partir de uma fonte de dados fornecida por desenvolvedores e de layouts *built-in*, os SSGs conseguem construir páginas web que são entregues aos visitantes de um site. Eles são usados para acelerar a construção de um tipo particular de websites, os websites estáticos. Isto é, websites em que as páginas permanecem com a mesma aparência para todos os visitantes. Assim, um uso comum para os SSGs é a construção de documentações, já que seu conteúdo não é personalizado para cada usuário.

Embora os SSGs auxiliem na disponibilização do conteúdo das documentações, atualmente não existem ferramentas que facilitem a manutenção da estruturação das páginas, o que dificulta a manutenção do conteúdo já que é necessário avaliar manualmente se a estrutura esperada está sendo seguida e torna o conteúdo menos preditivo e compreensivo para os usuários, uma vez que as páginas não seguem o mesmo formato. Buscando solucionar essas limitações, foi desenvolvida a ferramenta Docs-Checker, que analisa a estrutura das documentações por meio de uma especificação e disponibiliza um conjunto de regras que ajudam a garantir que a estrutura da página esteja em conformidade com a especificação informada. Por exemplo, uma estrutura comum a documentações que se referem a APIs, é a declaração de seus métodos, a apresentação de um código de referência, bem como uma descrição de seu propósito. Essa estrutura precisa ser assim especificada pelo desenvolvedor.

Para avaliação da qualidade do uso da ferramenta, foi realizado um estudo de caso do sistema em documentações populares, sendo elas a documentação da biblioteca React e TypeScript. A ferramenta foi executada nas páginas de APIs da documentação dessas bibliotecas, após a definição de uma especificação para as mesmas, constatamos que a solução consegue garantir a conformidade de estruturas de documentações, dado que nesse estudo de caso obtivemos em uma taxa de sucesso superior a 80%. Também analisamos o desempenho do Docs-Checker a fim de avaliar o comportamento da ferramenta em documentações de pequeno, médio e grande porte, e obtivemos resultados satisfatórios visto que nos experimentos realizados a solução apresentou uma relação linear entre o tamanho da documentação e o tempo de processamento.

Na Seção 2 descrevemos o problema e a solução adotada. Na Seção 3, apresentamos a arquitetura do sistema. Na seção 4, discutimos sobre a experiência de desenvolvimento e os principais desafios na construção da solução para na Seção 5, apresentarmos os resultados obtidos na avaliação da ferramenta em algumas documentações, seguida da seção de experimentos que avalia o desempenho da solução. Concluímos o artigo com uma discussão mais abrangente sobre as limitações da solução e os possíveis trabalhos futuros.

2. PROBLEMA E SOLUÇÃO

Static Site Generators

Os *SSGs* são ferramentas usadas para construção de websites, as quais a partir do conteúdo de fontes de dados, por exemplo, arquivos txt, markdown e outros, aplicam templates e geram a estrutura de uma página web pronta para ser entregue aos usuários. Os *SSGs* possuem um sistema de modelos (templates) essenciais para possibilitar a estrutura do site ser dinâmica, pois, a partir dele, é possível gerar a estrutura HTML básica do site, como cabeçalho, menu e rodapé, e definir o conteúdo exibido na página, que é obtido através da fonte de dados configurada.

Um caso de uso comum quando se trata de SSGs é a construção de documentações. Devido à estrutura de uma documentação permanecer fixa, isto é, possuir a mesma aparência para todos os usuários, os SSGs se mostram como mecanismos eficazes nesse cenário, tornando o processo de desenvolvimento rápido, fácil e de simples implantação.

Limitações

Embora os SSGs sejam usados por desenvolvedores para acelerar o desenvolvimento da infraestrutura de documentações, ainda carece de ferramentas que assegurem que a estrutura da página seja escrita de forma consistente. A falta de consistência em uma documentação dificulta a navegação e pesquisa em sua estrutura, aumentando o esforço dedicado pelos usuários para encontrar o que desejam, além de consumir mais tempo daqueles que fazem a manutenção da documentação, já que precisam manualmente garantir que a página segue um formato definido. Por mais que alguns SSGs disponham de plugins [25] que analisam as páginas de documentações, esses se limitam ao conteúdo, como por exemplo, avaliar se a lista de links presentes na página está disponível.

Soluções conhecidas

Uma abordagem comum para resolver o problema é a utilização de documentações auto-geradas, já que assim é possível garantir a conformidade na sua estrutura. Tal solução é apropriada quando a estrutura definida é simples, dado que as ferramentas que geram as documentações definem seus próprios formatos, no entanto, tornam um desafio a implementação de customizações, ocasionando a preferência à documentações não-autogeradas em cenários em que a documentação é complexa, por exemplo, quando envolve seções aninhadas, com diferentes conteúdos dependendo do nível de cada seção.

No caso de documentações não auto-geradas, algumas empresas como Gitlab[13] e Google[15], têm criado guias de boas práticas para construção de documentações, aprimorando a realização desse procedimento, no entanto, como os guias se restringem a recomendações e não obrigam nenhuma de suas regras na construção da documentação, ainda é necessária uma avaliação manual.

