INSTITUTO FEDERAL
SANTA CATARINA

CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE SAÚDE E SERVIÇOS
CURSO SUPERIOR DE TECNOLOGIA EM GESTÃO DA TECNOLOGIA DA INFORMAÇÃO

JENNIFER CAMILA DOS REIS TAKAGI

ESPECIFICAÇÃO DE PARÂMETROS PARA O USO DA FERRAMENTA *ESLINT* EM PROJETOS *REACT JS* PARA MANUTENÇÃO DE CÓDIGO LIMPO

Florianópolis - SC
2021

Ficha de identificação da obra elaborada pelo autor.

Takagi, Jennifer Camila dos Reis
Espeficição de parâmetros para o uso da ferramenta ESLint em projetos React JS para manutenção de Código Limpo / Jennifer Camila dos Reis Takagi ; orientação de Herval Daminelli; coorientação de Felipe Malfatti. - Florianópolis, SC, 2021.
216 p.
Trabalho de Conclusão de Curso (TCC) - Instituto Federal de Santa Catarina, Câmpus Florianópolis. CST em Gestão da Tecnologia da Informação. Departamento Acadêmico de Saúde e Serviços.
Inclui Referências.

1. Código Limpo. 2. JavaScript. 3. React JS. 4. ESLint. 5. Padrões de Codificação. I. Daminelli, Herval. II. Malfatti, Felipe. III. Instituto Federal de Santa Catarina. Departamento Acadêmico de Saúde e Serviços. IV. Título.

INSTITUTO FEDERAL DE SANTA CATARINA
DEPARTAMENTO ACADÊMICO DE SAÚDE E SERVIÇOS
CURSO SUPERIOR DE TECNOLOGIA EM GESTÃO DA TECNOLOGIA DA INFORMAÇÃO

JENNIFER CAMILA DOS REIS TAKAGI

ESPECIFICAÇÃO DE PARÂMETROS PARA O USO DA FERRAMENTA *ESLINT* EM PROJETOS *REACT JS* PARA MANUTENÇÃO DE CÓDIGO LIMPO

Trabalho de Conclusão de Curso submetido ao Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina como parte dos requisitos para obtenção do título de Tecnólogo em Gestão da Tecnologia da Informação.

Professor Orientador:
Prof. Herval Daminelli, Esp.

Coorientador:
Felipe Malfatti, Esp.

FLORIANÓPOLIS - SC
MARÇO/2021

ESPECIFICAÇÃO DE PARÂMETROS PARA O USO DA FERRAMENTA *ESLINT* EM PROJETOS *REACT JS* PARA MANUTENÇÃO DE CÓDIGO LIMPO

JENNIFER CAMILA DOS REIS TAKAGI

Este trabalho foi julgado adequado para obtenção do Título de Tecnólogo em Gestão da Tecnologia da Informação e aprovado na sua forma final pela banca examinadora do Curso Superior de Tecnologia em Gestão da Tecnologia da Informação do Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina.

Florianópolis-SC, 29 de março de 2021.

Prof. Felipe Cantório Soares, MSc.
Coordenador do CST em Gestão da Tecnologia da Informação
Instituto Federal de Santa Catarina

Banca Examinadora:

Herval Daminelli

Prof. Herval Daminelli, Esp.
Orientador
Instituto Federal de Santa Catarina

Gonçalves

Prof. Juliano Lucas Gonçalves, MSc.
Instituto Federal de Santa Catarina

Prof. Andrino Fernandes, MSc.
Instituto Federal de Santa Catarina

Dedico à minha família, especialmente a minha irmã Carla, meu pai Carlos, minha noiva Andressa e a Chewie. Também em memória de minha avó Fumico e minha tia Sandra.

AGRADECIMENTOS

Agradeço a minha família por me apoiar e estar torcendo por mim em todas as minhas conquistas. Carla, obrigada por me dar todo o amor fraternal e me inspirar com seu enorme coração.

Agradeço ao meu pai por todos os sacrifícios e por me ensinar desde pequena o quanto deveria me esforçar para atingir meus sonhos.

Agradeço também a Andressa, por nunca desacreditar do meu sucesso e me incentivar a dar o meu máximo em todos os desafios da vida, além de trazer a Chewie a nossas vidas, provando que o cão é sim o melhor amigo do ser humano.

A minha avó e minha tia, que já não estão comigo, agradeço por terem me ensinado todos os princípios capazes de me tornar um bom ser humano. Dedico essa conquista a vocês - sempre estarão ao meu lado.

Agradeço, também, a instituição acadêmica e ao corpo docente por me proporcionarem uma trajetória repleta de conhecimento e inovações, em especial ao Professor Herval Daminelli, Cleverson Tabajara e Julio Ribas por me auxiliarem em todo o percurso de execução e entrega do Trabalho de Conclusão de Curso.

"Writing clean code is what you must do in order to call yourself a professional. There is no reasonable excuse for doing anything less than your best."

Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship

RESUMO

A adoção de padrões de código e boas práticas da programação envolve conceitos de *clean code*, e se torna necessária para que times de desenvolvimento tenham uma qualidade de entrega confiável e garanta que não há *bugs* e falhas escondidos em imensas linhas de código mal concebidas. O *ESLint* é uma ferramenta de verificação de código estático capaz de garantir o uso de padrões configurados nele. Almejando auxiliar a maior parte dos atuais e futuros desenvolvedores, delimitou-se o tema a projetos que utilizem *React JS*, após análise da pesquisa anual do StackOverflow (2020). O presente trabalho desenvolve uma investigação profunda sobre conceitos de código limpo aplicados a *JavaScript* e a projetos escritos com a biblioteca *React JS* a fim de especificar parâmetros para o uso da ferramenta *ESLint*. Trata-se de uma pesquisa aplicada com abordagem qualitativa, descritiva e exploratória segundo seus objetivos, e classifica-se em bibliográfica, de levantamento e pesquisa de campo quanto seus procedimentos técnicos quanto seus objetivos. Para o desenvolvimento de seus objetivos foram utilizados três procedimentos técnicos: bibliográfico, de levantamento e pesquisa de campo, onde validou-se por meio de um questionário *online* quais regras são utilizadas pela maioria dos profissionais da área de programação que atuam com *React JS*. Aplicaram-se as regras selecionadas na ferramenta *linter ESLint* e testou-se a solução em um projeto real envolvendo as tecnologias delimitadas.

Palavras-chave: Código Limpo; *JavaScript; React JS*; *ESLint*; Padrões de Codificação.

ABSTRACT

The adoption of code standards and good programming practices involves clean code concepts and it is necessary for development teams to have a reliable delivery quality and to guarantee that there are no bugs and flaws hidden in huge lines of poorly handled code. ESLint is a static code verification tool capable of guaranteeing the use of standards configured in it. Aiming to assist most of the current and future developers, the theme was limited to React JS, after analyzing the annual research of StackOverflow (2020). The present work develops a deep investigation about clean code concepts applied to JavaScript and to projects written with the React JS library in order to specify parameters for the use of the ESLint tool. It is an applied research with a qualitative, descriptive and exploratory approach according to its objectives, and is classified in bibliographic, survey and field research regarding its technical procedures as its objectives. For the development of this objective, three technical procedures were used: bibliographic, survey and field research, where it was validated through an online questionnaire which rules are used by most professionals in the programming area who work with React JS. The rules selected in the linter ESLint tool were applied and the solution was tested in a real project involving the delimited technologies.

Key-words: Clean Code; JavaScript; React JS; ESLint; Coding Pattern.

LISTA DE ILUSTRAÇÕES

FIGURA 1 - Resultado geral da questão sobre Linguagens de: Programação, *Script* e *Markup* na pesquisa *StackOverflow Survey 2020*……...29
FIGURA 2 - Resultado por profissionais desenvolvedores da questão sobre Linguagens de: Programação, *Script* e *Markup* na pesquisa *StackOverflow Survey 2020*……..…….30
FIGURA 3 - Resultado geral da questão sobre *Web Frameworks* na pesquisa *StackOverflow Survey 2020*…...…..…….……..30
FIGURA 4 - Resultado por profissionais desenvolvedores da questão sobre *Web Frameworks* na pesquisa *StackOverflow Survey 2020*…...…..……..…..31
FIGURA 5 - Exemplo de declarações de variáveis, escrito em *JavaScript*.……....42
FIGURA 6 - Exemplo de comentário, escrito em *JavaScript*...…...……...…..42
FIGURA 7 - Exemplo de entrada e saída de dados, escrito em *JavaScript*..……...43
FIGURA 8 - Janela no navegador com um *"input"* para que o usuário digite sua idade...…..……...…..……...43
FIGURA 9 - Modal de saída com mensagem indicando que a idade é maior que 18.....…..……...……...…..……...44
FIGURA 10 - Exemplo de função recebendo parâmetro, escrito em *JavaScript*…...44
FIGURA 11 - Exemplo de uso de operadores aritméticos, escrito em *JavaScript*……...……...……...……...……...……...……45
FIGURA 12 - Exemplo de uso de operadores incrementais, escrito em *JavaScript*……...……...……...……...……...……...……45
FIGURA 13 - Exemplo de uso de operadores relacionais, escrito em *JavaScript*……...……...……...……...……...……...……46
FIGURA 14 - Exemplo de uso de operadores lógicos, escrito em *JavaScript*……...……...……...……...……...……...……46
FIGURA 15 - Exemplo de estrutura de decisão, escrito em *JavaScript*……...47
FIGURA 16 - Exemplo de estrutura de repetição, escrito em *JavaScript*……....48
FIGURA 17 - Exemplo de armazenamento de dados no *JavaScript*……...53
FIGURA 18 - Exemplo de armazenamento de dados em *memory heap*……53
FIGURA 19 - Exemplo de *hoisting* em uma variável……...……....54

FIGURA 20 - Exemplo da diferença entre *double* e *triple equals*.....55
FIGURA 21 - Exemplo da de *high order function*, utilizando a função *"map"*.....55
FIGURA 22 - Representação de um documento *HTML*.....56
FIGURA 23 - Representação da árvore *DOM* de um *HTML*.....57
FIGURA 24 - Exemplo de uso de variável utilizando *"let"*.....58
FIGURA 25 - Exemplo de uso de variável utilizando *"const"* e *"array"*.....59
FIGURA 26 - Exemplo de uso de *template literal string*.....60
FIGURA 27 - Exemplo de uso de *default value* em função.....60
FIGURA 28 - Exemplo de uso de *arrow function*.....61
FIGURA 29 - Exemplo de uso de *destructuring* em "objetos".....62
FIGURA 30 - Exemplo de uso de *destructuring* em *"array"*.....63
FIGURA 31 - Exemplo de uso de classe.....64
FIGURA 32 - Exemplo de uso de *ES6 modules*.....65
FIGURA 33 - Exemplo de criação de um elemento *React* usando *"React.createElement"*.....69
FIGURA 34 - Exemplo de renderização de elementos usando *ReactDOM*.....69
FIGURA 35 - Exemplo de renderização usando *JSX*.....71
FIGURA 36 - Exemplo de criação de componente.....72
FIGURA 37 - Exemplo de renderização de *children* no *JSX*.....73
FIGURA 38 - Função exemplo: boa prática para nomeação de variáveis.....77
FIGURA 39 - Função exemplo: boa prática em manter informações fixas em variáveis.....78
FIGURA 40 - Função exemplo: boa prática de desacoplamento de responsabilidade por funções.....79
FIGURA 41 - Função exemplo: solução de códigos com muitas condicionais.....79
FIGURA 42 - Função exemplo: criação de função genérica para evitar códigos duplicados.....80
FIGURA 43 - Função exemplo: comentário contendo regra de negócio.....80
FIGURA 44 - Função exemplo: boa prática de documentação de código.....81
FIGURA 45 - Arquivo de configuração exemplo do *ESLint* (*.eslintrc.js*).....98
FIGURA 46 - Exemplo de arquivo de configuração do *ESLint* utilizando *ESLint Recommended*.....99

FIGURA 47 - Exemplo de arquivo de configuração do ESLint utilizando *Google JavaScript Style Guide*.....100
FIGURA 48 - Exemplo de arquivo de configuração do ESLint utilizando *AirBnB*.....100
FIGURA 49 - Exemplo de arquivo de configuração do ESLint utilizando *Standard*.....100
FIGURA 50 - Fluxograma contemplando as etapas da pesquisa.....105
GRÁFICO 1 - Atualmente você atua diretamente com projetos que utilizam o *framework React JS*?.....108
GRÁFICO 2 - Qual o seu cargo na empresa atualmente?.....109
GRÁFICO 3 - Com qual tipo de desenvolvimento você trabalha atualmente?.....110
GRÁFICO 4 - Há quanto tempo você atua na área de tecnologia?.....111
GRÁFICO 5 - Qual o tipo de empresa onde você trabalha?.....111
GRÁFICO 6 - Com que frequência você sente que perde tempo tentando compreender códigos de projetos na sua vida profissional?.....112
GRÁFICO 7 - Você acredita que seguir as boas práticas de programação impacta positivamente no seu trabalho?.....113
GRÁFICO 8 - Você acredita que definir Padrões de Código junto ao seu time impacta positivamente no seu trabalho?.....113
GRÁFICO 9 - [*ARRAY*] Selecione quais padrões e práticas foram definidos pela sua equipe.....114
FIGURA 51 - Código exemplo: uso de retorno dentro de *high order function*.....115
FIGURA 52 - Código exemplo: uso de *destructuring* para acessar posições do *"array"*.....116
GRÁFICO 10 - [*BLOCKS*] Selecione quais padrões e práticas foram definidos pela sua equipe.....117
FIGURA 53 - Código exemplo: uso de chaves em comandos multilinhas.....117
FIGURA 54 - Código exemplo: uso de *"else"* na mesma linha das chaves do bloco.....118
GRÁFICO 11 - [*CLASSES*] Selecione quais padrões e práticas foram definidos pela sua equipe.....118
FIGURA 55 - Código exemplo: uso da *keyword "this"* para acesso de propriedades

da classe dentro dela.....119
FIGURA 56 - Código exemplo: evitar duplicação de nomes de métodos dentro da mesma classe.....119
GRÁFICO 12 - [*FUNCTIONS*] Selecione quais padrões e práticas foram definidos pela sua equipe.....120
FIGURA 57 - Código exemplo: uso da notação de *arrow function*.....121
FIGURA 58 - Código exemplo: uso de parênteses entre argumentos mesmo que a *arrow function* contenha apenas um.....121
FIGURA 59 - Código exemplo: omissão da *keyword "return"* em *arrow functions* que ocupem apenas uma linha.....122
FIGURA 60 - Código exemplo: evitando modificar os argumentos recebidos dentro da função.....122
GRÁFICO 13 - [*MODULES*] Selecione quais padrões e práticas foram definidos pela sua equipe.....123
FIGURA 61 - Código exemplo: unificar chamadas de módulos.....123
FIGURA 62 - Código exemplo: manter importações de módulos no topo do arquivo.....124
FIGURA 63 - Código exemplo: omissão da extensão *".js"* para importação de arquivos.....124
GRÁFICO 14 - [*STRING*] Selecione quais padrões e práticas foram definidos pela sua equipe.....125
FIGURA 64 - Código exemplo: uso de single quotes para *"strings"*.....125
FIGURA 65 - Código exemplo: uso de template literal para concatenar *"strings"*.....126
GRÁFICO 15 - [*OBJECTS*] Selecione quais padrões e práticas foram definidos pela sua equipe.....126
FIGURA 66 - Código exemplo: uso de *destructuring* para acessar propriedades de um "objeto".....127
FIGURA 67 - Código exemplo: uso de sintaxe literal para criação de "objetos".....127
FIGURA 68 - Código exemplo: uso de *shorthand* em "objetos".....128
FIGURA 69 - Código exemplo: acesso de propriedades de um "objeto" com *dot*

notation.....129
GRÁFICO 16 - [*REACT JS]* Selecione quais padrões e práticas foram definidos pela sua equipe.....129
FIGURA 70 - Código exemplo: fechamento de *tag* de componentes *React JS* nela mesma quando não houver *tags* filhas.....130
FIGURA 71 - Código exemplo: adição de texto para o atributo *"alt"* em *tags* de imagem.....130
FIGURA 72 - Código exemplo: alinhamento de estilos para sintaxe de *JSX*.....131
FIGURA 73 - Código exemplo: adição de um espaço em branco antes do fechamento de *tags* no *React JS*.....131
FIGURA 74 - Código exemplo: nomeação de componentes como *Pascal Case*.....131
FIGURA 75 - Código exemplo: evitar uso de "*index* de *array*" para identificar chaves de elementos.....132
FIGURA 76 - Código exemplo: uso de double quote em atributos *JSX*.....133
FIGURA 77 - Código exemplo: evitar espaços desnecessários dentro de chaves no *JSX*.....133
FIGURA 78 - Código exemplo: envolvimento do retorno de componentes em parênteses no caso de ocupar mais de uma linha.....134
FIGURA 79 - Código exemplo: omissão do valor da *"prop"* no caso de esta ser implicitamente de valor *"true"*.....134
GRÁFICO 17 - [*SYNTAX]* Selecione quais padrões e práticas foram definidos pela sua equipe.....135
FIGURA 80 - Código exemplo: uso de *triple equation* para comparação de valores.....136
FIGURA 81 - Código exemplo: evitar nomes de variáveis e funções com apenas uma letra.....136
FIGURA 82 - Código exemplo: uso de *Camel Case* para nomeação de variáveis, funções e instâncias.....137
FIGURA 83 - Código exemplo: uso de *"if"* ternário em apenas um nível.....137
FIGURA 84 - Código exemplo: definição de um máximo de caracteres por linha de

código.....138
FIGURA 85 - Código exemplo: uso de *Pascal Case* para nomeação de construtores e classes.....138
FIGURA 86 - Código exemplo: separação de operadores de comparação de lógicos.....139
FIGURA 87 - Código exemplo: uso de *HOF* ao invés de *"loops"* simples.....139
FIGURA 88 - Código exemplo: uso de ponto e vírgula no final de comandos.....140
GRÁFICO 18 - [*VARIABLE]* Selecione quais padrões e práticas foram definidos pela sua equipe.....141
FIGURA 89 - Código exemplo: uso de "const/let" na declaração de variáveis.....141
FIGURA 90 - Código exemplo: remoção de variáveis não utilizadas no código.....142
FIGURA 91 - Código exemplo: evitar uso de cadeia de variáveis.....143
FIGURA 92 - Código exemplo: uso de apenas uma *keyword "const/let"* por declaração de variável.....143
GRÁFICO 19 - [*WHITE SPACES]* Selecione quais padrões e práticas foram definidos pela sua equipe.....144
FIGURA 93 - Código exemplo: uso de espaço em branco dentro de chaves.....144
FIGURA 94 - Código exemplo: uso de espaço em branco entre operadores.....145
FIGURA 95 - Código exemplo: pular uma nova linha no final de cada arquivo……145
FIGURA 96 - Código exemplo: evitar espaços desnecessários no código.....146
FIGURA 97 - Código exemplo: uso de espaço em branco após *keywords*.....146
FIGURA 98 - Código exemplo: evitar uso de espaços em branco dentro de colchetes.....146
FIGURA 99 - Código exemplo: indentação de código utilizando dois espaços.....147
FIGURA 100 - Código exemplo: evitar uso de espaços em branco dentro de parênteses.....147
FIGURA 101 - Código exemplo: adição de espaço em branco no começo de um comentário.....148
FIGURA 102 - Tela da aplicação *Github Favorites* no cenário de sucesso.....149

FIGURA 103 - Tela da aplicação *Github Favorites* no cenário de usuário não existente.....150
FIGURA 104 - Tela da aplicação *Github Favorites* no cenário de erro genérico.....150
FIGURA 105 - Estrutura de pastas do projeto *GitHub Favorites*.....151
FIGURA 106 - Estrutura de do arquivo *"package.json"*.....153
FIGURA 107 - Terminal após execução do comando de instalação da ferramenta *ESLint* como dependência de desenvolvimento do projeto.....155
FIGURA 108 - Terminal após execução do comando para iniciar as configurações do *ESLint*.....156
FIGURA 109 - Arquivo *".eslintignore"* do projeto.....157
FIGURA 110 - Estado inicial do arquivo *".eslintrc.json"* do projeto.....158
FIGURA 111 - Terminal com a instalação dos *plugins*: *"import"* e *"jsx-a11y"*.....178
FIGURA 112 - Arquivo *".eslintrc.json"* configurado com *plugins*.....180
FIGURA 113 - Arquivo *".eslintrc.js"* configurado com *"rules"*.....181
FIGURA 114 - Arquivo *"package.json"* configurado com os *scripts* para executar o *ESLint*.....182
FIGURA 115 - Terminal apontando erros no código após executar o comando *"npm run eslint"*.....183
FIGURA 116 - Terminal apontando quais erros podem ser ajustados ao executar o comando *"npm run eslint:fix"*.....183
FIGURA 117 - Terminal apontando erros que devem ser ajustados manualmente.....184
FIGURA 118 - Terminal não acusando nenhum erro de padrão de código ao executar *"npm run eslint"*.....184
FIGURA 119 - *Plugin ESLint* disponível para o editor *VSCode*…….....185
FIGURA 120 - Arquivo *"settings.json"* com as configurações do *VSCode*.....185
FIGURA 121 - *Plugin ESLint* apontando erros em *living code*…….....186
FIGURA 122 - Resultado geral da questão sobre sobre Linguagens mais amadas, temidas e desejadas na pesquisa *StackOverflow Survey* 2020…….....190

LISTA DE TABELAS

TABELA 1 - Padrões de código e boas práticas para *React JS* e suas diretrizes no *ESLint*.....158
TABELA 2 - Parâmetros das regras no *ESLint* e configurações a serem implementadas.....163

LISTA DE ABREVIATURAS E SIGLAS

AJAX	*Asynchronous JavaScript and XML*
AlgoI	*ALGOrithmic Language*
API	*Application Programming Interface*
CISQ	*Consortium for Information Software Quality*
DOM	*Document Object Model*
DRY	*Don't Repeat Yourself*
FORTRAN	*Formula Translation*
GTI	Gestão da Tecnologia da Informação
HTML	*Hypertext Markup Language*
HOF	*High Order Function*
IBGE	Instituto Brasileiro de Geografia e Estatística
IDE	Ambiente de Desenvolvimento Integrado
IFSC	Instituto Federal de Santa Catarina
JS	*JavaScript*
JSON	*JavaScript Object Notation*
JSX	*JavaScript and XML*
KISS	*Keep It Simple, Stupid*
Lisp	*List Processing*
MIT	*Massachusetts Institute of Technology*
OOP	*Object-oriented programming*
PERL	*Practical Extraction and Reporting Language*
PPC	Projeto Pedagógico do Curso
Regex	*Regular Expression*
SPA	*Single Page Application*
TC39	Comitê Técnico 39
TI	Tecnologia da Informação
UI	*User Interface*
VSCode	*Visual Studio Code*
YAGNT	*You Aren't Gonna Need It*

SUMÁRIO

1. INTRODUÇÃO **23**
1.1. Justificativa **25**
1.2. Definição do problema **28**
1.3. Objetivo Geral **33**
1.4. Objetivos específicos **33**
1.5. Estrutura do trabalho **34**
2. EVOLUÇÃO DA PROGRAMAÇÃO, SUAS ABORDAGENS E CONCEITOS **36**
2.1. História e evolução da programação **36**
2.2. Programação: abordagens e conceitos **39**
2.2.1. Paradigmas da programação 39
2.2.2. Conceitos básicos da programação 40
2.2.2.1. Memória 41
2.2.2.2. Variáveis 41
2.2.2.3. Comentários 42
2.2.2.4. Entrada e saída de programa 42
2.2.2.5. Funções 44
2.2.2.6. Expressões e operadores 45
2.2.2.7. Estrutura de decisão 46
2.2.2.8. Estrutura de repetição 47
2.2.2.9. Classes e objetos 48
2.3. Princípios básicos da programação 49
3. LINGUAGEM DE PROGRAMAÇÃO JAVASCRIPT **50**
3.1. História e evolução da linguagem JavaScript **50**
3.2. Conceitos do JavaScript **52**
3.2.1. Tipos de dados no JavaScript 52
3.2.2. Armazenamento de dados em memória no JavaScript 52
3.2.3. Hoisting 54
3.2.4. Triple equals 54
3.2.5. High order functions (HOF) 55
3.2.6. Document Object Model (DOM) 56
3.3. Conceitos do JavaScript ES6+ **58**
3.3.1. Declaração de variáveis com: let e const 58
3.3.2. Template literal string 59
3.3.3. Default value 60
3.3.4. Arrow functions 60
3.3.5. Destructuring 61
3.3.6. Classes 63

3.3.7. ES6 Modules 64
3.3.8. Compilando o JavaScript ES6+ 65
3.4. Library React JS 67
3.4.1. Funcionamento do React 68
3.4.2. React Elements 68
3.4.3. JSX 70
3.4.4. Componentes 71
3.4.5. Props 72
3.4.6. Children 72
4. APLICABILIDADE DO CÓDIGO LIMPO 74
4.1. Código Limpo 74
4.2. Princípios do Código Limpo 76
4.3. Práticas de código limpo 77
4.3.1. Padrões de código e boas práticas em JavaScript 81
4.3.1.1. Padrões de código e boas práticas para Array 82
4.3.1.2. Padrões de código e boas práticas para Blocos 83
4.3.1.3. Padrões de código e boas práticas para Classes 84
4.3.1.4. Padrões de código e boas práticas para Funções 84
4.3.1.5. Padrões de código e boas práticas para Módulos 86
4.3.1.6. Padrões de código e boas práticas para String 86
4.3.1.7. Padrões de código e boas práticas para Objetos 87
4.3.1.8. Padrões de código e boas práticas para Sintaxe 87
4.3.1.9. Padrões de código e boas práticas para Variável 89
4.3.1.10. Padrões de código e boas práticas para Espaços em Branco 90
4.3.2. Padrões de código e boas práticas em React JS 91
4.4. Refatoração 93
4.5. Mensurando qualidade de código 94
4.6. Vantagens do código limpo 95
5. FERRAMENTA ESLINT 96
5.1. ESLint 96
5.2. ESLint: Style Guides 99
6. PROCEDIMENTOS METODOLÓGICOS 101
6.1. Caracterização da pesquisa 101
6.2. Delimitação e população da pesquisa 103
6.3. Etapas da pesquisa 104
6.3.1. ETAPA 1 - Revisão de literatura 105
6.3.2. ETAPA 2 - Aplicação de questionário 106
6.3.3. ETAPA 3 - Análise das respostas obtidas no questionário 106
6.3.4. ETAPA 4 - Especificação das diretrizes na ferramenta ESLint 107

6.3.5. ETAPA 5 - Conclusões 107

7. ANÁLISE DE PADRÕES DE CÓDIGO E BOAS PRÁTICAS DA PROGRAMAÇÃO NO MERCADO DE TECNOLOGIA 108

7.1. Limpeza de dados obtidos no questionário 108

7.2. Análise do cenário profissional dos respondentes 110

7.3. Análise sobre percepções de boas práticas e padrões de código dos respondentes 112

7.4. Análise sobre quais as boas práticas e padrões de código são utilizadas pelos respondentes no âmbito profissional 114

7.4.1. Array: padrões e boas práticas 114

7.4.2. Blocks: padrões e boas práticas 116

7.4.3. Classes: padrões e boas práticas 118

7.4.4. Functions: padrões e boas práticas 120

7.4.5. Modules: padrões e boas práticas 123

7.4.6. Strings: padrões e boas práticas 124

7.4.7. Objects: padrões e boas práticas 126

7.4.8. React JS: padrões e boas práticas 129

7.4.9. Syntax: padrões e boas práticas 135

7.4.10. Variables: padrões e boas práticas 140

7.4.11. White Spaces: padrões e boas práticas 143

8. IMPLEMENTAÇÃO DE DIRETRIZES NA FERRAMENTA ESLINT EM PROJETO ESCRITO COM REACT JS 149

8.1. Projeto Github Favorites 149

8.1.1. Estrutura de pastas do projeto Github Favorites 150

8.1.1.1 Estrutura do arquivo "package.json" 152

8.2. Implementação da ferramenta ESLint 154

8.2.1. Instalação do ESLint 154

8.2.2. Estrutura do arquivo de configuração ".eslintrc.json" 157

8.2.3. Regras selecionadas e diretrizes do ESLint 158

8.2.4. Configuração do ESLint 163

8.2.5. Executando comandos do ESLint no projeto 181

8.2.5.1. Plugin ESLint no Visual Studio Code 185

9. CONCLUSÕES 187

9.1. Em relação ao objetivo geral 187

9.2. Em relação aos objetivos específicos 188

9.3. Trabalhos futuros 189

REFERÊNCIAS 191

GLOSSÁRIO 194

ANEXO A - Perguntas de Configuração do ESLint 197

APÊNDICE A - Questionário sobre Padrões de Código e Boas Práticas da Programação 199

APÊNDICE B - Arquivo de configuração do ESLint para projetos construídos com React JS 205

1. INTRODUÇÃO

"Existem muitos aspectos da qualidade do código [...] código eficiente, que é um código que roda rápido ou que não desperdiça recursos [...] pode ser um código-fonte fácil de ler e entender para nós, humanos, e, portanto, fácil de ler e manter" (WASSBERG, 2020, s.p., tradução nossa).

No início da programação, desenvolvedores se preocupavam apenas com dois quesitos: o funcionamento do código e sua performance, visto que contavam com *hardwares* pouco avançados. Gradualmente surgiram preocupações com as regras de negócios da parte do cliente e mudanças constantes de escopo, demandando agilidade das equipes de desenvolvimento. Com esta evolução, os programadores começaram a destacar a importância da legibilidade de código e consequente qualidade de código.

De acordo com Wassberg (2020, s.p.), a organização *Consortium for Information Software Quality* (*CISQ*) - responsável por desenvolver padrões internacionais para automatizar a medição da qualidade de *software* - desenvolveu e apresentou um modelo de qualidade definindo cinco regras que podem ser usadas para medir a qualidade do código: confiabilidade, eficiência do desempenho, segurança, manutenção e tamanho do projeto.

As preocupações quanto à qualidade do código foram fomentadas em 2008 com o lançamento do livro Código Limpo: Habilidades Práticas do *Agile Software,* do autor Robert Cecil Martin. Gradualmente o panorama da programação se voltou à evolução na qualidade de produção de código. O livro é uma obra conceituada na área de Tecnologia da Informação (TI) e serve de padrão para projetos na mesma, além de inspirar livros de diversas linguagens de programação focados em padrões de código e boas práticas das mesmas.

Pressman, em seu livro Engenharia de *Software,* evidencia o conceito sobre a qualidade de *software*, definindo-o como "uma gestão da qualidade efetiva aplicada de modo a criar um produto útil que forneça valor mensurável para aqueles que o produzem e para aqueles que o utilizam" (PRESSMAN, 2011, p. 360).

"Um código limpo é simples e direto. Ele é tão bem legível quanto uma prosa bem escrita. Ele jamais torna confuso o objetivo do desenvolvedor, em

vez disso, ele está repleto de abstrações claras e linhas de controle objetiva" (BOOCH *apud* MARTIN, 2012, p. 8).

Wassberg afirma que "se o código for fácil de ler e entender, os programadores também terão uma chance maior de evitar a inserção de novos *bugs* no código, reduzindo assim o custo de corrigi-los" (WASSBERG, 2020, s.p., tradução nossa). Stefanov (2010, s.p.) descreve a relação de *bugs* e custos, afirmando que os "custos aumentam com o tempo, especialmente se os *bugs* invadirem o produto lançado publicamente. É melhor se você puder consertar um *bug* imediatamente, assim que o encontrar" (STEFANOV, 2010, s.p., tradução nossa).

Este trabalho expôs a ideia de que a responsabilidade de um programador vai além de escrever códigos com boa lógica e melhor performance, sugeriu-se que o real objetivo é produzir códigos legíveis, aplicando padrões de código e boas práticas da programação. Martin Fowler (2009, s.p., tradução nossa) pontua que "qualquer um consegue escrever código que um computador entende. Bons programadores escrevem código que humanos entendem".

Padolsey (2020, s.p.) afirma que ao escrever um código é essencial considerar como o cérebro humano o consumirá. "Legibilidade é o primeiro obstáculo que os desenvolvedores devem superar. Se eles forem incapazes de ler e navegar cognitivamente no código que você escreveu, eles serão menos capazes de usá-lo. Isso limita drasticamente a utilidade e o valor do seu código” (PADOLSEY, 2020, s.p., tradução nossa).

Como resultado final deste trabalho foi disponibilizado, de forma digital e com acesso público, um arquivo com as especificações para o uso da ferramenta *ESLint*, visando auxiliar times a programarem de forma padronizada e promovendo o código limpo para a linguagem *JavaScript*. Foram selecionadas diretrizes de melhores práticas de código com base bibliográfica e validadas por meio de um questionário aplicado entre profissionais desenvolvedores, de diversas empresas, que utilizam a *library React JS*.

As regras de validação de código foram transcritas através de configurações da ferramenta existente já citada. A solução foi testada em um projeto real de autoria própria da autora desenvolvido com a *library* de *JavaScript*, *React JS*.

1.1. Justificativa

Herrero (2019, s.p.) afirma que a tarefa de desenvolvimento de *software* não é mais uma tarefa de uma pessoa só e sim de um time, enfatizando que "nosso código será lido por nós e por nossa equipe, constantemente. O código será lido, revisado, usado e eventualmente estendido. Isso se torna um problema se ninguém entende nosso código, nem mesmo nós" (HERRERO, 2019, s.p., tradução nossa).

Os conceitos de código limpo não se limitam a linhas de código. No seu âmago, abrangem projetos de tecnologia da informação. Antes de códigos serem concebidos, a equipe separa - ou deveria separar - um tempo para entender o escopo que a solução deve atender. Afinal, "código é a linguagem na qual expressamos nossos requisitos" (MARTIN, 2012, p. 12), ou seja, programar é traduzir requisitos de negócios em funcionalidades automatizadas que satisfaçam as peculiaridades dos clientes.

Programadores passam a maior parte de sua jornada de trabalho lendo códigos já feitos e tentando entender sua funcionalidade. A proporção desta atividade chega a ser 10:1, segundo Robert C. Martin (2012, p. 23). O ato de codificar é sensível pois o código "não será executado apenas pelo computador. Também será lido por você e por outras pessoas. Portanto, é essencial que você escreva um código que seja tão fácil de ler e entender quanto possível" (WASSBERG, 2020, s.p., tradução nossa).

Stefanov (2010, s.p.) enfatiza a importância de manter um padrão dentro da programação, afirmando que padrões ajudam a escrever códigos melhores usando práticas comprovadas, fornecem um nível de abstração e melhoram a comunicação entre desenvolvedores e equipes. "Simplesmente colocar um rótulo em alguma técnica ou abordagem de codificação torna mais fácil ter certeza de que estamos falando sobre a mesma coisa” (STEFANOV, 2010, s.p., tradução nossa).

Criar códigos bons e de fácil manutenção deve ser uma das missões das equipes de desenvolvimento para que *features* possam ser lançadas facilmente, otimizações promovidas e erros e defeitos corrigidos rapidamente.

Herrero (2019, s.p.) desafia os desenvolvedores a tomarem a responsabilidade sobre seu código: "somos nós que sabemos quantas coisas ruins podem acontecer quando escrevemos um código ruim. E somos nós que temos que nos levantar com determinação e decidir escrever um bom código, um código limpo, um código profissional" (HERRERO, 2019, s.p., tradução nossa).

O tema está alinhado com um dos focos do curso de Gestão da Tecnologia da Informação (GTI), seguindo o Projeto Pedagógico do Curso (PPC, 2018, p. 5) do Instituto Federal de Santa Catarina (IFSC), que se compromete a desenvolver a visão e raciocínio estratégico do egresso para a definição e implementação dos princípios de gestão das tecnologias da informação. O conhecimento de código limpo e como aplicar de forma prática em sua equipe garante entregas concisas ao cliente, entregando aplicações seguras e certificando-se de que o projeto tenha capacidade de escalar futuras demandas.

Também é um compromisso do gestor de TI ter uma visão estratégica sobre custos de projetos que poderiam ser reduzidos. Pressman (2011, p. 366) compara o custo da qualidade nas empresas, onde a qualidade tem um preço, porém "a falta de qualidade também tem um preço - não apenas para os usuários finais, que terão de conviver com um *software* com erros, mas também para a organização de *software* que o criou, e além de tudo, terá de fazer a manutenção" (PRESSMAN, 2011, p. 366).

> Você pode escrever o *software* mais rápido, mas com qualidade inferior, tanto para os usuários quanto para os programadores que precisarão manter o código no futuro. Isso provavelmente será menos econômico do que criar código de alta qualidade.
> Também deve ser observado que, se iniciarmos um projeto com código mal escrito, esse projeto provavelmente sempre conterá código de baixa qualidade, pois o custo de voltar e melhorar todo o código será muito alto (WASSBERG, 2020, s.p., tradução nossa).

Herrero (2019, s.p.) afirma que a definição de regras de código limpo e de estrutura é obrigatória quando se trabalha em uma equipe de desenvolvedores, sugerindo também que essas regras e práticas devem ser determinadas o mais rápido possível e aplicadas em cada linha de código escrita.

"Ambientes de desenvolvimento hoje em dia tornam isso muito fácil, pois você pode criar as regras e exportá-las para um arquivo que pode ser compartilhado posteriormente entre a equipe" (HERRERO, 2019, s.p., tradução nossa).

"Na programação, o termo *lint* se refere ao código que, embora tecnicamente correto, é desagradável à vista, ou um possível *bug* [...] *linter* é uma ferramenta para detectar *lint* em seu código, e *linting* é o processo de executar um *linter* em seu código" (FLANAGAN, 2020, s.p., tradução nossa).

ESLint é um *Linter* para a linguagem de programação *JavaScript (JS)*, é a ferramenta mais conhecida por ser plugável e também estar disponível como extensão no *Visual Studio Code* (*VSCode*) - editor de código fonte desenvolvido pela *Microsoft*.

Apesar de existirem *style guides* para o *ESLint* já pré-definidos e utilizados atualmente - como *AirBnb* e *Google JavaScript Style Guide* -, times de desenvolvimento acabam definindo regras próprias e reescrevem as regras padrões destes *styles guide.*

Conceber um arquivo de configuração do *ESLint* onde as regras mais comuns estejam definidas, com base no questionário aplicado para desenvolvedores que trabalham com essas tecnologias, teria um bom uso na área de tecnologia e visa poupar esforço em configurar a ferramenta.

1.2. Definição do problema

Segundo Martin (2012, p. 4), "se você é um programador há mais de dois ou três anos, provavelmente o código confuso de outra pessoa já fez você trabalhar mais lentamente e provavelmente o seu próprio código já lhe trouxe problema".

Código ruim não é um problema somente da equipe de desenvolvimento. Ele nasce neste time mas se expande até o cliente. O código não legível causa desgaste por parte de manutenção e adição de melhorias. Em um dado momento, os desenvolvedores encaram muitas dificuldades e a manutenção no código torna-se uma árdua tarefa, gerando pouca confiança no código.

> Quando um *software* é bem-sucedido - atende às necessidades dos usuários, opera perfeitamente durante um longo período, é fácil de modificar e, mais fácil ainda, de utilizar -, ele é realmente capaz de mudar as coisas para melhor. Porém, quando um *software* falha - quando seus usuários estão insatisfeitos, quando é propenso a erros, quando é difícil modificá-lo e mais difícil ainda utilizá-lo -, fatos desagradáveis podem e, acontecem. (PRESSMAN, 2011, p. 31)

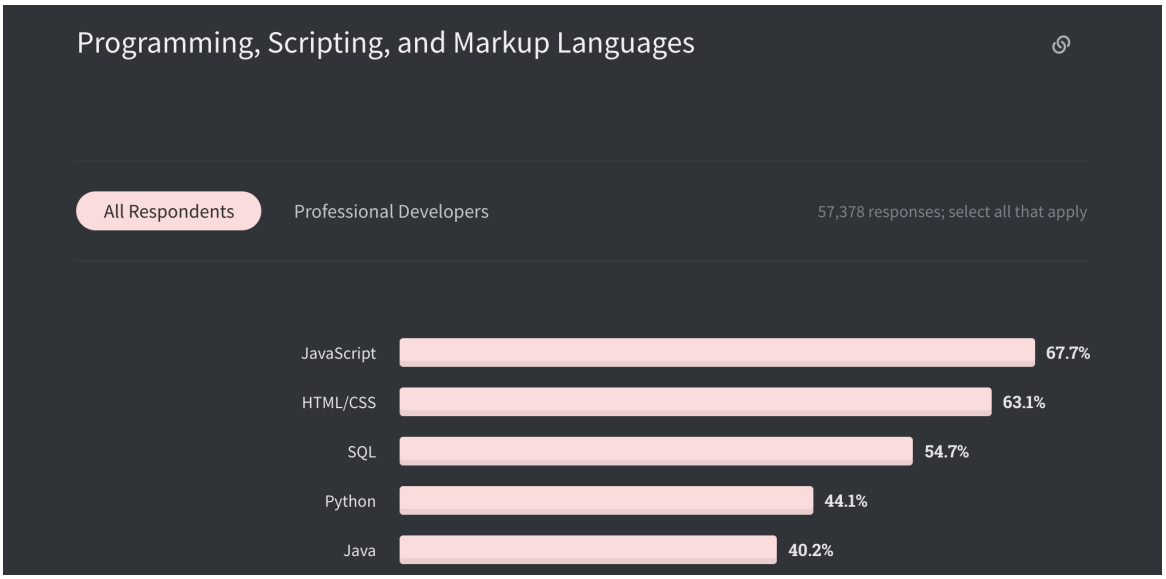
O código sem padrões acaba não sendo intuitivo para colaboradores novos e fica vulnerável a erros e defeitos - para Pressman (2019, p. 374) erro (*bug*) é um problema de qualidade encontrado antes de o *software* ser liberado ao usuário final e defeito é um problema encontrado após o *release* do *software*. A falta de qualidade no código reflete em acúmulo de erros e defeitos em produtos e serviços causando insatisfação no cliente - afinal, toda mudança demora a ser entregue e possivelmente vem acompanhada de um novo problema.

Padolsey (2020, s.p.) constrói um entrelace interessante sobre como o cérebro humano funciona ao lidar com padrões: "nossos cérebros altamente evoluídos, entre seus muitos talentos, tornaram-se incrivelmente hábeis em detectar padrões e inconsistências. Contamos com a diferença ou contraste para focar nossa atenção" (PADOLSEY, 2020, s.p., tradução nossa).

