



Instituto Federal de Brasília
Campus Brasília
Tecnologia em Sistemas para Internet

Eduardo Rodrigues Estrela

HIPERMÍDIA COMO MOTOR DE ESTADO DE APLICAÇÕES:

Uma análise comparativa frente ao uso de APIs JSON

Brasília
2025

Eduardo Rodrigues Estrela

HIPERMÍDIA COMO MOTOR DE ESTADO DE APLICAÇÕES:

Uma análise comparativa frente ao uso de APIs JSON

Trabalho de Conclusão de Curso apresentado ao curso de Tecnologia em Sistemas para Internet do Instituto Federal de Brasília do *Campus* Brasília, como parte da exigência para obtenção do título de tecnólogo.

Orientador: Fábio Ferraz Fernandez
Campus Brasília / Instituto Federal de Brasília

Brasília
2025

E82 Estrela, Eduardo Rodrigues.

Hipermídia como Motor de Estado de Atualizações: uma análise comparativa frente ao uso de APIs JSON. / Eduardo Rodrigues Estrela. – Brasília, 2025.
281 f. : il. color.

Orientador: Fábio Ferraz Fernandez.

Trabalho de conclusão de curso (Graduação) – Instituto Federal de Educação, Ciência e Tecnologia de Brasília, Tecnologia em Sistemas para Internet, 2025.

1. Hipermídia como Motor de Estado de Aplicações. 2. Transferência de estado Representativo. 3. Renderização na web. I. Fernandez, Fábio Ferraz. (orient.). II. Título.

CDU 004.738.5

Elaborado com os dados fornecidos pelo autor.

Eduardo Rodrigues Estrela

HIPERMÍDIA COMO MOTOR DE ESTADO DE APLICAÇÕES:

Uma análise comparativa frente ao uso de APIs JSON

Trabalho de Conclusão de Curso apresentado ao curso de Tecnologia em Sistemas para Internet do Instituto Federal de Brasília do *Campus* Brasília, como parte da exigência para obtenção do título de tecnólogo.

Aprovado em 9 de dezembro de 2025

BANCA EXAMINADORA

Fábio Ferraz Fernandez
Campus Brasília / Instituto Federal de Brasília
Orientador

Alisson Wilker Andrade Silva
Campus Brasília / Instituto Federal de Brasília
Membro interno

Fernando Wagner Brito Hortêncio Filho
Campus Brasília / Instituto Federal de Brasília
Membro interno

Gostaria de dedicar este trabalho à minha família, que apesar de todas as dificuldades e solavancos no meio do caminho, nunca deixou de acreditar que eu poderia fazer coisas grandiosas, e em especial ao meu pai, por me introduzir à área de tecnologia. Sem ele eu provavelmente teria demorado demais para descobrir a paixão que eu tenho por computadores. Também não poderia deixar de agradecer à minha melhor amiga Andressa por sempre estar ao meu lado e por questionar toda a procedência duvidosa do meu humor que, de algum modo me ajudou a chegar até aqui, a minha gatinha Jade que dormiu inúmeras tardes no meu colo enquanto eu escrevia este trabalho, impossibilitando qualquer tentativa minha de escapar dele. E por fim, gostaria de agradecer os meus mais queridos amigos Louie, Alex, Yves, Vitya, Agatha, Nicolas, Nikko, Lana, Bekie, Summer, Niki, Natalie, Sam, Juliana, Fabiana, Emisis, Chico, Ark, Major, Absol, Lo, Eight e Sebby. Sem estes, eu não teria metade da inspiração e nem mesmo um quarto da coragem para escrever um trabalho como esse.

AGRADECIMENTOS

Gostaria primeiramente de agradecer todos os professores do curso de Tecnologia de Sistemas para Internet do Instituto Federal de Brasília, por todo o conhecimento divulgado, debates propostos e experiências compartilhadas. Sem eles, muito do que é visto neste trabalho não seria remotamente possível.

Trago também um agradecimento especial para os professores Fábio Ferraz Fernandez, meu professor orientador, que nunca me deixou sem preparo e me munuiu com toda as informações que um orientando poderia pedir, ainda mais para um trabalho deste porte e escopo, à Paula Schlemper, professora da matéria de Projeto de Trabalho de Conclusão de Curso, por trazer à luz todo o caminho a ser percorrido neste trabalho com incrível excelência e profissionalidade, e à Dauster Pereira, Fernando Wagner e Sylvana Karla, todos professores orientadores do Grupo de Pesquisa em Tecnologias Computacionais (GPTCode), pelo suporte a partir de avaliações prévias do conteúdo.

Também agradeço aos meus colegas de turma Luidy Baldez, Alice Alves, Milena Araújo, Vitória Santana, Thiago Amâncio, Luiz Fernando, Lucas Matheus e Rayra Feitosa por auxiliar com diversos tópicos e com a estrutura geral deste trabalho.

Por fim, gostaria de trazer dois agradecimentos a alguns indivíduos que fizeram uma contribuição direta acerca do conteúdo ou formato deste trabalho. Primeiramente, um agradecimento especial aos dois administradores do servidor oficial da comunidade HTMX no aplicativo Discord, com nicknames "Latent" e "1cg Head Grug In Charge" que nos auxiliaram com os tópicos relacionados ao funcionamento interno da biblioteca. E por fim, outro agradecimento é devido à comunidade do HTMX por seu trabalho fantástico em demonstrar o uso desta biblioteca de maneira altamente didática com excelência e maestria, nos permitindo um repertório de conhecimento acerca da biblioteca que não seria possível sem estes.

“A perfeição não é alcançada quando já não há mais nada a ser adicionado, mas quando não há mais nada que se possa retirar.”

— Antoine de Saint-Exupéry

RESUMO

Neste trabalho é apresentada uma análise detalhada, contextualizada e parametrizada acerca de APIs que utilizam formatos de hipermídia, como *HTML*, com a estruturação e conteúdos aderentes aos padrões do modelo *REST* para a renderização de interfaces gráficas de clientes que utilizam navegadores para a web. Esta análise é feita em comparação com o uso do formato de dados *JSON* para a transmissão bruta de dados, onde clientes com navegadores necessitam de *scripts* adicionais para a transformação dos dados em páginas visualizáveis pelos usuários. A análise das abordagens foi realizada com a produção de duas provas de conceito de uma mesma aplicação de rede social que utilizam as duas abordagens para uma comparação estatística acerca do desempenho destas, de maneira a proporcionar uma reflexão acerca da viabilidade do uso do formato *HTML*, em conjunto com a *hipermídia como motor de estado de aplicações* para o desenvolvimento de sistemas. Com a realização dos testes propostos, foi possível confirmar cada ponto desenvolvido na hipótese de pesquisa, de modo a confirmar que o uso de *hipermídia como motor de estados* de uma aplicação, mesmo com as suas desvantagens em comparação com o uso do formato *JSON*, possui um maior desempenho técnico. A partir da confirmação das hipóteses, foi possível realizar a proposição de práticas a serem tomadas pelos desenvolvedores, a fim de proporcionar um melhor ecossistema de aplicações para a web, tomando o modelo *REST* como base fundamental.

Palavras-chave: Hipermídia como Motor de Estado de Aplicações. Transferência de estado Representativo. Renderização na web.

ABSTRACT

In this paper there is the presentation of a detailed, contextualized and parameterized analysis on APIs that make usage of hypermedia formats, such as *HTML*, with the structuring and contents adhering to the patterns and restraints of the *REST* model for the rendering of graphical user interfaces for clients with web browsers. This analysis is done in comparison with the usage of the *JSON* data format for the raw data transmission, where clients with web browsers need additional *scripts* in order for the transformation of the data into viewable pages by the users. The analysis of both approaches was achieved with the production of two proof of concepts of the same social network application for a statistical analysis on the performance of these, in a way that allowed a reflection over the viability on the usage of the *HTML* format, paired with the *hypermedia as the engine of application state* for systems development. With the execution of the proposed tests, it was possible to confirm each point stated in the research hypothesis, in a way to validate that the usage of *hypermedia as the engine of application state* on a system, although even with its disadvantages in comparison with the *JSON* format, still had an overall higher technical performance. From the confirmation of the hypothesis, it was also possible to realize the proposition of practices to be exercised by the developers of the area, with the objective of proportioning a better ecosystem of applications for the web, taking the *REST* model as a fundamental basis.

Keywords: Hypermedia as the Engine of Application State. Representational State Transfer. Web Rendering.

LISTA DE TABELAS

Tabela 1 – Cronograma de Atividades	29
Tabela 2 – Resultados WRK (4 <i>Threads</i> , 10 conexões paralelas, 5 minutos)	55
Tabela 3 – Resultados do LCP por rota (Sem Estresse) em segundos	58
Tabela 4 – Resultados do LCP por rota (Estresse - Processamento Lento 4x) em segundos	59
Tabela 5 – Resultados do LCP por rota (Estresse - Internet 4G) em segundos . . .	60

LISTA DE ABREVIATURAS E SIGLAS

AJAX	Assynchronous JavaScript and XML
API	Abstract Programmer Interface
CDN	Content Delivery Network
CORS	Cross Origin Resource Sharing
CPU	Central Processing Unit
GHz	Giga-Hertz
GIF	Graphics Interchange Format
GPU	Graphics Processing Unit
HATEOAS	Hypermedia as the Engine of Application State
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IANA	Internet Assigned Numbers Authority
IETF	Internet Engineering Task Force
IP	Internet Protocol
JSON	JavaScript Object Notation
JS	JavaScript
LCP	Largest Contentfull Paint
MT	Mega-Transfer
REST	Representational State Transfer
RFC	Request For Comment
SOAP	Simple Object Access Protocol
TOML	Tom's Obvious Markup Language
TTY	Teletypewriter
URI	Universal Resource Identifier

URL	Universal Resource Locator
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

SUMÁRIO

1	INTRODUÇÃO	15
1.1	Tema	16
1.2	Problema e Hipótese	17
1.2.1	<i>Objetivo geral</i>	17
1.2.2	<i>Objetivos específicos</i>	18
1.3	Estrutura do TCC	18
2	FUNDAMENTAÇÃO TEÓRICA	19
2.1	Definição de Hipermídia	19
2.2	A Arquitetura da Web e a Evolução das APIs	19
2.3	O Estilo Arquitetural REST	20
2.3.1	<i>Modelo Cliente-Servidor</i>	21
2.3.2	<i>Comunicação Sem Estado (Stateless)</i>	21
2.3.3	<i>Caching</i>	21
2.3.4	<i>Interface Uniforme</i>	21
2.3.5	<i>Sistema em Camadas</i>	23
2.3.6	<i>Código Sob Demanda (Opcional)</i>	23
2.3.7	<i>Síntese das restrições e relação com a hipótese</i>	23
2.4	JSON: origem, popularização e limitações	24
2.5	Hipermídia como Motor de Estado da Aplicação (HATEOAS)	25
3	METODOLOGIA	27
3.1	Classificação da Pesquisa	27
3.2	Procedimentos Metodológicos	27
3.3	Provas de Conceito	28
3.4	CrITÉRIOS de Avaliação	28
3.5	Ferramentas Utilizadas	28
3.6	Cronograma de Atividades	29
4	DESENVOLVIMENTO	30
4.1	Motivações para o desenvolvimento	30
4.2	Sobre o sistema	30
4.3	A arquitetura do sistema	30
4.3.1	<i>Sistema duplamente monolítico</i>	31
4.3.2	<i>Sistema unicamente monolítico</i>	31
4.3.3	<i>Sistema dividido em Microserviços</i>	31

4.3.4	<i>A separação de papéis</i>	31
4.4	A inicialização do sistema	32
4.4.1	<i>As rotas</i>	35
4.5	Diferenças entre as rotas	39
4.6	O modelo da testagem	51
4.7	Os testes	51
4.7.1	<i>O script WRK</i>	52
5	RESULTADOS E DISCUSSÃO	54
5.1	Resultados dos Testes	54
5.1.1	<i>Dos parâmetros de testagem</i>	54
5.1.2	<i>Requisições com WRK</i>	55
5.1.3	<i>Renderização do elemento mais lento</i>	58
5.1.4	<i>A viabilidade dos sistemas</i>	60
5.2	Uma breve discussão sobre os resultados	61
5.2.1	<i>O impacto do Vue</i>	61
5.2.2	<i>O impacto do HTMX</i>	61
6	CONSIDERAÇÕES FINAIS	62
6.1	Dos objetivos	62
6.1.1	<i>Do objetivo Geral</i>	62
6.1.2	<i>Dos objetivos específicos</i>	63
6.2	Da hipótese	63
6.2.1	<i>Desacoplamento</i>	63
6.2.2	<i>A lógica dos sistemas</i>	64
6.2.3	<i>A melhora da performance</i>	65
6.3	Uma discussão acerca da hipermídia como motor de estado de atualizações	65
6.4	Das práticas sugeridas	66
6.4.1	<i>Envio de JSON ou HTML</i>	66
6.4.2	<i>Arquivos estáticos</i>	67
6.4.3	<i>O papel da renderização de componentes</i>	67
6.5	Uma avaliação das ferramentas da coleta de dados	68
6.6	Limitações deste trabalho	68
6.7	Trabalhos futuros	68
	REFERÊNCIAS	69
	A – FUNCIONAMENTO INTERNO DO VUE	70
A.1	Ponto de entrada	70

A.2	Mixin De Inicialização	74
A.2.1	<i>Inicialização de Ciclos de Vida</i>	82
A.2.2	<i>Inicialização de Eventos</i>	83
A.2.3	<i>Inicialização da Renderização</i>	88
A.2.4	<i>Inicialização de Injeções</i>	103
A.2.5	<i>Inicialização de Estados</i>	105
A.2.6	<i>Inicialização de Fornecimento</i>	119
A.3	Mixin de Estados	121
A.4	Mixin de Eventos	126
A.5	Mixin de Ciclos de Vida	131
A.5.1	<i>A função patch()</i>	150
A.6	Mixin de renderização	186
A.7	Um resumo do funcionamento do Vue	191
A.7.1	<i>Inicialização</i>	191
A.7.2	<i>Compilação de Templates</i>	191
A.7.3	<i>Reatividade e o Documento Virtual</i>	191
A.7.4	<i>Ciclos de Vida</i>	191
	APÊNDICE B – FUNCTIONAMENTO INTERNO DO HTMX	192
B.1	Ponto de Entrada	192
B.2	Inicialização do HTMX	194
B.3	As funções do HTMX	245
B.3.1	<i>A função on()</i>	245
B.3.2	<i>A função off()</i>	246
B.3.3	<i>A função onLoad()</i>	246
B.3.4	<i>A função addClassToElement()</i>	246
B.3.5	<i>A função removeClassFromElement()</i>	247
B.3.6	<i>A função toggleClassOnElement()</i>	248
B.3.7	<i>A função parseInterval()</i>	248
B.3.8	<i>A função swap()</i>	249
B.4	Um resumo do funcionamento do HTMX	279
B.4.1	<i>Inicialização</i>	279
B.4.2	<i>Interceptação de eventos</i>	279
B.4.3	<i>Requisição</i>	280
B.4.4	<i>Tratamento da resposta</i>	280
B.4.5	<i>A troca de elementos</i>	280
B.4.6	<i>Trocas fora dos limites</i>	280
B.4.7	<i>Conclusão do processamento</i>	281

1 INTRODUÇÃO

A arquitetura de sistemas distribuídos para a *web*, caracterizada pela separação de papéis entre clientes e servidores de aplicações, tem evoluído significativamente ao longo dos anos, impulsionada pela necessidade de otimizar a comunicação entre servidores e clientes. Tradicionalmente, esse processo consolidou uma separação de responsabilidades: o servidor atua como provedor primário de informações, enquanto o cliente assume o papel de consumidor final. Embora esse modelo tenha sido fundamental para a construção da *web*, sua estrutura rígida impõe limitações, sobretudo no que diz respeito à fluidez e ao dinamismo da interação do usuário. Em aplicações que exigem respostas em tempo real ou elevada interatividade, essa dependência do cliente em relação ao servidor pode resultar em lentidão e experiência menos fluida (Masse, 2011; Richardson; Ruby, 2007).

Desde sua criação em 2001 e posterior padronização pela IANA (*Internet Assigned Numbers Authority*) em 2013 e pela IETF (*Internet Engineering Task Force*) em 2006, o formato JSON (*JavaScript Object Notation*) consolidou-se como padrão predominante para o intercâmbio de dados em aplicações *web* (Crockford, 2006). Sua popularização foi catalisada pela ascensão de *frameworks front-end* baseados em *JavaScript*, como *React*, *Angular*, *Vue* e *Svelte*, os quais operam de forma integrada com dados estruturados em JSON (TwoBitHistory, 2017). Grandes empresas de tecnologia como Google, Twitter e Facebook adotaram o JSON como formato principal de comunicação entre cliente e servidor, o que contribuiu para sua hegemonia na construção de interfaces de programação de aplicações (APIs) (Postman Team, 2021).

Apesar da simplicidade sintática e da compatibilidade com tecnologias modernas, o JSON apresenta uma limitação central: navegadores web não conseguem interpretá-lo diretamente como formato na camada de apresentação. Para isso, é necessário convertê-lo em HTML com o auxílio de *scripts* e *frameworks*, tais como *Vue*, *React*, *Next* e *Svelte*, adicionando uma camada de lógica adicional no lado do cliente (JSON:API, 2024). Essa abordagem contrasta com as propriedades nativas do HTML (*HyperText Markup Language*) e do XML (*eXtensible Markup Language*), que possibilitam não apenas a apresentação dos dados, mas também a inclusão de elementos de navegação e controle, denominados como hipermídia, integrando lógica e conteúdo em uma única resposta (Gross *et al.*, 2024).

O modelo REST (*Representational State Transfer*) surgiu no início dos anos 2000 como uma resposta às limitações do modelo baseado em SOAP (*Simple Object Access Protocol*), que era criticado por sua complexidade e acoplamento excessivo (Amazon Web Services, 2024). Proposto por Roy Fielding em 2000, o REST define um conjunto de **restrições arquiteturais** voltadas à comunicação eficiente entre sistemas distribuídos (Fielding, 2000). No contexto do REST, o termo “restrição” refere-se a diretrizes que limitam intencionalmente as decisões de projeto com o objetivo de promover simplicidade, escalabilidade e

interoperabilidade. As restrições incluem: separação cliente-servidor, ausência de estado, *caching*, interface uniforme a partir de localizadores de recursos uniformes (URLs), identificadores de recursos uniformes (URIs) e hipermídia como motor de estado de aplicações (HATEOAS), sistema em camadas e *scripting*.

Ao longo dos anos, no entanto, consolidou-se a interpretação equivocada de que REST seria sinônimo do uso de HTTP (*Hypertext Transfer Protocol*) com respostas em JSON. Essa leitura limitada foi criticada pelo próprio Fielding, que reforçou que REST não se resume a transporte de dados, mas sim a um estilo arquitetural completo, no qual a hipermídia é peça central (Fielding, 2008; Masse, 2011; Richardson; Ruby, 2007).

Diante desse cenário, o presente trabalho propõe uma análise comparativa entre o modelo tradicional de APIs com respostas em JSON e uma abordagem orientada à hipermídia, especificamente por meio da implementação de HATEOAS para a renderização de interfaces gráficas de usuário. A hipótese considerada é que o uso de hipermídia pode não apenas alinhar melhor o desenvolvimento de aplicações aos princípios REST, como também reduzir o acoplamento entre cliente e servidor, simplificar a lógica de renderização e melhorar a performance em determinados contextos. Tais vantagens podem se mostrar relevantes frente ao crescente uso de *frameworks* complexos, que demandam mais recursos computacionais no lado do cliente para lidar com a estrutura de dados JSON.

Adicionalmente, formatos como HTML e XML, que possibilitam a entrega de conteúdo com *links* embutidos e sem necessidade de pós-processamento, oferecem características alinhadas às mensagens auto-descritivas preconizadas pelo REST (JSON:API, 2024; Gross *et al.*, 2024). Essa abordagem permite que o cliente descubra dinamicamente os estados e transições possíveis da aplicação por meio das respostas da própria API, promovendo maior aderência ao modelo arquitetural original.

Neste contexto, o objetivo deste trabalho é investigar as propriedades funcionais dos formatos JSON e HTML e os conceitos de REST e HATEOAS, de forma a avaliar o uso atual dessas tecnologias e os avanços de aderência ao modelo arquitetural original da web, conforme definido pelo estilo REST. A proposta será operacionalizada pela construção de duas provas de conceito com funcionalidades equivalentes: uma utilizando JSON em conjunto com Vue.js, e outra utilizando HTML com HATEOAS, empregando a biblioteca HTMX, e também com uma análise detalhada do funcionamento destas bibliotecas. Por fim, os resultados serão analisados quanto à performance, manutenibilidade e aderência aos princípios arquiteturais discutidos.

1.1 Tema

Este trabalho tem como tema a análise comparativa entre duas abordagens para o desenvolvimento de APIs: o modelo tradicional baseado em JSON e o modelo orientado à hipermídia como motor de estado de aplicações, especificamente para o consumo de cliente em navegadores para web. A proposta centra-se na relação entre esses modelos

e as restrições arquiteturais do estilo REST, considerando suas aplicações em sistemas distribuídos para a *web*.

1.2 Problema e Hipótese

A adoção generalizada do formato JSON como padrão de comunicação entre cliente e servidor na *web* trouxe ganhos significativos em simplicidade e integração com *frameworks* modernos. No entanto, sua estrutura exige lógica adicional para interpretação e renderização no lado do cliente, o que pode aumentar a complexidade do sistema e o acoplamento entre as camadas da aplicação. Esse modelo, ainda que amplamente difundido, contrasta com os princípios definidos pelo estilo arquitetural REST, proposto por Roy Fielding (Fielding, 2000), especialmente no que se refere à utilização de hipermídia como motor de estado de aplicações.

Diante disso, o presente estudo parte do seguinte problema de pesquisa: *Em que medida o uso da hipermídia como motor de estado da aplicação em APIs web pode representar uma alternativa mais eficiente e aderente aos princípios REST do que o modelo tradicional baseado em respostas JSON?*

A hipótese considerada é que o uso de hipermídia nas respostas da API pode, além de aumentar a aderência ao estilo arquitetural REST, pode reduzir o acoplamento entre cliente e servidor, simplificar a lógica de renderização e melhorar a performance em determinados cenários de aplicação, tudo especificamente para o uso de um cliente final que utiliza um navegador para a *web* como meio de consumo para as aplicações.

Este trabalho não visa remover o uso do JSON ou quaisquer formatos futuros como meio de transmissão de dados por APIs para *web*, muito pelo contrário, ele reconhece que a sua importância ainda é altamente relevante para o contexto do desenvolvimento para *web*. O que este trabalho tem como objetivo é confirmar a hipótese que o uso de formatos não baseados em hipermídia como resposta para requisições de navegadores para *web* são prejudiciais para a performance da aplicação e da manutenção destas aplicações.

Para testar essa hipótese, são apresentados os objetivos deste trabalho, divididos em objetivo geral e objetivos específicos.

1.2.1 Objetivo geral

Investigar as propriedades funcionais dos formatos JSON e HTML, bem como os conceitos de REST e HATEOAS, a fim de avaliar o uso atual dessas tecnologias e seu grau de aderência ao modelo arquitetural original proposto pelo estilo REST no desenvolvimento de aplicações *web*.

1.2.2 Objetivos específicos

- Analisar a estrutura, arquitetura e funcionamento dos códigos-fonte de um *framework* ou biblioteca front-end que utiliza JSON como formato de comunicação e de um *framework* ou biblioteca que utiliza HTML como formato de comunicação.
- Desenvolver duas provas de conceito de uma aplicação de rede social simples, utilizando ambas as metodologias de comunicação, através das tecnologias escolhidas anteriormente;
- Comparar o desempenho entre os dois modelos propostos.

1.3 Estrutura do TCC

Este trabalho está organizado em cinco capítulos principais, e dois apêndices, além das seções introdutórias e finais, com o objetivo de apresentar de forma clara, sistemática e progressiva o desenvolvimento da pesquisa.

- O Capítulo 1 introduz o tema, contextualiza o objeto de estudo, delimita o problema de pesquisa, justifica sua relevância, estabelece os objetivos e define a limitação acerca do que será observado neste trabalho.
- O Capítulo 2 corresponde à fundamentação teórica, abordando os principais conceitos relacionados ao desenvolvimento de APIs *web*, como o estilo arquitetural REST, os formatos JSON e HTML, o conceito de HATEOAS, além da descrição das ferramentas utilizadas, como os *frameworks* Vue.js e HTMX.
- O Capítulo 3 trata da metodologia adotada, detalhando os procedimentos da pesquisa, os critérios de avaliação e as ferramentas utilizadas.
- O Capítulo 4, que será desenvolvido na etapa do TCC, será dedicado à implementação das soluções propostas, à aplicação dos testes e à análise comparativa entre as abordagens com base nos critérios definidos na metodologia.
- O Capítulo 5 apresenta as considerações finais do trabalho, destacando os principais resultados obtidos, as limitações identificadas e sugestões para estudos futuros relacionados ao tema.
- O apêndice A apresenta a análise do funcionamento interno do *framework* JavaScript Vue.
- Por fim, o apêndice B apresenta a análise do funcionamento interno da biblioteca JavaScript HTMX.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão apresentados os conceitos e bases teóricas essenciais para o desenvolvimento desta pesquisa, tendo como objetivo embasar a proposta de comparação entre as duas diferentes abordagens arquiteturais de desenvolvimento de APIs *web*. Serão abordados temas como a definição de hipermídia, a evolução da arquitetura da *web*, o estilo arquitetural REST, os formatos de comunicação JSON e HTML e o conceito de HATEOAS. Os conceitos apresentados serão utilizados para definição de técnicas metodológicas no capítulo 3 e a análise de resultados utilizadas para avaliação de alcance dos objetivos.

2.1 Definição de Hipermídia

O conceito de hipermídia é intrinsecamente ligado ao conceito de hipertexto, uma vez que o primeiro é um derivado do segundo, historicamente. Na sua difusão inicial, a *web* era amplamente utilizada para a transmissão de documentos de texto, devido as limitações práticas do modelo HTML na sua primeira versão. Com a sua definição, surge também o conceito de hipertexto: uma categoria de informação baseada em texto que, devido a sua implementação através de uma máquina informatizada, permite o seu consumo assincronamente em relação ao usuário do sistema.

Isto é, ao demonstrar informações em formato de hipertexto, o usuário do sistema pode ou não consumir essas informações na ordem em que aparecem em uma interface gráfica, sem em momento algum sofrer um impacto acerca do entendimento sobre a informação disponível.

Durante as subseqüentes atualizações do formato HTML, em suas especificações, há uma difusão de novos tipos de informação não-textuais dentro do formato HTML, o que permite a demonstração gráfica de novos tipos de mídia, como vídeos, imagens, formatos de troca de gráficos (GIFs), arquivos puros, dentre alguns outros.

Essa nova possibilidade de demonstrar variados formatos de mídia acaba esculpindo a extensão do conceito de hipertexto para o conceito de hipermídia: uma categoria de mídia, independente de sua base que devido a sua implementação através de uma máquina informatizada, permite o seu consumo assincronamente em relação ao usuário do sistema.

2.2 A Arquitetura da Web e a Evolução das APIs

A arquitetura da *web* foi concebida e existe até hoje com base no modelo cliente-servidor, onde o cliente realiza as requisições e o servidor responde com os recursos que foram solicitados. Esse modelo se baseia na separação de responsabilidades, proporcionando escalabilidade e simplicidade no desenvolvimento de aplicações distribuídas (Fielding, 2000). Porém, à medida que a *web* evoluiu de documentos estáticos para aplica-

ções interativas, tornou-se necessário criar formas mais eficientes de comunicação entre essas duas camadas.

Nesse contexto, surgiram as APIs, que passaram a mediar a interação entre sistemas de forma programável, estruturada e reutilizável. As APIs permitiram que aplicações (frontend) se comunicassem com servidores *backend* de maneira padronizada, utilizando protocolos como HTTP e formatos de dados como XML e, posteriormente JSON (Masse, 2011).

Inicialmente, APIs baseadas em XML eram comuns, principalmente em modelos como o SOAP, que prezavam por formalismo e robustez, mas exigiam maior complexidade de implementação (Richardson; Ruby, 2007). Com o crescimento da web e a necessidade de aplicações mais leves e responsivas, surgiu uma preferência por formatos mais simples, o que culminou na ampla adoção do JSON como padrão de fato para a troca de dados em APIs modernas. O JSON, criado por Douglas Crockford em 2001, tornou-se popular devido à sua simplicidade e legibilidade em comparação ao XML (Crockford, 2006).

Além da escolha do formato, a estrutura arquitetural das APIs também evoluiu. A partir dos anos 2000, com a publicação da tese de Roy Fielding, popularizou-se o estilo arquitetural REST, que propunha diretrizes para tornar a web mais escalável, flexível e padronizada (Fielding, 2000). O modelo REST passou a orientar a construção de APIs baseadas em recursos, utilizando os métodos do protocolo HTTP e promovendo a uniformidade da interface entre cliente e servidor.

Contudo, com o tempo, o termo “API REST” passou a ser utilizado de forma genérica, muitas vezes desvinculado das restrições propostas originalmente. Em muitas implementações modernas, REST se resume a um conjunto de rotas HTTP que expõem dados em formato JSON, sem considerar aspectos fundamentais como *caching*, presença de hipermídia ou descoberta dinâmica de recursos (Richardson; Ruby, 2007; Masse, 2011). Essa abordagem reduz REST a uma simples convenção de transporte de dados, ignorando suas propriedades arquiteturais essenciais.

Essa distorção conceitual tem implicações práticas negativas: o aumento do acoplamento entre cliente e servidor, a ausência de navegação orientada por *links* (hipermídia), e a perda de vantagens como escalabilidade e independência de estados. Tais limitações motivaram o surgimento de padrões complementares e novas abordagens que buscam recuperar os princípios originais da web, como o uso de HTML com controles de hipermídia e bibliotecas como o HTMX, que promovem interações mais fiéis ao modelo REST descrito por Fielding.

2.3 O Estilo Arquitetural REST

O estilo arquitetural REST, sigla para transferência de estado representativa, foi proposto por Roy Thomas Fielding. O modelo define um conjunto de restrições arquiteturais voltadas para a construção de sistemas distribuídos escaláveis, interoperáveis e baseados

em recursos acessados por meio do protocolo HTTP (Fielding, 2000).

A seguir, são apresentadas as seis restrições fundamentais que caracterizam esse estilo arquitetural:

2.3.1 Modelo Cliente-Servidor

Estabelece a separação de responsabilidades entre cliente e servidor. O cliente cuida da interface gráfica e da experiência do usuário, enquanto o servidor é responsável pela lógica de negócio e da persistência de dados. Essa separação permite maior escalabilidade e independência entre os componentes.

2.3.2 Comunicação Sem Estado (Stateless)

Cada requisição enviada pelo cliente deve conter todas as informações necessárias para que o servidor a compreenda e a processe, sem depender de nenhum estado mantido em memória no servidor entre requisições. Isso significa que o cliente não pode utilizar qualquer contexto adicional além do que está proposto em uma resposta HTTP.

Essa restrição, no entanto, costuma ser violada em aplicações web modernas por motivos de autenticação, que frequentemente utilizam *cookies* ou sessões gerenciadas no lado do cliente como meio de autenticar usuários, contrariando o modelo proposto por Fielding, uma vez que tornam-se contexto adicional à uma requisição HTTP.

Uma alternativa para o uso de *cookies* através de cabeçalhos HTTP seria a transmissão de *cookies* através de tags através do próprio corpo da resposta HTML, desta maneira se tornando não um contexto adicional à uma requisição, mas uma informação crucial para o entendimento da requisição.

2.3.3 Caching

A restrição de *caching* no estilo REST determina que os recursos de uma resposta de API devem ser explicitamente rotulados como armazenáveis ou não, permitindo que clientes ou *proxies* reutilizem respostas previamente armazenadas sempre que possível (Fielding, 2000). Essa prática reduz a necessidade de requisições repetidas ao servidor, melhora a performance e contribui para a escalabilidade da aplicação (Masse, 2011).

Apesar disso, muitas APIs modernas negligenciam esse aspecto, ignorando cabeçalhos HTTP como *Cache-Control* e *ETag* o que resulta em sistemas menos eficientes e com maior acoplamento (Richardson; Ruby, 2007). A cacheabilidade não é apenas uma otimização, mas uma das restrições centrais que sustentam a arquitetura REST.

2.3.4 Interface Uniforme

A interface uniforme é a característica central que diferencia o estilo REST de outros estilos arquiteturais. Ela estabelece um contrato padronizado entre cliente e servidor,

tornando o sistema mais previsível, reutilizável e delimitado. Essa restrição se desdobra em quatro subcaracterísticas principais:

- **Identificação de recursos:** realizada por meio de *URIs*, que identificam de forma única cada recurso disponível na API. Por exemplo, a informação `/contacts/42` de `https://meuwebsite.com/contacts/42` representa um recurso disponível nesta página da *web*.
- **Manipulação por representações:** os dados são enviados através de representações, por meio de requisições e respostas HTTP, que podem estar acompanhados de metadados que orientam a negociação de formatos de conteúdo entre cliente e servidor, para garantir uma experiência mais fluida do usuário final.
- **Mensagens auto-descritivas:** cada resposta HTTP deve conter todas as informações necessárias para seu processamento, incluindo dados e controles de hipermídia. Isso permite que o cliente interprete a resposta sem necessidade de contexto externo.
- **Hipermídia como motor do estado de aplicações (HATEOAS):** permite a demonstração de informações, de seus estados e das ações disponíveis a partir da sua demonstração gráfica através do uso de hipermídias.

A seguir, apresenta-se um exemplo comparativo entre duas respostas que representam o mesmo recurso:

Resposta em HTML:

```
<html>
  <body>
    <h1>Joe Smith</h1>
    <p>Email: joe@example.bar</p>
    <p>Status: Active</p>
    <a href="/contacts/42/archive">Archive</a>
  </body>
</html>
```

Resposta em JSON:

```
{
  "name": "Joe Smith",
  "email": "joe@example.org",
  "status": "Active"
}
```

Nos exemplos acima, a versão em *HTML* fornece um link explícito para a ação de arquivar o contato — ou seja, apresenta não apenas os dados, mas também uma forma direta de interagir com o recurso, por meio de um *link*. Já a versão em *JSON* apenas retorna os dados, sem indicar quais ações podem ser realizadas. Nesse caso, o cliente

precisa conhecer previamente os *endpoints*, isto é, os caminhos da API que representam operações específicas, como `PATCH /contacts/42/archive`, que poderia exemplificar a ação de arquivar um contato no contexto da aplicação.

2.3.5 Sistema em Camadas

O estilo REST permite que a arquitetura seja dividida em camadas hierárquicas, nas quais diferentes servidores intermediários podem ser introduzidos entre cliente e servidor final. Isso possibilita, por exemplo, o uso de *proxies*, balanceadores de carga, *gateways* e *caches* distribuídos, aumentando a escalabilidade e a segurança do sistema.

2.3.6 Código Sob Demanda (Opcional)

Permite que o servidor envie *scripts* ao cliente. Essa característica é opcional, diferentemente de todas as outras.

Segundo Fielding, essas restrições devem ser aplicadas em conjunto. As APIs que utilizam apenas HTTP e JSON, mas não seguem as demais restrições, especialmente o HATEOAS, não podem ser consideradas verdadeiramente *RESTful* (Fielding, 2000; Richardson; Ruby, 2007; Masse, 2011).

2.3.7 Síntese das restrições e relação com a hipótese

A análise das seis restrições fundamentais do estilo REST evidencia que muitas das implementações atuais de APIs web, embora se apresentem como “*RESTful*”, ignoram ou violam parte significativa dessas diretrizes. O uso de sessões no servidor, a ausência de estratégias consistentes de *cache* e, principalmente, a falta de hipermídia como mecanismo de descoberta de ações e transições de estado apontam um distanciamento em relação à proposta original descrita por Fielding (Fielding, 2000). Como discutido por Richardson e Ruby (Richardson; Ruby, 2007), assim como por Masse (Masse, 2011), grande parte das APIs modernas utiliza apenas HTTP com respostas em JSON, deixando de contemplar aspectos como *caching*, hipermídia e mensagens auto-descritivas. Frequentemente, a máquina do cliente precisa conhecer previamente os *endpoints* disponíveis, interpretar os dados manualmente e decidir como renderizá-los, aumentando o acoplamento entre as camadas. Essa dissociação entre teoria e prática reforça a hipótese central deste trabalho: abordagens orientadas à hipermídia — como o HATEOAS, especialmente quando aplicadas por meio de respostas em HTML — podem representar uma alternativa mais aderente ao modelo REST original e, em determinados contextos, mais eficiente do que APIs baseadas exclusivamente em JSON.

2.4 JSON: origem, popularização e limitações

O formato JSON foi criado por Douglas Crockford no início dos anos 2000 como uma forma leve, legível e eficiente de estruturar dados, especialmente para aplicações baseadas em *JavaScript* (Crockford, 2006). Com sintaxe inspirada em objetos da linguagem *JavaScript*, o JSON rapidamente se tornou um dos formatos mais utilizados na comunicação entre cliente e servidor.

Inicialmente utilizado de maneira informal, o JSON foi posteriormente padronizado como `application/json` pela IETF no RFC 4627, e mais tarde atualizado nos RFCs 7158, 7159 e 8259 (Internet Engineering Task Force, 2025). Sua simplicidade, facilidade de *parsing* e compatibilidade com linguagens de programação modernas fizeram com que se tornasse o formato dominante em APIs *web*, substituindo gradualmente o XML em muitos contextos (TwoBitHistory, 2017).

A popularização do JSON coincidiu com o crescimento dos *frameworks JavaScript* como *React*, *Vue.js* e *Angular*, que trabalham naturalmente com esse tipo de estrutura de dados. Com isso, APIs REST passaram a retornar majoritariamente dados em formato *JSON*, consolidando o que muitos autores chamam de “*JSON-over-HTTP*” como padrão de fato (Masse, 2011).

No entanto, esse modelo simplificado apresenta algumas limitações quando analisado à luz das restrições do estilo arquitetural REST proposto por Fielding (Fielding, 2000). Por exemplo, uma resposta em JSON geralmente contém apenas os dados solicitados, sem controles ou metadados que orientem o cliente sobre ações possíveis. Isso viola a restrição de hipermídia como motor do estado de aplicações, reduzindo o sistema a uma troca de dados sem navegação orientada.

Para ilustrar essa limitação, considere o seguinte exemplo de resposta JSON:

```
{
  "name": "Joe Smith",
  "email": "joe@example.org",
  "status": "Active"
}
```

Essa estrutura fornece os dados de um recurso, mas não orienta o cliente sobre como interagir com ele (por exemplo, arquivar ou atualizar um contato). O cliente precisa conhecer previamente as rotas e ações disponíveis, criando um forte acoplamento entre *front-end* e *back-end*.

Em resposta a essas limitações, surgiram tentativas de padronização como o **JSON:API**, que busca incluir *links*, metadados e convenções para tornar as APIs mais previsíveis e auto-descritivas (JSON:API, 2024). No entanto, mesmo com esses avanços, o JSON continua exigindo que o cliente interprete os dados e decida como apresentá-

los ou manipulá-los, diferentemente de respostas HTML que já incorporam os controles diretamente (como *links*, botões e formulários).

Portanto, embora o JSON tenha méritos evidentes em termos de desempenho e compatibilidade com aplicações modernas, seu uso isolado limita a aplicação plena dos princípios REST, especialmente no que diz respeito à *auto-discovery* (descoberta automática de recursos e transições de estado) e à presença de hipermídia nas respostas. O conceito de *auto-discovery* refere-se à capacidade do cliente de navegar pela aplicação sem conhecimento prévio da estrutura de rotas ou dos caminhos disponíveis, baseando-se apenas nas informações fornecidas dinamicamente pela própria API. Esse aspecto é central na motivação deste trabalho, que busca justamente comparar abordagens baseadas em JSON com outras que utilizam HTML e hipermídia de forma mais aderente às especificações originais do estilo REST.

2.5 Hipermídia como Motor de Estado da Aplicação (HATEOAS)

A restrição conhecida como HATEOAS é uma das mais frequentemente ignoradas em implementações modernas de APIs REST. Trata-se de um princípio definido por Fielding (Fielding, 2000), segundo o qual a interação do cliente com a aplicação deve ocorrer exclusivamente por meio de hipermídia fornecida nas respostas do servidor.

Ou seja, o cliente não deve conhecer previamente a estrutura da API ou os *end-points* disponíveis. Em vez disso, ele deve descobrir dinamicamente os recursos e ações possíveis através de links e controles de navegação contidos nas próprias respostas. Essa abordagem visa reduzir o acoplamento entre cliente e servidor e facilitar a evolução da API sem impactar negativamente os consumidores.