Docs-checker

A ferramenta Docs-checker permite validar se páginas de documentação de APIs seguem a especificação de uma estrutura definida. O usuário consegue expressar qual estrutura deve ser seguida usando um arquivo de configuração com uma descrição em JSON, conforme mostra a Figura 3. Através dele é possível definir quais seções a página contém, como é a estrutura de cada uma delas, além de quais elementos são opcionais e quais são obrigatórios. A partir da Figura 3, é possível observar que nas seções de nível 2 (h2) é esperado um texto, provavelmente a descrição da API, próximo a um código, que possivelmente se refere a um exemplo prático de uso da API.

Embora o exemplo utilizado neste trabalho seja focado em APIs, a ferramenta foi desenhada para que fosse flexível em outros contextos, comportamento que pode ser alcançado alterando a descrição na configuração de modo a seguir o padrão da estrutura desejada. Além da validação das estruturas, o sistema também verifica o tempo de leitura de cada seção da página, já que é difícil gerenciar e até mesmo processar a informação contida em seções com muito conteúdo. Após especificar como a página deve ser estruturada, o usuário consegue executar o sistema usando uma *command line interface (*CLI) e obterá um resultado similar ao que aparece na Figura 2.

3. ARQUITETURA DA SOLUÇÃO

O Docs-checker segue uma estrutura similar àquelas presentes em ferramentas de análise estática. A solução utiliza a avaliação de um analisador estático em arquivos *markdown*, para garantir a conformidade da estrutura fornecida pelo usuário com a estrutura retornada durante essa avaliação. Para facilitar essa verificação foi implementada uma *engine* de execução de regras, que define como as regras devem ser executadas baseando-se nas configurações do usuário.

Componentes do Sistema

Inicialmente o Docs-checker obtém a lista de arquivos *markdown* de uma documentação através de uma CLI, em seguida, carrega as verificações que o sistema deve realizar *(regras)* e as especificações da estrutura esperada e as repassa para a API da *engine* de execução das regras, que obtém o conteúdo gerado por um *parser markdown* e executa as regras conforme as configurações fornecidas, por fim, retorna o resultado da avaliação das regras, indicando se as páginas seguem a estrutura fornecida. A Figura 1 demonstra esse fluxo, onde o Moenda é a *engine* de referência de execução de regras, e o Docs-checker o responsável por estruturar as regras e exibir o resultado de testes em documentos.

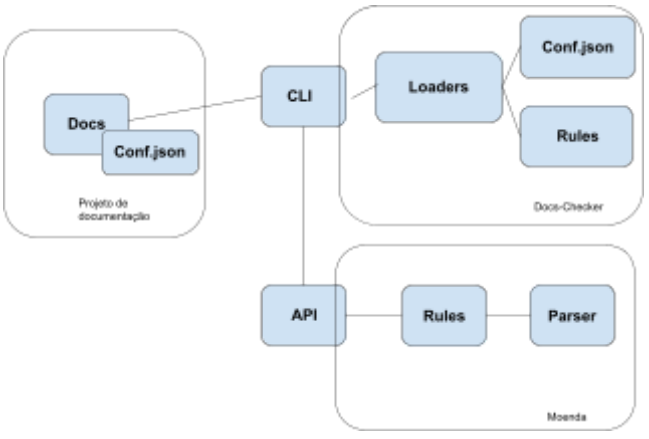

Figura 1 - Fluxo do Sistema

Command Line Interface

O ponto de entrada da aplicação é uma CLI**,** responsável por obter a lista de arquivos a serem analisados,

sendo especificada por meio de um padrão *glob*[14] ou através do nome dos arquivos, e retornar o resultado da avaliação das regras no console do usuário. Como pode ser visto na Figura 2. Os *errors* representam regras que devem ser seguidas obrigatoriamente, enquanto *warnings* representam regras que são opcionais.

```
❯ docs-checker run tests/react/docs/reference-*.md

   37  errors
   0 warnings
```

Figura 2 - *Output* da ferramenta.

Loaders

Os *loaders* são responsáveis por gerenciar o ambiente necessário para execução das regras do sistema**,** logo após receberem os arquivos a serem analisados, usando a CLI. Eles localizam as regras do sistema, que são aquelas definidas por padrão e as customizadas (construídas pelo usuário), assim como um arquivo de configuração que indica como as regras devem ser avaliadas. Caso o *loader* não consiga carregar as configurações a partir do diretório de arquivos lidos através da CLI, ele utiliza um algoritmo em cascata examinando os diretórios pais do atual até a raiz do projeto, retornando uma exceção em caso de falha.

Configuração Do Sistema

Para conseguir executar as regras é necessário fornecer um arquivo de configuração ***config.json***, em que é possível habilitar e desabilitar regras, fornecer objetos de customização e informações sobre os diretórios das regras. Todas as regras são habilitadas por padrão no sistema. A Figura 3 demonstra a configuração usada para especificar as páginas de documentação da API do React[24], uma biblioteca usada para construção de interfaces de usuário, criada pelo Facebook.

Após ter definido o arquivo de configuração e obtido o executável do sistema, o usuário conseguirá ter acesso ao relatório da ferramenta usando o comando **docs-checker run <glob | files.md>** no diretório da documentação.