> Se um código é facilmente modificado, significa que ele pode ser refatorado ou estendido de forma simples. Então também será fácil para qualquer um consertar *bugs* ou erros. Tudo é mais fácil se o desenvolvedor que deve fazer as modificações consegue entender completamente o código. As mudanças não devem nunca quebrar as funcionalidades que já existem na aplicação. (LEWIS, 2020, s.p., tradução nossa)

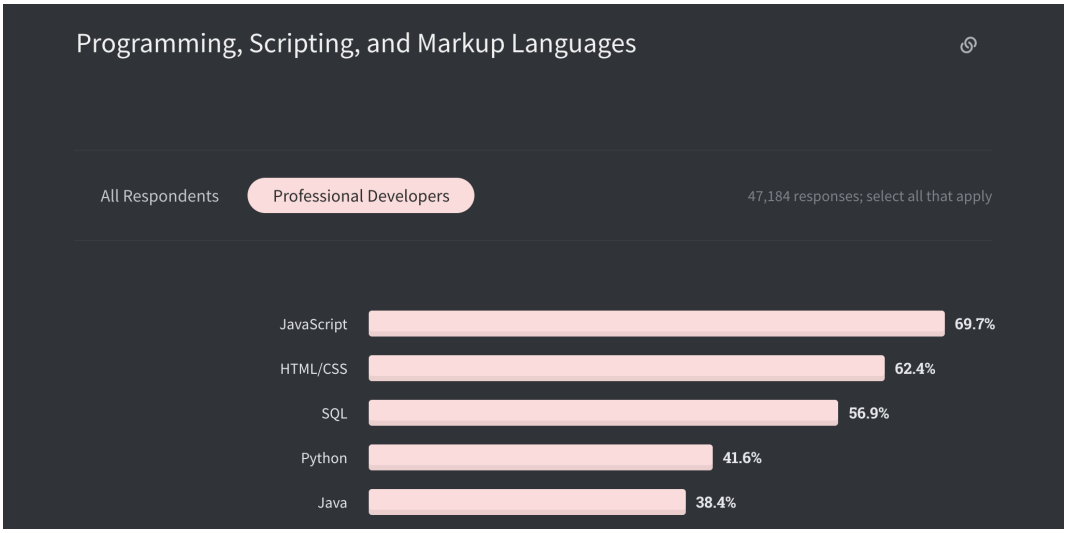
Em 2020, a última pesquisa para desenvolvedores realizada pelo *StackOverflow* - o maior *site* de perguntas e respostas da área de desenvolvimento – que envolveu, em média, cerca de 65.000 entrevistados, apontou que "pelo oitavo ano consecutivo, o *JavaScript* manteve sua fortaleza como a linguagem de programação mais comumente usada" (*STACKOVERFLOW SURVEY* 2020, tradução nossa) - representando 67,7% do total dos entrevistados (figura 1) e 69.7% dos entrevistados que atuam profissionalmente da área (figura 2). (*STACKOVERFLOW SURVEY* 2020)

Figura 1 - Resultado geral da questão sobre sobre Linguagens de: Programação, *Script* e *Markup* na pesquisa *StackOverflow Survey 2020.*



Fonte: *STACKOVERFLOW SURVEY,* 2020.

Figura 2 - Resultado por desenvolvedores profissionais da questão sobre Linguagens de: Programação, *Script* e *Markup* na pesquisa *StackOverflow Survey 2020.*



Fonte: *STACKOVERFLOW SURVEY,* 2020.

Quanto a *frameworks web* "*jQuery* ainda é o rei, mas está gradualmente perdendo espaço para *React.js* e *Angular* ano após ano" (*STACKOVERFLOW SURVEY* 2020, tradução nossa). *React JS* se posiciona em segundo lugar, com pouca diferença quanto ao uso do *jQuery* - com a porcentagem de 35.9% de todos os entrevistados (figura 3) e 36.8% dos profissionais desenvolvedores (figura 4) (*STACKOVERFLOW SURVEY,* 2020).

Figura 3 - Resultado geral da questão sobre *Web Frameworks* na pesquisa *StackOverflow Survey 2020.*

Fonte: *STACKOVERFLOW SURVEY,* 2020.

Figura 4 - Resultado por desenvolvedores profissionais da questão sobre *Web Frameworks* na pesquisa *StackOverflow Survey 2020.*

Fonte: *STACKOVERFLOW SURVEY,* 2020.

Com base nos dados apresentados nos parágrafos anteriores, percebeu-se que um estudo a fim de especificar as regras para o uso de uma ferramenta que atue na linguagem *JavaScript*, focando em *React JS*, com a finalidade de trazer dicas aos desenvolvedores para a manutenção de código limpo e padrões de projeto, seria relevante para grande parte dos profissionais da área de programação.

> Determinar um conjunto de regras e uma estrutura de formatação limpa é importante para um desenvolvedor e ainda mais importante para toda a equipe. Quando o código é compartilhado, revisado e refatorado, concordar com as mesmas regras de formato é obrigatório. Reduzindo consideravelmente o tempo necessário para entender o código de toda a equipe e aumentando a velocidade com que este código é usado. (HERRERO, 2019, s.p., tradução nossa)

"Uma das razões pelas quais usa-se *linters* é a imposição de um estilo de codificação consistente para que, quando uma equipe de programadores estiver trabalhando em uma base de código compartilhada, utilizem convenções de código compatíveis" (FLANAGAN, 2020, s.p., tradução nossa)

Este trabalho respondeu à seguinte questão: "Quais são as especificações mínimas necessárias para inclusão da ferramenta *ESLint* em um projeto escrito em *React JS,* com o objetivo de auxiliar desenvolvedores a manterem o código limpo?".

Para responder tal indagação, procedeu-se à documentação das etapas para incorporar a ferramenta a um projeto real da autoria da própria autora - escrito em *React JS -,* com o intuito de que o mesmo possa servir de base para estudantes do curso de GTI que venham a atuar com essa tecnologia e para auxiliar profissionais da área a configurarem de forma personalizada a ferramenta.

1.3. Objetivo Geral

Este projeto teve como objetivo geral a especificação de parâmetros de configurações necessárias para a utilização da ferramenta *ESLint* em projetos que utilizam a *library React JS* visando auxiliar desenvolvedores a escreverem códigos limpos na linguagem *JavaScript*.

1.4. Objetivos específicos

A. Identificar e selecionar padrões de código e boas práticas de código limpo utilizando a linguagem *JavaScript*.

B. Identificar e selecionar padrões de código e boas práticas de código limpo utilizando a library *React JS*.

C. Validar as regras escolhidas através de bibliografia por meio de aplicação de um questionário junto a profissionais do mercado.

D. Integrar o *ESLint* ao projeto real, especificando todas as suas configurações.

E. Testar as especificações no projeto escrito em *React JS.*

1.5. Estrutura do trabalho

Este trabalho foi estruturado em cinco capítulos, além dos elementos pré-textuais e pós-textuais. O capítulo atual refere-se a introdução sobre o trabalho, apresentando o tema, a justificativa de sua escolha, além de definir os objetivos gerais e específicos deste estudo.

O segundo capítulo - referencial teórico - se divide em quatro partes. A primeira - Evolução da programação, suas abordagens e conceitos - discorre sobre a história da programação e sua evolução, além de clarificar abordagens, conceitos e princípios básicos da programação, a fim de construir uma base de conhecimento coerente para prosseguir com a leitura deste trabalho.

A segunda parte, Linguagem de programação *JavaScript,* se dedica a detalhar a história da linguagem, elucidar os conceitos do *JS* e *features* da versão mais moderna da linguagem, e apresentar a *library React JS* e seus pilares básicos - fundamentando a compreensão de padrões de código que podem ser aplicadas na ferramenta *ESLint*.

A terceira parte, Aplicabilidade do código limpo, apresenta a definição e conceitos de *clean code*, as melhores práticas de codificação - atendo-se, também, a especificidades em *JavaScript* e *React JS* - e suas vantagens, além de explorar a ideia de refatoração de código.

A última parte, Ferramenta *ESLint*, demonstra a ferramenta *ESLint*, que é utilizada neste trabalho para definir regras e padrões de sintaxe a serem utilizadas no projeto. Também fazem-se comparações entre os *styles guides* que já são pré-definidos na ferramenta.

O terceiro capítulo, Procedimentos Metodológicos, é dedicado a caracterização da pesquisa, sendo aplicada quanto a sua natureza; qualitativa quanto a sua abordagem; descritiva quanto ao seu objetivo e, quanto aos procedimentos técnicos, encaixa-se em bibliográficas, de levantamento e pesquisa de campo - visto que, além da busca de informações em livros, artigos, revistas e *blogs*, também aplicou-se um questionário via *internet* a fim de catalogar quais padrões de código profissionais da área realmente utilizam em projetos de suas

empresas - possibilitando a comparação com as regras escolhidas através de bibliografia e a definição dos parâmetros a serem implantados na ferramenta *ESLint*.

Quanto à delimitação e população da pesquisa, têm-se quatro pontos: padrões de codificação, código limpo na linguagem *JavaScript* e *React JS*, especificações da ferramenta para uso junto a *library React JS* e delimitação de amostra para aplicação do questionário.

O quarto capítulo se divide em duas partes: a primeira se refere a Análise de padrões de código e boas práticas da programação no mercado de tecnologia, onde apresentou-se os resultados obtidos no questionário, através de gráficos e figuras de exemplos de código, a fim de explicitar quais regras serão escolhidas para serem utilizadas no *ESLint*.

Na segunda parte, Implementação de diretrizes na ferramenta *ESLint* em projeto escrito com *React JS*, estão documentadas todas as etapas seguidas para implementação da ferramenta em um projeto real escrito com *React JS*. Apresenta-se, neste capítulo, a estrutura do projeto em que será implementado o *ESLint* e seu objetivo. O roteiro da implementação percorreu a jornada desde a instalação até a configuração e execução de comandos para verificação de erros no projeto.

No último capítulo, estão expostas as conclusões acerca dos entregáveis deste trabalho comparando-o com os objetivos geral e específicos, mapeados na concepção deste estudo, além de idealizar trabalhos futuros a serem implementados posteriormente.

2. EVOLUÇÃO DA PROGRAMAÇÃO, SUAS ABORDAGENS E CONCEITOS

Este capítulo aborda a história da programação, a fim de traçar sua evolução durante os anos. Posteriormente, serão apresentadas abordagens e conceitos essenciais da programação.

2.1. História e evolução da programação

"Uma linguagem de programação é um sistema projetado para criar uma forma padrão de comandos que podem ser interpretados em um código compreendido por uma máquina" (MADHAWA, 2020, s.p., tradução nossa).

Segundo Ferguson (2000, p.1), as linguagens de computador foram inicialmente compostas de uma série de etapas para se conectar a um programa específico. Posteriormente, se transformaram em uma série de etapas digitadas no computador e gradualmente essas linguagens foram adquirindo recursos avançados, como ramificação lógica e orientação a objetos.

"No início, o mecanismo de diferença de Charles Babbage só podia executar tarefas, alterando as engrenagens que executavam os cálculos. Assim, a forma mais antiga de linguagem de computador era o movimento físico" (FERGUSON, 2000, p.1, tradução nossa).

Ada Augusta Byron King (1815-1852) - mais conhecida como Ada Lovelace - foi a primeira programadora de computadores. Segundo Madhawa (2020, s.p.), Ada ajudou Babbage a aperfeiçoar seu projeto relacionado a um computador mecânico e durante este processo inventou o conceito de programação.

Com o passar do tempo surgiu a necessidade da criação de linguagens específicas para solucionar cada tipo de problema, surgindo, então, o conceito de algoritmos. Algoritmo pode ser definido como "um conjunto de regras ou uma descrição das etapas a serem seguidas nas operações de solução de problemas" (WASSBERG, 2020, s.p., tradução nossa). Os programas de computadores são escritos por meio de algoritmos que atendem a uma linguagem específica. As linguagens foram evoluindo, algumas em paralelo, e outras seguiram caminhos específicos para atender uma função.

De acordo com Madhawa (2020, s.p.), Alan Turing (1912 - 1954) foi a primeira pessoa a construir um computador binário, conhecido como o mais distante da ciência da computação moderna.

"Em 1957, uma das principais linguagens apareceu na forma de *FORTRAN*. A linguagem foi projetada na *IBM* para computação científica [...] e fornecia ao programador acesso de baixo nível ao computador" (FERGUSON, 2000, p.1, tradução nossa).

"Em 1958, John McCarthy do *MIT* criou a linguagem *List Processing* (*LISP*) [...] o lançamento original do *LISP* tinha uma sintaxe única: essencialmente nenhuma. Os programadores escreviam código em árvores de análise" (FERGUSON, 2000, p.1, tradução nossa).

Ainda, em 1958, surgiu "a próxima grande linguagem, *Algol* (*ALGOrithmic Language*), foi a invenção dos cientistas da computação para permitir uma forma consistente e compacta de representar programas para publicação e como uma linguagem de pesquisa" (MADHAWA, 2020, s.p., tradução nossa). Segundo Madhawa (2020, s.p.), a linguagem introduziu muitas das ideias consideradas certas atualmente, como a distinção entre atribuição e igualdade, e foi uma das primeiras linguagens baseadas em pilha estruturada de blocos que evoluíram para *C, C++, C#* e *Java*.

No ano seguinte, a computação empresarial começou a ganhar destaque, então surgiu o *COBOL*, "seus únicos tipos de dados eram números e *strings*. Também permitia que os dados fossem agrupados em matrizes e registros, para que pudessem ser rastreados e organizados de uma forma melhor" (FERGUSON, 2000, p.1, tradução nossa).

"*Visual Basic* [...] é baseado na linguagem *BASIC* desenvolvida em 1964 por John Kemeny e Thomas Kurtz. *BASIC* é uma linguagem muito limitada e foi projetada para pessoas que não são da ciência da computação" (FERGUSON, 2000, p.1, tradução nossa).

"Pascal foi criado em 1968 por Niklaus Wirth. Seu desenvolvimento foi principalmente devido à necessidade de uma boa ferramenta de ensino. No início, os *designers* da linguagem não esperavam que ele fosse amplamente adotado" (FERGUSON, 2000, p.1, tradução nossa).

Em 1972 surgiu a linguagem *C*, e "seus principais recursos incluem acesso de baixo nível à memória, um conjunto simples de palavras-chave e estilo limpo [...] Muitas linguagens posteriores emprestaram sua sintaxe/recursos direta ou indiretamente" (MADHAWA, 2020, s.p., tradução nossa).

No final dos anos 1970, surgiu a Programação Orientada a Objetos (*OOP*). "Bjarne Stroustroup gostou deste método e desenvolveu extensões para *C* conhecidas como '*C* com classes'. Este conjunto de extensões desenvolveu-se na linguagem *C++* com todos os recursos, lançada em 1983" (FERGUSON, 2000, p.1, tradução nossa).

"O *Perl* foi desenvolvido por Larry Wall em 1987 porque as ferramentas *Unix sed* e *awk* (usadas para manipulação de texto) não eram mais fortes o suficiente para atender às suas necessidades" (FERGUSON, 2000, p.1, tradução nossa).

"No início de 1990, a televisão interativa era a tecnologia do futuro. A *Sun Microsystems* decidiu que a televisão interativa precisava de uma linguagem especial e portátil. Essa linguagem eventualmente se tornou o *Java*" (FERGUSON, 2000, p.1, tradução nossa).

Para Ferguson (2000, p.1), as primeiras grandes linguagens caracterizavam-se pelo simples fato de serem destinadas a um único propósito. As linguagens atuais são diferenciadas pela forma como são programadas, por serem utilizadas para quase todos os fins. E, talvez, as linguagens futuras sejam mais naturais com a invenção dos computadores quânticos e biológicos.

2.2. Programação: abordagens e conceitos

"Linguagem de programação é um sistema de escrita formal para enunciar a execução de operações [...] e tem como finalidade fundamental proporcionar meios para a resolução de problemas, mediante um processamento computadorizado " (ANDRADE, 2017, p. 8).

Atualmente, existem inúmeras linguagens de programação: *Java, Python, C++, C, C#, JavaScript, Clojure, Cobol*, *Assembly,* entre outras. Linguagens de programação são agrupadas em dois níveis: baixo nível e alto nível. Andrade (2017, p. 11) define linguagens de baixo nível como sendo a linguagem onde cada instrução é associada a números binários (0 e 1), como, por exemplo, *Assembly*. Por outro lado, linguagens de alto nível se aproximam mais da linguagem natural do desenvolvedor, como *JavaScript.*

Para que o código fonte escrito pelo desenvolvedor seja entendido pelas máquinas existem duas técnicas principais: interpretação e compilação. "A interpretação olha para uma única linha de código-fonte, a traduz em código de máquina, deixa o computador executar essa linha e então passar para a próxima linha de código" (WASSBERG, 2020, s.p., tradução nossa). "Quando compilamos o código-fonte em código de máquina, primeiro traduzimos todas as linhas de código, e o programa só é executado após todas as linhas de código serem traduzidas" (WASSBERG, 2020, s.p., tradução nossa).

2.2.1. Paradigmas da programação

Para guiar a construção de soluções a fim de resolver os problemas com código existem os paradigmas de programação. "Paradigma de programação é um modelo, padrão ou estilo de programação suportado por linguagens que agrupam certas características comuns." (ANDRADE, 2017, p. 14). Como bem definido por Sousa, Dias e Formiga (2017, p.15), programar seguindo um paradigma de programação nada mais é que representar soluções a partir de uma forma particular de raciocinar na elaboração dos algoritmos.

Os paradigmas de programação mais utilizados são:

- Programação imperativa (procedural): "Representa a computação como ações, enunciados ou comandos que alteram o estado (variáveis) de um programa. Consiste na elaboração de programa a partir de comandos que dizem o que o computador deve fazer a cada momento." (SOUSA; DIAS; FORMIGA, 2017, p. 15). Exemplos: *Pascal, C*.
- Programação orientada a objectos: "Enxerga o problema como uma coleção de objetos que se comunicam por meio da troca de mensagens. Os objetos são estruturas de dados que possuem estado (variáveis) e comportamento (lógica)." (SOUSA; DIAS; FORMIGA, 2017, p. 15). Exemplos: *C++, Java*.
- Programação Funcional: "Estilo de programação que tem origem na teoria das funções matemáticas. Enfatiza o processamento de valores através do uso de expressões e Funções." (ANDRADE, 2017, p. 14). Exemplo: *LISP, Clojure*, *Elixir*.

Linguagens de programação podem ser multi-paradigmas, ou seja, podem se adaptar a mais de um paradigma conforme a necessidade de uso do desenvolvedor, *PHP* e o *Python* conseguem atuar em paradigma procedural e orientado a objetos; o *JavaScript* além desses paradigmas pode ser escrito baseado no viés funcional.

2.2.2. Conceitos básicos da programação

Programas de sistema, mais conhecidos como *software,* "consistem em: (1) instruções (programa de computador) que, quando executadas, fornecem características, funções e desempenho desejados" (PRESSMAN, 2011, p. 56).

Apesar de cada linguagem possuir suas peculiaridades e sintaxes próprias, existem conceitos básicos que são válidos em quase todas elas. É o caso de memória, variáveis, comentários, entradas e saídas de dados, funções, estrutura de decisão, estrutura de repetição - no caso de linguagens do paradigma funcional trabalha-se com recursividade - e operadores lógicos e matemáticos.

2.2.2.1. Memória

A principal memória do computador, conhecida como memória RAM, é constituída por componentes eletrônicos capazes de armazenar dados. Cada pedaço da memória é ocupado por um dígito binário (0 ou 1) e é chamado de *bit*, o conjunto de 8 *bits* é chamado de *byte*. Para cada valor armazenado na memória, existe um número único de identificação chamado de endereço, este "tem a mesma função que os endereços de nossas casas, que é identificar de forma única nossas residências (...). No caso do computador, as correspondências são os dados que serão armazenados nas células de memória." (SOUSA; DIAS; FORMIGA, 2017, p. 17).

2.2.2.2. Variáveis

Variável é "um objeto identificado pelo seu nome, podendo conter dados que poderão ser modificados durante a execução do programa" (ANDRADE, 2017, p. 44). Existem diversos tipos de dados que podem variar entre linguagens de programação, como *"string"* (texto), *"array"* (lista), "objetos" (chave e valor) e *"number"* (número). Em algumas linguagens, o tipo número se divide em *"float"*, *"double"* e *"int"*; *"string''* se chama *"varchar"* ou *"char"; "array"* se chama tupla; "objeto" é conhecido como dicionário, enfim entre linguagens podem haver variações sobre os tipos de dados suportados.

"Uma variável tem um nome e um tipo de dados, o tipo da variável define o que pode ser armazenado nela e quanta memória irá usar" (WASSBERG, 2020, s.p., tradução nossa). A declaração e a atribuição de valores em variáveis são dois processos diferentes. No exemplo da figura 5 declaram-se as variáveis com a palavra-chave *"var"* seguida do nome desta, e atribui valores a elas com o operador de igualdade (=) e o valor à direita.

Figura 5 - Exemplo de declarações de variáveis, escrito em *JavaScript*.

```
var name = 'Jennifer';
var age = 27;
var hobbies = ['code', 'tv_shows', 'beer'];
```

Fonte: própria Autora.

2.2.2.3. Comentários

"Um comentário é uma linha de texto normal, inserida no código, lida apenas por humanos. Nós os usamos para explicar linhas de código que não são obviamente compreendidas à primeira vista" (WASSBERG, 2020, s.p., tradução nossa).

Wassberg (2020, s.p.) afirma que o compilador ou interpretador ignoram os comentários, mas para isso deve-se indicar que o bloco é um comentário. Há duas opções de comentários: um que vai até o final da linha (comentário de linha) e outro que abrange várias linhas (comentário de bloco). A sintaxe pode variar entre linguagens. Abaixo segue um exemplo de comentário em *JavaScript* com as duas variantes - figura 6.

Figura 6 - Exemplo de comentário, escrito em *JavaScript*.

```
// One line comment
/**
 * Several lines comment
 * Another comment
 */
```

Fonte: própria Autora.

2.2.2.4. Entrada e saída de programa

Entradas e saídas de um programa são conceitos imprescindíveis, afinal "um programa de computador é praticamente inútil se não apresentar algum tipo de interação com o usuário" (SOUSA; DIAS; FORMIGA, 2017, p. 27). É comum

que o programa precise captar informações do usuário e devolver alguma resposta final a ele, sendo, então, sua entrada e saída. Na linguagem *JavaScript,* o comando para aceitar entrada nativa do usuário é o *"window.prompt"*, e o para saída é *"window.alert"*.

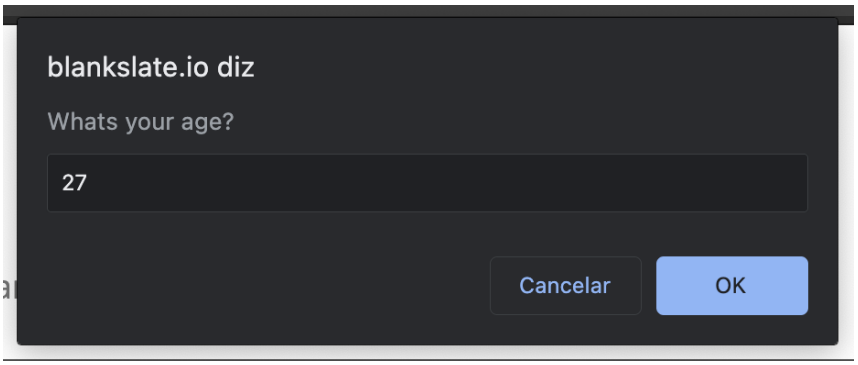
Na figura 7, a função ao ser executada irá abrir uma janela no navegador com um *"input"* para que o usuário digite sua idade (figura 8). No caso do dado de entrada ser maior que 18, o programa mostra um modal com uma mensagem apropriada (figura 9).

Figura 7 - Exemplo de entrada e saída de dados, escrito em *JavaScript*.

```
/**
 * Checks if age is bigger than 18
 */
function checkAge () {
    var age = window.prompt('Whats your age?', '');

    if (age > 18) {
        window.alert('You`re under 18!');
    }
}
```

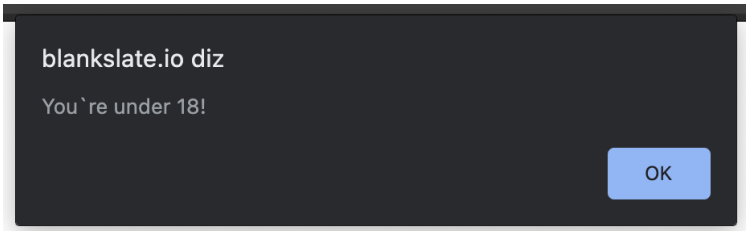
Fonte: própria Autora.

Figura 8 - Janela no navegador com um *"input"* para que o usuário digite sua idade



Fonte: própria Autora.

Figura 9 - Modal de saída com mensagem indicando que a idade é maior que 18.



Fonte: própria Autora.

2.2.2.5. Funções

Funções "podem ser compreendidas como trechos reutilizáveis de código. Uma função [...] pode, então, ser utilizada várias vezes por um mesmo programa" (SOUSA; DIAS; FORMIGA, 2017, p. 63). A ideia de escrever funções é poder chamá-la em qualquer momento do código sem ter sua lógica reescrita.

Para Wassberg (2020, s.p.) e Martin (2012, p. 34) as funções devem ser sucintas, ter apenas uma responsabilidade e receber um nome que reflita o motivo da sua existência.

"Às vezes, queremos que as funções produzam um valor como resultado de suas operações [...] também podemos passar dados para uma função, usando argumentos de função. Os argumentos são recebidos como variáveis locais chamadas parâmetros" (WASSBERG, 2020, s.p., tradução nossa).

No exemplo na figura 10, a função *"setGreeting"* recebe o *"name"* como parâmetro para obter como saída a saudação personalizada. Quando a função é chamada, na última linha, se envia um argumento dentro dos parênteses.

Figura 10 - Exemplo de função, recebendo parâmetro, escrito em *JavaScript*.

```
/**
 * Sets a personalized greeting, according to the name
 * @param { String } name person's name
 * @returns { String } personalized greeting
 */
function setGreetings (name) {
    return 'Hi, ' + name;
}

setGreetings('Jennifer');
```

Fonte: própria Autora.

2.2.2.6. Expressões e operadores

"Um programa tem como característica fundamental a capacidade de processar dados, ou seja, realizar operações com este dados. [...] As operações a serem realizadas com os dados podem ser determinadas por operadores" (ANDRADE, 2017, p. 47).

Existem quatro tipos de operadores básicos: de atribuição - já visto no tópico sobre variáveis como o sinal de = (igual) usado para atribuir valores -, aritméticos, incrementais, relacionais e lógicos

Operadores aritméticos "se relacionam às operações aritméticas básicas e possuem prioridade diferentes" (ANDRADE, 2017, p. 47), seguindo a matemática simples. Geralmente, as linguagens de programação contam com os seguintes operadores matemáticos: soma (+), subtração (-), divisão (/), multiplicação (*), módulo (%, onde retorna o resto da divisão). E para utilizá-los basta adicionar um operador entre dois operandos, construindo-se assim uma expressão - figura 11.

Figura 11 - Exemplo de uso de operadores aritméticos, escrito em *JavaScript*.

```
area = 2 * PI * raio;
delta = b * b - 4 * a * c;
```

Fonte: própria Autora.

Os operadores incrementais são utilizados como "instruções muito comuns chamadas de incremento e decremento. Uma instrução de incremento adiciona uma unidade ao conteúdo de uma variável e o de decremento subtrai. Os operadores são (++) e (--) respectivamente" (ANDRADE, 2017, p. 48) - figura 12.

Figura 12 - Exemplo de uso de operadores incrementais, escrito em *JavaScript*.

```
x++; // x = x + 1
x--; // x = x - 1
```

Fonte: própria Autora.

“Existe um conjunto de operadores relacionais, os quais podem ser usados na avaliação das expressões. Seu objetivo consiste no estabelecimento de relações entre os operandos” (ANDRADE, 2017, p. 48). Normalmente, são operadores já conhecidos pela matemática (figura 13) - maior (>), maior ou igual (>=), menor (<), menor ou igual (<=) - e mais dois exclusivos da programação que podem variar sua sintaxe de acordo com a linguagem - igual (==) e diferente de (!=).

Figura 13 - Exemplo de uso de operadores relacionais, escrito em *JavaScript*.

```
const a = 10; b = 1; c = 12;

a > b + c;
```

Fonte: própria Autora.

Por último, operador lógico é “um tipo de operador binário, ou seja, necessitam de dois valores para geração de um terceiro" (DOUGLAS, 2014, p. 38). Geralmente, são representados por: operador "e" (&&), operador "ou" (||) e operador "não" (! - sendo o único unário). Estes operadores convertem o valor do lado esquerdo para um booleano, para saberem qual valor será retornado dessa operação - figura 14.

Figura 14 - Exemplo de operadores lógicos, escrito em *JavaScript*.

```
var myBreakfast = coffee && milk;
var myLunch = hamburger || pizza;
var myDinner = !chicken || lasagna;
```

Fonte: própria Autora.

2.2.2.7. Estrutura de decisão

Na estrutura de decisão ou estrutura condicional "o programador necessita especificar uma condição que precisa ser avaliada pelo programa, juntamente com as instruções a serem executadas pelo programador" (ANDRADE, 2017, p. 53). São definidas condições que serão avaliadas quanto a sua veracidade

para execução do bloco de código contido dentro da estrutura. Para isso usa-se os operadores relacionais vistos no tópico anterior.

No código exemplo da figura 15, há duas condições sendo verificadas: se *"age"* é maior ou igual que 18. Caso não seja, ele pula para a próxima verificação, onde *"age"* deve ser menor que 18. Se alguma dessas condições forem satisfeitas, apenas o bloco de código que está dentro dessa condição será executado, caso nenhuma seja verdadeira, nada irá acontecer no exemplo abaixo.

Figura 15 - Exemplo de estrutura de decisão, escrito em *JavaScript*.

```
if (age ⩾ 18) {
    return 'You`re under 18!';
} else if (age < 18){
    return 'You`re not under 18!';
}
```

Fonte: própria Autora.

2.2.2.8. Estrutura de repetição

Estrutura de repetição, também conhecida como estrutura de laço ou simplesmente *"loop"* existe "para permitir que uma instrução ou um bloco de instruções seja repetido em um algoritmo computacional" (SOUSA; DIAS; FORMIGA, 2017, p. 42). O programador tem a liberdade de definir um bloco de lógica que será executado em laços quantas vezes for estabelecido pela estrutura que envolve este bloco.

O ponto crucial sobre a estrutura de repetição é que deve-se sempre explicitar uma condição de parada do *"loop",* ou então, o bloco irá se repetir infinitamente. No exemplo da figura 16, a variável *"counter"* é declarada como zero, e a condição de parada é que *"counter"* seja menor que o número cinco, sendo que, a cada iteração (ciclo ou *"loop"*), o *"counter"* soma mais uma unidade no seu valor - utilizando os operadores incrementais para isso.

Figura 16 - Exemplo de estrutura de repetição, escrito em *JavaScript*.

```
for (var counter = 0; counter < 5; counter++) {
    console.log(counter);
}
```

Fonte: própria Autora.

2.2.2.9. Classes e objetos

Para Wassberg (2020, s.p.), na programação orientada a objetos, uma classe é como um projeto, ou descrição, de algo. E para representar algo no programa, o desenvolvedor precisa abstrair a ideia de tal forma que ela se torne genérica.

"A programação orientada a objetos é muito focada em dados [...] Os dados que entram em nossa classe geralmente ditam o tipo de comportamento que desejamos. Esse comportamento geralmente determina as operações que precisamos realizar nos dados" (WASSBERG, 2020, s.p., tradução nossa).

"Um método de classe nada mais é que uma função que pertence a uma classe. O motivo de termos um nome diferente para essas funções é para que possamos distinguir entre uma função que faz parte de uma classe e outra que não é" (WASSBERG, 2020, s.p., tradução nossa).

2.3. Princípios básicos da programação

Existem princípios básicos da programação que foram criados pela comunidade e aceitos por manterem uma escrita de códigos limpos. Aderir a estes princípios provê um código com maior manutenibilidade e performance.

No livro Alcançando um *Software* Maduro, Amoedo (2019, s.p.) cita três desses princípios, destacando seus conceitos e impactos na qualidade do código:

- *Don't Repeat Yourself* (*DRY*): "é um princípio que prega a eliminação de duplicações e que repetições na lógica devem ser eliminadas via abstração e repetições via automatização". (AMOEDO, 2019, s.p.)
- *Keep It Simple, Stupid* (*KISS*): "é um princípio que prega a simplicidade do sistema e que essa simplicidade deve ser o objetivo-chave no *design* e que complexidades desnecessárias devem ser evitadas." (AMOEDO, 2019, s.p.)
- *You Aren't Gonna Need It* (*YAGNT*): "prega fazer a coisa mais simples possível que poderia funcionar. Em muitos casos, desenvolvedores criam códigos "caso a gente precise um dia". Se você não precisa, aquilo não precisa estar lá. Implemente apenas quando realmente precisar daquilo." (AMOEDO, 2019, s.p.)

A adoção destes três princípios garante um caminho mais curto para que os conceitos de código limpo sejam incorporados em todas as linhas de código. Os conceitos apresentados acima são básicos para que o desenvolvedor seja capaz de produzir códigos elegantes e que façam sentido a qualquer um que entrar em contato com ele.

3. LINGUAGEM DE PROGRAMAÇÃO *JAVASCRIPT*

Este capítulo trata da linguagem *JavaScript*, abordando sua história, conceitos da linguagem da programação e apresenta a *library* mais utilizado atualmente, o *React JS*.

3.1. História e evolução da linguagem *JavaScript*

JavaScript é uma linguagem de programação *web* criada por Brendan Eich*,* em 1995. David Flanagan (2020, s.p.) resume a história da criação da linguagem em seu livro, *JavaScript*: o guia definitivo. Brendan trabalhava na empresa *Netscape*, que tinha como empreendimento a exploração da *Web* que estava surgindo na época. Criaram o navegador que se tornou o mais popular na década de 90: *Netscape Navigator*.

A *Netscape* contratou Brendan para que criasse uma linguagem que tornasse a *web* mais dinâmica, pois perceberam que o *Navigator* sempre fazia requisições ao servidor para obter respostas para alimentação do navegador. Após a criação da linguagem, a empresa decidiu normalizar a linguagem através da organização *ECMA International* - companhia especializada em padrões e normativas - para conter possíveis ambições de empresas maiores com o *JS,* que se mostrou efetivo em seus propósitos.

A *ECMA* é mantida pelo Comitê Técnico 39 (TC39), composto por especialistas de empresas como *Microsoft*, *Mozilla* e *Google*, além do próprio Brendan Eich. As versões da linguagem são referidas como *ECMAScript* seguidas do número da versão ou do ano de lançamento desta. A versão mais recente é a *ES2020,* lançada no ano de 2020.

Segundo Flanagan (2020, s.p.), na última década todos os navegadores *web* implementaram a versão 3 do padrão *ECMAScript*, pois não havia preocupação com o número de versões. Em meados de 2011, com o *release* da versão 5, os navegadores trabalharam massivamente para conseguir alterar suas implementações para o conhecido *ES5*.

Em 2015 foi lançado o *ES6*, trazendo muitas mudanças na

arquitetura da linguagem. Ainda nos dias de hoje nem todos *browsers* suportam totalmente versões acima da 6. Como solução, surgiram bibliotecas que conseguem transpilar essas versões para *ES5* - como o *Babel* e *Polyfills*.

Desde então, a linguagem evoluiu de uma forma exponencial e atualmente é utilizada para desenvolver *sites*, aplicações *desktop* (*React JS, Vue JS, Angular JS*) e aplicações *mobile* (*React Native*), além de poder ser utilizada em servidores *backend* (*Node JS*). O uso em peso da linguagem pode ser comprovado pela análise dos resultados da pesquisa realizada pelo *StackOverflow* (2020), onde *JavaScript* aparece com a linguagem mais utilizada em 2020 - pelo oitavo ano consecutivo -, além de dominar as três primeiras posições na categoria de *Web Frameworks* com *jQuery*, *React JS* e *Angular JS*.

A versatilidade e curva rápida de aprendizado do *JavaScript* torna a linguagem popular entre desenvolvedores jovens, e também atrai este público por possibilitar o uso da linguagem junto a paradigmas modernos da programação, aceitando programação estruturada, orientada a objetos e funcional.

3.2. Conceitos do *JavaScript*

Cada linguagem de programação apresenta suas peculiaridades quanto a sintaxe e modo de compilação de código. Neste capítulo serão apresentados pontos onde o *JavaScript* se diferencia do que foi apresentado no capítulo sobre conceitos da programação.

3.2.1. Tipos de dados no *JavaScript*

Programação é basicamente manipulação de dados. No *JavaScript,* dados são chamados de valores e são divididos em dois tipos: primitivos e complexos.

Segundo Flanagan (2020, s.p.), os tipos primitivos da linguagem são: *"string"* (sequência de caracteres/texto)*, "number"* (números)*, "boolean"* (booleanos ou valores de verdade: *true/false*)*, "null"* (nulo) e *"undefined" (indefinido).*

Os tipos complexos, ou de objetos, por sua vez são: *"array"* (vetores), "funções", *"Date"* (data), *"RegExp"* (expressões regulares) e *"Error"*. Apesar da linguagem apresentar tipos de dados ela é fracamente tipada, ou seja, não verifica tipos ao realizar operações, sendo possível em *JS* por exemplo somar uma *"string"* com um *"number".*

3.2.2. Armazenamento de dados em memória no *JavaScript*

Existe uma grande diferença entre os grupos de dados. "Os valores primitivos são imutáveis: não há como alterar (ou "mudar") um valor primitivo" (FLANAGAN, 2020, s.p., tradução nossa), já os valores complexos podem ser alterados. Essa diferença se deve a como o *JavaScript* trabalha com armazenamento de dados em sua memória.

Wassberg (2020, s.p.) diferencia o modo como variáveis podem ser armazenadas na memória, podendo ser armazenada como um valor ou tipo de referência. "Quando é armazenado como um valor, os dados ficam localizados no endereço de memória da variável. Uma variável de referência não salva seu valor

diretamente, mas armazena um endereço onde os dados podem ser encontrados" (WASSBERG, 2020, s.p., tradução nossa).

No código da figura 17, ao declarar a variável *"number1"* com o valor "1", é criado uma chave única de identificação, essa chave é alocada em um endereço de memória onde é armazenado este valor. A partir do momento que o *"number2"* recebe o valor da variável *"number1"* é definido que *"number2"* é igual ao endereço de memória e não realmente igual a variável.

Portanto, quando se muda o valor de *"number1"*, o *JavaScript* cria um novo endereço de memória e armazena o novo valor, isso porque esta variável contém um tipo primitivo, que não permite alteração de seu valor.

Figura 17 - Exemplo de armazenamento de dados no *JavaScript*.

```
var number1 = 1;
var number2 = number1;

number1 = number1 + 1;
```

Fonte: própria Autora.

Já os tipos de dados complexos seguem outro fluxo de armazenamento, eles são mantidos na *memory heap*, ou memória dinâmica. A memória dinâmica é capaz de armazenar dados que crescem dinamicamente, como *"array"* e "objetos".

Sendo assim, no momento em que se declara a variável *"beerList"* (figura 18), o mecanismo do *JavaScript* cria a chave única, aloca o endereço na memória e armazena seu valor na *memory heap*. E quando é alterado o valor deste *"array"*, está sendo alterado o valor apenas na *memory heap*, assim os tipos complexos são consideráveis mutáveis.

Figura 18 - Exemplo de armazenamento de dados em *memory heap.*

```
var beerList = ['IPA', 'APA'];
beerList.append('Stout');
```

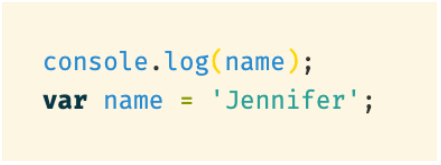
Fonte: própria Autora.

3.2.3. *Hoisting*

Hoisting, ou içamento, é um comportamento onde "o código *JavaScript* se comporta como se todas as declarações de variável em uma função (mas não em qualquer atribuição associada) fossem "içadas" para o topo da função" (FLANAGAN, 2020, s.p., tradução nossa). Em resumo, não importa onde esteja declarada uma função ou variável. Quando o programa compilar, a sua declaração irá para o topo do código.

Isso permite que o desenvolvedor possa utilizar variáveis e funções antes de sua definição, sem causar erros no programa. Na figura 19, o comando *"console.log(name)"* iria retornar o valor *"undefined"*, pois apenas quem sofre *hosting* é a declaração de uma variável e não a sua inicialização - o ato de apontar um valor.

Figura 19 - Exemplo de *hoisting* de uma variável.



Fonte: própria Autora.

O *hoisting* tem como efeito colateral transformar variáveis em tipo global, sendo acessível e modificável para todas as linhas de código. Para evitar este comportamento é recomendado declarar e inicializar variáveis dentro de escopos - de funções, ou blocos.

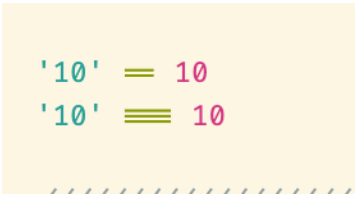
3.2.4. *Triple equals*

Triple equals é simplesmente o seguinte símbolo: "===". Como visto no capítulo sobre conceitos de programação, para atribuir valores usa-se apenas um sinal de igualdade e para comparar igualdade de valores, utiliza-se dois sinais (==). Porém, no *JavaScript* existe uma segunda forma de se comparar valores.

A grande diferença entre um operador e outro é que "== testa igualdade e === testa igualdade de forma restrita" (DOUGLAS, 2014, p. 37). Ou

seja, no primeiro caso da figura 20, compara-se se a *"string"* de valor 10 tem apenas o valor igual ao número 10; porém, na segunda linha, além dessa comparação, o uso de *triple equals* garante que os dois valores comparados são do mesmo tipo - neste caso, espera-se que sejam os dois do tipo *"string"*.

Figura 20 - Exemplo da diferença entre *double* e *triple equals.*



Fonte: própria Autora.

3.2.5. *High order functions (HOF)*

"Funções que operam em outras funções, sejam elas apenas devolvendo argumentos, são chamadas de funções de ordem superior. [...] O termo vem da matemática" (HAVERBEKE, 2018, p. 65, tradução nossa). Simplificando o que Haverbeke definiu, *high order functions* são funções que aceitam funções dentro dela, a linguagem possui várias funções como essas para trabalhar, por exemplo, com *"arrays"*: *"forEach", "filter", "map", "reduce", "some", "every", "find", "findIndex"*.

No exemplo da figura 21, o método *"map"* mapeia o *"array"* e em cada posição aplica o que está escrito dentro da função recebida por parâmetro - chamada de *callback.* O retorno mapeado é armazenado na variável *"numbers"*. Esse conceito de *high order functions* dá um enorme poder de manipulação aos programadores da linguagem.

Figura 21 - Exemplo da de *high order function*, utilizando a função *"map".*

```
const numbers = [1, 4, 6, 18];

numbers.map(number => number * 2);
```

Fonte: própria Autora.

3.2.6. *Document Object Model (DOM)*

"*Document Object Model*, ou *DOM*, é a *API* fundamental para representar e manipular o conteúdo de documentos *HTML* e *XML*" (FLANAGAN, 2020, s.p., tradução nossa).

Haverbeke (2018, p. 162, tradução nossa) simplifica o exemplo do que seria o *DOM* dizendo que quando abrimos uma página *web*, o navegador resgata o texto em *HTML* da página e o interpreta. O *browser* constrói um modelo da estrutura do documento - *DOM* - e depois usa esse modelo para desenhar a página na tela.

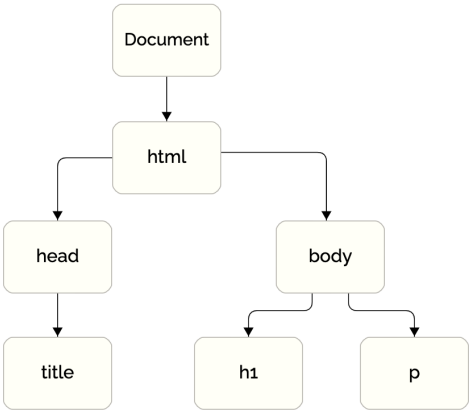
Elementos de um documento *HTML* ou *XML* são representados na *DOM* como uma árvore de objetos (figura 23). A representação em árvore de um documento *HTML* contém nós representando marcações ou elementos *HTML*, conhecidos com tags. Na figura 22 tem-se: *<html>, <head>, <title>, <body>, <h1>* e *<p>.*

Figura 22 - Representação de um documento *HTML*.

```
<html>
  <head>
    <title>Homepage</title>
  </head>
  <body>
    <h1>This is the homepage</h1>
    <p>Hello, I'm Jennifer and this is my homepage</p>
  </body>
</html>
```

Fonte: própria Autora

Figura 23 - Representação da árvore DOM de um HTML.