Considere os dois exemplos a seguir:

Resposta em HTML com hipermídia:

```
<html>
  <body>
    <h1>Joe Smith</h1>
    <div>Email: joe@example.bar</div>
    <div>Status: Active</div>
    <a href="/contacts/42/archive">Archive</a>
  </body>
</html>
```

Resposta equivalente em JSON:

```
{
  "name": "Joe Smith",
```

```

    "email": "joe@example.org",
    "status": "Active"
  }

```

Na resposta HTML, o cliente já recebe um controle de ação (o link para arquivar), sem necessidade de lógica extra para descobrir esse caminho. Já na resposta JSON, não há qualquer informação sobre quais ações são possíveis — o cliente precisa conhecer antecipadamente a estrutura da aplicação, o que viola diretamente a proposta do HATEOAS.

Além disso, o uso de hipermídia permite representar diferentes estados do mesmo recurso com ações correspondentes. Por exemplo, ao alterar o status do contato de “Active” para “Archived”, a resposta *HTML* pode incluir um novo link para “Unarchive”, refletindo as possibilidades atuais de transição de estado:

```

<html>
  <body>
    <h1>Joe Smith</h1>
    <div>Email: joe@example.bar</div>
    <div>Status: Archived</div>
    <a href="/contacts/42/unarchive">Unarchive</a>
  </body>
</html>

```

Com isso, o cliente permanece genérico e pode navegar a aplicação sem lógica específica codificada para cada *endpoint*.

Apesar dessas vantagens, o HATEOAS é amplamente subutilizado. Segundo Gross, Stepinski e Akşimşek (Gross *et al.*, 2024), um dos motivos é a dificuldade percebida de implementação, especialmente quando se utiliza formatos como JSON, que não oferecem suporte nativo à hipermídia. Essa limitação motivou o surgimento de bibliotecas como *HTMX* e *FIXI*, que utilizam HTML como formato de resposta e permitem a construção de interfaces reativas respeitando os princípios da web e do REST.

A adoção de hipermídia como motor do estado representa, portanto, uma forma de tornar as APIs mais alinhadas à arquitetura ideal da web, promovendo interoperabilidade, *autodiscovery*, e redução de acoplamento. No contexto deste trabalho, essa abordagem será comparada à tradicional troca de dados via JSON, com o objetivo de avaliar seu desempenho em relação à performance, através do uso de ferramentas de teste de performance, bem como com uma avaliação da viabilidade prática.

3 METODOLOGIA

3.1 Classificação da Pesquisa

Esta pesquisa se enquadra como um estudo de natureza **aplicada**, uma vez que objetiva propor e validar soluções práticas no contexto do desenvolvimento de APIs para aplicações *web*. Quanto aos seus objetivos, classifica-se como uma pesquisa **descritiva e explicativa**: descritiva porque apresenta as características técnicas das abordagens analisadas, e explicativa por buscar compreender os impactos de cada abordagem sobre a estrutura, desempenho e aderência aos princípios arquiteturais da *web*.

Em relação à abordagem metodológica, esta pesquisa adota o **método quantitativo**, pois se apoia em medições objetivas de desempenho do sistema. Quanto aos procedimentos técnicos, trata-se de uma **pesquisa bibliográfica e experimental**. A fundamentação teórica foi construída por meio da análise de literatura especializada, enquanto a etapa prática envolve a implementação e avaliação de duas soluções contrastantes para o mesmo conjunto de funcionalidades.

Estas classificações são feitas de acordo com o que é proposto por Gil (Gil, 2002), o qual desenvolve que pesquisas explicativas possuem como objetivo principal o entendimento dos fatores que constituem ou formam os fenômenos estudados. Ele também discorre sobre as pesquisas descritivas, que procuram estabelecer as explicações entre as relações entre variáveis.

3.2 Procedimentos Metodológicos

A metodologia adotada compreende quatro etapas principais: (i) levantamento bibliográfico, (ii) análise do funcionamento dos *frameworks* escolhidos, (iii) desenvolvimento das provas de conceito e (iv) análise comparativa.

Na primeira etapa, foi realizada uma revisão de literatura utilizando livros técnicos, artigos científicos, publicações acadêmicas, RFCs (Request for Comments) e documentação oficial de ferramentas e frameworks, com foco nos temas centrais do trabalho: estilo arquitetural REST, JSON, HTML e hipermídia.

Na segunda etapa, será feita uma análise dos *frameworks* e bibliotecas utilizadas para o consumo de informações de APIs, neste caso o **Vue.js** e **HTMX** com o objetivo de compreender o seu funcionamento interno e sua estrutura.

Na terceira etapa, serão desenvolvidas duas **provas de conceito** independentes, implementadas com o mesmo conjunto de funcionalidades, a fim de garantir isonomia nos testes. Ambas simulam uma rede social simplificada, com recursos como cadastro de usuários, autenticação, criação de postagens, visualização de perfis e interações básicas.

Na quarta etapa, cada solução será submetida a testes de desempenho e análise

crítica, de modo a comparar suas características sob critérios técnicos e arquiteturais definidos previamente.

3.3 Provas de Conceito

A primeira implementação utilizará uma API que retorna dados em **formato JSON**, consumida por um cliente baseado no *framework* **Vue.js**. Essa abordagem segue o modelo tradicional de APIs REST atuais, no qual o cliente interpreta os dados e gerencia a navegação com base em lógica implementada no *front-end*.

A segunda implementação utilizará uma API que retorna respostas em **formato HTML**, seguindo a proposta do **HATEOAS**. Nesse caso, o cliente será implementado com a biblioteca **HTMX**, que permite renderizar diretamente o HTML recebido via JavaScript e XML assíncronos (AJAX), respeitando os princípios da *web* e o modelo REST original.

Ambas as aplicações terão as mesmas funcionalidades e estrutura de dados, a fim de garantir uma comparação justa entre as abordagens.

3.4 Critérios de Avaliação

As duas implementações serão avaliadas com base nos seguintes critérios:

- **Viabilidade prática de uso:** As provas de conceitos terão sua viabilidade prática avaliados, no que diz respeito à arquitetura do sistema.
- **Requisições por segundo:** Tempo de resposta médio, número de requisições por segundo e quantidade de dados transferidos;
- **Renderização das páginas pelo cliente:** Tempo para a renderização do elemento Largest Contentful Paint (LCP), tempo de renderização do elemento LCP com simulação de limitação de conectividade e processamento;

3.5 Ferramentas Utilizadas

- **Back-end:** Rust com Ntex para o desenvolvimento de servidor HTTP, Serde para a criação serialização de informações em JSON, Askama para a serialização de informações em HTML, banco de dados PostgreSQL;
- **Front-end (JSON):** Vue.js 3 com Composition API e Fetch API para consumo de APIs REST;;
- **Front-end (HTML/HATEOAS):** HTMX com templates pré-renderizados no lado do servidor e atualização dinâmica via atributos HTML;
- **Testes de desempenho:** WRK, e Chrome DevTools para simulação de carga, tempo de resposta e testes de estresse;
- **Ambiente e versionamento:** Git e GitHub para versionamento.

3.6 Cronograma de Atividades

O Quadro 5 apresenta o cronograma de execução previsto para o projeto, considerando as atividades do segundo semestre de 2025.

Tabela 1 – Cronograma de Atividades

Atividade	Mês
Análise do código fonte do Vue.js e HTMX	Ago
Elaboração dos requisitos e especificações técnicas	Ago
Desenvolvimento das provas de conceito (JSON + HTML)	Set-Out
Testes de desempenho e análise comparativa	Nov
Redação final e conclusão do trabalho	Nov-Dez

4 DESENVOLVIMENTO

Neste capítulo será feita a explicação de todo o processo de desenvolvimento, conforme proposto no capítulo anterior, para demonstrar as intenções do autor acerca das provas de conceito propostas. Todos os códigos apresentados neste capítulo estarão disponíveis em formato de repositório Git, localizado na URL <<https://github.com/EduardoStarZ/hyrbet>>, permitindo a análise, execução, modificação e testagem do programa. Todos os códigos apresentados neste trabalho são cópias dos códigos apresentados neste repositório, ora idênticas ao material original, outrora formatados para melhor leitura deste trabalho, mas em momento nenhum sem alterar a compreensão ou o funcionamento ideal do código.

4.1 Motivações para o desenvolvimento

A proposta de criar uma prova de conceito de um aplicativo simples de rede social, que foi apresentada no capítulo anterior, tem um objetivo simples em vista: comparar ativamente o desempenho de uma aplicação que consome hipermídia como formato de resposta com o desempenho de outra aplicação que consome JSON como formato de resposta. A escolha específica para um aplicativo de rede social foi feita pelo fato de que aplicações que tem este mesmo propósito utilizam de vários tipos de mídia diferentes e possuem diversos controles de hipermídia, permitindo assim uma testagem que de fato utilize os vários recursos propostos no modelo REST.

4.2 Sobre o sistema

O sistema desenvolvido foi prontamente nomeado como Hyrbet pelo autor e desenvolvedor. O nome é uma sigla, que quando destrinchada pode possuir um de dois significados:

- ***HY**permedia and **Restfull BE**nchmarking **Th**esis*
- ***HY**permedia and **Restless BE**nchmarking **Th**esis*

O sistema foi desenvolvido com as mesmas tecnologias propostas no capítulo de metodologia: A linguagem Rust foi utilizada para a composição total do *back-end*, com o uso do banco de dados PostgreSQL, e a biblioteca HTMX e o *framework front-end* Vue foram utilizados para o desenvolvimento dos diferentes *front-ends* da aplicação.

4.3 A arquitetura do sistema

A partir da proposta de desenvolvimento das provas de conceito, algumas propostas diferentes para a organização do sistema foram realizadas pelo autor.

4.3.1 Sistema duplamente monolítico

Primeiramente, o sistema seria composto por uma única aplicação monolítica para cada prova de conceito, resultando assim em dois servidores HTTP. Esta ideia foi rejeitada pois isso significaria que ambas as provas de conceito, com a exceção do *front-end* da aplicação, seriam em grande parte uma repetição de código sem qualquer tipo de necessidade.

4.3.2 Sistema unicamente monolítico

Em seguida, foi considerada a proposta de um único sistema monolítico integrado, que continha internamente as funcionalidades propostas para as duas provas de conceito, funcionando como apenas um único servidor HTTP. Essa abordagem foi descartada a partir da previsão do autor de que haveria uma grande complexidade de separação de código dentro do repositório.

4.3.3 Sistema dividido em Microserviços

Por fim, a última proposta de sistema foi a separação dos sistemas em microserviços independentes para cada papel diferente da aplicação, o que permitia uma melhor separação de papéis, mas a organização dos microserviços em um mono repositório, isto é, todos os microserviços seriam organizados em um único repositório. Essa abordagem foi escolhida porque permitia uma reutilização de código robusta para ambas as provas de conceito, ao mesmo tempo que garantia uma organização de diretórios e arquivos.

4.3.4 A separação de papéis

A separação dos papéis dos microserviços foi decidida a partir das diferentes funcionalidades que o sistema precisaria exercer. As funcionalidades imputadas pelo autor como cruciais foram: Autenticação de usuários, distribuição de arquivos estáticos, fornecimento de dados, manipulação do banco de dados, base comum de funcionalidades, o *front-end* utilizando-se do Vue e o *front-end* utilizando-se do HTMX.

A partir destas funcionalidades, uma separação de papéis foi proposta pelo autor para a criação dos servidores:

- Servidor de Autenticação: Responsável pela criação e autenticação de usuários e distribuição de *cookies* que permitem a autenticação de usuários mesmo após o término da sessão do usuário no navegador. Este servidor também é responsável pela manipulação de uma instância de banco de dados que contém os dados dos usuários.
- Servidor de arquivos estáticos: Responsável pelo armazenamento e distribuição de arquivos estáticos para os demais servidores, garantindo a entrega destes arquivos

mesmo com as restrições impostas por navegadores acerca do CORS (*Cross Origin Resource Sharing*), através de cabeçalhos HTTP.

- Servidor de API: Responsável pelo banco de dados principal do sistema, que guarda os dados sobre postagens. Esse servidor também é responsável por transmitir as respostas para as requisições HTTP, e é o servidor responsável por fornecer o front-end em HTMX.
- Servidor do Vue: Responsável apenas pelo front-end em Vue.

A partir desta proposta também, o repositório foi separado em mono repositórios que contém funcionalidades diferentes, muito similar à separação dos servidores, porém contém alguns mono repositórios a mais que a quantidade de servidores. Estes mono repositórios que vão além dos servidores são:

- *Hyrebet* (nomeado como **src**), é onde se encontra o binário que inicializa todos os outros servidores.
- *Common*, é o repositório que contém todo o código que é utilizado por mais de um servidor.
- *Vue-launcher* é o repositório que contém a interface necessária para inicializar o servidor do Vue pelo binário do *Hyrebet*.

Com a definição desta estrutura de projeto, concluí-se a seção de arquitetura do sistema.

4.4 A inicialização do sistema

A inicialização do sistema a partir do binário *hyrebet* inicia os servidores e garante que, durante uma saída normal do programa via o atalho apertando as teclas 'control' e 'C', os programas vão se encerrar de maneira programada e segura. Quando os servidores são inicializados, todos recebem uma porta específica dentro do endereço de IP (Internet Protocol) '127.0.0.1':

- O servidor de autenticação recebe a porta 3000.
- O servidor da API recebe a porta 4000.
- O servidor de arquivos estáticos recebe a porta 8080.
- O servidor que fornece o Vue recebe a porta 5000.

Após isso, cada um dos servidores se inicializa de maneira diferente. Abaixo pode se verificar a instanciação do servidor de autenticação por exemplo:

```
#[ntex::main]
async fn main() -> std::io::Result<()> {

    match env::get_hash() {
        Some(_) => (),
        None => set_hash(hash::create_hash())
    }
}
```

```

};

let options : Server = Server::from(
    ServerID::Auth,
    Ipv4Addr::new(127, 0, 0, 1), 3000);

let server = web::HttpServer::new(move || {
    web::App::new()
        .wrap(CheckLogin)
        .service(
            web::scope("/json")
                .service(routes::login_json)
                .service(routes::register_json)
                .service(routes::check_json)
            )
        .service(routes::login)
        .service(routes::register)
        .service(routes::login_redirect)
        .service(routes::register_redirect)
        .service(routes::logout)
        .state(options.clone())
    });

server.bind((options.ip, options.port)).run().await
}

```

Nesta definição, o servidor verifica se o valor do *hash* de autenticação utilizado pelo servidor de autenticação é definido. Caso esse valor seja definido, nada acontece, mas caso não seja, o valor do *hash* é definido com uma nova chamada para a função **set_hash()**. Em seguida é criada uma variável **options** que cria define os valores que o servidor vai possuir, neste caso os valores de IP e da porta do endereço, e por fim, é feita a criação do servidor em si, que começa a escutar as rotas definidas em **login_json()**, **register_json()** e **check_json()** no escopo da rota *'/json'*, e as rotas contidas em **login()**, **register()**, **login_redirect()**, **register_redirect()** e **logout()**. Ao final, o servidor é iniciado com o uso do método **bind()**.

Com esse exemplo em vista, é possível entender como funcionam as inicializações de todos os servidores baseados em Rust. Após a inicialização dos servidores, todos começam a esperar por requisições HTTP prontamente. Como a inicialização de todos os servidores é extremamente similar, e não afeta a performance dos servidores para os testes que serão realizados, não há a necessidade de mostrar a inicialização de cada um dos

servidores.

```
impl<S, Err> Service<WebRequest<Err>> for CheckLoginMiddleware<S>
where
  S: Service<WebRequest<Err>, Response = WebResponse, Error = Error>,
  Err: ErrorRenderer,
{
  type Response = WebResponse;
  type Error = Error;

  ntex::forward_ready!(service);
  ntex::forward_shutdown!(service);

  async fn call(
    &self,
    req: WebRequest<Err>,
    ctx: ServiceCtx<'_, Self>,
  ) -> Result<Self::Response, Self::Error> {

    let cookie = req.cookie("Auth");

    println!("{}", req.uri());

    if cookie.is_some() && session::check_token_val(
      &LoginToken::Value(
        cookie
        .unwrap()
        .value()
        .to_string()
      )
    ) {
      ctx.call(&self.service, req).await
    } else {
      match req.path() {
        "/register" |
        "/login" |
        "/json/login" |
        "/json/register" => {
          ctx.call(&self.service, req).await
        },
      }
    }
  }
}
```

```

_ => {
  Ok(req.into_response(
    HttpResponse::PermanentRedirect()
      .header(http::header::LOCATION, "/login")
      .finish()
      .into_body(),
  ))
}
}
}
}
}
}

```

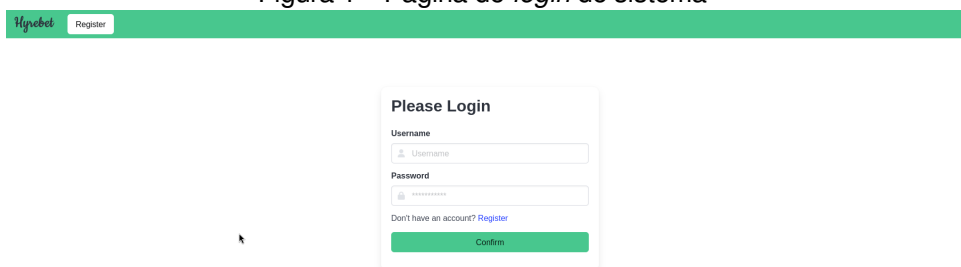
Uma coisa que é necessária mostrar e que está presente nos servidores de autenticação e de API é a funcionalidade do *middleware* de autenticação. Este middleware é responsável por verificar cada uma das requisições passadas ao servidor e verificar se o requisitante possui um *cookie* de autenticação válido, redirecionando ele para a autenticação de usuário caso não possua, a não ser que este já esteja em uma rota de autenticação.

Com a execução deste *middleware* para todas as requisições, haverá um impacto negativo na performance do servidor, porém isto não irá afetar os resultados dos testes de modo a enviesar o resultado, pois esse impacto na performance será igualmente aplicado à ambos os *front-ends*.

4.4.1 As rotas

Neste trabalho não serão exploradas em detalhe as funcionalidades das rotas do servidor, mas serão analisadas algumas diferenças sistemáticas entre as rotas que alimentam ambos os *front-ends* para comparação.

Abaixo serão apresentadas imagens de todas as rotas que ambos os *front-ends* podem acessar, cada um a sua própria maneira.

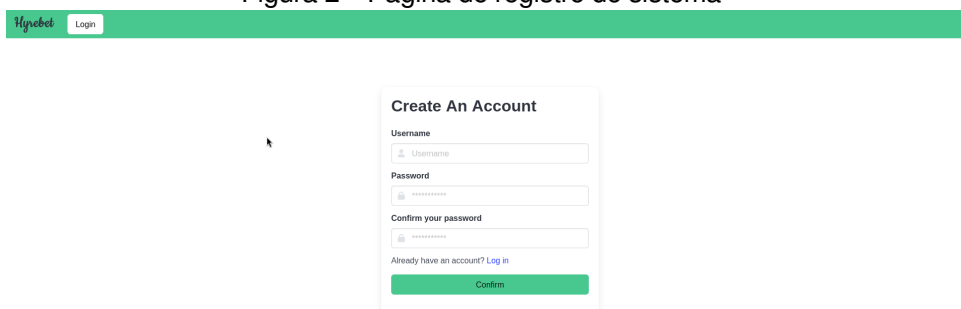
Figura 1 – Página de *login* do sistema

The screenshot shows the login page of the Hypebet system. At the top, there is a green header bar with the 'Hypebet' logo on the left and a 'Register' button on the right. The main content area is white and features a 'Please Login' form. The form has two input fields: 'Username' and 'Password', both with icons of a person and a lock respectively. Below the password field, there is a link that says 'Don't have an account? Register'. At the bottom of the form is a green 'Confirm' button.

Fonte: Criado pelo autor

Na imagem acima é possível observar a página de *login* do sistema, com um campo para um nome de usuário e outro para uma senha.

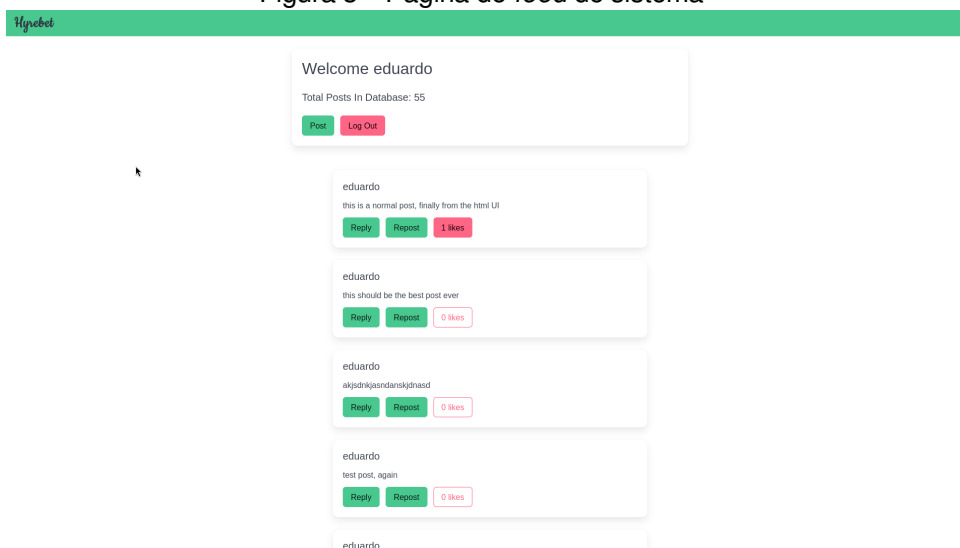
Figura 2 – Página de registro do sistema



The screenshot shows the registration page of the Hypebet system. At the top, there is a green header bar with the 'Hypebet' logo on the left and a 'Login' button on the right. The main content area is white and features a 'Create An Account' form. The form has three input fields: 'Username', 'Password', and 'Confirm your password', each with an icon of a person, a lock, and a lock respectively. Below the 'Confirm your password' field, there is a link that says 'Already have an account? Log in'. At the bottom of the form is a green 'Confirm' button.

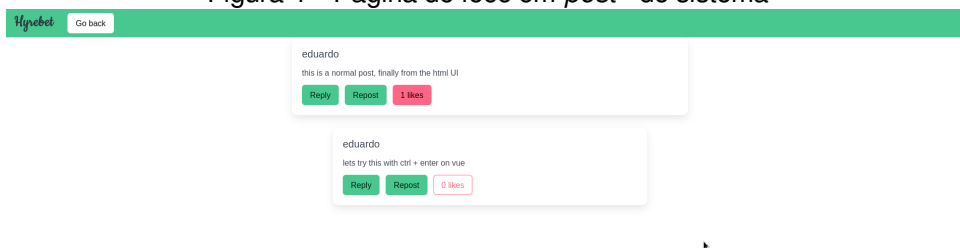
Fonte: Criado pelo autor

Nesta imagem, é observável a página de registro. Similar à página de *login*, porém possui dois campos de senha, sendo um deles para verificar se ambos os campos de senha contém a exata mesma senha.

Figura 3 – Página de *feed* do sistema

Fonte: Criado pelo autor

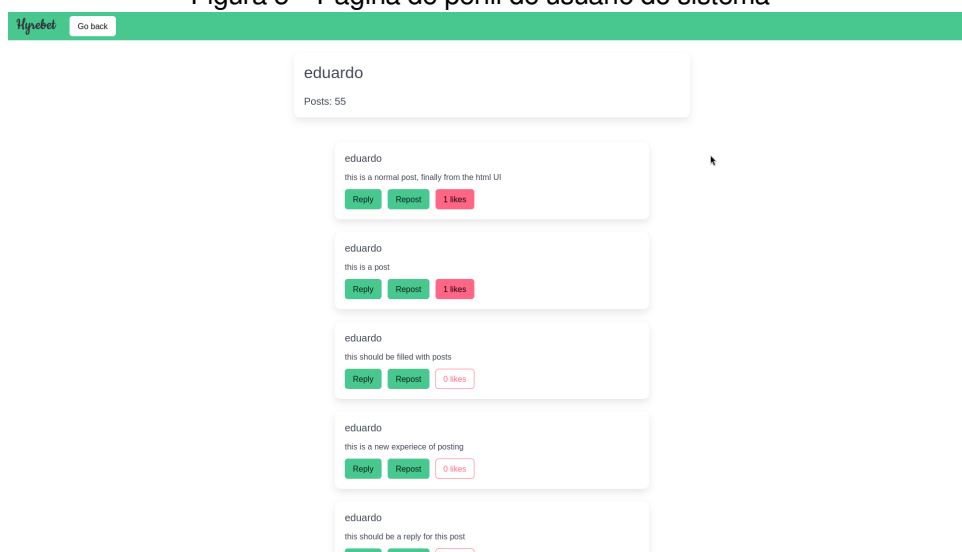
Nesta página, serão renderizados diversos *posts*, de maneira completamente aleatória e de diversos usuários. Existem também os *reposts*, que apresentam o conteúdo do *post* original que está sendo repostado dentro de uma caixa. É possível observar que os *posts* possuem 3 controles de hipermídia: um botão de resposta, um botão de *repost* e um botão de curtir a postagem. Este modelo de *post* é um padrão no sistema e se encontra em outras páginas padronizadamente.

Figura 4 – Página de foco em *post* do sistema

Fonte: Criado pelo autor

A imagem acima mostra uma página que oferece uma visão em foco do *post* que é selecionado pelo usuário. Nele, há um foco maior na postagem selecionada, bem como a mostragem das respostas para aquele determinado *post*.

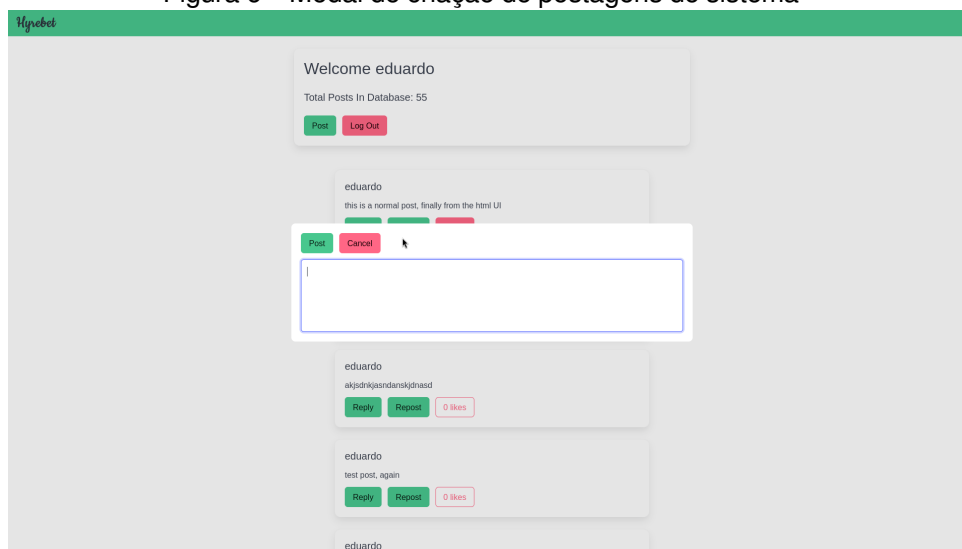
Figura 5 – Página de perfil de usuário do sistema



Fonte: Criado pelo autor

Nesta página, é renderizado todos os *posts* que um usuário já fez, incluindo um valor de informação com o número de postagens total do usuário e o nome do usuário cujo perfil está sendo visualizado.

Figura 6 – Modal de criação de postagens do sistema



Fonte: Criado pelo autor

Por fim, nesta imagem é mostrada o modal de postagens do sistema. Este modal está presente nas últimas 3 páginas mostradas, mas sempre está em um estado invisível por padrão, sendo apenas visível quando o usuário utiliza algum dos controles de hipermídia disponíveis para realizar uma postagem ou repostagem.

4.5 Diferenças entre as rotas

Abaixo, serão mostradas algumas diferenças internas entre as rotas que entregam o dado em um formato HTML e as rotas que entregam o dado em um formato JSON e a rota que consome o formato JSON.

Primeiramente, serão analisadas duas rotas diferentes, uma de *login*, e outra de perfil de usuário, para mostrar as diferenças que evidenciam um possível problema a ser enfrentado durante a execução dos testes de desempenho. A escolha destas duas rotas como exemplo se deve ao fato de que a rota de *login*, é um exemplo de página que não contém qualquer tipo de informação que possa alterar a apresentação da página em nenhuma das aplicações, mas a página de perfil é composta quase exclusivamente por componentes reativos.

```
#[web::get("/login")]
pub async fn login() -> web::HttpResponse {
    #[derive(Template, Deserialize)]
    #[template(path= "auth/login.html")]
    struct LoginTemplate;

    let template : String = match LoginTemplate.render() {
        Ok(value) => value,
        Err(_) => return web::HttpResponse::InternalServerError().finish()
    };

    return web::HttpResponse::Ok().body(template);
}
```

Esta é a rota da API que retorna a página de *login* para o usuário. Nela é definida a função que cria um *template* da página HTML contida no arquivo 'login.html', renderiza este *template* para um formato de *string* e retorna esse valor em formato de uma resposta HTTP.

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="stylesheet"
,→ href="https://cdn.jsdelivr.net/npm/bulma@1.0.4/css/bulma.min.css" />
<link rel="stylesheet" href="http://127.0.0.1:8080/css/main.css" />
```

```

<link rel="stylesheet"
,→ href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.5.2/css/all.min.css"
,→ />
<link rel="preconnect" href="https://fonts.googleapis.com">
<link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
<link href="https://fonts.googleapis.com/css2?family=Cookie&display=swap"
,→ rel="stylesheet">
<link rel="apple-touch-icon" sizes="180x180"
,→ href="http://127.0.0.1:8080/apple-touch-icon.png">
<link rel="icon" type="image/png" sizes="32x32"
,→ href="http://127.0.0.1:8080/favicon-32x32.png">
<link rel="icon" type="image/png" sizes="16x16"
,→ href="http://127.0.0.1:8080/favicon-16x16.png">
<link rel="manifest" href="http://127.0.0.1:8080/site.webmanifest">
<title>Login</title>
<script src="http://cdn.rawgit.com/h2non/jsHashes/master/hashes.js"
,→ defer></script>
<script src="http://127.0.0.1:8080/js/login.js" defer></script>
</head>
<body>
<nav id="navbar" class="navbar" role="navigation" aria-label="main
,→ navigation">
<h1 class="title is-3 mx-4 my-2 logo">Hyrebet</h1>

<div class="navbar-menu">
<div class="navbar-start">
<div class="navbar-item">
<div id="left-buttons" class="buttons">
<a class="button is-white" href="/register">Register</a>
</div>
<div>
</div>
</div>
</div>
</nav>

<section class="hero is-fullbright">
<div class="hero-body">
<div class="container">
<div class="columns is-centered">

```

```

<div class="column is-6-tablet is-5-desktop is-4-widescreen">
<form id="form" class="box my-6" action="http://127.0.0.1:3000/login"
,→ method="post">
<h1 class="title">Please Login</h1>

<div class="field">

<label for="username" class="label mt-5 mx-auto">Username</label>

<div class="control has-icons-left">
<input type="text" class="input" id="username" name="username"
,→ placeholder="Username" required/>
<span class="icon is-small is-left">
<i class="fa fa-user"></i>
</span>
</div>
</div>

<div class="field">

<label for="password" class="label mx-auto">Password</label>

<div class="control has-icons-left">
<input type="password" class="input" id="password" name="password"
,→ placeholder="*****" required/>
<span class="icon is-small is-left">
<i class="fa fa-lock"></i>
</span>
</div>
</div>

<p>Don't have an account? <a class="link"
,→ href="/register">Register</a></p>

<div class="field buttons my-4">
<input type="submit" class="button is-success is-fullwidth" id="confirm"
,→ value="Confirm" />
</div>

```

```

</form>
</div>
</div>
</div>
</div>
</section>
</body>
</html>

```

Neste código está o exemplo do que a página de *login* retorna para o usuário como resposta. Uma estrutura similar é oferecida quando o usuário faz a mesma requisição, mas no servidor do Vue. A única diferença é a utilização de *scripting* para gerenciar o envio de formulários.

```

<script setup lang="ts">
  import { ref } from 'vue'
  import NavbarComponent from './NavbarComponent.vue'
  import { useRouter } from 'vue-router'

  const formData = ref({
    username: "",
    password: ""
  });

  const router = useRouter();

  const handleSubmit = async () => {
    try {
      const response = await fetch('/auth/json/login', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json'
        },
        body: JSON.stringify(formData.value)
      });

      if (response.ok) {
        router.push("/home");
      }
    }
  }

```

```

    }

    formData.value = { name: "", email: "" };
  } catch (error) {
    console.error('Error submitting form:', error);
  }
}
</script>

<template>
<NavbarComponent>
<RouterLink class="button is-white" to="/">Register</RouterLink>
</NavbarComponent>

<section class="hero is-fullbright">
<div class="hero-body">
<div class="container">
<div class="columns is-centered">

<div class="column is-6-tablet is-5-desktop is-4-widescreen">

<form class="box my-6" @submit.prevent="handleSubmit">
<h1 class="title">Please Login</h1>

<div class="field">

<label for="username" class="label mt-5 mx-auto">Username</label>

<div class="control has-icons-left">
<input type="text" class="input" v-model="formData.username"
,→ placeholder="Username" required/>
<span class="icon is-small is-left">
<i class="fa fa-user"></i>
</span>
</div>
</div>

<div class="field">

```



```

let treated_posts : Vec<PostWrapper> = all_posts
.iter()
.map(|post| PostWrapper::new(&post))
.collect::<Vec<PostWrapper>>();

let body : String = ProfileTemplate{
    username: path.clone(),
    total_posts: treated_posts.len() as i32,
    posts: PostTemplate {
        posts: treated_posts
    }.render()
}.unwrap()
}.render().unwrap();

let template : String = Base {
    title: format!("Profile - {} ", &path),
    scripts: None, body
}.render().unwrap();

return web::HttpResponse::Ok().body(template);
}

```

Neste trecho de código, é feita uma *query* no banco de dados que procura por todos os *posts* do usuário solicitado a partir da rota, e formata estes *posts* para permitir uma melhor exibição. Em seguida, os posts são renderizados e a página final é renderizada de acordo, sendo retornada como resposta pela função.

```

<div class="columns is-centered">
<div class="column is-6-tablet is-5-desktop is-5-widescreen my-5">
<div class="box">
<h1 class="subtitle is-3">{{ username }}</h1>
<p class="subtitle is-5">Posts: {{ total_posts }}</p>
</div>
</div>
</div>

<div id="htmx-target">
{{ posts|safe }}
</div>

```

Este é o *template* de perfil que é utilizado para renderizar um perfil de usuário. É possível ver que há um espaço para a inserção de *posts* no *template*. Este espaço é renderizado de acordo com outro *template*, o de *posts*.

```
{% for post in posts %}
```

```
<div class="columns is-centered">
<div class="column is-6-tablet is-5-desktop is-4-widescreen">
<div class="box post-box" whereto="{{ post.owner }}/{{ post.id
,→ }}?fullview=true">
<div class="field">
<a class="subtitle is-5 nopropagate"
,→ href="{{post.owner}}">{{post.owner}}</a>
</div>
```

```
<div class="field">
<p class="text">{{post.contents}}</p>
</div>
```

```
{% match post.repost %}
```

```
{% when Some with (repost) %}
```

```
<div class="box post-box nopropagate" whereto="{{ repost.owner }}/{{
,→ repost.id }}?fullview=true">
```

```
<div class="field">
<a class="subtitle is-5 nopropagate" href="{{ repost.owner }}" >
{{ repost.owner }} •
<span hx-get="/like/{{repost.owner}}/{{repost.id}}?number=true"
,→ hx-target="this" hx-trigger="every 30s">
{{ repost.total_likes }}
</span>
likes
</a>
</div>
```

```
<div class="field">
<p class="text nopropagate">{{ repost.contents }}</p>
</div>
</div>
```

```
{% when None %}
```

```
{% endmatch %}
```

```
<div class="field buttons">
```

```
<button class="button is-success reply nopropagate" value="{{ post.owner
```

```
, → }}/{{ post.id }}">Reply</button>
```

```
<button class="button is-success repost nopropagate" value="{{ post.owner
```

```
, → }}/{{ post.id }}">Repost</button>
```

```
<div class="control has-icons-left">
```

```
<div hx-get="/like/{{ post.owner }}/{{ post.id }}" hx-target="this"
```

```
, → hx-trigger="load" hx-swap="outerHTML"></div>
```

```
</div>
```

```
</div>
```

```
</div>
```

```
</div>
```

```
</div>
```

```
{% endfor %}
```

Neste *template* é feita a renderização de uma postagem, que é injetada no *template* anterior de perfil. Isso conclui o entendimento da renderização de uma resposta para o cliente de HTMX. Agora será mostrado a renderização de uma resposta para o cliente de Vue.

```
#[web::get("/{user}")]
```

```
pub async fn profile_json(path : web::types::Path<String>) -> web::HttpResponse {
```

```
    #[derive(Serialize)]
```

```
    struct Response {
```

```
        username : String,
```

```
        total_posts: i32,
```

```
        posts: Vec<PostWrapper>
```

```
    }
```

```
    let all_posts : Vec<Post> = match database::get_all_posts_from_user(&path) {
```

```
        Some(value) => value,
```

```
        None => return web::HttpResponse::NoContent().finish()
```

```
    };
```

```

let treated_posts: Vec<PostWrapper> = all_posts.iter().map(|post| PostWrapper::new(&post)).

let total_posts : i32 = all_posts.len() as i32;

let response : Response = Response {
    posts: treated_posts,
    username: path.to_string(),
    total_posts
};

return web::HttpResponse::Ok().json(&response);
}

```

Neste trecho de código há a mesma funcionalidade de recolher as postagens de um usuário e transformá-las em resposta, mas desta vez em um formato JSON. Durante o desenvolvimento, o autor preferiu por tratar de requisições diferentes para tipos de dados diferentes para não enviesar o tempo de resposta de um *front-end* ou outro, uma vez que a mesma função estaria fornecendo a resposta para os dois clientes ao mesmo tempo.