Parser

O *parser* é responsável por analisar o conteúdo das páginas *markdown*, e retornar uma representação em árvore dos elementos[1] que cada seção da página possui. Para auxiliar essa análise foi realizado um estudo comparativo entre bibliotecas de parser *markdown*, e aquela que se destacou foi a ***markdown-it*** [20] devido sua popularidade e ferramental oferecido pela comunidade.

O *parser* analisa os elementos da página linha a linha, o que torna cada elemento independente na estrutura retornada e não descreve a relação hierárquica entre os objetos, isso porque não é possível agrupar elementos por seção ao usar *markdown,* dado que não existe nenhum símbolo que delimite o início da seção e seu fim, o que tornou esse, um problema comum entre todas as bibliotecas avaliadas. Por exemplo, se uma seção tiver uma subseção, a estrutura retornada pelos *parsers* não retorna este aninhamento, outro problema é que não é possível ter acesso aos elementos que uma seção contém.

```
{
  "comment": "Rules for react website",
  "default": false,
  "rules": {
    "customRules": null,
    "require-structure": {
      "structure": {
        "h1": {
          "p": "required",
          "h2": "optional"
        },
        "h2": [
          {
            "p": "required"
          },
          {
            "h3": "optional"
          }
        ],
        "h3": [
          {
            "p": "required",
            "ul": "required"
          },
          {
            "p": "required",
            "code": "required",
            "h4": "optional"
          },
          {
            "code": "required",
            "p": "required",
            "h4": "optional"
          }
        ],
        "h4": {
          "p": "required",
          "code": "optional"
        }
      }
    }
  }
}
```

Figura 3 - *React* config.json

A descrição hierárquica dos elementos é crucial para o desenvolvimento do sistema, tendo em vista que as seções de uma página podem ser caracterizadas como um componente fundamental na compreensão da estrutura de uma documentação. Devido a essa limitação, um *parser* customizado foi construído, responsável por fornecer essa capacidade adicional para as regras e, principalmente, facilitar a análise da estrutura de cada página.

O *parser* customizado obtém a lista de elementos (*tokens*) retornada pelo ***markdown-it***, e os manipula de forma que seja possível ter acesso ao conteúdo de cada seção. A lógica principal do *parser* utiliza a ideia de que seções com nível maior serão filhas de seções com nível menor. Para construção do algoritmo do parser customizado decidiu-se fazer uso de uma abordagem similar a estratégia de parseamento e construção de um DOM[6], pois possibilita acessar toda a árvore de elementos da página, em contraposição à estratégia de SAX[27], que permite apenas a análise de trechos do documento por vez, já que a ferramenta requer a avaliação da estrutura da página como um todo.

Regras

As regras definem a forma que a estrutura do documento será avaliada. Essas regras avaliam a presença e a estrutura de certos padrões nos elementos das páginas da documentação. A definição de como as regras são avaliadas é próxima de uma linguagem regular [19]. Assim, as especificações

definidas pelo usuário são traduzidas em expressões regulares e interpretadas perante os elementos da DOM.

Atualmente o sistema possui duas regras. A primeira avalia a estrutura de documentação de APIs, já que durante a fase de estudo essa demonstrou-se a estrutura mais frequente, embora ainda seja possível especificar outros tipos, alterando algumas propriedades no arquivo de configuração. A segunda avalia o tempo de leitura de cada seção, ademais o usuário consegue construir regras próprias seguindo a API do Docs-checker.

Moenda

Moenda é a *engine* responsável por uma das partes centrais da ferramenta, a execução das regras e a definição de quais informações serão transmitidas para as mesmas. A Moenda possibilita o processamento e análise de um conjunto de arquivos e retorna um relatório a partir de propriedades desses arquivos.

Ela opera de duas formas, através de uma CLI e por meio de uma API, e necessita de alguns parâmetros para o seu funcionamento adequado, são eles: o conjunto de regras do usuário, que nada mais são do que trechos de código seguindo uma especificação em comum para assegurar a compatibilidade; a configuração das regras; o conjunto de arquivos a ser analisado; o tipo desses arquivos, responsável por definir qual *parser* será usado no processamento das regras; e caso o usuário tenha interesse em fornecer propriedades adicionais às regras, é possível definir um processador de tokens customizado que possibilita a manipulação dos *tokens* retornados pelo *parser*.

A integração com a *engine* foi feita a partir de sua API, o Docs-checker repassa a lista de arquivos, bem como as regras com suas configurações, o tipo de arquivos que serão processados, nesse caso, *markdown*, e o parser customizado, o trecho de código responsável por essa integração pode ser visto na Figura 4. A Moenda obtém essas informações, injeta as dependências necessárias em cada regra e as executa nos arquivos indicados, retornando por fim os resultados do processamento que em seguida são repassados para a CLI do sistema.

```
function run(options) {
  const moenda = new Moenda({
    parser: 'md',
    rules: options.customRules,
    files: options.files,
    processor: require(path.resolve(__dirname, './parser')).createContext,
    rulesConfig: options.config,
  });

  moenda.runRules();
  console.log(formatter(moenda.getResults()));
  process.exit();
}
```

Figura 4 - Integração da moenda ao docs-checker

Tecnologias Utilizadas

O projeto foi desenvolvido utilizando Javascript, a escolha se deu porque os geradores de sites estáticos analisados utilizam essa linguagem, o que facilitou a adoção, integração e o desenvolvimento do sistema, além disso, Javascript é uma linguagem flexível, popular e simples, propiciando rapidez na busca e implementação de soluções mesmo quando em diferentes contextos de problemas. Como citado anteriormente, também foi feito o uso de uma biblioteca para facilitar a avaliação da estrutura das páginas *markdown*, a ***markdown-it*** [20], devido a sua configuração simples. Além disso, foram usadas bibliotecas para auxiliar no desenvolvimento, como é o caso de formatadores de código e transpiladores para ter acesso a funcionalidades mais recentes de JavaScript.