Fonte: própria Autora.

Utilizando a linguagem *JS*, o desenvolvedor consegue acessar os elementos DOM. Além disso o desenvolvedor tem o poder de "alterá-los, remover alguns deles ou adicionar novos. Atualizar o *DOM* pode fazer com que o navegador repinte a tela e também reflua (recalcule a geometria dos elementos), o que pode ser caro" (STEFANOV, 2010, s.p., tradução nossa). Surgem, assim, problemas quanto à performance na *web*, que serão discutidos mais à frente.

3.3. Conceitos do *JavaScript ES6+*

Este projeto baseia-se em *React JS,* uma *library* moderna de *JavaScript* que usufrui de versões mais recentes da linguagem. Neste capítulo serão explicitados algumas *features* que surgiram na versão *ES6+.*

3.3.1. Declaração de variáveis com: *let* e *const*

No *ES6* foram disponibilizadas mais duas formas de definir variáveis em *JS,* utilizando-se as *keywords: "let"* e *"const"*. A distinção entre eles se fundamenta principalmente na mudança de escopo - "simples termo usado na programação para definir onde você pode chamar a variável ou o código e onde você não pode" (VOICE, 2019, s.p., tradução nossa). Utilizando o *"var"* a variável é disponível em escopo global, podendo ser modificada a qualquer momento por qualquer linha do código.

"*JavaScript* tem escopo de variável léxica [...] criamos blocos de código com chaves ({}). Em funções, essas chaves bloqueiam o escopo de qualquer variável declarada com *var*. Porém o comportamento não é o mesmo com instruções *if/else*" (PORCELLO e BANKS, 2020, s.p., tradução nossa), o uso do *"let"* resolve este problema.

No exemplo da figura 24, a variável *"hasLegalAge"* foi declarada apenas no escopo da estrutura de repetição, não sendo retornada para fora do escopo. Ou seja, ao tentarmos acessá-la fora do escopo, isto ocasionará um erro de não existência da variável - o editor *VSCode* difere a variável com uma cor de texto meio apagada, indicando que ela foi declarada, porém não está sendo utilizada.

Figura 24 - Exemplo de uso de variável utilizando *"let"*.

```
if (age > 18) {
    let hasLegalAge = true;
}

console.log(hasLegalAge)
```

Fonte: própria Autora.

A declaração de variável utilizando *"const"* - o mais recomendado na maioria das situações -, "consegue fazer tudo que o *let* faz, porém é imutável [...] variáveis e funções declaradas com *const* não podem ser re-atribuídas" (VOICE, 2019, s.p., tradução nossa).

Há uma confusão de conceito quando se trata de manipulação de "objetos" e *"arrays"* declarados como constante. Na figura 25, a variável *"subjects"* é um *"array"* e o programador pode entender que não é possível adicionar mais valores a essa lista, pois é imutável. Porém, esta premissa está errada, como visto no capítulo sobre como funciona armazenamento de dados dentro do *JavaScript*, o *"array"* é do tipo complexo, portanto alterar o valor dele é alterar sua referência na memória dinâmica.

Figura 25 - Exemplo de uso de variável utilizando *"const"* e *"array"*.

```
const subjects = ['POO', 'Web Programming I'];

subjects.push('Web Programming II');
```

Fonte: própria Autora.

3.3.2. *Template literal string*

Segundo Voice (2019, s.p., tradução nossa), as versões anteriores ao *ES6* apresentavam problemas com relação a manipulação de *"strings"*, como:

- A única forma de concatenar *"strings"* era utilizar o símbolo de adição entre elas.
- Não havia uma boa forma de escrever múltiplas linhas de textos.
- Ainda utilizavam-se os chamados *escape quotation*, no *JS* representado pelo símbolo "\". O escape é usado para permitir a utilização de caracteres que o navegador normalmente não reconhece.

"*Template literal string* nos fornece uma alternativa para a concatenação de *string*. Eles também nos permitem inserir variáveis em uma *string*" (PORCELLO e BANKS, 2020, s.p., tradução nossa). Apenas ao adicionar a *"string"*

dentro de duas crases ``, e os símbolos "${}" para poder utilizar variáveis *JS*, além de conseguir realizar comandos, operações e funções. Observe o código da figura 26, onde refatorou-se a mesma função mostrada anteriormente neste trabalho.

Figura 26 - Exemplo de uso de *template literal string*.

```
/**
 * Sets a personalized greeting, according to the name
 * @param { String } name person's name
 * @returns { String } personalized greeting
 */
function setGreetings (name) {
    return `Hi, my name if ${name}`;
}
```

Fonte: própria Autora.

3.3.3. *Default value*

"Linguagens como *C++* e *Python* permitem que os desenvolvedores declarem *default value* para argumentos de uma função. *Default value* foi incluído no *ES6*. No caso de um valor não ser fornecido para o argumento, o *default value* será usado" (PORCELLO e BANKS, 2020, s.p., tradução nossa).

Na figura 27, se, ao utilizar a função *"setGreetings",* o desenvolvedor não passar nenhum valor para o argumento *"name"*, este assumirá o valor "Jennifer" ao invés de *"undefined"*.

Figura 27 - Exemplo de uso de *default value* em função.

```
function setGreetings (name = 'Jennifer') {
    return `Hi, my name if ${name}`;
}
```

Fonte: própria Autora.

3.3.4. *Arrow functions*

No *ES6* foi apresentada uma nova forma de definir funções,

utilizando a sintaxe de *arrow function.* "Esta sintaxe é uma reminiscência da notação matemática e usa uma => "seta" para separar os parâmetros da função do corpo da função. A palavra-chave *function* não é usada e [...] também não há necessidade de um nome de função" (FLANAGAN, 2020, s.p., tradução nossa).

"As *arrow function* se diferem das funções tradicionais de uma maneira crítica: elas herdam o valor da *keyword this* do ambiente no qual são definidas, em vez de definir seu próprio contexto de invocação" (FLANAGAN, 2020, s.p., tradução nossa). Na figura 28 mostra-se como utilizar *arrow funtion* é menos verboso que uma função regular. "Temos uma declaração de função inteira em uma linha [...] Também removemos o retorno porque a seta aponta para o que deve ser retornado. Outro benefício é que, se a função receber apenas um argumento, podemos remover os parênteses ao redor dos argumentos" (PORCELLO e BANKS, 2020, s.p., tradução nossa).

Figura 28 - Exemplo de uso de *arrow function*.

```
const setGreetings = (name) => `Hi, ${name}`;
setGreetings('Jennifer')
```

Fonte: própria Autora.

3.3.5. *Destructuring*

"*ES6* veio com um jeito mais simples de extrair variáveis de objetos e *arrays*, *Destructuring*" (VOICE, 2019, s.p., tradução nossa). Anteriormente, para acessar propriedades de um objeto tinham-se duas opções: *dot notation* - *"object.property"* -, ou *square-bracket notation* - *"object[‘property’]"*. Apesar destas opções serem simples, causavam um certo transtorno em casos de muitos níveis de acesso, além do desenvolvedor ter sempre que manter consistência em seu código ao testar se a propriedade anterior existe antes de acessar a posterior.

Figura 29 - Exemplo de uso de *destructuring* em objetos.

```
const person = {
    name: 'Jennifer',
    isDeveloper: true,
    hobbies = ['code', 'tv_shows', 'beer'],
    languages: {
        JavaScript: 5,
        Go: 3,
        Java: 1,
    }
}

const { name } = person;
const { isDeveloper = false } = person;
const { hobbies = ['freeTime'] } = person;
const { languages = { Go: Golang = 4 } = {} } = person;
```

Fonte: própria Autora.

No exemplo da figura 29, temos um *"*objeto *person"* com várias propriedades. Para acessar uma propriedade usando *destructuring*, usa-se a conotação de "{}" para indicar que está extraindo-se a propriedade dentro das chaves de dentro de uma variável. Apenas ao colocar um "= valor", consegue-se configurar um valor padrão no caso de a propriedade não existir dentro do objeto.

Também há como renomear a propriedade no momento da extração, a fim de manter algum padrão de código. No exemplo, renomeia-se *"Go"* para *"Golang"*. Na última linha do código, há um acesso de dois níveis do objeto, e para garantir que não haverá quebra caso o objeto seja *"undefined"*, tem o valor padrão de *"*objeto vazio*"* configurado. Com uma simples linha foi evitado o uso de pelo menos dois *"if/else"* no projeto.

Com a mesma facilidade é possível desestruturar valores em posições do *"array"*, como no exemplo da figura 30, onde é passado entre colchetes os nomes de variável para cada valor que se está extraindo do objeto, e caso tenha menos posições no *"array"* do que variáveis do lado esquerdo, o código não quebrará, apenas retornará *"undefined"* para essas variáveis - e consegue-se definir um valor *default* para ele.

"Também podemos ignorar valores desnecessários com *list matching* usando vírgulas. A *list matching* ocorre quando as vírgulas ocupam o lugar dos elementos que devem ser ignorados" (PORCELLO e BANKS, 2020, s.p., tradução nossa).

Figura 30 - Exemplo de uso de *destructuring* em *"array".*

```
const names = ['Jennifer', 'Andy', 'Chewie'];

const [name1, name2, name3, name4, name5 = 'Woody'] = names;
```

Fonte: própria Autora.

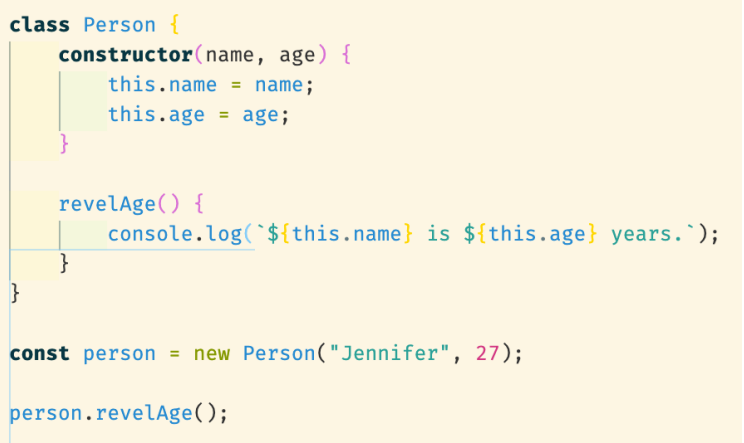
3.3.6. Classes

"Antes do *ES2015*, não havia sintaxe de classe oficial na especificação *JavaScript* [...] a biblioteca *React* se apoiou fortemente em classes para construir componentes de interface do usuário" (PORCELLO e BANKS, 2020, s.p., tradução nossa). Atualmente, o *React* se baseia em funções para construir componentes ao invés de classes.

Porcello e Banks (2020, s.p.) elucidam que a ideia de classes funciona da mesma forma no *ES6+*. A nova versão apenas introduziu uma forma de declaração de classe mais simples.

O código da figura 31 cria uma classe chamada *"Person"* que recebe *"name"* e *"age"* e detém um método chamado *"revelAge".* Para instanciar a classe usa-se a *keyword "new".*

Figura 31 - Exemplo de uso de classe.



```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    revelAge() {
        console.log(`${this.name} is ${this.age} years.`);
    }
}

const person = new Person("Jennifer", 27);

person.revelAge();
```

Fonte: própria Autora.

3.3.7. *ES6 Modules*

"Um módulo *JavaScript* é um pedaço de código reutilizável que pode ser facilmente incorporado a outros arquivos *JavaScript* sem causar colisões de variáveis. Módulos *JavaScript* são armazenados em arquivos separados, um arquivo por módulo" (PORCELLO e BANKS, 2020, s.p., tradução nossa).

Para importar e exportar módulos, existem duas opções: exportar vários objetos *JavaScript* de um único módulo ou apenas um objeto *JavaScript* por módulo. Com base nas definições de Porcello e Banks (2020, s.p.), a figura 32 exemplifica as maneiras de se exportar e importar arquivos.

Pode-se usar para exportar qualquer tipo de *JavaScript* que será consumido em outro módulo. Também pode-se exportar uma única variável principal com o uso das *keywords "export default"*

"Os módulos podem ser consumidos em outros arquivos *JavaScript* usando *import*. Módulos com múltiplas exportações podem utilizar a desestruturação de objetos. Os módulos que usam o padrão de exportação são importados para uma única variável" (PORCELLO e BANKS, 2020, s.p., tradução nossa).

Figura 32 - Exemplo de uso de *ES6 modules*.

```
export const sumAge = (age1, age2) => age1 + age2;
export const multiplyAge = (age1, age2) => age1 * age2;

export default new Person('Jennifer', 27);

////////

import { sumAge, multiplyAge as m} from "./tcc-example";
import example from "./example1";
import * as tcc from './example2';

sumAge(27, 28);
m(28, 28);

example.print();

tcc.Foo();
```

Fonte: própria Autora.

"As sintaxes *import* e *export* ainda não são totalmente suportadas por todos os *browsers* ou pelo *Node*. No entanto [...] é compatível com *Babel*. Isso significa que você pode utilizar essas *features* e o *Babel* saberá onde encontrar os módulos" (PORCELLO e BANKS, 2020, s.p., tradução nossa).

Porém, existe um módulo padrão suportado em todas as versões no *Node*, chamado *CommonJS*. Neste módulo, os objetos *JavaScript* são exportados usando *"module.exports"*.

3.3.8. Compilando o *JavaScript ES6+*

Quando um novo recurso *JavaScript* é proposto e ganha suporte não é imediata a adoção em projetos, pois não há certeza de que os navegadores podem suportar as novas alterações. "A única maneira de ter certeza de que o código funcionará é convertê-lo em um código compatível antes de executá-lo no navegador. Este processo é chamado de compilação" (PORCELLO e BANKS, 2020, s.p., tradução nossa). Uma das ferramentas mais populares para compilação de *JavaScript* é o *Babel*.

"O *Babel* tornou possível usar os recursos mais recentes do

JavaScript imediatamente [...] nosso código não é compilado para binário. Em vez disso, é transformado em sintaxe que pode ser interpretada por uma ampla gama de navegadores" (PORCELLO e BANKS, 2020, s.p., tradução nossa).

O processo de compilação de *JavaScript* é normalmente automatizado por uma ferramenta de construção como *Webpack* ou *Parcel*.

3.4. *Library React JS*

"O *React* é mantido por desenvolvedores no *Facebook* e por colaboradores da comunidade *JS* [...] é uma das bibliotecas mais populares para a criação de interfaces de usuário e conhecida por ser rápida, graças à sua maneira inteligente de manipular o *DOM*" (ROLDAN, 2019, s.p., tradução nossa).

Porcello e Banks (2020, s.p.) esclarecem a origem da biblioteca (*library*): ela foi criada por Jordan Walke, um engenheiro de *software* do *Facebook* e incorporado à *newsfeed* do *Facebook* em 2011. Em 2012, foi adicionada ao *Instagram*. No evento *JSConf*, em 2013, o *React* tornou-se *open source* e se juntou às bibliotecas de *UI* (*user interface)* como *jQuery, Angular, Dojo, Meteor*.

> A partir daí, a adoção pela comunidade começou a se espalhar. Em janeiro de 2015, a *Netflix* anunciou que estava usando o *React* para potencializar o desenvolvimento da interface do usuário. Mais tarde naquele mês, *React Native*, uma biblioteca para construir aplicativos móveis usando *React*, foi lançada. O *Facebook* também lançou o *ReactVR*, outra ferramenta que levou o *React* a uma gama mais ampla de alvos de renderização. Em 2015 e 2016, um grande número de ferramentas populares como *React Router, Redux* e *Mobx* entraram em cena para lidar com tarefas como roteamento e gerenciamento de estado. Afinal, o *React* foi anunciado como uma biblioteca: preocupada em implementar um conjunto específico de recursos, não fornecendo uma ferramenta para cada caso de uso.
>
> Outro grande evento na linha do tempo foi o lançamento do *React Fiber* em 2017. O *Fiber* foi uma reescrita do algoritmo de renderização do *React* que foi meio mágico em sua execução. Foi uma reescrita completa da parte interna do *React* que quase nada mudou na *API* pública. Foi uma forma de tornar o *React* mais moderno e eficiente, sem afetar seus usuários.
>
> Mais recentemente, em 2019, vimos o lançamento de *Hooks*, uma nova maneira de adicionar e compartilhar lógica com estado entre componentes. Também vimos o lançamento do *Suspense*, uma forma de otimizar a renderização assíncrona com o *React* (PORCELLO e BANKERS, 2020, s.p. tradução nossa).

"*React* é mais corretamente chamado de *library* do que de *framework* porque ele não inclui tudo que você precisa para desenvolver seu

aplicativo. No entanto, todos os pacotes necessários estão disponíveis para serem adicionados" (KEKERI, 2018, s.p., tradução nossa).

3.4.1. Funcionamento do React

Porcello e Banks (2020, s.p.) traçam a evolução e diferença da maneira como o *React* renderiza informações em tela. Tradicionalmente o navegador solicitava e carregava documentos *HTML* a cada navegação do usuário entre páginas. Com a invenção do *AJAX* (*Asynchronous JavaScript and XML*) surgiu a ideia de *Single Page Application* (*SPA*), onde navegadores podem solicitar e carregar pequenos *bits* de dados usando *AJAX*.

> Em um *SPA*, o navegador carrega inicialmente um documento *HTML*. Conforme os usuários navegam pelo site, eles realmente permanecem na mesma página. *JavaScript* destrói e cria uma nova interface de usuário à medida que o usuário interage com o aplicativo. Pode parecer que você está pulando de uma página para outra, mas na verdade ainda está na mesma página *HTML* e o *JavaScript* está fazendo o trabalho pesado (PORCELLO e BANKS, 2020, s.p., tradução nossa).

"React é uma biblioteca projetada para atualizar o *DOM* do navegador [...], não interagimos diretamente com a *API DOM*. Em vez disso, fornecemos instruções sobre o que queremos que o *React* construa" (PORCELLO e BANKS, 2020, s.p.).

3.4.2. *React Elements*

O *DOM* do navegador é composto de elementos *DOM*. Da mesma forma, o *React DOM* é composto de elementos *React*. "Um elemento *React* é uma descrição de como o elemento *DOM* real deve se parecer. Em outras palavras, os elementos *React* são as instruções de como o *DOM* do navegador deve ser criado" (PORCELLO e BANKS, 2020, s.p.).

Para criar um elemento do *React*, usa-se a função nativa da biblioteca: *"React.createElement"*, onde o primeiro argumento define o tipo de

elemento, o segundo representa as propriedades do elemento (como *id,* classe), e a terceira representa os filhos do elemento. Durante a renderização, o *React* converterá este elemento em um elemento *DOM* real, como visto na figura 33.

Figura 33 - Exemplo de criação de um elemento *React* usando *"React.createElement".*

```
React.createElement("h1", { id: "title" }, "TCC Title");

<h1 id="title">TCC Title</h1>
```

Fonte: própria Autora.

"Depois de criar um elemento *React*, vamos querer vê-lo no navegador. *ReactDOM* contém as ferramentas necessárias para renderizar os elementos *React* no navegador" (PORCELLO e BANKS, 2020, s.p.).

Ao criar um *React Element*, utiliza-se a função *"ReactDOM.render"* para renderizar o elemento no navegador, onde o elemento a ser renderizado é passado como o primeiro argumento e o nó de destino é passado como segundo argumento. Na figura 34, o resultado da renderização no navegador seria a inserção de uma *"div"* com o identificador *"root"* e dentro desta estaria o elemento criado via *React*.

Figura 34 - Exemplo de renderização de elementos usando *ReactDOM.*

```
const title = React.createElement("h1", null, "TCC Title");

ReactDOM.render(title, document.getElementById("root"));

<body>
    <div id="root">
        <h1>TCC Title</h1>
    </div>
</body>
```

Fonte: própria Autora.

"Quando o tipo do elemento é uma função, o *React* o chama [...] e continua chamando recursivamente até obter uma árvore *DOM*. Esse processo é chamado de reconciliação e é usado pelo *React DOM* e pelo *React Native*"

(ROLDAN, 2019, s.p., tradução nossa).

3.4.3. *JSX*

"O *React* oferece duas maneiras de definir nossos elementos. O primeiro é usando funções *JavaScript* e o segundo é usando *JSX*, uma sintaxe semelhante a *XML* opcional" (ROLDAN, 2019, s.p., tradução nossa).

"Em *JSX*, o tipo de um elemento é especificado com uma *tag*. Os atributos da *tag* representam as propriedades. Os filhos do elemento podem ser adicionados entre as *tags* de abertura e fechamento" (PORCELLO e BANKS, 2020, s.p.).

Porcello e Banks (2020, s.p.) listam cinco dicas para que o desenvolvedor se adapte ao JSX e utilize a ferramenta da melhor maneira possível:

1. Componentes aninhados: o *JSX* permite adicionar componentes como filhos de outros componentes - na figura 32 o componente <*List*> consegue comportar vários componentes <*Item*>.

2. *ClassName*: como a palavra *"class"* é uma palavra reservada em *JavaScript*, "className" é utilizado para definir o atributo *class* (figura 32).

3. Expressões *JavaScript:* pode-se utilizar *JS* ao adicionar chaves, indicando onde as variáveis serão avaliadas e seus valores resultantes retornados - na figura 31, a variável *"title"* pode ser inserida diretamente entre a *tag* <*h1*> e ser renderizada a partir do uso de chaves, indicando a expressão *JS*.

4. *Evaluation*: o código *JS* adicionado entre as chaves será lido, ou seja, operações como concatenação ou adição serão executadas, assim como as funções encontradas em expressões *JavaScript* serão invocadas - ao ser identificada a expressão *JS* pode-se adicionar métodos da linguagem, como na figura 32, onde é aplicada uma função para transformar todos os caracteres em letras minúsculas no momento da renderização.

5. Mapeando arrays: pode-se incorporar *JSX* diretamente nas funções *JavaScript* - na figura 35, ao percorrer o array *"items"*, consegue-se renderizar cada elemento dentro da lista. Um ponto importante é a inserção do atributo *"key"* dentro do elemento pai de dentro da lista, para que o *React* consiga

encontrar rapidamente cada elemento.

Figura 35 - Exemplo de renderização usando *JSX*.

```
<List>
  <Item />
  <Item />
</List>

<h1 className="title">TCC Title</h1>

<h1>{title}</h1>

<h1>{title.toLowerCase()}</h1>

<ul>
  {props.items.map((item, i) => (
    <li key="{i}">{item}</li>
  ))}
</ul>
```

Fonte: própria Autora.

3.4.4. Componentes

A ideia principal por trás de trabalhar com *React* é que tudo é um componente. Os componentes renderizados geram componentes *HTML* como visto anteriormente, "os dados vêm para o *HTML* de fora por meio de *props* (propriedades) e são mantidos internamente pelo *state* (estado). Sempre que houver uma mudança nas *props* ou *state*, o *React* se encarrega de atualizar o *HTML*" (KEKERI, 2018, s.p., tradução nossa).

Kekeri (2018, s.p.) define que um bom princípio de *design* é que cada componente seja responsável por uma única tarefa e, se o componente precisa fazer mais, ele deve ser decomposto em componentes menores. Na figura 36, o componente *"Button"* tem apenas a responsabilidade de renderizar uma *tag* <*button*> recebendo algumas propriedades para personalizar o elemento.

Figura 36 - Exemplo de criação de componente.

```
import React from 'react';

import './styles.css';

function Button({ type, onClick, children }) {
  return (
    <button type={type} onClick={onClick}>{children }</button>
  );
}

export default Button;
```

Fonte: própria Autora.

3.4.5. *Props*

"*JSX* é muito conveniente quando seus elementos *DOM* ou componentes *React* têm *props*. Usando XML é muito fácil de definir atributos em elementos" (ROLDA, 2019, s.p., tradução nossa).

Kekeri (2018, s.p.) simplifica a análise de quando utilizar *props* ou *state*: "se dois (ou mais) componentes compartilham dados (ou se um componente produz dados de que outro componente precisa). Deve pertencer a um componente superior, que o transmitirá conforme necessário: os dados fluem para baixo" (KEKERI, 2018, s.p., tradução nossa) - na figura 33, o componente recebe *props* de seu componente pai que são utilizadas para renderizar corretamente o elemento.

3.4.6. *Children*

"*JSX* permite definir *children* para descrever a árvore de elementos e compor interfaces de usuário complexas [...] não estamos limitados a ter apenas ter elementos como *children*, podemos usar expressões *JavaScript*, como funções ou variáveis" (ROLDAN, 2019, s.p., tradução nossa).

Na figura 37, o elemento tem *children* que incluem uma variável (*user*) e também uma função, já que tudo é possível ao se utilizar as chaves.

Figura 37 - Exemplo de renderização de *children* no *JSX*.

```
<div>
  Hello, {user}.
  I'm a {() ⇒ console.log('Function')}.
</div>
```

Fonte: própria Autora.

4. APLICABILIDADE DO CÓDIGO LIMPO

4.1. Código Limpo

"Aprender a criar códigos limpos é uma tarefa árdua e requer mais do que o simples conhecimento dos princípios e padrões. Você deve suar a camisa; praticar sozinho e ver que cometeu erros; assistir os outros praticarem e errarem." (MARTIN, 2012, p. 8). O conceito de código limpo surgiu de maneira mais clara após a publicação da obra de Robert C. Martin em 2008 - Código Limpo: Habilidades Práticas do *Agile Software*. Este livro serviu de base para compor padrões de várias linguagens, como Padolsey (2020, s.p.) em sua obra *Clean Code in JavaScript* e Stefanov (2010, s.p.) ao publicar *JavaScript Patterns*.

"O código que escrevemos é principalmente para pessoas [...] Todas essas coisas são para pessoas: pessoas que são os usuários finais [...] ou pessoas que são os programadores que fazem uso de nossas abstrações e interfaces" (PADOLSEY, 2020, s.p., tradução nossa).

Raphael Amoedo (2019, s.p.) define *clean code* em seu livro Alcançando um *Software* Maduro como o conceito dado quando o código é bem escrito de forma que você leia o código e entenda exatamente o que está sendo feito, e que cada função tenha sua própria responsabilidade.

Martin (2012, p. 17) expõe o que alguns programadores famosos disseram sobre qualidade do código ou código limpo. Bjarne Stroustrup, o pai da linguagem de programação *C++*, diz que não se deve utilizar de rodeios, pois estes escondem problemas e dificultam alterações. Os erros devem ser tratados estrategicamente, evitando que sejam criadas rotinas que prejudiquem o desempenho do programa. Sua abordagem crê na beleza e na efetividade do código.

Concordando com ele, Grady Booch (*apud* MARTIN, 2012, p. 18), criador das raízes do projeto orientado a objetos, compara um código a uma prosa, pois a objetividade simples e direta criada pelo programador torna sua leitura facilmente compreensível.

Martin (2012, p. 16) define que a prática de escrever um código

bom não é tão complexa como podemos imaginar. Porém existe um tempo de aprendizagem. Afinal é um hábito e deve ser melhorado a cada dia.

Por fim, Lewis (2020, s.p.) esclarece o porquê de se preocupar com a escrita e manutenção de códigos limpos:

> Quando você escreve um código limpo, você dá chances de você e outros leitores conseguirem compreender seu código. Você reduz os custos de manutenção do código e da aplicação onde o código será utilizado. Você facilita a estimativa de tempo para implementação de novas funcionalidades. Qualquer problema ou *bug* será facilmente consertado. Quando é fácil de entender um código, é fácil de usar este código em outras aplicações e ferramentas. (LEWIS, 2020, s.p., tradução nossa).

4.2. Princípios do Código Limpo

Padolsey (2020, s.p.) elenca quatro princípios do código limpo: para o autor, classifica-se um software como bom se ele tiver todas essas qualidades e ruim caso nao tenha nenhuma delas. "Crucialmente, no entanto, esses princípios não são regras. Em vez disso, é útil vê-los como lentes através das quais você pode olhar seu código" (PADOLSEY, 2020, s.p., tradução nossa).

São os princípios:

1. Confiabilidade: "é a base fundamental de um bom sistema de *software*. Todo o propósito da tecnologia pode ser prejudicado pela falta de confiabilidade [...] pode-se definir em três qualidades distintas: [...] qualidade de ser correto, estável e resiliente" (PADOLSEY, 2020, s.p., tradução nossa).

2. Eficiência: "os recursos são finitos. Para escrever o melhor código possível, precisamos levar essa escassez em consideração. Portanto, ao projetar e implementar nossas ideias, devemos fazê-lo visando a eficiência" (PADOLSEY, 2020, s.p., tradução nossa).

3. Manutenibilidade: é a facilidade com que as alterações apropriadas podem ser feitas em seu código. "Tornar o código sustentável não deve ser uma prioridade marginal. É tão vital quanto qualquer outro requisito que o código esteja cumprindo" (PADOLSEY, 2020, s.p., tradução nossa).

4. Usabilidade: é tornar o código o mais útil e fácil de usar. "Usabilidade não é apenas atendimento aos requisitos do usuário; trata-se de criar experiências que permitam ao usuário atingir seus objetivos com o mínimo de aborrecimento, tempo e esforço cognitivo" (PADOLSEY, 2020, s.p., tradução nossa).

4.3. Práticas de código limpo

"Cada programador olhará para o código limpo de uma maneira diferente porque é subjetivo do ponto de vista de cada um. Alguns conceitos são considerados como as melhores práticas, enquanto outros são baseados na comunidade e no mercado." (LEWIS, 2020, s.p., tradução nossa)

Algumas práticas de código limpo relatadas nas bibliografias dos autores Martin (2012), Amoedo (2019), Wassberg (2020) e Lewis (2020) foram compiladas com o objetivo de sintetizar algumas diretrizes de código limpo e solidificar uma base para avançar no código limpo voltado a *JavaScript* e *React JS* nos próximos capítulos:

- Nomear variáveis de forma concisa: de acordo com Robert Martin (2012, p. 12), o nome de uma variável, método ou classe deveria responder a todas as questões acerca do elemento sendo nomeado. Não é recomendável nomear variáveis com nomes genéricos ou que tenham um sentido dúbio sobre do que trata o seu conteúdo. O nome da variável por si só deve conseguir contar o motivo de sua existência e o que ela guarda.

Na figura 38, o nome *"emailInput"* deixa claro que se trata de um elemento do tipo input no qual está sendo pego o e-mail do próprio *DOM*. Caso fosse utilizada a variável como *"element"*, seria genérica e a função demandaria mais tempo para ser entendida.

Figura 38 - Função exemplo: boa prática para nomeação de variáveis.

```
/**
 * Get input elements and add a change listener
 * that updates user email
 */
function watchEmailInput() {
    const emailInput = document.querySelectorAll(
        'input[name="login[username]"], input[name="username"]'
    );

    emailInput.forEach(input => {
        input.addEventListener('change', () => {
            updateUserEmail(input.value)
        })
    });
}
```

Fonte: própria Autora.

- Não ter código *hardcoded*: os conhecidos números mágicos, dados fixos no meio do código que o próximo programador pode não saber o motivo de estar ali. O melhor é mantê-lo em uma variável que faça sentido.

Na figura 39 a informação sobre *"monthBonus"* faz total sentido ao ser mantida em uma variável que explique que o valor de 100 tem a finalidade de calcular o salário. Assim, se daqui a algum tempo essa informação mudar, o programador consegue mudar apenas o conteúdo da variável, sem ter que inferir informações da função.

Figura 39 - Função exemplo: boa prática em manter informações fixas em variáveis.

```
/**
 * Gets the salary
 *
 * @param { Number } workedHours employer's worked hours
 * @param { Number } baseSalary employer's base salary
 * @returns { Number } month's salary
 */
function getSalary(workedHours, baseSalary) {
    // Month's bonus is a fixed value, changes each year
    const MONTH_BONUS = 100
    return (workedHours * baseSalary) + MONTH_BONUS - calculateTaxes();
}
```

Fonte: própria Autora.

- Desacoplar funções para que tenham responsabilidades únicas: simplifique funções para que façam apenas o que devem fazer, desacoplando em funções menores e mais fáceis de serem entendidas. "Funções devem fazer apenas uma coisa. Devem fazê-la bem. Devem fazer apenas ela." (MARTIN, 2012, p. 44).

Observando o exemplo abaixo - figura 40 - a função que calcula o salário mantém como responsabilidade única multiplicar os valores: horas trabalhadas e o salário base. Caso queira retirar alguns valores como imposto, a melhor estratégia seria criar outra função para executar essas ações e apenas chamar o método no retorno da *"getSalary"*, mantendo-se, assim, cada responsabilidade separada.

Figura 40 - Função exemplo: boa prática de desacoplamento de responsabilidade por funções.

```
/**
 * Calculates taxes
 *
 * @param { Number } baseTax tax's percent base
 * @param { Number } baseSalary employer's base salary
 * @returns { Number } calculate taxes
 */
function calculateTaxes(baseTax, baseSalary) {
    return baseTax * baseSalary;
}

/**
 * Gets the salary
 *
 * @param { Number } workedHours employer's worked hours
 * @param { Number } baseSalary employer's base salary
 * @returns { Number } month's salary
 */
function getSalary(workedHours, baseSalary) {
    // Month's bonus is a fixed value, changes each year
    const MONTH_BONUS = 100
    return (workedHours * baseSalary) + MONTH_BONUS - calculateTaxes();
}
```

Fonte: própria Autora.

- Diminuir códigos aninhados: "às vezes, é tentador colocar várias instruções *if* ou *loops* dentro umas das outras. Mas muitos níveis de instruções *if* aninhadas ou *loops* for podem torná-los difíceis de ler e compreender" (WASSBERG, 2020, s.p., tradução nossa).

Na figura 41, ao invés de escrever operadores condicionais para processar as operações matemáticas, é mais claro e limpo acoplar em um objeto a chamada das funções com cada operação e chamar essa propriedade passando os valores a serem processados.

Figura 41 - Função exemplo: solução de códigos com muitas condicionais.

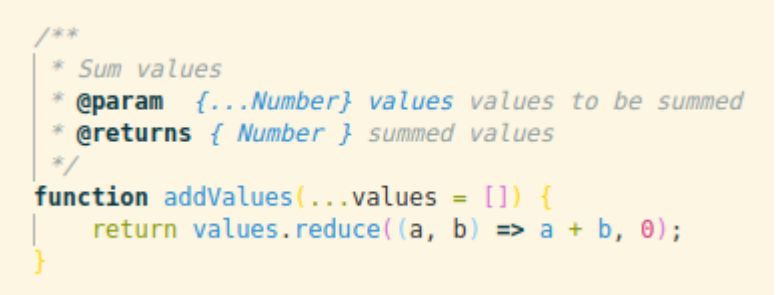
```
const mathOperations = {
    addition: addValues( ... values),
    subtraction: subtractValues( ... values),
    multiplication: multiplyValues( ... values),
    division: divideValues( ... values)
};

mathOperations.addValues(5, 3)
```

Fonte: própria Autora.

- Evitar códigos duplicados e redundantes: "você deve sempre certificar-se de remover o código redundante. Código redundante é o código que não afeta a saída da aplicação, mas será executado" (WASSBERG, 2020, s.p., tradução nossa).

Ao criar uma função genérica para somar quantos valores forem necessários (figura 42) em qualquer momento do código, consegue-se evitar a duplicidade de blocos de código e reusabilidade do mesmo.

Figura 42 - Função exemplo: criação de função genérica para evitar códigos duplicados.



Fonte: própria Autora.

- Evitar comentários em excesso em códigos: "comentar sobre linhas de código complicadas é uma ótima maneira de documentar. Mas os comentários também podem tornar o código menos legível. Nunca comente sobre coisas que são óbvias" (WASSBERG, 2020, s.p., tradução nossa).

"Códigos mudam e evoluem. Movem-se blocos para lá e para cá [...]. Infelizmente, os comentários nem sempre os seguem - nem sempre é possível" (MARTIN, 2012, p. 60).

Figura 43 - Função exemplo: comentário contendo regra de negócio.

Fonte: própria Autora.

- Documentar o código: a documentação de código é uma prática bem interessante para que o processo de repasse de projeto para outros desenvolvedores seja mais orgânico. Em *JavaScript,* a ferramenta *JSDocs* é

bastante utilizada. Ela mantém padrões de como escrever e separar a documentação.

No exemplo abaixo - figura 44 - conseguimos entender o que a função faz, quais parâmetros e seus tipos a função recebe, e qual o retorno dessa função.

Figura 44 - Função exemplo: boa prática de documentação de código.

Fonte: própria Autora.

- Manter indentação no projeto: "você pode precisar escrever algum código dentro de outro código. Isso significa que o código está aninhado. É levemente complexo trabalhar com esse tipo de estrutura caso você não escreva bem o seu código." (LEWIS, 2020, s.p., tradução nossa).

Indentar é separar o código por blocos - apertando a tecla *tab* duas vezes ou apertando a tecla *space* quatro vezes antes de começar a linha -, é um método para facilitar o entendimento de quando finaliza-se um bloco ou escopo, podendo ser um método, uma condição, uma classe ou um *loop*.

- Formatação vertical e horizontal: segundo Martin (2012, p. 87), funções devem ser declaradas o mais próximo possível de sua chamada, e deve-se manter um comprimento máximo de carácter por linhas: 100 a 120.

4.3.1. Padrões de código e boas práticas em *JavaScript*

"No desenvolvimento de *software*, um padrão é uma solução para um problema comum [...] uma prática recomendada, uma abstração útil e um modelo para resolver categorias de problemas" (STEFANOV, 2010, s.p., tradução nossa).

Não existe um padrão de código oficial para a linguagem de

programação *JavaScript*, porém é de extrema importância que a empresa defina junto a seus times de desenvolvimento um padrão para que o código mantenha uma uniformidade e para que se consiga fazer uso de uma ferramenta como o *ESLint*.

Cada desenvolvedor programa de uma forma tão única que é como se o código fosse uma impressão digital, sendo comprovado por um trecho extraído da revista Exame:

> Segundo uma pesquisa elaborada por especialistas das Universidades Drexel, de Maryland, de Goettingen e Princeton, a forma com que códigos são escritos é praticamente uma impressão digital que pode inclusive ser usada para identificar quem é o responsável pelo arquivo. (GUSMÃO, 2015, s.p.)

Padrões de codificação são "padrões específicos de *JavaScript* e boas práticas relacionadas aos recursos exclusivos da linguagem, como os vários usos de funções" (STEFANOV, 2010, s.p., tradução nossa).

Stefanov (2010, s.p.) e Padolsey (2020, s.p.) listaram padrões de codificação e boas práticas para a linguagem *JavaScript* em suas obras que, de acordo com o site oficial do *ESLint* (2021), podem ser adicionadas como regras para verificação estática de código pela ferramenta. Também seguem trechos explicando uso de alguns padrões retirados das bibliografias de Porcello e Banks (2020, s.p.) e de Flanagan (2020, s.p.).

Foram selecionados padrões utilizados pelos *styles guide* populares que se entende como pertinentes para serem utilizados juntamente ao *React JS*.

4.3.1.1. Padrões de código e boas práticas para *Array*

1. Utilizar *destructuring* para acessar várias posições de um *"array"*

Para Padolsey (2020, s.p.), o uso de desestruturação em *arrays* é muito útil para evitar linhas de código confusas. Porém, acredita que seu uso só deve ocorrer no caso de fornecer maior legibilidade e simplicidade genuínas.

2. Utilizar sintaxe literal para criação de *"array"*.

"*Arrays* [...] são objetos. Eles podem ser criados com a função

construtora interna *Array (),* mas também têm uma notação literal e, assim como o literal de objeto, a notação literal de *array* é mais simples e preferida" (STEFANOV, 2010, s.p.).

3. Sempre retornar algo quando utilizar *high order functions.*

Porcello e Banks (2020, s.p.) esclarecem que o uso de high order functions tem viés funcional, onde o retorno de uma função pode ser utilizada na próxima função. Portanto, é importante que o retorno tenha valor definido.

4.3.1.2. Padrões de código e boas práticas para Blocos

1. Utilizar chaves em comandos multi-linhas.

O uso de chaves deve ser obrigatório "mesmo nos casos em que são opcionais. Se você tiver apenas uma instrução em *if* ou a *for*, as chaves não são necessárias, mas você deve usá-las de qualquer maneira. Isso torna o código mais consistente e fácil de atualizar" (STEFANOV, 2010, s.p., tradução nossa).

2. Manter o *"else"* na mesma linha das chaves.

"É uma questão de preferência, mas há casos em que o programa pode se comportar de maneira diferente dependendo de onde a chave está [...] Concluindo, sempre use chaves e sempre coloque a de abertura na mesma linha da instrução anterior" (STEFANOV, 2010, s.p., tradução nossa).

3. No caso de um retorno no *"if/else"*, não utilizar essa palavra-chave e sim manter o retorno fora do escopo.

"*Return* é uma *keyword* de controle de mudança de uma função. Isso acontece por meio de uma instrução de retorno explícita dentro da própria função ou implicitamente quando a função é executada até a conclusão" (PADOLSEY, 2020, s.p., tradução nossa). Padolsey (2020, s.p.) afirma que o uso da palavra-chave *"else"* é desnecessária em casos onde pode-se substituir o uso por *"return"*, executando apenas o bloco de condição onde o *"else"* está implícito.

4.3.1.3. Padrões de código e boas práticas para Classes

1. Sempre definir um *constructor*. No caso de não existir, criar a função e deixar vazia.

Stefanov (2010, s.p.) alerta sobre o comportamento do *JavaScript* ao criar um construtor: "os construtores retornam *this* implicitamente, mesmo quando você não tem uma instrução de retorno na função" (STEFANOV, 2010, s.p., tradução nossa). Portanto, para um código de maior controle, é interessante retonar ao menos um objeto vazio para evitar algum comportamento indevido.

2. Evitar definir dois métodos com o mesmo nome dentro de uma classe.

Stefanov (2010, s.p.) afirma que o comportamento para o cenário de dois métodos com o mesmo nome é de que a linguagem irá manter apenas o último na compilação.

3. Utilizar a *keyword "this"* quando se estiver utilizando propriedades da classe.

Para Stefanov (2010, s.p.), o uso da palavra-chave *"this"* para se referir ao escopo local dentro das classes é imprescindível para entendimento por parte da máquina e dos desenvolvedores.

4.3.1.4. Padrões de código e boas práticas para Funções

1. Utilizar *default value* em parâmetros de função.

"É bastante comum, por exemplo, ter uma assinatura de função que possui um argumento opcional. E se esse argumento não for passado ou for explicitamente definido como nulo, é normal retornar algum *default value*" (PADOLSEY, 2020, s.p., tradução nossa).

2. Não modificar os argumentos recebidos dentro da função.

Porcello e Banks (2020, s.p.) acreditam que funções devem ser puras, seguindo o

conceito de programação funcional. "Funções puras não mudam nada em seu ambiente ou “mundo” [...] Tudo que uma função pura precisa para operar, ela acessa por meio de argumentos" (PORCELLO e BANKS, 2020, s.p., tradução nossa).

3. Para funções com vários parâmetros, deixe cada um destes em uma linha.

"Um aspecto frequentemente esquecido da legibilidade do código é o uso de espaço em branco vertical. Você pode usar linhas em branco para separar unidades de código, assim como os parágrafos são usados na literatura para separar ideias" (STEFANOV, 2010, s.p., tradução nossa).

4. Utilizar a notação de *arrow function*.

"O uso da sintaxe da *arrow function* nos impede de acessar a instância por meio do this [...] *arrow function* é mais sucinta e uma ferramenta muito útil" (PADOLSEY, 2020, s.p., tradução nossa).