```

<script setup lang="ts">
  import {onMounted, reactive, ref} from 'vue'
  import PostComponent from './PostComponent.vue'
  import NavbarComponent from './NavbarComponent.vue'
  import { useRoute, useRouter } from 'vue-router'

  const route = useRoute();

  const router = useRouter();

  const formData = ref({
    contents: ""
  });

  const bodyData = reactive({
    username : "",
    total_posts : "",
    posts : []
  });

  onMounted( async () => {

```

```

const response = await fetch(`/api/json/${route.params.user}`, {
  method: 'GET',
  headers : {"Content-Type": "application/json"},
  credentials: 'include'
});

const data = await response.json();

bodyData.username = data.username;
bodyData.total_posts = data.total_posts;
bodyData.posts = data.posts;
});

const location = ref("");

const dialog = ref(null);

const openDialog = (data) => {
  location.value = data;

  dialog.value.showModal() ;
}

const closeDialog = () => {
  dialog.value.close();
}

const submit = async () => {
  const response = await fetch(location.value, {
    method: 'POST',
    headers : {"Content-Type": "application/json"},
    credentials: 'include',
    body: JSON.stringify(formData.value)
  });

  closeDialog();

  router.go(0);

```

```

    }
  </script>
  <template>
    <NavbarComponent>
    <button class="button is-white" @click="$router.back()">Go Back</button>
    </NavbarComponent>
    <div class="columns is-centered">
    <div class="column is-6-tablet is-5-desktop is-5-widescreen my-5">
    <div class="box">
    <h1 class="subtitle is-3">{{ bodyData.username }}</h1>
    <p class="subtitle is-5">Posts: {{ bodyData.total_posts }}</p>
    </div>
    </div>
    </div>

    <div>
    <PostComponent v-for="data in bodyData.posts" :post="data"
    ,→ :key="'${data.owner}/${data.id}'" @child-reply="openDialog"
    ,→ @child-repost="openDialog"/>
    </div>

    <dialog ref="dialog">
    <form @submit.prevent="submit">
    <div class="field buttons">
    <input type="submit" class="button is-success" value="Post" />
    <input type="button" class="button is-danger" @click="closeDialog"
    ,→ value="Cancel" />
    </div>

    <div class="control">
    <textarea class="textarea is-normal has-fixed-size"
    ,→ @keyup.ctrl.enter="submit" v-model="formData.contents" rows="5"
    ,→ cols="80" maxlength="240" required autofocus></textarea>
    </div>
    </form>
    </dialog>
  </template>
  <style scoped></style>

```

Neste trecho há o *template* em Vue que será renderizado para uma resposta

HTML. Como mencionado anteriormente, no capítulo de fundamentação deste trabalho, este *template* utiliza o *scripting* para conseguir renderizar a resposta baseada em JSON, uma vez que um navegador não sabe como interpretar essa informação contida na resposta.

Com base nestes dois exemplos é possível notar alguns padrões que irão surgir em todas as outras rotas, para as duas aplicações. Na aplicação que consome HTML como formato de dados, haverá uma forte dependência de lógica por parte do servidor, uma vez que a resposta fornecida pelo servidor chega completamente renderizada.

Por outro lado, na aplicação que consome JSON como formato de dados, haverá uma forte dependência de lógica no lado do cliente para renderizar as informações, já que as informações repassadas chegam apenas no formato JSON, precisando ser renderizados pelo cliente. Além disso, durante os testes de performance do sistema, o Vue irá trazer um problema maior. Quando uma requisição é feita para o Vue, o que o servidor do Vue retorna inicialmente é apenas o *template* inicial com os *scripts* necessários para renderizar a informação e iniciar uma página HTML. Isso significa que, na prática, para medir o tempo de resposta de qualquer rota na aplicação do Vue será necessário adicionar o tempo de resposta de todas as rotas inclusas no processo.

Com essa demonstração, é feita a conclusão das diferenças entre rotas, e será explicado em seguida o modelo dos testes de performance das provas de conceito.

4.6 O modelo da testagem

Para definir os parâmetros e a lógica da testagem, é necessário considerar o problema apresentado no capítulo anterior, onde a testagem de uma rota de uma página em Vue precisaria da testagem de todas as rotas que compõe a renderização inicial desta. Na testagem, também foi prevista o teste de estresse das aplicações, mas não foi elaborado a maneira em que elas seriam estressadas.

A partir do uso da ferramenta *Dev Tools* do navegador Chrome, serão utilizadas as funcionalidades de simulação de conexão de baixa velocidade e de processamento lento, para simular dispositivos que possuem uma taxa de processamento menor que o sistema que irá testá-las. Por fim, a testagem de requisições por segundo será feita utilizando a ferramenta WRK, para verificar os números de requisições por segundo para cada uma das rotas.

4.7 Os testes

A implementação dos testes será feita executando o servidor e o cliente na mesma máquina local, para garantir a menor latência possível durante a testagem, o que garante que o tempo de resposta para uma requisição se daria integralmente pelo tempo em que o servidor toma para receber a requisição, processá-la e responde-la.

Para a testagem utilizando as ferramentas do *Chrome Dev Tools*, será feita uma testagem manual com o navegador e o servidor sendo executados na mesma máquina localmente. Para cada rota que possui uma interface no *front-end*, será realizado manualmente uma coleta de dados do LCP (*Largest Contentfull Paint*), métrica que estabelece qual elemento HTML que possui conteúdo demorou mais para ser renderizado e em quanto tempo este elemento foi renderizado.

A coleta de dados do LCP será feita da seguinte maneira: a partir do carregamento inicial da página, será iniciada a ferramenta do *Dev Tools* e então será feito o recarregamento da página 5 vezes. Para cada uma destas 5 vezes, será anotado o valor do LCP e serão calculados os valores estatísticos necessários.

Para a testagem utilizando o WRK, será criado um *script* em Bash que executa automaticamente as testagens e as coleta para um arquivo após o término da execução do programa. A execução será feita em um ambiente sem nenhuma interface gráfica para garantir o máximo desempenho do servidor e do cliente, garantindo resultados mais sólidos.

4.7.1 O script WRK

Abaixo está o *script* para a testagem utilizando o WRK:

```
#!/bin/sh
```

```
# Login Page
```

```
wrk -c 10 -d 5m -t 4 -H --latency http://127.0.0.1:4000/login >  
,→ test-results/5m-login-rf.txt
```

```
wrk -c 10 -d 5m -t 4 -H --latency http://127.0.0.1:5000/login >  
,→ test-results/5m-login-rl.txt
```

```
# Register Page
```

```
wrk -c 10 -d 5m -t 4 -H --latency http://127.0.0.1:4000/register >  
,→ test-results/5m-register-rf.txt
```

```
wrk -c 10 -d 5m -t 4 -H --latency http://127.0.0.1:5000/ >  
,→ test-results/5m-register-rl.txt
```

```
# Home Page
```

```
wrk -c 10 -d 5m -t 4 -H "Cookie: <cookie>" --latency  
,→ http://127.0.0.1:4000/ > test-results/5m-home-rf.txt
```

```
wrk -c 10 -d 5m -t 4 -H "Cookie: <cookie>" --latency  
,→ http://127.0.0.1:5000/home > test-results/5m-home-rl.txt
```

```
wrk -c 10 -d 5m -t 4 -H "Cookie: <cookie>" --latency  
,→ http://127.0.0.1:4000/json/ > test-results/5m-home-json-rl.txt
```

Profile Page

```
wrk -c 10 -d 5m -t 4 -H "Cookie: <cookie>" --latency  
,→ http://127.0.0.1:4000/eduardo > test-results/5m-profile-rf.txt
```

```
wrk -c 10 -d 5m -t 4 -H "Cookie: <cookie>" --latency  
,→ http://127.0.0.1:5000/eduardo > test-results/5m-profile-rl.txt
```

```
wrk -c 10 -d 5m -t 4 -H "Cookie: <cookie>" --latency  
,→ http://127.0.0.1:4000/json/eduardo >  
,→ test-results/5m-profile-json-rl.txt
```

Post Page

```
wrk -c 10 -d 5m -t 4 -H "Cookie: <cookie>" --latency  
,→ http://127.0.0.1:4000/eduardo/104924087?fullview=true >  
,→ test-results/5m-post-rf.txt
```

```
wrk -c 10 -d 5m -t 4 -H "Cookie: <cookie>" --latency  
,→ http://127.0.0.1:5000/eduardo/104924087 > test-results/5m-post-rl.txt
```

```
wrk -c 10 -d 5m -t 4 -H "Cookie: <cookie>" --latency  
,→ http://127.0.0.1:4000/json/eduardo/104924087 >  
,→ test-results/5m-post-json-rl.txt
```

Acima no *script*, há o uso de um *cookie* utilizado para autenticar o usuário, garantindo que a resposta do servidor durante a testagem seja a resposta devida, e não o redirecionamento do usuário para a página de *login*.

5 RESULTADOS E DISCUSSÃO

Nesta seção do trabalho, serão apresentados todos os dados coletados a partir dos testes propostos no capítulo de metodologia do projeto e que tiveram seu escopo definido no capítulo de desenvolvimento. Será também feita uma argumentação com as problemáticas levantadas acerca das rotas que acessam o *front-end* em Vue, bem como a apresentação de uma solução para suportar a confirmação da hipótese com dados válidos.

5.1 Resultados dos Testes

Após a realização de todos os testes previstos no escopo deste trabalho, os dados foram coletados e tratados de modo padronizado. Os valores foram inseridos em formatos de tabelas, valores com mais que 3 casas decimais fracionárias tiveram seus valores encurtados para melhor visualização dos dados no documento. Todos os valores sem nenhuma formatação podem ser encontrados dentro dos arquivos que foram utilizados para a coleta de dados dentro do mesmo repositório onde se encontra o código do sistema desenvolvido neste trabalho, na URL <<https://github.com/EduardoStarZ/hyrebet>>, dentro do diretório '*test-results*'.

5.1.1 Dos parâmetros de testagem

Todos os testes apresentados abaixo foram realizados como foram propostos no capítulo de desenvolvimento do trabalho. Para os testes envolvendo a métrica de LCP, foi realizado a testagem utilizando a versão oficial do navegador Chrome, com apenas a instância do Chrome aberta e a execução paralela do servidor. Para os testes envolvendo o uso de WRK, os testes foram realizados em instâncias do TTY (Teletypewriter) sem nenhuma interface gráfica.

Os testes foram realizados em um ambiente utilizando a distribuição Arch Linux, com a versão 6.17.9 do *Kernel Zen*, utilizando uma sessão do *X Server* manipulada pelo gerenciador de janelas *I3* para os testes de LCP. Este ambiente, enquanto em espera utiliza cerca de 600 MB de memória RAM consistentemente. Os testes utilizando o WRK foram realizados utilizando a versão 4.2.0 da ferramenta, e os testes com o *Dev Tools* foram realizados com a versão 142.0.7444.175 do navegador Chrome fornecido pelo Arch User Repository.

A máquina onde os testes foram executados possui um processador Intel 4570s com 3.600 GHz (Giga-Hertz), Memória RAM de 8 GB com 1600 MT/s (Mega-Transfers) de frequência e GPU (Graphics Processing Unit) ATI Radeon HD 6570.

Qualquer execução dos testes propostos em diferentes ambientes, sistemas operacionais, e/ou com componentes de *hardware* diferentes ou até mesmo iguais dos utilizados

nestes testes podem trazer resultados diferentes dos apresentados neste trabalho.

5.1.2 Requisições com WRK

Tabela 2 – Resultados WRK (4 *Threads*, 10 conexões paralelas, 5 minutos)

	Dados Totais	Valor máximo	Desvio Padrão	Valor Médio	Nº Requisições
Login	61,55GB	18,35ms	557,41us	292,87us	22.866.732
Login (JSON)	621,67MB	146,37ms	1,77ms	4,33ms	559.065
Register	68,36GB	14,00ms	569,74us	292,45us	22.181.935
Register (JSON)	0,98GB	16,44ms	372,11us	2,66ms	902.522
Home	1,48GB	321,13ms	18,59ms	65,96ms	36.408
Home (JSON)	586,48MB	25,34ms	497,85us	4,55ms	527.419
Home (JSON API)	223,14MB	465,45ms	28,63ms	79,34ms	30.431
Profile	1,85GB	273,76ms	15,35ms	57,60ms	41.696
Profile (JSON)	618,61MB	143,79ms	1,77ms	4,35ms	556.311
Profile (JSON API)	286,28MB	377,65ms	22,43ms	67,80ms	35.559
Post	0,86GB	266,04ms	11,53ms	43,10ms	55.752
Post (JSON)	661,01MB	28,93ms	491,55us	4,04ms	594.446
Post (JSON API)	112,80MB	464,50ms	27,45ms	57,81ms	42.350

Acima estão os resultados da testagem do WRK com o *script* apresentado no capítulo de desenvolvimento do projeto. Com base nas informações deste quadro é possível inferir diversas coisas acerca do funcionamento do sistema e a motivação para alguns dos resultados. As testagens foram feitas utilizando 4 *threads* do sistema e 10 conexões paralelas, com 5 minutos de testagem por rota.

Primeiramente, é necessário destacar que de todas as rotas que são apresentadas nos resultados, apenas as rotas de *login* e de registro no *front-end* em HTMX, e todas as

rotas do *front-end* em Vue (mas não as rotas de API que o Vue consome), não utilizam nenhuma manipulação de banco de dados e nenhuma apresentação de componentes reativos é realizada.

Isso foi devidamente mostrado na apresentação de algumas rotas do sistema no capítulo de desenvolvimento, e é evidenciado com os resultados do teste.

Com os resultados das rotas de *login* e registro, é possível verificar que há uma grande disparidade de resultados. Enquanto as rotas do HTMX conseguiram responder em média 22 milhões e 400 mil requisições, as rotas do Vue responderam uma média de 700 mil requisições.

É possível perceber também que o *back-end* em Rust torna as respostas da aplicação mais estáveis também, trazendo um valor máximo de resposta de 16 milissegundos em média, com um desvio padrão com um valor de 560 microssegundos em média nas duas rotas, enquanto o servidor do Vue traz um valor máximo de resposta entre 70 milissegundos, com um desvio padrão do tempo de resposta das requisições de 700 microssegundos em média, indicando que esse tempo de resposta é estável para a aplicação do Vue.

Com essas informações em mente, será feita a análise das rotas que renderizam componentes reativos, e que fazem uso do banco de dados do sistema. Primeiramente, há a rota para a página principal do sistema. Nesta rota o *front-end* em HTMX respondeu apenas 36 mil requisições. Em contraste com as 22 milhões respondidas nas rotas de *login* e registro, isso evidencia que o uso do banco de dados do sistema diminui o processamento de requisições drasticamente.

Enquanto isso, o *front-end* em Vue conseguiu responder uma quantidade similar de requisições, com um número de 527 mil requisições respondidas. Como relatado no capítulo de desenvolvimento, isso se deve ao fato de que o *front-end* do Vue envia apenas um simples *template* com um *script* maior para a renderização dos componentes, o que diria que o Vue consegue responder mais requisições por segundo em templates reativos que o HTMX. Porém, é necessário lembrar que o Vue necessita de realizar outras requisições em todas as suas rotas com valores reativos para o *back-end*, com o objetivo de receber os valores em formato JSON.

Com isso em mente, as rotas do *back-end* que providenciam os dados em formato JSON foram adicionados aos testes. Em relação à página principal do sistema, o *front-end* em Vue precisa realizar uma chamada para a API em JSON da página principal, que quando testada, realizou um valor inferior de respostas para as requisições quando em comparação com o número de respostas que o *front-end* em HTMX fez.

Como mostrado no capítulo de desenvolvimento, uma rota que retorna JSON realiza as mesmas operações de banco de dados que uma rota que retorna HTML no sistema, porém é feita também a transformação dos valores requisitados para o formato JSON através do uso da biblioteca Serde. Isso pode significar dizer que a codificação dos dados para o formato JSON pode ser mais lenta que a renderização dos *templates* com

dados em HTML.

Com estes dados em mente, é possível inferir que para a renderização de uma página por completo, cada rota no servidor do Vue necessita realizar pelo menos 2 requisições, uma para o servidor do Vue, e outra para o servidor do *back-end* do sistema. Com base nessa inferência, é possível também dizer que mesmo que a rota em Vue consiga responder mais requisições que a rota em HTMX, para a renderização completa da página seria necessária a inclusão da requisição para a rota do *back-end*, o que significa dizer que o número real de requisições respondidas pelo servidor é limitado pela rota com o menor número de requisições.

Desta maneira, na rota da página principal, o número máximo de requisições que a aplicação conseguiria responder na prática, seria o valor de 30 mil requisições, e não o de 520 mil requisições. É possível notar com as rotas que acessam o banco de dados que há um grande desvio padrão das requisições e um grande valor no tempo máximo de resposta alcançado em uma requisição, todos nas casas das dezenas ou centenas de milissegundos. Isso reforça a tese de que operações com bancos de dados no sistema são muito custosas no geral.

Em seguida, na rota de perfil do sistema, o *front-end* em HTMX conseguiu responder 41 mil requisições, enquanto o *front-end* em Vue respondeu 556 mil requisições, mas é limitado pelo número de 35 mil requisições da rota que fornece o JSON para ela.

Logo após, por fim, em uma rota de um *post* específico, o *front-end* em HTMX respondeu 55 mil requisições, enquanto o *front-end* em Vue respondeu 594 mil requisições, mas foi limitado pelo número de 42 mil requisições da rota que a fornece JSON.

Com esses números, é possível chegar as seguintes conclusões, com base nos dados estatísticos, assim como foi indagado anteriormente:

- A manipulação do banco de dados consome mais recursos que a transformação dos dados para um *template* em HTML ou para uma codificação em JSON.
- A codificação para JSON é mais lenta que a codificação de *templates* HTML.
- O desvio padrão da latência de requisições é consistente em ambos os sistemas de *front-end*.
- Apesar de não ser uma métrica totalmente plausível, a soma do tempo máximo de uma resposta entre as rotas que fazem parte do *front-end* em Vue e as rotas que fornecem os dados para este *front-end* na maioria das vezes vai resultar em um valor superior do que nas rotas do *front-end* em HTMX. A única exceção notável é a rota principal em HTMX que possui quase 6 vezes o número de tempo máximo de resposta. Isso pode ocorrer devido ao fato de que dentro do *template* da página principal existem vários estados em que as informações podem ser apresentadas e existem pelo menos um componente filho para cada componente renderizado. Esta métrica não é totalmente plausível pois o tempo máximo de requisição pode variar de acordo com a testagem feita, e a soma desses valores não é muito viável, uma vez

que a ocorrência do valor máximo de uma rota pode não ser a mesma em que ocorre o tempo máximo de resposta na rota em que os dados são enviados.

5.1.3 Renderização do elemento mais lento

Tabela 3 – Resultados do LCP por rota (Sem Estresse) em segundos

	Valor mínimo	Valor máximo	Desvio Padrão	Valor Médio
Login	0,18	0,20	0,008	0,194
Login (JSON)	0,35	0,47	0,041	0,416
Register	0,26	0,28	0,008	0,194
Register (JSON)	0,38	0,42	0,013	0,396
Home	0,23	0,31	0,026	0,272
Home (JSON)	0,42	0,58	0,066	0,48
Profile	0,30	0,32	0,007	0,312
Profile (JSON)	0,39	0,68	0,117	0,5
Post	0,26	0,42	0,063	0,324
Post (JSON)	0,40	0,44	0,013	0,422

Acima está o quadro que representa os valores de LCP (*Largest Contentfull Paint*) de um elemento. Esta métrica diz respeito ao elemento com algum conteúdo, como texto, imagem ou vídeo, que demorou mais tempo para ser renderizado por uma página HTML.

Diferentemente do teste realizado com o WRK, este teste não necessita a testagem das rotas que oferecem os dados em JSON, uma vez que o teste utiliza um navegador que renderiza o conteúdo por completo, o que significa que as requisições para as rotas que oferecem esses dados estão inclusas nas métricas.

Primeiramente, é possível verificar que, nesta testagem, nenhum valor chega a casa de um segundo inteiro para renderizar o LCP. Em seguida, é possível observar que o desvio padrão do valor do LCP para as quase todas as rotas do *front-end* em HTMX são menores que as do *front-end* em Vue, o que indica uma maior consistência por parte do HTMX. A única rota em que o desvio padrão foi maior no *front-end* HTMX é a rota de um *post* específico. Porém todos os valores mínimos, máximos e médios de LCP no *front-end* em HTMX foram menores do que os do *front-end* em Vue.

Essa primeira testagem começa a confirmar a hipótese deste trabalho de que a renderização no lado do cliente traz um impacto negativo na performance para o cliente.

Tabela 4 – Resultados do LCP por rota (Estresse - Processamento Lento 4x) em segundos

	Valor mínimo	Valor máximo	Desvio Padrão	Valor Médio
Login	0,29	0,50	0,072	0,416
Login (JSON)	0,94	0,98	0,016	0,956
Register	0,47	0,51	0,016	0,494
Register (JSON)	0,95	1,11	0,054	1,026
Home	0,47	0,57	0,032	0,512
Home (JSON)	1,03	2,12	0,492	1,51
Profile	0,50	0,65	0,063	0,554
Profile (JSON)	1,00	1,14	0,050	1,056
Post	0,42	0,58	0,070	0,48
Post (JSON)	0,93	1,10	0,061	1,018

Em seguida na testagem foi realizada o teste de LCP com as mesmas rotas, porém utilizando o recurso de simulação de limitação de CPU (Central Processing Unit), que foi utilizada para limitar a simulação para um ambiente com o processamento quatro vezes mais lento que o normal.

Neste cenário, os valores de desvio padrão para as rotas diminuíram quando em comparação com o teste normal, porém, os valores mínimos, máximos e médios de todas as rotas aumentaram significativamente. Desta vez a diferença destes valores entre o *front-end* em HTMX e *front-end* em Vue aumentaram robustamente, tendo uma disparidade de pelo menos o dobro em quase todas as rotas.

A demonstração deste cenário reforça a hipótese de que a renderização do lado do cliente dos dados traz impactos negativos significativos para o tempo de renderização de uma página HTML.

Tabela 5 – Resultados do LCP por rota (Estresse - Internet 4G) em segundos

	Valor mínimo	Valor máximo	Desvio Padrão	Valor Médio
Login	0,48	0,52	0,016	0,488
Login (JSON)	2,84	2,91	0,025	2,878
Register	0,56	0,62	0,022	0,59
Register (JSON)	3,02	3,20	0,067	3,122
Home	0,63	0,73	0,036	0,68
Home (JSON)	3,06	3,71	0,232	3,252
Profile	0,61	0,86	0,084	0,7
Profile (JSON)	3,22	3,54	0,132	3,342
Post	0,57	0,66	0,032	0,61
Post (JSON)	3,08	3,18	0,036	3,15

Por fim, neste último quadro, são apresentados os dados que mostram a testagem que simula o ambiente com uma restrição de conexão, utilizando uma conexão de celular de 4G ao invés da conexão normal do primeiro teste.

Nesta testagem, todos os valores de desvio padrão do *front-end* em HTMX são inferiores aos do *front-end* em Vue. É possível notar também que a disparidade dos valores do tempo mínimo, máximo e médio de resposta para todas as rotas entre os dois *front-ends* aumentou expressivamente, com os valores de respostas de requisições do *front-end* em HTMX podendo ser até 6 vezes menor que os valores do *front-end* em Vue.

Estes dados mostram que a dependência no uso de mais requisições para uma renderização inicial da página traz impactos muito negativos para a renderização de uma página, pois quando há a limitação no tempo de resposta de uma das rotas, todas as outras são prejudicadas, independentemente do *throughput* que aquela rota suporta.

5.1.4 A viabilidade dos sistemas

A partir do desenvolvimento do sistemas mencionados como provas de conceito, foi possível compreender a arquitetura que os sistemas precisariam tomar para realizar as atividades propostas por este trabalho.

Apesar de, por natureza, ser descrita como um valor qualitativo, a arquitetura de um sistema pode ser representada como um valor quantitativo, a partir da representação do sistema de acordo com os seus componentes, que é como este trabalho almeja representá-los a seguir.

Como ambas as provas de conceito utilizam os três microsserviços que foram desenvolvidos para o *back-end* das aplicações em conjunto, ele será considerado como

apenas um único sistema, a fim de não trazer uma complicação analítica para esta seção do trabalho.

Para que o sistema em Vue fosse viável, foi necessário a operacionalização do seu próprio servidor, onde foram definidas as estruturas e a lógica que o *front-end* usaria durante a testagem. A partir desta testagem, houve uma necessidade de utilizar outro servidor, desta vez fornecendo o *back-end* com as informações necessárias para a execução do sistema, como autenticação, arquivos estáticos e os dados da rede social. Em contraste, o servidor em HTMX foi viabilizado apenas com o mesmo servidor utilizado como *back-end* para o Vue, uma vez que ele estava sendo distribuído diretamente pelo *back-end* do sistema.

Isso significa que, para o desenvolvimento em Vue, há a necessidade do uso de pelo menos 2 servidores, um para o fornecimento do *front-end* e outro para o fornecimento dos dados que compõem os componentes neste determinado *front-end*, enquanto para o desenvolvimento utilizando o HTMX, há a necessidade do uso de pelo menos 1 servidor que fornece o *front-end* diretamente.

5.2 Uma breve discussão sobre os resultados

É possível realizar uma testagem deste porte, sem em nenhum momento compreender o que as ferramentas escolhidas fazem durante os testes realizados. Isso não impede, porém, de que seja feita uma análise mais rebuscada dos efeitos destas no resultado final. Desta maneira, nesta seção, será avaliado como o funcionamento do *framework* Vue e da biblioteca HTMX podem ter impactado nos resultados dos testes.

5.2.1 O impacto do Vue

Devido a sua estrutura, que utiliza dois documentos, um para o cálculo das mudanças que a interação do usuário causa nos componentes e outro que apresenta os estados destas mudanças, boa parte da performance que o Vue necessita é utilizada no processamento destes dois *templates* e na comunicação entre eles.

5.2.2 O impacto do HTMX

Enquanto isso, o HTMX, utiliza apenas um documento onde as alterações são feitas automaticamente. Porém há o uso de uma área especial no documento utilizando única e exclusivamente para o armazenamento temporário de respostas a serem inseridas no documento HTML. O maior gasto de desempenho do HTMX vem com o processamento dos elementos que possuem alguma ação HTMX, inserindo escutas de eventos para estes elementos com as ações que eles são responsáveis por executar.

6 CONSIDERAÇÕES FINAIS

Este trabalho, tanto no meio acadêmico quanto no contexto de desenvolvimento de aplicações para *web*, vem com uma importância significativa, podendo impactar tanto sistemas já existentes quanto a criação de sistemas futuros, de tal ponto a remodelar até mesmo a maneira como algumas técnicas são lecionadas no geral. Ele traz contribuições diretas para a área do desenvolvimento, para as áreas de pesquisa relacionadas com o tema, e não menos importante, traz também uma grande contribuição para o pensamento crítico dos desenvolvedores, independentemente das diversas tecnologias utilizadas por estes profissionais.

Neste capítulo, encerra-se a apresentação do trabalho com a indagação acerca dos objetivos e da hipótese estabelecidos na introdução do trabalho, uma discussão acerca do uso de hipermídia como motor de estado de aplicações no trabalho, a apresentação de propostas de práticas de desenvolvimento baseadas nos resultados obtidos neste trabalho, as apresentações das limitações encontradas neste trabalho e a sugestão de trabalhos futuros.

6.1 Dos objetivos

Os objetivos estabelecidos previamente no trabalho almejavam a análise tanto do formato constituinte dos formatos de hipermídia e de transmissão de dados, quanto do desenvolvimento das provas de conceito para a análise de performance dos modelos de desenvolvimento, com o estudo da hipótese em vista.

6.1.1 Do objetivo Geral

O objetivo geral estabelecido pretendia realizar uma análise dos formatos de dados HTML e JSON, observando suas propriedades funcionais e diferenças. Com a análise de ambos na fundamentação teórica do projeto, foi possível observar que a estrutura do HTML, em contraste com a estrutura do JSON, apesar de deter menos flexibilidade quanto à transmissão de dados brutos, permite a estruturação de páginas não só com as informações desejadas, mas também com a atribuição automática de controles de hipermídia para os dados.

Enquanto isso, por não possuir estrutura suficiente a construção de uma página para web, requer a transformação de um dado para outro formato que permite essa visualização e permite o uso de controles de hipermídia, deste modo tratando o JSON como um formato mais eficiente para o transporte robusto de dados em contraste com a demonstração mais adequada do formato HTML.

6.1.2 Dos objetivos específicos

Os objetivos específicos delimitados no trabalho previam a análise de um *framework* ou biblioteca de *front-end* que utiliza o JSON como formato de comunicação e outro *framework* ou biblioteca que utiliza o HTML como formato de comunicação, o desenvolvimento de duas provas de conceito de um aplicativo de rede social e a comparação destas provas de conceito em uma análise estatística.

Com estes objetivos estabelecidos, no capítulo de desenvolvimento foi realizado o desenvolvimento das provas de conceito propostas, com as funcionalidades desejadas. Logo após, ainda no desenvolvimento do trabalho foram apresentados os resultados dos testes realizados, bem como uma discussão sobre estes, evidenciando a diferença entre os diferentes sistemas desenvolvidos. Por fim, a análise do *framework front-end* Vue e da biblioteca HTMX foram realizados e encontram-se nos apêndices A e B deste trabalho, devido ao tamanho de seus respectivos textos.

6.2 Da hipótese

Com base em todos estes dados apresentados no capítulo anterior, será feita uma análise da relação dos dados com a hipótese proposta no capítulo de introdução ao trabalho. A hipótese proposta pelo trabalho considera que o uso de hipermídia como formato de resposta para APIs pode trazer os benefícios de desacoplar o cliente do servidor, no sentido em que o cliente se torna principalmente dependente do servidor para uma renderização inicial de uma página para receber os dados necessários, traz também a vantagem de simplificar a lógica de renderização de páginas, uma vez que estas páginas já estariam sendo transmitidas em seu formato final, e por fim traz a melhora da performance em determinados cenários de aplicações.

Essa hipótese será destrinchada ponto por ponto a seguir, para poder confirmar ou refutar os devidos pontos apresentados por ela.

6.2.1 Desacoplamento

O desacoplamento nas aplicações baseadas em Vue vem a partir do uso de uma API em formato JSON que são consumidas após a requisição para uma rota que é alimentada pelo Vue. O uso dessa API faz com que, para todas as páginas que são apresentadas, caso haja a necessidade de visualização de qualquer componente que possua um estado que está armazenado no servidor de uma aplicação, é feita uma segunda requisição para este ou outro servidor que guarda os dados.

No escopo deste trabalho, com o uso do HTMX em uma das provas de conceito, quando uma requisição é realizada para uma rota que possui um valor reativo, a página que é enviada para o cliente possui todos os dados necessários para a renderização inicial, sendo apenas para a atualização destes dados que é necessária a implementação de novas

requisições. Segundo Fielding (Fielding, 2000), com a restrição que diz respeito à falta de estados em aplicações aderentes ao modelo REST, uma aplicação sem estados induz a propriedade de visibilidade uma vez que um cliente não precisa observar para além de uma requisição para determinar o contexto completo de uma resposta.

Apesar de ainda haver um acoplamento com o servidor, este acoplamento é reduzido substancialmente, trazendo mais independência do cliente em relação ao servidor. Desta maneira, é certo afirmar que o emprego correto de hipermídia como o motor de estado de atualizações reduz o acoplamento entre o cliente e servidor, apesar de não eliminá-lo.

6.2.2 A lógica dos sistemas

A simplificação da lógica dos sistemas refere-se ao excesso de lógica aplicada no cliente necessária para a renderização de uma página. Foi mostrado no capítulo de desenvolvimento o formato compõe uma aplicação Vue, e é possível observar que há uma grande quantidade de lógica enviada para o cliente poder realizar a renderização de uma página, além da lógica necessária para o envio dos dados das respectivas páginas.

Além disso, é possível perceber que, caso haja a atualização de um sistema, especificamente no envio de dados que não possuem formatos suficientes para se tornarem hipermídias, além da necessidade de atualizar o *back-end* do sistema, será necessário atualizar o *front-end* de acordo, para que este tenha em seu escopo o novo formato que este determinado dado toma.

Com o emprego do HTMX nas páginas, é possível perceber que, pelo fato dos dados enviados ao *front-end* já serem o formato final a ser encaixado no corpo do documento HTML, caso haja a necessidade de atualizar o formato do dado a ser apresentado, a atualização a ser feita só precisará ser realizada pelo servidor, uma vez que o fornecimento do *front-end* é completamente feito por este.

Além do mais, de acordo com Fielding (Fielding, 2000), a opcionalidade do *scripting* permite que a quantidade de funcionalidades já pré-implementadas em um sistema seja reduzida, implementado-as com o uso de código em demanda. Inerentemente, a partir da opcionalidade de código em demanda, é natural que surjam clientes de servidores que recusam expressamente a execução destas funcionalidade em seus navegadores. Com isso o uso de *tags* HTML, tais como 'noscript' podem ser utilizadas pelo desenvolvedor para transmitir respostas para estes clientes, sem impactar na sua experiência de usuário.

A partir destes pontos, é correto afirmar que o emprego do formato de hipermídia como respostas de servidores traz uma redução na lógica, principalmente na parte que poderia ter sido atribuída ao cliente de um sistema, uma vez que essa lógica é processada pelo servidor de forma mais robusta e simplificada e ainda quando um cliente pode não aceitar a execução de *scripts* no seu navegador.

6.2.3 A melhora da performance

A melhora da performance de aplicações no geral é uma métrica difícil de ser avaliada, uma vez que são diversos os fatores que contribuem para a performance. Como um exemplo, nas provas de conceitos apresentada neste trabalho, foi possível observar que a manipulação do banco de dados do sistema de ambas as aplicações trouxe um impacto negativo para a performance, portanto, este trabalho irá se reservar apenas para medir o impacto do uso de hipermídia como motor de estados na performance de aplicações.

No contexto das provas de conceito aplicadas, foi possível observar um impacto significativamente positivo do uso de HATEOAS na prova de conceito enquanto estava sendo comparada com a aplicação que utilizava uma abordagem de sistema de consumo de APIs JSON. É necessário ponderar, porém, que este desempenho foi medido especificamente apenas neste sistema.

Essa observação é necessária, pois existem sistemas que podem possuir interfaces não convencionais, que podem acabar sendo afetados negativamente pelo uso de hipermídia como formato de resposta.

Ademais, é necessário apontar que, de acordo com o que Fielding (Fielding, 2000) dissertou, tanto o uso de interfaces uniformes quanto a falta de estados em aplicações resultam em impactos negativos na performance de aplicações que aderem ao modelo REST, mas traz suas respectivas vantagens com a melhora da eficiência de transferência de dados de hipermídia e com a consistência e sincronia de estados entre cliente e servidor.

Desta maneira, é correto afirmar que o uso da hipermídia como motor de estado de atualizações, apesar de reduzir a performance em alguns cenários, é benéfica para o desempenho de uma aplicação como um todo e contanto que seja utilizada em componentes e sistemas que trazem informações relativamente simples tanto em conteúdo quanto em formato.

6.3 Uma discussão acerca da hipermídia como motor de estado de atualizações

Com base nos resultados obtidos no trabalho, foi possível observar um impacto positivo para a performance da aplicação proposta. Apesar deste impacto positivo, isso não significa dizer que o uso de hipermídia para o gerenciamento de estados de uma aplicação deva se tornar o único e exclusivo formato de comunicação para aplicações como um todo.

Existem diversos tipos de aplicações para web que cumprem papéis diferentes que, em geral, que precisam de métodos diferentes de comunicação para a obtenção de dados necessários para o seu funcionamento. Algumas ideias, para a visualização destes sistemas, seriam aplicações como o *Youtube*, *X* (antigo *Twitter*) ou o *BlueSky* que utilizam de APIs em diferentes formatos para a transmissão de dados para aplicações externas.

Como as aplicações externas tomam diversos formatos diferentes, algumas até mesmo fora do contexto da web, seria inadequado transmitir estes dados de qualquer

maneira em que os dados não estejam em seu formato bruto, uma vez que qualquer formatação dos dados necessitaria de um pré-processamento dos dados antes do seu uso. Por essas razões, o uso de formatos como o JSON ou quaisquer outros formatos não pode ser simplesmente dispensado, pois ele cumpre bem o papel que lhe é aferido.

O que este trabalho propõem, é que para a criação de interfaces que possuem componentes convencionais, com o uso de textos, imagens, vídeos, hiperlinks e fazem uma utilização de controles destes elementos de hipermídia, seja feito o uso da hipermídia como formato mediados para a transmissão dos dados entre um servidor HTTP e um cliente navegador especificamente.

6.4 Das práticas sugeridas

Nesta seção, serão propostas práticas para o desenvolvimento futuro de aplicações para a web, com base em todas as informações dispostas neste trabalho, bem como as experiências do autor durante o desenvolvimento das provas de conceitos e dos testes realizados.

6.4.1 Envio de JSON ou HTML

Como o JSON não pode ser simplesmente eliminado de sistemas que possuem APIs que precisam transmitir dados brutos, há a necessidade do envio de respostas em ambos os formatos, dependendo das necessidades do cliente que está a acessar o sistema.

Durante o desenvolvimento dos sistemas neste trabalho, foi utilizado uma abordagem com rotas diferentes para os diferentes formatos: uma rota que transmitia o formato HTML possuía uma estrutura como '`<ip>/<recurso>`', enquanto uma rota que transmitia o formato JSON possuía uma estrutura como '`<ip>/json/<recurso>`'. Essa abordagem foi escolhida apenas no contexto deste projeto para permitir que a testagem das provas de conceito não carregasse um viés que impactasse um formato ou outro durante os testes.

Para o desenvolvimento de aplicações no geral, o que é proposto neste trabalho é que seja feito o uso de cabeçalhos HTTP para a negociação do formato a ser distribuído, de modo que uma mesma rota possa transmitir o mesmo recurso de maneiras diferentes. Para melhor visualização desta proposta, será apresentado um exemplo a seguir.

Partindo do exemplo em que um cliente faça uma requisição HTTP para o servidor, caso o cliente envie um cabeçalho HTTP com o valor '`Accept: application/json`', indicando que o formato que ele consegue compreender é um formato JSON, o servidor deve responder a requisição com os dados a serem enviados no formato JSON. Caso o header '`Accept`' seja omitido ou possua um valor de '`text/html`', o servidor deve responder com o formato HTML. É importante observar que caso haja a necessidade, o servidor também pode suportar o envio dos dados requisitados em outros formatos, como XML, TOML (Tom's Obvious Markup Language) e YAML (YAML Ain't Markup Language) por exemplo.

6.4.2 Arquivos estáticos

Quando o envio de arquivos estáticos, como *scripts*, folhas de estilo, imagens, vídeos, fontes e quaisquer outro formato, são necessários para o desenvolvimento de aplicações, algumas considerações devem ser feitas antes de decidir como estas serão distribuídas.

Caso a aplicação não necessite de uma velocidade de resposta muito alta para as requisições, e os arquivos estáticos possam ser carregados de maneira mais lenta, é preferível que o desenvolvedor opte por usar CDNs (Content Delivery Network) para o envio destes arquivos para o cliente final de uma requisição, até onde seja possível. Isto pode, porém, prejudicar o funcionamento de algumas partes da aplicação caso algum dos CDNs utilizados se encontrar em um estado de indisponibilidade ou alto tempo de resposta. O aumento da dependência de um sistema externo deve ser considerado pelo desenvolvedor durante a modelagem do sistema.

Quando a aplicação necessita de uma alta disponibilidade e velocidade de resposta para um serviço crítico, e o servidor consegue responder de acordo com esta necessidade, é preferível que o servidor envie todos os arquivos estáticos necessários para o funcionamento da página. Isso remove a dependência de um sistema externo, mas atribui o papel de fornecimento destes arquivos para o sistema que o desenvolvedor propõe a fazer.

6.4.3 O papel da renderização de componentes

Durante o desenvolvimento, é possível que, com alguma antecedência, o desenvolvedor perceba que um determinado recurso por completo possa tomar mais tempo de renderização que o desejado. Desta maneira, caso seja possível, é preferível que este componente seja renderizado após o corpo principal de um recurso. Como um exemplo, será mostrado o exemplo de uma interface que mostra *posts* de uma rede social, similar à prova de conceito realizada neste trabalho.

Começando pelo exemplo em que esta interface sugerida renderize um total indefinido de *posts* e renderize também, logo abaixo entre uma a duas resposta para cada *post*, caso existam. Algumas técnicas podem ser utilizadas para diminuir o tempo de resposta da rota principal deste exemplo, que ficaria responsável por responder a requisição com todos os *posts*.

Primeiramente, o desenvolvedor poderia utilizar, no caso do HTMX, de eventos como 'hx-trigger=load' para que apenas quando uma página seja carregada, as respostas de um *post* sejam carregadas em seguida. De modo similar, caso o usuário esteja interessado em continuar visualizando os *posts* que vem logo em seguida, é interessante de usar um elemento invisível com um evento similar, 'hx-trigger=revealed', que permite realizar uma requisição para obter mais *posts*, limitando o número de *posts* a serem enviados na requisição inicial, mas não prejudicando o funcionamento do sistema.

6.5 Uma avaliação das ferramentas da coleta de dados

A coleta dos dados realizada acerca dos testes de performance das aplicações propostas recebeu um papel de grande importância neste trabalho. Desta maneira, as escolhas das métricas e dos modelos de testagem receberam uma grande atenção do autor durante o desenvolvimento do trabalho.

A ferramenta WRK, apesar de antiga, mas muito bem atualizada ao longo dos anos, cumpriu muito bem o seu papel de estressar o servidor para realizar requisições e obter os valores já formatados com as métricas propostas. O *Dev Tools* do Chrome, da mesma maneira, por ser uma ferramenta relativamente nova e possuir uma manutenção excelente por parte dos mantenedores do projeto Chromium, demonstrou um padrão muito bom para a testagem de renderização das páginas, apesar de ser necessária a anotação manual da coleta dos dados adquiridos.

6.6 Limitações deste trabalho

Apesar de se tratarem de tópicos expressivos acerca do uso e da modelagem de APIs em formato HTML ou em formato JSON, alguns tópicos não foram explorados neste trabalho, uma vez que excedem o que o autor acredita ser o tema principal deste. Não foram explorados neste trabalho:

- As questões de segurança da informação acerca do uso de diferentes formatos de respostas para APIs.
- A sugestão de novas arquiteturas de modelagem que contemplem o formato JSON na criação de sistemas distribuídos para *web*.
- O uso de hipermídia para aplicações *mobile* e em sistemas com interfaces não-convencionais, uma vez que no livro de sistemas de hipermídia (Gross *et al.*, 2024), já foi prontamente elaborado como a hipermídia pode ser utilizada para o desenvolvimento nestes sistemas.

6.7 Trabalhos futuros

A seguir, estão alguns trabalhos futuros que podem ser desenvolvidos, com base nos resultados obtidos. Estes são, mas não se limitam a:

- A criação de formatos de hipermídia para a transmissão de dados para interfaces não-convencionais.
- As questões de segurança que envolvem o uso do JSON como formato de resposta
- As questões de segurança que envolvem o uso de hipermídia como formato de resposta.
- A criação de novos modelos de testes de estresse e performance de sistemas para *web*.

REFERÊNCIAS

- Amazon Web Services. **Diferença entre REST e SOAP**. 2024. <<https://aws.amazon.com/pt/compare/the-difference-between-soap-rest/>>. Acesso em: jul. 2025.
- CROCKFORD, D. **The application/json Media Type for JavaScript Object Notation (JSON)**. 2006. RFC 4627. Disponível em: <<https://datatracker.ietf.org/doc/html/rfc4627>>.
- FIELDING, R. **REST APIs must be hypertext-driven**. 2008. <<https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>>. Acesso em: jul. 2025.
- FIELDING, R. T. **Architectural Styles and the Design of Network-based Software Architectures**. Tese (Ph.D. Dissertation) — University of California, Irvine, Irvine, 2000. Disponível em: <https://ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf>.
- GIL, A. C. **Como Elaborar Projetos de Pesquisa**. 4. ed. São Paulo: Atlas, 2002.
- GROSS, C.; ADAM, S.; AKSİMEK, D. **Hypermedia Systems**. 2024. <<https://hypermedia.systems/>>. Acesso em: jul. 2025.
- Internet Engineering Task Force. **IETF DataTracker**. 2025. <<https://datatracker.ietf.org/>>. Acesso em: jul. 2025.
- JSON:API. **JSON:API — A specification for building APIs in JSON**. 2024. <<https://jsonapi.org/>>. Disponível em: <<https://jsonapi.org/>>. Acesso em: jul. 2025.
- MASSE, M. **REST API Design Rulebook**. Sebastopol: O'Reilly Media, 2011.
- Postman Team. **Intro to APIs: History of APIs**. 2021. <<https://blog.postman.com/intro-to-apis-history-of-apis/>>. Acesso em: jul. 2025.
- RICHARDSON, L.; RUBY, S. **RESTful Web Services**. Sebastopol: O'Reilly Media, 2007.
- TWOBITHISTORY. **The Rise and Rise of JSON**. 2017. <<https://twobithistory.org/2017/09/21/the-rise-and-rise-of-json.html>>. Acesso em: jul. 2025.

APÊNDICE A – FUNCIONAMENTO INTERNO DO VUE

Neste trabalho, será feita a análise do funcionamento interno do *framework* front-end Vue, com a linguagem JavaScript. O código está disponível em formato de repositório Git na URL <<http://github.com/vuejs/vue>>. Todos os códigos apresentados neste trabalho são cópias dos códigos apresentados neste repositório, ora idênticas ao material original, outrora formatados para melhor leitura deste trabalho, mas em momento nenhum sem alterar a compreensão ou o funcionamento ideal do código.

A.1 Ponto de entrada

Ao abrir o arquivo 'package.json', é possível encontrar algumas informações necessárias para o projeto do Vue, tais como a versão requerida do *pnpm* (8.9.2) e alguns *scripts* que o *npm* irá executar caso invocados.

Para a leitura do código fonte, foi escolhida a versão "v2.7.16" devido ao fato de que esta é a *tag* de versão mais atual e simples de analisar até o período de escrita deste trabalho, em relação a configuração do ambiente para análise.