4. EXPERIÊNCIA DE DESENVOLVIMENTO

Principais desafios

O *parser* provê o acesso aos elementos que uma página *markdown* possui, no entanto, diferentemente da sintaxe de um código em que é possível delimitar o início e fim de um bloco através de chaves, no *markdown* isso não é feito, cada elemento é avaliado linha a linha, criando a necessidade de adaptar o resultado retornado pelo *parser* para ser possível ter um agrupamento dos elementos por seção.

Esse comportamento foi crucial para construir a regra que avalia a estrutura de uma API, já que necessita dos elementos presentes em cada seção para definir a forma da estrutura. Sem esse agrupamento, esta tarefa se tornaria complicada.

Devido a tais problemáticas, decidiu-se construir um algoritmo baseado na ideia de um avaliador de expressões[9], presente em compiladores durante a fase de avaliação de *tokens*. O algoritmo usa uma pilha para controlar a relação de hierarquia entre as seções e o nível de cada uma delas para controlar a sua precedência. O pseudocódigo pode ser visto no Algoritmo 1.

Algorithm 1 Custom Parser

```
stack ← [root]
for all tokens do
    if token is HEADING then
        while precedence(token) <= precedence(stack) and stack.length
do
            stack.pop()
        end while
        stack.top.children.push(token)
        stack.push(token)
    else
        stack.top.children.push(token)
    end if
end for
```

Como uma forma de agilizar e facilitar o desenvolvimento das regras do sistema, decidiu-se construir uma *engine*, a Moenda, cujo funcionamento foi descrito na seção 3.7. Isso porque ao avaliar as soluções estabelecidas na comunidade que permitiam a construção de regras sobre um conjunto de arquivos, percebeu-se que elas possuíam diversas restrições, por exemplo, não permitiam a inclusão de informações adicionais às regras, além daquelas que as bibliotecas definiam, ou customizar a formatação do resultado das regras, o que limitaria o desenvolvimento das funcionalidades do sistema. Além disso, ferramentas que analisam arquivos por meio de regras se mostram bastante úteis em tarefas como análise de métricas e erros de estilo. Tendo isso em mente, a *engine* foi construída, sendo essa construção feita de forma conjunta ao orientador e um dos alunos do curso, em um projeto de mentoria chamado *Andromedev*.

Processo

O processo de desenvolvimento consistiu em ciclos semanais de entrega, com duração de 1 a 2 semanas, dependendo da disponibilidade dos envolvidos. O orientador Matheus Gaudencio, atuou direcionando a priorização das atividades,

fornecendo *feedbacks* sobre as funcionalidades do sistema e auxiliando no desenho da arquitetura do sistema.

5. RESULTADOS

Após a finalização da fase de desenvolvimento, deu-se início a fase de avaliação da ferramenta, nesta etapa foram selecionadas algumas documentações para um estudo de caso, a fim de assegurar que os resultados retornados pelo sistema eram coerentes com aqueles esperados pelo usuário. Entre as escolhas de documentações avaliadas, destaca-se a da React[24] e a da Typescript[30], que serão descritas nesta seção.

Cenário 1 - Documentação da React

Após obter o projeto da documentação, clonando o repositório do Github[24] através do commit de hash **364c6613858f9badc9d4c420858ba615423775dc** no dia 27 de Fevereiro de 2021, foi criado o arquivo de configurações descrevendo a estrutura de documentação de APIs que as páginas do React seguem, a mesma exibida na Figura 3. A estrutura contém seções que definem uma tabela de conteúdos através de uma lista de parágrafos e links. Assim como a descrição das APIs através de parágrafos e códigos de referência. As páginas de APIs do React são divididas em dois tipos: aquelas que possuem o prefixo *reference* e aquelas com o sufixo *reference*, que atualmente se limitam a um arquivo, **hooks-reference.md**, assim, o docs-checker foi executado usando os seguintes comandos:

docs-checker run docs/reference-*.md &
docs-checker run docs/hooks-reference

Para melhor representação dos resultados coletados pela ferramenta, foi criada uma tabela com a quantidade total de seções presentes em cada página e a quantidade de seções sem erro em cada uma delas, em seguida foi calculado o percentual de sucesso, como mostra o Quadro 1. É possível observar que a maioria dos percentuais analisados são maiores ou iguais a 70%, retornando uma média de 83%, o que representa uma grande consonância da estrutura de documentação proposta com a estrutura original, além de confiança na implementação do sistema, dado que os resultados obtidos foram positivos em uma documentação tão popular e detalhada como a do React. Também é possível observar que em alguns casos a taxa de sucesso chegou a 100%, confirmando a efetividade da estrutura, e em outros a 0%, o que pode ser analisado individualmente.