5. Omitir a *keyword "return"* em *arrow functions* que ocupem apenas uma linha.

"*Arrow function* é, em muitos aspectos, apenas uma versão um pouco mais sucinta da função tradicional [...] vem em dois sabores: a variante concisa inclui um retorno implícito, enquanto a variante regular [...] requer que você defina um corpo de função" (PADOLSEY, 2020, s.p., tradução nossa).

6. Utilizar parênteses entre argumentos (em *arrow function*) mesmo que contenha somente um.

"*Arrow functions* permitem evitar o uso de parênteses ao declarar uma função com apenas um argumento. Nesses casos, você pode apenas colocar o identificador do argumento sozinho antes da seta" (PADOLSEY, 2020, s.p., tradução nossa).

4.3.1.5. Padrões de código e boas práticas para Módulos

1. Unificar chamadas de módulos.

"Você pode definir suas constantes, variáveis, funções e classes [...] e, em seguida, escrever uma única instrução de exportação que declare exatamente o que é exportado em um único lugar" (FLANAGAN, 2020, s.p., tradução nossa).

2. Em módulos com apenas uma exportação, utilizar exportação *default*.

"Exportações *default* como exportação padrão podem exportar qualquer expressão, incluindo expressões de função anônima e expressões de classe anônima" (FLANAGAN, 2020, s.p., tradução nossa).

3. Manter todas as importações no topo do arquivo.

"Por convenção quase universal, as importações necessárias para um módulo são colocadas no início do módulo. No entanto, isso não é necessário: como as declarações de função, as importações são "içadas" para o topo" (FLANAGAN, 2020, s.p., tradução nossa).

4. Não incluir a extensão *".js"* no final dos arquivos importados.

Este padrão é suportado pelos *styles guide* populares.

4.3.1.6. Padrões de código e boas práticas para *String*

1. Utilizar *single quote* (aspas simples) para *"string".*

"*Strings* são expressas delimitando sequências de caracteres com aspas simples, aspas duplas ou crases" (PADOLSEY, 2020, s.p., tradução nossa). Para manter um padrão é importante escolher apenas um tipo para ser utilizado.

2. Utilizar *template literal string* ao concatenar *"strings"*

"Hoje em dia, *template literals* são preferidos, pois podem adicionar novas linhas e nos permitem interpolar expressões arbitrárias" (PADOLSEY, 2020, s.p., tradução nossa).

4.3.1.7. Padrões de código e boas práticas para Objetos

1. Utilizar sintaxe literal para criação de objetos.

"Os objetos que você cria em *JavaScript* são mutáveis a qualquer momento. [...] Você pode começar com um objeto vazio e adicionar funcionalidade conforme avança. A notação literal do objeto é ideal para esse tipo de criação de objeto sob demanda" (STEFANOV, 2010, s.p., tradução nossa).

2. Utilizar *destructuring* para acessar várias propriedades de um objeto.

"Com a desestruturação de objeto é possível declarar o identificador ou propriedade que deseja atribuir e colocá-lo como se fosse um valor em um literal de objeto" (STEFANOV, 2010, s.p., tradução nossa). Da mesma forma que a utilização da técnica em *array* deixa o código mais claro e limpo.

3. Utilizar *shorthand* em objetos.

Flanagan (2020, s.p.) descreve um atalho do *JS* com relação a criação de propriedades de um objeto no *ES6*. Quando o nome da propriedade for igual ao nome da variável que será atribuída, pode-se unir as duas informações.

4. Acessar propriedades com *dot notation*.

"A sintaxe de *dot notation* é a mais simples forma de acesso de propriedade de objetos, mas só pode ser usada quando a propriedade que você deseja acessar tem um identificador legal e quando você sabe o nome deste" (FLANAGAN, 2020, s.p., tradução nossa).

4.3.1.8. Padrões de código e boas práticas para Sintaxe

1. Comparar valores com *triple equation.*

"O *JS* tipifica implicitamente as variáveis quando você as compara [...] Para evitar confusão causada pela conversão implícita de tipos, sempre use os operadores === e !== que verificam os valores e o tipo das expressões que você

compara" (STEFANOV, 2010, s.p., tradução nossa).

2. Utilizar operador ternário em apenas um nível.

"O operador condicional é o único operador ternário em *JavaScript* [...] é escrito?:, [...] Como esse operador tem três operandos, o primeiro vai antes de?, O segundo vai entre? e o:, e o terceiro vai após o:" (FLANAGAN, 2020, s.p., tradução nossa). O operador ternário apresenta uma sintaxe que pode confundir quem está lendo o código, por isso é preferível escrever apenas um nível de comparação utilizando a técnica.

3. Separar operadores de comparação de lógicos.

"Para evitar confusão no entendimento de operações é convencional mover operadores unários para mais perto de seus operandos e até mesmo empregar parênteses para deixar absolutamente claro qual é a intenção" (PADOLSEY, 2020, s.p., tradução nossa).

4. Utilizar *High Order Functions* para iterações ao invés de *"loops"* simples.

"Em *loops for*, você itera sobre *arrays* ou objetos semelhantes a *array* [...] Um problema com esse padrão é que o comprimento do *array* é acessado em cada iteração do *loop*. Isso pode tornar seu código mais lento" (STEFANOV, 2010, s.p., tradução nossa).

5. Nomear variáveis, funções e instâncias com *Camel Case*.
6. Nomear *constructors* e classes como *Pascal Case*.

Stefanov (2010, s.p.) aconselha o uso da convenção *Pascal Case*, para construtores - onde usa-se a primeira letra maiúsculas e todas outras minúsculas -, e *Camel Case* para nomes de funções, métodos e variáveis, onde todas as letras são usadas como minúscula.

7. Evitar nomes de variáveis e funções com apenas uma letra.

Para Padolsey (2020, s.p.) pode-se dizer que um bom nome para

ser utilizado em um código deve ter: objetivo, conceito e contrato e apenas um carácter é incapaz de possuir as três características.

8. Utilizar ponto e vírgula no final de comandos.

"Você deve sempre usar ponto-e-vírgula, mesmo quando eles estão implícitos no compilador do *JavaScript*. Isso não apenas promove disciplina e uma abordagem mais rigorosa do código, mas também ajuda a resolver ambigüidades" (STEFANOV, 2010, s.p.).

9. Adicionar vírgulas no final de *keywords* que utilizam a sintaxe.

Vírgulas são utilizadas para delimitar as propriedades e métodos dentro do objeto, como também valores dentro de um *array*. Stefanov (2010, s.p.) alerta que, apesar de ser permitido o uso de vírgula após o último par chave-valor, este uso produz erros no navegador *Internet Explorer.*

10. Ter um máximo de caracteres por linha.

"Linhas longas nunca são uma boa ideia [...] Uma regra prática é que se a linha for mais larga do que o que pode caber na tela, ela é muito larga. Use seu bom senso e divida o código em várias linhas" (WASSBERG, 2020, s.p., tradução nossa).

4.3.1.9. Padrões de código e boas práticas para Variável

1. Utilizar *"const/let"* para declaração de variáveis ao invés de *"var".*

"O problema com as variáveis globais é que elas são compartilhadas entre todo o código em seu aplicativo *JavaScript* ou página da *web*. Eles vivem no mesmo espaço global e sempre há uma chance de colisões de nomenclatura" (STEFANOV, 2010, s.p., tradução nossa).

2. Utilizar apenas um *"const/let"* por declaração de variável.

"Você pode escrever uma instrução para para criar uma variável e

declarar várias variáveis delimitadas por vírgulas" (STEFANOV, 2010, s.p., tradução nossa).

3. Não utilizar cadeia de variáveis.

"Outro antipadrão que cria globais implícitos é encadear atribuições como parte de uma declaração *var*" (STEFANOV, 2010, s.p., tradução nossa).

4. Excluir variáveis não utilizadas no código.

Ter o chamado código morto no projeto "tornará o código mais difícil de ler [...] Com o código desnecessário removido, podemos ver como podemos usar o *hardware* do computador com mais eficiência" (WASSBERG, 2020, s.p., tradução nossa).

4.3.1.10. Padrões de código e boas práticas para Espaços em Branco

1. Indentar o código utilizando dois espaços.

"É importante padronizar o uso de indentação [...] E o que você deve recuar? [...]- qualquer coisa entre chaves. Isso significa os corpos de funções, *loops* (*do, while, for, for-in*), *ifs, switches* e propriedades do objeto na notação literal do objeto" (STEFANOV, 2010, s.p., tradução nossa).

2. Adicionar espaço após chamada de funções, estruturas de repetição e condicionais.

3. Adicionar espaço entre operadores.

4. Pular uma nova linha no final de cada arquivo.

5. Não pular linha após início de bloco.

6. Não adicionar espaço em parênteses.

7. Não adicionar espaço em colchetes.

8. Adicionar espaço dentro de chaves.

9. Evitar espaços a mais no final de linhas de código (*trailing spaces*).

10. No começo de um comentário, manter um espaço.

"O uso de espaço em branco também pode contribuir para melhorar

a legibilidade e a consistência do código [...] Em *JavaScript*, você adiciona intervalos após expressões semelhantes a listas (equivalente a vírgulas) e fim de declarações (equivalente a completar um “pensamento”)" (STEFANOV, 2010, s.p., tradução nossa).

Stefanov (2010, s.p.) elenca várias boas práticas para uso de espaços em branco: após os pontos-e-vírgulas que separam as partes de um *loop for*, delimitando argumentos de função, antes das chaves nas declarações de função, separando todos os operadores e seus operandos, adicionando uma linha em branco ao final do arquivo, espaçamento entre chaves, antes de abrir chaves ({) em funções, *if-else, loops* e objeto.

4.3.2. Padrões de código e boas práticas em *React JS*

Da mesma forma que autores discutem as melhores práticas de codificação para *JavaScript*, Roldan (2020, s.p.) escreve suas considerações sobre o tema utilizando *React JS*. Seguem definições de padrões para se manter uma legibilidade maior e que sejam totalmente possíveis de serem integradas à ferramenta *ESLint*.

1. Nomear componentes como *Pascal Case*.
2. Sempre fechar *tags* nela mesmo, quando não houver filhas.
3. Alinhar estilos para a sintaxe de *JSX*.
4. Adicionar um espaço antes do fechamento de *tags*.
5. Não adicionar espaço dentro de chaves no *JSX*.
6. Utilizar *double quotes* (aspas duplas) para atributos *JSX*.
7. Omitir valor da *prop* no caso de ser implicitamente *true*.

Roldan (2020, s.p.) lista boa práticas para manter a sintaxe de *JSX* alinhada e legível: nomear componentes com a convenção *Pascal Case*; manter o fechamento da *tag* nela mesma caso não haja filhas; utilizar elementos aninhados, deve-se sempre usar várias linhas, com exceção para o caso dos filhos não serem elementos; evitar espaços desnecessários dentro do *JSX* e utilizar aspas duplas em atributos *JSX* se assemelhando ao comportamento do *HTML;* manter a *prop* como implicitamente *true* se for possível.

8. Sempre adicionar a propriedade *"alt"* em *tags* de <*img*>.

Tags de imagem sempre devem conter o atributo *tag*, "ou uma *string* vazia se a imagem não afetar a compreensão do usuário sobre o conteúdo" (PORCELLO e BANKS, 2020, s.p., tradução nossa).

9. Evitar utilizar *index* de *array* como chave ao utilizar a função *map*.

"Se escrevermos uma função que retorna um array dentro de nosso template JSX, cada elemento do array é compilado em um elemento [...] a maneira mais comum de gerar um array de elementos é usar *map"* ROLDAN (2020, s.p., tradução nossa).

10. Utilizar parênteses quando o retorno for apenas de uma linha.

"Lembre-se sempre de envolver seus elementos entre parênteses [...]. *JSX* sempre é substituído por funções, e funções escritas em uma nova linha podem fornecer um resultado inesperado por causa da inserção automática de ponto-e-vírgula" ROLDAN (2020, s.p., tradução nossa).

4.4. Refatoração

"Refatoração é o processo de alterar o código-fonte de um *software* de uma maneira que seu comportamento atual seja mantido e melhore sua estrutura interna" (FOWLER, 2009, s.p., tradução nossa). O objetivo de tal processo é transformar um código-fonte que teve uma concepção sem escalabilidade e se tornou complexo, em algo simples de entender e de alterar, e claramente pronto para ser amplamente utilizado.

Ao refatorar um código o objetivo é de melhoria do projeto, tornando o código mais legível, auxiliando a busca de erros - onde desenvolvedores envolvidos no processo tem uma visão clara de cada regra de negócio -, além da prevenção de defeitos.

"Deve-se ressaltar que existem diversos riscos atrelados ao processo de refatoração, pois erros sutis podem ser inseridos por quem a aplica, afinal somos humanos e cometemos erros " (FOWLER, 2009, s.p., tradução nossa). A refatoração não é algo trivial como excluir meia dúzia de funções e substituir por apenas duas. No âmago é abstrair de tal forma que seja claro para todo o time, e para colaboradores que virão a se juntar ao time.

Wassberg (2020, s.p.) pontua que a legibilidade tem sempre que ser o foco principal ao reescrever um código. Porém, às vezes, precisa-se sacrificar a legibilidade para tornar o código mais eficiente ou mais rápido.

4.5. Mensurando qualidade de código

Segundo Wassberg (2020, s.p.), a organização *Consortium for Information Software Quality* definiu cinco regras para mensurar a qualidade de código. Primeiramente, foram encabeçadas pensando apenas em *softwares* de negócios. Porém acabou se estendendo para todos os tipos de *software*.

As cinco regras são:

1. Confiabilidade: "mede o nível de risco e a probabilidade de falhas. Ele também medirá os defeitos que são injetados no código existente quando ele é atualizado ou modificado" (WASSBERG, 2020, s.p., tradução nossa). O objetivo é evitar que a aplicação fique impossibilitada de ser executada por determinado tempo devido a erros.

2. Eficiência de desempenho: quando um aplicativo está em execução, a velocidade com que executa suas operações depende de como o código é escrito e estruturado. "Medir a eficiência de código ajudará a melhorar o tempo de resposta geral de um aplicativo e identifica o risco de potencial falha de aplicativos que precisam processar dados em alta velocidade" (WASSBERG, 2020, s.p., tradução nossa).

3. Segurança: "a regra de segurança medirá a probabilidade de possíveis violações de segurança devido a práticas de codificação inadequadas" (WASSBERG, 2020, s.p., tradução nossa).

4. Capacidade de manutenção: quando se fala de manutenção, refere-se a três pilares. O código deve ser: "adaptável, capaz de se adaptar às mudanças de acordo com os requisitos; portátil, podendo ser usado em diferentes plataformas e sistemas operacionais; e transferível, podendo ser transferido de uma equipe de desenvolvimento para outra" (WASSBERG, 2020, s.p., tradução nossa).

5. Tamanho: "o tamanho não é um atributo de qualidade em si, mas o tamanho do código pode ter um impacto em sua manutenção. Quanto mais código tivermos, mais difícil será navegar, entender e seguir sua lógica" (WASSBERG, 2020, s.p., tradução nossa).

4.6. Vantagens do código limpo

O autor Elijah Lewis (2020, s.p., tradução nossa) listou quatro vantagens do *clean code* em seu livro *CLEAN CODE: Advanced and Effective Strategies To Use Clean Code Methods*, conseguindo trazer uma visão da empresa como um todo sendo beneficiada com esses padrões, conforme vemos abaixo:

1. Responsabilidade sobre o código: todo programador tem que ter em mente que o seu código será lido por outros colegas. Então, se está complicado redigir o código, será mais complicado ainda fazer a manutenção dele ou aplicar alguma correção em erros ou defeitos que possivelmente possam surgir na implementação.
2. Trabalhando em time: normalmente, desenvolvedores atuam em times. Se o código escrito não pode ser entendido pelos outros membros do time, o trabalho não está bem feito. A consciência do repasse de código tem que ser baseada no entendimento do próximo.
3. Facilmente escalável e manutenibilidade: adicionar novas funcionalidades ou fazer mudanças em módulos já existentes é uma tarefa delicada, e só fica mais complexa com códigos que não passam de suas funções no projeto. Se o código seguir padrões de código limpo, qualquer desenvolvedor poderá trabalhar nele.
4. Aja como dono do produto: o pensamento que se busca em empresas no mundo atual, é o de colaboradores manterem suas entregas como se o produto fosse deles, ou seja, no momento que o desenvolvedor garante que terminou o projeto, ele entende que está pronto para ser entregue ao cliente e em seu perfeito funcionamento.

5. FERRAMENTA *ESLINT*

5.1. *ESLint*

"O linter não apenas nos ajuda a cometer menos erros, ou pelo menos, encontrar esses erros mais cedo, mas também impõe alguns guias de estilo de codificação comuns, o que é importante, especialmente em grandes equipes com muitos desenvolvedores" (PADOLSEY, 2020, s.p., tradução nossa).

> *ESLint* é um utilitário de *linting* de *JavaScript* de código aberto originalmente criado por Nicholas C. Zakas em junho de 2013. O *linting* de código é um tipo de análise estática frequentemente usada para encontrar padrões problemáticos ou código que não seguem certas diretrizes de estilo. Existem *linters* de código para a maioria das linguagens de programação e, às vezes, os compiladores incorporam *linting* ao processo de compilação. *(ESLINT,* 2021*,* tradução nossa)

Segundo o *website* do *ESLint* (*ESLINT*, 2021) o principal motivo pelo qual a ferramenta surgiu foi para permitir que os desenvolvedores criassem suas próprias regras de *linting*. "*ESLint* foi projetado para ter todas as regras completamente plugáveis. As regras padrão são escritas como quaisquer regras de *plug-in*" (*ESLINT*, 2021, tradução nossa).

Flanagan (2020, s.p.) revela que apesar de a ferramenta *ESLint* ter a capacidade de definir muitas regras de *linting* e apresentar um ecossistema de *plugins* que adicionam muito mais, o *ESLint* é totalmente configurável e o desenvolvedor pode definir um arquivo de configuração que ajusta a ferramenta para aplicar exatamente as regras que acredita serem necessárias.

O *ESLint* se fez necessário pois, como visto anteriormente, a linguagem *JavaScript* é fracamente tipada e se torna especialmente "sujeita a erros do desenvolvedor. Sem o benefício de um processo de compilação, o código *JavaScript* normalmente é executado para encontrar sintaxe ou outros erros" (*ESLINT*, 2021, tradução nossa). Ferramentas de *linting* permitem que os desenvolvedores descubram problemas com seu código *JavaScript* sem executá-lo.

O *ESLint* é amplamente utilizado por desenvolvedores que

trabalham com *JavaScript* por ser o *linter* mais poderoso e extensível disponível para a linguagem, além de conseguir estender seus benefícios aos *frameworks*, *libraries* e ao *JSX*.

As filosofias seguidas no *ESLint* estão listadas no *website* da ferramenta (*ESLINT*, 2021):

- Tudo é plugável, as regras podem ser customizadas ou utilizadas através de pré-configurações. Regras adicionais podem ser adicionadas a qualquer momento.
- Cada regra é independente, podendo ser ligada e desligada quando necessário, e podem ser definidas como um aviso ou um erro, de forma individual.
- A ferramenta *ESLint* não promove nenhum estilo de codificação em particular, dando total liberdade de escolha aos usuários.

Ao iniciar o *ESLint* em um projeto, o desenvolvedor tem uma série de escolhas para personalizar o uso da ferramenta, entre elas pode-se escolher: 1) adicionar regras prontas de *plugins*, configurações já criadas e reconhecidas, ou 2) criar manualmente suas próprias regras. A ferramenta tem como entrada um arquivo de configuração - *eslintrc*, a extensão pode variar em *.js*, *.yml* e *.json* - com propriedades como as da figura 45 exemplificada abaixo, onde consegue-se adicionar regras, configurações pré-definidas e *plugins*.

A partir dessa configuração, o desenvolvedor terá acesso ao comando para rodar o *ESLint* no projeto. Por ser um comando programático, consegue-se também definir fluxos de *deploy* contínuo que rodem o *ESLint* e impeçam o *deploy* em produção no caso de acontecer falhas de código.

Figura 45 - Arquivo de configuração exemplo do *ESLint* (*.eslintrc.js*).

```
module.exports = {
    "env": {
        "browser": true,
        "es6": true
    },
    "parserOptions": {
        "ecmaFeatures": {
            "experimentalObjectRestSpread": true,
            "jsx": true
        },
        "sourceType": "module"
    },
    "extends": "custom"
};
```

Fonte: própria Autora.

Ao escolher utilizar o *ESLint* o projeto apresenta uma melhora para produtividade de seus mantenedores, conforme descrito no *site* oficial (*ESLint*, 2021), criando-se:

- Estilo consistente de código: *linters* permitem a verificação do estilo do código quanto a problemas como espaçamento, indentação e colocação de chaves. Sendo acordado entre a equipe qual estilo de codificação seguir, ele pode ser documentado em um arquivo de configuração e verificado automaticamente.
- Identificação de potenciais erros e padrões ruins: *linters* também podem ser usados para realizar verificações mais avançadas, possibilitando descobrir possíveis erros: variáveis duplicadas, código inacessível ou expressões regulares inválidas.
- Reforço na qualidade: optando por seguir um determinado guia de estilo no projeto, é importante aplicá-lo com ferramentas. Caso contrário, sempre haverá pessoas tentadas a usar atalhos. Se uma ferramenta de *linting* estiver conectada ao processo de *deploy*, haverá etapas para impedir que o projeto seja enviado ao repositório se houver erros não corrigidos.
- Economia de tempo: *linters* poupam esforços durante o desenvolvimento. Não há mais perda de tempo em discussões sobre sintaxes melhores ou piores no código, além de descobertas de erros em fases iniciais. Talvez esse seja o maior benefício visto por toda uma equipe de desenvolvimento, além de gestor e cliente.

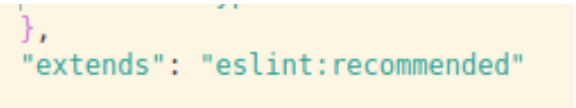
5.2. *ESLint: Style Guides*

Style Guide é um manual sobre como o código deverá ser estruturado sintaticamente, como visto nos capítulos acima. Também, como citado anteriormente, no momento de configuração do *ESLint* existe a possibilidade de escolher configurações já conhecidas em seu projeto, poupando o trabalho de definir cada regra no arquivo *.eslintrc*. Os mais procurados, atualmente, são os que dão suporte ao *ES6+* e aos componentes *JSX*, como: *ESLint Recommended, Google JavaScript Style Guide, AirBnB* e *Standard.*

Para utilizar padrões populares, basta adicionar o nome do padrão dentro de "*extends*" no arquivo de configuração do *ESLint* - figura 46.

"O *ESLint Recommended* vem por padrão ativado, contendo regras já integradas para torná-lo útil desde o início" (*ESLINT*, 2021, tradução nossa). Estas regras referem-se apenas a erros mais comuns e genéricos, como não permitir: variáveis globais, re-declaração de variáveis, variáveis não utilizadas no código, misturar uso de *tabs* e *spaces* na indentação, entre outras. Todas as regras podem ser conferidas no site oficial da ferramenta *ESLint*.

Figura 46 - Exemplo de arquivo de configuração do *ESLint* utilizando *ESLint Recommended*.



Fonte: própria Autora.

O padrão *Google JavaScript Style Guide* (figura 47), aborda "não somente questões que abrangem os padrões de estéticas de formatação, mas também outros tipos de convenções ou padrões de codificação" (*GOOGLE JAVASCRIPT STYLE GUIDE*, 2021, tradução nossa). Algumas destas regras: utilizar ponto e vírgula, parênteses entre o argumento de uma *arrow function,* uso de *JSDocs*, proibição de "*console.log"* no código, espaço após o nome de função quando sendo declarada, entre outros. A desvantagem deste guia é que não existe suporte ao *React JS*. Por isso não é amplamente utilizado pela comunidade.

Figura 47 - Exemplo de arquivo de configuração do *ESLint* utilizando *Google JavaScript Style Guide*

```
},
"extends": ["eslint:recommended", "google"]
```

Fonte: própria Autora.

O *AirBnB Style Guide* (figura 48) é um dos mais conhecidos e consegue dar o suporte ao *React JS*. O guia apresenta "uma abordagem bastante razoável para *React* e *JSX* [...] é baseado principalmente nos padrões que prevalecem atualmente em *JavaScript*, embora algumas convenções [...] ainda possam ser incluídas ou proibidas caso a caso" (*AIRBNB/JAVASCRIPT*, 2021, tradução nossa). Tem, em sua maioria, os mesmos requisitos do *Google*, mas com adicionais para a *library React* como validação de *props* e desestruturação de objetos como preferência.

Figura 48 - Exemplo de arquivo de configuração do *ESLint* utilizando *AirBnB*.

```
},
"extends": ["eslint:recommended", "airbnb"]
```

Fonte: própria Autora.

Por fim, existe o *Standard Style Guide* (figura 49), que também dá suporte ao *React JS*. Segundo o site oficial deste padrão, adotar este "significa classificar a importância da clareza do código e das convenções da comunidade acima do estilo pessoal" (*JAVASCRIPT STANDARD STYLE*, 2021, tradução nossa). Ele apresenta requisitos um pouco diferentes dos dois anteriores, como por exemplo, não tendo como obrigatoriedade o uso de ponto e vírgula e nem a validação de *props*.

Figura 49 - Exemplo de arquivo de configuração do *ESLint* utilizando *Standard*.

```
},
"extends": ["eslint:recommended", "standard"]
```

Fonte: própria Autora.

6. PROCEDIMENTOS METODOLÓGICOS

"A Metodologia tem como função mostrar a você como andar no "caminho das pedras" da pesquisa, ajudá-lo a refletir e instigar um novo olhar sobre o mundo: um olhar curioso, indagador e criativo." (SILVA, 2005, p. 9).

Minayo (1993, p. 23) trouxe uma visão filosófica sobre o conceito de pesquisa, considerando-a como "atividade básica das ciências na sua indagação e descoberta da realidade". É uma atitude e uma prática teórica de constante busca que define um processo intrinsecamente inacabado e permanente. É uma atividade de aproximação sucessiva da realidade que nunca se esgota, fazendo uma combinação particular entre teoria e dados.

Este capítulo é dedicado ao relato sobre todos os estágios da pesquisa que compõem o embasamento deste projeto, citando caracterizações, expondo delimitações e repassando todas as etapas a serem seguidas.

6.1. Caracterização da pesquisa

"Pesquisa é um conjunto de ações, propostas para encontrar a solução para um problema, que têm por base procedimentos racionais e sistemáticos. A pesquisa é realizada quando se tem um problema e não se tem informações para solucioná-lo." (SILVA, 2005, p. 20)

Quanto a sua natureza, essa pesquisa se caracteriza como aplicada, tendo como objetivo "gerar conhecimentos novos e úteis para o avanço da ciência sem aplicação prática prevista." (SILVA, 2005, p. 20).

Pela sua forma de abordagem do problema, se classifica como qualitativa, pois "não requer o uso de métodos e técnicas estatísticas. O ambiente natural é a fonte direta para coleta de dados e o pesquisador é o instrumento-chave." (SILVA, 2005, p. 20).

Do ponto de vista do seu objetivo, considera-se descritiva, visando descrever características de determinado fenômeno e estabelecer relações entre variáveis. Segundo Gil (2007, p. 43), este tipo de pesquisa envolve o uso de técnicas padronizadas de coleta de dados, como questionários e observação sistemática.

Também classificado como pesquisa exploratória, envolvendo "levantamento bibliográfico; entrevistas com pessoas que tiveram experiências práticas com o problema pesquisado; análise de exemplos que estimulem a compreensão" (GIL, 2007, p. 21).

Quanto aos procedimentos técnicos, classifica-se em bibliográfica, de levantamento e pesquisa de campo. Conforme define Fonseca (2002, p. 32), "a pesquisa bibliográfica é feita a partir do levantamento de referências teóricas já analisadas, e publicadas por meios escritos e eletrônicos, como livros, artigos científicos e páginas de *web sites*". De acordo com Gil (2007, p. 52), os estudos descritivos são os que mais se adequam aos levantamentos. Este projeto valida as regras de configuração para implementar o *ESLint* por meio de questionário e, portanto, é apropriada à classificação de pesquisa de levantamento. E por fim, agrega-se à classificação de pesquisa de campo, que "caracteriza-se pelas investigações em que, além da pesquisa bibliográfica e/ou documental, se realiza coleta de dados junto a pessoas, com o recurso de diferentes tipos de pesquisa." (FONSECA, 2002, p. 32).

Neste estudo, é relatada a natureza do código limpo e seus conceitos seguindo bibliografias selecionadas conforme influência na área de programação. Também são coletadas opiniões sobre o assunto no dia a dia de profissionais através da aplicação de um questionário *online* para um grupo delimitado de pessoas.

Após a inclusão da ferramenta no projeto modelo - seguindo as especificações baseadas em estudos bibliográficos - o trabalho tem como conclusão um arquivo de configuração do *ESLint* disponibilizados por meio do *GitHub*, sendo de acesso público, buscando facilitar o uso da ferramenta em projetos desenvolvidos utilizando *React JS*.

6.2. Delimitação e população da pesquisa

"Escolher um tema significa eleger uma parcela delimitada de um assunto, estabelecendo limites ou restrições para o desenvolvimento da pesquisa pretendida" (SILVA, 2005, p. 30).

Este projeto delimita-se em quatro aspectos:

1. Padrões de codificação: segundo Stefanov (2010, s.p.) existem mais padrões na programação, como *design patterns* (visando a arquitetura do projeto) e *coding patterns*. O estudo se limitou a apenas padrões de codificação.

2. Código limpo na linguagem *JavaScript*. Apesar de todos os conceitos de *clean code* apresentados neste projeto serem pertinentes a qualquer linguagem de programação, a ferramenta utilizada como solução dos problemas causados por códigos mal concebidos reconhece apenas a linguagem *JS*.

3. Especificações da ferramenta para a *library React JS*. O projeto modelo que foi utilizado para a implementação foi construído utilizando a *library* visando alcançar o maior número de desenvolvedores, visto que é a tecnologia mais utilizada atualmente, segundo o *StackOverflow* (2020). Para uso da ferramenta em projetos escritos em *JavaScript* puro ou com outro *framework* moderno, como *Vue JS* ou *Angular*, as especificações são outras, modelando-se suas próprias sintaxes.

4. População para aplicação do questionário na etapa final: pessoas que atuam diretamente em códigos (desenvolvedores, líderes técnicos), com qualquer nível de experiência, que trabalhem utilizando *React JS*. Os profissionais que responderam o questionário trabalham em empresas do tipo: micro (até nove empregados), pequeno (de 10 a 49 empregados), médio (de 50 a 99 empregados) e grande (mais de 100 empregados).

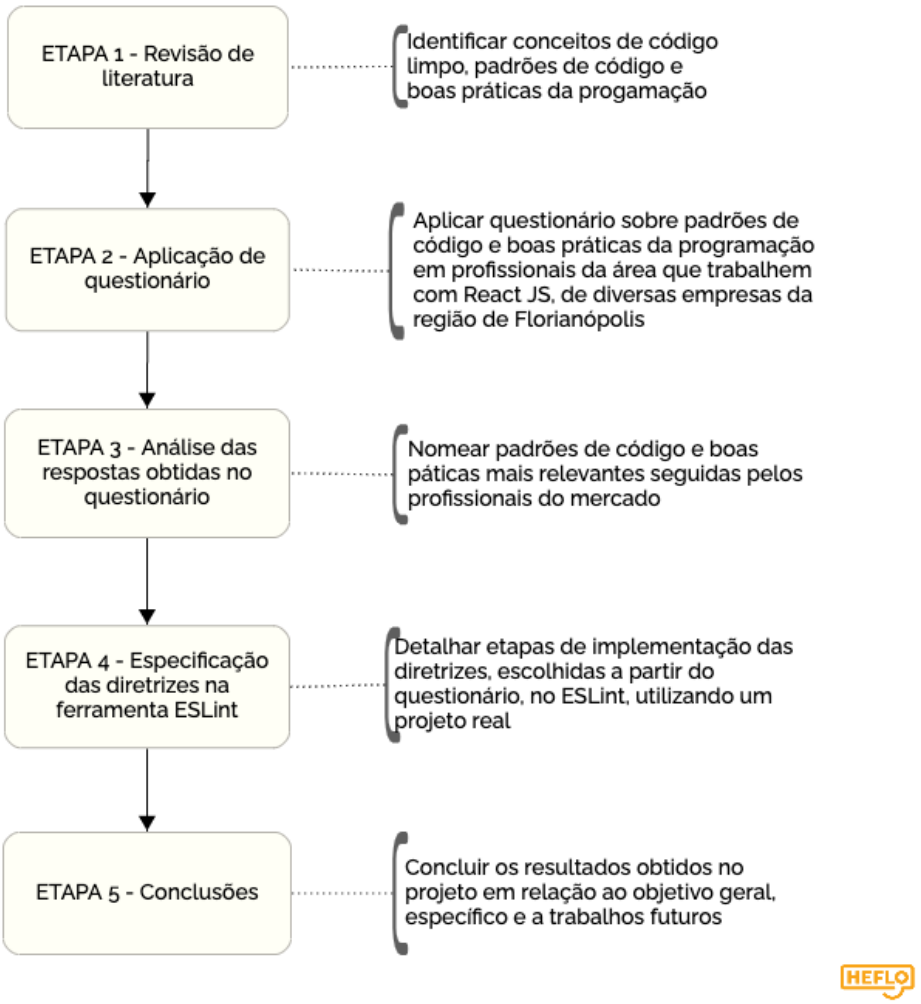
6.3. Etapas da pesquisa

"O planejamento e a execução de uma pesquisa fazem parte de um processo sistematizado que compreende etapas" (SILVA, 2005, p. 29). O desenvolvimento deste projeto foi realizado em cinco etapas, sendo elas: revisão da literatura, aplicação de questionário por meio *online*, análise das respostas obtidas na aplicação do questionário, especificação das diretrizes na ferramenta *ESLint* e, por fim, as conclusões obtidas neste projeto.

A figura 50 apresenta as etapas em ordem cronológica e com um pequeno resumo sobre cada uma. Cada etapa dependeu inerentemente da anterior para se fosse construída uma base de conhecimento que permitisse a entrega de um resultado final satisfatório. A revisão literária permitiu que questões relevantes fossem levantadas no questionário. A aplicação do questionário possibilitou, por sua vez, que padrões de códigos e boas práticas da programação que realmente são utilizadas pelo mercado fossem mapeados.

Através das análises das respostas obtidas pela aplicação do questionário, levantou-se quais diretrizes deveriam ser aplicadas na ferramenta *ESLint* e para que a ferramenta pudesse ser testada de forma plenamente funcional em um projeto real. Como conclusão, originou-se em um arquivo de configuração do *ESLint* que poderá ser utilizado pela comunidade de desenvolvedores que trabalham com *React JS*.

Figura 50 - Fluxograma contemplando as etapas da pesquisa



Fonte: própria Autora.

6.3.1. ETAPA 1 - Revisão de literatura

A revisão da literatura - capítulo dois deste documento - é uma das mais importantes, pois, segundo Silva (2005, p. 37) é por meio da análise da literatura publicada que o autor será capaz de traçar um quadro teórico e estruturar os conceitos que darão sustentação ao desenvolvimento da pesquisa.

Procedeu-se ao levantamento de obras de diversos tipos de plataforma, seja digital ou física, que fossem relevantes para o tema clean code, padrões de código, boas práticas da programação, introdução a conceitos da programação e ferramentas *ESLint*. O intuito não foi apenas endossar o trabalho,

mas também evitar que a pesquisa seja apenas uma duplicação de conteúdo com o mesmo enfoque do tema.

6.3.2. ETAPA 2 - Aplicação de questionário

Questionário é uma série ordenada de perguntas que devem ser respondidas por escrito pelos usuários. O questionário deve ser "objetivo, limitado em extensão e estar acompanhado de instruções As instruções devem esclarecer o propósito de sua aplicação, ressaltar a importância da colaboração do informante e facilitar o preenchimento" (SILVA, 2005, p. 33).

Após levantamento de padrões de código e boas práticas da programação, voltadas ao uso de *React JS*, através da revisão de literatura, as mesmas foram agrupadas em blocos como questões fechadas e de múltiplas escolhas para que profissionais da área de programação apontassem quais regras são utilizadas pelas suas equipes de trabalho. Além de incluir perguntas no âmbito de conhecer os indivíduos que responderam o questionário, como: cargo atual, qual área de programação atua, há quanto tempo está na área. Por fim, agregaram-se perguntas relacionadas à percepção de melhorias na qualidade de trabalho ao se utilizar padrões de código e boas práticas da programação - as questões na íntegra estão disponíveis no Apêndice A deste documento.

A coleta de dados foi realizada de forma *online* e enviado a profissionais da área de diversas empresas.

6.3.3. ETAPA 3 - Análise das respostas obtidas no questionário

A tabulação e apresentação dos dados foi realizada com ajuda de recursos computacionais. Os dados foram organizados a partir do questionário para elaborar gráficos para melhor visualização de resultados - exposto na primeira parte do quarto capítulo deste projeto. Foram obtidas 25 respostas através do questionário e utilizadas apenas 20 - respostas de profissionais que apontaram não trabalhar atualmente com *React JS* foram descartadas.

Para a escolha das regras de padrões de código e boas práticas da

programação, foram consideradas pelo menos metade das pessoas entrevistadas que tenham marcado tal regra como utilizada em sua equipe. Ou seja, regras que tenham pelo menos nove votos foram selecionadas para a próxima etapa.

6.3.4. ETAPA 4 - Especificação das diretrizes na ferramenta *ESLint*

Nesta etapa, documentaram-se todas as fases para implementação da ferramenta *ESLint* em um projeto real, escrito com *React JS* e sem nenhuma definição de padrão de código, da própria autora. Ao final, apresenta-se o teste plenamente funcional da ferramenta no projeto, detalhando-se mensagem de erro da ferramenta e comandos para que os erros fossem corrigidos, de forma automática.

6.3.5. ETAPA 5 - Conclusões

Na última etapa, foram expostas as conclusões acerca do entregável, sendo um arquivo de configurações do *ESLint* para uso da comunidade de desenvolvedores que utilizam *React JS.* Além de mapear trabalhos futuros que possam ser desenvolvidos posteriormente.

7. ANÁLISE DE PADRÕES DE CÓDIGO E BOAS PRÁTICAS DA PROGRAMAÇÃO NO MERCADO DE TECNOLOGIA

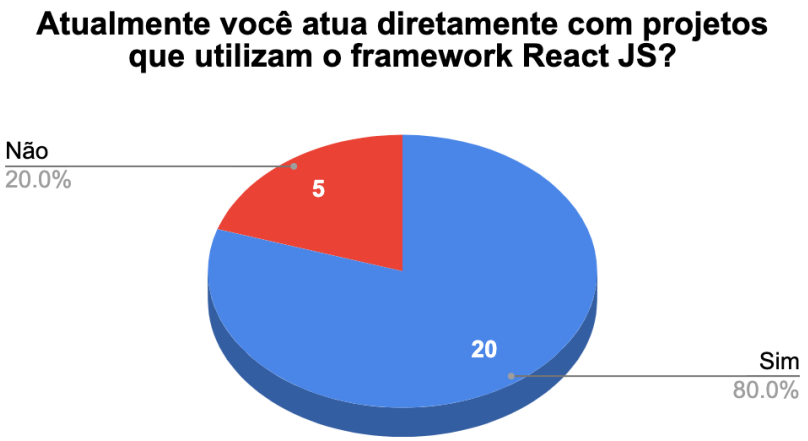
Este capítulo é dedicado à análise dos dados obtidos através do questionário aplicado de forma *online* - resultando em 25 respondentes -, a fim de compreender quais os padrões de código e boas práticas da programação são utilizados pelos profissionais no mercado de tecnologia.

7.1. Limpeza de dados obtidos no questionário

A primeira e a terceira pergunta do questionário visavam garantir que os respondentes atendessem os dois critérios da pesquisa: 1) atuassem diretamente em projetos que utilizam o *library React JS e* 2) estar investido de um cargo profissional em que atue com código, tendo assim uma base de dados mais perto da realidade possível do mercado.

Das 25 pessoas que responderam o questionário, cinco tiveram suas respostas desconsideradas por não atuarem atualmente com a ferramenta e 80% das respostas foram consideradas válidas para seguir com as análises - gráfico 1.

Gráfico 1 - Atualmente você atua diretamente com projetos que utilizam o *library React JS*?



Fonte: própria Autora.

A partir daqui, todos os dados apresentados consideraram o número total de 20 amostras. Quanto aos cargos em que os respondentes exercem - gráfico 2 -, àqueles que estão diretamente ligados à produção de códigos, sendo 2 como líderes técnicos e 18 como desenvolvedores. Sendo assim, serão mantidas 20 amostras para análise.

Gráfico 2 - Qual o seu cargo na empresa atualmente?

Fonte: própria Autora.

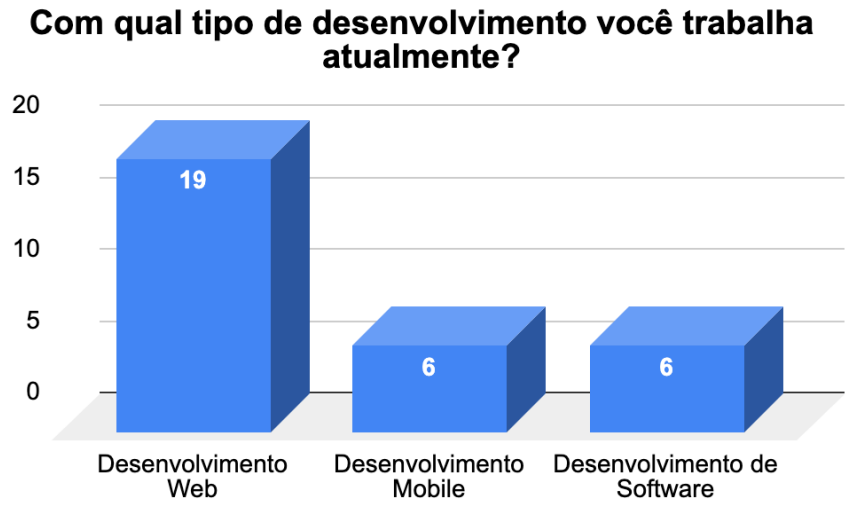
Quanto às respostas obtidas destes 20 participantes, todas foram consideradas válidas, pois os profissionais responderam todas as questões, tendo deixado algumas em branco no caso de não utilizarem nenhuma abordagem em sua equipe.

7.2. Análise do cenário profissional dos respondentes

A fim de entender melhor o cenário profissional das pessoas que responderam o questionário, três perguntas relacionadas a esta área foram adicionadas em seguida. Quanto ao tipo de desenvolvimento em que atuam (gráfico 3), onde cada respondente poderia selecionar mais de um tipo de desenvolvimento, quase a maioria atua com o desenvolvimento *web*. Houve seis respostas para desenvolvimento *mobile* e seis para desenvolvimento de *software*.

Como visto no capítulo de apresentação sobre o *React, esta tecnologia pode atuar nas 3 vertentes:* o *JS* para *web* e *software* e ou *Native* para *mobile*, compartilhando entre si as boas práticas e padrões de código.