```
"scripts": {
  "build": "node scripts/build.js",
  "build:ssr": "npm run build -- runtime-cjs,server-renderer",
  "test:ssr": "test:ssr && npm run test:sfc",
  "test:unit": "vitest run test/unit",
  "test:ssr": "npm run build:ssr && vitest run server-renderer",
  "test:sfc": "vitest run compiler-sfc",
  "release": "node scripts/release.js",
  "changelog": "conventional-changelog -p angular -i CHANGELOG.md -s"
}
```

O *script* que importa para chegar ao ponto de entrada é o 'build': **"build": "node scripts/build.js"**, ele aponta diretamente para o arquivo 'build.js' no diretório *scripts*. O propósito deste arquivo é configurar e executar o processo de compilação do código *Typescript* do *Vue* para código *Javascript*. O que nos interessa neste arquivo é a definição do arquivo de configuração.

```
const fs = require('fs')
const path = require('path')
const zlib = require('zlib')
const rollup = require('rollup')
const terser = require('terser')
```

```

if (!fs.existsSync('dist')) {
  fs.mkdirSync('dist')
}

let builds = require('./config').getAllBuilds()

// filter builds via command line arg
if (process.argv[2]) {
  const filters = process.argv[2].split(',')
  builds = builds.filter(b => {
    return filters.some(f => b.output.file.indexOf(f) > -1 ||
      b._name.indexOf(f) > -1)
  })
}

```

let builds = require('./config').getAllBuilds() nos aponta para *scripts/config.js*. Esse arquivo é um grande aglomerado de configurações de compilação do projeto. A parte importante do arquivo é o ponto de entrada para compilação do projeto, no nosso caso, o ponto de entrada é o **full-dev**, que pode ser encontrado na linha 119 do arquivo.

```

'full-dev': {
  entry: resolve('web/entry-runtime-with-compiler.ts'),
  dest: resolve('dist/vue.js'),
  format: 'umd',
  env: 'development',
  alias: { he: './entity-decoder' },
  banner
}

```

O ponto de entrada aponta para o arquivo **src/platforms/web/entry-runtime-with-compiler.ts**. Esse arquivo é apenas um *wrapper* que junta as funcionalidades do *Vue* e exporta elas para compilação.

```

import Vue from './runtime-with-compiler'
import * as vca from 'v3'
import { extend } from 'shared/util'

```

```

extend(Vue, vca)

```

```

import { effect } from 'v3/reactivity/effect'

```

```
Vue.effect = effect
```

```
export default Vue
```

Porém, o que é interessante é a chamada de importação do *Vue* do arquivo “**src/platforms/web/runtime-with-compiler.ts**”.

```
import config from 'core/config'
import { warn, cached } from 'core/util/index'
import { mark, measure } from 'core/util/perf'

import Vue from './runtime/index'
import { query } from './util/index'
import { compileToFunctions } from './compiler/index'
import {
  shouldDecodeNewlines,
  shouldDecodeNewlinesForHref
} from './util/compat'
import type { Component } from 'types/component'
import type { GlobalAPI } from 'types/global-api'
```

Este arquivo importa o *Vue* a partir do arquivo “**src/platforms/web/runtime/index.ts**”, que, por sua vez, importa o *Vue* a partir do arquivo “**src/core/index.ts**”.

```
import Vue from 'core/index'
import config from 'core/config'
import { extend, noop } from 'shared/util'
import { mountComponent } from 'core/instance/lifecycle'
import { devtools, inBrowser } from 'core/util/index'
```

E este, finalmente, importa o *Vue* do arquivo “**src/core/instance/index.ts**”.

```
import Vue from './instance/index'
import { initGlobalAPI } from './global-api/index'
import { isServerRendering } from 'core/util/env'
import { FunctionalRenderContext } from 'core/vdom \
/create-functional-component'
import { version } from 'v3'
```

E então, é demonstrada a implementação original do da inicialização do *Vue* com a função **Vue()**.

```
import { initMixin } from './init'
import { stateMixin } from './state'
import { renderMixin } from './render'
import { eventsMixin } from './events'
import { lifecycleMixin } from './lifecycle'
import { warn } from '../util/index'
import type { GlobalAPI } from 'types/global-api'
```

```
function Vue(options) {
  if (__DEV__ && !(this instanceof Vue)) {
    warn('Vue is a constructor and should be \
      called with the `new` keyword')
  }
  this._init(options)
}
```

```
initMixin(Vue)
stateMixin(Vue)
eventsMixin(Vue)
lifecycleMixin(Vue)
renderMixin(Vue)
```

```
export default Vue as unknown as GlobalAPI
```

Este arquivo de inicialização é um tanto quanto pequeno, pois ele só serve para juntar as partes menores do *Vue* em um único arquivo, e iniciar o projeto usando **this._init(options)**. É possível ver também neste código que tempos a inicialização de alguns *Mixins* a partir da inicialização do *Vue*. Isso se deve a organização da estrutura do *Vue*, cuja qual foi apresentada por todos os arquivos até este ponto. Essa estrutura é formada por camadas, que segue a ordem de mais interna para mais externa a seguir: **Core** -> **Mixins** -> **Platform** -> **Entry**. Foi demonstrado a camada *Entry* ao passar pelo arquivo “**src/platforms/web/entry-runtime-with-compiler.ts**” e “**build/config.js**”, fornece funções de montagem e configurações pro módulo *Core* (é possível ver que existem outras entradas no diretório “**src/platforms/web**”). Em seguida, há a camada **Platform**, que é responsável pelo fornecimento de coisas específicas para cada plataforma para o módulo *Core*, e adiciona *patches* e funções de montagem públicas. Logo depois, há a camada de **Mixins**, que são 5 frações do *Vue*: **Init**, **State**, **Events**, **Lifecycle** e **Render**. Esses *mixins* são responsáveis por algumas funcionalidades centrais do *Vue*. E por último, o módulo **Core**, que é responsável por inicializar o funcionamento do *Vue*.

Essa separação em camadas traz algumas vantagens póstumas para o processo

de manutenção do *Vue*:

- Desacoplação: Camadas diferentes focam em tarefas diferentes.
- Encapsulação: Cada camada lida apenas com si mesma.
- Reutilização: Quanto mais próximo do módulo Core, mais genérico é o código e mais fácil de ser reutilizado ele será.

A.2 Mixin De Inicialização

Voltando a focar no código em “**src/core/instance/index.ts**”, as chamadas para funções e inicialização de *mixins* são importantes! A chamada para a função **initMixin()** vem do arquivo “**src/core/instance/init.ts**”. Ela requer um parâmetro que é um objeto de instância do *Vue*. Esse classe instancia um objeto que possui todas as propriedades centrais do *Vue*.

```
export function initMixin(Vue: typeof Component) {
  Vue.prototype._init = function (options?: Record<string, any>)
```

Esta função serve para inicializar a propriedade **Vue.prototype._init** como uma função que toma como parâmetro um Objeto do tipo **Record<K, T>**. Este objeto é fornecido pela biblioteca padrão do *Typescript*, e é resumidamente uma coleção que guarda chaves de propriedades **K** que possuem apenas um tipo de dado **T**.

```
    const vm: Component = this
    vm._uid = uid++

    let startTag, endTag
    if (__DEV__ && config.performance && mark) {
      startTag = `vue-perf-start:${vm._uid}`
      endTag = `vue-perf-end:${vm._uid}`
      mark(startTag)
    }
  }
```

A criação da constante **vm** instancia o mesmo valor que é passado como parâmetro da função **initMixin()**, com a chamada através de **this**. Então é incrementado o **uid** da constante **vm** uma vez, e são criados as variáveis **startTag** e **endTag**. Na estrutura condicional, é definido a marcação da variável **startTag**.

```
    vm._isVue = true
    vm.__v_skip = true
    vm._scope = new EffectScope(true)
```

É então instanciado os valores de `_isVue`, `__v_skip` e `_scope`, todas propriedades de `vm`.

```
if (options && options._isComponent) {
  initInternalComponent(vm, options as any)
} else {
  vm.$options = mergeOptions(
    resolveConstructorOptions(vm.constructor as any),
    options || {},
    vm
  )
}
```

Este trecho checa a existência de valores na variável **options**, e configura e mescla alguns valores a partir de configurações já existentes.

```
if (__DEV__) {
  initProxy(vm)
} else {
  vm._renderProxy = vm
}
vm._self = vm
```

Aqui, o **proxy** é instanciado. O **proxy** é usado durante o desenvolvimento para mostrar mais informações sobre a renderização. E então a propriedade `_self` é instanciada finalmente.

```
initLifecycle(vm)
initEvents(vm)
initRender(vm)
callHook(vm, 'beforeCreate', undefined, false)
initInjections(vm)
initState(vm)
initProvide(vm)
callHook(vm, 'created')
```

Aqui, existem várias inicializações de *Mixins* e funcionalidades do Vue. O foco será direcionado para 6 delas, **Lifecycle**, **Events**, **Render**, **Injections**, **State** e **Provide**. As outras duas funções são apenas a chamada de *hooks* fornecidos.

```
export function callHook(
  vm: Component,
```

```

hook: string,
args?: any[],
setContext = true
) {
  pushTarget()
  const prevInst = currentInstance
  const prevScope = getCurrentScope()
  setContext && setCurrentInstance(vm)
  const handlers = vm.$options[hook]
  const info = `${hook} hook`
  if (handlers) {
    for (let i = 0, j = handlers.length; i < j; i++) {
      invokeWithErrorHandling(handlers[i], vm, args || null, vm, info)
    }
  }
  if (vm._hasHookEvent) {
    vm.$emit('hook:' + hook)
  }
  if (setContext) {
    setCurrentInstance(prevInst)
    prevScope && prevScope.on()
  }

  popTarget()
}

```

Acima há todo o código da função **callHook()**. Ela requer uma instância das propriedades do *Vue* do tipo **Component**, um **hook** do tipo string, argumentos de qualquer tipo e um booleano para apropriar o contexto, que por padrão, é verdadeiro. A primeira linha chama a função **pushTarget()**.

```

Dep.target = null
const targetStack: Array<DepTarget | null | undefined> = []

export function pushTarget(target?: DepTarget | null) {
  targetStack.push(target)
  Dep.target = target
}

```

Acima está todo o contexto necessário para entender a função **pushTarget**. **pushTarget** define um parâmetro **target** com valor de **DepTarget** ou nulo. Então a função adiciona

à constante global **targetStack** o alvo selecionado. **targetStack** é apenas um **Array** que guarda os alvos selecionados. Então a função coloca a propriedade **target** de **Dep** com o mesmo valor de **target**. Isso é feito, como o comentário sugere, para selecionar o alvo que vai ser avaliado no momento.

```
export default class Dep {
  static target?: DepTarget | null
  id: number
  subs: Array<DepTarget | null>
  // pending subs cleanup
  _pending = false
}
```

Dep é uma classe que guarda um **target DepTarget**, um id numérico, um valor **subs** que guarda um **Array** de **TargetWatcher**, e um valor **_pending** de tipo booleano que sinaliza se ainda existem valores em **subs** a serem avaliados.

```
export interface DepTarget extends DebuggerOptions {
  id: number
  addDep(dep: Dep): void
  update(): void
}
```

DepTarget é uma interface que guarda um id numérico e estende a interface **DebuggerOptions**, que possui duas funções, **onTrack** e **onTrigger**, ambas retornam **void** e ambas requerem **DebuggerEvent** como parâmetro. **DebuggerEvent** é apenas um tipo **wrapper** que contém uma propriedade de nome **effect** com tipo **any**.

```
export interface DebuggerOptions {
  onTrack?: (event: DebuggerEvent) => void
  onTrigger?: (event: DebuggerEvent) => void
}
```

```
export type DebuggerEvent = {
  /**
   * @internal
   */
  effect: any
} & DebuggerEventExtraInfo
```

```
export type DebuggerEventExtraInfo = {
```

```

target: object
type: TrackOpTypes | TriggerOpTypes
key?: any
newValue?: any
oldValue?: any
}

```

Porém, na sua definição, o tipo **DebuggerEvent** é concebido de tal forma que permite acesso aos valores de **DebuggerEventExtraInfo** simultaneamente. **DebuggerEventExtraInfo** contém os valores **target** de tipo **object**, **type** de tipo **TrackOpTypes** ou **TriggerOpTypes** e **key**, **newValue** e **oldValue**, todas as três possuem valor de tipo **any**.

```

export const enum TrackOpTypes {
  GET = 'get',
  TOUCH = 'touch'
}

export const enum TriggerOpTypes {
  SET = 'set',
  ADD = 'add',
  DELETE = 'delete',
  ARRAY_MUTATION = 'array mutation'
}

```

TrackOpTypes e **TriggerOpTypes** são apenas enumeradores para operações e gatilhos.

```
export let currentInstance: Component | null = null
```

Todo o contexto da chamada para **pushTarget** foi explicado. A partir deste ponto, os valores das variáveis **prevInst** e **prevScope** são atribuídos. A variável **prevInst** recebe o valor de um componente ou um nulo segundo a sua definição.

```
export let activeEffectScope: EffectScope | undefined
```

Já o valor de **prevScope** é provido pela função **getCurrentScope()**, uma função wrapper para obter o valor da variável global **activeEffectScope** que possui o tipo de **EffectScope** ou **undefined**.

```

export class EffectScope {
  active = true
  effects: Watcher[] = []
  cleanups: (() => void)[] = []
}

```

```

parent: EffectScope | undefined
scopes: EffectScope[] | undefined
_vm?: boolean
private index: number | undefined

```

A classe **EffectScope** é definida com os valores de **active** (um booleano), **effects** (um **Array** de vigias), **cleanups** (um **Array** de funções que não retornam valor), **parent** (uma instância de **EffectScope** ou **undefined**), **scopes** (também uma instância de **EffectScope** ou **undefined**), **_vm** (um booleano) e **index** (um **number** ou **undefined** privado).

```

const prevScope = getCurrentScope()
setContext && setCurrentInstance(vm)
const handlers = vm.$options[hook]
const info = `${hook} hook`

```

Então a variável `prevScope` receberá o valor da variável global `activeEffectScope` indiretamente, pela chamada à função `getCurrentScope()`. A constante **handler** vai receber o valor de `vm.$options[hook]`, trazendo as funções **hooks** e guardando elas dentro de um **Array**. A constante **info** é criada e recebe o valor tipo de um **hook**, podendo ser 'beforeCreate' ou 'created', essa constante serve para chamarmos apenas os **hooks** de um tipo ou outro. Se existem **hooks** a serem chamados antes da inicialização, eles serão lidos na primeira chamada de `callHook()`, se tivermos **hooks** a serem chamados após a inicialização, eles serão lidos na segunda chamada à `callHook()`.

```

if (handlers) {
  for (let i = 0, j = handlers.length; i < j; i++) {
    invokeWithErrorHandling(handlers[i], vm, args || null, vm, info)
  }
}
if (vm._hasHookEvent) {
  vm.$emit('hook:' + hook)
}
if (setContext) {
  setCurrentInstance(prevInst)
  prevScope && prevScope.on()
}

```

```

popTarget()

```

Neste trecho de **callHook()** é verificado a existência de manipuladores. Caso haja, é feita uma iteração sobre os manipuladores chamando a função **invokeWithErrorHandling()** para cada um deles.

```
export function invokeWithErrorHandling(
  handler: Function,
  context: any,
  args: null | any[],
  vm: any,
  info: string
) {
  let res
  try {
    res = args ? handler.apply(context, args) : handler.call(context)
    if (res && !res._isVue && isPromise(res) &&
      !(res as any)._handled) {
      res.catch(e => handleError(e, vm, info + ` (Promise/async)`)
      ;(res as any)._handled = true
    }
  } catch (e: any) {
    handleError(e, vm, info)
  }
  return res
}
```

A função **invokeWithErrorHandling()** tenta chamar as funções manipuladoras providenciadas pelo **hook**, caso não consiga, lida com o erro e retorna o valor de resposta. Após a iteração de todos os manipuladores através de **invokeWithErrorHandling()**, o *Vue* chegará a existência de eventos de **hook** no componente, e se houver, chamará **vm.\$emit()**.

\$emit: (event: string, ...args: **Array**<any>) => Component

Esta função tem diferentes implementações por plataforma, mas o importante é que há o fornecimento de um evento, por meio de uma **string**, e outros argumentos por meio de um **Array** de elementos **any** e retorna um componente. Depois o *Vue* checa se o valor de **setContext**, parâmetro da função, é verdadeiro. Se for, é utilizado o valor de **setCurrentInstance()** para definir a instância atual do componente com o componente que foi lido.

```
if (setContext) {
  setCurrentInstance(prevInst)
```

```
prevScope && prevScope.on()
}
```

Esta estrutura irá checar se o valor de **prevScope** está definido e não é **null**, **0**, uma **string** vazia, **undefined**, **NaN** ou **false**. Caso o valor de **prevScope** não seja nenhum desses valores será chamada a função **on()** de **prevScope**. Foi observado anteriormente que **prevScope** recebe o valor de **activeEffectScope** indiretamente, que é um **EffectScope**. Observando o método **on()** de **EffectScope**, é possível compreender que **on()** está definindo o valor de **activeEffectScope** como o valor atual de **prevScope**. Isso previne a má definição de escopo durante a chamada de **setCurrentInstance** para prevenir chamadas duplicadas de **hooks**.

```
export function popTarget() {
  targetStack.pop()
  Dep.target = targetStack[targetStack.length - 1]
}
```

Ao final, é chamada a função **popTarget()** que irá remover o **DepTarget** atual da pilha de **targetStack** e define **Dep.target** com o valor anterior. Esta foi toda a funcionalidade de **callHooks()**. Por fim, na inicialização de mixins em **initMixin()**, o seguinte código é apresentado:

```
if (__DEV__ && config.performance && mark) {
  vm._name = formatComponentName(vm, false)
  mark(endTag)
  measure(`vue ${vm._name} init`, startTag, endTag)
}

if (vm.$options.el) {
  vm.$mount(vm.$options.el)
}
```

Este trecho marca a tag de performance caso o Vue esteja em perfil de desenvolvimento, e no final, é chamada **vm.\$mount()** caso **vm.\$options.el** seja um valor instanciado e não seja uma **string** vazia. A função **\$mount** é definida na classe **Component**:

```
$mount: (
  el?: Element | string,
  hydrating?: boolean
) => Component & { [key: string]: any }
```

Recebe um **Element** ou uma **string** como parâmetro para **el**, recebe um booleano **hydrating** e retorna uma instância de **Component** que terá uma assinatura indexada, que permite a adição de propriedades arbitrárias durante o funcionamento do *Vue* após a inicialização deste componente.

Aqui, é encerrada a função de inicialização. Outras funções de inicialização definidas em `initMixin()` serão analisadas nos próximos capítulos.

A.2.1 Inicialização de Ciclos de Vida

```
export function initLifecycle(vm: Component) {
  const options = vm.$options

  // locate first non-abstract parent
  let parent = options.parent
  if (parent && !options.abstract) {
    while (parent.$options.abstract && parent.$parent) {
      parent = parent.$parent
    }
    parent.$children.push(vm)
  }

  vm.$parent = parent
  vm.$root = parent ? parent.$root : vm

  vm.$children = []
  vm.$refs = {}

  vm._provided = parent ? parent._provided : Object.create(null)
  vm._watcher = null
  vm._inactive = null
  vm._directInactive = false
  vm._isMounted = false
  vm._isDestroyed = false
  vm._isBeingDestroyed = false
}
```

Ao início de **initLifecycle** há um parâmetro **vm** de tipo **Component**. É definida também uma constante **options** com as propriedades deste componente. Na estrutura de repetição, o valor procurado é o do primeiro componente não-abstrato pai, e seus são definidos filhos em **parent.\$children.push()**. É definido então o componente parente em

"vm.\$parent = parent" e o componente raiz, que pode ser tanto o componente **vm** quanto a raiz do componente parente. Os elementos filhos de **vm** são inicializados como um **Array** vazio, e as referências como um objeto vazio. É então definido se um parente foi providenciado para **_provided** e são inicializados os valores de **_watcher** e **_inactive** com valor **null** e as propriedades **_directNative**, **_isMounted**, **_isDestroyed** e **isBeingDestroyed** com um valor falso.

A.2.2 Inicialização de Eventos

Esta função inicializa algumas variáveis e as atualiza de acordo com as escutas dos seus genitores.

```
export function initEvents(vm: Component) {
  vm._events = Object.create(null)
  vm._hasHookEvent = false
  // init parent attached events
  const listeners = vm.$options._parentListeners
  if (listeners) {
    updateComponentListeners(vm, listeners)
  }
}
```

A definição de **initEvents** requer um elemento **Component** como parâmetro. Então o valor da propriedade **_events** de **vm** é definido como um objeto nulo, e o valor da propriedade **_hasHookEvent** como falso. Depois é criada uma constante **listeners** com os valores de **vm.\$options._parentListeners** que pode ser um objeto ou um valor nulo de acordo com a sua definição:

_parentListeners?: object | null

Depois, na estrutura condicional, é checado se há ouvintes para o parente do elemento, e se houver, os ouvintes do elemento são atualizados com a função wrapper **updateComponentListeners()**.

```
export function updateComponentListeners(
  vm: Component,
  listeners: Object,
  oldListeners?: Object | null
) {
  target = vm
  updateListeners(
    listeners,
```

```

    oldListeners || {},
    add,
    remove,
    createOnceHandler,
    vm
  )
  target = undefined
}

```

É recebido como parâmetros os valores **vm** de tipo **Component**, o componente a ser avaliado, **listeners** de tipo **Object** e opcionalmente **oldListeners** de tipo **Object** ou **null** caso não exista. **Definimos** o alvo como o valor de **vm**, e é executada a função principal **updateListeners()**.

```

function add(event, fn) {
  target.$on(event, fn)
}

```

```

function remove(event, fn) {
  target.$off(event, fn)
}

```

```

function createOnceHandler(event, fn) {
  const _target = target
  return function onceHandler() {
    const res = fn.apply(null, arguments)
    if (res !== null) {
      _target.$off(event, onceHandler)
    }
  }
}

```

Nas suas definições, **add()**, **remove()** e **createOnceHandler()** são funções específicas definidas em cada componente, uma vez que o componente é o elemento atual em **target**.

```

let name, cur, old, event
for (name in on) {
  cur = on[name]
  old = oldOn[name]
  event = normalizeEvent(name)
}

```

Voltando para a função **updateListeners()**, são criadas as variáveis **name**, **cur**, **old** e **event**. Depois, é feita uma iteração pelos valores de **on** usando **names** para capturar o seu valor. É definido o valor de **cur** como o objeto atual, **old** com o valor do objeto de **oldOn** e evento com o valor de **normalizeEvent()**.

```
const normalizeEvent = cached(
  (
    name: string
  ): {
    name: string
    once: boolean
    capture: boolean
    passive: boolean
    handler?: Function
    params?: Array<any>
  } => {
    const passive = name.charAt(0) === '&'
    name = passive ? name.slice(1) : name
    const once = name.charAt(0) === '~'
    name = once ? name.slice(1) : name
    const capture = name.charAt(0) === '!'
    name = capture ? name.slice(1) : name
    return {
      name,
      once,
      capture,
      passive
    }
  }
)
```

A definição de **normalizeEvent()** nos mostra que ela é uma variável que recebe uma função de nome **cached**, que recebe uma string **name** como parâmetro e retorna um objeto com um **name** de valor string, **once**, **capture** e **passive** de valor booleano, **handler** um valor opcional de tipo **Function** e **params** um valor opcional de tipo **Array** de any. A partir daqui, um padrão de operações será estabelecido, checando o primeiro caractere de **name** e retornando um booleano para essa checagem. Os caracteres procurados são '&', '~' e '!'. Primeiro é verificada a igualdade com '&' e o valor é devolvido para **passive**, depois é checado a igualdade com '~' e devolvendo o valor para **once** e por fim é checado a igualdade com '!' e o valor é devolvido para **capture**. Caso sejam encontrados qualquer

um destes valores, a string será cortada entre o primeiro e o segundo caractere e será devolvida apenas a segunda metade do valor. Por fim, o valor é devolvido em formato de objeto com os valores de **name**, **once**, **capture** e **passive**.

```
if (isUndef(cur)) {
  __DEV__ &&
  warn(
    `Invalid handler for event "${event.name}" \
: got ` + String(cur),
    vm
  )
}
```

Voltando para **updateListeners()**, é verificado se o valor de **cur** é definido. Caso seja, e caso o *Vue* esteja em perfil de desenvolvimento, um aviso é retornado pela função.

```
else if (isUndef(old)) {
  if (isUndef(cur.fns)) {
    cur = on[name] = createFnInvoker(cur, vm)
  }
}
```

Depois é checado se **old** não está definido, e caso seja verdade, é verificado se as funções de **cur** não são definidas, e se for verdade, o valor de **on[name]** é definido como uma função com os parâmetros de **cur** e **vm** e é retornado o valor dessa chamada para **cur**.

```
export function createFnInvoker(
  fns: Function | Array<Function>,
  vm?: Component
): Function {
  function invoker() {
    const fns = invoker.fns
    if (isArray(fns)) {
      const cloned = fns.slice()
      for (let i = 0; i < cloned.length; i++) {
        invokeWithErrorHandling(
          cloned[i],
          null,
          arguments as any,
          vm,

```

```

        `v-on handler`
    )
}
} else {
    return invokeWithErrorHandling(
        fns,
        null,
        arguments as any,
        vm,
        `v-on handler`
    )
}
}
invoker.fns = fns
return invoker
}

```

A função **createFnInvoker()** recebe uma função ou uma **Array** de funções e opcionalmente um **Component** para criar uma ou mais funções que serão invocadas com tratamento de erros posteriormente pela função **invokeWithErrorHandling()**, vista anteriormente no capítulo “Inicialização de Mixins”.

```

if (isTrue(event.once)) {
    cur = on[name] = createOnceHandler(event.name, cur, event.capture)
}

```

Depois, é verificado se o valor de **event.once** é verdadeiro, e caso seja, é definido o valor de **on[name]** como um **wrapper** para a função **createOnceHandler()** é retornado o seu valor a **cur**.

```
add(event.name, cur, event.capture, event.passive, event.params)
```

Então, é chamada a função **add()** com os valores do nome do evento, as funções definidas em **cur**, os booleanos **event.capture** e **event.passive** e os parâmetros do evento.

```

else if (cur !== old) {
    old.fns = cur
    on[name] = old
}

```

Caso a **old** não seja indefinido na última estrutura condicional, é verificado se **cur** seja estritamente diferente de **old**, é definido o valor das funções de **old** como as funções de **cur**, e é definido **on[name]** com o valor de **old**.

```

for (name in oldOn) {
  if (isUndef(on[name])) {
    event = normalizeEvent(name)
    remove(event.name, oldOn[name], event.capture)
  }
}

```

Por fim, fora da estrutura de repetição principal, é feita uma iteração sobre os valores de **oldOn**. Se caso o valor de **on[name]** seja indefinido, isso significa que o evento não está mais operacional, então o evento é normalizado utilizando **normalizeEvent()** e é guardado na variável **event**, e é então utilizado **remove()** para remover o evento. Voltando para **updateComponentListeners()**, é definido por fim, o valor de **target** como indefinido, marcando que o alvo atual a ser processado pode ser assumido por um novo valor, marcando assim o fim da inicialização de eventos.

A.2.3 Inicialização da Renderização

Inicializa **_vnode**, **_staticTrees** e algumas outras variáveis e métodos para renderização. É aqui que se inicia a criação do documento virtual, técnica utilizada pelo *Vue* para calcular com mais velocidade as mudanças em trechos do documento.

```

export function initRender(vm: Component) {
  vm._vnode = null // the root of the child tree
  vm._staticTrees = null // v-once cached trees
  const options = vm.$options

```

No início de **initRender()** há a declaração do parâmetro **vm** de tipo **Component**. No corpo da função, é atribuído um valor nulo para **vm._vnode** (a raiz da árvore de filhos) e para **vm._staticTrees**. É criado também uma constante **options** com o valor de **vm.\$options**. É então feita uma asserção de valor não-nulo com **"vm.\$vnode = options._parentVnode !"** e esse valor é atribuído a variável **parentVnode**.

```

const renderContext = parentVnode && (parentVnode.context as Component)
vm.$slots = resolveSlots(options._renderChildren, renderContext)
vm.$scopedSlots = parentVnode
  ? normalizeScopedSlots(
    vm.$parent!,
    parentVnode.data!.scopedSlots,
    vm.$slots
  )
  : emptyObject

```

É criado uma constante **renderContext** que recebe o valor de **parentVnode** no formato de um **Component** contanto que o valor de **parentVnode** não seja nulo, string vazia, **undefined** ou **NaN**. É guardado na propriedade **vm.\$slots** o resultado da função **resolveSlots()**.

```
export function resolveSlots(
  children: Array<VNode> | null | undefined,
  context: Component | null
): { [key: string]: Array<VNode> }
```

A definição de **resolveSlots()** tem como parâmetros uma variável **children** de tipo **Array<Vnode>**, null ou undefined e uma variável **context** de tipo **Component** ou nulo, e retorna um **Array** de **Vnode** com indexação de assinaturas.

```
if (!children || !children.length) {
  return {}
}
```

É checado então se **children** e **children.length** são definidos e retornando um objeto vazio se sim.

```
const slots: Record<string, any> = {}
for (let i = 0, l = children.length; i < l; i++) {
  const child = children[i]
  const data = child.data
  if (data && data.attrs && data.attrs.slot) {
    delete data.attrs.slot
  }
}
```

É criado também uma constante **slots** de tipo **Records<string,any>** que recebe um objeto vazio. Um **Record** é uma estrutura cujos itens são uma chave **K** para um elemento **J**, neste caso a chave é a string e o elemento é o valor de tipo **any**. É iniciado uma iteração sobre os elementos de **children**. Uma constante **child** é definida com o valor do filho atual, e uma constante **data** com o valor de data deste filho. É feita uma checagem se os valores de **data**, **data.attrs** e **data.attrs.slot** são não-nulos e caso sejam, é deletado o valor de **data.attrs.slot**.

```
if (
  (child.context === context || child.fnContext === context) &&
  data &&
  data.slot !== null
) {
```

```

const name = data.slot
const slot = slots[name] || (slots[name] = [])
if (child.tag === 'template') {
  slot.push.apply(slot, child.children || [])
} else {
  slot.push(child)
}

```

É verificado então se o valor de **child.context** é estritamente igual a **context** ou se o valor de **child.fnContext** é estritamente igual a **context** e se **data** é definido e se **data.slot** não é nulo. Caso todas essas condições sejam avaliadas como verdadeiras, é criado uma constante **name**, com o valor de **data.slot** e uma constante **slot** com o valor de **slots[name]** ou com valor de um **Array** vazio caso o valor de **slots[name]** não esteja inicializado. É avaliado então se o valor de **child.tag** é estritamente igual à uma string 'template'. Caso seja verdade, é adicionado à **slot** os valores de **slot** como chave e de **child.children** como valor, ou um **Array** vazio caso **child.children** não esteja inicializado. Se **child.tag** não for igual à 'template', o valor de **child** é guardado dentro de **slot**.

```

else {
  ;(slots.default || (slots.default = [])).push(child)
}

```

Caso as expressões anteriores não resultem em verdadeiro, é instruído ao *Vue* guardar o valor de **child** em **slots.default**, e inicializar **slots.default** caso este não esteja inicializado.

```

for (const name in slots) {
  if (slots[name].every(isWhitespace)) {
    delete slots[name]
  }
}
return slots
}

```

Por fim, a constante **slots** é iterada utilizando os nomes das chaves, e é checado se as chaves dos elementos são apenas espaço em branco. Caso sejam, o valor atribuído a aquele **slot** é removido. Então o valor de **slots** é retornado.

```

export function normalizeScopedSlots(
  ownerVm: Component,
  scopedSlots: { [key: string]: Function } | undefined,

```

```

    normalSlots: { [key: string]: VNode[] },
    prevScopedSlots?: { [key: string]: Function }
  ): any {

```

Voltando a **initRender()** é atribuído o valor à **vm.\$scopedSlots** o valor de **normalizeScopedSlots()** caso **parentVnode** seja definido ou um objeto vazio caso não. A função **normalizeScopedSlots()** é definida com os parâmetros **ownerVm** de tipo **Component**, **scopedSlots** de tipo objeto que guarda funções indexadas por nomes em strings ou de tipo indefinido, **normalSlots** de tipo objeto que guarda o tipo **Array** de **Vnode** indexados por nomes em strings e **prevScopedSlots** que guarda funções indexadas por nomes em strings. A função então retorna um valor de qualquer tipo.

```

let res
const hasNormalSlots = Object.keys(normalSlots).length > 0
const isStable = scopedSlots ? !!scopedSlots.$stable : !hasNormalSlots
const key = scopedSlots && scopedSlots.$key

```

A função é iniciada criando uma variável **res**, uma constante **hasNormalSlots** que avalia se o tamanho das chaves de **normalSlots** é maior que 0, uma constante **isStable** que avalia se **scopedSlots** é definido e define o seu valor com **scopedSlots.\$stable** caso seja e com a negação de **hasNormalSlots** caso não seja e é definido também uma constante **key** com o valor de **scopedSlots.\$key** caso esta não seja indefinido ou nulo.

```

if (!scopedSlots) {
  res = {}
} else if (scopedSlots._normalized) {
  return scopedSlots._normalized
}

```

É então iniciada uma série de checagens acerca dos valores dos parâmetros e das constantes criadas. No início é checado se o valor de **scopedSlots** é indefinido, atribuindo um objeto vazio para **res** caso seja. Após isso, é checado se a propriedade **_normalized** de **scopedSlots** é verdadeira, retornando esse mesmo valor caso seja.

```

else if (
  isStable &&
  prevScopedSlots &&
  prevScopedSlots !== emptyObject &&
  key === prevScopedSlots.$key &&
  !hasNormalSlots &&
  !prevScopedSlots.$hasNormal

```

```

    ){
      return prevScopedSlots
    }

```

Depois é checado se **isStable** é verdadeiro, se **prevScopedSlots** é definido e não é um objeto vazio, se **key** é o mesmo valor da propriedade **\$key** de **prevScopedSlots**, se **hasNormalSlots** é falso e se a propriedade **\$hasNormal** de **prevScopedSlots** é falsa. Caso essa expressão seja avaliada como verdadeira, é retornado o valor de **prevScopedSlots** pela função.

```

else {
  res = {}
  for (const key in scopedSlots) {
    if (scopedSlots[key] && key[0] !== '$') {
      res[key] = normalizeScopedSlot(
        ownerVm,
        normalSlots,
        key,
        scopedSlots[key]
      )
    }
  }
}

```

Após isso, caso a expressão anterior seja falsa, é definido o valor da variável **res** como um objeto vazio, e é criada uma iteração sobre os valores de **scopedSlots** com uma constante **key** como chave. É então checado se o valor da chave **key** em **scopedSlots** é definido e se o valor do índice 0 em **key** é igual a '\$'. Caso a expressão seja avaliada como verdadeira, o valor da chave **key** em **res** é definido com o resultado da chamada para a função **normalizeScopedSlot()**.

```

function normalizeScopedSlot(vm, normalSlots, key, fn) {
  const normalized = function () {

```

A função **normalizeScopedSlot()** recebe os valores **vm**, **normalSlots**, **key** e **fn** como parâmetros. É então criada uma função anônima que é assinalada para a constante **normalized**.

```

const cur = currentInstance
setCurrentInstance(vm)

```

A função inicia guardando o valor do componente atual sendo avaliado para uma constante **cur**, e é definido com o valor de **vm** o componente atual a ser avaliado com a chamada para a função **setCurrentInstance()**.

```
let res = arguments.length ? fn.apply(null, arguments) : fn({})
```

É então checado se o valor de **arguments.length** é definido e o resultado é guardado em uma variável **res**. Caso seja, o valor é definido com a chamada para **fn.apply()** adicionando nulo como o primeiro valor a ser chamado pela função **fn()**, caso contrário, é definido o valor dos parâmetros de **fn()** com um objeto vazio.

```
res =
  res && typeof res === 'object' && !isArray(res)
    ? [res] // single vnode
    : normalizeChildren(res)
```

Depois o valor de **res** é definido com a avaliação de se **res** é definido, se **res** é um objeto e se **res** não é um **Array**. Caso seja avaliado como verdadeiro, o valor de **res** é definido como um **Array** cujo único valor definido é **res**, caso contrário, o valor de **res** é definido com a chamada de **normalizeChildren()** com o valor de **res** como parâmetro.

```
export function normalizeChildren(children: any):
  Array<VNode> | undefined {
  return isPrimitive(children)
    ? [createTextVNode(children)]
    : isArray(children)
      ? normalizeArrayChildren(children)
      : undefined
  }
```

```
export function isPrimitive(value: any): boolean {
  return (
    typeof value === 'string' ||
    typeof value === 'number' ||
    // $flow-disable-line
    typeof value === 'symbol' ||
    typeof value === 'boolean'
  )
}
```

A função **normalizeChildren()** é definida com um parâmetro **children** de qualquer tipo e retorna um valor de **Array** que guarda valores de tipo **Vnode** ou é indefinido. A função

apenas cria uma estrutura condicional ternária que checa se o valor de **children** é primitivo através da função **isPrimitive()**, no caso, se o valor é de tipo **number**, **string**, **symbol** ou **boolean**, de acordo com a definição da função. Caso o valor de **children** seja primitivo, o valor de **children** é fornecido como parâmetro para a função **createTextNode()**, e é retornado como único valor dentro de um **Array**, caso contrário, é checado se o valor de **children** é um **Array** através da função **isArray()**. Caso seja, o valor a ser retornado é o mesmo retornado pela função **normalizeArrayChildren()**, caso contrário, o valor a ser retornado é indefinido.