As páginas **reference-javascript-requirements-environment**, **reference-pure-render-mixin** e **reference-dom-elements** apresentam resultados destoantes das demais. No caso das duas primeiras, notou-se que elas continham apenas uma seção conforme mostra a Figura 5, tipicamente as páginas dessa documentação são mais detalhadas, apresentando mais seções e conteúdo, configurando um cenário particular dado que só ocorre duas vezes. No caso da página **reference-dom-elements,** todos os erros foram relacionados a seções que não possuíam código exibindo como a API poderia ser usada, o que pode ser observado na Figura 6, embora esse erro ocorra em outras páginas a sua frequência é baixa. Portanto, concluiu-se que a ferramenta teve êxito na detecção de erros, tendo em vista que nos cenários de erro a estrutura proposta deveria ter sido seguida, já que em outros exemplos similares a estrutura aplicada foi a proposta.

Quadro 1 - Resultados da avaliação na documentação da React

Título da página	Quantidade de seções	Quantidade de seções sem erro	Taxa de sucesso
DOM Elements	13	5	0,38
Events	27	27	1,00
Glossary	19	17	0,89
Javascript requirements environment	1	0	0,00
Profiler	3	3	1,00
Pure render mixin	1	0	0,00
React component	41	29	0,70
React DOM Server	7	6	0,85
React DOM	9	8	0,87
React	33	29	0,87
Test Renderer	26	22	0,84
Hooks Reference	33	32	0,96
Total	213	178	**0.83**

Figura 5 - *reference-javascript-enviroment:* Página que inclui apenas uma seção.

Figura 6 - *reference-dom-elements:* Página em que as seções não possuem código

Cenário 2 - Documentação da TypeScript

Para a documentação do Typescript, o projeto também foi obtido clonando o repositório do Github[30], através do commit de hash **5c4d55a9bdd4e8730eebdc9a436620b480572423** no dia 15 de Março de 2021, e definindo a estrutura conforme mostra a Figura 7. As páginas avaliadas foram aquelas que se encontram no diretório **reference** da documentação, já que todos os arquivos se referem a API da biblioteca.

Em seguida, o docs-checker foi executado usando o comando **docs-checker run reference/*.md.** Os resultados coletados podem ser visualizados no Quadro 2.

A partir dos resultados, podemos observar uma taxa de sucesso elevada, sendo ela de 95% superando o valor observado para a documentação do React, além disso, nenhuma das páginas apresenta um valor inferior a 60%, isso ocorre porque mais cenários particulares foram especificados, já que a estrutura do Typescript não é tão genérica, aumentando assim a cobertura de estruturas válidas avaliada pela ferramenta na documentação. Por exemplo, examinando o arquivo de configurações da Figura 7, vemos que as seções de nível 2 (h2) podem ter 3 tipos diferentes de estrutura.

No caso da seção com menor percentual de sucesso, isto é, **Iterators and Generators**, notou-se que as seções de nível 3 não apresentavam nenhum conteúdo, o que demonstrou ser um cenário inválido já que tipicamente as seções possuem pelo menos um parágrafo introduzindo o conceito que será discutido na seção. Outro erro apontado pela ferramenta se referia a seções que continham apenas um parágrafo, que também configura um caso inválido, dado que geralmente as seções apresentam códigos para auxiliar a visualização de como a API pode ser usada. Por fim, como esses padrões não apresentaram uma frequência de repetição alta, ocorrendo apenas em 3 páginas, considerou-se que não fazia sentido incluir uma regra para as mesmas e que a estrutura proposta no arquivo de configurações deveria ser seguida, um trecho da estrutura da página com erro pode ser vista na Figura 8.

Figura 7 - Typescript *config.json*

Quadro 2 - Resultados da avaliação na documentação da Typescript

Título da página	Quantidade de seções	Quantidade de seções sem erro	Taxa de sucesso
DOM Elements	27	27	1,00
Declaration Merging	9	8	0,88
Decorators	11	10	0,90
Enums	11	11	1,00
Iterators and Generators	8	5	0,62
JSX	13	13	1,00
Mixins	8	8	1,00
Module Resolution	20	14	0,70
Modules	111	102	0,91
Namespaces and Modules	6	6	1,00
Namespaces	21	19	0,90
Symbols	12	12	1,00
Triple-Slash-Directives	13	10	0,76
Type Compatibility	13	12	0,92
Type Inference	2	2	1,00
Utility Types	55	51	0,92
Variable Declarations	17	17	1,00
Total	344	327	**0.95**