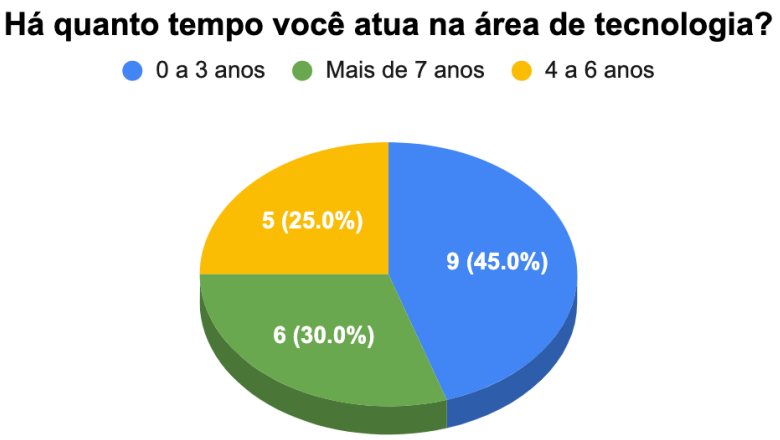
Gráfico 3 - Com qual tipo de desenvolvimento você trabalha atualmente?



Fonte: própria Autora.

Sobre o tempo em que cada profissional atua na programação, as respostas foram diversas - como visto no gráfico 4: a maioria, 45%, está iniciando na faixa de até 3 anos; 25% atua em faixa intermediária de 4 a 6 anos e 30% atua em faixa sênior, tendo uma carreira de mais de 7 anos. Podemos, assim, analisar respostas de diferentes pontos de vista quanto a tempo de mercado.

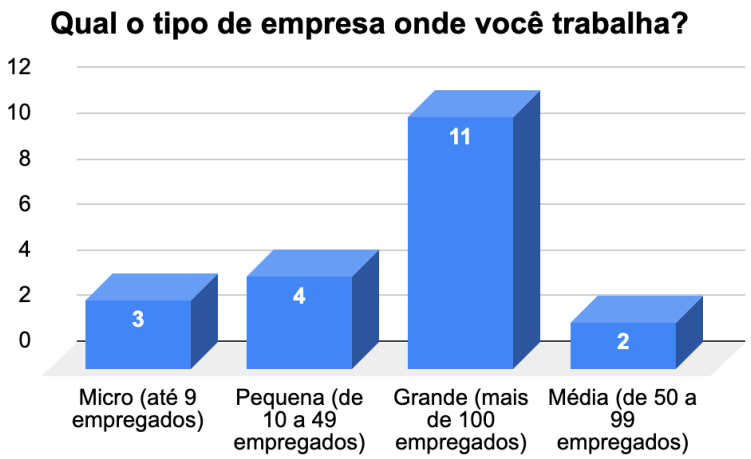
Gráfico 4 - Há quanto tempo você atua na área de tecnologia?



Fonte: própria Autora.

Analisando qual o tipo de empresa em que trabalham, ou seja o porte desta, 11 respondentes apontaram trabalhar em empresas de grande porte (com mais de 100 empregados), seguidos por 4 que atuam em pequeno porte (de 10 a 49 empregados), 3 que atuam em micro (até 9 empregados) e apenas 2 atuam em médio porte (de 50 a 99 empregados) - gráfico 5. A classificação de empresas conforme o número de colaboradores foi baseada em uma publicação do IBGE (2018, s.p.).

Gráfico 5 - Qual o tipo de empresa onde você trabalha?

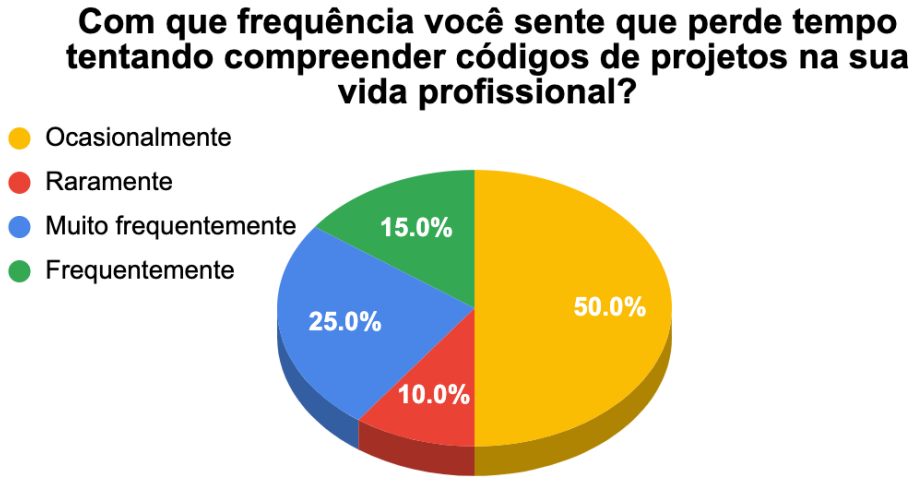


Fonte: própria Autora.

7.3. Análise sobre percepções de boas práticas e padrões de código dos respondentes

Com a finalidade de entender as percepções de boas práticas e padrões de código dos respondentes agregou-se três perguntas de teor opinativo sobre estas questões. A primeira questão foi sobre a frequência em que os profissionais sentem que perdem tempo tentando compreender códigos em projetos profissionais - gráfico 6 -: metade deles afirmaram que ocasionalmente sentem isso. Um quarto dos respondentes declararam que muito frequentemente estão neste cenário, 15% afirmou frequentemente e apenas 10% apontou que raramente perdem tempo com isto.

Gráfico 6 - Com que frequência você sente que perde tempo tentando compreender códigos de projetos na sua vida profissional?



Fonte: própria Autora.

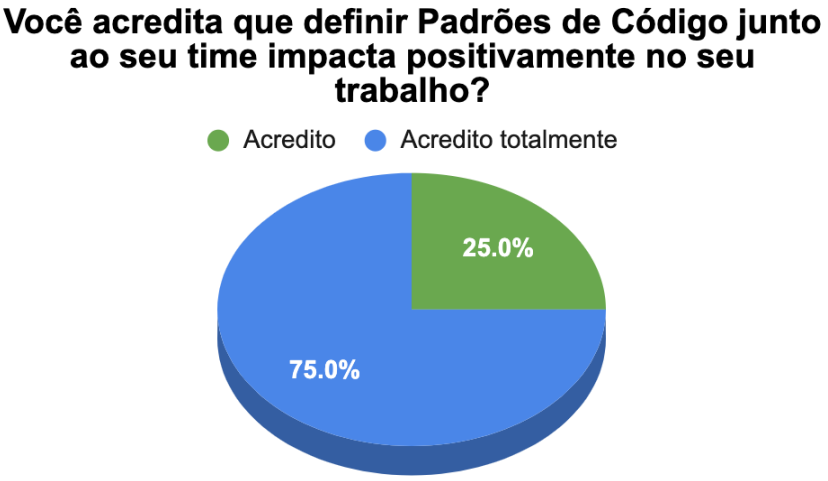
A respeito da crença em que seguir boas práticas de programação impacta positivamente no trabalho, 80% dos profissionais acreditam totalmente nesta afirmação, 10% acredita e 10% se manteve neutro por não conseguir opinar sobre a questão. Estes dados podem ser visualizados no gráfico 7.

Gráfico 7 - Você acredita que seguir as boas práticas de programação impacta positivamente no seu trabalho?

Fonte: própria Autora.

Com relação a acreditar que definir padrões de códigos junto ao time de trabalho impacta positivamente em seu trabalho as respostas foram homogêneas: 75% acredita totalmente e 25% acredita na afirmação - visto no gráfico 8.

Gráfico 8 - Você acredita que definir Padrões de Código junto ao seu time impacta positivamente no seu trabalho?



Fonte: própria Autora.

7.4. Análise sobre quais as boas práticas e padrões de código são utilizadas pelos respondentes no âmbito profissional

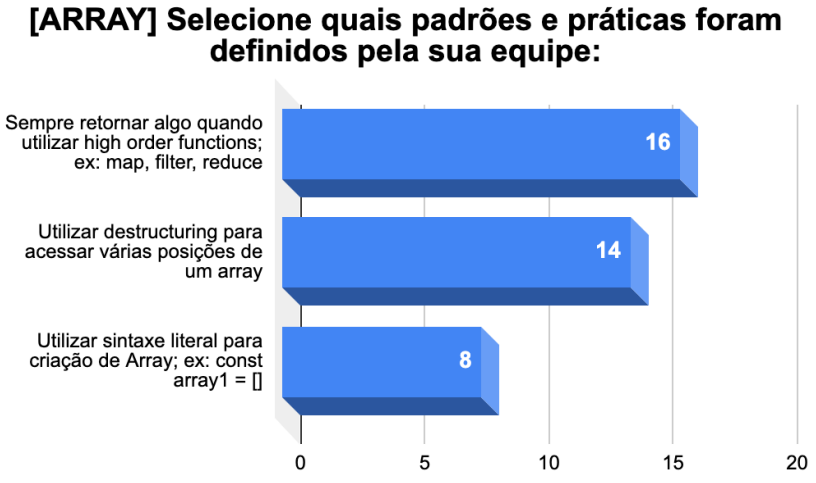
Com o intuito de decidir quais as boas práticas e padrões de código que seriam utilizados na implementação do *ESLint* ao projeto construído com *React JS*, fez-se perguntas acerca de quais regras os profissionais utilizam atualmente em seus times. As regras foram escolhidas quando, pelo menos, metade dos profissionais tenham afirmado utilizá-la em ambiente profissional, ou seja, regras com um mínimo de 10 votantes.

Cada questão tem seu resultado exposto visualmente através de gráfico e seu resultado debatido. Também há um exemplo prático de cada regra escolhida a fim de elucidar quais padrões seriam adotados na ferramenta no próximo capítulo.

7.4.1. *Array*: padrões e boas práticas

Conforme o gráfico 9, duas regras foram apontadas por pelo menos 10 respondentes como utilizadas no dia-a-dia profissional para a manipulação de *arrays*, sendo descartada apenas a que tinha relação com criação de dados deste tipo de forma literal.

Gráfico 9 - [*ARRAY*] Selecione quais padrões e práticas foram definidos pela sua equipe.



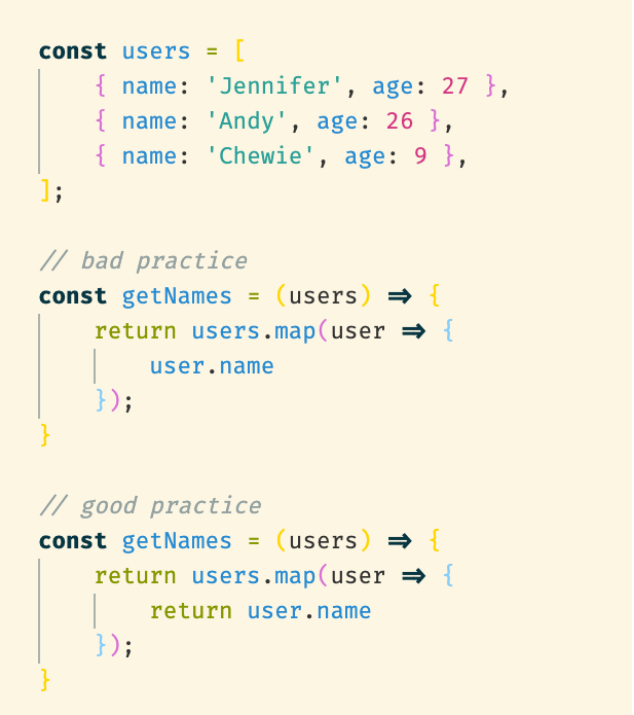
Fonte: própria Autora.

1. Sempre retornar algo quando utilizar *high order functions*: como visto em capítulos anteriores, *high order functions* são funções de ordem superior que aceitam outras funções como argumentos ou retornam funções.

Manter como prática o retorno de algum valor no final da execução de uma *HOF* se faz necessário para não obter valores inesperados na execução das linhas de código. "Se nos esquecermos de escrever a instrução de retorno em uma *HOF*, provavelmente é um erro. Se você não quiser usar um retorno ou não precisar dos resultados retornados, considere usar .*forEach*" (*ESLINT*, 2021, tradução nossa).

Na figura 51, o intuito é retornar uma lista com nomes dos usuários contidos no *array* de objetos, na variável "*users*". Utilizou-se a função "*map*" para iterar a variável e acessar o "*name*" de cada objeto. Porém, na primeira tentativa, onde não retornasse nada de dentro do "*map*" (uma função *HOF* de iteração em *array*), o resultado seria uma lista de três valores *undefined*. Na segunda tentativa, onde retornasse o "*name*" de dentro de cada objeto - concatenando com o "*array*" final -, o resultado é o esperado.

Figura 51 - Código exemplo: uso de retorno dentro de *high order function.*



```
const users = [
    { name: 'Jennifer', age: 27 },
    { name: 'Andy', age: 26 },
    { name: 'Chewie', age: 9 },
];

// bad practice
const getNames = (users) => {
    return users.map(user => {
        user.name
    });
}

// good practice
const getNames = (users) => {
    return users.map(user => {
        return user.name
    });
}
```

Fonte: própria Autora.

2. Utilizar *destructuring* para acessar várias posições de um *array*: esta prática economiza linhas de código e mantém uma clareza e elegância no projeto - como já mencionada no no capítulo Conceitos do *JavaScript ES6+*. Pode-se perceber, no exemplo da figura 52, que o primeiro exemplo acessa os valores do *array* de modo manual - acessando a posição “0, 1 e 2” -, e o segundo exemplo faz o uso do processo de desestruturação, podendo acessar valores ao mesmo tempo que atribui estes a variáveis, tudo em uma mesma linha e em sequência de posições.

Figura 52 - Código exemplo: uso de *destructuring* para acessar posições do *"array".*

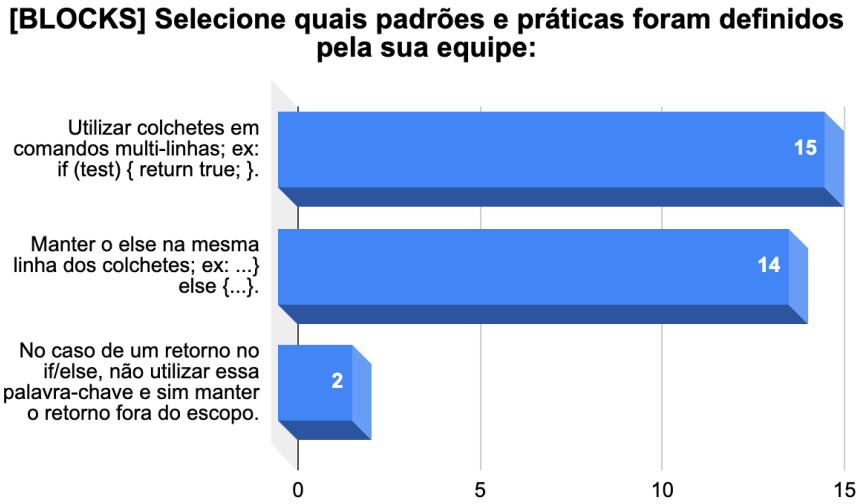
```
const names = ['Jennifer', 'Andy', 'Chewie'];

// bad practice
const name1 = names[0];
const name2 = names[1];
const name3 = names[2];

// good practice
const [name1, name2, name3] = ['Jennifer', 'Andy', 'Chewie'];
```

Fonte: própria Autora.

7.4.2. *Blocks*: padrões e boas práticas

Em relação a blocos de códigos (*blocks*) envolvendo condicionais (*if/else*), dois tópicos foram votados por cerca de 75% dos respondentes - gráfico 10, sendo utilizados a fim de manter o código mais legível para a equipe.

Gráfico 10 - [*BLOCKS*] Selecione quais padrões e práticas foram definidos pela sua equipe.



Fonte: própria Autora.

1. Utilizar chaves em comandos multi-linhas: "ao escrever *if*, *else*, *while*, *do-while* e *for*, o bloco pode estar em uma única linha em vez de um bloco" (*ESLINT*, 2021, tradução nossa). Nas situações em que se tem mais de duas linhas no bloco, a regra nos força a deixar cada comando em uma linha e manter as chaves envolvendo o bloco de código, a fim de deixar claro os limites de contexto do bloco em questão. Como visto na figura 53, a segunda opção deixa muito claro o escopo de bloco da função *bar*.

Figura 53 - Código exemplo: uso de chaves em comandos multi-linhas.

```
// bad practice
function foo() { return false; }

// good practice
function bar() {
    return false;
}
```

Fonte: própria Autora.

2. Manter o “*else*” na mesma linha das chaves: "o uso de chaves está intimamente relacionado ao estilo de indentação na programação e descreve a colocação das chaves em relação à instrução de controle e ao corpo" (*ESLINT*, 2021, tradução nossa). Este padrão considera o estilo mais comum em *JavaScript*,

onde a chave de abertura de um bloco é colocada na mesma linha de sua instrução ou declaração correspondente - figura 54.

Figura 54 - Código exemplo: uso de "*else*" na mesma linha das chaves do bloco.

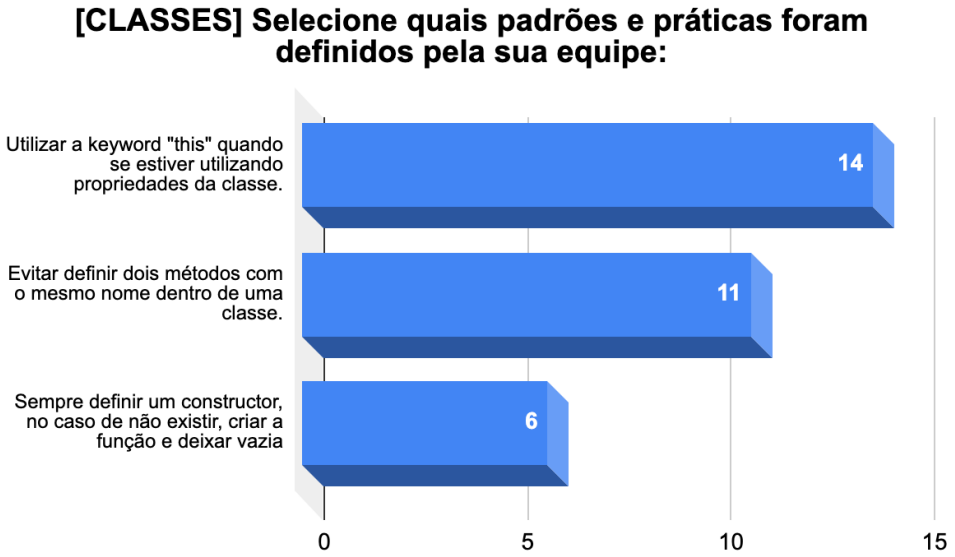
```
// bad practice
if (test) {
    thing1();
}
else {
    thing3();
}

// good practice
if (test) {
    thing1();
} else {
    thing3();
}
```

Fonte: própria Autora.

7.4.3. *Classes*: padrões e boas práticas

Acerca dos padrões para o uso de classes em *JS*, como pode-se analisar no gráfico 11, mais duas regras foram apontadas como em uso para profissionais da área de programação.

Gráfico 11 - [*CLASSES*] Selecione quais padrões e práticas foram definidos pela sua equipe.



Fonte: própria Autora.

1. Utilizar a *keyword* "*this*" quando se estiver utilizando propriedades da classe: "se um método de classe não usa "this", às vezes pode ser transformado em uma função estática" (*ESLINT*, 2021, tradução nossa), além disso o uso da palavra-chave "*this*" deixa claro o escopo a que esta pertence - figura 55.

Figura 55 - Código exemplo: uso da *keyword "this"* para acesso de propriedades da classe dentro dela.

Fonte: própria Autora.

2. Evitar definir dois métodos com o mesmo nome dentro de uma classe: "se houver declarações com o mesmo nome dentro da classe, a última declaração substitui as outras declarações silenciosamente. Isso pode causar comportamentos inesperados" (*ESLINT*, 2021, tradução nossa), exemplo na figura 56.

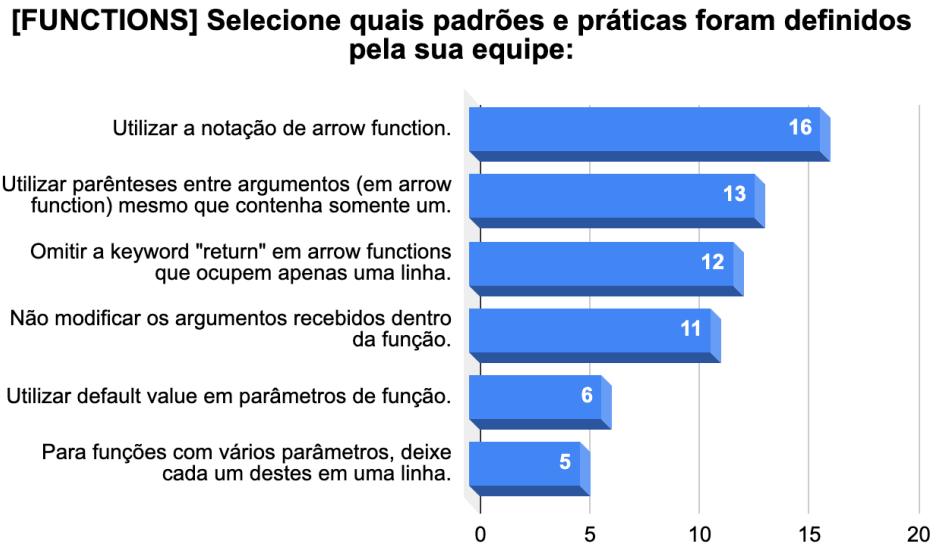
Figura 56 - Código exemplo: evitar duplicação de nomes de métodos dentro da mesma classe.

Fonte: própria Autora.

7.4.4. *Functions*: padrões e boas práticas

Sobre padrões e boas práticas na utilização de funções (*functions*) quatro regras foram as mais votadas como mais utilizadas por mais da metade dos profissionais - gráfico 12.

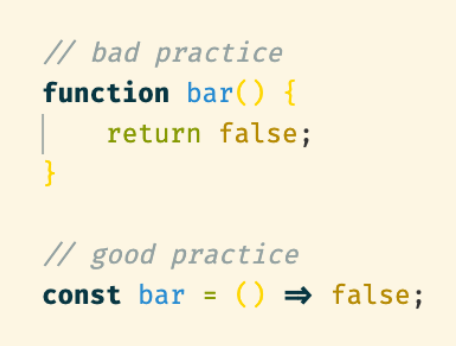
Gráfico 12 - [*FUNCTIONS*] Selecione quais padrões e práticas foram definidos pela sua equipe.



Fonte: própria Autora.

1. Utilizar a notação de *arrow function:* "*arrow functions* são menos prolixas e mais fácil de raciocinar, além de serem vinculadas lexicalmente, independentemente de onde ou quando eles são chamados" (*ESLINT*, 2021, tradução nossa). Vale ressaltar que, quando o desenvolvedor precisa acessar o escopo da própria função, deve-se optar pelo uso da função tradicional. Para ilustrar essa regra segue a figura 57.

Figura 57 - Código exemplo: uso da notação de *arrow function.*

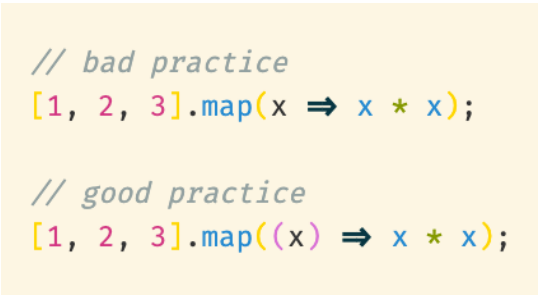


```
// bad practice
function bar() {
    return false;
}

// good practice
const bar = () => false;
```

Fonte: própria Autora.

2. Utilizar parênteses entre argumentos (em *arrow function*) mesmo que contenha somente um: "*arrow functions* podem omitir parênteses quando têm exatamente um parâmetro. Em todos os outros casos, os parâmetros devem ser colocados entre parênteses" (*ESLINT*, 2021, tradução nossa), como visto na figura 58.

Figura 58 - Código exemplo: uso de parênteses entre argumentos mesmo que a *arrow function* contenha apenas um.



```
// bad practice
[1, 2, 3].map(x => x * x);

// good practice
[1, 2, 3].map((x) => x * x);
```

Fonte: própria Autora.

3. Omitir a *keyword* "*return*" em *arrow functions* que ocupem apenas uma linha: "as *arrow functions* têm duas formas sintáticas para seus corpos de função. Eles podem ser definidos com um corpo de bloco () => {...} ou com uma única expressão () => ..., cujo valor é retornado implicitamente" (*ESLINT*, 2021, tradução nossa). Reescrevendo a função exemplo da figura 59 com esta regra o código aparenta mais clareza.

Figura 59 - Código exemplo: omissão da *keyword "return"* em *arrow functions* que ocupem apenas uma linha.

```
// bad practice
[1, 2, 3].map((number) => {
    const nextNumber = number + 1;
    `A string containing the ${nextNumber}.`;
});

// good practice
[1, 2, 3].map((number) => `A string containing the ${number + 1}.`);
```

Fonte: própria Autora.

4. Não modificar os argumentos recebidos dentro da função: re-atribuir valores de argumentos recebidos "pode ser enganoso e levar a um comportamento confuso, pois a modificação dos parâmetros da função também altera o objeto de argumentos" (*ESLINT*, 2021, tradução nossa). Na figura 60, ao invés de modificar o argumento recebido na função, copia-se o valor de referência deste e faz-se as modificações necessárias, evitando efeitos colaterais.

Figura 60 - Código exemplo: evitando modificar os argumentos recebidos dentro da função.

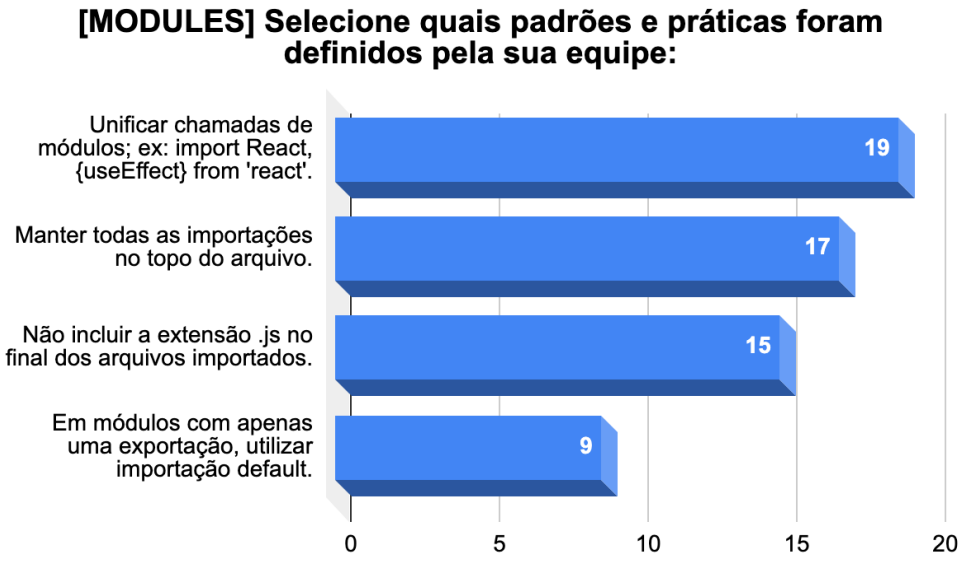
```
// bad practice
function f1(obj) {
    obj.key = 1;
}

// good practice
function f2(obj) {
    let obj1 = obj;
    obj1 = 1;
}
```

Fonte: própria Autora.

7.4.5. *Modules*: padrões e boas práticas

A respeito de exportação de módulos (*modules*) em arquivos no JavaScript, apenas uma regra não é utilizada por mais de 10 profissionais da área - como visto no gráfico 13 -, tendo, assim, três regras adicionadas à lista final.

Gráfico 13 - [*MODULES*] Selecione quais padrões e práticas foram definidos pela sua equipe.



Fonte: própria Autora.

1. Unificar chamadas de módulos: "utilizar uma única instrução de importação por módulo tornará o código mais claro, porque você pode ver tudo sendo importado desse módulo em uma linha" (*ESLINT*, 2021, tradução nossa). Como no exemplo da figura 61, onde todos os módulos que deseja-se importar do pacote "*foo"* são importados na mesma linha.

Figura 61 - Código exemplo: unificar chamadas de módulos.

```
// bad practice
import foo from 'foo';
import { named1, named2 } from 'foo';

// good practice
import foo, { named1, named2 } from 'foo';
```

Fonte: própria Autora.

2. Manter todas as importações no topo do arquivo: importações sofrem o efeito do *hoisting*, o içamento de código que ocorre no *JavaScript*. Manter essas instruções no topo do código pode evitar efeitos indesejados - figura 62.

Figura 62 - Código exemplo: manter importações de módulos no topo do arquivo.

```
// bad practice
import foo from './foo';
initWith(foo);
import bar from './bar';


// good practice
import foo from './foo';
import bar from './bar';
initWith(foo);
```

Fonte: própria Autora.

3. Não incluir a extensão *.js* no final dos arquivos importados: utilizando algoritmos para resolver o caminho de arquivos importados permite a omissão da extensão destes, como na figura 63.

Figura 63 - Código exemplo: omissão da extensão *.js* para importação de arquivos.

```
// bad practice
import foo from './foo.js';
import bar from './bar.jsx';
import baz from './baz/index.jsx';

// good practice
import foo from './foo';
import bar from './bar';
import baz from './baz';
```

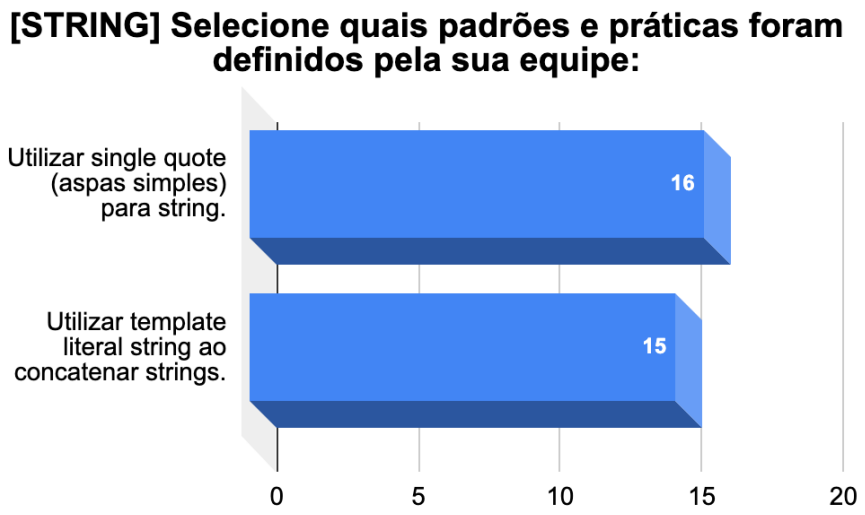
Fonte: própria Autora.

7.4.6. *Strings*: padrões e boas práticas

A respeito de manipulação e uso de *strings* (caracteres de texto), as

2 regras propostas pela bibliográfica foram apontadas como utilizadas por pelo menos 75% dos respondentes, como visto no gráfico 14.

Gráfico 14 - [*STRING*] Selecione quais padrões e práticas foram definidos pela sua equipe.



Fonte: própria Autora.

1. Utilizar *single quote* (aspas simples) para *string*: "o *JavaScript* permite definir *strings* de uma das três maneiras: aspas duplas, aspas simples e crases (a partir do *ECMAScript 6*)" (*ESLINT*, 2021, tradução nossa). Para evitar uma confusão, adicionar essa regra no *ESLint* garante que apenas a aspas simples seja utilizada neste cenário - como visto na figura 64.

Figura 64 - Código exemplo: uso de *single quotes* para "*strings*".

```
// bad practice
const name = "Jennifer Takagi";

// bad practice
const name = `Jennifer Takagi`;

// good practice
const name = 'Jennifer Takagi';
```

Fonte: própria Autora.

2. Utilizar *template literal string* ao concatenar *strings*: o uso desse método foi apresentado a partir do *ES6*, substituindo o uso de soma para unir textos, como visto em capítulos anteriores e na figura 65.

Figura 65 - Código exemplo: uso de *template literal* para concatenar "*strings*".

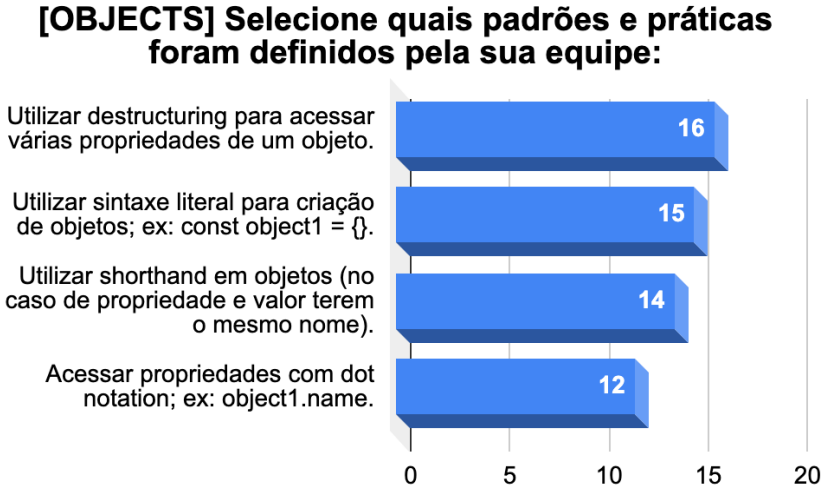
```
// bad practice
function sayHi(name) {
    return 'How are you, ' + name + '?';
}

// good practice
function sayHi(name) {
    return `How are you, ${name}?`;
}
```

Fonte: própria Autora.

7.4.7. *Objects*: padrões e boas práticas

Acerca dos padrões para o uso de objetos (*objects*) todas as regras propostas no questionário foram validadas como utilizadas por pelo menos 50% dos profissionais de programação, agregando-se mais 4 regras - gráfico 15.

Gráfico 15 - [*OBJECTS*] Selecione quais padrões e práticas foram definidos pela sua equipe.



Fonte: própria Autora.

1. Utilizar *destructuring* para acessar várias propriedades de um

objeto: "a desestruturação de objetos fornece um único local de definição da estrutura do objeto que é usado no bloco, em vez de exigir a leitura de todo o bloco para determinar o que é usado" (*ESLINT*, 2021, tradução nossa). Essa prática traz os mesmos benefícios relatados no escopo do *array*, mantendo um código menos repetitivo e mais objetivo - representação gráfica na figura 66.

Figura 66 - Código exemplo: uso de *destructuring* para acessar propriedades de um "objeto".

```
// bad practice
function getFullName(user) {
    const firstName = user.firstName;
    const lastName = user.lastName;

    return `${firstName} ${lastName}`;
}

// good practice
function getFullName(user) {
    const { firstName, lastName } = user;
    return `${firstName} ${lastName}`;
}
```

Fonte: própria Autora.

2. Utilizar sintaxe literal para criação de objetos: para a criação de um objeto pode-se decidir instanciar o construtor, como no primeiro exemplo da figura 67, usando a palavra-chave *"new"*, ou pode-se optar pela criação de forma literal. "Embora não haja diferenças de desempenho entre as duas abordagens, a economia de *bytes* e a concisão ao optar pela criação com a forma literal é o que a torna a mais adequada " (*ESLINT*, 2021, tradução nossa).

Figura 67 - Código exemplo: uso de sintaxe literal para criação de "objetos".

```
// bad practice
const item = new Object();

// good practice
const item = {};
```

Fonte: própria Autora.

3. Utilizar *shorthand* em objetos: "*ECMAScript 6* fornece uma maneira mais concisa para definir métodos e propriedades literais de objetos. Essa sintaxe pode tornar a definição de literais de objetos complexos muito mais limpa" (*ESLINT*, 2021, tradução nossa). Resumidamente, no cenário em que o nome da propriedade do objeto e seu valor são os mesmos o desenvolvedor pode utilizar o *shorthand*, escrevendo apenas a propriedade, o *JS* entenderá que a chave recebe como valor a variável de mesmo nome - como visto na figura 68.

Figura 68 - Código exemplo: uso de *shorthand* em "objetos".

```
const lukeSkywalker = 'Luke Skywalker';

// bad practice
const obj = {
  lukeSkywalker: lukeSkywalker,
};

// good practice
const obj = {
  lukeSkywalker,
};
```

Fonte: própria Autora.

4. Acessar propriedades com *dot notation*: "em *JS*, é possível acessar propriedades usando *dot notation* ou *square-bracket notation*. No entanto, é preferível utilizar a notação de ponto porque é mais fácil de ler, menos prolixo e funciona melhor com minimizadores" (*ESLINT*, 2021, tradução nossa), figura 69. Para o caso de acesso a chaves dinâmicas deve-se utilizar o *square-bracket notation.*

Figura 69 - Código exemplo: acesso de propriedades de um "objeto" com *dot notation*.

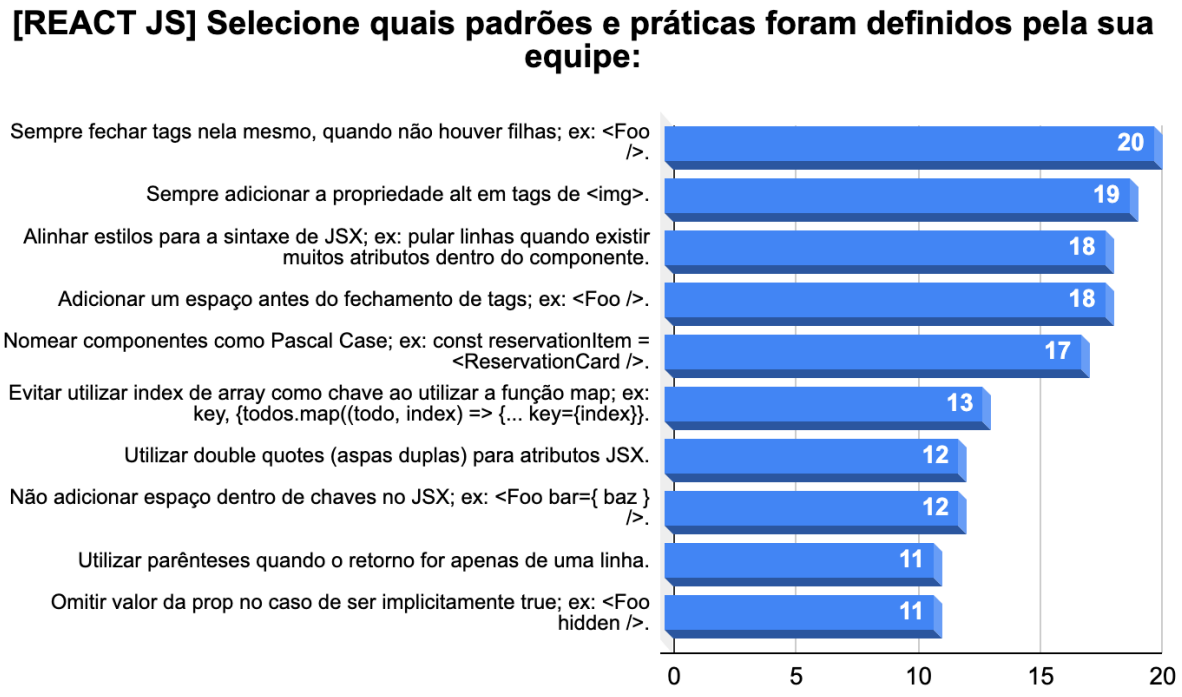
```
const luke = {
    jedi: true,
    age: 28,
  };

// bad practice
const isJedi = luke['jedi'];

// good practice
const isJedi = luke.jedi;
```

Fonte: própria Autora.

7.4.8. *React JS*: padrões e boas práticas

Quanto às boas práticas e padrões de código utilizando a *library React JS*, todas as 10 regras apresentadas no questionário foram selecionadas, baseando-se no fato de que pelo menos metade dos respondentes indicaram utilizá-las. Como visto no gráfico 16, 5 regras são utilizadas por pelo menos 85% dos profissionais.

Gráfico 16 - [*REACT JS*] Selecione quais padrões e práticas foram definidos pela sua equipe.



Fonte: própria Autora.

1. Sempre fechar *tags* nela mesmo, quando não houver filhas: para manter uma legibilidade em códigos *React JS* pode-se fechar a *tag* nela mesma, assim como usado em *HTML*, quando esta não tiver tags filhas dentro dela - figura 70.

Figura 70 - Código exemplo: fechamento de *tag* de componentes *React JS* nela mesma quando não houver *tags* filhas.

```
// bad practice
const HelloPerson = <Hello name="Jennifer"></Hello>;

// good practice
const HelloPerson = <Hello name="Jennifer" />;
```

Fonte: própria Autora.

2. Sempre adicionar a propriedade *alt* em *tags* de <*img*>: quando utilizada *tags* de imagem é importante adicionar o atributo *"alt"* a nível de acessibilidade, essa propriedade é lida por leitores de tela no caso de pessoas com alguma deficiência visual - exemplo de aplicação na figura 71.

Figura 71 - Código exemplo: adição de texto para o atributo *"alt"* em *tags* de imagem.

```
// bad practice
<img src="hello.jpg" />

// good practice
<img src="hello.jpg" alt="Me waving hello" />
```

Fonte: própria Autora.

3. Alinhar estilos para a sintaxe de *JSX*: é importante definir estilos para a sintaxe de *JSX*, como por exemplo: pular linha quando houver mais de uma prop dentro do componente, fechar *tags* na mesma linha em que ela foi aberta - figura 72.

Figura 72 - Código exemplo: alinhamento de estilos para sintaxe de *JSX*.

```
// bad practice
<Foo superLongParam="bar"
  anotherSuperLongParam="baz" />

// good practice
<Foo
  superLongParam="bar"
  anotherSuperLongParam="baz"
/>
```

Fonte: própria Autora.

4. Adicionar um espaço em branco antes do fechamento de *tags*: ao fechar *tags* em componentes do *React JS*, deixar um espaço vazio antes do fechamento, somente para manter um respiro harmonioso para o código - figura 73.

Figura 73 - Código exemplo: adição de um espaço em branco antes do fechamento de *tags* no *React JS*.

```
// bad practice
<Foo/>

// very bad practice
<Foo                 />

// good practice
<Foo />
```

Fonte: própria Autora.

5. Nomear componentes como *Pascal Case*: utilizar o padrão *Pascal Case* para nomear componentes do *React JS* a fim de diminuir a bagunça de sintaxe no código, exemplo de uso na figura 74.

Figura 74 - Código exemplo: nomeação de componentes como *Pascal Case*.

```
// bad practice
import reservationCard from './ReservationCard';

// good practice
import ReservationCard from './ReservationCard';
```

Fonte: própria Autora.

6. Evitar utilizar *index* de *array*: "as *keys* ajudam o *React* a identificar quais itens foram alterados, adicionados ou removidos. As chaves devem ser fornecidas aos elementos dentro da matriz" (FACEBOOK OPEN SOURCE, 2021, tradução nossa). Ou seja, sempre que se utilizar uma iteração para renderização de vários componentes na *library*, deve-se passar uma chave (*key*) para cada um deles.

Não é aconselhável usar apenas o *"index"* da iteração por não ser único, se houver outra lista de componentes e que utilizem a mesma estratégia, haverá prejuízo na renderização, podendo acrescentar mais tempo de carregamento até o *React* identificar qual elemento sofreu a mudança,. Usar *id* é uma saída - figura 75.

Figura 75 - Código exemplo: evitar uso de "*index* de *array"* para identificar chaves de elementos.

```
// bad practice
{todos.map((todo, index) =>
    <Todo
      {...todo}
      key={index}
    />
)}

// good practice
{todos.map(todo => (
<Todo
    {...todo}
    key={todo.id}
/>
))}
```

Fonte: própria Autora.

7. Utilizar *double quotes* (aspas duplas) para atributos *JSX*: optar por utilizar aspas duplas em atributos das *tags JSX* (figura 76), segundo o padrão de atributos de *tags HTML*, e utilizando aspas simples para outras *strings* no código *JavaScript*, como visto anteriormente.