```
function normalizeArrayChildren(
  children: any,
  nestedIndex?: string
): Array<VNode>
```

A função **normalizeArrayChildren()** é definida com os parâmetros **children** de qualquer tipo e **nestedIndex** de tipo **string**, e retorna um **Array** de **Vnode**.

```
const res: VNode[] = []
let i, c, lastIndex, last
```

A função é iniciada com a criação de uma constante **res** de tipo **Array** de **Vnode** vazio e as variáveis **i**, **c**, **lastIndex** e **last**.

```
for (i = 0; i < children.length; i++)
```

É então criado uma iteração sobre o tamanho da propriedade **length** de **children**.

```
c = children[i]
if (isUndef(c) || typeof c === 'boolean') continue
lastIndex = res.length - 1
last = res[lastIndex]
```

O valor de **c** é definido como a chave **i** de **children**, e caso **c** seja indefinido ou o tipo de **c** seja booleano, a estrutura de repetição é forçada a iniciar a próxima iteração. Caso contrário a estrutura continua com a definição do valor da variável **lastIndex** com o valor de **res.length** decrementado, e com a iniciação do valor de **last** com o valor do índice **lastIndex** de **res**.

```
if (isArray(c)) {
  if (c.length > 0) {
    c = normalizeArrayChildren(c, `${nestedIndex || ""}_${i}`)
    if (isTextNode(c[0]) && isTextNode(last)) {
```

```

        res[lastIndex] = createTextVNode(last.text + c[0].text)
        c.shift()
    }
    res.push.apply(res, c)
}
}

```

Uma estrutura condicional é criada para checar se o valor de **c** é um **Array**. Caso seja, é checado se o tamanho de **c** é maior que 0, e caso seja, é definido o valor de **c** com o valor da chamada recursiva para **normalizeArrayChildren** passando o valor atual de **c** como parâmetro, e é checado se o valor do índice 0 de **c** e o valor de **last** são ambos nódulos de texto, fazendo uma junção dos dois em um único valor de **Vnode** e chamando a função **shift()** de **c** para remover o primeiro elemento. É então adicionado a **res** o valor de **c** chamando **res.push.apply()**.

```

else if (isPrimitive(c)) {
    if (isTextNode(last)) {
        res[lastIndex] = createTextVNode(last.text + c)
    } else if (c !== "") {
        res.push(createTextVNode(c))
    }
}

```

Caso o valor de **c** não seja um **Array** na última estrutura condicional, é checado então se o valor de **c** é primitivo e se o valor de **last** é um nódulo de texto. Caso seja o valor do índice **lastIndex** em **res** é definido com o valor do agrupamento dos valores de **last** e **c**. Caso **last** não seja um nódulo de texto, é checado se **c** possui o valor diferente de uma string vazia. Caso seja verdadeiro, **c** é adicionado a **res** através da chamada **push()** depois de ser convertido para um nódulo de texto.

```

else {
    if (isTextNode(c) && isTextNode(last)) {
        res[lastIndex] = createTextVNode(last.text + c.text)
    } else {
        if (
            isTrue(children._isVList) &&
            isDef(c.tag) &&
            isUndef(c.key) &&
            isDef(nestedIndex)
        ) {
            c.key = `__vlist${nestedIndex}_${i}__`
        }
    }
}

```

```

        res.push(c)
    }
}
}
return res
}

```

Caso **c** não seja um valor primitivo, é checado então se ambos **last** e **c** são nódulos de texto, combinando ambos em um único nódulo de texto e adicionando eles ao índice **lastIndex** em **res** caso verdadeiro. Do contrário, é checado se **children** é uma lista virtual, se a propriedade **tag** de **c** é definida, se a propriedade **key** de **c** é indefinida, e se **nestedIndex** é definido. Caso essa expressão seja avaliada como verdadeira, o valor da propriedade **key** de **c** é definida com o valor de uma string formatada com os valores de **nestedIndex** e **i**. É então adicionado o valor de **c** a **res** utilizando o método **push()** e o valor de **res** é retornado pela função.

```

const vnode: VNode | null = res && res[0]
setCurrentInstance(cur)
return res &&
  (!vnode ||
    (res.length === 1 && vnode.isComment && !isAsyncPlaceholder(vnode))) // #9658, #103
  ? undefined
  : res
}

```

Ao retornar para a função anônima guardada em **normalized** na função **normalizeScopedSlots()**, é criado uma constante **vnode** que guarda um valor de **Vnode** ou nulo, e recebe o valor do índice 0 de **res** caso o valor não seja nulo. O valor do componente a ser avaliado é definida como **cur** novamente com a chamada para a função **setCurrentInstance()** e é retornado um valor baseado na seguinte expressão: se **res** é definido ou se **vnode** é indefinido ou a propriedade **length** de **res** é igual a 1 e a propriedade **isComment** de **Vnode** é verdadeira e a chamada para **isAsyncPlaceholder()** com o valor de **vnode** como parâmetro é falso. Caso essa expressão seja avaliada como verdadeira, o valor a ser retornado por **normalized** é indefinido, caso falsa, o valor a ser retornado é o mesmo de **res**.

```

if (fn.proxy) {
  Object.defineProperty(normalSlots, key, {
    get: normalized,
    enumerable: true,
    configurable: true
  })
}

```

```

    })
  }
  return normalized
}

```

Ao final de **normalizeScopedSlots()** é checado se o valor da propriedade **proxy** de **fn** é definido, e caso seja, o valor da propriedade **key** é definido com o valor de **normalSlots** e a opção **get** é definida com o valor de **normalized**. Ao final da função o valor de **normalized** é retornado.

```
vm._c = (a, b, c, d) => createElement(vm, a, b, c, d, false)
```

```
vm.$createElement = (a, b, c, d) => createElement(vm, a, b, c, d, true)
```

Depois há a definição de **vm._c** e **vm.\$createElement**, ambas embalagens para **createElement()**. Essas definições causam problemas com o *Typescript Language Server*, por isso tem um comentário com **@ts-expect-error**. É criado também uma constante **parentData** que checa se **parentVnode** é definido e se for, recebe o valor de **parentVnode.data**. Existe então a definição de uma estrutura condicional que checa se o *Vue* está em perfil de desenvolvimento. Como não é do interesse para o entendimento do funcionamento do *Vue*, a análise vai para o **else** desta estrutura, pois ambas fazem basicamente a mesma coisa

```

else {
  defineReactive(
    vm,
    '$attrs',
    (parentData && parentData.attrs) || emptyObject,
    null,
    true
  )
  defineReactive(
    vm,
    '$listeners',
    options._parentListeners || emptyObject,
    null,
    true
  )
}

```

Existem duas chamadas à **defineReactive()**, uma com a propriedade **\$attrs** e outra com a propriedade **\$listeners** de **vm**.

```
export function defineReactive(
  obj: object,
  key: string,
  val?: any,
  customSetter?: Function | null,
  shallow?: boolean,
  mock?: boolean,
  observeEvenIfShallow = false
)
```

A definição de **defineReactive()** toma como parâmetros um **obj** de tipo **object**, um **key** de tipo **string**, um **val** de tipo **any**, um **customSetter** de tipo **Function**, um **shallow** e **mock** de tipo booleano e um **observeEvenIfShallow** de tipo booleano que tem como padrão o valor falso caso ele não seja definido.

```
const dep = new Dep()
```

A função começa criando uma instância da classe **Dep** e assinalando o seu valor para a constante **dep**. **Dep** é uma classe mostrada na subseção "Inicialização de Mixins".

```
const property = Object.getOwnPropertyDescriptor(obj, key)
if (property && property.configurable === false) {
  return
}
```

É criado então uma constante **property** que guarda o valor da chamada de **Object.getOwnPropertyDescriptor()** utilizando o valor de **key** para encontrar a propriedade em **obj**. Depois é checado se a propriedade é um valor definido e se a propriedade **property.configurable** é falsa. Caso seja, a função termina com um retorno prévio.

```
const getter = property && property.get
const setter = property && property.set
if (
  (!getter || setter) &&
  (val === NO_INITIAL_VALUE || arguments.length === 2)
) {
  val = obj[key]
}
```

É criada uma constante **getter** com o valor de **property.get** caso esta não seja nula ou indefinida, e uma constante **setter** com o valor de **property.set** caso esta também não seja nula ou indefinida. Na estrutura condicional, é checado se o valor de **getter** é nulo

ou indefinido ou se **setter** tem um valor definido e se o valor de **val** não possui nenhum valor inicial, neste caso, é um objeto vazio ou se o número de argumentos utilizados para a chamada da função é exatamente 2. Se essa expressão for avaliada como verdadeira, o código irá definir o valor de **val** com a propriedade **key** de **obj**.

```
let childOb = shallow ? val && val.__ob__ : observe(val, false, mock)
```

Em seguida, é criada uma variável **childOb** que avalia o valor de **shallow**: caso tenha um valor definido e verdadeiro, é definido o valor com a propriedade de **val.__ob__** caso **val** não seja nula ou indefinida, ou como nula ou indefinida se for, e caso tenha um valor indefinido, falso ou nulo, a variável recebe o valor de **observe()**.

```
export function observe(
  value: any,
  shallow?: boolean,
  ssrMockReactivity?: boolean
): Observer | void {
  if (value && hasOwn(value, '__ob__') &&
    value.__ob__ instanceof Observer) {
    return value.__ob__
  }
  if (
    shouldObserve &&
    (ssrMockReactivity || !isServerRendering()) &&
    (isArray(value) || isPlainObject(value)) &&
    Object.isExtensible(value) &&
    !value.__v_skip /* ReactiveFlags.SKIP */ &&
    !isRef(value) &&
    !(value instanceof VNode)
  ) {
    return new Observer(value, shallow, ssrMockReactivity)
  }
}
```

A função **observe()** checa se o valor **value** apresentado como parâmetro tem uma propriedade **__ob__** e se essa propriedade é uma instância da classe **Observer**, e se ela for, apenas retorna o valor de **val.__ob__**. Caso contrário é avaliado se o *Vue* está usando renderização no lado do servidor para renderizar o **template**, se **value** é um **Array** ou um objeto pleno, se **value** é extensível, se **value** não é uma referência e se **value** não é uma instância de **Vnode**. Caso todos esses valores sejam avaliados como verdade, é criado uma instância de **Observer** e retornado para a variável **childOb** em **defineReactive()**.

```
Object.defineProperty(obj, key, {
  enumerable: true,
  configurable: true
```

Ao voltar para **defineReactive()**, começa-se a definição de uma propriedade **key** em **obj** com os valores do objeto passado. Neste objeto a definição de **enumerable** é feita com valor verdadeiro, **configurable** com valor verdadeiro e duas propriedades com funções: **get** e **set**. O valor de **get** é definido dentro de **defineReactive()** com a seguinte estrutura:

```
get: function reactiveGetter() {
  const value = getter ? getter.call(obj) : val
  if (Dep.target) {
    if (__DEV__) {
      dep.depend({
        target: obj,
        type: TrackOpTypes.GET,
        key
      })
    } else {
      dep.depend()
    }
    if (childOb) {
      childOb.dep.depend()
      if (isArray(value)) {
        dependArray(value)
      }
    }
  }
}
```

A função de **get**, **reactiveGetter()**, se inicia com a criação de uma constante **value** que recebe o valor a partir da checagem do valor de **getter**: caso seja definido, o valor de **getter.call()** é devolvido, caso não seja definido, o valor de **val** é devolvido. É checado então se o valor de **Dep.target** é definido. Caso seja é verificado se o *Vue* está em perfil de desenvolvimento. As duas opções de execução resultam na execução do método **depend()** de **dep**, porém, caso o perfil de desenvolvimento seja confirmado, é enviado para os inscritos dessa dependência uma quantidade maior de informações para depuração. Após essa checagem, é verificado se o valor de **childOb** tem valor definido. Caso tenha, é feita uma chamada para o método **depend()** de **childOb.dep**, e é checado se o valor de **value** é um **Array**, executando a função **dependArray()** caso seja.

```
function dependArray(value: Array<any>) {
  for (let e, i = 0, l = value.length; i < l; i++) {
    e = value[i]
    if (e && e.__ob__) {
      e.__ob__.dep.depend()
    }
    if (isArray(e)) {
      dependArray(e)
    }
  }
}
```

A função **dependArray()** recebe um único parâmetro **value** de tipo **Array** que coleta valores de qualquer tipo. É então criada uma iteração com limite definido acima do tamanho de **value**, e é criada uma variável **e** que recebe o valor do índice de **i** em **value**. É então checado se **e** é definido e se sua propriedade **__ob__** é definida, executando o método **dep.depend()** de **e.__ob__**. Depois é checado se o valor de **e** é um **Array**, e caso seja, é criada uma recursão com uma nova chamada para **dependArray()**.

```
return isRef(value) && !shallow ? value.value : value
```

Ao final de **reactiveGetter()** em **defineReactive()** é retornado o valor a partir de uma expressão ternária: se **value** é uma referência e o valor de **shallow** é falso, o valor da propriedade **value** de **value** é retornado, do contrário, o valor de **value** é enviado como valor de retorno.

```
set: function reactiveSetter(newVal) {
  const value = getter ? getter.call(obj) : val
  if (!hasChanged(value, newVal)) {
    return
  }
  if (__DEV__ && customSetter) {
    customSetter()
  }
  if (setter) {
    setter.call(obj, newVal)
  } else if (getter) {
    return
  } else if (!shallow && isRef(value) && !isRef(newVal)) {
    value.value = newVal
    return
  }
}
```

```

    } else {
      val = newVal
    }
    childOb = shallow ? newVal && newVal.__ob__ : observe(newVal, false, mock)
    if (__DEV__) {
      dep.notify({
        type: TriggerOpTypes.SET,
        target: obj,
        key,
        newValue: newVal,
        oldValue: value
      })
    } else {
      dep.notify()
    }
  }
}

```

O valor de set é definido com uma função **reactiveSetter()** que recebe um valor **newVal** como parâmetro. É criada uma constante **value** com o valor de uma expressão ternária. Se **getter** for definido, é feita uma chamada para seu método **call()** e é devolvido o valor desta chamada, se não, o valor de **val** é devolvido.

É então feita uma checagem acerca do valor de **value**: caso ele não tenha mudado, a função acaba aqui. Depois é checado se o valor de **setter** é definido, e caso seja, o método **call()** de **setter** é chamado com os valores de **obj** e **newVal** como parâmetros.

Caso **setter** não seja definido, é checado se o valor de **getter** é definido, e se não for o caso, a função é encerrada aqui. Caso **getter** não seja definido, é checado se o valor de **shallow** é verdadeiro, se **value** é uma referência e se **newVal** não é uma referência. Caso essa expressão seja avaliada como verdadeira, o valor da propriedade **value** de **value** é definido com o valor de **newVal** e a função é encerrada aqui.

Por fim, caso a expressão anterior seja avaliada como falsa, o valor de **val** recebe diretamente o valor de **newVal**. Ao final de **reactiveSetter()** é definido o valor de **childOb** a partir de uma expressão ternária que checa se o valor de **shallow** é definido. Caso seja é verificado se o valor de **newVal** é definido, e caso seja o valor de **newVal.__ob__** é devolvido. Caso contrário, o valor de retorno da chamada para **observe()** é retornado.

Ao final de **defineReactive()** é realizado o retorno de **dep** como valor. Isso também marca o final da inicialização do *Mixin* de renderização.

A.2.4 Inicialização de Injeções

É uma função simples e direta que compreende as injeções das opções e assinala elas para o respectivo componente.

```
export function initInjections(vm: Component) {
  const result = resolveInject(vm.$options.inject, vm)
```

A função **initInjections()** requer um parâmetro **vm** de tipo **component**. A função começa com a associação do valor de **resolveInject()** para a constante **result**.

```
export function resolveInject(
  inject: any,
  vm: Component
): Record<string, any> | undefined | null {
  if (inject) {
    const result = Object.create(null)
    const keys = hasSymbol ? Reflect.ownKeys(inject) : Object.keys(inject)

    for (let i = 0; i < keys.length; i++) {
      const key = keys[i]
      if (key === '___ob___') continue
      const provideKey = inject[key].from
      if (provideKey in vm.__provided) {
        result[key] = vm.__provided[provideKey]
      } else if ('default' in inject[key]) {
        const provideDefault = inject[key].default
        result[key] = isFunction(provideDefault)
          ? provideDefault.call(vm)
          : provideDefault
      } else if (___DEV___) {
        warn(`Injection "${key as string}" not found`, vm)
      }
    }
  }
  return result
}
```

A função **resolveInject()** recebe um **inject** de tipo **any**, e um **vm** de tipo **Component** como parâmetros, e devolve ou um **Record** com chave **string** e valor **any** ou indefinido ou nulo. A função começa com uma estrutura condicional que checka se **inject** é definido.

Caso seja, é criada uma constante **result** como um **Object** sem valores, e uma constante **keys** que guarda o valor de **Reflect.ownkeys()** ou **Object.keys()** se **hasSymbol** for verdadeiro ou falso respectivamente.

É declarado então uma iteração sobre os valores de **keys**. É declarada uma constante **key** que recebe o valor a ser iterado e é verificado se **key** é igual a `'__ob__'`, caso seja, esta chave é ignorada e o próximo valor da iteração é verificado. É criado então uma constante **provideKey**, com o valor de **inject[key]**, e então é checado se o valor de **provideKey** está declarado em **vm._provided**. Caso seja, é adicionado ao valor de **result** o valor **key** de **vm._provided**.

Caso não seja, é verificado então se o valor 'default' está em **inject**, caso esteja, é definido a constante **provideDefault** com o valor de **inject[key].default** e é definida a chave **key** de **result** com o valor da checagem de **isFunction()** de **provideDefault**, caso seja verdadeiro é retornado o valor da chamada para **provideDefault.call()** passando o valor de **vm**, caso não seja, o valor de **provideDefault** é retornado.

Por fim, o valor de **result** é retornado ao final da função.

```
if (result) {
  toggleObserving(false)
```

Voltando para **initInjection()**, é checado se o valor de **result** é definido. Caso seja, é executada a função **toggleObserving()** com um valor falso.

```
export let shouldObserve: boolean = true

export function toggleObserving(value: boolean) {
  shouldObserve = value
}
```

A função **toggleObserving()** simplesmente altera o valor da variável global **shouldObserve** para o valor indicado.

```
Object.keys(result).forEach(key => {
  if (__DEV__) {
    defineReactive(vm, key, result[key], () => {
      warn(
        `Avoid mutating an injected value directly \
since the changes will be ` +
        `overwritten whenever the provided \
component re-renders. ` +
        `injection being mutated: "${key}"`,
        vm
```

```

    )
  })
  } else {
    defineReactive(vm, key, result[key])
  }
})
toggleObserving(true)
}

```

Depois, é criada uma iteração sobre cada chave indicada em **result**. e para cada uma delas, é executada a função **defineReactive()** com os valores de **vm**, **key** e a instância de **key** em **result**.

Como mostrado no último capítulo, **defineReactive()** é uma função que captura uma propriedade de um objeto e transforma esta propriedade em uma propriedade reativa, e adiciona vigias a esta propriedade, para monitorar quando o valor desta é alterado.

Ao final de **initInjections()**, o valor de **shouldObserve** é definido com a função **toggleObserving**, desta vez com um valor verdadeiro, e a função encerra sua execução.

A.2.5 Inicialização de Estados

Função onde são obtidos os adereços, métodos, dados, propriedades computadas e funções vigias. Inicializa os valores de cada uma destas em ordem:

- **initProps**: faz validações e embala os adereços, assinalando eles ao componente.
- **initMethod**: simplesmente assinala métodos ao componente.
- **initData**: faz mais validações, mas usa o proxy para assinalar os dados.
- **initComputed**: extrai as funções que você assinala a um componente, cria um vigia com elas e guarda em uma lista, que será processada eventualmente em um componente computado.
- **initWatch**: cria um vigia para cada vez que a função **createWatcher()** é chamada.

```

export function initState(vm: Component) {
  const opts = vm.$options
  if (opts.props) initProps(vm, opts.props)

```

A função **initState()** toma um parâmetro **vm** de tipo **Component**. Ela inicia com a instanciação de uma constante **opts** que recebe o valor **\$options** de **vm**. Depois é checado se a propriedade **props** de **opts** é definida, e caso seja, a função **initProps()** é executada.

```

function initProps(vm: Component, propsOptions: Object) {
  const propsData = vm.$options.propsData || {}
  const props = (vm._props = shallowReactive({}))

```

```
const keys: string[] = (vm.$options._propKeys = [])
const isRoot = !vm.$parent
```

A função **initProps()** é definida com um **vm** de tipo **Component** e um **propsOptions** de tipo **Object** como parâmetros. No seu início, há a criação de uma constante **propsData** com os valores de **vm.\$options.propsData**, ou um objeto vazio caso este seja indefinido. Há também a criação de uma constante **props** que recebe o valor de **shallowReactive()**, que também é atribuído a **vm._props**, uma constante **keys** que recebe um **Array** de **string** vazio, e a constante **isRoot** que recebe a negação de **vm.\$parent**.

```
if (!isRoot) {
  toggleObserving(false)
}
```

Depois, há a checagem de **isRoot**, caso possua um valor falso, é utilizada a função **toggleObserving()** para alterar o valor de **shouldObserve** para falso.

```
for (const key in propsOptions) {
  keys.push(key)
  const value = validateProp(key, propsOptions, propsData, vm)
```

Então, há a criação de uma constante **key** com o valor iterado de cada chave em **propsOptions**. O valor de **key** é adicionado à **keys** e é criada uma constante **value** com o valor da chamada para **validateProps()**. Depois é checado se o *Vue* está em perfil de desenvolvimento, como não é relevante, é observado apenas a cláusula do **else**.

```
else {
  defineReactive(props, key, value, undefined, true)
}
if (!(key in vm)) {
  proxy(vm, `_${key}`, key)
}
```

A função **defineReactive()** é utilizada para transformar os valores **key** de **props** em propriedades reativas. Depois é verificado se o valor **key** está definido em **vm**. Caso não esteja, é utilizado **proxy()** para definir o valor **key** em **__props** no objeto **vm**.

```
initSetup(vm)
```

Em seguida, ao voltar para **initState()**, há a chamada para a função **initSetup()**.

```
export function initSetup(vm: Component) {
  const options = vm.$options
  const setup = options.setup
```

A função **initSetup()** é definida com um elemento **vm** de tipo **Component**. É definida uma constante **options** com valor de **vm.\$options**, é definido uma constante **setup** com o valor de **options.setup** também.

```
if (setup) {
  const ctx = (vm._setupContext = createSetupContext(vm))

  setCurrentInstance(vm)
  pushTarget()
```

É definida uma estrutura condicional que checa se **setup** é um valor definido. Caso seja, é criado uma constante **ctx** com o valor de **createSetupContext()**, que também é definido para **vm._setupContext**. Depois é utilizado a função de **setCurrentInstance()** para definir o valor de **vm** como a instância atual a ser avaliada. É chamado a função **pushTarget()** para adicionar o valor de **vm** como o topo da *stack* de alvos.

```
const setupResult = invokeWithErrorHandling(
  setup,
  null,
  [vm._props || shallowReactive({}), ctx],
  vm,
  `setup`
)
```

É chamado então a função **invokeWithErrorHandling()**, que devolve à **setupResult** o valor da função passada por **setup**.

```
popTarget()
setCurrentInstance()
```

O novo alvo é definido como o valor do alvo anterior e o alvo atual é removido da instância da lista de alvos.

```
if (isFunction(setupResult)) {
  options.render = setupResult
```

Depois é verificado se **setupResult** é uma função, e caso seja, o valor de **options.render** é definido com o valor de **setupResult**.

```
else if (isObject(setupResult))
```

Caso não seja, é verificado se **setupResult** é de tipo objeto. Depois há uma checagem se o *Vue* está em perfil de desenvolvimento, e se **setupResult** é uma instância de **VNode** para emitir um aviso.

```

vm._setupState = setupResult
  if (!setupResult.__sfc) {
    for (const key in setupResult) {
      if (!isReserved(key)) {
        proxyWithRefUnwrap(vm, setupResult, key)
      } else if (__DEV__) {
        warn(`Avoid using variables that start \
          with _ or $ in setup().`)
      }
    }
  }
}

```

Depois é definido o valor de **vm._setupState** com o valor de **setupResult**. Depois é verificado e **setupResult.__sfc** é um valor definido, caso seja, é feita uma iteração pelas chaves de **setupResult**, é checado se a chave é reservada e não seja, é utilizado o valor de **setupResult** para definir o valor desta chave em **vm** por meio da função **proxyWithRefUnwrap()**.

```

else {
  const proxy = (vm._setupProxy = {})
  for (const key in setupResult) {
    if (key !== '__sfc') {
      proxyWithRefUnwrap(proxy, setupResult, key)
    }
  }
}

```

Caso **setupResult.__sfc** seja definido, é definido uma constante **proxy** com o valor de um objeto vazio que também é definido no valor de **vm._setupProxy()**. Depois é feita uma iteração pelos valores chave de **setupResult**, e é checado se essa chave tem o valor de **'__sfc'**, e caso tenha, é utilizado **proxyWithRefUnwrap()** para definir a chave correspondente do valor de **proxy**.

```

else if (__DEV__ && setupResult !== undefined) {
  warn(
    `setup() should return an object. Received: ${
      setupResult === null ? 'null' : typeof setupResult
    }`
  )
}

```

Depois é checado se o *Vue* está em perfil de desenvolvimento, para alertar possíveis erros ou condições não ideais de tipos para **setupResult**.

```
if (opts.methods) initMethods(vm, opts.methods)
```

Voltando para **initState()**, é checado se **options** possui uma propriedade **methods** definida, e caso haja, é chamada a função **initMethods()**.

```
function initMethods(vm: Component, methods: Object) {
  const props = vm.$options.props
  for (const key in methods) {
    if (__DEV__) {
      if (typeof methods[key] !== 'function') {
        warn(
          `Method "${key}" has type "${typeof methods[key]}" in the component definition. ` +
          `Did you reference the function correctly?`,
          vm
        )
      }
      if (props && hasOwn(props, key)) {
        warn(`Method "${key}" has already been defined as a prop.`, vm)
      }
      if (key in vm && isReserved(key)) {
        warn(
          `Method "${key}" conflicts with an existing Vue instance method. ` +
          `Avoid defining component methods that start with _ or $.`
        )
      }
      vm[key] = typeof methods[key] !== 'function' ?
        noop : bind(methods[key], vm)
    }
  }
}
```

A função **initMethods()** é definida com um **vm** de tipo **Component** e um **methods** de tipo **Object** como parâmetros. É criado uma constante **props** que recebe o valor de

vm.\$options.props, e é começado uma iteração pelas chaves de valor em **methods** usando **key** como variável. Caso o perfil de desenvolvimento seja confirmado, é feita uma série de checagens quanto ao tipo e estado dos dados tratados, como se a instância **key** em **methods** não é uma função, se **props** é definido e possui uma propriedade **key**, se **key** é definido em **vm** e se **key** é uma propriedade reservada. Ao final de **initMethods()** é assinalado a chave **key** de **vm** um de dois valores:

- Se a instância de **key** em **methods** não é uma função, o valor de **noop** é retornado para **vm[key]**.
- Se a instância de **key** em **methods** é uma função, é utilizado **bind()** para atribuir esta função para **vm**.

```
export const bind = Function.prototype.bind ? nativeBind : polyfillBind
```

A função **bind()** é específica para cada plataforma, e na sua definição, retorna os valores de uma função **nativeBind()** ou de uma função **polyfillBind()**.

```
if (opts.data) {
  initData(vm)
}
```

Ao voltar para a função **initState()** é checado se o valor **data** em **opts** é definido. Caso seja, é chamada a função **initData()**.

```
function initData(vm: Component)
```

A função **initData()** é definida com um elemento **vm** de tipo **Component** como parâmetro.

```
let data: any = vm.$options.data
data = vm._data = isFunction(data) ? getData(data, vm) : data || {}
if (!isPlainObject(data)) {
  data = {}
  __DEV__ &&
  warn(
    'data functions should return an object\n' +
    'https://v2.vuejs.org/v2/guide/components.html',
    vm
  )
}
```

É então criado uma variável **data** com o valor de **vm.\$options.data**, e é atribuído o valor de **getData()** caso **data** seja uma função ou o valor de **data** caso não seja uma

função, e ainda, caso **data** não seja um valor definido, é atribuído o valor de um objeto vazio. O valor que é atribuído para **data** também é atribuído para **vm._data**. Depois é checado se **data** é um objeto plano, ou seja, um objeto sem valores, com a chamada para função **isPlainObject()**. Caso ele não seja um objeto plano, **data** recebe um valor de objeto vazio, e caso o *Vue* esteja em perfil de desenvolvimento, é chamado um aviso sobre esse valor.

```
const keys = Object.keys(data)
const props = vm.$options.props
const methods = vm.$options.methods
let i = keys.length
```

Depois, é criado uma constante **keys** com o valor das chaves de data, uma constante **props** com o valor de **vm.\$options.props**, uma constante **methods** com o valor de **vm.\$options.methods** e uma variável **i** com o valor de **keys.length**.

```
while (i--) {
  const key = keys[i]
  if (__DEV__) {
    if (methods && hasOwn(methods, key)) {
      warn(`Method "${key}" has already been defined \
as a data property.`, vm)
    }
  }
  if (props && hasOwn(props, key)) {
    __DEV__ &&
    warn(
      `The data property "${key}" is already \
declared as a prop. ` +
      `Use prop default value instead.`,
      vm
    )
  } else if (!isReserved(key)) {
    proxy(vm, `_data`, key)
  }
}
```

É então criada uma estrutura de repetição, que dura até o valor de **keys.length** chegar a 0. É definido uma constante **key** que recebe o valor de **keys[i]**, o que significa que as chaves serão checadas do fim ao começo. Então é feita uma série de checagens:

- Se **methods** é definido e se **key** está definido em **methods**.
- Se **props** é definido e se **key** está definido em **props**.

- Caso **key** não esteja definido em **props**, se a chave **key** é reservada.

Para o primeiro e segundo caso, há apenas o uso de avisos caso o *Vue* esteja em perfil de desenvolvimento, para o último caso, é usada a função **proxy()** para instanciar o valor **key** em **vm**.

```
const ob = observe(data)
ob && ob.vmCount++
```

Ao final da função, e fora da estrutura de repetição, é criado um vigia para o valor de **data**, e é checado se o *Vue* conseguiu criar esse vigia com sucesso, se sim, o valor de **vmCount** é incrementado, caso contrário, nada acontece.

```
else {
  const ob = observe((vm._data = {}))
  ob && ob.vmCount++
}
```

Ao voltar para **initState()**, caso **opts.data** não seja definido, ainda na mesma estrutura condicional anterior, é atribuído ao valor de **vm._data** um objeto vazio, e há uma tentativa de criar um vigia para esse valor através de **observe()**, depois há a checagem de sucesso, e caso seja sucedido, **vmCount** é incrementado.

```
if (opts.computed) initComputed(vm, opts.computed)
```

Após isso, é feita uma checagem da definição de valor de **opts.computed**. Caso seja definido, é chamada a função **initComputed()**.

```
const computedWatcherOptions = { lazy: true }
```

```
function initComputed(vm: Component, computed: Object)
```

A definição de **initComputed()** começa com a instanciação de um valor **vm** de tipo **Component** e um valor **computed** de tipo **Object**. É também definido uma constante global **computedWatcherOptions** que recebe um objeto com a definição inicial de uma propriedade **lazy** com valor **true**.

```
const watchers = (vm._computedWatchers = Object.create(null))
const isSSR = isServerRendering()
```

A função começa criando uma constante **watchers** com o valor de um objeto nulo, cujo valor também é atribuído à **vm._computedWatchers**, e uma constante **isSSR** que recebe o valor de **isServerRendering()**.

```

let _isServer
export const isServerRendering = () => {
  if (_isServer === undefined) {
    if (!inBrowser && typeof global !== 'undefined') {
      _isServer =
        global['process'] && global['process'].env.VUE_ENV === 'server'
    } else {
      _isServer = false
    }
  }
  return _isServer
}

```

A função **isServerRendering()** faz uma avaliação lenta de várias variáveis e parâmetros de inicialização do *Vue*, pois o *Vue* pode acabar sendo inicializado e chamar esta função antes que a definição do renderizador seja feita.

```

for (const key in computed) {
  const userDef = computed[key]
  const getter = isFunction(userDef) ? userDef : userDef.get

```

Voltando para **initComputed()**, é definido uma iteração pelas chaves de **computed** usando **key** como nome. É então criado uma constante **userDef** com o valor de **computed[key]**, e uma constante **getter** com a avaliação do tipo de **userDef**, se é uma função o valor de **userDef** é retornado, caso seja qualquer outro tipo, **userDef.get** é retornado.

```

if (!isSSR) {
  watchers[key] = new Watcher(
    vm,
    getter || noop,
    noop,
    computedWatcherOptions
  )
}

```

É checado então se renderização por parte do servidor está ativado, caso não esteja, é criado uma instância de um vigia para cada chave existente com o valor de **ComputedWatcherOptions**.

```

if (!(key in vm)) {
  defineComputed(vm, key, userDef)

```

```

} else if (__DEV__) {
  if (key in vm.$data) {
    warn(`The computed property "${key}" \
is already defined in data.`, vm)
  } else if (vm.$options.props && key in \
vm.$options.props) {
    warn(`The computed property "${key}" \
is already defined as a prop.`, vm)
  } else if (vm.$options.methods && key \
in vm.$options.methods) {
    warn(
      `The computed property "${key}" is \
already defined as a method.`,
      vm
    )
  }
}

```

Após isso, é checado se a chave está definida em **vm**, caso não esteja, é chamado a função **defineComputed()**, após essa checagem, alguns avisos são feitos caso o *Vue* esteja em perfil de desenvolvimento.

```

export function defineComputed(
  target: any,
  key: string,
  userDef: Record<string, any> | (() => any)
) {
  const shouldCache = !isServerRendering()
  if (isFunction(userDef)) {
    sharedPropertyDefinition.get = shouldCache
      ? createComputedGetter(key)
      : createGetterInvoker(userDef)
    sharedPropertyDefinition.set = noop
  }
}

```

A função **defineComputed()** é definida com uma variável **target** de tipo **any**, uma variável **key** de tipo **string** e uma variável **userDef** de tipo **Record** com chave **string** e valor **any** ou uma função que retorna **any**.

A função começa criando uma constante **shouldCache** com o resultado da negação de **isServerRendering()**. Depois é verificado se o valor de **userDef** é uma função. Caso seja, é definido o valor **get** de **sharedPropertyDefinition** com a avaliação de **shouldCache**,

se for verdadeiro é utilizado **createComputedGetter()** para obter o valor, se for falso, é utilizado **createGetterInvoker()** para obter o valor. Ao definir a propriedade **get** de **sharedPropertyDefinition**, é então definido a propriedade **set** com o valor **noop**.

```
function createGetterInvoker(fn) {
  return function computedGetter() {
    return fn.call(this, this)
  }
}
```

A função **createGetterInvoker()** recebe uma função **fn** como parâmetro e retorna o valor da chamada para essa função através de **fn.call()**.

```
else {
  sharedPropertyDefinition.get = userDef.get
    ? shouldCache && userDef.cache !== false
      ? createComputedGetter(key)
      : createGetterInvoker(userDef.get)
    : noop
  sharedPropertyDefinition.set = userDef.set || noop
}
```

Na mesma estrutura condicional, caso **userDef** não seja uma função, é definido o valor de **sharedPropertyDefinition.get** com o valor da seguinte expressão: se **userDef.get** é definido, é checado se **shouldCache** é definido e se **userDef.cache** é diferente de falso. Caso seja verdade, é retornado **createComputedGetter()** com o valor de **key**, caso seja falso é retornado **createGetterInvoker()** com o valor de **userDef.get**. Caso **userDef.get** não seja definido, ainda na primeira expressão, **noop** é retornado como valor. Após essa definição, o valor de **sharedPropertyDefinition.set** é definido como **userDef.set** ou **noop** caso o anterior seja nulo ou indefinido.

Object.defineProperty(target, key, sharedPropertyDefinition)

Ao final de **defineComputed()**, é definida a propriedade **key** para o elemento alvo definido em **target**. Esta função marca o final de **defineComputed()**, que marca também o final da função **initComputed()**.

```
if (opts.watch && opts.watch !== nativeWatch) {
  initWatch(vm, opts.watch)
}
```

Ao voltar para **initState()**, a última parte do código define uma estrutura condicional que checa se **opts.watch** é definido e se **opts.watch** tem um valor diferente de **nativeWatch**. Caso isso seja verdadeiro, a função **initWatch()** é executada, caso contrário, a função é encerrada.

```
export const nativeWatch = {}.watch
```

A constante **nativeWatch** é referente a uma constante de ambiente do *Vue*, onde, em ambientes controlados pelo motor de renderização do *Firefox*, existe uma função nativa para vigiar mudanças em objetos.

```
function initWatch(vm: Component, watch: Object) {
  for (const key in watch) {
    const handler = watch[key]
    if (isArray(handler)) {
      for (let i = 0; i < handler.length; i++) {
        createWatcher(vm, key, handler[i])
      }
    } else {
      createWatcher(vm, key, handler)
    }
  }
}
```

A função **initWatch()** é definida com uma variável **vm** de tipo **Component** e uma variável **watch** de tipo **Object**. No início da função, há a declaração de um iterador sobre os elementos de **watch** usando uma constante **key** para capturar os valores, e uma declaração de constante **handler** que recebe o valor da instância de **key** em **watcher**. Depois é feita a checagem se **handler** é um **Array**.

Caso seja, é feita uma iteração sobre os elementos de **handler**, e vigias são criados para os elementos de **handler** utilizando **createWatcher()**. Caso **handler** não seja um **Array**, a função **createWatcher()** é executada apenas uma vez para **handler**.

```
function createWatcher(
  vm: Component,
  expOrFn: string | (() => any),
  handler: any,
  options?: Object
) {
  if (isPlainObject(handler)) {
    options = handler
```

```

    handler = handler.handler
  }
  if (typeof handler === 'string') {
    handler = vm[handler]
  }
  return vm.$watch(expOrFn, handler, options)
}

```

A função **createWatcher()**, define um **vm** de tipo **Component**, um **expOrFn** de tipo **string** ou função, um **handler** de tipo **any** e um **options** de tipo **Object**. A função começa com uma estrutura condicional que checa se **handler** é um objeto pleno através da função **isPlainObject()** e, caso seja, o valor de **handler** é atribuído à **options** e o valor de **handler** se torna a sua própria propriedade **handler**.

Após isso, mais uma checagem é feita para saber se o tipo de **handler** é uma **string**, e caso seja, é definido o valor de **handler** como a chave de **handler** em **vm**. Ao final da função, é retornado o valor da chamada de **vm.\$watch()**.

```

Vue.prototype.$watch = function (
  expOrFn: string | (() => any),
  cb: any,
  options?: Record<string, any>
): Function

```

O método **vm.\$watch** é na verdade definido a partir de **Vue.prototype.\$watch()**. A definição de **\$watch()** toma como parâmetros uma variável **expOrFn** que tem tipo **string** ou uma função com retorno de tipo **any**, uma variável **cb** de tipo **any** e uma variável **options** de tipo **Record** com chave **string** e valor **any**. O tipo de retorno para **\$watch()** é uma função.

```

const vm: Component = this
if (isPlainObject(cb)) {
  return createWatcher(vm, expOrFn, cb, options)
}
options = options || {}
options.user = true
const watcher = new Watcher(vm, expOrFn, cb, options)

```

A função se inicia com a criação de uma constante **vm** com o valor de **this**, que no caso é o elemento a chamar **\$watch()**. Depois é definida uma estrutura condicional que checa se **cb** é um objeto pleno a partir da função **isPlainObject()**, caso seja a função termina aqui com o retorno do valor para **createWatcher()**, a mesma função que inicialmente chama **\$watch()**.

A partir desta recursão ocorre um destrinchamento do **handler** para esta instância do componente. Com toda chamada para **\$watch()**, é feita uma verificação do tipo de **cb**. Se **cb** for um objeto pleno, é feita a chamada para **createWatcher()**, que então altera os valores de **handler** e **options**, e faz uma nova chamada para **\$watch()**.

Após a execução desta checagem, caso **cb** não seja um objeto pleno, é checado se o valor de **options** é nulo ou indefinido, definindo então o seu próprio valor para um objeto vazio caso seja. Então o valor da propriedade **user** de **options** é definido como verdadeiro, e é criado um vigia para o componente através da chamada de **new Watcher()**.