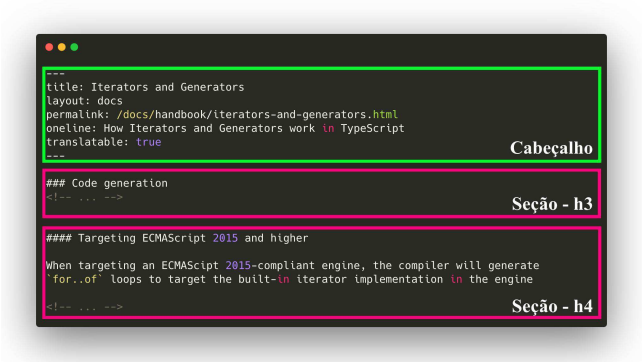

Figura 8 - *Iterators and Generators*: Página em que as seções não possuem conteúdo e código

Resultados Gerais

Portanto, verificamos que o sistema se comporta adequadamente tanto para documentações que têm estruturas uniformes, como foi o caso da React, quanto para aquelas de estruturas diversificadas, como ocorreu na Typescript. Isso porque é possível expressar ambos os casos na linguagem definida pelo arquivo de configurações. Além disso, para aquelas páginas que apresentaram erros, observou-se que de fato a estrutura proposta deveria ser seguida, dado que essa estrutura aparecia com frequência alta nas páginas e era usada em seções similares àquelas que a ferramenta apontou erros, demonstrando a validade da especificação.

Apesar da diferença nas avaliações observadas, notou-se que a alta taxa de sucesso do Typescript se devia principalmente à especificação que define que é possível possuir apenas um parágrafo nas seções de nível 2, isso porque, ao testar a ferramenta sem essa estrutura, o resultado diminuiu para 70%. Embora essa estrutura seja genérica, uma vez que só requer a presença de um parágrafo como conteúdo e tenha uma frequência alta de repetição nas páginas, não ficou claro em quais cenários o seu uso seria adequado, sendo necessário avaliar com a comunidade se a sua inclusão como estrutura é benéfica para a documentação.

6. EXPERIMENTOS

Com o intuito de avaliar a performance do sistema, foram realizados alguns experimentos baseando-se no tempo de resposta da ferramenta de acordo com a quantidade de páginas de uma documentação. Para tal, dentre as documentações avaliadas, foi selecionada a documentação da TypeScript[30], devido à documentação possuir uma maior quantidade de páginas se comparada com a da React. Enquanto TypeScript possui 18 páginas no que se refere a APIs, React possui 13. Além disso, suas páginas possuem estruturas mais complexas, com seções aninhadas e conteúdo extenso, por exemplo, algumas delas chegam a ultrapassar 1000 linhas.

Para condução do experimento, foi construído um *script* em *Nodejs,* que executa o comando da CLI do Docs-checker para disparar a execução da ferramenta, e mede o tempo de execução da ferramenta através da API nativa *perf_hooks[23]* do Node. Esse *script* foi executado em amostras contendo 10, 100, 1k, 10k e 100k páginas, como a documentação usada possuía uma quantidade de páginas inferior aos cenários acima de 10 arquivos, as 10 páginas iniciais selecionadas foram replicadas, em que 80% delas não possuíam erro e as 20% restantes possuíam. Para evitar possíveis interferências, como o tempo de aquecimento do compilador e a ocupação da *main thread* do javascript com outras tarefas, cada amostra foi executada 40 vezes, em que as primeiras 10 execuções foram ignoradas.

A partir dos gráficos apresentados nas Figuras 9 e 10, podemos observar que o sistema se comporta adequadamente tanto para documentações de pequeno porte como para aquelas de grande porte, já que é possível notar que o tempo necessário para execução é relativamente rápido. As execuções mais rápidas foram as três primeiras que não ultrapassaram 1 segundo e seguem o comportamento do primeiro histograma, sendo elas as amostras de 10, 100 e 1000 páginas que levaram 146,41, 300,05 e 999,94 milissegundos em média respectivamente. Em contrapartida, a amostra com maior disparidade foi aquela com 100k páginas pois, enquanto a execução da amostra anterior, de 10k páginas levou 7.138,21 milissegundos, esta durou 9.1310,28, o que ainda parece ser um tempo adequado, considerando a quantidade de páginas. A Figura 11 demonstra a distribuição geral dos dados.

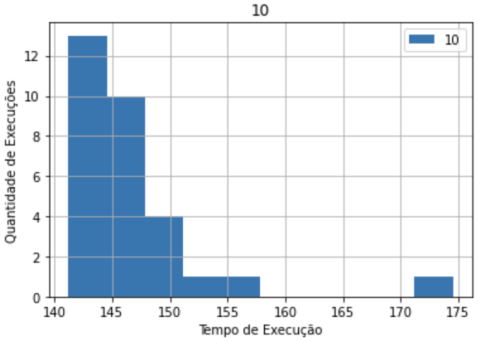

Figura 9 - Histograma de quantidade de execuções por tempo de resposta relativos a documentações contendo 10 arquivos

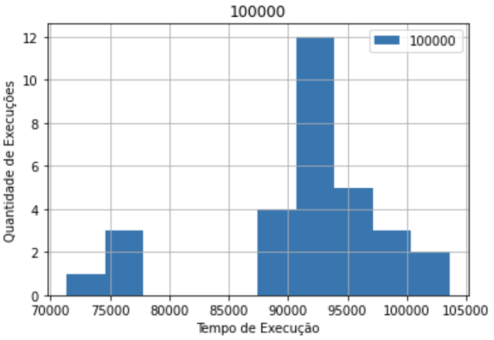

Figura 10 - Histograma de quantidade de execuções por tempo de resposta relativos a documentações contendo 100K arquivos