Figura 76 - Código exemplo: uso de *double quote* em atributos *JSX*.

```
// bad practice
<Foo bar='bar' />

// good practice
<Foo bar="bar" />
```

Fonte: própria Autora.

8. Não adicionar espaço dentro de chaves no *JSX*: regra para desabilitar espaços extras dentro de chaves nos componentes *JSX*, como visto na figura 77.

Figura 77 - Código exemplo: evitar espaços desnecessários dentro de chaves no *JSX*.

```
// bad practice
<Foo bar={ baz } />

// good practice
<Foo bar={baz} />
```

Fonte: própria Autora.

9. Utilizar parênteses quando o retorno ocupar mais de uma linha: envolver as *tags* do componente dentro de parênteses quando ocuparem mais de uma linha facilita a leitura (figura 78). No caso de ser uma *tag* que ocupe apenas uma linha, não se faz necessário o caráter.

Figura 78 - Código exemplo: envolvimento do retorno de componentes em parênteses no caso de ocupar mais de uma linha.

```
// bad practice
render() {
  return <MyComponent variant="long body" foo="bar">
    <MyChild />
  </MyComponent>;
}

// good practice
render() {
  return (
    <MyComponent variant="long body" foo="bar">
      <MyChild />
    </MyComponent>
  );
}
```

Fonte: própria Autora.

10. Omitir valor da *"prop"* no caso de ser implicitamente *"true":* segundo o site oficial do *React JS* (FACEBOOK OPEN SOURCE, 2021, tradução nossa) ao utilizar um atributo booleano em *JSX*, pode-se definir o valor do atributo como verdadeiro ou omitir o valor - figura 79.

Figura 79 - Código exemplo: omissão do valor da *"prop"* no caso de esta ser implicitamente de valor "*true*".

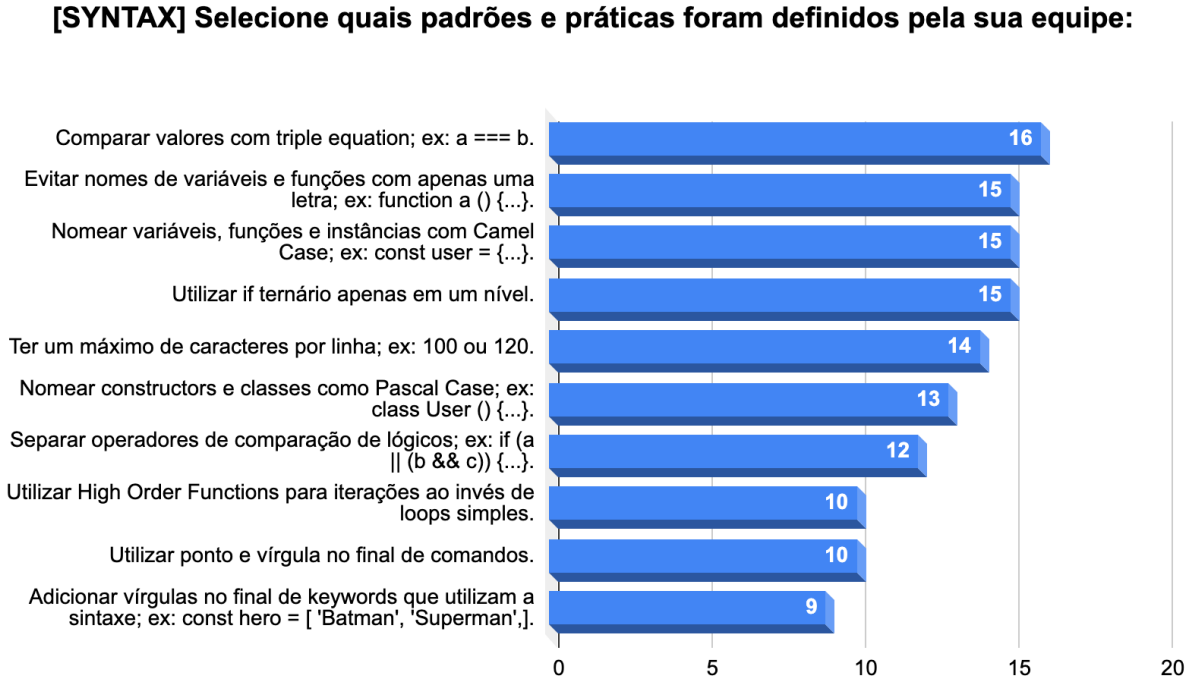
```
// bad practice
<Foo
  hidden={true}
/>

// good practice
<Foo hidden />
```

Fonte: própria Autora.

7.4.9. *Syntax:* padrões e boas práticas

Acerca de padrões de sintaxe (*syntax*), apenas um não foi escolhido pela nota de corte, resultando em nove regras selecionadas, como visto no gráfico 17.

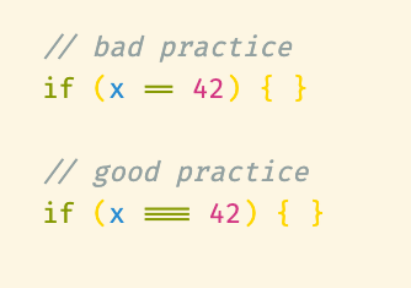
Gráfico 17 - [*SYNTAX*] Selecione quais padrões e práticas foram definidos pela sua equipe.



Fonte: própria Autora.

1. Comparar valores com *triple equation*: é uma boa prática utilizar triple equation ao invés de comparação simples, "a razão para isso é que == e != fazem coerção de tipo que segue o Algoritmo de Comparação de Igualdade Abstrata" (*ESLINT*, 2021, tradução nossa). Como visto em capítulos anteriores, essa prática garante que valores e tipos serão comparados, como visto na figura 80.

Figura 80 - Código exemplo: uso de *triple equation* para comparação de valores.



Fonte: própria Autora.

2. Evitar nomes de variáveis e funções com apenas uma letra: "nomes de identificadores muito curtos [...] ou muito longos [...] podem tornar o código mais difícil de ler e potencialmente menos sustentável" (*ESLINT*, 2021, tradução nossa). Como na figura 81, onde o nome da função ser "q" não diz o motivo de ela ter sido criada, substituindo para *"getQuery"* já deixa claro que a função está pegando algum termo.

Figura 81 - Código exemplo: evitar nomes de variáveis e funções com apenas uma letra.

Fonte: própria Autora.

3. Nomear variáveis, funções e instâncias com *Camel Case*: para não bagunçar o código com diferentes maneiras sintáticas de nomeação, sugere-se que use apenas o *Camel Case* em variáveis, funções e instâncias (figura 82).

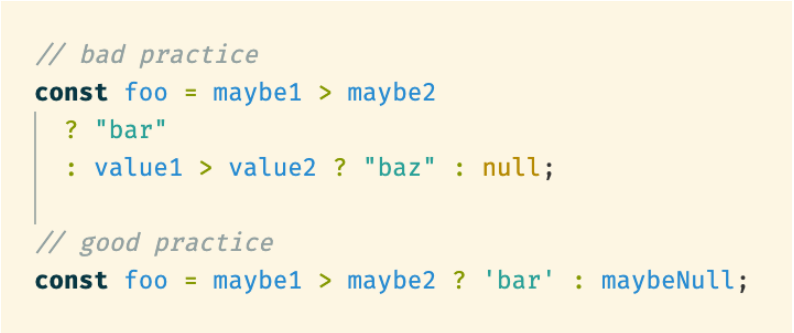
Figura 82 - Código exemplo: uso de *Camel Case* para nomeação de variáveis, funções e instâncias.

```
// bad practice
const OBJEcttssssss = {};
const this_is_my_object = {};
function c() {}

// good practice
const thisIsMyObject = {};
function thisIsMyFunction() {}
```

Fonte: própria Autora.

4. Utilizar operador ternário em apenas um nível: o uso de condicionais ternárias em vários níveis pode acabar poluindo o código e tornando o código difícil de entender, na figura 83 expõe-se um caso de uso para evitar esse cenário.

Figura 83 - Código exemplo: uso de operador ternário em apenas um nível.



```
// bad practice
const foo = maybe1 > maybe2
  ? "bar"
  : value1 > value2 ? "baz" : null;

// good practice
const foo = maybe1 > maybe2 ? 'bar' : maybeNull;
```

Fonte: própria Autora.

5. Ter um máximo de caracteres por linha: linhas muito longas no código podem dificultar a leitura do comando todo (figura 84), tendo que rolar a página do editor de código para visualizá-las. "A fim de ajudar na legibilidade e manutenção, muitos codificadores desenvolveram uma convenção para limitar as linhas de código a um número X de caracteres (tradicionalmente 80 caracteres)" (*ESLINT*, 2021, tradução nossa).

Figura 84 - Código exemplo: definição de um máximo de caracteres por linha de código.

```
// bad practice
const foo = jsonData && jsonData.foo && jsonData.foo.bar && jsonData.foo.bar.baz && jsonData.foo.bar.baz.quux;

// good practice
const foo = jsonData
  && jsonData.foo
  && jsonData.foo.bar
  && jsonData.foo.bar.baz
  && jsonData.foo.bar.baz.quux
  && jsonData.foo.bar.baz.quux.xyzzy;
```

Fonte: própria Autora.

6. Nomear *constructors* e classes como *Pascal Case*: "as funções nativas do *JavaScript* começam com uma letra maiúscula para distinguir as funções que devem ser usadas como construtores das funções que não são" (*ESLINT*, 2021, tradução nossa). Na figura 85 ilustra-se bem a sentença do site oficial do *ESLint*, onde a função "*user*" está sendo instanciada como uma classe. Porém é facilmente confundida com uma função tradicional. No exemplo de boa prática, o uso da primeira letra maiúscula faz associação a instâncias com construtores.

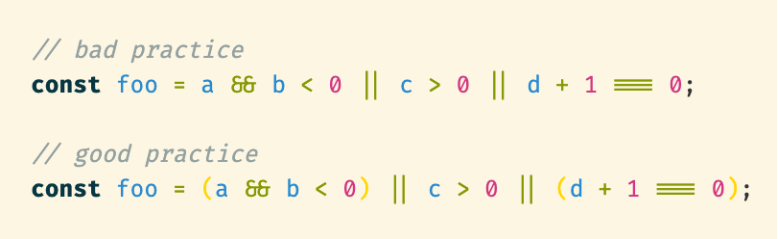
Figura 85 - Código exemplo: uso de *Pascal Case* para nomeação de construtores e classes.

```
// bad practice
function user(options) {
    this.name = options.name;
}

const bad = new user({ name: 'nope' });

// good practice
class User {
    constructor(options) {
        this.name = options.name;
    }
}

const good = new User({ name: 'yup' });
```

Fonte: própria Autora.

7. Separar operadores de comparação de lógicos: "incluir expressões complexas entre parênteses esclarece a intenção do desenvolvedor, o que torna o código mais legível" (*ESLINT*, 2021, tradução nossa). No exemplo da figura 86 fica claro a ordem de leitura e acontecimentos das operações lógicas ao fazer a separação por blocos de parênteses.

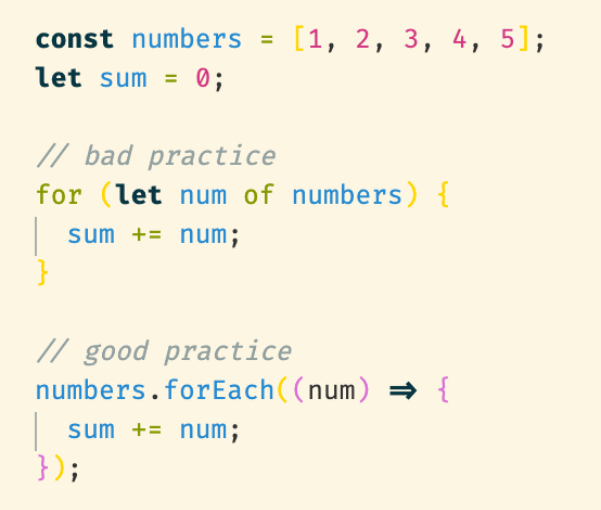
Figura 86 - Código exemplo: separação de operadores de comparação de lógicos.



Fonte: própria Autora.

8. Utilizar *High Order Functions* para iterações ao invés de *"loops"* simples: essa regra traz a preferência pelo uso de *HOF* em vez de *loops* como "*for-in"* ou *for-of* a fim de reforçar nossa regra imutável. Na figura 87, opta-se pelo do método "*forEach"* para somar números de um array, podendo ser melhorada para um uso de "*reduce"*, mantendo a imutabilidade.

Figura 87 - Código exemplo: uso de *HOF* ao invés de *"loops"* simples.



Fonte: própria Autora.

9. Utilizar ponto e vírgula no final de comandos: "*JavaScript* não

requer ponto-e-vírgula no final de cada instrução. Em muitos casos, o mecanismo *JavaScript* pode determinar que um ponto-e-vírgula deve estar em um determinado local e irá adicioná-lo automaticamente" (*ESLINT*, 2021, tradução nossa). Esse comportamento pode interromper o código no caso de interpretar incorretamente alguma quebra de linha. A maneira de evitar este erro é finalizar comandos com ponto e vírgula (figura 88).

Figura 88 - Código exemplo: uso de ponto e vírgula no final de comandos.

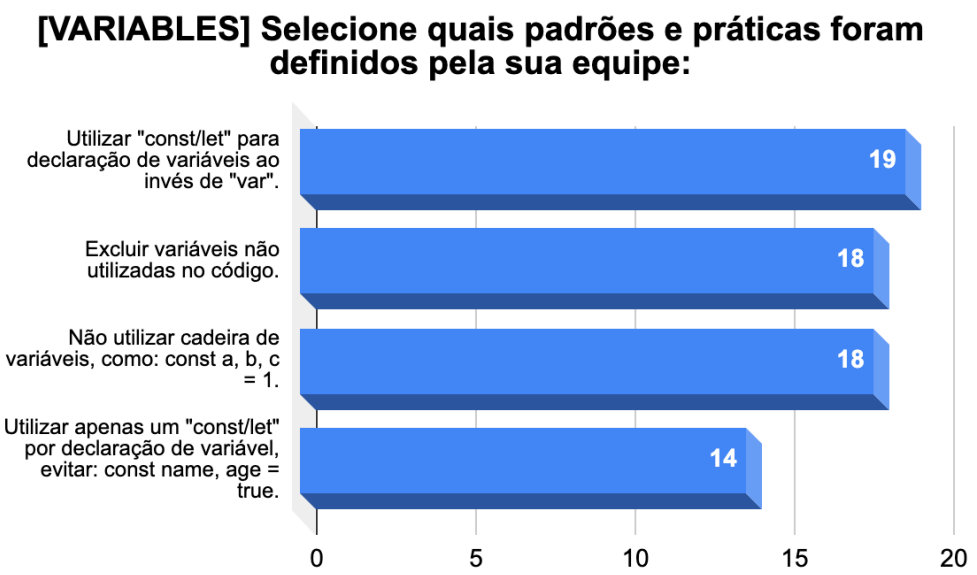
```
// bad practice
const luke = {}
const leia = {}
[luke, leia].forEach((jedi) => jedi.father = 'vader')

// good practice
const luke = {};
const leia = {};
[luke, leia].forEach((jedi) => {
  jedi.father = 'vader';
});
```

Fonte: própria Autora.

7.4.10. *Variables:* padrões e boas práticas

Quanto a práticas utilizadas no uso de variáveis (*variables*), todas as regras apresentadas por estudos bibliográficos foram selecionadas por, pelo menos, 14 respondentes, somando mais quatro à lista para implementação - vide gráfico 18 .

Gráfico 18 - [*VARIABLES*] Selecione quais padrões e práticas foram definidos pela sua equipe.



Fonte: própria Autora.

1. Utilizar *"const/let"* para declaração de variáveis ao invés de *"var": "*se uma variável nunca for re-atribuída, usar a declaração *"const"* é melhor. A declaração "*const"* diz aos leitores: esta variável nunca é reatribuída" (*ESLINT*, 2021, tradução nossa). No caso de retribuição de valores a variável, é preferível utilizar *"let"* para manter o contexto léxico (figura 89).

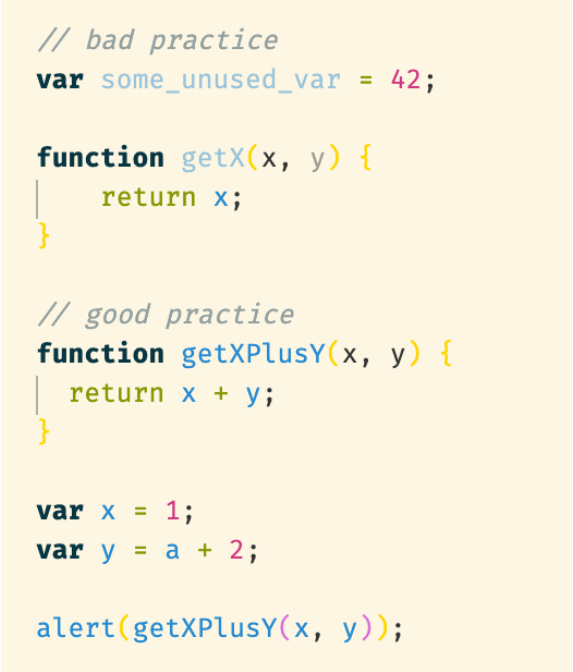
Figura 89 - Código exemplo: uso de *"const/let"* na declaração de variáveis.

```
// bad practice
var a = 1;
var b = 2;

var count = 1;
if (true) {
  count += 1;
}

// good practice
const a = 1;
const b = 2;

let count = 1;
if (true) {
  count += 1;
}
```

Fonte: própria Autora.

2. Excluir variáveis não utilizadas no código: "variáveis que são declaradas e não usadas em nenhum lugar do código são provavelmente um erro devido à refatoração incompleta. Essas variáveis ocupam espaço no código e podem causar confusão aos leitores" (*ESLINT*, 2021, tradução nossa). Como visto na figura 90, funções sofrem o mesmo problema ao serem declaradas e nunca utilizadas.

Figura 90 - Código exemplo: remoção de variáveis não utilizadas no código.



```
// bad practice
var some_unused_var = 42;

function getX(x, y) {
    return x;
}

// good practice
function getXPlusY(x, y) {
  return x + y;
}

var x = 1;
var y = a + 2;

alert(getXPlusY(x, y));
```

Fonte: própria Autora.

3. Não utilizar cadeia de variáveis: essa prática prejudica a leitura do código. Como visto na figura 91, é mais claro cada variável estar em uma linha por mais que recebam o mesmo valor.

Figura 91 - Código exemplo: evitar uso de cadeia de variáveis.

```
// bad practice
(function example() {
    let a = b = c = 1;
}());

// good practice
(function example() {
    let a = 1;
    let b = a;
    let c = a;
}());
```

Fonte: própria Autora.

4. Utilizar apenas um *"const/let"* por declaração de variável: "existem muitos estilos e preferências relacionados à declaração de variáveis, e um deles é decidir quantas declarações de variáveis devem ser permitidas em uma única função" (*ESLINT*, 2021, tradução nossa). Na figura 92, é preferível escrever uma palavra-chave de declaração de variável para cada linha ao invés de omitir essa informação.

Figura 92 - Código exemplo: uso de apenas uma *keyword "const/let"* por declaração de variável.

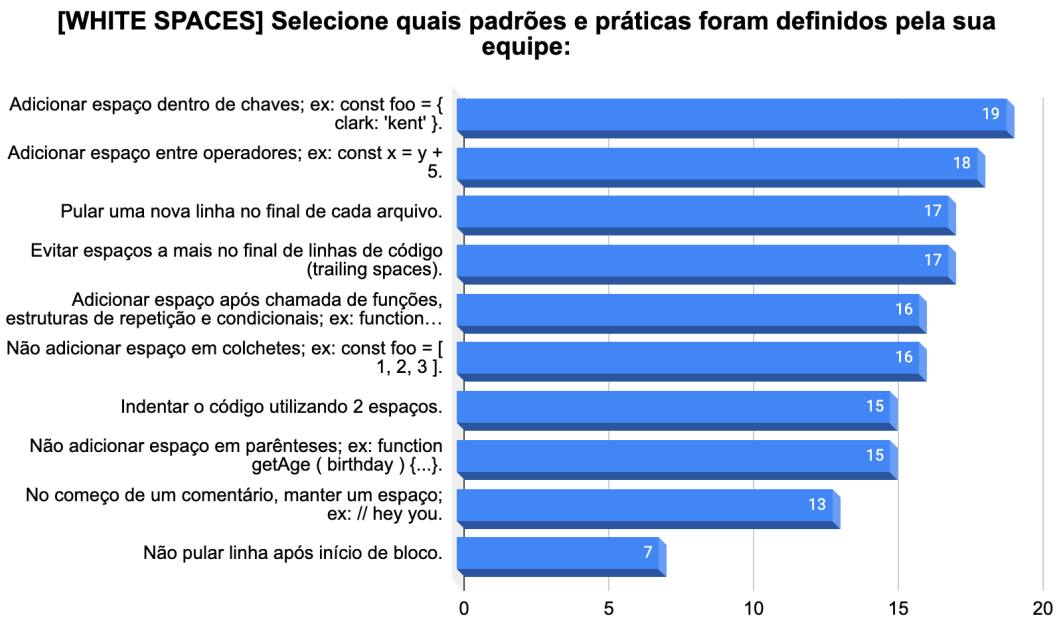
```
// bad practice
const items = getItems(),
    goSportsTeam = true,
    dragonball = 'z';


// good practice
const items = getItems();
const goSportsTeam = true;
const dragonball = 'z';
```

Fonte: própria Autora.

7.4.11. *White Spaces:* padrões e boas práticas

O último bloco de regras de padrões e boas práticas refere-se a espaços em branco (*white spaces*). Omitiu-se apenas uma regra. As outras nove foram selecionadas por pelo menos 13 profissionais (gráfico 19).

Gráfico 19 - [*WHITE SPACES*] Selecione quais padrões e práticas foram definidos pela sua equipe.



Fonte: própria Autora.

1. Adicionar espaço dentro de chaves: "embora as preferências de formatação sejam muito pessoais, vários *style guides* exigem ou não permitem espaços entre chaves" (*ESLINT*, 2021, tradução nossa) - figura 93.

Figura 93 - Código exemplo: uso de espaço em branco dentro de chaves.

```
// bad practice
const foo = {clark: 'kent'};

// good practice
const foo = { clark: 'kent' };
```

Fonte: própria Autora.

2. Adicionar espaço entre operadores: da mesma forma que a regra anterior, é totalmente pessoal adicionar espaços entre operadores, porém a prática deixa o código mais legível, como na figura 94.

Figura 94 - Código exemplo: uso de espaço em branco entre operadores.

Fonte: própria Autora.

3. Pular uma nova linha no final de cada arquivo: "os benefícios de novas linhas finais incluem a capacidade de concatenar ou anexar a arquivos, bem como arquivos de saída para o terminal sem interferir nos *prompts* do *shell*" (*ESLINT*, 2021, tradução nossa). O espaço para nova linha deve ser adicionado no final da última linha que contém comando, como no exemplo da figura 95.

Figura 95 - Código exemplo: pular uma nova linha no final de cada arquivo.

Fonte: própria Autora.

4. Evitar espaços a mais no final de linhas de código (*trailing spaces*): "espaços em branco podem ser detectados por sistemas de controle de origem e marcadas como *diffs*, causando frustração para os desenvolvedores. Embora esse espaço em branco extra não cause problemas funcionais" (*ESLINT*, 2021, tradução nossa). No editor *Visual Studio Code*, espaços são representados por pequenos pontos. Na figura 96, o primeiro exemplo apresenta *trailing spaces*.

Figura 96 - Código exemplo: evitar espaços desnecessários no código.

Fonte: própria Autora.

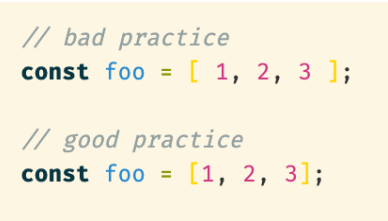
5. Adicionar espaço após chamada de funções, estruturas de repetição e condicionais: "*keywords* têm um significado especial para a linguagem e, com frequência, aparecem em uma cor diferente nos editores de código [...] os *style guides* geralmente se referem ao espaçamento que deve ser usado em torno das *keywords*" (*ESLINT*, 2021, tradução nossa) - figura 97.

Figura 97 - Código exemplo: uso de espaço em branco após *keywords*.

Fonte: própria Autora.

6. Não adicionar espaço em colchetes: "vários *style guides* exigem ou não permitem espaços entre colchetes de *array* e outros *tokens*. Esta regra se aplica a literais de matriz e atribuições de desestruturação" (*ESLINT*, 2021, tradução nossa) - figura 98.

Figura 98 - Código exemplo: evitar uso de espaços em branco dentro de colchetes.



Fonte: própria Autora.

7. Indentar o código utilizando dois espaços: é importante definir uma regra para indentação para que o código não fique desalinhado, com uma parte do time utilizando dois espaços e outra parte utilizando quatro espaços, por exemplo na figura 99.

Figura 99 - Código exemplo: indentação de código utilizando dois espaços.

Fonte: própria Autora.

8. Não adicionar espaço em parênteses: alguns *style guides* não permitem espaço em branco dentro dos parênteses, como na figura 100.

Figura 100 - Código exemplo: evitar uso de espaços em branco dentro de parênteses.

Fonte: própria Autora.

9. No começo de um comentário, manter um espaço: "o espaço em branco após // ou / * facilita a leitura do texto nos comentários" (*ESLINT*, 2021, tradução nossa) - figura 101.

Figura 101 - Código exemplo: adição de espaço em branco no começo de um comentário.

```
// bad practice
/**
 *make() returns a new element
 *based on the passed-in tag name
 */
function make(tag) {
  // ...
  return element;
}

// good practice
/**
 * make() returns a new element
 * based on the passed-in tag name
 */
function make(tag) {
  // ...
  return element;
}
```

Fonte: própria Autora.

8. IMPLEMENTAÇÃO DE DIRETRIZES NA FERRAMENTA *ESLINT* EM PROJETO ESCRITO COM *REACT JS*

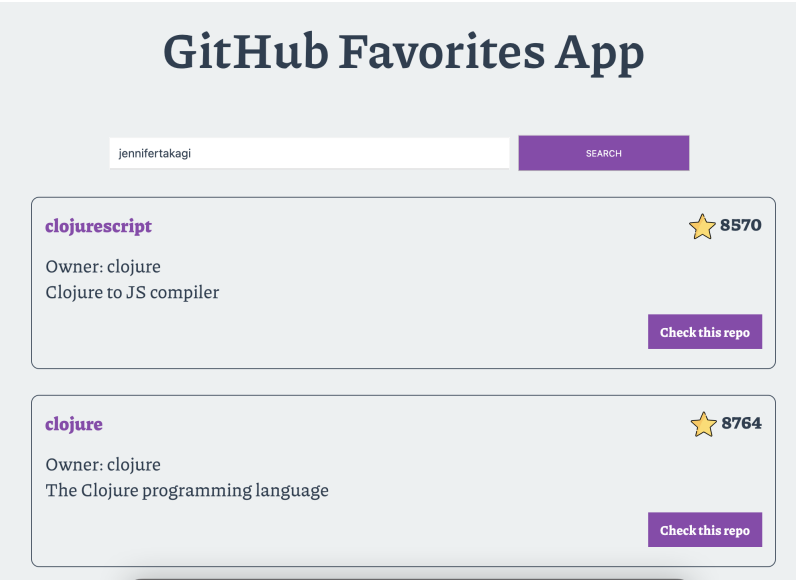
Este capítulo é dedicado à implementação das diretrizes de padrões de código e boas práticas da programação na linguagem *JavaScript* e na *library React JS*. Os padrões foram selecionados no capítulo anterior por meio de um questionário *online*.

8.1. Projeto *Github Favorites*

O projeto utilizado neste capítulo para implementação da ferramenta *ESLint* junto com a definição de regras de sintaxe está disponível no *site GitHub* da autora com acesso público com o nome *GitHub Favorites*, disponível no *link*: https://github.com/jennifertakagi/github-favorites. Para utilizá-lo, basta fazer um clone do projeto ou fazer o *download* do projeto como *zip*.

A ideia do projeto é mostrar quais os repositórios do *GitHub* que foram favoritados pelo usuário digitado no *"input"* da aplicação. Na figura 102 ilustra-se o comportamento da aplicação após inclusão de usuário válido e clique no botão de busca. Cada cartão de repositório tem um *link* externo para o seu endereço no *GitHub*.

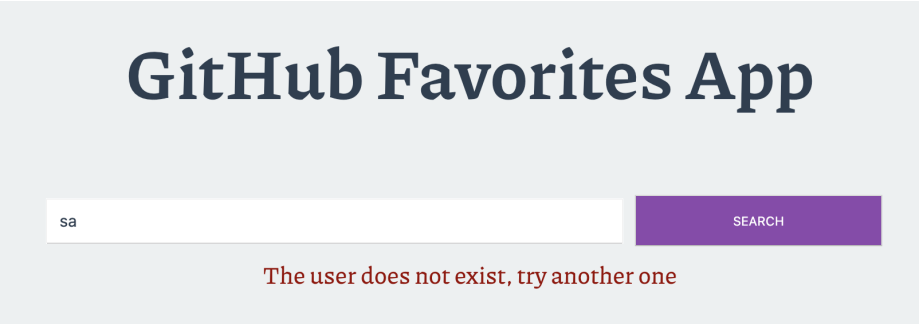
Figura 102 - Tela da aplicação *Github Favorites* no cenário de sucesso.



Fonte: própria Autora.

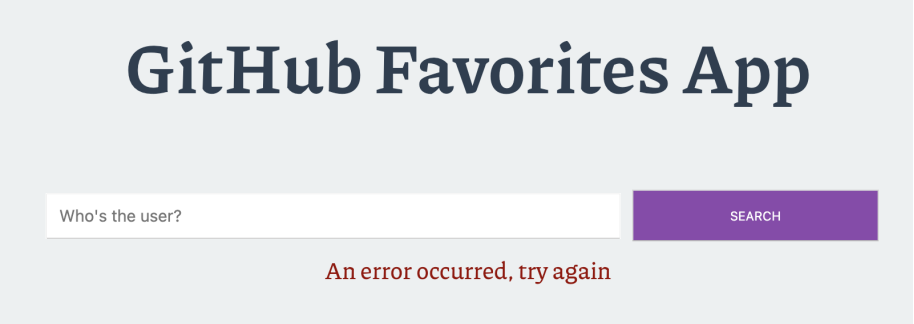
No caso de o usuário digitado não existir no *GitHub*, é mostrado uma mensagem clara do erro para o usuário, conforme a figura 103. Também há um tratamento de erro para algum erro genérico da requisição (figura 104).

Figura 103 - Tela da aplicação *Github Favorites* no cenário de usuário não existente.



Fonte: própria Autora.

Figura 104 - Tela da aplicação *Github Favorites* no cenário de erro genérico.



Fonte: própria Autora.

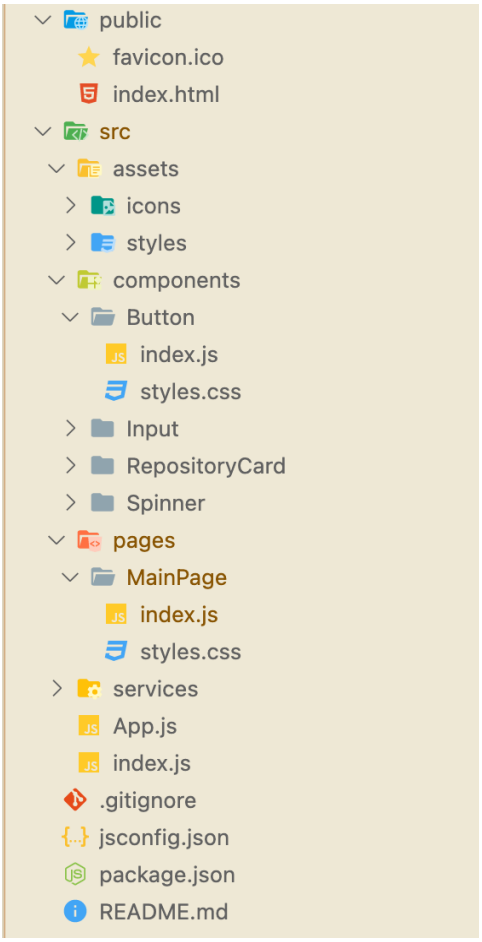
8.1.1. Estrutura de pastas do projeto *Github Favorites*

O repositório deste projeto é dividido em três pastas: *web*, *api* e *mobile*, a pasta onde será aplicado o *ESLint* é a *web* por conter o código da aplicação escrito com *React JS*. A pasta *mobile* é a aplicação escrita com *React Native* capaz de rodar em aparelhos móveis que utilizam *iOS* e *Android*. E a pasta api é responsável pelo servidor *backend* que faz uma requisição via *API* para o *GitHub*.

Quanto à estrutura da pasta *web* - figura 105 - o projeto contém em sua raiz:

- *"public":* contendo o corpo *HTML* que envolve toda a aplicação.
- *"src"*: pasta contendo todas as informações de funcionamento da aplicação, organizada da seguinte forma: *assets* (para imagens, ícones e estilo global), *components* (para adicionar todos os componentes do projeto, cada componente tem seu arquivo de *JSX* e de estilo), *pages* (para inserir páginas do projeto), *services* (onde se faz a chamada para *API*), *App.js* (arquivo que orquestra páginas) e *index.js* (o arquivo responsável por renderizar a aplicação toda).
- *".gitignore"*: para apontar arquivos que não devem ser versionados pelo *GitHub*.
- *"jsconfig.json"*: arquivo de configuração de *JS* para o *VSCode*.
- *"package.json"*: arquivo onde define-se dependências do projeto, aqui também define-se scripts que podem ser rodados via terminal.
- *"README.md"*: arquivo de documentação utilizado pelo *GitHub*.

Figura 106 - Estrutura de pastas do projeto *GitHub Favorites.*



Fonte: própria Autora.

8.1.1.1 Estrutura do arquivo *"package.json"*

O *"package.json"* é extremamente importante em projetos escritos com *JavaScript* - seja para *web, mobile* ou *backend* -, pois estipula e configura dependências do projeto, detectando quais outros pacotes ele vai precisar para ser executado, e *scripts* automatizados. A figura 106 ilustra este arquivo no projeto, utilizado neste projeto, onde:

- *"name":* define o nome do projeto ou aplicação.
- *"version":* versão do projeto.
- *"private":* acesso ao projeto.
- *"depencencies":* dependências intrinsecamente ligadas ao projeto em produção. No caso de uma dessas dependências não existirem no ambiente de produção, a aplicação não funciona corretamente. No projeto em questão tem-se: *"axios"* e *"axios-mock-adapter"* para chamadas de *API*; *"react", "react-dom"* e *"react-scripts"* para uso da *library React JS*.
- *"scripts":* lista de scripts Node que podem ser utilizados programaticamente, ou seja, ao rodar *"npm run start"* no terminal, a aplicação será iniciada localmente.
- *"browsersList":* define quais browsers e versões o projeto suporta.

Figura 106 - Estrutura de do arquivo *"package.json"*.

```
{
  "name": "web",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "axios": "^0.19.2",
    "axios-mock-adapter": "^1.18.1",
    "react": "^16.13.1",
    "react-dom": "^16.13.1",
    "react-scripts": "3.4.1"
  },
  ▷ Debug
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "eject": "react-scripts eject"
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

Fonte: própria Autora.

8.2. Implementação da ferramenta *ESLint*

"Sempre tentamos escrever o melhor código possível, mas às vezes acontecem erros [...] Felizmente, algumas ferramentas podem nos ajudar a verificar a exatidão do nosso código assim que o digitamos" (ROLDAN, 2019, s.p., tradução nossa).

A ferramenta *ESLint* é baseada "em regras que podem ser habilitadas ou desabilitadas ou configuradas de acordo com preferências específicas. Você pode até desenvolver suas próprias regras. Existem também pacotes de regras que permitem evitar a configuração individual" (KEREKI, 2018, s.p., tradução nossa).

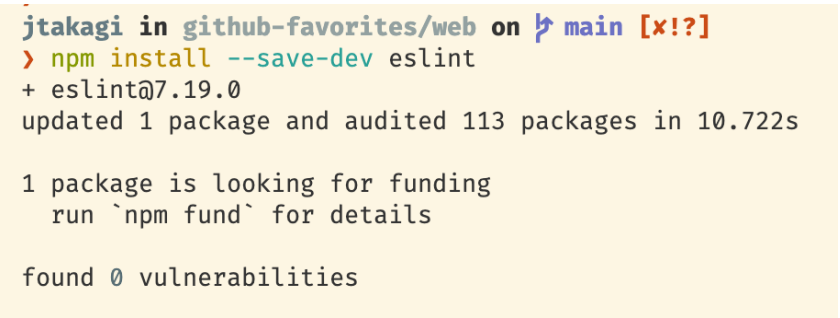
8.2.1. Instalação do *ESLint*

ESLint é construído com a linguagem *Node JS* e, ao instalar a linguagem na máquina, "você também instala o *npm*, o gerenciador de pacotes do *Node*. Na comunidade *JavaScript*, os engenheiros compartilham projetos *open source* para evitar a necessidade de reescrever estruturas, bibliotecas ou funções auxiliares por conta própria" (PORCELLO E BANKS, 2020, s.p., tradução nossa).

Para adicionar dependências ao projeto, ou pacotes, têm-se dois comandos: *"npm install nome_pacote"* e *"npm install --save-dev nome_pacote"*. A diferença entre os dois está relacionada com a estrutura *"dependencies"* apresentada anteriormente no arquivo *package.json*. No caso de que o projeto seja incapaz de rodar da maneira correta em produção sem a dependência, instala-se com a primeira opção. No caso de a dependência ser apenas para uso do desenvolvedor, localmente, como para testes, instalam-se pacotes com o segundo comando.

"O pacote chamado *eslint* [...] nos ajuda a identificar quaisquer padrões potencialmente problemáticos em nosso código *JavaScript*" (DERKS, 2019, s.p., tradução nossa). A ferramenta é uma dependência apenas para desenvolvimento do projeto, portanto segue a figura 107 com o comando que foi rodado no terminal da pasta *web*.

Figura 107 - Terminal após execução do comando de instalação da ferramenta *ESLint* como dependência de desenvolvimento do projeto.



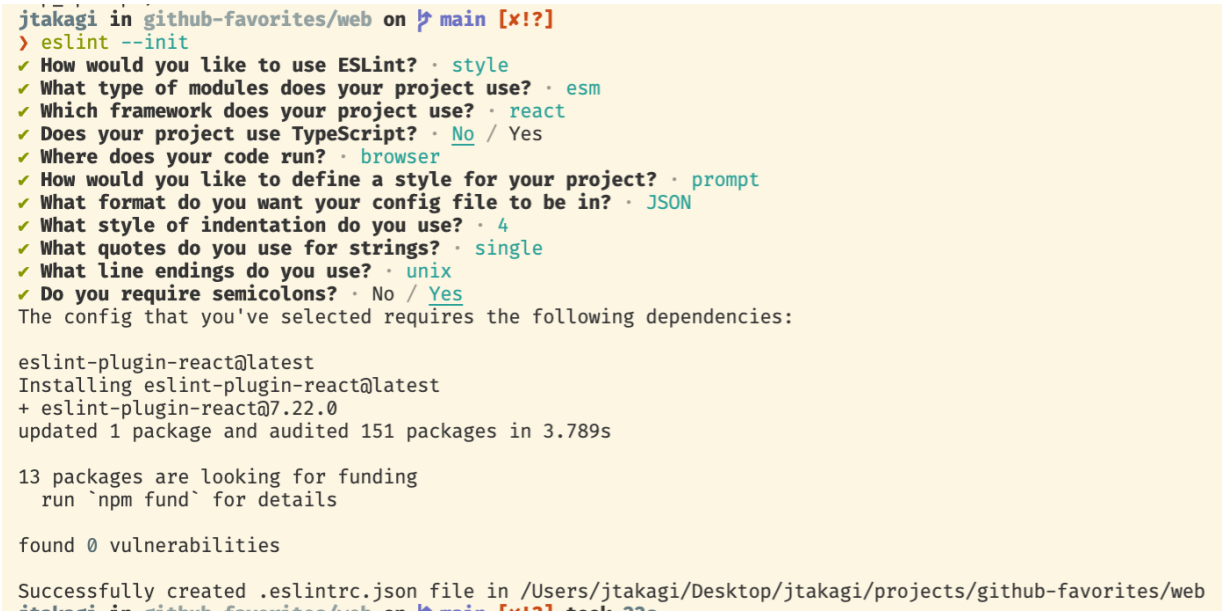
Fonte: própria Autora.

Para iniciar o arquivo de configuração do *ESLint* rodou-se o comando *"eslint --init"* no terminal. A partir daqui, a ferramenta fez uma série de perguntas a fim de montar o arquivo *.eslintrc* personalizado conforme o projeto do desenvolvedor. As perguntas e suas múltiplas escolhas estão no Anexo A (Perguntas de Configuração do *ESLint*).

Na figura 108 estão explicitadas as respostas dadas a fim de fazer uma pré-configuração do projeto. Segue-se a explicação para cada escolha:

1. Uso do *ESLint:* para checar a sintaxe, encontrar problemas e forçar estilo de código.
2. Módulo de importação: padrão para *E6+*, usando *import/export.*
3. *Library* utilizado: *React*.
4. Uso de *TypeScript*: resposta negativa.
5. Local onde o código roda: *browser* (navegador).
6. Definição de estilo para o projeto: adequando-se às respostas e adicionando regras à parte.
7. Formato do arquivo de configuração: *JSON*.
8. Estilo de indentação: tecla espaço.
9. Estilo de aspas em *string*: *single*.
10. Estilo de linha no final de arquivos: *unix*.
11. Estilo para uso de ponto e vírgula: obrigatório.

Figura 108 - Terminal após execução do comando para iniciar as configurações do *ESLint*.



Fonte: própria Autora.

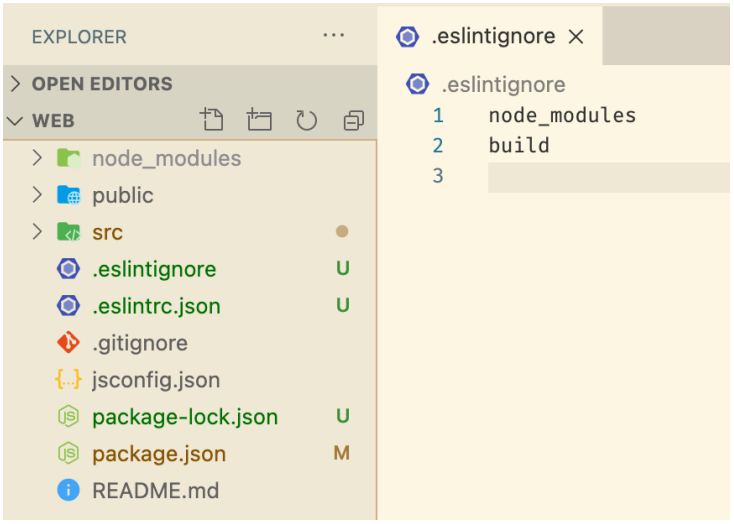
Após responder as questões iniciais de configuração, o pacote do *ESLint* adiciona algumas dependências que estão relacionadas com a forma como o desenvolvedor pretende utilizar a ferramenta. No caso deste projeto, indicou-se que está sendo utilizado o *React*. Portando o *ESLint* adicionou o pacote *"eslint-plugin-react"* (última versão disponível), "o *eslint-plugin-react* adiciona regras específicas para o *ESLint* para aplicações *React*" (DERKS, 2019, s.p., tradução nossa).

O *ESLint* consegue analisar códigos *JSX* sem qualquer plugin, porém para Roldan (2019, s.p., tradução nossa), o uso do plugin *"eslint-plugin-react"* é importante por permitir a aplicação de melhores práticas da *library*, mantendo um modelo consistente entre desenvolvedores e equipes.

Após a instalação desses pacotes, é importante criar um arquivo de nome *".eslintignore"* para adicionar pastas ou arquivos que serão ignorados pelo *ESLint*. A pasta *"node_modules"* e *"build"* foram adicionadas a este documento (figura 109). A *"node_modules"* é uma pasta onde estão todas as dependências do projeto, ela é criada e atualizada toda vez que roda-se o comando de instalação de pacotes no projeto. Não há necessidade de corrigir padrão de código nela. A pasta *"build"* é criada quando o desenvolvedor faz o *build* do projeto para ir para produção,

gerando um código compilado para esta etapa. Também não há necessidade de verificar padrões para um código que não será usado para desenvolvimento.

Figura 109 - Arquivo *".eslintignore"* do projeto.



Fonte: própria Autora.

8.2.2. Estrutura do arquivo de configuração *".eslintrc.json"*

O estado inicial do arquivo de configuração *".eslintrc.json"* está ilustrado na figura 110. Kekeri (2018, s.p.) define cada uma das configurações de forma concisa:

- *"env":* permite a especificação de ambientes, transformando-as em variáveis globais. No arquivo gerado, considera-se o ambiente de *browser* e o *JS* mais moderno.
- *"extends"*: permite selecionar um conjunto predefinido de regras, "a chave significa que estamos estendendo as regras recomendadas da configuração *ESLint*, mas sempre podemos substituir regras únicas manualmente dentro de nosso arquivo *.eslintrc* usando a *chave* rules" (ROLDAN, 2019, s.p., tradução nossa). O arquivo está com os padrões *"eslint:recommended"* e *"plugin:react/recommended".* O primeiro padrão sempre vem configurado, o segundo veio por conta da resposta de uso do *ESLint* em projeto *React*.
- *"parserOptions"*: permite especificar qual versão *JS* será processada e se irá utilizar módulos No projeto está adicionado também a *feature*

para verificações de *JSX*.

- *"plugins"*: "cada plugin é um módulo *npm* com um nome no formato *eslint-plugin- <plugin-name>*" (*ESLINT*, 2021, tradução nossa).
- *"rules"*: permite alterar regras para se adequar melhor ao estilo do projeto. O arquivo foi configurado inicialmente com quatro regras: indentação, quebra de linha, aspas e ponto e vírgula. Para cada regra "o primeiro parâmetro determina a gravidade do problema. O segundo parâmetro diz ao *ESLint* o que fazer. O *ESLint* e seus *plugins* são muito bem documentados e, para qualquer regra, você pode encontrar a descrição da regra" (ROLDAN, 2019, s.p., tradução nossa).

Figura 110 - Estado inicial do arquivo *".eslintrc.json"* do projeto.