```
if (options.immediate) {
  const info = `callback for immediate watcher "${watcher.expression}"`
  pushTarget()
  invokeWithErrorHandling(cb, vm, [watcher.value], vm, info)
  popTarget()
}
```

É feito então uma checagem do valor da propriedade **immediate** de **options**. Caso seja verdadeiro, é criada uma constante **info** com uma **string** para enviar uma mensagem de aviso, é utilizado **pushTarget()** para bloquear o valor de **Dep.target** com um nulo, é chamado então a função **invokeWithErrorHandling()** que executará a função em **cb** e por fim, é chamado **popTarget()** para remover o valor nulo de **Dep.target**.

```
return function unwatchFn() {
  watcher.teardown()
}
```

Ao final de **\$watch()**, é definida uma função **unwatchFn()**, que é retornada como valor por **\$watch()**. O corpo de **unwatchFn()** chama apenas o método **teardown()** de **Watcher** com a instância presente em **watcher**.

```
teardown() {
  if (this.vm && !this.vm._isBeingDestroyed) {
    remove(this.vm._scope.effects, this)
  }
  if (this.active) {
    let i = this.deps.length
    while (i--) {
      this.deps[i].removeSub(this)
    }
    this.active = false
    if (this.onStop) {
```

```

        this.onStop()
    }
}
}

```

A função **teardown()** remove todos os inscritos do vigia que chamou esta função, se define como inativo, e chama as respectivas funções de parada com **this.onStop()**. Esta chamada marca o fim da execução de atividades de **initState()**, concluindo então a inicialização de estados do *Vue*.

A.2.6 Inicialização de Fornecimento

Esta função extrai fornecedores das opções e chama elas no componente indicado.

```

export function initProvide(vm: Component) {
    const provideOption = vm.$options.provide

```

A função **initProvide()** começa com a instanciação de um parâmetro **vm** de tipo **Component**. É então criado uma constante **provideOptions** que recebe os valores de **vm.\$options.provide**.

```

provide?: Record<string | symbol, any> |
(() => Record<string | symbol, any>)

```

A propriedade **provide** é definida como um **Record** que recebe uma chave **string** ou **symbol** e uma chave de tipo **any**, ou recebe uma função que retorna este valor anterior.

```

if (provideOption) {
    const provided = isFunction(provideOption)
        ? provideOption.call(vm)
        : provideOption
    if (!isObject(provided)) {
        return
    }
}

```

É então definida uma estrutura condicional que checa se o valor de **provideOptions** é definido, caso seja, é criada uma constante **provided** que recebe o valor a partir da validação de **provideOption** como função: caso seja, o valor de **provideOption.call()** é retornado, caso contrário o valor de **provideOption** é retornado. Caso o valor de **provided** não seja um objeto durante a próxima estrutura condicional, a função termina aqui.

```

const source = resolveProvided(vm)

```

É então criada a constante **source** com o resultado de **resolveProvided()**.

```
export function resolveProvided(vm: Component): Record<string, any> {
  const existing = vm._provided
  const parentProvides = vm.$parent && vm.$parent._provided
  if (parentProvides === existing) {
    return (vm._provided = Object.create(parentProvides))
  } else {
    return existing
  }
}
```

A função **resolveProvided()** é definida com uma constante **vm** de tipo **component** e retorna um valor **Record** com chave **string** e valor **any**. É então criada uma constante **existing** com o valor de **vm._provided**, e uma constante **parentProvides** que recebe o valor de **vm.\$parent._provided** caso **vm.\$parent** seja definido e não nulo.

Depois é feita uma estrutura condicional que checa se o valor de **parentProvides** e **existing** é estritamente igual, e se for, define o valor de **vm._provided** para um objeto com as propriedades de **parentProvides**. Caso contrário, o valor de **existing** é retornado como valor.

```
const keys = hasSymbol ?
Reflect.ownKeys(provided) :
Object.keys(provided)
```

É então criada uma constante **keys** com o valor da checagem de **hasSymbol**: se for verdadeiro, o valor de **Reflect.ownKeys()** é atribuído, se for falso, o valor de **Object.Keys()** é atribuído.

```
for (let i = 0; i < keys.length; i++) {
  const key = keys[i]
  Object.defineProperty(
    source,
    key,
    Object.getOwnPropertyDescriptor(provided, key)!
  )
}
```

Ao final da função, é criada uma iteração pelos valores de **keys**, e é definido uma constante **key**. Então, para cada objeto, é definido uma propriedade **key** no objeto **source** com os valores de **Object.getOwnPropertyDescriptor()**, finalizando assim a inicialização de fornecimento do *Vue*.

A.3 Mixin de Estados

Define os valores de:

- **dataDef** e **propsDef** e seus métodos **get** e **set**.
- **dataDef** como propriedade de **Vue.prototype** em formato de **\$data**
- **propsDef** como propriedade de **Vue.prototype** em formato de **\$props**
- **Vue.prototype.\$set**, **Vue.prototype.\$delete** e **Vue.prototype.\$watch**

```
export function stateMixin(Vue: typeof Component) {
```

O início do *Mixin* de estados é definido com um parâmetro **Vue**, de tipo **Component**.

```
  const dataDef: any = {}
  dataDef.get = function () {
    return this._data
  }
  const propsDef: any = {}
  propsDef.get = function () {
    return this._props
  }
```

São então criadas duas constantes, **dataDef** e **propsDef** com os valores dos dados e dos adereços deste objeto, no caso de *Vue*, a partir de **this._data** e **this._props**.

```
  if (__DEV__) {
    dataDef.set = function () {
      warn(
        'Avoid replacing instance root $data. ' +
        'Use nested data properties instead.',
        this
      )
    }
    propsDef.set = function () {
      warn(`$props is readonly.`, this)
    }
  }
```

Caso o *Vue* esteja em perfil de desenvolvimento, o fluxo do código irá criar funções **set** para **propsDef** e para **dataDef** com avisos.

```
Object.defineProperty(Vue.prototype, '$data', dataDef)
Object.defineProperty(Vue.prototype, '$props', propsDef)
```

É então definido nas propriedades **\$data** e **\$props** de **Vue.prototype** os valores de **dataDef** e **dataProps** respectivamente.

```
Vue.prototype.$set = set
Vue.prototype.$delete = del
```

Então as propriedades **\$set** e **\$delete** são definidas com as funções **set()** e **delete()**, respectivamente.

```
export function set<T>(array: T[], key: number, value: T): T
export function set<T>(object: object, key: string | number, value: T): T
export function set(
  target: any[] | Record<string, any>,
  key: any,
  val: any
): any
```

A função **set** é definida com 3 possíveis estruturas diferentes:

- Com um tipo genérico **T**, com uma variável **array** de tipo **Array** deste tipo **T**, uma variável **key** de tipo **number** e uma variável **value** de tipo **T**, retornando então um tipo **T**.
- Com um tipo genérico **T**, com uma variável **object** de tipo objeto, uma variável **key** de tipo **string** ou **number** e uma variável **value** de tipo **T**, retornando **T** como valor.
- Sem um tipo genérico **T**, com uma variável **target** de tipo **Array** que agrega valores de tipo **any** ou um **Record** com chave **string** e valor **any**, uma variável **key** de tipo **any** e uma variável **val** de tipo **any**, retornando então um tipo **any**.

```
if (isReadonly(target)) {
  __DEV__ && warn(`Set operation on key \
    "${key}" failed: target is readonly.`)
  return
}
```

A função se inicia com uma checagem da capacidade de escrita acima de **target**, caso não seja possível adicionar valores a **target**, a função se encerra aqui.

```
const ob = (target as any).__ob__
if (isArray(target) && isValidArrayIndex(key)) {
  target.length = Math.max(target.length, key)
  target.splice(key, 1, val)

  if (ob && !ob.shallow && ob.mock) {
```

```

    observe(val, false, true)
  }
  return val
}

```

Ocorre então a criação de uma constante **ob** com a propriedade de **__ob__** do valor **target**. A variável **target** é definida como **any** em razão das 3 definições possíveis da função. É então checado se **target** é um **Array** e se **key** é um índice válido. Caso essa expressão seja verdadeira, é definido a propriedade **length** de **target** com o valor da chamada para **Math.max()** com os valores de **target.length** e **key**. Depois é executada a função **splice()** sobre os itens de **target**. A função **splice()** usa **key** como início da lista, o valor 1 como quantidade de itens para remover e **val** como o valor a substituir o item removido. Após essa execução, é checado se **ob** é definido, se não é um observador raso e se esta instância do *Vue* está renderizando os *templates* no servidor. Caso essa expressão seja avaliada como verdadeira, é criado um observador para o valor de **val** e esse valor é retornado pela função.

```

if (key in target && !(key in Object.prototype)) {
  target[key] = val
  return val
}

```

É então definida uma estrutura condicional que checa se o valor de **target** está definido em **key** e se **key** não está definido em **Object.prototype**. Caso a expressão seja avaliada como verdadeira, o valor da chave **key** de **target** é definido com o valor de **val** e esse valor é retornado pela função.

```

if ((target as any)._isVue || (ob && ob.vmCount)) {
  __DEV__ &&
    warn(
      'Avoid adding reactive properties \
      to a Vue instance or its root $data ' +
      'at runtime - declare it upfront\
      in the data option.'
    )
  return val
}
if (!ob) {
  target[key] = val
  return val
}

```

Após essa checagem, mais duas estruturas condicionais são criadas. Uma para verificar se o valor de **target** é o componente principal do *Vue* ou se os valores de **ob** e **ob.vmCount** são definidos retornando o valor de **val** e adicionalmente enviando um aviso caso o *Vue* esteja em perfil de desenvolvimento, e outra para avaliar se o valor de **ob** é indefinido, definindo o valor da chave **key** em **target** com o valor de **val**, e retornando este valor pela função.

```
defineReactive(ob.value, key, val, undefined, ob.shallow, ob.mock)
```

Após todas essas checagens, caso nenhum dos pontos de retorno sejam atingidos, uma chamada para a função **defineReactive()** é feita, definindo **ob.value** como uma propriedade reativa.

```
if (__DEV__) {
  ob.dep.notify({
    type: TriggerOpTypes.ADD,
    target: target,
    key,
    newValue: val,
    oldValue: undefined
  })
} else {
  ob.dep.notify()
}

return val
```

Após a execução de **defineReactive()**, é checado se o *Vue* está em perfil de desenvolvimento. As duas possibilidades da estrutura condicional executam **ob.dep.notify()** para avisar os inscritos desta dependência, porém se o perfil de desenvolvimento é confirmado, é passado mais informações para depuração do desenvolvedor. Após essa chamada, a função retorna o valor de **val** e a função **set()** é encerrada.

```
export function del<T>(array: T[], key: number): void
export function del(object: object, key: string | number): void
export function del(target: any[] | object, key: any)
```

Analisando a função **del()**, é notável que assim como **set()** possui 3 possíveis implementações:

- Com a definição de um tipo genérico **T**, a definição de uma variável **array** com o tipo de **Array** que engloba valores do tipo **T** e uma variável **key** de tipo **number**.
- Com a definição de uma variável **object** de tipo objeto e uma variável **key** de tipo **string** ou **number**.

- Com a definição de uma variável **target** de tipo **Array** que engloba valores de tipo **any** ou um objeto, e uma variável **key** de tipo **any**.

```
if (isArray(target) && isValidArrayIndex(key)) {
  target.splice(key, 1)
  return
}
```

No começo da função, é definido uma estrutura condicional que checa se **target** é um **Array** válido e se **key** é um valor válido para o índice. Caso seja, a operação de **splice()** é executada em **target**, utilizando apenas **key** como o índice a ser removido, e o valor 1 para delimitar a quantidade de itens a serem removidos, sem nenhum valor para substituir o item removido, e a função é finalizada aqui.

```
const ob = (target as any).__ob__
if ((target as any)._isVue || (ob && ob.vmCount)) {
  __DEV__ &&
  warn(
    'Avoid deleting properties on a \
    Vue instance or its root $data ' +
    '- just set it to null.'
  )
  return
}
```

É então definida a constante **ob** que guarda o valor de **target.__ob__**. Depois é feita uma estrutura condicional que checa se o valor de **target** é a instância do *Vue* ou se **ob** e **ob.vmCount** são definidos, e encerra a função aqui, caso essa expressão seja verdadeira.

```
if (isReadonly(target)) {
  __DEV__ &&
  warn(`Delete operation on key "${key}" failed: target is readonly.`)
  return
}
```

Depois é checado se **target** é um valor apenas de leitura, e caso seja, a função é encerrada.

```
if (!hasOwn(target, key)) {
  return
}
```

Depois disso, é checado se **target** possui a chave **key**, e caso não possua, a função é encerrada.

```
delete target[key]
if (!ob) {
  return
}
```

Ao final de todas essas checagens, caso o código chegue a este ponto, a chave **key** de **target** é removida, e se **ob** for definido, a função é encerrada.

```
if (__DEV__) {
  ob.dep.notify({
    type: TriggerOpTypes.DELETE,
    target: target,
    key
  })
} else {
  ob.dep.notify()
}
```

Ao final da função, é checado se o perfil do *Vue* está em desenvolvimento. Ambas as possibilidades de execução da estrutura condicional resultam na execução de **ob.dep.notify()**, porém caso o *Vue* esteja em perfil de desenvolvimento, informações adicionais são informadas para depuração. Este trecho do código marca o final da função **del()**.

```
Vue.prototype.$watch = function (
  expOrFn: string | (() => any),
  cb: any,
  options?: Record<string, any>
): Function {
```

Ao final de **stateMixin()**, é definido **Vue.prototype.\$watch()**. Esta função foi explicada na seção “Inicialização de Estados”, portanto, não será elaborada aqui novamente.

A.4 Mixin de Eventos

Define os valores de:

- **Vue.prototype.\$on**
- **Vue.prototype.\$once**
- **Vue.prototype.\$off**

- **Vue.prototype.\$emit**

Todas essas propriedades são utilizadas internamente pelo Vue, e manipuladas indiretamente pelo desenvolvedor ao fazer uma manipulação de eventos elegante.

```
export function eventsMixin(Vue: typeof Component) {
```

A definição do *Mixin* de eventos é feita com a definição de uma variável **Vue** de tipo **Component**.

```
const hookRE = /^hook:/
Vue.prototype.$on = function (
  event: string | Array<string>,
  fn: Function
): Component
```

É então definida uma constante **hookRE** que recebe o valor de uma expressão regular `“/hook:”`. Essa expressão será usada para checar o nome de **hooks** e validá-los. É então feita a definição do método **\$on** de **Vue.prototype**, usando uma variável **event** de tipo **string** ou **Array** de strings e uma variável **fn** de tipo **Function** como parâmetro e retornando um **Component**.

```
const vm: Component = this
if (isArray(event)) {
  for (let i = 0, l = event.length; i < l; i++) {
    vm.$on(event[i], fn)
  }
} else {
  ;(vm._events[event] || (vm._events[event] = [])).push(fn)
  if (hookRE.test(event)) {
    vm._hasHookEvent = true
  }
}
return vm
```

No início da função é criada uma constante **vm** de tipo **Component** que recebe o valor de **this**, no caso o objeto que chama o método **\$on()**. Depois é checado se o valor de **event** tem o valor de um **Array**, criando uma iteração sobre os valores de **event** e os usando como parâmetro para uma chamada recursiva de **\$on()**. Caso **event** não seja um **Array** é checado se o valor da propriedade **_events** de **vm** é definido e caso seja, o valor da chave **event** em **_events** é removido, adicionando um **Array** vazio no lugar, e ao final, o valor de **fn** é adicionado ao final de **_events** com o método **push()**. Depois é checado

se o valor de **event** é validado pela expressão regular **hookRE**, e caso seja, o valor da propriedade **_hasHookEvent** é definido como verdadeiro, e o valor de **vm** é retornado pela função ao final.

```
Vue.prototype.$once = function (event: string, fn: Function): Component {
  const vm: Component = this
  function on() {
    vm.$off(event, on)
    fn.apply(vm, arguments)
  }
  on.fn = fn
  vm.$on(event, on)
  return vm
}
```

Após a definição de **\$on()**, é feita a definição de **\$once()**. Na sua concepção, é definido o valor de uma variável **event** de tipo **string** e uma variável **fn** de tipo **Function**, e o tipo de retorno da função é um **Component**. No início de **\$once()** é criado uma constante **vm** com o valor do **Component** que chamou este método. Depois é definida uma função **on()**, que executa o método **\$off()** em **vm** e aplica o valor da função **fn** como método de **vm** e aplica o valor de **arguments** como o valor de argumentos desta função. Após essa definição de função, o valor da propriedade **fn** da função **on()** é definido com o valor de **fn** e é feita uma chamada para o método **\$on()** de **vm**, usando **event** e **on()** como parâmetros. Ao final de **\$once()**, o valor de **vm** é retornado pela função.

```
Vue.prototype.$off = function (
  event?: string | Array<string>,
  fn?: Function
): Component
```

Após a definição de **\$once()**, ocorre a definição de **\$off()**, com uma variável **event** de tipo **string** ou uma **Array** de string, e uma variável **fn** de tipo **Function** como parâmetros, e retorna como valor um tipo **Component**.

```
const vm: Component = this
if (!arguments.length) {
  vm._events = Object.create(null)
  return vm
}
```

É então criado uma constante **vm** com o valor do **Component** que chamou esta função, e é checado se o tamanho dos argumentos de inicialização do *Vue* é definido. Caso

não seja, o valor da propriedade **_events** de **vm** é definido com um valor nulo e o valor de **vm** é retornado aqui.

```
if (isArray(event)) {
  for (let i = 0, l = event.length; i < l; i++) {
    vm.$off(event[i], fn)
  }
  return vm
}
```

Em seguida, é verificado se o valor da chamada para **isArray()** com o valor de **event** é verdadeiro, e caso seja, é criada uma iteração que executa o método **\$off** de **vm** com cada um dos valores de **event**, e o valor de **vm** é retornado ao final.

```
const cbs = vm._events[event!]
if (!cbs) {
  return vm
}
if (!fn) {
  vm._events[event!] = null
  return vm
}
```

Após essa checagem, é criada uma constante **cbs** com o valor da chave **event** na propriedade **_events** de **vm** contanto que **event** não seja nulo. É então checado se o valor de **cbs** é definido, e caso não seja o valor de **vm** é retornado. Depois é verificado se o valor de **fn** é definido, e caso seja, o valor da chave **event** de **vm._events** é definido com um valor nulo e o valor de **vm** é retornado.

```
let cb
let i = cbs.length
while (i--) {
  cb = cbs[i]
  if (cb === fn || cb.fn === fn) {
    cbs.splice(i, 1)
    break
  }
}
return vm
```

Após essas duas checagens é criado uma variável **cb**, e uma outra variável **i** com o valor do tamanho de **cbs**. Depois é criada uma estrutura de repetição que dura enquanto

o valor de **i** for maior que zero, e a cada iteração seu valor é decrementado em 1. Dentro da estrutura, o valor de **cb** é definido com o índice **i** de **cbs** e é checado se **cb** é estritamente igual a **fn** ou se a propriedade **fn** de **cb** é estritamente igual a **fn**. Caso qualquer uma dessas afirmações seja verdadeira, o valor de do índice **i** em **cbs** é removido e a estrutura de repetição passa para a próxima iteração. Ao final de **\$off()** o valor de **vm** é retornado como valor.

```
Vue.prototype.$emit = function (event: string): Component {
```

A última definição dentro do Mixin de eventos é a de **\$emit()**, que recebe como parâmetro uma variável **event** de tipo **string** e retorna um valor de tipo **Component**.

```
const vm: Component = this
if (__DEV__) {
  const lowerCaseEvent = event.toLowerCase()
  if (lowerCaseEvent !== event && vm._events[lowerCaseEvent]) {
    tip(
      `Event "${lowerCaseEvent}" is emitted in component ` +
      `${formatComponentName(
        vm
      )} but the handler is registered for ` +
      `"${event}"`. ` +
      `Note that HTML attributes are ` +
      `case-insensitive and you cannot use ` +
      `v-on to listen to camelCase events ` +
      `when using in-DOM templates. ` +
      `You should probably use ` +
      `"${hyphenate(
        event
      )}" instead of "${event}"`. `
    )
  }
}
```

É criada uma constante **vm** com o valor do **Component** que chamou esta função, e é verificado se o *Vue* está em perfil de desenvolvimento, para checar o valor de **event** e emitir avisos e dicas para o desenvolvedor caso seja necessário.

```
let cbs = vm._events[event]
if (cbs) {
  cbs = cbs.length > 1 ? toArray(cbs) : cbs
```

```

const args = toArray(arguments, 1)
const info = `event handler for "${event}"`
for (let i = 0, l = cbs.length; i < l; i++) {
  invokeWithErrorHandling(cbs[i], vm, args, vm, info)
}
}
return vm

```

Depois destas verificações, é criada uma variável **cbs** com o valor da chave **event** na propriedade **_events** de **vm**. É então checado se o valor de **cbs** é definido. Caso seja, o valor de **cbs** é definido com a avaliação do tamanho de **cbs** ser maior que 1. Sendo maior que 1, o valor de **cbs** é transformado em um **Array**, caso contrário o valor de **cbs** permanece inalterado. Depois é criada uma constante **args** com o valor de **arguments** transformado em um **Array**, e uma constante **info** com uma formatação de **string** utilizando o valor de **event**. Após isso é criada uma iteração sobre os valores de **cbs**, e para cada valor, é chamada a função **invokeWithErrorHandling()** para execução da função do índice **i** de **cbs**, com contexto de **vm**, argumentos **arguments**, uma definição de **vm** e com informações adicionais de **info**. Ao final da função o valor de **vm** é retornado pela função.

A.5 Mixin de Ciclos de Vida

Essa função define os valores de:

- **Vue.prototype._update** (onde acontecem as atualizações do documento)
- **Vue.prototype.\$forceUpdate**
- **Vue.prototype.\$destroy**

```
export function lifecycleMixin(Vue: typeof Component) {
```

A definição do *Mixin* de ciclos de vida é feita com uma instância do *Vue*, de tipo **Component**.

```
Vue.prototype._update = function (vnode: VNode, hydrating?: boolean)
```

A primeira propriedade a ser definida pelo Mixin é **_update()**. Ela toma as variáveis **vnode** de tipo **Vnode** e **hydrating** de tipo booleano como parâmetros, e não retorna valor.

```

const vm: Component = this
const prevEl = vm.$el
const prevVnode = vm._vnode
const restoreActiveInstance = setActiveInstance(vm)

```

A função inicia criando as constantes **vm** com o valor do **Component** que chamou esta função, **prevEl** com o valor da propriedade **\$el** de **vm**, **prevVnode** com o valor da propriedade **_vnode** de **vm** e **restoreActiveInstance** que recebe o valor de **setActiveInstance()**.

```
export function setActiveInstance(vm: Component) {
  const prevActiveInstance = activeInstance
  activeInstance = vm
  return () => {
    activeInstance = prevActiveInstance
  }
}
```

A função **setActiveInstance()** é definida com um parâmetro **vm** de tipo **Component**. No seu corpo é criada uma constante **prevActiveInstance** que recebe o valor da variável global **activeInstance** e o valor desta é alterado para o valor de **vm**. Após isso, é retornado como valor uma função anônima que define o valor de **activeInstance** com o valor de **prevActiveInstance**. Resumidamente, esta função cria um contexto prévio para a execução da sua função anônima interna, e quando ela é chamada, o contexto é capturado para sua execução, de modo que o valor de **prevActiveInstance** não é removido da memória do programa até que a função anônima seja chamada.

```
vm._vnode = vnode
if (!prevVnode) {
  vm.$el = vm.__patch__(vm.$el, vnode, hydrating, false /* removeOnly */)
} else {
  vm.$el = vm.__patch__(prevVnode, vnode)
}
```

Ao voltar para **_update()** o valor da propriedade **_vnode** de **vm** é definido com o valor de **vnode** e é checado se o valor de **prevNode** é indefinido. Caso seja, o valor da propriedade **\$el** de **vm** é definido com o valor da chamada para o método **__patch__()** de **vm** com os valores de **vm.\$el** e caso não seja **__patch__()** é chamado com os valores de **prevVnode**.

A função **__patch__()** é uma das funções que é específica para cada plataforma onde o *Vue* está sendo executado, portanto é difícil saber sobre seu funcionamento exato, porém, um trecho de sua funcionalidade, a função **patchVnode()** será analisada com mais detalhe em uma parte mais a frente deste capítulo, por agora, será finalizada a definição de **_update()**.

```
restoreActiveInstance()
if (prevEl) {
```

```

    prevEl.__vue__ = null
  }
  if (vm.$el) {
    vm.$el.__vue__ = vm
  }

```

Após a chamada para `__patch__()` o valor de **activeInstance** é redefinido como o anterior com a chamada para a constante que se transformou em uma função **restoreActiveInstance()**, e é checado se o valor de **prevEl** é definido, indefinindo ele caso seja verdadeiro, e é checado se o valor da propriedade **\$el** de **vm** é definida, adicionando o valor de **vm** à propriedade **__vue__** de **\$el**.

```

let wrapper: Component | undefined = vm
while (
  wrapper &&
  wrapper.$vnode &&
  wrapper.$parent &&
  wrapper.$vnode === wrapper.$parent._vnode
) {
  wrapper.$parent.$el = wrapper.$el
  wrapper = wrapper.$parent
}

```

Após essas checagens é criada uma variável **wrapper** de tipo **Component** ou indefinido e é dado a ela o valor de **vm**. É então criada uma estrutura de repetição que é executada enquanto a seguinte expressão for verdadeira: se **wrapper** for definido, se as propriedades **\$vnode** e **\$parent** de **wrapper** forem definidas e se a propriedade **\$vnode** e a propriedade **_vnode** de **\$parent** são estritamente iguais. Com a avaliação da expressão, o valor de **wrapper.\$parent.\$el** é definido com o valor de **wrapper.\$el** e o valor de **wrapper** é definido com o valor de **wrapper.\$parent**. Apesar desta estrutura condicional não executar nenhuma função, apenas a definição das variáveis de **wrapper** são suficientes para atualizar os valores filhos de **\$el** pois o agendador de tarefas do *Vue* irá chamar o **hook** de atualizações que consequentemente executará **_update()**.

```

Vue.prototype.$forceUpdate = function () {
  const vm: Component = this
  if (vm._watcher) {
    vm._watcher.update()
  }
}

```

Após a definição de **_update()** a definição de **\$forceUpdate()** é realizada. Esta função não requer nenhum parâmetro, e apenas cria uma constante **vm** com o componente que fez a chamada para este método e verifica se **vm** possui uma propriedade **_watcher** definida, chamando o método **update()** de **watcher** caso tenha.

```
update() {
  if (this.lazy) {
    this.dirty = true
  } else if (this.sync) {
    this.run()
  } else {
    queueWatcher(this)
  }
}
```

O método **update()** simplesmente verifica que tipo de vigia chamou esta função e o classifica: se for de tipo **lazy**, a propriedade **dirty** é definida como verdadeira. Se for **sync**, o método **run()** deste vigia é chamado, e se for **queue**, é feita uma chamada para **QueueWatcher()** com o valor de **this** como parâmetro. Primeiramente, o método **run()** será analisado e após isso, a função **queueWatcher()**.

```
run() {
  if (this.active) {
    const value = this.get()
```

O método **run()** é chamado, sem a definição de parâmetros, uma vez que os valores que serão analisados ou manipulados são as próprias propriedades da instância que chama este método. A função se inicia com a verificação do valor da propriedade **active** de **this**. Caso o valor seja verdadeiro, é criada uma constante **value** com o valor da chamada para **this.get()**.

```
get() {
  pushTarget(this)
  let value
  const vm = this.vm
```

O método **get()**, assim como **run()** é chamado sem parâmetros. A função se inicia com uma chamada para **pushTarget()** definindo o valor do elemento a ser avaliado como a instância do vigia que chamou este método. Depois é criada uma variável **value** e uma constante **vm** com o valor da propriedade **vm** da instância do **vigia**.

```

try {
  value = this.getter.call(vm, vm)
} catch (e: any) {
  if (this.user) {
    handleError(e, vm, `getter for watcher "${this.expression}"`)
  } else {
    throw e
  }
}

```

Após isso é criada uma estrutura de tratamento de erros, que tenta chamar **this.getter.call()** com os parâmetros definidos como **vm**, e que retorna seu valor para a variável **value**. Caso a chamada reproduza alguma exceção, é verificado se o valor da propriedade **user** é definido, e caso seja, é feita uma chamada para a função **handleError()**. Caso a propriedade **user** não seja definida, a exceção é lançada para o desenvolvedor.

```

export function handleError(err: Error, vm: any, info: string) {
  pushTarget()
  try {
    if (vm) {
      let cur = vm
      while ((cur = cur.$parent)) {
        const hooks = cur.$options.errorCaptured
        if (hooks) {
          for (let i = 0; i < hooks.length; i++) {
            try {
              const capture = hooks[i].call(cur, err, vm, info) === false
              if (capture) return
            } catch (e: any) {
              globalHandleError(e, cur, 'errorCaptured hook')
            }
          }
        }
      }
    }
    globalHandleError(err, vm, info)
  } finally {
    popTarget()
  }
}

```

A função **handleError()** define o topo da pilha de alvos como nulo para tratamento do erro, impedindo uma possível renderização infinita. Após isso, a função tenta chamar todos os **hooks** guardados nesta instância de **vm** e quando ele encontrar um **hook** problemático, chama **globalHandleError()** com o valor do erro levantado pela execução. Caso contrário, a chamada para **globalHandlerError** é feita com o valor do erro definido em **err**. Ao final da função, o valor atual da pilha de alvos é definido como o anterior com a chamada para **popTarget()**.

```
function globalHandleError(err, vm, info) {
  if (config.errorHandler) {
    try {
      return config.errorHandler.call(null, err, vm, info)
    } catch (e: any) {
      if (e !== err) {
        logError(e, null, 'config.errorHandler')
      }
    }
  }
  logError(err, vm, info)
}
```

A função **globalHandlerError()** toma como parâmetros os valores **err**, **vm** e **info**. A função começa criando uma estrutura condicional que verifica se o *Vue* foi configurado para tratar erros, através de **config.errorHandler**. Se não for o caso, a estrutura é ignorada e apenas uma chamada para **logError()** é feita. Caso contrário, uma estrutura de tratamento de erros é levantada e tenta retornar o valor da chamada para **errorHandler.call()**, se a chamada não levantar nenhuma exceção, o valor é devidamente retornado, caso contrário o valor da exceção é capturado em uma variável **e** e é então checado se o valor de **e** é estritamente diferente do valor de **err**. Caso seja, o valor de **e** é passado para a chamada de **logError()**. Esta estrutura condicional é usada para garantir que o mesmo erro não seja registrado duas vezes.

```
finally {
  if (this.deep) {
    traverse(value)
  }
  popTarget()
  this.cleanupDeps()
}
return value
```

Ao final da função **get()**, há a verificação do valor da propriedade **deep** da instância atual. Caso verdadeiro, é feita uma chamada para **traverse()** com o valor de **value** como parâmetro. Após isso, o valor da pilha de alvos é definido com o próximo valor, através da chamada para **popTarget()**, e é feita uma chamada para o método **cleanupDeps()** nesta instância do objeto. Após isso, o valor de **value** é retornado como valor.

```
const seenObjects = new Set()

export function traverse(val: any) {
  _traverse(val, seenObjects)
  seenObjects.clear()
  return val
}
```

A função **traverse()** usa um parâmetro **val** de tipo qualquer, e executa apenas uma chamada de função e uma chamada de método, o que o torna uma função **wrapper**. É feito primeiro uma chamada para **_traverse()** com **val** e **seenObjects** como parâmetros, e após essa chamada, os valores que serão definidos em **seenObjects**, que é definido como um conjunto, são limpos com a chamada para o método **clear()**. Ao final da função, o valor de **val** é retornado.

```
function _traverse(val: any, seen: SimpleSet) {
  let i, keys
  const isA = isArray(val)
```

A definição de **_traverse()** define um parâmetro **val** de tipo qualquer e **seen** de tipo que aplica a interface **SimpleSet**. No início da função, são criadas as variáveis **i** e **keys**, sem valores definidos, e uma constante **isA** com o valor da chamada para **isArray()** usando **val** como parâmetro.

```
if (
  (!isA && !isObject(val)) ||
  val.__v_skip /* ReactiveFlags.SKIP */ ||
  Object.isFrozen(val) ||
  val instanceof VNode
) {
  return
}
```

É então verificado se o valor de **isA** é falso e se **val** não é um objeto ou se a propriedade **__v_skip** de **val** é definida, ou se o valor de **val** está congelado (o que significa

que suas propriedades e valores não podem ser modificados, e novos valores não podem ser adicionados) ou por fim se **val** é uma instância de **Vnode**. Caso essa expressão seja avaliada como verdadeira, a função é encerrada aqui.

```
if (val.__ob__) {
  const depId = val.__ob__.dep.id
  if (seen.has(depId)) {
    return
  }
  seen.add(depId)
}
```

Depois uma estrutura condicional que checa se o valor da propriedade **__ob__** de **val** é definida. Caso verdade, é criada uma constante **depId** com o valor de **val.__ob__.dep.id**, e então é verificado se o valor de **seen** possui um valor **depId**, encerrando a função aqui caso verdadeiro. Caso contrário o valor de **depId** é adicionado à **seen**. Esta estrutura ainda não faz sentido lógico, pois nenhum valor é definido em **seen**, o que não permite as instruções da estrutura condicional serem executadas em momento algum. Para entender essa parte, é necessário verificar o próximo bloco de código.

```
if (isA) {
  i = val.length
  while (i--) _traverse(val[i], seen)
} else if (isRef(val)) {
  _traverse(val.value, seen)
} else {
  keys = Object.keys(val)
  i = keys.length
  while (i--) _traverse(val[keys[i]], seen)
}
```

Aqui, é verificado se **isA** é verdadeiro. Caso seja, o valor de **i** é definido com o tamanho de **val**, e é criada uma estrutura de repetição que dura enquanto o valor de **i** for maior que 0, decrementando o valor de **i** a cada iteração, e fazendo uma chamada direta para **_traverse()**. Aqui é possível observar por que a estrutura condicional anterior existe, uma vez que haverá uma chamada recursiva para **_traverse()**, sem redefinir o valor de **seen** em nenhum momento. Caso **isA** seja falso, é verificado se o valor de **val** é uma referência, chamando **_traverse()** com o valor da propriedade **value** de **val** como parâmetro ao invés de **val**. Caso **val** não seja uma referência, isso só pode significar que **val** é um objeto. Então é definido o valor de **keys** com o valor das chaves de **val**, o valor de **i** com o tamanho de **keys** e é criada uma estrutura de repetição que dura enquanto o valor de **i** for maior que

0, decrementando o valor de **i** a cada iteração, e chamando recursivamente **_traverse()**, marcando também o final de **_traverse()**.

```

    popTarget()
    this.cleanupDeps()
  }
return value

```

Voltando para **get()**, após a verificação de **deep**, o valor da pilha de alvos é definido com o próximo valor da pilha, e é feita uma chamada para o método **cleanupDeps()**. Ao final da função é retornado o valor de **value**.

```

cleanupDeps() {
  let i = this.deps.length
  while (i--) {
    const dep = this.deps[i]
    if (!this.newDeplds.has(dep.id)) {
      dep.removeSub(this)
    }
  }
  let tmp: any = this.deplds
  this.deplds = this.newDeplds
  this.newDeplds = tmp
  this.newDeplds.clear()
  tmp = this.deps
  this.deps = this.newDeps
  this.newDeps = tmp
  this.newDeps.length = 0
}

```

O método **cleanupDeps()** define uma variável **i** com o valor do tamanho das dependências da instância que chamou este método. Após isso é criada uma estrutura de repetição que dura enquanto o valor de **i** for maior que 0. Dentro da estrutura é criada uma constante **dep** com o valor do índice **i** de **this.deps**, e é verificado se o valor da propriedade **newDeplds** tem o valor de **dep.id**, chamando o método **removeSub()** para a instância de **dep** caso verdadeiro. Agora fora da estrutura de repetição, é criada uma variável temporária **tmp** que recebe o valor de **this.deplds**. Após isso o valor de **this.deplds** é definido com o valor de **this.newDeplds**, depois o valor da propriedade **newDeplds** é limpo com o método **clear()**, e o valor de **tmp** recebe o valor de **this.deps**. Após isso o valor de **this.deps** recebe o valor de **this.newDeps** e o valor de **this.newDeps** recebe o valor de **tmp**. Por fim o valor

do tamanho da propriedade **newDeps** é definido como 0, marcando o fim de **cleanupDeps()** e de **get()**.

```
if (
  value !== this.value ||
    isObject(value) ||
    this.deep
) {
  const oldValue = this.value
  this.value = value
  if (this.user) {
    const info = `callback for watcher "${this.expression}"`
    invokeWithErrorHandling(
      this.cb,
      this.vm,
      [value, oldValue],
      this.vm,
      info
    )
  }
}
```

Ao voltar para **get()** é criada uma estrutura condicional que verifica se o valor de **value** é estritamente igual ao valor de **this.value** ou se o valor de **value** é um objeto ou se o valor da propriedade **deep** de **this** é definido. Caso a expressão seja avaliada como verdadeira, é criada uma constante **oldValue** com o valor de **this.value** e o valor de **this.value** e o valor de **this.value** é definido com o valor de **value**. Após isso é verificado se o valor de **this.user** é definido. Caso seja, é criada uma constante **info** com uma **string** formatada com o valor de **this.expression**, e é feita uma chamada para **invokeWithErrorHandling()** chamando o valor da função de **this.cb** e passando como parâmetros os valores de **value** e **oldValue**.

```
else {
  this.cb.call(this.vm, value, oldValue)
}
```

Caso a expressão na estrutura condicional seja falsa, é feita uma chamada direta para o valor de **this.cb** com os parâmetros de **this.vm**, **value** e **oldValue**. Este trecho marca o fim do método **run()**.

```
export function queueWatcher(watcher: Watcher) {
  const id = watcher.id
```

```

if (has[id] !== null) {
  return
}

```

A função **queueWatcher()** é definida com um vigia **watcher** como parâmetro. No início da função é criada uma constante **id** com o valor do **id** do vigia, e é verificado se o valor de **has** possui um índice **id** sem valor nulo. Caso seja verdade, a função termina sua execução aqui.

```

if (watcher === Dep.target && watcher.noRecurse) {
  return
}

```

Depois é verificado se o valor deste vigia é o mesmo valor que o alvo atual da pilha de alvos e se o valor da propriedade **noRecurse** é verdadeiro, terminando a execução da função aqui caso a expressão seja avaliada como verdadeira.

```

has[id] = true
if (!flushing) {
  queue.push(watcher)
}

```

Após isso, o valor do índice **id** da variável **has** é definido como verdadeiro e então é verificado se o valor de **flushing** é falso, e caso seja, o valor do vigia é adicionado a fila com a chamada para o método **push()**.

```

let i = queue.length - 1
while (i > index && queue[i].id > watcher.id) {
  i--
}
queue.splice(i + 1, 0, watcher)

```

Depois é criado uma variável **i** com o tamanho de **queue** menos 1, e é criado uma estrutura de repetição que dura enquanto o valor de **i** for maior que o valor de **index** e enquanto o valor do **id** do índice **i** de **queue** for maior que o valor do **id** do **vigia**, decrementando o valor de **i** a cada iteração. Ao final da execução o valor de **i** estará definido como o valor exato da posição em que **watcher** precisa ser adicionado na fila, chamando a função **splice()** para alterar o valor do índice **i** mais 1 de **queue** com o valor do vigia.

```

if (!waiting) {
  waiting = true
}

```

```

    if (__DEV__ && !config.async) {
      flushSchedulerQueue()
      return
    }
    nextTick(flushSchedulerQueue)
  }
}

```

Depois é criada uma estrutura condicional que verifica se o valor de **waiting** é falso. Caso o valor seja avaliado como verdadeiro, o valor de **waiting** é definido como verdadeiro, é verificado se o *Vue* está em perfil de desenvolvimento e se **async** não é definido nas configurações do *Vue*, chamando **flushSchedulerQueue()** diretamente e terminando a execução da função aqui caso a expressão seja verdadeira. Caso a expressão seja falsa, o valor de **flushSchedulerQueue()** é passado como parâmetro para **nextTick()** para ser executado.