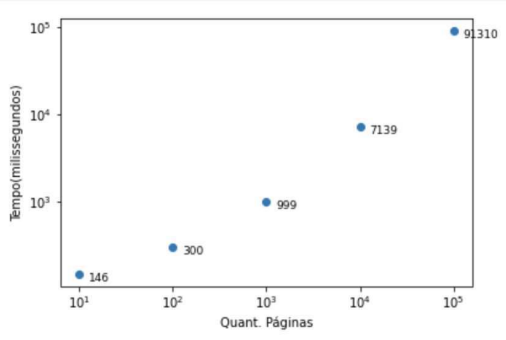

Figura 11 - Scatter plot da quantidade de páginas por tempo de resposta em milissegundos

Considerando os cenários testados verificamos então um padrão de comportamento linear, tendo em vista que o tempo que o sistema leva para completar sua execução, segue uma proporção muito similar à variação da entrada, enquanto a entrada crescia 10 vezes mais a cada cenário observado, o tempo de execução não ultrapassava significativamente essa proporção, a Figura 12 demonstra esse comportamento através de um gráfico de regressão linear entre a quantidade de páginas e o tempo de execução, usando uma escala logarítmica para facilitar a compreensão.

Além disso, ao construir uma regressão linear tendo a quantidade de páginas como variável dependente e o tempo de execução como variável independente, obtivemos um coeficiente de determinação de valor **0,948,** o que indica que o modelo gerado é adequado para explicar a variação do tempo de execução através da quantidade de páginas, em adição a isso, o valor retornado pela média de erros ao quadrado foi relativamente baixo, sendo **0,280**, corroborando a hipótese de que as variáveis se relacionam de forma linear.

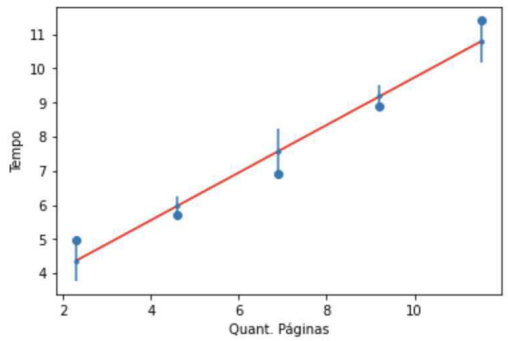

Figura 12 - Comportamento da performance do sistema

7. CONCLUSÃO

Concluímos portanto que a ferramenta proposta é adequada para validar a conformidade de estruturas de documentações que seguem uma especificação, tendo em vista que nos cenários avaliados a solução apresentou uma taxa de sucesso elevada, superior a 80%, além de ter um desempenho satisfatório em documentações de pequeno, médio e grande porte, já que apresenta uma relação linear entre o tamanho da documentação e o seu tempo de resposta. Abaixo são descritas as limitações encontradas no desenvolvimento da ferramenta, possíveis melhorias em trabalhos futuros e sugestões de uso da solução.

Limitações

Atualmente as regras não são avaliadas de forma estrita, logo, uma página será considerada válida se o padrão definido combinar com as estruturas das seções da página, não considerando se as seções possuem elementos adicionais além dos especificados. Por exemplo, se a especificação de uma seção requer que a estrutura contenha parágrafos e códigos, e nessa mesma seção houver a presença de outros elementos, como uma imagem, a ferramenta não apontará erros.

Ao tentar implementar a validação de forma estrita, ou seja, não permitindo elementos adicionais além dos válidos, notou-se que o regex se tornava complexo tanto de construir quanto de validar, também percebeu-se que a forma estrita só era necessária em casos muito específicos. Como o sistema apresentou uma boa cobertura na sua forma simples, decidiu-se não investir tempo buscando uma alternativa.

Trabalhos Futuros

Como o objetivo do trabalho é provar um conceito, existem algumas melhorias a serem feitas na solução a fim de potencializar a solução em projetos de uso real. Algumas delas são: detecção automática da estrutura de documentação, através de padrões que a mesma apresenta que podem ser coletados durante a análise dos tokens; implementação de regras mais complexas que envolvem por exemplo, a análise do conteúdo com processamento de linguagem natural de modo a detectar melhorias na escrita do conteúdo; eficiência na interação entre os *tokens* já que atualmente para ter acesso a eles é necessário iterar sobre toda a árvore de elementos; e a inclusão de outros tipos de conteúdo, como HTML. As melhorias apresentadas possuem solução, dependendo apenas de tempo para serem implementadas. Através da ferramenta, também concluímos ser possível fazer uso dela em outros cenários, não se limitando apenas a APIs, o que foi alcançado com algumas alterações nas configurações repassadas à ferramenta.

Assim, recomendamos a adoção da solução tanto para documentações de pequeno quanto para as de médio porte. Projetos *open source e* documentações que utilizam *markdown* como fonte de dados nas documentações podem ser beneficiadas pelo uso dessa ferramenta, já que essa seria um mecanismo para garantir que novos contribuidores seguirão as convenções de sua estrutura, além disso, caso o sistema não ofereça suporte a alguma verificação, o usuário também consegue implementar suas verificações próprias.