```
{
    "env": {
        "browser": true,
        "es2021": true
    },
    "extends": [
        "eslint:recommended",
        "plugin:react/recommended"
    ],
    "parserOptions": {
        "ecmaFeatures": {
            "jsx": true
        },
        "ecmaVersion": 12,
        "sourceType": "module"
    },
    "plugins": [
        "react"
    ],
    "rules": {
        "indent": ["error", 4],
        "linebreak-style": ["error", "unix"],
        "quotes": ["error", "single"],
        "semi": ["error", "always"]
    }
}
```

Fonte: própria Autora.

8.2.3. Regras selecionadas e diretrizes do *ESLint*

As regras aplicadas na ferramenta *ESLint* foram selecionadas conforme os dados levantados através da aplicação do questionário sobre padrões

de código e boas práticas seguidas por desenvolvedores que trabalham em empresas utilizando a *library React JS*.

O questionário apresentou 59 regras baseadas em pesquisa bibliográfica e em estilos seguidos por *styles guides* utilizados como *plugin* no *ESLint*, os respondentes selecionaram quais regras são seguidas pelo seu time.

Foram obtidas 25 respostas, sendo apenas 20 consideradas válidas - excluíram-se respostas de profissionais que não atuassem com *React JS* -, para escolha das regras considerou-se que, pelo menos, metade dos desenvolvedores trabalham com elas, tornando-as relevantes para este estudo.

Seguem, abaixo, a tabela 1 apresentando os 51 padrões de código e boas práticas escolhidos, juntamente com sua regra no *ESLint*. Estas regras foram retiradas do *site* oficial da ferramenta (*ESLINT*, 2021, tradução nossa), e foram implementadas na ferramenta no próximo tópico.

Tabela 1 - Padrões de código e boas práticas para *React J*S e suas diretrizes no *ESLint*.

(continua)

Padrões de código e boas práticas	**Regras no *ESLint***
[*ARRAY*] Sempre retornar algo quando utilizar *high order functions*	*array-callback-return*
[*ARRAY*] Utilizar *destructuring* para acessar várias posições de um *array*	*prefer-destructuring*
[*BLOCKS*] Utilizar chaves em comandos multi-linhas	*curly*
[*BLOCKS*] Manter o *else* na mesma linha dos chaves	*brace-style*
[*CLASSES*] Utilizar a *keyword* "*this*" quando se estiver utilizando propriedades da classe	*class-methods-use-this*
[*CLASSES*] Evitar definir dois métodos com o mesmo nome dentro de uma classe	*no-dupe-class-members*
[*FUNCTIONS*] Utilizar a notação de *arrow function*	*prefer-arrow-callback*

Tabela 1 - Padrões de código e boas práticas para *React J*S e suas diretrizes no *ESLint*.

(continua)

Padrões de código e boas práticas	**Regras no *ESLint***
[*FUNCTIONS*] Utilizar parênteses entre argumentos (em *arrow function*) mesmo que contenha somente um	*arrow-parens*
[*FUNCTIONS*] Omitir a *keyword* "*return*" em *arrow functions* que ocupem apenas uma linha	*arrow-body-style*
[*FUNCTIONS*] Não modificar os argumentos recebidos dentro da função	*no-param-reassign*
[*MODULES*] Unificar chamadas de módulos	*no-duplicate-imports*
[*MODULES*] Manter todas as importações no topo do arquivo	*import/first*
[*MODULES*] Não incluir a extensão *.js* no final dos arquivos importados	*import/extensions*
[*STRING*] Utilizar *single quote* (aspas simples) para *string*	*quotes*
[*STRING*] Utilizar *template literal string* ao concatenar *strings*	*prefer-template, template-curly-spacing*
[*OBJECTS*] Utilizar *destructuring* para acessar várias propriedades de um objeto	*prefer-destructuring*
[*OBJECTS*] Utilizar sintaxe literal para criação de objetos	*no-new-object*
[*OBJECTS*] Utilizar *shorthand* em objetos	*object-shorthand*
[*OBJECTS*] Acessar propriedades com *dot notation*	*dot-notation*
[*REACT JS*] Sempre fechar *tags* nela mesmo, quando não houver filhas	*react/self-closing-comp*
[*REACT JS*] Sempre adicionar a propriedade *alt* em *tags* de <*img*>	*jsx-a11y/alt-text*

Tabela 1 - Padrões de código e boas práticas para *React J*S e suas diretrizes no *ESLint*.

(continua)

Padrões de código e boas práticas	**Regras no *ESLint***
[*REACT JS]* Alinhar estilos para a sintaxe de JSX	*react/jsx-closing-bracket-location react/jsx-closing-tag-location*
[*REACT JS*] Adicionar um espaço em branco antes do fechamento de *tags*	*no-multi-spaces, react/jsx-tag-spacing*
[*REACT JS*] Nomear componentes como *Pascal Case*	*react/jsx-pascal-case*
[*REACT JS*] Evitar utilizar *index* de *array*	*react/no-array-index-key*
[*REACT JS*] Utilizar *double quotes* (aspas duplas) para atributos *JSX*	*jsx-quotes*
[*REACT JS*] Não adicionar espaço dentro de chaves no *JSX*	*react/jsx-curly-spacing*
[*REACT JS*] Utilizar parênteses quando o retorno ocupar mais de uma linha	*react/jsx-wrap-multilines*
[*REACT JS*] Omitir valor da *"prop"* no caso de ser implicitamente *"true"*	*react/jsx-boolean-value*
[*SYNTAX*] Comparar valores com *triple equation*	*eqeqeq*
[*SYNTAX*] Evitar nomes de variáveis e funções com apenas uma letra	*id-length*
[*SYNTAX*] Nomear variáveis, funções e instâncias com *Camel Case*	*camelcase*
[*SYNTAX*] Utilizar *if* ternário apenas em um nível	*no-nested-ternary*
[*SYNTAX*] Ter um máximo de caracteres por linha	*max-len*
[*SYNTAX*] Nomear *constructors* e classes como *Pascal Case*	*new-cap*
[*SYNTAX*] Separar operadores de comparação de lógicos	*no-mixed-operators*

Tabela 1 - Padrões de código e boas práticas para *React J*S e suas diretrizes no *ESLint*.

(continua)

Padrões de código e boas práticas	**Regras no *ESLint***
[*SYNTAX*] Utilizar *High Order Functions* para iterações ao invés de *"loops"* simples	*no-iterator no-restricted-syntax*
[*SYNTAX*] Utilizar ponto e vírgula no final de comandos	*semi*
[*VARIABLE*] Utilizar *"const/let*" para declaração de variáveis ao invés de *"var"*	*prefer-const, no-const-assign, no-var, no-undef*
[*VARIABLE*] Excluir variáveis não utilizadas no código	*no-unused-vars, react/jsx-uses-react, react/jsx-uses-vars*
[*VARIABLE*] Não utilizar cadeia de variáveis	*no-multi-assign*
[*VARIABLE*] Utilizar apenas um *"const/let"* por declaração de variáveL	*one-var*
[*WHITE SPACES*] Adicionar espaço dentro de chaves	*object-curly-spacing*
[*WHITE SPACES*] Adicionar espaço entre operadores	*space-infix-ops*
[*WHITE SPACES*] Pular uma nova linha no final de cada arquivo	*eol-last*
[*WHITE SPACES*] Evitar espaços a mais no final de linhas de código (*trailing spaces*)	*no-trailing-spaces*
[*WHITE SPACES*] Adicionar espaço após chamada de funções, estruturas de repetição e condicionais	*keyword-spacing*
[*WHITE SPACES*] Não adicionar espaço em colchetes	*array-bracket-spacing*
[*WHITE SPACES*] Indentar o código utilizando dois espaços	*indent*
[*WHITE SPACES*] Não adicionar espaço em parênteses	*space-in-parens*

Tabela 1 - Padrões de código e boas práticas para *React JS* e suas diretrizes no *ESLint*.

(conclusão)

Padrões de código e boas práticas	**Regras no *ESLint***
[*WHITE SPACES*] No começo de um comentário, manter um espaço	*spaced-comment*

Fonte: própria Autora.

8.2.4. Configuração do *ESLint*

"Decidir quais regras habilitar ou desabilitar é algo que geralmente acontece no início de um projeto e pode-se esperar que algumas novas mudanças nas regras ocorram com o tempo" (KEREKI, 2018, s.p., tradução nossa).

"As configurações possíveis para uma regra são *"off"* ou 0 (se quiser desabilitá-la), *"warn"* ou 1 (se quiser receber um aviso, mas aceitar) e *"error"* ou 2 (rejeitar o arquivo). Algumas regras aceitam configurações extras (KEREKI, 2018, s.p., tradução nossa).

Para Kekeri (2018, s.p.) o ideal é configurar regras apenas com *"off"* e *"error"* pois os desenvolvedores podem se acostumar com os avisos e em um dado momento eles acabam não prestando atenção neles.

Na tabela 2, organizou-se cada regra e seu modo de configuração, que será utilizada, seguindo o *site* oficial da ferramenta (*ESLINT*, 2021, tradução nossa), para ajustes em regras que aceitam configurações extras. Na coluna de configurações foi considerada a lógica: [<severidade>, "opções *string*", {"opções objeto"}]. Cada regra do *ESLint* e de seus *plugins* aceitam parâmetros específicos. Estes estão na coluna parâmetro. Conforme o que foi selecionado no questionário, montou-se a configuração para a regra.

Tabela 2 - Parâmetros das regras no *ESLint* e configurações a serem implementadas.

(continua)

Regras	**Opção *String***	**Opção *Object***	**Configuração**
array-callback-return	-	*"allowImplicit":* (padrão *false*) permite retorno *"undefined"* para métodos que requerem um retorno. *"checkForEach":* (padrão *false*) ativa retorno explícito para o método *"forEach".*	*["error", { "allowImplicit": true }]*
prefer-destructuring	-	*"VariableDeclarator":* (padrão *true*) força a desestruturação em declarações de variáveis. *"AssignmentExpression"*: (padrão *true*) força a desestruturação em atribuições de variáveis. *"enforceForRenamedProperties"*: força a desestruturação de objetos que estão sendo acessados e alocados em outra variável.	*["error", { "array": true, "object": true }, { "enforceForRenamedProperties": false }]*
curly	*"all": (padrão) f*orça sempre o uso de chaves. *"multi":* força o uso de chaves quando blocos tem mais de uma linha. *"multi-line":* atua da mesma forma que o "multi" porém adicionar o "if-else".	-	*["error", "multi-line"]*

Tabela 2 - Parâmetros das regras no *ESLint* e configurações a serem implementadas.

(continua)

Regras	**Opção *String***	**Opção *Object***	**Configuração**
brace-style	*"1tbs":* (padrão) força verdadeiro estilo de chaves. *"stroustrup*": força o estilo *Stroustrup*. *"allman"*: força o estilo *Allman*.	*"allowSingleLine"*: (padrão *false*) permite que as chaves de abertura e fechamento de um bloco fiquem na mesma linha.	*["error", "1tbs", { "allowSingleLine": true }]*
class-methods-use-this	-	*"exceptMethods":* permite definir um array de nomes de métodos que serão ignorados pela regra.	*["error", { "exceptMethods": [] }]*
no-dupe-class-members	-	-	*["error"]*
prefer-arrow-callback	-	*"allowNamedFunctions": (padrão false) permite nomear funções callback.* *"allowUnboundThis": (padrão true)* permite que funções que contenham *"this"* possam ser utilizadas como *callback.*	*["error", { "allowNamedFunctions": false, "allowUnboundThis": true }]*
arrow-parens	*"always": (padrão)* força sempre o uso de parênteses em argumentos. *"as-needed":* permite o não uso de parênteses em cenários em que pode ser omitido.	*"requireForBlockBody": (padrão true)* força uso de parênteses em retorno de funções.	*["error", "always", { "requireForBlockBody": true }]*

Tabela 2 - Parâmetros das regras no *ESLint* e configurações a serem implementadas.

(continua)

Regras	Opção *String*	Opção *Object*	Configuração
arrow-body-style	*"always":* força sempre o uso de parênteses em argumentos. *"as-needed":* permite o não uso de parênteses em cenários em que pode ser omitido. *"never":* não força uso de parênteses em argumentos em nenhum cenários.	*"requireReturnForObjectLiteral":* (padrão *false*) aplicável somente com o parâmetro *"as-needed"*, força retorno literal de objetos quando for o caso.	*["error", "as-needed", { "requireReturnForObjectLiteral": false }]*
no-param-reassign	-	*"props":* (padrão *false*) não permite modificar propriedades de objeto recebido pela função. *"ignorePropertyModificationsFor":* permite modificações em elementos que estejam na lista. *"ignorePropertyModificationsForRegex":* permite modificações em elementos que estejam na lista, usando *regex*.	*["error", {"props": true, "ignorePropertyModificationsFor": []}]*
no-duplicate-imports	-	*"includeExports":* (padrão *false*) não permite reescrita de módulo para *export*.	*["error", { "includeExports": false }]*
import/first	*"absolute-first":* não permite que importações absolutas venham após as relativas.	-	*["error", "absolute-first"]*

Tabela 2 - Parâmetros das regras no *ESLint* e configurações a serem implementadas.

(continua)

Regras	**Opção *String***	**Opção *Object***	**Configuração**
import/extensions	*"never":* (padrão) não permite o uso de extensões em importações. *"always":* força o uso de extensões em importações.	*"ignorePackages":* lista para ignorar importação de pacotes.	*["error"]*
quotes	*"double":* (padrão) força o uso de aspas duplas. *"single":* força o uso de aspas singles. *"backtick":* força o uso de crases.	*"avoidEscape":* (padrão *true*) força o uso de aspas duplas ou simples para *escape*. *"allowTemplateLiterals":* (padrão *true*) permite o uso de crases em *strings*.	*["error", "single", { "avoidEscape": true, "allowTemplateLiterals": false }]*
prefer-template	-	-	*["error"]*
template-curly-spacing	*"never":* (padrão) não permite uso de espaço entre parênteses em *template string.* *"always":* força o uso de espaço entre parênteses.	-	*["error", "never"]*
no-new-object	-	-	*["error"]*

Tabela 2 - Parâmetros das regras no *ESLint* e configurações a serem implementadas.

(continua)

Regras	**Opção *String***	**Opção *Object***	**Configuração**
object-shorthand	*"always":* (padrão) força sempre o uso de *shorthand.* *"methods":* força o uso de *shorthand* em métodos. *"properties":* força o uso de *shorthand* em propriedades. *"never":* não permite o uso de *shorthand.* *"consistent":* verifica se apenas uma forma de acessar objetos está sendo usada. *"consistent-as-needed":* verifica se apenas uma forma de acessar objetos está sendo usada, porém há preferência por *shorthand*.	*"avoidQuotes":* (padrão *false*) não permite use se *shorthand* quando a chave do objeto é uma *string* literal. *"ignoreConstructors":* (padrão *false*) ignora a regra para funções construtoras. *"avoidExplicitReturnArrows":* (padrão *false*) indica que métodos devem utilizar retornos explícitos em *arrow function.*	*["error", "always", { "avoidQuotes": true, "ignoreConstructors": false, "avoidExplicitReturnArrows": false }]*
dot-notation	-	*"allowKeywords":* (padrão *true*) evita *dot notation* em palavras reservadas. *"allowPattern":* permite *bracket notation* em palavras que deem *match* com a *regex*.	*["error", { "allowKeywords": true }]*
react/self-closing-comp	-	*"component":* (padrão *true*) força a regra para componentes. *"html":* (padrão *true*) força a regra para tags *HTML*.	*["error", { "component": true, "html": true }]*

Tabela 2 - Parâmetros das regras no *ESLint* e configurações a serem implementadas.

(continua)

Regras	**Opção *String***	**Opção *Object***	**Configuração**
jsx-a11y/alt-text	-	*"elements":* elementos que serão ignorados pela regra.	*[2]*
react/jsx-closing-bracket-location	*"tag-aligned":* (padrão) alinhamento com a *tag* de abertura. *"line":* alinhamento com a linha da *tag* de abertura. *"after-props":* alinhamento logo após a última *"prop".* *"props-aligned":* alinhamento após a última *"prop".*	*"nonEmpty":* (padrão *"tag-aligned"*) define o padrão para espaços não vazios. *"selfClosing":* (padrão *"tag-aligned"*) define o padrão para fechamentos de *tag* nela mesma.	*[2, { "nonEmpty": "tag-aligned", "selfClosing": "tag-aligned" }]*
react/jsx-closing-tag-location	-	-	*[2]*
no-multi-spaces	-	*"ignoreEOLComments":* (padrão *false*) ignora múltiplos espaços dentro de comentários. *"exceptions":* ignora propriedades passadas.	*["error", { "ignoreEOLComments": false }]*

Tabela 2 - Parâmetros das regras no *ESLint* e configurações a serem implementadas.

(continua)

Regras	**Opção *String***	**Opção *Object***	**Configuração**
react/jsx-tag-spacing	-	*"closingSlash":* (padrão *never*) espaços dentro da *tag*. *"beforeSelfClosing":* (padrão *always*) espaço no final da *tag*. *"afterOpening":* (padrão *never*) espaços dentro da *tag* após abertura. *"beforeClosing":* (padrão *allow*) espaços dentro da *tag* após abertura.	*[2, { "closingSlash": "never", "beforeSelfClosing": "always", "afterOpening": "never", "beforeClosing": "allow"}]*
react/jsx-pascal-case	-	*"allowAllCaps":* (padrão *false*) permite nomes de componente com todas as letras em caixa alta. *"ignore":* lista de elementos que serão ignorados pela regra.	*[2]*
react/no-array-index-key	-	-	*[2]*
jsx-quotes	*"prefer-double":* (padrão) força o uso de aspas duplas em *JSX*. *"prefer-single":* força o uso de aspas simples em *JSX*.	-	*["error", "prefer-double"]*
react/jsx-curly-spacing	-	*"when":* (padrão *never*) permite espaço dentro de parênteses. *"children":* (padrão *false*) permite espaço dentro de parênteses de *tags* filhas.	*[2, { "when": "never" }]*

Tabela 2 - Parâmetros das regras no *ESLint* e configurações a serem implementadas.

(continua)

Regras	Opção *String*	Opção *Object*	Configuração
react/jsx-wrap-multilines	-	*"declaration":* (padrão *parens*) uso em declarações. *"assignment":* (padrão *parens*) uso em atribuições. *"return":* (padrão *parens*) uso em retornos. *"arrow":* (padrão *parens*) uso em *arrow function*. *"condition":* (padrão *ignore*) uso em condições. *"logical":* (padrão *ignore*) uso em condições lógicas. *"prop":* (padrão *ignore*) uso em *props*.	*[2, {* *"declaration": "parens",* *"assignment": "parens",* *"return": "parens",* *"arrow": "parens",* *"condition": "ignore",* *"logical": "ignore",* *"prop": "ignore"* *}]*
react/jsx-boolean-value	-	*"never":* lista de "*props"* a que nunca devem receber valores. *"always":* lista de "*props"* a que sempre devem receber valores.	*[2]*
eqeqeq	*"always":* (padrão) força o uso de *triple equals.* *"smart":* força o uso de *triple equals,* exceto em caso de comparação de dois valores literais, quando utilizar *typeof* e comparação com o valor *null*.	-	*["error", "always"]*

Tabela 2 - Parâmetros das regras no *ESLint* e configurações a serem implementadas.

(continua)

Regras	**Opção *String***	**Opção *Object***	**Configuração**
id-length	-	*"min":* (padrão *2*) quantidade mínima de carácter. *"max":* (padrão *infinito*) quantidade máxima de carácter. *"properties":* estender regra para propriedades de objetos. *"exceptions":* lista de elementos a serem ignorados pela regra. *"exceptionPatterns":* lista de elementos a serem ignorados pela regra através de *regex*.	*["error"]*
camelcase	-	*"properties":* (padrão *always*) força regra em nomes de propriedades. *"ignoreDestructuring":* (padrão *false*) força regra em desestruturação. *"ignoreImports":* (padrão *true*) força regra em importações. *"ignoreGlobals":* (padrão *true*) força regra em variáveis globais. *"allow":* lista de propriedades ignoradas pela regra.	*["error", { "ignoreImports": true, "ignoreGlobals": true, "properties": "never", "ignoreDestructuring": false }]*

Tabela 2 - Parâmetros das regras no *ESLint* e configurações a serem implementadas.

(continua)

Regras	**Opção *String***	**Opção *Object***	**Configuração**
no-nested-ternary	-	-	*["error"]*
max-len	*"code":* (padrão 80) força um número máximo de caracteres por linha. *"tabWidth":* (padrão 4) força o tamanho do caracter *tab*.	*"comments":* (padrão 80) força um número máximo de caracteres por linha de comentário. *"ignorePattern":* ignora a regra para elementos que dão *match* com a *regex*. *"ignoreComments":* (padrão *true*) ignora comentários. *"ignoreTrailingComments":* (padrão *true*) ignora comentários no final da linha. *"ignoreUrls":* (padrão *true*) ignora comentários com *links*. *"ignoreStrings":* (padrão *true*) ignora linhas com *string*. *"ignoreTemplateLiterals":* (padrão *true*) ignora linhas com *template string*. *"ignoreRegExpLiterals":* (padrão *true*) ignora linhas com *regex*.	*["error", 100, 2, { "ignoreUrls": true, "ignoreComments": false, "ignoreRegExpLiterals": true, "ignoreStrings": true, "ignoreTemplateLiterals": true}]*

Tabela 2 - Parâmetros das regras no *ESLint* e configurações a serem implementadas.

(continua)

Regras	**Opção *String***	**Opção *Object***	**Configuração**
new-cap	-	*"newIsCap":* (padrão *true*) força letra maiúscula em novas instâncias. *"capIsNew":* (padrão *true*) força *keyword* new em instâncias. *"newIsCapExceptions":* lista a ser ignorada. *"newIsCapExceptionPattern":* lista com *regex* a ser ignorada. *"capIsNewExceptions":* lista a ser ignorada. *"capIsNewExceptionPattern"*: lista com *regex* a ser ignorada. *"properties":* (padrão *true*) força a regra em propriedades de objetos.	*["error", { "newIsCap": true, "capIsNew": true, "properties": true}]*
no-mixed-operators	-	*"groups"*: lista de operadores a serem verificados pela regra. *"allowSamePrecedence":* (padrão *true*) permite que a regra seja ignorada se os operadores forem do mesmo grupo.	*["error", { "allowSamePrecedence": false }]*
no-iterator	-	-	*["error"]*

Tabela 2 - Parâmetros das regras no *ESLint* e configurações a serem implementadas.

(continua)

Regras	**Opção *String***	**Opção *Object***	**Configuração**
no-restricted-syntax	-	*"selector: message":* adiciona seletor e mensagem de erro.	*["error", { "selector": "ForInStatement", "message": 'for..in loops iterate over the entire prototype chain, which is virtually never what you want. Use Object.{keys,values,entries}, and iterate over the resulting array.", }, { "selector": "ForOfStatement", "message": "iterators/generators require regenerator-runtime, which is too heavyweight for this guide to allow them. Separately, loops should be avoided in favor of array iterations.", }]*
semi	*"always":* (padrão) força uso de ponto e vírgula. *"never":* não força uso de ponto e vírgula.	-	*["error", "always"]*
prefer-const	-	*"destructuring":* (padrão *any*) força uso de *"const"* em desestruturação. *"ignoreReadBeforeAssign" :* (padrão *false*) ignora variáveis antes de lidas.	*["error", { "destructuring": "any", "ignoreReadBeforeAssign": true }]*

Tabela 2 - Parâmetros das regras no *ESLint* e configurações a serem implementadas.

(continua)

Regras	**Opção *String***	**Opção *Object***	**Configuração**
no-const-assign	-	-	*["error"]*
no-var	-	-	*["error"]*
no-undef	-	*"typeof":* (padrão *false*) não permite variáveis para checagem de tipo.	*["error"]*
no-unused-vars	-	*"vars":* (padrão *all*) não permite a checagem de todas as variáveis ou apenas as locais. *"args":* (padrão *after-used*) regra aplicada para casos de argumentos de funções. *"ignoreRestSiblings":* (padrão *false*) ignora parâmetro *"rest"*.	*["error", { "vars": "all", "args": "after-used", "ignoreRestSiblings": true }]*
react/jsx-uses-react	-	-	*[2]*
react/jsx-uses-vars	-	-	*[2]*
no-multi-assign	-	-	*["error"]*
one-var	*"always":* (padrão) força apenas uma declaração por escopo. *"never":* permite mais de uma declaração por escopo. *"consecutive":* segue a opção "never" mas requer que declarações de variáveis consecutivas sejam combinadas em uma única declaração	*"var":* (padrão *always*) força apenas uma declaração *"var"* por escopo. *"let":* (padrão *always*) força apenas uma declaração *"let"* por escopo. *"const":* (padrão *always*) força apenas uma declaração *"const"* por escopo.	*["error", "never"]*

Tabela 2 - Parâmetros das regras no *ESLint* e configurações a serem implementadas.

(continua)

Regras	**Opção *String***	**Opção *Object***	**Configuração**
object-curly-spacing	*"never":* (padrão) não permite espaço entre chaves. *"always":* permite espaço entre chaves.	*"arraysInObjects":* (padrão *true*) força espaço dentro de chaves de objetos dentro de *arrays*. *"objectsInObjects":* (padrão *true*) força espaço dentro de chaves de objetos dentro de objetos.	*["error", "always"],*
space-infix-ops	-	*"int32Hint":* (padrão *false*) desabilita a regra para o cenário de variável valer um valor ou outro: A|B.	*["error"]*
eol-last	*"always":* (padrão) força que o arquivo termine com uma linha em branco. As opções *"unix"* e *"windows"* foram descontinuadas e equivalem a esta. *"never":* não força que o arquivo termine com uma linha em branco.	-	*["error", "always"],*
no-trailing-spaces	-	*"skipBlankLines":* (padrão *false*) desabilita espaços em branco desnecessários em linhas vazias. *"ignoreComments":* (padrão *false*) desabilita regra para comentários.	*["error", { "skipBlankLines": false, "ignoreComments" : false }]*

Tabela 2 - Parâmetros das regras no *ESLint* e configurações a serem implementadas.

(continua)

Regras	**Opção *String***	**Opção *Object***	**Configuração**
keyword-spacing	-	*"before":* (padrão *true*) força espaço antes de *keywords*. *"after":* (padrão *true*) força espaço após *keywords*.	*["error", { "before": true, "after": true }]*
array-bracket-spacing	*"never":* (padrão) não permite espaços em branco dentro de colchetes. *"always":* permite espaços em branco dentro de colchetes.	*"singleValue":* (padrão *false*) permite espaço quando há apenas um valor no *array*. *"objectsInArrays":* (padrão *false*) permite espaço entre colchetes e chaves quando existem objetos dentro de um *array*. *"arraysInArrays":* (padrão *false*) permite espaço entre colchetes dentro de colchetes.	*["error", "never"]*
indent	-	-	*["error", 2]*
space-in-parens	*"never":* (padrão) não permite espaços em branco dentro de parênteses. *"always":* permite espaços em branco dentro de parênteses.	-	*["error", "never"]*
spaced-comment	*"always":* (padrão) força espaços em branco dentro de comentário. *"never":* não força espaços em branco dentro de comentário.	-	*["error", "always"]*

Fonte: própria Autora.

Ao consultar as regras para os padrões de código e boas práticas escolhidas, verificou-se que algumas só podem ser adicionadas através de *plugins*: são os casos de regras com nomenclatura *<plugin>/<regra>*. Portanto para seguir com a implementação, foram instalado os *plugins*: *"import"* e *"jsx-a11y"* (figura 111), o *plugin "react"* já havia sido instalado no primeiro passo de integração do *ESLint* ao projeto.

Figura 111 - Terminal com a instalação dos *plugins: "import"* e *"jsx-a11y"*.

```
jtakagi in github-favorites/web on  main [x!?]
❯ npm install eslint-plugin-import eslint-plugin-jsx-a11y --save-dev

> core-js-pure@3.8.3 postinstall /Users/jtakagi/Desktop/jtakagi/projects/github-favorites/web/node_modules/core-js-pure
> node -e "try{require('./postinstall')}catch(e){}"

Thank you for using core-js ( https://github.com/zloirock/core-js ) for polyfilling JavaScript standard library!

The project needs your help! Please consider supporting of core-js on Open Collective or Patreon:
> https://opencollective.com/core-js
> https://www.patreon.com/zloirock

Also, the author of core-js ( https://github.com/zloirock ) is looking for a good job -)

+ eslint-plugin-jsx-a11y@6.4.1
+ eslint-plugin-import@2.22.1
added 55 packages from 51 contributors and audited 206 packages in 30.831s

39 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Fonte: própria Autora.

Para ter acesso às regras de plugins adicionados, foram inseridos na chave *"plugins"* do arquivo *".eslintrc.json"*. Também foram apagadas configurações pré-definidas de dentro das chaves *"rules"* e *"extends"* para evitar que regras extras fossem adicionadas. O arquivo após esses ajustes está ilustrado na figura 112.

Figura 112 - Arquivo *".eslintrc.json"* configurado com *plugins*.

```
{
    "env": {
        "browser": true,
        "es2021": true
    },
    "extends": [],
    "parserOptions": {
        "ecmaFeatures": {
            "jsx": true
        },
        "ecmaVersion": 12,
        "sourceType": "module"
    },
    "plugins": [
        "import",
        "jsx-a11y",
        "react"
    ],
    "rules": {}
}
```

Fonte: própria Autora.

Todas as configurações da Tabela 2 foram adicionadas na chave *"rules"* do arquivo *".eslintrc"* no formato: *<nome da rega>:<configuração>*. O arquivo final com a implementação de regras de padrões de código e boas práticas ficou com a estrutura na figura 113. Acima de cada regra implementada foi adicionado um comentário a fim de esclarecer seu uso e um *link* para o *site* oficial da ferramenta ou do *plugin*. Para uso de comentários alterou-se a extensão do arquivo para *".js",* pois arquivos *JSON* não suportam esse comportamento.

Figura 113 - Arquivo *".eslintrc.js"* configurado com *"rules"*.

Fonte: própria Autora.

O arquivo na integra está disponível em um *link* público no *GitHub*: https://gist.github.com/jennifertakagi/987436d95644a9855526f930cfa6c00e da própria autora e no Apêndice B deste documento.

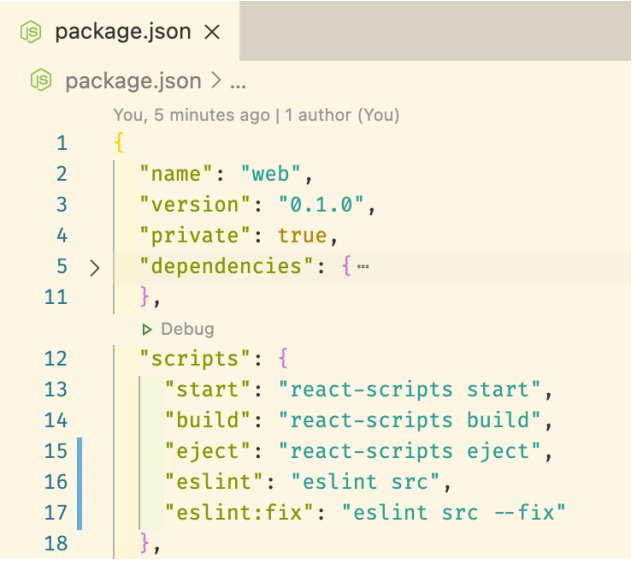
8.2.5. Executando comandos do *ESLint* no projeto

Após configurar corretamente o arquivo do *ESLint* o próximo passo é rodar a ferramenta para verificar erros no projeto. "Primeiro, adicionaremos um *script* para rodar o *ESLint* em nosso diretório de origem (*src*). Com o *npm run eslint* no *package.json*" (KEKERI, 2018, s.p., tradução nossa).

"Há uma *flag* para o comando *ESlint* chamado *--fix*. Com esta opção, *ESLint* tentará corrigir todos os nossos erros de *linter* automaticamente (nem todos eles)" (ROLDAN, 2019, s.p., tradução nossa). Para verificar quais regras conseguem ser corrigidas automaticamente pela opção basta verificar o *site* oficial da ferramenta.

No arquivo package.json foram adicionados mais dois *scripts* na chave de mesmo nome: *"eslint"* e *"eslint:fix"*, o primeiro a fim de verificar possíveis falhas no código e o segundo com a intenção de corrigir automaticamente as regras que possibilitam essa opção - figura 114.

Figura 114 - Arquivo *"package.json"* configurado com os *scripts* para executar o *ESLint*.



Fonte: própria Autora.

Ao rodar o comando *"npm run eslint"* no terminal do projeto, a ferramenta identificou vários erros conforme as definições na configuração da mesma. A estrutura da apresentação dos erros está ilustrada na figura 115. O *ESLint* separa erros por blocos de arquivos onde eles ocorrem. Também aponta-se a linha do erro, o nome e a qual regra ele pertence.

Figura 115 - Terminal apontando erros no código após executar o comando *"npm run eslint".*

```
jtakagi in github-favorites/web on  main [x!?]
❯ npm run eslint

> web@0.1.0 eslint /Users/jtakagi/Desktop/jtakagi/projects/github-favorites/web
> eslint src


/Users/jtakagi/Desktop/jtakagi/projects/github-favorites/web/src/App.js
  1:19  error  Strings must use singlequote  quotes
  2:22  error  Strings must use singlequote  quotes
  4:8   error  Strings must use singlequote  quotes

/Users/jtakagi/Desktop/jtakagi/projects/github-favorites/web/src/components/Button/index.js
  1:19  error  Strings must use singlequote  quotes
  3:8   error  Strings must use singlequote  quotes

/Users/jtakagi/Desktop/jtakagi/projects/github-favorites/web/src/components/Input/index.js
  1:19  error  Strings must use singlequote                              quotes
  3:8   error  Strings must use singlequote                              quotes
  7:1   error  This line has a length of 107. Maximum allowed is 100     max-len
  7:83  error  There should be no space after '{'                        react/jsx-curly-spacing
  7:90  error  There should be no space before '}'                       react/jsx-curly-spacing

/Users/jtakagi/Desktop/jtakagi/projects/github-favorites/web/src/components/RepositoryCard/index.js
   1:19  error  Strings must use singlequote                 quotes
   3:8   error  Strings must use singlequote                 quotes
   4:22  error  Strings must use singlequote                 quotes
  15:37  error  A space is required before closing bracket   react/jsx-tag-spacing
  17:15  error  Trailing spaces not allowed                  no-trailing-spaces
```

Fonte: própria Autora.

No final do comando a ferramenta aponta o total de erros e avisos encontrados e quantos destes podem ser ajustados com o uso da *flag "--fix"* no comando para executar a verificação - figura 116. No projeto utilizado neste estudo, 67 erros de um total de 70 podem ser ajustados.

Figura 116 - Terminal apontando quais erros podem ser ajustados ao executar o comando *"npm run eslint:fix".*

```
✖ 70 problems (70 errors, 0 warnings)
  67 errors and 0 warnings potentially fixable with the `--fix` option.

npm ERR! code ELIFECYCLE
npm ERR! errno 1
npm ERR! web@0.1.0 eslint: `eslint src`
npm ERR! Exit status 1
npm ERR!
```

Fonte: própria Autora.

Ao executar o comando *"npm run eslint:fix"* - configurado no arquivo *package.json* para rodar o *ESLint* com a *flag --fix* o resultado é o da figura 117, onde quatro erros devem ser corrigidos manualmente:

1. No arquivo *"index.js"* dentro das pastas *"components/Input":*

uma linha apresenta 105 caracteres, sendo que o máximo permitido é 100.

2. No arquivo *"index.js"* dentro das pastas *"components/Spinner"*: a *tag* <*img*> está sem o atributo *alt.*

3. No arquivo *"index.js"* dentro das pastas *"components/Main"*: há uma variável cujo nome só contém um caractere, a regra definida requer pelo menos dois caracteres.

4. No arquivo *"index.js"* dentro das pastas *"components/Main"*: há um erro de indexação.

Figura 117 - Terminal apontando erros que devem ser ajustados manualmente.

```
❯ npm run eslint:fix

> web@0.1.0 eslint:fix /Users/jtakagi/Desktop/jtakagi/projects/github-favorites/web
> eslint src --fix


/Users/jtakagi/Desktop/jtakagi/projects/github-favorites/web/src/components/Input/index.js
  7:1  error  This line has a length of 105. Maximum allowed is 100  max-len

/Users/jtakagi/Desktop/jtakagi/projects/github-favorites/web/src/components/Spinner/index.js
  9:7  error  img elements must have an alt prop, either with meaningful text, or an empty string for decorative images  jsx-a11y/alt-text

/Users/jtakagi/Desktop/jtakagi/projects/github-favorites/web/src/pages/MainPage/index.js
  46:114  error  Identifier name 'e' is too short (< 2)         id-length
  56:1    error  Expected indentation of 6 spaces but found 24  indent

✖ 4 problems (4 errors, 0 warnings)
  1 error and 0 warnings potentially fixable with the `--fix` option.
```

Fonte: própria Autora.

Depois de ajustar os quatro erros manualmente, quebrando a linha que estava com muitos caracteres, adicionando o atributo *"alt"* na *tag* <*img*>, alterando o nome da variável *"e"* para *"event"* e ajustando a indentação, ao executar novamente o comando *"npm run eslint"* a fim de verificar se o projeto está seguindo os padrões definidos, o resultado é o da figura 118, onde o *ESLint* não acusa mais nenhum erro.

Figura 118 - Terminal não acusando nenhum erro de padrão de código ao executar *"npm run eslint"*.

```
jtakagi in github-favorites/web on  main [!]
❯ npm run eslint

> web@0.1.0 eslint /Users/jtakagi/Desktop/jtakagi/projects/github-favorites/web
> eslint src
```

Fonte: própria Autora.

8.2.5.1. *Plugin ESLint* no *Visual Studio Code*

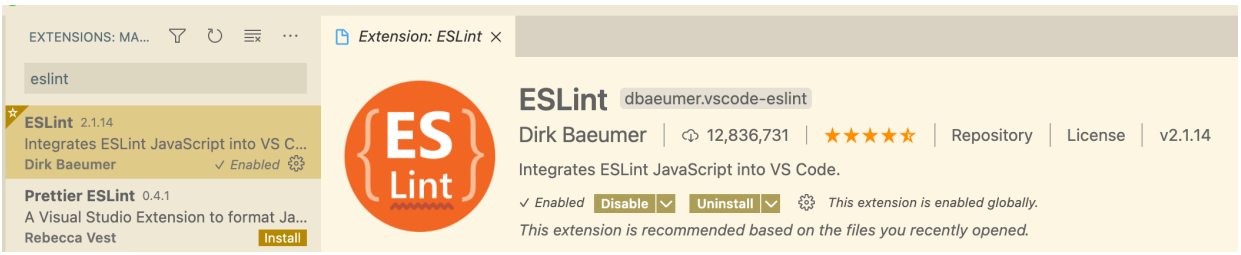
“Existem diferentes maneiras de fazer com que o *ESLint* nos ajude a escrever códigos melhores. Um é o que fizemos até agora - execute-o na linha de comando e obtenha a lista de erros. Isso funciona, mas não é muito conveniente executá-lo manualmente o tempo todo” (ROLDAN, 2019, s.p., tradução nossa).

“Todas essas verificações são muito boas, mas se você tivesse que parar de trabalhar, salvar tudo e executar uma ferramenta separada toda vez que quisesse verificar se há problemas em seu código, isso logo se tornaria insuportável” (KEKERI, 2018, s.p., tradução nossa).

Com o intuito de tornar o uso do ESLint mais eficiente, Roldan (2019, s.p.) e Kekeri (2018, s.p.) sugerem o uso do plugin *ESLint* diretamente no editor de código. Segundo os autores o *plugin* está disponível para *Atom, Visual Studio Code* e editores mais populares.

Ao habilitar o *plugin* é possível adicionar o processo de linting dentro de nosso editor para obter feedback imediato assim que digitarmos" (ROLDAN, 2019, s.p., tradução nossa). Para adicioná-lo ao VSCode, basta ir no menu de extensões do do editor, pesquisar pela palavra-chave *ESLint* e selecionar a opção de instalação - figura 119.

Figura 119 - *Plugin ESLint* disponível para o editor *VSCode*.



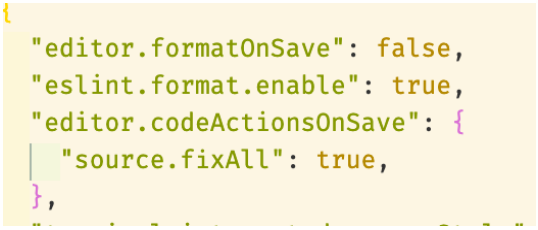
Fonte: própria Autora.