```

export function nextTick(): Promise<void>
export function nextTick<T>(this: T, cb: (this: T, ...args: any[]) => any): void
export function nextTick<T>(cb: (this: T, ...args: any[]) => any, ctx: T): void

```

A função **nextTick()** é definida de 3 maneiras possíveis:

- Sem parâmetros para execução, retornando apenas um tipo **Promise<void>**.
- Com um parâmetro **this** que incorpora um tipo genérico **T**, uma função **cb** que incorpora um tipo genérico **T** e uma cadeia de argumentos de qualquer tipo retornando um tipo qualquer, e um tipo de retorno vazio.
- Com um parâmetro **cb** de tipo função que tem como parâmetro um valor **this** que incorpora um tipo genérico **T** e uma cadeia de argumentos de qualquer tipo, retornando como valor um tipo qualquer e um contexto que incorpora um tipo genérico **T**, a função não retorna nada.

Vale lembrar que **Promise** é um tipo que marca o fim de uma operação assíncrona, retornando algum tipo como valor. A partir destas 3 possíveis implementações, o corpo da função **nextTick()** é definido.

```

export function nextTick(cb?: (...args: any[]) => any, ctx?: object) {
  let _resolve
  callbacks.push(() => {
    if (cb) {
      try {
        cb.call(ctx)
      } catch (e: any) {
        handleError(e, ctx, 'nextTick')
      }
    }
  })
}

```

```

    } else if (_resolve) {
      _resolve(ctx)
    }
  })
  if (!pending) {
    pending = true
    timerFunc()
  }
  // $flow-disable-line
  if (!cb && typeof Promise !== 'undefined') {
    return new Promise(resolve => {
      _resolve = resolve
    })
  }
}

```

A função se inicia com a criação de uma variável **_resolve** e com a adição de um valor de função para a constante global **callbacks** com a estrutura a seguir: Se **cb** é definido, é criada uma estrutura de tratamento de erros para tentar chamar a função **cb**, coletando um erro e chamando **handleError()** em caso de falha. Caso **cb** seja indefinido, é verificado se **_resolve** é definido, e caso seja, a função contida em **_resolve** é chamada. Após essa definição de função, é verificado se o valor da variável **pending** é falso, e caso seja, o valor é definido como verdadeiro e é feita uma chamada para a função **timerFunc()**. A definição de **timerFunc()** é variável de acordo com o tipo de plataforma onde o *Vue* está sendo executado, mas o funcionamento adequado é idealmente o mesmo que uma chamada para **setTimeout()** da biblioteca padrão de *JavaScript/Typescript*.

Após a verificação da estrutura condicional, é verificado se o valor de **cb** é indefinido e se o tipo de **Promise** é indefinido. Caso essa expressão seja verdade, o valor retornado pela função **nextTick()** é definido como uma **Promise** com um construtor que define **resolve** como uma função que define **_resolve** com o valor de **resolve**.

Até aqui foi possível observar o funcionamento da função **nextTick()** que será executada dentro da função **queueWatcher()**, mas a função a ser executada por **nextTick()**, **flushSchedulerQueue()**, não foi explicada ainda, o que será feito a seguir.

```

function flushSchedulerQueue() {
  currentFlushTimestamp = getNow()
  flushing = true
  let watcher, id

```

A função **flushSchedulerQueue()** não possui parâmetros, e começa com a definição da variável global **currentFlushTimestamp** com o valor da chamada para **getNow()**.

```
let getNow: () => number = Date.now
```

A função **getNow()** é um *wrapper* que guarda uma função que retorna um número, neste caso, o valor de **Date.now**.

Após isso, o valor global de **flushing** é definido como **true** e são criadas duas variáveis **watcher** e **id** sem definição de valores.

```
queue.sort(sortCompareFn)
```

Após isso, os valores dos vigias dentro de **queue** são organizados, para otimizar o processo de ativação ou desativação de vigias.

```
for (index = 0; index < queue.length; index++) {
  watcher = queue[index]
  if (watcher.before) {
    watcher.before()
  }
  id = watcher.id
  has[id] = null
  watcher.run()
  // in dev build, check and stop circular updates.
  if (__DEV__ && has[id] !== null) {
    circular[id] = (circular[id] || 0) + 1
    if (circular[id] > MAX_UPDATE_COUNT) {
      warn(
        'You may have an infinite update loop ' +
        (watcher.user
          ? `in watcher with expression "${watcher.expression}"`
          : `in a component render function.`),
        watcher.vm
      )
      break
    }
  }
}
```

Após isso é uma estrutura de repetição que tem a duração definida pelo tamanho de **queue**. É definido o valor de **watcher** com o valor do índice **index** de **queue**, e é verificado se o vigia possui uma propriedade **before** definida, chamando-a caso tenha. Depois o valor de **id** é definido com o número de identificação do vigia, e o valor do índice **id** em **has** é definido como nulo, e é feita uma chamada para o método **run()** deste vigia. Após isso há uma verificação do perfil de desenvolvimento do *Vue* e a checagem do valor no índice **id** de

has para garantir que ele é nulo, mas caso não seja, é verificado se o valor do índice **id** em **circular** é maior que o valor da constante **MAX_UPDATE_COUNT**, o que pode indicar uma repetição infinita de atualização, quebrando então o ciclo de repetição.

```
const activatedQueue = activatedChildren.slice()
const updatedQueue = queue.slice()
```

```
resetSchedulerState()
```

```
callActivatedHooks(activatedQueue)
callUpdatedHooks(updatedQueue)
cleanupDeps()
```

```
if (devtools && config.devtools) {
  devtools.emit('flush')
}
```

Após isso, são criadas duas constantes: **activatedQueue** com o valor de **activatedChildren** e **updatedQueue** com o valor de **queue**.

Após isso são feitas chamadas para **resetSchedulerState()**, **callActivatedHooks()**, **callUpdatedHooks()** e **cleanupDeps()** e é verificado se o *Vue* possui uma propriedade **devtools** definida, para emitir um evento **flush**.

```
function resetSchedulerState() {
  index = queue.length = activatedChildren.length = 0
  has = {}
  if (__DEV__) {
    circular = {}
  }
  waiting = flushing = false
}
```

A função **resetSchedulerState()** é definida sem parâmetros e é usada para definir os valores de tamanho de **index**, **queue** e **activatedChildren** como 0. Depois disso, o valor de **has** é definido como vazio e se o perfil de desenvolvimento for confirmado, o valor de **circular** é definido como vazio. Por fim os valores de **waiting** e **flushing** são definidos como falso.

```
function callActivatedHooks(queue) {
  for (let i = 0; i < queue.length; i++) {
    queue[i]._inactive = true
```

```

    activateChildComponent(queue[i], true /* true */)
  }
}

```

A função **callActivatedHooks()** toma um valor **queue** como parâmetro e cria uma iteração com a duração do tamanho de **queue**, que define o valor da propriedade **_inactive** do índice **i** de **queue** como verdadeiro e faz uma chamada para **activateChildComponent()** com o valor do índice **i** de **queue**.

```

export function activateChildComponent(vm: Component, direct?: boolean) {
  if (direct) {
    vm._directInactive = false
    if (isInInactiveTree(vm)) {
      return
    }
  } else if (vm._directInactive) {
    return
  }
  if (vm._inactive || vm._inactive === null) {
    vm._inactive = false
    for (let i = 0; i < vm.$children.length; i++) {
      activateChildComponent(vm.$children[i])
    }
    callHook(vm, 'activated')
  }
}

```

A função **activateChildComponent()** é definida com um parâmetro **vm** de tipo **Component** e um parâmetro **direct** de tipo booleano. A função checa se o valor de **direct** é verdadeiro ou definido, e define o valor da propriedade **_directInactive** como falso e checa se o valor de **vm** está na árvore de inativos, terminando a função aqui caso verdadeiro. Após isso, se o valor de **direct** for falso ou indefinido, a função termina sua execução aqui. Depois é checado se o valor da propriedade **_inactive** de **vm** é definida e possui um valor nulo. Caso essa expressão seja verdadeira, o valor de **_inactive** é definido como falso e é feita uma chamada recursiva para **activateChildComponent()** com os valores de **vm.\$children()**. Após isso é feita uma chamada para **callHook()**, chamando os **hooks** deste componente, e encerrando esta função.

```

function callUpdatedHooks(queue: Watcher[]) {
  let i = queue.length
  while (i--) {

```

```

const watcher = queue[i]
const vm = watcher.vm
if (vm &&
  vm._watcher === watcher &&
  vm._isMounted &&
  !vm._isDestroyed) {
  callHook(vm, 'updated')
}
}
}

```

A função **callUpdatedHooks()** toma como parâmetro um valor **queue** de tipo **Watcher**. Ela se inicia criando uma variável **i** com o valor do tamanho de **queue** e cria uma estrutura de repetição que dura enquanto o valor de **i** for maior que 0, decrementando seu valor a cada iteração. Dentro da iteração é criada uma constante **watcher** com o valor do índice **i** de **queue** e uma constante **vm** com o valor da propriedade **vm** de **watcher**. Após isso é verificado se o valor de **vm** é definido, se o valor de **vm._watcher** é igual a **watcher** e se **vm._isMounted** é verdadeiro e se **vm.isDestroyed** é falso. Caso essa expressão seja verdadeira, é feita uma chamada para **callHook()**.

```

export const cleanupDeps = () => {
  for (let i = 0; i < pendingCleanupDeps.length; i++) {
    const dep = pendingCleanupDeps[i]
    dep.subs = dep.subs.filter(s => s)
    dep._pending = false
  }
  pendingCleanupDeps.length = 0
}

```

A constante **cleanupDeps** é definida como uma função anônima sem parâmetros. No seu começo é criado uma estrutura de repetição que tem a duração do tamanho de **pendingCleanupDeps**. É criada uma constante **dep** com o valor do índice **i** de **pendingCleanupDeps**, e é definido o valor da propriedade **subs** de **dep** com o valor filtrado de **subs** filtrado e após isso o valor da propriedade **_pending** é definido como falso. Por fim o valor do tamanho de **pendingCleanupDeps** é definido como 0.

A finalização da explicação de **cleanupDeps()** marca o fim de **Vue.prototype.\$forceUpdate()**. Por fim, no *Mixin* de Ciclos de Vida, é feita a definição de **Vue.prototype.\$destroy()**

```

Vue.prototype.$destroy = function () {
  const vm: Component = this
  if (vm._isBeingDestroyed) {

```

```

    return
  }

```

A função **\$destroy()** é definida sem nenhum parâmetro e sem tipos de retorno. No seu início, ocorre a criação de uma constante **vm** de tipo **Component** que guarda o valor do componente que chamou esta função. Após isso é verificado se o valor da propriedade **_isBeingDestroyed** é verdadeiro, e caso seja, a função é encerrada aqui.

```

callHook(vm, 'beforeDestroy')
vm._isBeingDestroyed = true
const parent = vm.$parent
if (parent && !parent._isBeingDestroyed && !vm.$options.abstract) {
  remove(parent.$children, vm)
}

```

Após isso é feita uma chamada para a função **callHook()** com um valor “before-Destroy”, e o valor de **vm._isBeingDestroyed** é definido como verdadeiro. Após isso é criada uma constante **parent** que recebe o valor de **vm.\$parent**. Após isso é criada uma estrutura condicional que verifica se o valor de **parent** é definido, se o valor da propriedade **_isBeingDestroyed** de **parent** é falsa e se a propriedade **\$options.abstract** de **vm** é falsa. Caso essa expressão seja avaliada como verdadeira é feita uma chamada para **remove()**, uma função que remove valores definidos de um **Array**, passando **parent.\$children** e **vm** como parâmetros.

```

vm._scope.stop()
if (vm._data.__ob__) {
  vm._data.__ob__.vmCount--
}
vm._isDestroyed = true
vm.__patch__(vm._vnode, null)
callHook(vm, 'destroyed')
vm.$off()

```

Após isso é feita uma chamada para o método **stop()** de **vm._scope**, é verificado se o valor da propriedade **__ob__** é definido, guardando o valor de **__ob__.vmCount** como seu próprio valor menos 1, é definido o valor da propriedade **_isDestroyed** de **vm** como falso, é feita uma chamada para **__patch__()** usando o valor da propriedade **_vnode** como parâmetro, é feita uma chamada para **callHook()** usando o valor “destroyed” como parâmetro e é feita uma chamada para o método **\$off()** de **vm**. A definição de **\$off()** é específica para cada plataforma e já foi explicado sobre **__patch__()**, o que sobra por último a definição de **stop()**.

```

stop(fromParent?: boolean) {
  if (this.active) {
    let i, l
    for (i = 0, l = this.effects.length; i < l; i++) {
      this.effects[i].teardown()
    }
    for (i = 0, l = this.cleanups.length; i < l; i++) {
      this.cleanups[i]()
    }
    if (this.scopes) {
      for (i = 0, l = this.scopes.length; i < l; i++) {
        this.scopes[i].stop(true)
      }
    }
    // nested scope, dereference from parent to avoid memory leaks
    if (!this.detached && this.parent && !fromParent) {
      // optimized O(1) removal
      const last = this.parent.scopes!.pop()
      if (last && last !== this) {
        this.parent.scopes![this.index!] = last
        last.index = this.index!
      }
    }
    this.parent = undefined
    this.active = false
  }
}

```

A definição de **stop()** toma como parâmetro um valor booleano opcional **fromParent** e não retorna valor. No início da função é definido uma estrutura condicional que verifica o valor da propriedade **active** do valor da instância que chamou este método. Caso seja avaliado como verdadeiro, são criadas duas variáveis **i** e **l** e é criado 3 séries de estruturas de iteração:

- Uma iteração sobre os valores de **this.effect**, chamando o método **teardown()** sobre cada elemento.
- Uma iteração sobre os valores de **this.cleanups**, chamando a função assinalada no valor.
- Uma iteração sobre os valores de **this.scopes** chamando recursivamente **stop()** com o valor booleano definido como verdadeiro, caso o valor de **this.scopes** seja definido. Após isso é verificado se o valor de **this.detached** é falso, se o valor de **this.parent**

é definido e se o valor de **fromParent** é falso. Caso a expressão seja avaliada como verdadeira é criada uma constante **last** com o último valor de **this.parent.scopes** chamando o método `pop()` para obter o elemento, enquanto o elemento for definido. Após isso é verificado se o valor de **last** é definido e se o seu valor é estritamente diferente de **this**, e caso seja verdade, o valor de **this.parent.scopes** é definido com o valor de **last** no índice **this.index** caso ambos sejam definidos, e o valor da propriedade **index** de **last** é definido com o valor da propriedade **index** de **this** caso este seja definido. Em seguida, fora das estruturas condicionais, é definido o valor das propriedades **parent** e **active** de **this** com os valores indefinido e falso, respectivamente. Este trecho marca também o fim da função **stop()**.

```
if (vm.$el) {
  vm.$el.__vue__ = null
}
if (vm.$vnode) {
  vm.$vnode.parent = null
}
```

Ao final de **Vue.prototype.\$destroy()** é verificado se a propriedade **\$el** de **vm** é definida, e caso seja, o valor da propriedade **__vue__** desta propriedade é definido como nulo. Após isso é verificado se o valor pra propriedade **\$vnode** é definido, e caso seja, é definido o valor da propriedade **parent** de **\$vnode** é definido como nulo. Estas definições marcam o fim do método **\$destroy** de **Vue.prototype** e por consequência o fim do *Mixin* de Ciclos de Vida.

A.5.1 A função *patch()*

Agora, será analisada a definição de **__patch__()**. Esta função é definida apenas artificialmente como uma interface dentro de **Component**:

```
__patch__: (
  a: Element | VNode | void | null,
  b: VNode | null,
  hydrating?: boolean,
  removeOnly?: boolean,
  parentElm?: any,
  refElm?: any
) => any
```

Esta definição não auxilia ao entendimento da funcionalidade da função, portanto, ao procurar mais a fundo, é possível encontrar a função que a cria durante a execução do *Vue* a função de **__patch__()**.

```
export const patch: Function = createPatchFunction({ nodeOps, modules })
```

```
export function createPatchFunction(backend)
```

Ao encontrar a função **createPatchFunction()** foi descoberto que seu parâmetro é um **backend** de tipo qualquer implícito.

```
let i, j
const cbs: any = {}
```

Dentro desta função, são criadas duas variáveis **i** e **j** e uma constante **cbs** que guarda um tipo qualquer.

```
return function patch(oldVnode, vnode, hydrating, removeOnly)
```

É então feita a definição de uma funcionalidade **patch()** que é retornada pela função **createPatchFunction()**. A função é definida com um valor **oldVnode**, **vnode**, **hydrating** e **removeOnly**, igualmente a chamada de função de **__patch__()**. Isso delimita que foi encontrada pelo menos uma definição da função de **patch**, no caso a definição que é feita durante a execução do *Vue* no contexto da web.

```
if (isUndef(vnode)) {
  if (isDef(oldVnode)) invokeDestroyHook(oldVnode)
  return
}
```

A função se inicia verificando se o valor de **vnode** é indefinido, e caso seja, é checado se o valor de **oldVnode** é definido, chamando **invokeDestroyHook()** com o valor de **oldVnode** como parâmetro e retornando da função.

```
function invokeDestroyHook(vnode) {
  let i, j
  const data = vnode.data
  if (isDef(data)) {
    if (isDef((i = data.hook)) && isDef((i = i.destroy))) i(vnode)
    for (i = 0; i < cbs.destroy.length; ++i) cbs.destroy[i](vnode)
  }
  if (isDef((i = vnode.children))) {
    for (j = 0; j < vnode.children.length; ++j) {
      invokeDestroyHook(vnode.children[j])
    }
  }
}
```

A função se inicia verificando se o valor de **vnode** é indefinido, e caso seja, é checado se o valor de **oldVnode** é definido, chamando **invokeDestroyHook()** com o valor de **oldVnode** como parâmetro e retornando da função. A definição de **invokeDestroyHook()** toma um único parâmetro **vnode**.

São declaradas duas variáveis, **i** e **j**, e uma constante **data** com o valor da propriedade **data** de **vnode**. É então verificado se **data** é definido, e caso seja, é verificada a expressão a seguir:

Se a assinalação do valor de **data.hook** na variável **i** tem valor definido e se assinalação do valor da propriedade **destroy** de **i** para a variável **i** tem valor definido.

Caso essa expressão seja verdadeira é chamada a função **i()**, que foi definida dentro da estrutura condicional, com o valor de **vnode** para que este seja destruído.

É então criado uma iteração sobre os valores da propriedade **destroy** de **cbs** e é chamado a função de destruição para cada valor de **cbs.destroy** com o valor de **vnode**.

Após essa estrutura condicional, é checado se **vnode** possui valores filhos e caso haja, é criado uma estrutura de repetição que itera cada por cada valor filho de **vnode** e chama **invokeDestroyHook()** com seu valor.

```
let isInitialPatch = false
const insertedVnodeQueue: any[] = []
```

Ao voltar para **patch()** ocorre a declaração de uma variável **isInitialPatch** com o valor falso definido, e uma constante **insertedVnodeQueue** de tipo **Array** de qualquer tipo.

```
if (isUndef(oldVnode)) {
  isInitialPatch = true
  createElm(vnode, insertedVnodeQueue)
}
```

É então checado se o valor de **oldVnode** é definido, e caso seja, o valor de **isInitialPatch** é atualizado para verdadeiro, e é criado um componente com o valor de **vnode** através da função **createElm()**.

```
function createElm(
  vnode,
  insertedVnodeQueue,
  parentElm?: any,
  refElm?: any,
  nested?: any,
  ownerArray?: any,
  index?: any
)
```

A função **createElm()** toma como parâmetros os valores **vnode**, **insertedVnodeQueue**, **parentElm**, **refElm**, **nested**, **ownerArray** e **index**, todos de tipo qualquer.

```
if (isDef(vnode.elm) && isDef(ownerArray)) {
  vnode = ownerArray[index] = cloneVNode(vnode)
}
```

Ao voltar para **patch()** ocorre a declaração de uma variável **isInitialPatch** com o valor falso definido, e uma constante **insertedVnodeQueue** de tipo **Array** de qualquer tipo. É então checado se o valor de **oldVnode** é definido, e caso seja, o valor de **isInitialPatch** é atualizado para verdadeiro, e é criado um componente com o valor de **vnode** através da função **createElm()**.

A função **createElm()** toma como parâmetros os valores **vnode**, **insertedVnodeQueue**, **parentElm**, **refElm**, **nested**, **ownerArray** e **index**, todos de tipo qualquer. No começo da função é definido uma estrutura condicional que verifica se o valor de **ownerArray** e da propriedade **elm** de **vnode** são definidos.

Caso sejam o valor de **vnode** é alterado e recebe o valor da chamada para **cloneVnode()**, enquanto o valor de **ownerArray** no índice **index** também recebe este mesmo valor. O comentário indica que isto é feito para garantir que não haja erros posteriormente, pois a execução deste trecho indica que este **vnode** já foi usado em uma renderização passada

```
vnode.isRootInsert = !nested
  if (createComponent(vnode, insertedVnodeQueue, parentElm, refElm)) {
    return
  }
```

Depois é atribuído ao valor da propriedade **isRootInsert** da variável **vnode** o valor da negação da checagem do valor de **nested**. Apesar de ser definida com um tipo de qualquer valor, isso implica que **nested** sempre é chamado como um tipo booleano. Após essa atribuição, é checado se a criação de componente utilizando a função **createComponent()** retorna um valor não nulo. Caso seja, a função é encerrada aqui.

```
function createComponent(vnode, insertedVnodeQueue, parentElm, refElm) {
  let i = vnode.data
  if (isDef(i)) {
    const isReactivated = isDef(vnode.componentInstance) && i.keepAlive
    if (isDef((i = i.hook)) && isDef((i = i.init))) {
      i(vnode, false)
    }
    if (isDef(vnode.componentInstance)) {
      initComponent(vnode, insertedVnodeQueue)
    }
  }
```

```

    insert(parentElm, vnode.elm, refElm)
    if (isTrue(isReactivated)) {
        reactivateComponent(vnode, insertedVnodeQueue, parentElm, refElm)
    }
    return true
  }
}
}

```

A função **createComponent()** toma como parâmetros **vnode**, **insertedVnodeQueue**, **parentElm** e **refElm**, todos de tipo qualquer. A função inicia com a criação de uma variável **i** que recebe o valor da propriedade **data** de **vnode**. É então checado se esse valor é definido. Caso seja, é criada uma constante **isReactivated** que recebe o valor booleano da checagem de propriedade **keepAlive** de **i** e **vnode.componentInstance**. É então checado se o valor de **i.hook** é definido e se o valor de **i.init** é definido.

Caso verdade, o valor de **i** é definido com o valor da função **init()**, transformando **i** em uma função, que é chamada após a checagem da estrutura condicional. Após isso, é verificado se o valor da propriedade **componentInstance** de **vnode** é definido.

Caso seja, é feita uma chamada para **initComponent()**, uma chamada para **insert()** e caso **isReactivated** seja verdadeiro, uma chamada para **reactivateComponent()** é feita, e a função é encerrada com um valor de retorno verdadeiro.

```

function reactivateComponent(vnode,
insertedVnodeQueue,
parentElm,
refElm) {
  let i
  let innerNode = vnode
  while (innerNode.componentInstance) {
    innerNode = innerNode.componentInstance._vnode
    if (isDef((i = innerNode.data)) && isDef((i = i.transition))) {
      for (i = 0; i < cbs.activate.length; ++i) {
        cbs.activate[i](emptyNode, innerNode)
      }
      insertedVnodeQueue.push(innerNode)
      break
    }
  }
  insert(parentElm, vnode.elm, refElm)
}

```

A definição de **reactivateComponent** toma como parâmetros os valores **vnode**, **insertedVnodeQueue**, **parentElm** e **refElm**, e não retorna um valor. No início da função, é criada uma variável **i** sem a inicialização de seu valor, e a criação de uma variável **innerNode** com o valor de **vnode**. Após isso, é criada uma estrutura condicional que dura enquanto o valor da propriedade **componentInstance** de **innerNode** é definido. No corpo da estrutura o valor de **innerNode** é definido com o valor de **innerNode.componentInstance._vnode**, é verificado se o valor da propriedade **data** de **innerNode** é definida enquanto há uma atribuição deste valor para a variável **i**, e há a verificação da validade do valor da propriedade **transition** de **i**. Com a checagem desta estrutura condicional, é criada uma estrutura de repetição que tem a duração definida com o valor do tamanho da propriedade **activate** de **cbs**. Dentro da estrutura, é chamada a função interna dentro do índice **i** da propriedade **activate** de **cbs**, usando **emptyNode** e **innerNode** como parâmetros. Agora fora da estrutura condicional, é feita uma chamada para o método **push()** de **insertedVnodeQueue** para adicionar o valor de **innerNode**, e há uma chamada para **break**, terminando a execução da estrutura de repetição externa. Após isso é feito uma chamada para **insert()** utilizando os valores de **parentElm**, **vnode.Elm** e **refElm** como parâmetros.

```
function insert(parent, elm, ref) {
  if (isDef(parent)) {
    if (isDef(ref)) {
      if (nodeOps.parentNode(ref) === parent) {
        nodeOps.insertBefore(parent, elm, ref)
      }
    } else {
      nodeOps.appendChild(parent, elm)
    }
  }
}
```

A definição da função **insert()** define como parâmetros os valores **parent**, **elm** e **ref**, sem nenhum retorno de valor. A função inicia com uma estrutura condicional que verifica se o valor de **parent** é definido, e após isso outra estrutura que verifica se o valor de **ref** é definido. Caso ambas as avaliações sejam verdadeiras, é então verificado se o valor retornado pela chamada para **parentNode()** em **nodeOps** é estritamente igual a **parent**, executando uma chamada para o método **insertBefore()** de **nodeOps**, usando os valores de **parent**, **elm** e **ref** como parâmetros. Caso a penúltima estrutura avalie sua expressão como falsa, é feita então uma chamada para o método **appendChild()** de **nodeOps** com os valores de **parent** e **elm** como parâmetros.

```
export function insertBefore(
  parentNode: Node,
```

```

    newNode: Node,
    referenceNode: Node
  ) {
    parentNode.insertBefore(newNode, referenceNode)
  }

```

A definição da função **insert()** define como parâmetros os valores **parent**, **elm** e **ref**, sem nenhum retorno de valor. A função inicia com uma estrutura condicional que verifica se o valor de **parent** é definido, e após isso outra estrutura que verifica se o valor de **ref** é definido. Caso ambas as avaliações sejam verdadeiras, é então verificado se o valor retornado pela chamada para **parentNode()** em **nodeOps** é estritamente igual a **parent**, executando uma chamada para o método **insertBefore()** de **nodeOps**, usando os valores de **parent**, **elm** e **ref** como parâmetros. Caso a penúltima estrutura avalie sua expressão como falsa, é feita então uma chamada para o método **appendChild()** de **nodeOps** com os valores de **parent** e **elm** como parâmetros. A definição de **insertBefore()** tem como parâmetros os valores de **parentNode**, **newNode** e **referenceNode**, todos de tipo que implementa a interface **Node**. A função possui apenas uma única chamada para o método **insertBefore()** de **parentNode**, utilizando o valor de **newNode** e **referenceNode** como parâmetros que definem o valor a ser adicionado e qual valor deve ser tomado como referência ao adicionar o valor. O valor de **Node** é uma interface da biblioteca padrão que é adotada por valores de objetos que representam o documento de uma página web.

```

export function appendChild(node: Node, child: Node) {
  node.appendChild(child)
}

```

Similarmente, **appendChild()**, define dois parâmetros **node** e **child** com tipo que implementa **Node**. Dentro da função há apenas uma chamada para o método **appendChild()** que insere o valor de **child** em **node**. Ao finalizar a análise de **reactivateComponent()** e **insert()**, resta apenas a análise de **initComponent()**.

```

function initComponent(vnode, insertedVnodeQueue) {
  if (isDef(vnode.data.pendingInsert)) {
    insertedVnodeQueue.push.apply(
      insertedVnodeQueue,
      vnode.data.pendingInsert
    )
    vnode.data.pendingInsert = null
  }
  vnode.elm = vnode.componentInstance.$el
  if (isPatchable(vnode)) {

```

```

    invokeCreateHooks(vnode, insertedVnodeQueue)
    setScope(vnode)
  } else {
    registerRef(vnode)
    insertedVnodeQueue.push(vnode)
  }
}

```

A função **initComponent()** é definida com os valores de **vnode** e **insertedVnodeQueue** como parâmetros, e não retorna um valor. No início da função, é criada uma estrutura condicional que verifica se o valor da propriedade **data.pendingInsert** de **vnode** é definido, fazendo uma chamada para o método **push.apply()** de **insertedVnodeQueue** e definindo o valor da propriedade **data.pendingInsert** como nulo.

É então definido o valor da propriedade **elm** de **vnode** o valor da propriedade **componentInstance.\$el**, e é criada uma estrutura condicional que verifica se o valor de **vnode** é **remendável**, chamando as funções **invokeCreateHooks()** e **setScope()** caso verdadeiro, ou chamando **registerRef()** e adicionando o valor de **vnode** para **insertedVnodeQueue** utilizando o método **push()**.

```

function invokeCreateHooks(vnode, insertedVnodeQueue) {
  for (let i = 0; i < cbs.create.length; ++i) {
    cbs.create[i](emptyNode, vnode)
  }
  i = vnode.data.hook
  if (isDef(i)) {
    if (isDef(i.create)) i.create(emptyNode, vnode)
    if (isDef(i.insert)) insertedVnodeQueue.push(vnode)
  }
}

```

A função **invokeCreateHooks()** toma como parâmetros os valores **vnode** e **insertedVnodeQueue**. No início da função, é criada uma estrutura de repetição que itera sobre os valores da propriedade **create** de **cbs** usando um índice **i**, chamando a função neste índice com os parâmetros de **emptyNode** e **vnode**. Após isso, é definido o valor da variável **i** com o valor da propriedade **data.hook** de **vnode**, e é verificado se **i** é definido. Caso ele seja, duas verificações são feitas:

- Se **i.create** é definido, o método **create()** de **i** será chamado, usando **emptyNode** e **vnode** como parâmetros.
- Se **i.insert** é definido, o valor de **vnode** é inserido em **insertedVnodeQueue**, através do método **push()**.

Após a chamada para **invokeCreateHooks()** em **initComponent()**, uma chamada para **setScope()** é realizada.

```
function setScope(vnode) {
  let i
  if (isDef((i = vnode.fnScopeld))) {
    nodeOps.setStyleScope(vnode.elm, i)
  }
```

A função é definida com um valor **vnode** como parâmetro. A função inicia criando uma variável **i**, e atribuindo o valor da propriedade **fnScopeld** de **vnode** para **i**, verificando então se esse valor é definido. Caso seja, o método **setStyleScope()** de **nodeOps** é executado, usando **vnode.elm** e **i** como valores para os parâmetros.

```
export function setStyleScope(node: Element, scopeld: string) {
  node.setAttribute(scopeld, "")
}
```

A definição de **setStyleScope()** define dois parâmetros, uma variável **node** de tipo **Element** e uma variável **scopeld** de tipo **string**. É feita apenas uma chamada para o método **setAttribute()** de **node**, usando **scopeld** e uma **string** vazia como parâmetros. Essa chamada é feita para atribuir o valor de um atributo definido em **scopeld** como uma **string** vazia.

```
else {
  let ancestor = vnode
  while (ancestor) {
    if (isDef((i = ancestor.context)) && isDef((i = i.$options._scopeld))) {
      nodeOps.setStyleScope(vnode.elm, i)
    }
    ancestor = ancestor.parent
  }
}
```

Após isso, caso o valor de **i** não seja definido, é criada uma variável **ancestor** com o valor de **vnode**, e é feita uma estrutura de repetição que dura enquanto o valor de **ancestor** for avaliado como definido. Dentro da estrutura é verificado por uma estrutura condicional se o valor de **i**, agora com o valor de **ancestor.context** é definido, e se o valor de **i**, então estruturado com o valor da propriedade **\$options._scopeld** de **i**, tem valor definido. Se essa expressão for avaliada como verdadeira, é feita uma chamada para o método **setStyleScope()** de **nodeOps**, redefinindo o valor da propriedade **i** em **vnode.elm**. Por fim, é definido o valor de **ancestor** com o valor da sua própria propriedade **parent**, para continuação da estrutura de repetição.

```

if (
  isDef((i = activeInstance)) &&
  i !== vnode.context &&
  i !== vnode.fnContext &&
  isDef((i = i.$options._scopeld))
) {
  nodeOps.setStyleScope(vnode.elm, i)
}

```

Por fim, é verificado, a partir de uma estrutura condicional, se o valor de **activeInstance**, apropriado por **i**, é definido, se os valores das propriedades **context** e **fnContext** de **vnode** são estritamente diferentes de **i**, e se o valor de **\$options._scopeld**, guardado em **i**, é definido.

Caso essa expressão seja avaliada como verdadeira, é feita mais uma chamada para o método **setStyleScope()** de **nodeOps**, definindo o valor da propriedade de **i** como uma **string** vazia.

Após a definição de **setScope()**, na outra cadeia de execução da estrutura condicional de **initComponent()**, uma chamada para **registerRef()** é feita.

```
export function registerRef(vnode: VNodeWithData, isRemoval?: boolean)
```

A definição de **registerRef()** toma como parâmetros os valores **vnode** de tipo **VNodeWithData** e opcionalmente **isRemoval** de tipo booleano.

```

export type VNodeWithData = VNode & {
  tag: string
  data: VNodeData
  children: Array<VNode>
  text: void
  elm: any
  ns: string | void
  context: Component
  key: string | number | undefined
  parent?: VNodeWithData
  componentOptions?: VNodeComponentOptions
  componentInstance?: Component
  isRootInsert: boolean
}

```

A definição de **VNodeWithData** nos informa que este tipo é uma extensão dos valores de **Vnode**, mas com a definição de novos valores ou alteração de valores existentes ou suas opcionalidades.

```

const ref = vnode.data.ref
if (!isDef(ref)) return

const vm = vnode.context
const refValue = vnode.componentInstance || vnode.elm
const value = isRemoval ? null : refValue
const $refsValue = isRemoval ? undefined : refValue

```

No início de **registerRef()** há a definição de uma constante **ref** com o valor da propriedade **data.ref** de **vnode**, e a verificação da definição de seu valor. Se o valor for indefinido, a função é encerrada. Após isso, há a criação de uma constante **vm**, com o valor da propriedade **context** de **vnode**, uma constante **refValue** que recebe o valor de **vm.elm** caso **vnode.componentInstance** seja definido, uma constante **value** que verifica o valor de **isRemoval** e retorna nulo caso verdadeiro ou o valor de **refValue** caso falso e por fim, uma constante **\$refsValue** que também verifica o valor de **isRemoval**, mas retorna indefinido como valor caso verdadeiro ao invés de nulo, e retorna o valor de **refValue** caso falso.

```

if (isFunction(ref)) {
  invokeWithErrorHandling(ref, vm, [value], vm, `template ref function`)
  return
}

```

Após isso, é verificado se o valor de **ref** é uma função, e caso seja, a sua função é invocada através da chamada para **invokeWithErrorHandling()**, encerrando também a execução de **registerRef()**.

```

const isFor = vnode.data.refInFor
const _isString = typeof ref === 'string' || typeof ref === 'number'
const _isRef = isRef(ref)
const refs = vm.$refs

```

Após isso, quatro novas constantes são criadas: **isFor**, que recebe o valor da propriedade **data.refInFor** de **vnode**, **_isString** que recebe o valor da expressão que avalia se **ref** é uma **string** ou um número, **_isRef** que verifica se **ref** é uma referência através da chamada para **isRef()** e **refs** que recebe o valor da propriedade **\$refs** de **vm**.

```

if (_isString || _isRef)

```

Após isso, é criada uma estrutura condicional que verifica se os valores de **_isString** ou **_isRef** são verdadeiros.

```

if (isFor) {
    const existing = _isString ? refs[ref] : ref.value
    if (isRemoval) {
        isArray(existing) && remove(existing, refValue)
    } else {
        if (!isArray(existing)) {
            if (_isString) {
                refs[ref] = [refValue]
                setSetupRef(vm, ref, refs[ref])
            } else {
                ref.value = [refValue]
            }
        } else if (!existing.includes(refValue)) {
            existing.push(refValue)
        }
    }
}

```

Caso sejam, é feita uma verificação do valor de **isFor**, e caso seja verdadeiro, é criada uma constante **existing** que recebe o valor de **refs[ref]** caso o valor de **_isString** seja verdadeiro ou **ref.value** caso seja falso. Após isso é verificado se o valor de **isRemoval** é verdadeiro, e é então verificado se **existing** é um **Array**, executando uma chamada para **remove()** caso seja.

A função **remove()** é apenas uma função auxiliar que remove um valor de um **Array**. Caso **isRemoval** seja falso, é verificado se o valor de **existing** é falso, e caso seja é checado se o valor de **_isString** é verdadeiro, definindo o valor da propriedade **ref** em **refs** com o valor de **refValue** e chamando a função **setSetupRef()** com os valores de **vm**, **ref** e a propriedade **ref** de **refs** como parâmetros, ou caso o valor de **_isString** seja falso, é definido apenas o valor da propriedade **value** de **ref** com o valor de **refValue**.

Ao final, caso **existing** não seja um **Array**, é verificado se o valor de **existing** inclui o valor de **refValue**, e caso ele não inclua, esse valor é adicionado a **existing** através de seu método **push()**.

```

else if (_isString) {
    if (isRemoval && refs[ref] !== refValue) {
        return
    }
    refs[ref] = $refsValue
    setSetupRef(vm, ref, value)
}

```

Caso **isFor** seja falso, é verificado se o valor de **_isString** é verdadeiro, checando se o valor de **isRemoval** é verdadeiro e se o valor da propriedade **ref** de **refs** é diferente

de **refValue**, terminando a execução da função caso essa expressão seja avaliada como verdadeira. Após isso, fora da estrutura condicional, é definido o valor da propriedade **ref** de **refs** com o valor de **\$refsValue**, e é feita uma chamada para **setSetupRef()** com os valores de **vm**, **ref** e **value**.

```
else if (_isRef) {
  if (isRemoval && ref.value !== refValue) {
    return
  }
  ref.value = value
```

Após a verificação de **_isString**, caso esta seja falsa, é verificado se o valor de **_isRef** é verdadeiro. Caso seja, é checado se o valor de **isRemoval** é verdadeiro e se os valores da propriedade **value** de **ref** e **refValue** são diferentes, terminando a execução de **registerRef()** caso a expressão seja avaliada como verdadeira, ou apenas definindo o valor da propriedade **value** de **ref** com o valor da constante **value**.

```
else if (__DEV__) {
  warn(`Invalid template ref type: ${typeof ref}`)
}
```

Por fim, caso o *Vue* esteja em perfil de desenvolvimento, é avisado ao desenvolvedor que este **template** de referência é inválido, caso todas as outras checagens falhem, marcando também o final de **registerRef()**. Agora será feita a verificação da funcionalidade de **setSetupRef()**, que pode ser chamada em dois momentos diferentes dentro de **registerRef()**.

```
function setSetupRef(
  { _setupState }: Component,
  key: string | number,
  val: any
) {
  if (_setupState && hasOwn(_setupState, key as string)) {
    if (isRef(_setupState[key])) {
      _setupState[key].value = val
    } else {
      _setupState[key] = val
    }
  }
}
```

A função **setSetupRef()** toma como parâmetros um objeto de tipo **Component** porém é extraído apenas o valor da propriedade **_setupState**, um valor **key** de tipo **string** ou numérico e um valor **val** de tipo qualquer.

No corpo da função é verificado por uma estrutura condicional se o valor de **_setupState** é definido e se **_setupState** possui uma propriedade **key** em formato de **string**. Caso possua, é verificado então se o valor da propriedade **key** em **_setupState** é definido, guardando o valor de **val** na propriedade **value** da propriedade **key** de **_setupState** caso verdade.

Caso a última checagem seja falsa, o valor de **val** é guardado diretamente na propriedade **key** de **_setupState**, definindo o fim da função e de todo o contexto necessário para entender a função **initComponent()**, e por consequência tudo que é necessário para compreender **createComponent()**.

```
const data = vnode.data
const children = vnode.children
const tag = vnode.tag
```

Ao retornar para a função **createElm()**, ocorre a definição das constantes **data**, **children** e **tag**, com os valores das propriedades de mesmo nome e na mesma respectiva ordem de **vnode**.

```
if (isDef(tag)) {
  if (__DEV__) {
    if (data && data.pre) {
      creatingElmInVPre++
    }
    if (isUnknownElement(vnode, creatingElmInVPre)) {
      warn(
        'Unknown custom element: <' +
          tag +
          '> - did you ' +
          'register the component correctly? For \
recursive components, ' +
          'make sure to provide the "name" option.',
        vnode.context
      )
    }
  }
}
```

Após essas definições, é verificado se o valor de **tag** é definido e se o *Vue* está em perfil de desenvolvimento. Depois é verificado se o valor de **data** e de sua propriedade **pre**

são definidos, incrementando o valor de **creatingElmInVPre** caso verdadeiro. Após esta estrutura, é verificado se o elemento de **vnode** é desconhecido, levantando um aviso para o desenvolvedor.