8. AGRADECIMENTOS

Agradeço primeiramente a Deus por tornar esse trabalho possível. Agradeço a minha família e ao meu querido Alex Junior, pelo apoio incondicional. Aos amigos que reuni durante a minha trajetória na graduação, a comunidade OpenDevUFCG que muito contribuíram para o meu crescimento pessoal e profissional. A comunidade open source do Facebook chamada Docusaurus, que tanto me ensinaram sobre o assunto. E ao professor Matheus Gaudencio, por seu conhecimento, paciência e todas as preocupações e felicidades que compartilhamos ao longo do desenvolvimento deste trabalho.

9. REFERÊNCIAS

[1] Abstract Syntax Tree. Retrieved February 2021 from https://en.wikipedia.org/wiki/Abstract_syntax_tree.

[2] API doc survey result: Automating API Rest documentation. Retrieved February 2021 from https://idratherbewriting.com/2015/01/06/api-doc-survey-automating-rest-api-documentation/#isanyaspectofyourrestapidocumentationauto-generatede.g.customscriptsifsohow

[3] Compiladores: Análise Léxica Retrieved March 2021 from https://dcc.ufrj.br/~fabiom/comp20131/02AnaliseLexica.pdf.

[4] Compiladores, Aula 2. Retrieved March 2021 from http://www2.fct.unesp.br/docentes/dmec/olivete/compiladores/arquivos/Aula2.pdf.

[5] CSS Lint Developer Guide. Retrieved January 2021 from https://github.com/CSSLint/csslint/wiki/Working-with-Rules.

[6] Document Object Model Parsing. Retrieved April 2021 from https://en.wikipedia.org/wiki/Document_Object_Model.

[7] Docusaurus Showcase. Retrieved February 2021 from https://docusaurus.io/showcase.

[8] Drop links to your favorite docs. And what you liked about them? Retrieved January 2021 from https://twitter.com/dan_abramov/status/1280949739534630919.

[9] Endanger. Retrieved February 2021 from https://github.com/discord/endanger.

[10] Eslint Architecture. Retrieved January 2021 from https://eslint.org/docs/developer-guide/architecture.

[11] Expression Evaluation Algorithm. Retrieved March 2021 from https://www.geeksforgeeks.org/expression-evaluation/.

[12] Gatsby, Use Cases: Technical Documentation. Retrieved February 2021 from https://www.gatsbyjs.com/use-cases/technical-documentation

[13] Gitlab Documentation Style Guide. Retrieved January 2021 from https://docs.gitlab.com/ee/development/documentation/styleguide/.

[14] Glob Pattern. Retrieved January 2021 from https://en.wikipedia.org/wiki/Glob_(programming).

[15] Google Tech Writing. Retrieved January 2021 from https://developers.google.com/tech-writing/overview.

[16] How Linters, Compilers & Other Cool Things Work, CascadiaJS. Retrieved December 2020 from https://www.youtube.com/watch?v=JMqZgUNkqgk.

[17] Introduction to Linear Regression. Retrieved April 2021 from https://observablehq.com/@kdhartmann/lr-project.

[18] Is structured authoring a good fit for publishing on a website? Retrieved February 2021 from https://idratherbewriting.com/2013/05/14/structured-authoring-versus-the-web/.

[19] Linguagem Regular. Retrieved March 2021 from https://pt.wikipedia.org/wiki/Linguagem_regular#:~:text=Na%20teoria%20da%20ci%C3%AAncia%20da,os%20elementos%20de%20um%20alfabeto.

[20] Markdown-it. Retrieved December 2021 from https://github.com/markdown-it/markdown-it.

[21] MarkdownLint. Retrieved December 2021 from https://github.com/DavidAnson/markdownlint.

[22] Mozilla Doc Linter. Retrieved January 2021 from https://github.com/mdn/doc-linter-webextension.

[23] NodeJS: Performance Measurement APIs. Retrieved April 2021 from https://nodejs.org/api/perf_hooks.html.

[24] ReactJS Website. Retrieved February 2021 from https://github.com/reactjs/reactjs.org.

[25] Remark Lint Rules. Retrieved March 2021 from https://github.com/remarkjs/remark-lint#rules.

[26] Rome Tools. Retrieved January 2021 from https://rome.tools/.

[27] SAX Parsing. Retrieved April 2021 from https://en.wikipedia.org/wiki/Simple_API_for_XML.

[28] TextLint. Retrieved December 2021 from https://textlint.github.io/.

[29] The super tiny compiler. Retrieved December 2020 from https://github.com/jamiebuilds/the-super-tiny-compiler.

[30] TypeScript Website. Retrieved March 2021 from https://github.com/microsoft/TypeScript-Website.

[31] What is a static site generator? Retrieved January 2021 from https://www.netlify.com/blog/2020/04/14/what-is-a-static-site-generator-and-3-ways-to-find-the-best-one/.

[32] Will structure and style make documentation processes less costly. Retrieved March 2021 from https://idratherbewriting.com/2014/01/26/will-structure-and-style-make-documentation-processes-less-costly/.