Ao instalar o *plugin* deve-se adicionar algumas regras no arquivo de configuração do próprio *VSCode*: *"setting.json"* (figura 120), com o propósito de ativar a correção de padrões de código no ato de salvar as alterações feitas no arquivo - essa *feature* se comporta da mesma forma que o comando *"eslint"* juntamente com a *flag "--fix"*.

As opções que devem ser configuradas são:

- *editor.formatOnSave:* desativar o comportamento do editor de formatar o arquivo ao salvar.
- e*slint.format.enable*: ativa o comportamento do *plugin* do *ESLint* para que ele seja capaz de formatar os documentos no ato de salvar arquivos.
- *editor.codeActionsOnSave:* dentro desta configuração adicionar *"source.fixAll"* para possibilitar que seja formatado todos os arquivos no ato de salvar.

Figura 120 - Arquivo *"settings.json"* com as configurações do *VSCode.*

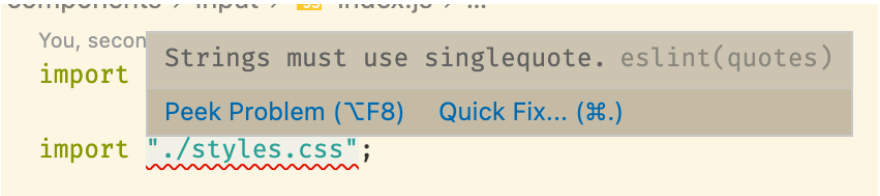


Fonte: própria Autora.

Ao ter o *plugin* do *ESLint* instalado, habilitado e configurado corretamente, basta reiniciar o *VSCode* para que a ferramenta tenha o comportamento de apontar erros de maneira orgânica. Na figura 121, ao adicionar aspas duplas em uma *string*, o *VSCode* sublima o elemento com uma linha vermelha e ao passar o mouse por cima aparece um modal com o erro - da mesma forma que acontece ao executar o comando no terminal.

Ao salvar o arquivo, o *ESLint* já ajusta para aspas simples, isso em casos de regras que aceitam o *--fix*.

Figura 121 - *Plugin ESLint* apontando erros em *living code*.



Fonte: própria Autora.

9. CONCLUSÕES

Este último capítulo é dedicado à exposição das conclusões acerca dos resultados obtidos quanto ao objetivo geral e objetivos específicos definidos no começo do trabalho. Também definiram-se os próximos passos para trabalhos futuros.

9.1. Em relação ao objetivo geral

No ecossistema *JavaScript*, onde as bibliotecas e técnicas mudam muito rapidamente, é crucial ter uma ferramenta que pode ser facilmente estendida com *plugins* e regras que podem ser ativadas e desativadas quando necessário. Também é importante definir práticas e padrões a serem seguidos pela equipe de desenvolvimento a fim de manter um código limpo.

O objetivo geral deste trabalho, qual seja o de apresentar a "especificação de parâmetros de configurações necessárias para a utilização da ferramenta *ESLint* em projetos que utilizam a *library React JS* visando auxiliar desenvolvedores a escreverem códigos limpos na linguagem *JavaScript*", e foi considerado totalmente atingido.

No referencial teórico discorreu-se a respeito dos conhecimentos necessários para compreender o básico sobre programação, conceitos da linguagem *JavaScript* e caracterização do *React JS* e *ESLint,* viabilizando concepções para compreender conceituação de código limpo.

Foram apresentadas, em seguida, melhores práticas na linguagem *JavaScript*, convenções utilizadas na biblioteca *React JS* e esclarecidas as vantagens da adoção do *clean code* e técnicas de mensuração sobre essas métricas.

Após estudos teóricos acerca do tema, houve a validação destas teorias por meio de aplicação de um questionário junto a profissionais que atuam no mercado da tecnologia, onde 20 desenvolvedores e líderes técnicos opinaram sobre quais práticas de codificação são seguidas no seu dia-a-dia.

Foram selecionadas 51 regras de padrões de código e de boas práticas da programação. A partir desta listagem, relacionou-se cada parâmetro com sua respectiva regra no *ESLint* e seus *plugins*, tornando possível que a ferramenta pudesse atuar na verificação de código estático por meio destes padrões.

Todos os passos seguidos foram relatados na última parte do capítulo anterior, dando base para que interessados em adicionar, remover ou moldar regras para outros *frameworks* ou apenas outros padrões atingissem tal objetivo.

O arquivo de configuração está disponível na *internet*, com acesso público: https://gist.github.com/jennifertakagi/987436d95644a9855526f930cfa6c00e. Também encontra-se na íntegra no Apêndice B - Arquivo de configuração do *ESLint* para projetos construídos com *React JS*.

9.2. Em relação aos objetivos específicos

Quanto aos objetivos específicos, foram mapeados cinco pontos:

1. Identificar e selecionar padrões de código e boas práticas de código limpo utilizando a linguagem *JavaScript*.

Seguindo os autores Stefanov (2010, s.p.) e Padolsey (2020, s.p.) pode-se selecionar uma lista de padrões de codificação e boas práticas voltadas à linguagem *JavaScript*. Também foram introduzidas explicações de uso de alguns padrões retirados das bibliografias de Porcello e Banks (2020, s.p.) e de Flanagan (2020, s.p.).

2. Identificar e selecionar padrões de código e boas práticas de código limpo utilizando a *library React JS.*

Quanto às melhores práticas e convenções utilizadas na biblioteca *React JS,* a base de estudo foi a obra de Roldan (2020, s.p.). A lista, apesar de não ser extensa, trouxe pontos importantes para a legibilidade principalmente da sintaxe do *JSX.*

3. Validar as regras escolhidas através de bibliografia por meio de aplicação de um questionário junto a profissionais do mercado.

Após elencar as regras embasadas em bibliografias de destaque, aplicou-se um questionário *online* junto a profissionais da área de programação que atuassem com a *library React JS* na empresa. Obtiveram-se 20 respondentes válidos, e, a partir deste números, as regras que tivessem ao menos 10 adeptos foram selecionadas para a integração na ferramenta.

4. Integrar o *ESLint* ao projeto real, especificando todas as suas configurações.

Para realizar a integração dos padrões escolhidos por meio de bibliografias junto ao *ESLint*, pesquisou-se no site oficial da ferramenta e de *plugins* conhecidos quais eram as regras relacionadas. Em seguida, foram relacionadas as opções suportadas para cada regra a fim de moldar a ferramenta da maneira como à maioria dos profissionais que responderam o questionário trabalham.

Aplicaram-se as regras e configurou-se a ferramenta para que o *ESLint* se tornasse totalmente funcional e capaz de auxiliar na concepção e manutenção de código limpo em projetos escritos com *React JS*.

5. Testar as especificações no projeto escrito em *React JS*.

Para o último objetivo, foi configurado um script para executar a ferramenta e realizar as verificações estáticas do código seguindo as configurações realizadas. Erros apareceram e foram corrigidos automaticamente e manualmente. Porém neste processo identificou-se um ponto que Roldan (2020, s.p.) abordou em seu livro: tocar o *ESLint* manualmente a cada código escrito não era suficiente e é cansativo. A partir deste problema mapeou-se a instalação de um *plugin* no editor de código *VSCode* possibilitando a verificação de código de forma automática e orgânica.

9.3. Trabalhos futuros

No decorrer deste trabalho identificou-se que o superset *TypeScript* está se popularizando entre os desenvolvedores, como analisado pela pesquisa do *StackOverflow* (2020) na figura 122. Segundo Porcello e Alex (2020, s.p.) *TypeScript* é uma ferramenta popular para verificação de tipos em aplicativos *React* e foi projetado para auxiliar os desenvolvedores a encontrarem erros e iterar mais

rapidamente nos projetos.

Portanto, seria um avanço interessante neste estudo abranger padrões de códigos e boas práticas envolvendo o *superset*,a fim de integrá-lo com *plugin* disponível para *ESLint* construído com intuito de auxiliar o trabalho com *TypeScript*.

Figura 122 - Resultado geral da questão sobre sobre Linguagens mais amadas, temidas e desejadas na pesquisa *StackOverflow Survey 2020.*

Fonte: *STACKOVERFLOW SURVEY,* 2020.

Um segundo ponto notado a partir da obra de Roldan (2020, s.p.) foi o uso da ferramenta *Husky* no projeto. A ferramenta é um pacote *NPM* que possibilita estruturar passos para integração contínua. O uso deste poderia evitar que códigos que não seguem as regras do *ESLint* fossem adicionados ao repositório.

Como a primeira coisa a fazer seria adicionar o *ESLint* ao processo usando *Git Hooks*, poderíamos habilitar o *Husky* para executar o *linter* em um *pre-commit.*

Por fim, um próximo passo seria testar a solução entregue no fim deste trabalho em outros projetos construídos com *React JS*, a fim de validar a ferramenta em códigos de diversos desenvolvedores.

REFERÊNCIAS

AIRBNB/JAVASCRIPT. ***Airbnb/Javascript**: JavaScript Style Guide*. Disponível em: <https://github.com/airbnb/javascript>. Acesso em: 27 jan. 2021.

AMOEDO, Raphael. **Alcançando um *Software* Maduro**. São Paulo. 2019. *Ebook.* Sem paginação.

ANDRADE, Elisabeth. **Informática aplicada CSI 1304**: Princípios de programação. Nairobi: Universidade Virtual Africana, 2017.

DERKS, Roy. ***React Projects***. Packt Publishing. 2019. *Ebook.* Sem paginação.

DOUGLAS, Eric. ***JavaScript* furtivo**. LeanPub, 2014.

ESLINT. ***ESLint - Pluggable JavaScript lint**.* Disponível em: <https://eslint.org/>. Acesso em: 26 jan. 2021.

FACEBOOK OPEN SOURCE. **GLOSSÁRIO de termos *React***. Disponível em: <https://pt-br.reactjs.org/docs/glossary.html>. Acesso em: 28 jan. 2021.

FERGUSON, Andrew D.. *A history of Computer Programming Languages*. **Brown University**, Providence, v.1, n. 1, p.1, 2000. Disponível em <https://cs.brown.edu/~adf/programming_languages.html>. Acesso em: 30 jan. 2020.

FLANAGAN, David. ***Javascript***: O Guia Definitivo. 7ª edição. O'Reilly Media Inc.. 2020. *Ebook*, sem paginação.

FOWLER, Martin. ***Patterns of enterprise application architecture*.** Addison-Wesley Longman Publishing Co., Inc., 2002. *Ebook*, sem paginação.

FONSECA, J. J. S. **Metodologia da pesquisa científica**. Fortaleza: UEC, 2002. Apostila.

GIL, A. C. **Como elaborar projetos de pesquisa**. 4ª edição. São Paulo: Atlas, 2007.

GOOGLE JAVASCRIPT STYLE GUIDE. ***Google JavaScript Style Guide***. Disponível em: <https://google.github.io/styleguide/jsguide.html>. Acesso em: 27 jan. 2021.

GUSMÃO, Gustavo. **Seu estilo de programar é quase tão único quanto uma impressão digital**. Exame, 05 fev. de 2015. Disponível em <https://exame.com/tecnologia/seu-estilo-de-programar-e-quase-tao-unico-quanto-uma-impressao-digital/>. Acesso em: 11 ago. 2020.

HAVERBEKE, Marijin. ***Eloquent Javascript****: A Modern Introduction to Programming*. 3ª edição. 2018. *Ebook.*

HERRERO, Guillermo. ***Clean Code****: Return from the dark side*. 2019. *Ebook.* Sem paginação.

IBGE - Instituto Brasileiro de Geografia e Estatística. Estatísticas do Cadastro Central De Empresas. 2018.

INSTITUTO FEDERAL DE SANTA CATARINA, 2018, Florianópolis. **Projeto Pedagógico do Curso (PPC) - 2018**. Florianópolis: Ifsc, 2017. 123 p. Disponível em: <https://moodle.ifsc.edu.br/pluginfile.php/58896/mod_resource/content/1/PPC_GTI_2016_19_09_2017.pdf>. Acesso em: 12 ago. 2020.

JAVASCRIPT STANDARD STYLE. ***JavaScript Standard Style***. Disponível em: <https://standardjs.com/rules.html>. Acesso em: 27 jan. 2021.

KEREKI, Federico. ***Modern JavaScript Web Development Cookbook.*** Packt Publishing. 2018. *Ebook.* Sem paginação.

LEWIS, Elijah. ***CLEAN CODE****: Advanced and Effective Strategies To Use Clean Code Methods*. São Paulo. 2020. *Ebook.* Sem paginação.

MADHAWA, Aeroporto de. ***A brief history of Computer Programming***. Medium, 24 de mar. de 2020. Disponível em: <https://medium.com/@madhawa.uop/a-brief-history-of-computer-programming-8e1f3c0f5d3a>. Acesso em: 30 jan. 2021.

MARTIN, Robert C.. **Código Limpo**: Habilidades Práticas do *Agile Software*. São Paulo: Atlas. 2012.

MINAYO, Maria Cecília de Souza. **O desafio do conhecimento**. São Paulo: Hucitec, 1993.

PADOLSEY, James. **Clean Code in JavaScript**. Packt Publishing. 2020. *Ebook.* Sem paginação.

PORCELLO, Eve; BANKS, Alex. **Learning React**. 2ª edição. O'Reilly Media, Inc. 2020. *Ebook.* Sem paginação.

PRESSMAN, Roger S.. **Engenharia de *Software***: uma abordagem profissional. 7ª edição. Porto Alegre: AMGH. 2011.

ROLDAN, Carlos Santana. ***React Design Patterns and Best Practices***. Packt Publishing. 2019. *Ebook.* Sem paginação.

SILVA, Edna Lúcia da. **Metodologia da pesquisa e elaboração de dissertação**. 4ª edição. Florianópolis: UFSC. 2005.

SOUSA, Bruno Jefferson de; DIAS, José Jorge Lima.; FORMIGA, Andrei de Araújo. **Introdução a Programação**. João Pessoa: Editora da UFPB. 2014.

STACKOVERFLOW. ***Develop Survey Result*** **2020**. Disponível em: <https://insights.stackoverflow.com/survey/2020>. Acesso em: 04 jan. de 2021.

STEFANOV, Stoyan. ***JavaScript Patterns***.O'Reilly Media, Inc. 2010. *Ebook.* Sem paginação.

VOICE, Ray. ***Simply ES6****: Mastering JavaScript and ES6 to its fullest*. 2019. *Ebook*. Sem paginação.

WASSBERG, Joakim. **Computer Programming for Absolute Beginners.** Packt Publishing. 2020. *Ebook*. Sem paginação.

GLOSSÁRIO

***Allman*:** "o estilo *Allman*, que leva o nome do programador americano Eric Allman, coloca todos os colchetes de abertura em uma nova linha" (WASSBERG, 2020, s.p., tradução nossa).

Argumento de uma função: "é chamado com um argumento, o valor que é comunicado ao procedimento" (SOUSA; DIAS; FORMIGA, 2014, p. 68).

Binário: "um sistema de numeração que funciona na base 2 - ou seja, usa apenas dois dígitos: zero e um" (WASSBERG, 2020, s.p., tradução nossa).

Byte: "grupo de dígitos binários, geralmente 8, que funciona como uma única unidade" (WASSBERG, 2020, s.p., tradução nossa).

***Booleano*:** este tipo de dado "tem apenas dois valores: verdadeiro e falso (que são escritos como *true* e *false* respectivamente)" (HAVERBEKE, 2018, p. 8, tradução nossa).

***Bug*:** "uma *tag* usada como comentário para indicar o código que contém um *bug* que ainda não foi corrigido" (WASSBERG, 2020, s.p.).

***Camel Case*:** "uma prática para nomear identificadores [...] No *Camel Case,* a primeira palavra é escrita em letras minúsculas e o primeiro caractere em qualquer palavra que se segue é escrito em letras maiúsculas. Todos os espaços são removidos" (WASSBERG, 2020, s.p., tradução nossa).

***Código-fonte*:** "o código que um programador escreve é chamado de código-fonte" (WASSBERG, 2020, s.p., tradução nossa).

***Console.log*:** é um método do console que envia uma mensagem ao console da *web*.

***Core*:** em termos de *software*, o termo se refere ao núcleo do projeto.

***Deploy*:** "é o processo de garantir que o *software* esteja instalado no local correto, que esteja protegido para evitar ataques de *hackers* e que os privilégios sejam dados ao *software* para que ele possa ler e gravar arquivos conforme necessário" (WASSBERG, 2020, s.p., tradução nossa).

***Hypertext Markup Language* (*HTML*):** a linguagem de marcação padrão para documentos projetados para serem exibidos em um navegador da *web*.

Indentação: "uma técnica para tornar o código mais legível, onde as linhas de código têm um espaço vazio inicial no início da linha" (WASSBERG, 2020, s.p., tradução nossa).

JSON: "significa *"JavaScript Object Notation"* e sua sintaxe é muito semelhante à do objeto *JavaScript*" (FLANAGAN, 2020, s.p., tradução nossa).

***Keyword*:** "em uma linguagem de programação, uma palavra-chave (*keyword*) é uma palavra reservada que possui um significado específico nessa linguagem." (WASSBERG, 2020, s.p., tradução nossa).

Minificar: "é o processo de eliminação de espaços em branco, comentários e outras partes não essenciais do código *JavaScript* para diminuir o tamanho dos arquivos que precisam ser transferidos do servidor para o navegador" (STEFANOV, 2010, s.p., tradução nossa).

Objetos: "tabelas *hash* de pares de valores-chave [...] Os valores podem ser primitivos ou outros objetos; em ambos os casos, eles são chamados de propriedades. Os valores também podem ser funções, caso em que são chamados de métodos" STEFANOV, 2010, s.p., tradução nossa).

***Pascal Case*:** uma denominação em inglês para a prática de escrever as palavras

compostas ou frases, onde cada palavra é iniciada com maiúsculas e unidas sem espaços.

Parâmetro de uma função: utilizada para quando se quer "tornar um procedimento (ou função) mais flexível, é preciso que informações sejam passadas para o procedimento" (SOUSA; DIAS; FORMIGA, 2014, p. 67).

Regex: "expressões regulares (também conhecidas como *RegExp* e *regex*), delimitadas por barras, para definir um padrão de caracteres para corresponder a uma *string*" (PADOLSEY, 2020, s.p., tradução nossa).

Sugar Syntax**:** são considerados códigos atalhos, onde se escreve menos para se fazer a mesma coisa, arrow function é um exemplo.

Transpilar: ato de receber o código-fonte de um programa escrito em uma linguagem de programação como sua entrada e produzir um código-fonte equivalente na mesma linguagem de programação ou em outra.

TypeScript**:** "*superset* de código aberto de *JavaScript*, o que significa que adiciona recursos adicionais à linguagem. Criado na Microsoft, o TypeScript foi projetado para ser usado para ajudar os desenvolvedores a encontrar *bugs* e iterar mais rapidamente nos projetos" (PORCELLO e BANKS, 2020, s.p., tradução nossa).

ANEXO A - Perguntas de Configuração do *ESLint*

Seguem abaixo as perguntas e múltiplas escolhas dadas ao rodar o comando *"eslint --init"*, a fim de se contruir o arquivo de configuração do *ESLint* (tradução nossa):

1. Qual o objetivo do uso do *ESLint*?

❏ Para checar apenas a sintaxe.

❏ Para chegar a sintaxe e encontrar problemas.

❏ Para checar a sintaxe, encontrar problemas e forçar estilo de código.

2. Qual tipo de módulos o projeto utiliza?

❏ Módulo *JavaScript* (*import/export*).

❏ *CommomJS* (*require/exports*).

❏ Nenhuma dessas.

3. Qual *framework* o projeto utiliza?

❏ *React.*

❏ *Vue.js.*

❏ Nenhum deles.

4. O projeto utiliza *TypeScript*?

❏ Não.

❏ Sim.

5. Onde o código irá ser rodado?

❏ *Browser.*

❏ *Node.*

6. Como você gostaria de definir o estilo para o seu projeto?

❏ Usar um estilo popular.

❏ Responder perguntas sobre seu estilo.

❏ Inspecionar seu arquivo *JavaScript*.

7. Qual extensão será utilizada pelo arquivo de configuração?

❏ *JavaScript.*

❏ *YAML.*

❏ *JSON.*

8. Qual estilo de indentação você usa?

❏ *Tabs.*

❏ Espaço.

9. Qual tipo de aspas você utiliza para *string*?

❏ *Double.*

❏ *Single.*

10. Qual tipo de linha para final do arquivo você utiliza?

❏ *Unix*

❏ *Windows.*

11. Você utiliza ponto e vírgula?

❏ Sim.

❏ Não.

APÊNDICE A - Questionário sobre Padrões de Código e Boas Práticas da Programação

Este questionário será utilizado como base para o desenvolvimento do meu Trabalho de Conclusão de Curso de Gestão da Tecnologia da Informação no IFSC, com o tema: ESPECIFICAÇÃO DE PARÂMETROS PARA O USO DA FERRAMENTA ESLINT EM PROJETOS *REACT JS* PARA MANUTENÇÃO DE CÓDIGO LIMPO.

O objetivo é entender quais os Padrões de Código e Boas Práticas da Programação mais utilizados pelos profissionais que trabalham com a *library React JS* a fim de implementar estas regras na ferramenta *ESLint*. Como resultado final será disponibilizado um arquivo com essas configurações no *GitHub*.

1. Qual o seu cargo na empresa atualmente?

❏ Desenvolvedor
❏ *Tester* ou *Quality Assurance*
❏ Líder Técnico
❏ Outro: _____

2. Com qual tipo de desenvolvimento você trabalha atualmente?

❏ Desenvolvimento *Web*
❏ Desenvolvimento *Mobile*
❏ Desenvolvimento de *Software*
❏ Outro: _____

3. Atualmente você atua diretamente com a *library React JS?*

❏ Sim
❏ Não

4. Há quanto tempo você está na área de tecnologia?

❏ 0 a 3 anos

❏ 4 a 6 anos
❏ Mais de 7 anos

5. Qual o tipo de empresa onde você trabalha?
❏ Micro (até 9 empregados)
❏ Pequena (de 10 a 49 empregados)
❏ Média (de 50 a 99 empregados)
❏ Grande (mais de 100 empregados)

6. Com que frequência você sente que perde tempo tentando compreender códigos de projetos na sua vida profissional?
❏ Muito frequentemente
❏ Frequentemente
❏ Ocasionalmente
❏ Raramente
❏ Nunca

7. Você acredita que seguir as boas práticas de programação impacta positivamente no seu trabalho?
❏ Acredito totalmente
❏ Acredito
❏ Não consigo opinar sobre
❏ Não acredito

8. Você acredita que definir Padrões de Código junto ao seu time impacta positivamente no seu trabalho?
❏ Acredito totalmente
❏ Acredito
❏ Não consigo opinar sobre
❏ Não acredito
9. [*ARRAY*] Selecione quais padrões e práticas foram definidos pela sua equipe:

- ❏ Utilizar *destructuring* para acessar várias posições de um *array.*
- ❏ Utilizar sintaxe literal para criação de *Array*; ex: *const array1 = []*.
- ❏ Sempre retornar algo quando utilizar *high order functions*; ex: *map, filter, reduce.*

10. [*BLOCKS*] Selecione quais padrões e práticas foram definidos pela sua equipe:

- ❏ Utilizar chaves em comandos multi-linhas; ex: *if (test) { return true; }.*
- ❏ Manter o *else* na mesma linha dos chaves; ex: *...} else {...}.*
- ❏ No caso de um retorno no *if/else*, não utilizar essa palavra-chave e sim manter o retorno fora do escopo.

11. [CLASSES] Selecione quais padrões e práticas foram definidos pela sua equipe:

- ❏ Sempre definir um *constructor*, no caso de não existir, criar a função e deixar vazia.
- ❏ Evitar definir dois métodos com o mesmo nome dentro de uma classe.
- ❏ Utilizar a *keyword "this"* quando se estiver utilizando propriedades da classe.

12. [*FUNCTIONS*] Selecione quais padrões e práticas foram definidos pela sua equipe:

- ❏ Utilizar *default value* em parâmetros de função.
- ❏ Não modificar os argumentos recebidos dentro da função.
- ❏ Para funções com vários parâmetros, deixe cada um destes em uma linha.
- ❏ Utilizar a notação de *arrow function*.
- ❏ Omitir a *keyword "return"* em *arrow functions* que ocupem apenas uma linha.
- ❏ Utilizar parênteses entre argumentos (em *arrow function*) mesmo

que contenha somente um.

13. [*MODULES*] Selecione quais padrões e práticas foram definidos pela sua equipe:

❏ Unificar chamadas de módulos; ex: *import React, {useEffect} from 'react'.*

❏ Em módulos com apenas uma exportação, utilizar exportação *default*.

❏ Manter todas as importações no topo do arquivo.

❏ Não incluir a extensão *.js* no final dos arquivos importados.

14. [*STRING*] Selecione quais padrões e práticas foram definidos pela sua equipe:

❏ Utilizar *single quote* (aspas simples) para *string.*

❏ Utilizar *template literal string* ao concatenar *strings.*

15. [*OBJECTS*] Selecione quais padrões e práticas foram definidos pela sua equipe:

❏ Utilizam sintaxe literal para criação de objetos; ex: *const object1 = {}.*

❏ Utilizar *destructuring* para acessar várias propriedades de um objeto.

❏ Utilizar *shorthand* em objetos (no caso de propriedade e valor terem o mesmo nome).

❏ Acessar propriedades com *dot notation*; ex: *object1.name.*

16. [*REACT JS*] Selecione quais padrões e práticas foram definidos pela sua equipe:

❏ Nomear componentes como *Pascal Case*; ex: *const reservationItem = <ReservationCard />.*

❏ Alinhar estilos para a sintaxe de *JSX*; ex: pular linhas quando existir muitos atributos dentro do componente.

- ❏ Utilizar *double quotes* (aspas duplas) para atributos *JSX*.
- ❏ Adicionar um espaço antes do fechamento de *tags*; ex: *<Foo />*.
- ❏ Não adicionar espaço dentro de chaves no *JSX*; ex: *<Foo bar={ baz } />.*
- ❏ Omitir valor da *prop* no caso de ser implicitamente *true*; ex: *<Foo hidden />.*
- ❏ Sempre adicionar a propriedade *alt* em *tags* de *<img>.*
- ❏ Evitar utilizar *index* de *array*; ex: *key, {todos.map((todo, index) => {... key={index}}.*
- ❏ Utilizar parênteses quando o retorno for apenas de uma linha.
- ❏ Sempre fechar *tags* nela mesmo, quando não houver filhas; ex: *<Foo />*.

17. [*SYNTAX*] Selecione quais padrões e práticas foram definidos pela sua equipe:

- ❏ Comparar valores com *triple equation*; ex: *a === b.*
- ❏ Utilizar operador ternário em apenas um nível.
- ❏ Separar operadores de comparação de lógicos; ex: *if (a || (b && c)) {...}.*
- ❏ Utilizar *High Order Functions* para iterações ao invés de *loops* simples.
- ❏ Nomear variáveis, funções e instâncias com *Camel Case*; ex: *const user = {...}.*
- ❏ Nomear *constructors* e classes como *Pascal Case*; ex: *class User () {...}.*
- ❏ Evitar nomes de variáveis e funções com apenas uma letra; ex: *function a () {...}.*
- ❏ Utilizar ponto e vírgula no final de comandos.
- ❏ Adicionar vírgulas no final de keywords que utilizam a sintaxe; ex: *const hero = ['Batman', 'Superman',].*
- ❏ Ter um máximo de caracteres por linha; ex: 100 ou 120.

18. [VARIÁVEIS] Selecione quais padrões e práticas foram definidos pela sua equipe:

❏ Utilizar "*const/le*t" para declaração de variáveis ao invés de "*var*".

❏ Utilizar apenas um "*const/let*" por declaração de variável, evitar: *const name, age = true*.

❏ Não utilizar cadeira de variáveis, como: *const a, b, c = 1.*

❏ Excluir variáveis não utilizadas no código.

19. [*WHITE SPACES*] Selecione quais padrões e práticas foram definidos pela sua equipe:

❏ Indentar o código utilizando 2 espaços.

❏ Adicionar espaço após chamada de funções, estruturas de repetição e condicionais; ex: *function callMe () {}.*

❏ Adicionar espaço entre operadores; ex: *const x = y + 5.*

❏ Pular uma nova linha no final de cada arquivo.

❏ Não pular linha após início de bloco.

❏ Não adicionar espaço em parênteses; ex: *function getAge (birthday) {...}.*

❏ Não adicionar espaço em colchetes; ex: *const foo = [1, 2, 3].*

❏ Adicionar espaço dentro de chaves; ex: *const foo = { clark: 'kent' }.*

❏ Evitar espaços a mais no final de linhas de código (*trailing spaces*).

❏ No começo de um comentário, manter um espaço; ex: *// hey you.*

20. Caso tenha faltado algum Padrão ou Boa Prática que seu time utilize, este campo é destinado a isso: _____

APÊNDICE B - Arquivo de configuração do *ESLint* para projetos construídos com *React JS*

```
module.exports = {
       "env": {
       "browser": true,
       "es2021": true
   },
   "extends": [],
   "parserOptions": {
       "ecmaFeatures": {
           "jsx": true
       },
       "ecmaVersion": 12,
       "sourceType": "module"
   },
   "plugins": [
       "import",
       "jsx-a11y",
       "react"
   ],
   "rules": {
       // enforces return statements in callbacks of array's methods
       // https://eslint.org/docs/rules/array-callback-return
       "array-callback-return": ["error", {
               "allowImplicit": true
           }
       ],

        // enforces usage of destructuring instead of accessing a property through a
member expression
                                                                                //
https://eslint.org/docs/rules/prefer-destructuring#prefer-destructuring-from-arrays
-and-objects-prefer-destructuring
       "prefer-destructuring": ["error", {
               "array": true,
               "object": true
           },
           {
               "enforceForRenamedProperties": false
           }
```

```
        ],

        // specify curly brace conventions for all control statements: multiline
                                                                                //
https://eslint.org/docs/rules/curly#require-following-curly-brace-conventions-curly
        "curly": ["error", "multi-line"],

        // enforce one true brace style
        // https://eslint.org/docs/rules/brace-style#require-brace-style-brace-style
        "brace-style": ["error", "1tbs", {
                "allowSingleLine": true
            }
        ],

        // enforce that class methods use "this"
        // https://eslint.org/docs/rules/class-methods-use-this
        "class-methods-use-this": ["error", {
                "exceptMethods": []
            }
        ],

        // disallow duplicate names in class members
                                                                                //
https://eslint.org/docs/rules/no-dupe-class-members#disallow-duplicate-name-in-clas
s-members-no-dupe-class-members
        "no-dupe-class-members": ["error"],

        // require method and property shorthand syntax for object literals
        // https://eslint.org/docs/rules/object-shorthand
        "prefer-arrow-callback": ["error", {
                "allowNamedFunctions": false,
                "allowUnboundThis": true
            }
        ],

        // enforces parentheses around arrow function parameters regardless of arity
                                                                                //
https://eslint.org/docs/rules/arrow-parens#require-parens-in-arrow-function-argumen
ts-arrow-parens
        "arrow-parens": ["error", "always", {
                "requireForBlockBody": true
            }
```

```
        ],

        // enforce or disallow the use of braces around arrow function body
                                                                                //
https://eslint.org/docs/rules/arrow-body-style#require-braces-in-arrow-function-bod
y-arrow-body-style
        "arrow-body-style": ["error", "as-needed", {
                "requireReturnForObjectLiteral": false
            }
        ],

             // disallow reassignment of function parameters except for specific
exclusions
        // https://eslint.org/docs/rules/no-param-reassign.html
        "no-param-reassign": ["error", {
                "props": true,
                "ignorePropertyModificationsFor": []
            }
        ],

         // require that all imports from a single module exists in a single import
statement
                                                                                //
https://eslint.org/docs/rules/no-duplicate-imports#disallow-duplicate-imports-no-du
plicate-imports
        "no-duplicate-imports": ["error", {
                "includeExports": false
            }
        ],

        // report any imports that come after non-import statements
                                                                                //
https://github.com/benmosher/eslint-plugin-import/blob/master/docs/rules/first.md
        "import/first": ["error", "absolute-first"],

        // forbid the use for any extension
                                                                                //
https://github.com/benmosher/eslint-plugin-import/blob/master/docs/rules/extensions
.md
        "import/extensions": ["error"],

        // specify whether double or single quotes should be used
```

```
                                                                                //
https://eslint.org/docs/rules/quotes#enforce-the-consistent-use-of-either-backticks
-double-or-single-quotes-quotes
       "quotes": ["error", "single", {
               "avoidEscape": true,
               "allowTemplateLiterals": false
           }
       ],

       // disallow the usage of + operators with strings
                                                                                //
https://eslint.org/docs/rules/prefer-template#suggest-using-template-literals-inste
ad-of-string-concatenation-prefer-template
       "prefer-template": ["error"],

       // maintain consistency around the spacing inside of template literals
                                                                                //
https://eslint.org/docs/rules/template-curly-spacing#enforce-usage-of-spacing-in-te
mplate-strings-template-curly-spacing
       "template-curly-spacing": ["error", "never"],

       // disallow use of the Object constructor
                                                                                //
https://eslint.org/docs/rules/no-new-object#disallow-object-constructors-no-new-obj
ect
       "no-new-object": ["error"],

       // enforces the use of the shorthand syntax
                                                                                //
https://eslint.org/docs/rules/object-shorthand#require-object-literal-shorthand-syn
tax-object-shorthand
       "object-shorthand": ["error", "always", {
               "avoidQuotes": true,
               "ignoreConstructors": false,
               "avoidExplicitReturnArrows": false
           }
       ],

       // enforce use of dot notation whenever possible
                                                                                //
https://eslint.org/docs/rules/dot-notation#require-dot-notation-dot-notation
       "dot-notation": ["error", {
```

```
                "allowKeywords": true
            }
        ],

        // enforce components without children to be self-closed
                                                                                //
https://github.com/yannickcr/eslint-plugin-react/blob/master/docs/rules/self-closin
g-comp.md
        "react/self-closing-comp": ["error", {
                "component": true,
                "html": true
            }
        ],

         // enforce that all elements that require alternative text have meaningful
information to relay back to the end user
                                                                                //
https://github.com/jsx-eslint/eslint-plugin-jsx-a11y/blob/master/docs/rules/alt-tex
t.md
        "jsx-a11y/alt-text": [2],

        // enforce the closing bracket location for JSX multiline elements
                                                                                //
https://github.com/yannickcr/eslint-plugin-react/blob/master/docs/rules/jsx-closing
-bracket-location.md
        "react/jsx-closing-bracket-location": [2, {
                "nonEmpty": "tag-aligned",
                "selfClosing": "tag-aligned"
            }
        ],

        // enforce the closing tag location for multiline JSX elements.
                                                                                //
https://github.com/yannickcr/eslint-plugin-react/blob/master/docs/rules/jsx-closing
-tag-location.md
        "react/jsx-closing-tag-location": [2],

        // disallow use of multiple spaces
                                                                                //
https://eslint.org/docs/rules/no-multi-spaces#disallow-multiple-spaces-no-multi-spa
ces
        "no-multi-spaces": ["error", {
```

```
                "ignoreEOLComments": false
            }
        ],

          // enforce or forbid spaces after the opening bracket, before the closing
bracket, before the closing
         // bracket of self-closing elements, and between the angle bracket and slash
of JSX closing or self-closing elements.
                                                                                //
https://github.com/yannickcr/eslint-plugin-react/blob/master/docs/rules/jsx-tag-spa
cing.md
        "react/jsx-tag-spacing": [2, {
                "closingSlash": "never",
                "beforeSelfClosing": "always",
                "afterOpening": "never",
                "beforeClosing": "allow"
            }
        ],

          // enforces coding style that user-defined JSX components are defined and
referenced in PascalCase.
                                                                                //
https://github.com/yannickcr/eslint-plugin-react/blob/master/docs/rules/jsx-pascal-
case.md
        "react/jsx-pascal-case": [2],

        // warn if an element uses an Array index in its key.
                                                                                //
https://github.com/yannickcr/eslint-plugin-react/blob/master/docs/rules/no-array-in
dex-key.md
        "react/no-array-index-key": [2],

        // specify whether double or single quotes should be used in JSX attributes
        // https://eslint.org/docs/rules/jsx-quotes
        "jsx-quotes": ["error", "prefer-double"],

         // enforce or disallow spaces inside of curly braces in JSX attributes and
expressions
                                                                                //
https://github.com/yannickcr/eslint-plugin-react/blob/master/docs/rules/jsx-curly-s
pacing.md
        "react/jsx-curly-spacing": [ 2, { "when": "never" }],
```

```
        // prevent missing parentheses around multiline JSX
                                                                                //
https://github.com/yannickcr/eslint-plugin-react/blob/master/docs/rules/jsx-wrap-mu
ltilines.md
        "react/jsx-wrap-multilines": [2, {
                "declaration": "parens",
                "assignment": "parens",
                "return": "parens",
                "arrow": "parens",
                "condition": "ignore",
                "logical": "ignore",
                "prop": "ignore"
            }
        ],

        // enforce boolean attributes notation in JSX
                                                                                //
https://github.com/yannickcr/eslint-plugin-react/blob/master/docs/rules/jsx-boolean
-value.md
        "react/jsx-boolean-value": [2],

        // require the use of === and !==
        // https://eslint.org/docs/rules/eqeqeq
        "eqeqeq": ["error", "always"],

        // enforces a minimum and/or maximum identifier length convention.
                                                                                //
https://eslint.org/docs/rules/id-length#enforce-minimum-and-maximum-identifier-leng
ths-id-length
        "id-length": ["error"],

        // require camel case names
        // https://eslint.org/docs/rules/camelcase#require-camelcase-camelcase
        "camelcase": ["error", {
                "ignoreImports": true,
                "ignoreGlobals": true,
                "properties": "never",
                "ignoreDestructuring": false
            }
        ],
```

```
        // disallow nested ternary expressions
                                                                                //
https://eslint.org/docs/rules/no-nested-ternary#disallow-nested-ternary-expressions
-no-nested-ternary
        "no-nested-ternary": ["error"],

        // specify the maximum length of a line in your program
        // https://eslint.org/docs/rules/max-len
        "max-len": ["error", 100, 2, {
                "ignoreUrls": true,
                "ignoreComments": false,
                "ignoreRegExpLiterals": true,
                "ignoreStrings": true,
                "ignoreTemplateLiterals": true
            }
        ],

        // requires constructor names to begin with a capital letter
                                                                                //
https://eslint.org/docs/rules/new-cap#require-constructor-names-to-begin-with-a-cap
ital-letter-new-cap
        "new-cap": ["error", {
                "newIsCap": true,
                "capIsNew": true,
                "properties": true
            }
        ],

        // disallow un-paren'd mixes of different operators
        // https://eslint.org/docs/rules/no-mixed-operators
        "no-mixed-operators": ["error", {
                "allowSamePrecedence": false
            }
        ],

        // disallow usage of __iterator__ property
        // https://eslint.org/docs/rules/no-iterator#disallow-iterator-no-iterator
        "no-iterator": ["error"],

        // disallow certain syntax forms
        // https://eslint.org/docs/rules/no-restricted-syntax
        "no-restricted-syntax": ["error", {
```

```
                "selector": "ForInStatement",
                  "message": "for..in loops iterate over the entire prototype chain,
which is virtually never what you want. Use Object.{keys,values,entries}, and
iterate over the resulting array.",
            }, {
                "selector": "ForOfStatement",
                  "message": "iterators/generators require regenerator-runtime, which
is too heavyweight for this guide to allow them. Separately, loops should be
avoided in favor of array iterations.",
            }
        ],

        // require or disallow use of semicolons instead of ASI
                                                                                //
https://eslint.org/docs/rules/semi#require-or-disallow-semicolons-instead-of-asi-se
mi
        "semi": ["error", "always"],

         // suggest using of const declaration for variables that are never modified
after declared
                                                                                //
https://eslint.org/docs/rules/prefer-const#suggest-using-const-prefer-const
        "prefer-const": ["error", {
                "destructuring": "any",
                "ignoreReadBeforeAssign": true
            }
        ],

        // disallow modifying variables that are declared using const
                                                                                //
https://eslint.org/docs/rules/no-const-assign#disallow-modifying-variables-that-are
-declared-using-const-no-const-assign
        "no-const-assign": ["error"],

        // require let or const instead of var
                                                                                //
https://eslint.org/docs/rules/no-var#require-let-or-const-instead-of-var-no-var
        "no-var": ["error"],

          // disallow use of undeclared variables unless mentioned in a /*global */
block
                                                                                //
```

```
https://eslint.org/docs/rules/no-undef#disallow-undeclared-variables-no-undef
       "no-undef": ["error"],

       // disallow declaration of variables that are not used in the code
                                                                                //
https://eslint.org/docs/rules/no-unused-vars#disallow-unused-variables-no-unused-va
rs
       "no-unused-vars": ["error", {
               "vars": "all",
               "args": "after-used",
               "ignoreRestSiblings": true
           }
       ],

       // prevent React to be incorrectly marked as unused
                                                                                //
https://github.com/yannickcr/eslint-plugin-react/blob/master/docs/rules/jsx-uses-re
act.md
       "react/jsx-uses-react": [2],

       // prevent variables used in JSX to be incorrectly marked as unused
                                                                                //
https://github.com/yannickcr/eslint-plugin-react/blob/master/docs/rules/jsx-uses-va
rs.md
       "react/jsx-uses-vars": [2],

       // disallow use of chained assignment expressions
       // https://eslint.org/docs/rules/no-multi-assign
       "no-multi-assign": ["error"],

       // allow just one var statement per function
                                                                                //
https://eslint.org/docs/rules/one-var#enforce-variables-to-be-declared-either-toget
her-or-separately-in-functions-one-var
       "one-var": ["error", "never"],

       // require padding inside curly braces

//https://eslint.org/docs/rules/object-curly-spacing#enforce-consistent-spacing-ins
ide-braces-object-curly-spacing
       "object-curly-spacing": ["error", "always"],
```

```
        // require spaces around operators
                                                                                //
https://eslint.org/docs/rules/space-infix-ops#require-spacing-around-infix-operator
s-space-infix-ops
        "space-infix-ops": ["error"],

        // enforce newline at the end of file, with no multiple empty lines
                                                                                //
https://eslint.org/docs/rules/eol-last#require-or-disallow-newline-at-the-end-of-fi
les-eol-last
        "eol-last": ["error", "always"],

        // disallow trailing whitespace at the end of lines
                                                                                //
https://eslint.org/docs/rules/no-trailing-spaces#disallow-trailing-whitespace-at-th
e-end-of-lines-no-trailing-spaces
        "no-trailing-spaces": ["error", {
                "skipBlankLines": false,
                "ignoreComments": false
            }
        ],

        // require a space before & after certain keywords
                                                                                //
https://eslint.org/docs/rules/keyword-spacing#enforce-consistent-spacing-before-and
-after-keywords-keyword-spacing
        "keyword-spacing": ["error", {
                "before": true,
                "after": true
            }
        ],

        // enforce spacing inside array brackets
                                                                                //
https://eslint.org/docs/rules/array-bracket-spacing#disallow-or-enforce-spaces-insi
de-of-brackets-array-bracket-spacing
        "array-bracket-spacing": ["error", "never"],

        // this option sets a specific tab width for your code
        // https://eslint.org/docs/rules/indent
        "indent": ["error", 2],
```

```
      // require or disallow spaces inside parentheses
                                                                    //
https://eslint.org/docs/rules/space-in-parens#disallow-or-enforce-spaces-inside-of-
parentheses-space-in-parens
      "space-in-parens": ["error", "never"],

         // require or disallow a space immediately following the // or /* in a
comment
      // https://eslint.org/docs/rules/spaced-comment
      "spaced-comment": ["error", "always"],
   }
}
```