```
vnode.elm = vnode.ns
  ? nodeOps.createElementNS(vnode.ns, tag)
  : nodeOps.createElement(tag, vnode)
setScope(vnode)
```

Após esse aviso, agora fora da estrutura que verifica o perfil de desenvolvimento do *Vue*, é atribuído à propriedade **elm** de **vnode** o valor da checagem da definição de sua propriedade **ns**: caso seja definida, a variável recebe o valor da execução do método **createElementNS()** de **nodeOps**, e caso seja indefinido a variável recebe o valor da execução ao método **createElement()** de **nodeOps**. Após essa atribuição, há uma execução à função **setScope()**.

```
export function createElementNS(namespace: string, tagName: string): Element {
  return document.createElementNS(namespaceMap[namespace], tagName)
}
```

A definição do método **createElementNS** define os valores **namespace** e **tagName** de tipo **string** como parâmetros, e retorna um valor de tipo **Element**. A função retorna imediatamente o valor da execução do método **createElementNS()** de **document**, usando o valor da propriedade **namespace** em **namespaceMap** e **tagName** como parâmetros que retorna um valor de um elemento HTML.

```
export function createElement(tagName: string, vnode: VNode): Element {
  const elm = document.createElement(tagName)
  if (tagName !== 'select') {
    return elm
  }
  if (
    vnode.data &&
    vnode.data.attrs &&
    vnode.data.attrs.multiple !== undefined
  ) {
    elm.setAttribute('multiple', 'multiple')
  }
  return elm
}
```

A definição de **createElement()**, recebe os valores de **tagName** de tipo **string** e **vnode** de tipo **Vnode** como parâmetros, retornando um valor de tipo **Element**. A função tem início com a criação de uma constante **elm** que recebe o valor da criação de um elemento HTML através da execução do método **createElement()** de **document**.

Após isso é checado se o valor de **tagName** é diferente da **string** 'select', retornando o valor de **elm** caso verdadeiro. Caso contrário a função continua, verificando se o valor das propriedades **data**, **data.attrs** e **data.attrs.multiple** de **vnode** são definidas, executando uma chamada para o método **setAttribute()** para definir o valor da propriedade **multiple** com o valor de **multiple** do elemento **elm** e por fim, fazendo o retorno do valor de **elm** pela função.

```
createChildren(vnode, children, insertedVnodeQueue)
```

Ao voltar para **createElm()**, há uma chamada para **createChildren()**, com os valores de **vnode**, **children** e **insertedVnodeQueue** como parâmetros.

```
function createChildren(vnode, children, insertedVnodeQueue)
```

Os mesmos parâmetros definidos na chamada da função são utilizados na definição da função, e não há nenhum retorno de valor.

```
if (isArray(children)) {
  if (__DEV__) {
    checkDuplicateKeys(children)
  }
}
```

A função começa verificando se o valor de **children** é um **Array**, e depois checa se o *Vue* se encontra em perfil de desenvolvimento, verificando se **children** possui chaves duplicadas através da execução de **checkDuplicateKeys()**.

```
function checkDuplicateKeys(children) {
  const seenKeys = {}
  for (let i = 0; i < children.length; i++) {
    const vnode = children[i]
    const key = vnode.key
    if (isDef(key)) {
      if (seenKeys[key]) {
        warn(
          `Duplicate keys detected: '${key}'. \
          This may cause an update error.`,
          vnode.context
        )
      }
    }
  }
}
```

```

    } else {
      seenKeys[key] = true
    }
  }
}
}

```

A função **checkDuplicateKeys()** toma um valor **children** como parâmetro e não retorna nenhum valor. É criado no seu início uma constante **seenKeys** com um objeto vazio como valor, e é feita uma estrutura de repetição que tem a duração definida pelo tamanho de **children**. Dentro da estrutura é definida uma constante **vnode** com o valor do índice **i** de **children** e uma constante **key** com o valor da propriedade **key** de **vnode**. Após isso é verificado se o valor de **key** é definido e se o valor da chave **key** em **seenKeys** é verdadeiro ou definido, indicando uma repetição de valores, e chamando a função **warn()** para avisar o desenvolvedor sobre esta chave duplicada. Caso o valor da chave **key** em **seenKeys** seja falso, o valor desta chave é definido como verdadeiro, e a definição da função é encerrada aqui.

```

for (let i = 0; i < children.length; ++i) {
  createElm(
    children[i],
    insertedVnodeQueue,
    vnode.elm,
    null,
    true,
    children,
    i
  )
}

```

Ao voltar para **createChildren()** e após a execução de **checkDuplicateKeys()** caso o *Vue* esteja em perfil de desenvolvimento, é feita uma iteração sobre os valores de **children**, e é feita uma chamada recursiva indireta para **createElm()**, para que os elementos filhos de **vnode** sejam criados.

```

else if (isPrimitive(vnode.text)) {
  nodeOps.appendChild(vnode.elm,
    nodeOps.createTextNode(String(vnode.text)))
}

```

Após isso, fora da estrutura de repetição, e caso o valor de **children** não seja de tipo **Array**, é verificado se o valor da propriedade **text** de **vnode** é de tipo primitivo, fazendo

uma chamada para o método **appendChild()** de **nodeOps** caso verdadeiro, utilizando os valores de **vnode.elm** e **nodeOps.createTextNode()** com o valor de **vnode.text** como parâmetro.

```
export function createTextNode(text: string): Text {
  return document.createTextNode(text)
}
```

O método **createTextNode()** de **nodeOps** é definido com um parâmetro **text** de tipo **string** e retorna um tipo **Text** como valor. Uma única chamada para o método **createTextNode()** de **document** é feita e retornada como valor para a função.

```
if (isDef(data)) {
  invokeCreateHooks(vnode, insertedVnodeQueue)
}
```

Ao voltar para **createElm()**, há a criação de uma estrutura condicional que verifica se o valor de **data** é definido, fazendo uma chamada para a função **invokeCreateHooks()** com os valores de **vnode** e **insertedVnodeQueue** como parâmetros.

```
insert(parentElm, vnode.elm, refElm)
```

```
if (__DEV__ && data && data.pre) {
  creatingElmInVPre--
}
```

Após isso, no final da estrutura condicional de **createElm()**, há a execução para a função **insert()** e uma verificação do perfil de desenvolvimento do *Vue* e da definição de **data** e sua propriedade **pre**, executando uma operação de decremento em **creatingElmInVPre**.

```
else if (isTrue(vnode.isComment)) {
  vnode.elm = nodeOps.createComment(vnode.text)
  insert(parentElm, vnode.elm, refElm)
}
```

Após isso, fora da estrutura condicional principal de **createElm()**, é verificado então se o valor da propriedade **isComment** de **vnode** é verdadeiro, definindo o valor da sua propriedade **elm** com o valor da chamada para o método **createComment()** de **nodeOps** e fazendo uma chamada para a função **insert()**.

```
export function createComment(text: string): Comment {
  return document.createComment(text)
}
```

O método **createComment()** de **nodeOps** toma como parâmetro único o valor **text** de tipo **string** e retorna um valor de tipo **Comment**, e executa uma única chamada para o método **createComment()** de **document** com um parâmetro **text**, e retornando esse valor.

```
else {
  vnode.elm = nodeOps.createTextNode(vnode.text)
  insert(parentElm, vnode.elm, refElm)
}
```

Por fim, ao voltar para **createElm()**, caso **vnode.isComment** seja falso, é atribuído ao valor da propriedade **elm** de **vnode** o valor da chamada para o método **createTextNode()** de **nodeOps**, e é feita uma chamada para a função **insert()**. A definição deste trecho marca o fim da função **createElm()**.

```
if (isUndef(vnode)) {
  if (isDef(oldVnode)) invokeDestroyHook(oldVnode)
  return
}
```

```
let isInitialPatch = false
const insertedVnodeQueue: any[] = []
```

```
if (isUndef(oldVnode)) {
  isInitialPatch = true
  createElm(vnode, insertedVnodeQueue)
}
```

Ao voltar para **patch()**, é possível observar que apenas este trecho de código foi executado até então. Caso **vnode** seja indefinido e o **vnode** antigo seja definido a função encerra mais cedo, destruindo o **vnode** mais antigo e caso o **vnode** mais antigo seja indefinido, isso significa que o **template** está criando um novo **vnode**, fazendo assim a chamada para **createElm()**.

```
else {
  const isRealElement = isDef(oldVnode.nodeType)
  if (!isRealElement && sameVnode(oldVnode, vnode))
```

Isso significa dizer, que a partir desta definição de segundo fluxo da estrutura condicional em frente, tudo o que for executado dentro deste fluxo acontece por ser uma atualização de um **vnode** antigo para o valor do **vnode** novo.

Já neste trecho do código ocorre a definição de uma constante **isRealElement** que recebe o valor da checagem da definição da propriedade **nodeType** de **oldVnode**.

Após isso, é criada uma estrutura condicional que verifica se **isRealElement** é falso e se **oldVnode** e **vnode** são o mesmo **vnode**.

```
patchVnode(oldVnode, vnode, insertedVnodeQueue, null, null, removeOnly)
```

Caso a expressão seja avaliada como verdade, haverá a execução da função **patchVnode()**. Esta função é onde acontece a verdadeira funcionalidade de **patch()**, onde os valores de um **vnode** velho são alterados para igualar os valores de um **vnode** mais recente.

```
function patchVnode(
  oldVnode,
  vnode,
  insertedVnodeQueue,
  ownerArray,
  index,
  removeOnly?: any
)
```

A função **patchVnode()** é definida com os parâmetros **oldVnode**, **vnode**, **insertedVnodeQueue**, **ownerArray**, **index** e **removeOnly**, todos de tipo qualquer, e apenas **removeOnly** é opcional. A função não retorna nenhum valor.

```
if (oldVnode === vnode) {
  return
}
```

Primeiramente, a função verifica se **oldVnode** e **vnode** são estritamente iguais, o que indica que não há a necessidade de fazer qualquer atualização, encerrando a função caso seja verdade.

```
if (isDef(vnode.elm) && isDef(ownerArray)) {
  vnode = ownerArray[index] = cloneVNode(vnode)
}
```

Após isso, é verificado se o valor da propriedade **elm** de **vnode** e de **ownerArray** são definidos, atribuindo ao valor de **vnode** e **ownerArray** o valor de **vnode** duplicado por **cloneVNode()**.

```
export function cloneVNode(vnode: VNode): VNode {
  const cloned = new VNode(
    vnode.tag,
```

```

    vnode.data,
    vnode.children && vnode.children.slice(),
    vnode.text,
    vnode.elm,
    vnode.context,
    vnode.componentOptions,
    vnode.asyncFactory
  )
  cloned.ns = vnode.ns
  cloned.isStatic = vnode.isStatic
  cloned.key = vnode.key
  cloned.isComment = vnode.isComment
  cloned.fnContext = vnode.fnContext
  cloned.fnOptions = vnode.fnOptions
  cloned.fnScopId = vnode.fnScopId
  cloned.asyncMeta = vnode.asyncMeta
  cloned.isCloned = true
  return cloned
}

```

A definição de **cloneVnode()** apenas cria uma constante **cloned** para inicializar um **vnode** novo com os valores do antigo e realizar um método de **slice()** nos nódulos filhos de **vnode**, para garantir que os valores sejam duplicados e impedindo que os valores originais sofram mutação. Isso se dá por conta de uma característica dos ambientes de execução de código interpretado em que, durante a clonagem de uma variável para uma segunda variável, é possível que ao invés de passar o próprio valor para a variável, é repassada para ela uma referência de memória, causando problemas de mutação de valores em duas variáveis diferentes.

```

const elm = (vnode.elm = oldVnode.elm)

if (isTrue(oldVnode.isAsyncPlaceholder)) {
  if (isDef(vnode.asyncFactory.resolved)) {
    hydrate(oldVnode.elm, vnode, insertedVnodeQueue)
  } else {
    vnode.isAsyncPlaceholder = true
  }
  return
}

```

Voltando para **patchVnode()**, após a última estrutura condicional, há a criação de uma constante **elm** que recebe o valor da propriedade **elm** de **vnode**, que recebe, na mesma chamada, o valor da propriedade **elm** de **oldVnode**. Após isso é verificado se o valor da propriedade **isAsyncPlaceholder** é verdadeiro, e após isso, se o valor de **asyncFactory.resolved** é definido, ambas propriedades de **vnode**. Caso as duas sejam verdadeiras, é feita uma chamada para **hydrate()**. Caso apenas a segunda checagem seja avaliada como falsa, o valor da propriedade **isAsyncPlaceholder** recebe um valor verdadeiro, e a função termina sua execução aqui.

```
function hydrate(elm, vnode, insertedVnodeQueue, inVPre?: boolean)
```

A definição de **hydrate()** define a função com os parâmetros **elm**, **vnode**, **insertedVnodeQueue**, todas de tipo qualquer e opcionalmente **inVPre** de tipo booleano.

```
let i
const { tag, data, children } = vnode
inVPre = inVPre || (data && data.pre)
vnode.elm = elm
```

A função inicia definindo uma variável **i**, três constantes **tag**, **data** e **children** com os valores desconstruídos destas mesmas propriedades de **vnode**, guardando o valor da expressão que verifica se o valor de **inVPre** é definido ou se os valor de **data** e sua propriedade **pre** são definidos, e por fim, definindo o valor da propriedade **elm** de **vnode** com o valor de **elm**.

```
if (isTrue(vnode.isComment) && isDef(vnode.asyncFactory)) {
  vnode.isAsyncPlaceholder = true
  return true
}
```

É então verificado se o valor da propriedade **isComment** de **vnode** é verdadeiro e se o valor da propriedade **asyncFactory** de **vnode** é definido. Caso essa avaliação seja verdadeira, o valor da propriedade **asyncPlaceholder** é definido como verdadeiro, e a função retorna com um valor verdadeiro.

```
if (__DEV__) {
  if (!assertNodeMatch(elm, vnode, inVPre)) {
    return false
  }
}
```

Após isso, caso o *Vue* esteja em perfil de desenvolvimento, é feita uma checagem para avaliar se o nó virtual e normal são correspondentes através da função **assertNodeMatch()**, fazendo a função retornar falso caso não seja verdade.

```
function assertNodeMatch(node, vnode, inVPre) {
  if (isDef(vnode.tag)) {
    return (
      vnode.tag.indexOf('vue-component') === 0 ||
      (!isUnknownElement(vnode, inVPre) &&
        vnode.tag.toLowerCase() ===
          (node.tagName && node.tagName.toLowerCase()))
    )
  } else {
    return node.nodeType === (vnode.isComment ? 8 : 3)
  }
}
```

A definição de **assertNodeMatch()** é feita com os valores de **node**, **vnode** e **inVPre** como parâmetros. A função é iniciada verificando se o valor de **vnode.tag** é definido, retornando o valor da verificação do índice de **tags** de **vnode** e checando se esse número é igual a zero, ou se o valor negado da chamada para **isUnknownElement()** e o nome da **tag** de **vnode** é igual ao valor de **tagName** e **tagName** apenas em caracteres minúsculos pela função. Caso a checagem da definição de **vnode.tag** seja avaliada como falsa, o valor de **node.nodeType** é comparado com os valores de oito ou de três, dependendo se o valor da propriedade **isComment** de **vnode** for definido como verdadeiro ou falso respectivamente.

```
if (isDef(data)) {
  if (isDef((i = data.hook)) && isDef((i = i.init)))
    i(vnode, true)
  if (isDef((i = vnode.componentInstance))) {
    initComponent(vnode, insertedVnodeQueue)
    return true
  }
}
```

Ao voltar para **hydrate()**, é verificado se o valor de **data** é definido, e após isso, é verificado se o valor de **data.hook.init** é definido, incorporando seu valor e sua funcionalidade para a variável **i** e executando sua função de hidratação caso a expressão seja avaliada como verdadeira, utilizando **vnode** e um valor verdadeiro como parâmetros. Após isso, é verificado se o valor da propriedade **componentInstance** de **vnode** é definido, incorporando seu valor à variável **i**, e caso a expressão seja avaliada como verdadeira, executando

initComponent() com os valores de **vnode** e **insertedVnodeQueue** e retornando um valor verdadeiro.

```
if (isDef(tag)) {
  if (isDef(children)) {
    if (!elm.hasChildNodes()) {
      createChildren(vnode, children, insertedVnodeQueue)
    }
  }
}
```

É então verificado se o valor de **tag** e **children** são definidos, e então se o valor do método **hasChildNodes()** de **elm** retorna um valor falso. Caso verdadeiro, há a execução da função **createChildren()**.

```
else {
  if (
    isDef((i = data)) &&
    isDef((i = i.domProps)) &&
    isDef((i = i.innerHTML))
  )
```

Caso o valor de **hasChildNodes()** retorne um valor verdadeiro, é então verificado se o valor de **data.domProps.innerHTML** é definido, atribuindo seu valor a **i**.

```
if (
  __DEV__ &&
  typeof console !== 'undefined' &&
  !hydrationBailed
) {
  hydrationBailed = true
  console.warn('Parent: ', elm)
  console.warn('server innerHTML: ', i)
  console.warn('client innerHTML: ', elm.innerHTML)
}
return false
```

Após isso, é verificado se o valor de **i** é igual à propriedade **innerHTML** de **elm**. Dentro desta verificação, é checado se o *Vue* se encontra em perfil de desenvolvimento, se o tipo de **console** é diferente de indefinido e se o valor de **hydrationBailed** é falso. Caso essa expressão seja verdadeira, o valor de **hydrationBailed** é definido como verdadeiro, e três avisos são feitos para o console de execução do *Vue*, indicando o elemento pai, o valor do HTML no servidor e do cliente, respectivamente. Ao final desta verificação, a função é encerrada com um valor de retorno falso.

```

else {
  let childrenMatch = true
  let childNode = elm.firstChild

```

Caso **data.domProps.innerHTML** não seja definido na estrutura anterior, o fluxo alternativo de código definirá duas variáveis: **childrenMatch** com o valor de verdadeiro e **childNode** com o valor da propriedade **firstChild** de **elm**.

```

for (let i = 0; i < children.length; i++) {
  if (
    !childNode ||
    !hydrate(childNode, children[i],
      insertedVnodeQueue, inVPre)
  ) {
    childrenMatch = false
    break
  }
  childNode = childNode.nextSibling
}

```

Será feita então uma iteração que tem a duração definida pelo tamanho de **children**, que verifica se o valor de **childNode** é definido, e se a chamada recursiva para **hydrate()** usando **childNode** e o índice **i** de **children** como parâmetros é falsa. Caso essa expressão seja verdadeira, o valor de **childrenMatch** é definido como falso e a estrutura de repetição é encerrada com **break**. Caso a estrutura condicional não tenha sua execução interna feita, o valor de **childNode** é definido com o valor do próximo irmão.

```

if (!childrenMatch || childNode) {
  if (
    __DEV__ &&
    typeof console !== 'undefined' &&
    !hydrationBailed
  ) {
    hydrationBailed = true
    console.warn('Parent: ', elm)
    console.warn(
      'Mismatching childNodes vs. VNodes: ',
      elm.childNodes,
      children
    )
  }
}

```

```

    return false
}

```

É então feita uma checagem que verifica se o valor de **childrenMatch** é falso e se o valor de **childNodes** é definido, e então é checado se o *Vue* está em perfil de desenvolvimento, se o console possui um tipo diferente de indefinido e se o valor de **hydrationBailed** é falso. Caso essa expressão seja verdadeira, o valor de **hydrationBailed** é definido como verdadeiro, e dois avisos são emitidos no console de execução do *Vue*: um indicando o valor do elemento pai, e outro demonstrando que os valores dos nós filhos no elemento são diferentes daqueles no nó virtual. Após isso a função termina a sua execução com um valor falso de retorno.

```

if (isDef(data)) {
  let fullInvoke = false
  for (const key in data) {
    if (!isRenderedModule(key)) {
      fullInvoke = true
      invokeCreateHooks(vnode, insertedVnodeQueue)
      break
    }
  }
  if (!fullInvoke && data['class']) {
    traverse(data['class'])
  }
}

```

É verificado se o valor de **data** é definido, definindo uma variável **fullInvoke** com um valor falso e criando uma estrutura de repetição que itera sobre os valores de **data** capturando o valor atual na variável **key** e verifica se o valor de **key** não é um módulo renderizado. Caso a avaliação seja verdadeira, o valor de **fullInvoke** é definido como verdadeiro e é feita uma chamada para **invokeCreateHooks()** com os valores de **vnode** e **insertedVnodeQueue** e encerrando a execução da estrutura de repetição. Após isso é verificado se o valor de **fullInvoke** é falso e se o valor da chave “class” em **data** é definido, fazendo uma chamada para **traverse()** com o valor da chave “class” caso a expressão seja verdadeira.

```

else if (elm.data !== vnode.text) {
  elm.data = vnode.text
}
return true

```

Por fim, caso a primeira estrutura condicional da função não tenha seu corpo principal executado, é avaliado se os valores de **elm.data** e **vnode.text** são diferentes, definindo o valor de **elm.data** com o valor de **vnode.text** caso a avaliação seja verdadeira, e por fim encerrando a função com um valor verdadeiro como retorno.

```
if (
  isTrue(vnode.isStatic) &&
  isTrue(oldVnode.isStatic) &&
  vnode.key === oldVnode.key &&
  (isTrue(vnode.isCloned) || isTrue(vnode.isOnce))
) {
  vnode.componentInstance = oldVnode.componentInstance
  return
}
```

Ao voltar para **patchVnode()**, é criada outra estrutura condicional que checa se os valores das propriedades **isStatic** de **vnode** e **oldVnode** são verdadeiros, se os valores das propriedades **key** de **vnode** e **oldVnode** são iguais e se o valor de **vnode** foi clonado ou se o valor de **vnode** é único. Caso essa expressão seja avaliada como verdadeira, o valor da propriedade **componentInstance** de **vnode** é definido com o valor da mesma propriedade em **oldVnode** e a função é encerrada aqui.

```
let i
const data = vnode.data
if (isDef(data) && isDef((i = data.hook)) && isDef((i = i.prepatch))) {
  i(oldVnode, vnode)
}

const oldCh = oldVnode.children
const ch = vnode.children
if (isDef(data) && isPatchable(vnode)) {
  for (i = 0;
    i < cbs.update.length;
    ++i) cbs.update[i](oldVnode, vnode)
  if (isDef((i = data.hook)) &&
    isDef((i = i.update))) i(oldVnode, vnode)
}
```

Neste trecho, será feita a preparação para o remendo definitivo, o que será atribuído o nome de *pré-patch*. Aqui ocorre a criação de uma variável **i** e uma constante **data** com o valor da propriedade **data** de **vnode**.

Após isso é verificado se o valor de **data** é definido, se o valor de **data.hook** é definido e se o valor da **data.hook.prepatch** é definido, atribuindo o valor de **prepatch** para **i** caso verdadeiro, e executando a função de *pré-patch* agora definida em **i**, com os valores de **oldVnode** e **vnode** como parâmetros.

Após isso, são criadas duas constantes **oldCh** e **ch** com os valores da propriedade **children** de **oldVnode** e **vnode** respectivamente. Após isso, é verificado se o valor de **data** é definido e se o **vnode** é remendável.

Caso seja verdade, é criado uma estrutura de repetição que tem a duração definida pelo tamanho da propriedade **update** de **cbs**, que chama a função interna do índice **i** de **update** a cada iteração, utilizando os valores de **oldVnode** e **vnode** como parâmetros. Após isso é criada uma estrutura condicional que verifica se as propriedades **hook** e **hook.update** de **data** são definidas, define o valor da última para a variável **i** e executa a função de atualização agora incorporada em **i** com os valores de **oldVnode** e **vnode** como parâmetros.

```
if (isUndef(vnode.text)) {
  if (isDef(oldCh) && isDef(ch)) {
    if (oldCh !== ch)
      updateChildren(elm, oldCh, ch, insertedVnodeQueue, removeOnly)
  } else if (isDef(ch)) {
    if (__DEV__) {
      checkDuplicateKeys(ch)
    }
    if (isDef(oldVnode.text)) nodeOps.setTextContent(elm, "")
    addVnodes(elm, null, ch, 0, ch.length - 1, insertedVnodeQueue)
  } else if (isDef(oldCh)) {
    removeVnodes(oldCh, 0, oldCh.length - 1)
  } else if (isDef(oldVnode.text)) {
    nodeOps.setTextContent(elm, "")
  }
}
```

Neste trecho a seguir, é definido o bloco de código que realiza a parte real de *patching* do *Vue*. É criada uma estrutura condicional que verifica se o valor da propriedade **text** de **vnode** é definida. Depois é verificado se os valores de **oldCh** e **ch** são definidos e se são diferentes, executando a função **updateChildren()** caso verdadeiro. A função **updateChildren()**, de modo resumido, irá verificar uma série de condições e executar uma série otimizada de instruções para a execução recursiva de **patchVnode()**. Após a verificação de **oldCh** e **ch**, caso apenas **ch** seja definido e o *Vue* esteja em perfil de desenvolvimento, haverá uma chamada para a função **checkDuplicateKeys()**. Após isso,

será verificado se o valor da propriedade **text** de **vnode** é definida, fazendo uma chamada para o método **setTextContent()** de **nodeOps**.

```
export function setTextContent(node: Node, text: string) {
  node.textContent = text
}
```

A função **setTextContent()** apenas define o valor do texto de conteúdo de um nó fornecido com o valor de um texto fornecido através de um parâmetro **text**. Em seguida é feita uma chamada para **addVnodes()**.

```
function addVnodes(
  parentElm,
  refElm,
  vnodes,
  startIdx,
  endIdx,
  insertedVnodeQueue
) {
  for (; startIdx <= endIdx; ++startIdx) {
    createElm(
      vnodes[startIdx],
      insertedVnodeQueue,
      parentElm,
      refElm,
      false,
      vnodes,
      startIdx
    )
  }
}
```

A função **addVnodes()** toma os valores **parentElm**, **refElm**, **vnodes**, **startIdx**, **endIdx** e **insertedVnodeQueue** como parâmetros, e simplesmente cria uma estrutura de repetição que tem a duração definida pelos valores de **startIdx** e **endIdx**, e chama a função **createElm()**, usando o valor do índice **startIdx** de **vnodes** para a criação de cada elemento.

Após a chamada para **addVnodes()**, e caso o valor de **ch** não seja definido, é verificado se apenas o valor de **oldCh** é definido, executando a função **removeVnodes()**.

```
function removeVnodes(vnodes, startIdx, endIdx) {
  for (; startIdx <= endIdx; ++startIdx) {
```

```

const ch = vnodes[startIdx]
if (isDef(ch)) {
  if (isDef(ch.tag)) {
    removeAndInvokeRemoveHook(ch)
    invokeDestroyHook(ch)
  } else {
    // Text node
    removeNode(ch.elm)
  }
}
}
}

```

A função **removeVnodes()** é definida com os parâmetros **vnodes**, **startIdx** e **endIdx** como parâmetros. No início da função, é criada uma estrutura de repetição que tem a sua duração definida pelos valores de **startIdx** e **endIdx**. No início da iteração, é criada uma constante **ch** com o valor do índice **startIdx** de **vnodes**. É então verificado se o valor de **ch** é definido, e se o valor da propriedade **tag** de **ch** é definida. Caso seja, são feitas chamadas para as funções **removeAndInvokeRemoveHook()** e **invokeDestroyHook()**. Caso **ch.tag** não seja definido, é feita uma chamada para **removeNode()**.

```

function removeNode(el) {
  const parent = nodeOps.parentNode(el)
  if (isDef(parent)) {
    nodeOps.removeChild(parent, el)
  }
}

```

A definição de **removeNode()** toma apenas o valor **el** como parâmetro e não tem valor de retorno. No corpo da função é criado uma constante **parent** com o valor da chamada para o método **parentNode()** de **nodeOps**, e é verificado se essa constante é definida. Caso seja, é feita uma chamada para o método **removeChild()** de **nodeOps**.

```

export function parentNode(node: Node) {
  return node.parentNode
}

```

O método **parentNode()** tem como parâmetro um valor **node** que implementa a interface **Node** e retorna o valor da propriedade **parentNode** do parâmetro **node**.

```
export function removeChild(node: Node, child: Node) {
  node.removeChild(child)
}
```

O método **removeChild()** é definido com os parâmetros **node** e **child** de tipo que implementa a interface **Node**, e executa uma remoção do nó filho no nó pai através da execução do método **removeChild()** do parâmetro **node**.

```
function removeAndInvokeRemoveHook(vnode, rm?: any) {
```

Após a definição de **removeNode()**, basta apenas fazer a definição de **removeAndInvokeRemoveHook()** para concluir a análise de **removeVnodes()**. A função **removeAndInvokeRemoveHook()** toma como parâmetros os valores **vnode** e **rm** de tipo qualquer e opcional.

```
if (isDef(rm) || isDef(vnode.data)) {
  let i
  const listeners = cbs.remove.length + 1
  if (isDef(rm)) {
    rm.listeners += listeners
  }
```

No início da função, é verificado se o valor de **rm** é definido ou se o valor da propriedade **data** de **vnode** é definido, criando uma variável **i** e uma constante **listeners** com o valor do tamanho da propriedade **remove** de **cbs** menos 1. É então verificado se o valor de **rm** é definido, acrescentando o número da propriedade **listeners** de **rm** com o valor da constante **listeners**.

```
else {
  rm = createRmCb(vnode.elm, listeners)
}
```

Caso **rm** não seja definido, o valor de **rm** é definido com o valor de retorno da chamada para **createRmCb()**.

```
function createRmCb(childElm, listeners) {
  function remove() {
    if (--remove.listeners === 0) {
      removeNode(childElm)
    }
  }
  remove.listeners = listeners
```

```

    return remove
  }

```

A função **createRmCb()** toma como parâmetros os valores **childElm** e **listeners**, e não retorna nenhum valor. A função é definida com uma função interna **remove()** que cria uma estrutura condicional que verifica se o número da propriedade **listeners** de **remove** decrementado é igual a 0, e chama a função **removeNode()** com o valor de **childElm** como parâmetro. Após isso a função mais externa define o valor de **remove.listeners**, neste caso a variável e não a função, com o valor de **listeners** e retorna a função **remove()**.

```

if (
  isDef((i = vnode.componentInstance)) &&
  isDef((i = i._vnode)) &&
  isDef(i.data)
) {
  removeAndInvokeRemoveHook(i, rm)
}

```

Ao voltar para **removeAndInvokeRemoveHook()**, é então verificado se os valores de **vnode.componentInstance**, **_vnode** e **data**, todos sendo uma propriedade do anterior, são definidos, e o valor deles é atribuído à variável **i**. Caso a expressão seja avaliada como verdadeira, é feita uma chamada recursiva para a função **removeAndInvokeRemoveHook()**.

```

for (i = 0; i < cbs.remove.length; ++i) {
  cbs.remove[i](vnode, rm)
}

```

Após essa estrutura condicional, é criada uma estrutura de repetição que tem a duração definida pelo tamanho da propriedade **remove** de **cbs**, e chama a função incorporada pelo elemento no índice **i** da propriedade **remove** de **cbs**.

```

if (isDef((i = vnode.data.hook)) && isDef((i = i.remove))) {
  i(vnode, rm)
}

```

Após essa iteração, é checado se o valor de **vnode.data.hook.remove** é definido, atribuindo seu valor para **i**, e chamando a função incorporada a **i**.

```

else {
  rm()
}

```

Caso o valor de **vnode.data.hook.remove** seja indefinido, é executada a função internalizada em **rm**.

```
else {
  removeNode(vnode.elm)
}
```

Por fim, caso a primeira estrutura condicional da função seja falsa, é feita uma chamada direta para a função **removeNode()** com o valor da propriedade **elm** de **vnode**, o que marca o fim da função **removeVnodes()** e o fim da análise das funções que compõem o bloco de *patch* de **patchVnode()**.

```
else if (oldVnode.text !== vnode.text) {
  nodeOps.setTextContent(elm, vnode.text)
}
if (isDef(data)) {
  if (isDef((i = data.hook)) &&
    isDef((i = i.postpatch))) i(oldVnode, vnode)
}
```

Por fim, é feita a definição do bloco de *pós-patch* de **patchVnode()**. A primeira parte cria uma estrutura condicional que verifica se o valor da propriedade **text** de **oldVnode** e **vnode** são diferentes, caso o valor de **vnode.text** seja definido. Caso a avaliação seja verdadeira, é feita uma chamada para o método **setTextContent()** de **nodeOps** com os valores de **elm** e **vnode.text**. Após isso, é verificado se o valor de **data** é definido. Caso seja, é então verificado se o valor de **data.hook.postpatch** é definido, atribuindo o valor desta propriedade para **i**, e executando a função de execução de **hooks pós-patch** pela incorporação feita à variável **i**, marcando assim o fim da **patchVnode()**.

Ao voltar para a função **patch()**, caso a avaliação da estrutura condicional anterior seja avaliada como falsa, o próximo fluxo de código é executado:

```
else {
  if (isRealElement) {
    if (oldVnode.nodeType === 1 && oldVnode.hasAttribute(SSR_ATTR)) {
      oldVnode.removeAttribute(SSR_ATTR)
      hydrating = true
    }
  }
}
```

É criada uma estrutura condicional que verifica se o valor de **isRealElement** é verdadeiro. Caso seja, é então verificado se o valor da propriedade **nodeType** de **oldVnode** é igual a um e se a propriedade **SSR_ATTR** está contida em **oldVnode**, removendo o valor de **SSR_ATTR** de **oldVnode** e atribuindo o valor de verdadeiro à variável **hydrating**.

```

if (isTrue(hydrating)) {
  if (hydrate(oldVnode, vnode, insertedVnodeQueue)) {
    invokeInsertHook(vnode, insertedVnodeQueue, true)
    return oldVnode
  } else if (__DEV__) {
    warn(
      'The client-side rendered virtual DOM tree \
      is not matching ' +
      'server-rendered content. This is likely \
      caused by incorrect ' +
      'HTML markup, for example nesting block-level \
      elements inside ' +
      '<p>, or missing <tbody>. Bailing hydration and \
      performing ' +
      'full client-side render.'
    )
  }
}
oldVnode = emptyNodeAt(oldVnode)

```

Depois é verificado se o valor de **hydrating** é verdadeiro, e caso seja, é feita uma chamada para **hydrate()** que é avaliada por uma estrutura condicional, e caso seja avaliada como verdadeira, é feita uma execução da função **invokeInsertHook()** e é retornado o valor da função **oldVnode** pela função. Caso a chamada para **hydrate()** retorne um valor falso, é verificado se o *Vue* está em perfil de desenvolvimento, e caso o valor dessa verificação seja verdadeiro, é emitido um aviso acerca da dessincronia dos valores das árvores virtuais de elementos entre o cliente e o servidor. Após isso, é assinalado à variável **oldvnode** o valor da chamada para **emptyNodeAt()**, que é executada, como o comentário indica, quando a árvore de elementos não está renderizada no servidor ou a hidratação falhou.

```

function invokeInsertHook(vnode, queue, initial) {
  if (isTrue(initial) && isDef(vnode.parent)) {
    vnode.parent.data.pendingInsert = queue
  } else {
    for (let i = 0; i < queue.length; ++i) {
      queue[i].data.hook.insert(queue[i])
    }
  }
}

```

A definição de **invokeInsertHook()** tem como parâmetros os valores de **vnode**, **queue** e **initial** e não retorna valores. A função se inicia com uma estrutura condicional que checa se o valor de **initial** é verdadeiro e se o valor da propriedade **parent** de **vnode** é definido. Caso seja verdade, o valor da propriedade **parent.data.pendingInsert** é definido com o valor de **queue**. Caso contrário é criada uma iteração que tem a duração definida pelo tamanho de **queue**, que chama o método interno **data.hook.insert()** de cada índice **i** de **queue**.

```
function emptyNodeAt(elm) {
  return new VNode(nodeOps.tagName(elm).toLowerCase(),
    {}, [], undefined, elm)
}
```

A função **emptyNodeAt()**, cria um novo **Vnode** vazio e retorna ele como valor.

```
const oldElm = oldVnode.elm
const parentElm = nodeOps.parentNode(oldElm)

createElm(
  vnode,
  insertedVnodeQueue,
  oldElm._leaveCb ? null : parentElm,
  nodeOps.nextSibling(oldElm)
)
```

Após a definição da estrutura condicional que checa se o valor de **isRealElement** é verdadeiro, há a definição da constante **oldElm** com o valor da propriedade de mesmo nome pertencente a **oldVnode** e da constante **parentElm** com o valor da chamada para o método **parentNode()** de **nodeOps**. Após isso é feita uma chamada para **createElm()**.

```
if (isDef(vnode.parent))
```

Após essas instruções, é criada uma estrutura condicional que verifica se o valor da propriedade **parent** de **vnode** é definido.

```
let ancestor = vnode.parent
const patchable = isPatchable(vnode)
while (ancestor)
```

Dentro da estrutura condicional, é criada uma variável **ancestor** com o valor da propriedade **parent** de **vnode** e uma constante **patchable** com o valor da chamada para a função **isPatchable()**. Após isso, é definida uma estrutura de repetição que dura enquanto o valor de **ancestor** for definido.

```

for (let i = 0; i < cbs.destroy.length; ++i) {
  cbs.destroy[i](ancestor)
}

```

É então criado uma estrutura de repetição que tem a duração definida pelo tamanho da propriedade **destroy** de **cbs**, e que itera pelos valores de **cbs.destroy** usando um índice **i**, e chamando seus valores de funções internas.

```

ancestor.elm = vnode.elm
if (patchable) {
  for (let i = 0; i < cbs.create.length; ++i) {
    cbs.create[i](emptyNode, ancestor)
  }
  const insert = ancestor.data.hook.insert
  if (insert.merged) {
    const cloned = insert.fns.slice(1)
    for (let i = 0; i < cloned.length; i++) {
      cloned[i]()
    }
  }
}

```

Após isso, é definido o valor da propriedade **elm** de **ancestor** com o valor da propriedade **elm** de **vnode**. É então verificado se o valor de **patchable** é verdadeiro, e caso seja, é feita uma estrutura de repetição que tem a duração definida pelo tamanho da propriedade **create** de **cbs**, e itera pelos valores de **cbs.create** usando um índice **i** e chamando suas funções internas. Após essa estrutura de repetição, é criada uma constante **insert** que recebe o valor de **ancestor.data.hook.insert**. É então verificado se o valor da propriedade **merged** de **insert** tem valor definido, e caso seja, é criada uma constante **cloned** com o valor das funções de **inserted** em um **Array**, menos a primeira função. Após isso é criada uma estrutura de repetição que tem a duração definida pelo tamanho de **cloned** e chama essas funções internas de **cloned()**.

```

if (isDef(parentElm)) {
  removeVnodes([oldVnode], 0, 0)
} else if (isDef(oldVnode.tag)) {
  invokeDestroyHook(oldVnode)
}

```

Após a definição da estrutura condicional que verifica se **vnode.parent** é definido, é então criada uma estrutura condicional que verifica se o valor de **parentElm** é definido,

chamando a função **removeVnodes()** caso verdadeiro, ou se falso, é verificado se o valor da propriedade **tag** de **oldVnode** é definida, chamando **invokeDestroyHook()**.

```
invokeInsertHook(vnode, insertedVnodeQueue, isInitialPatch)
return vnode.elm
```

Ao final de **patch()**, é feita uma chamada para **invokeInsertHook()**, com os valores de **vnode**, **insertedVnodeQueue** e **isInitialPatch** como parâmetros, e há o retorno do valor de **vnode.elm** pela função **patch()**, marcando o final definitivo da funcionalidade de **patch**.

A.6 Mixin de renderização

Define o valor de **Vue.prototype._render** e algumas funções auxiliares.

```
export function renderMixin(Vue: typeof Component)
```

A definição de **renderMixin()**, o último *Mixin* a ser iniciado pelo *Vue*, toma como parâmetro a instância do *Vue* que chamou esta função em formato de **Component**.

```
installRenderHelpers(Vue.prototype)
```

No início da função, há a chamada para a função **installRenderHelpers()**, que define uma série de funções auxiliares para **Vue.prototype**.

```
export function installRenderHelpers(target: any) {
  target._o = markOnce
  target._n = toNumber
  target._s = toString
  target._l = renderList
  target._t = renderSlot
  target._q = looseEqual
  target._i = looseIndexOf
  target._m = renderStatic
  target._f = resolveFilter
  target._k = checkKeyCodes
  target._b = bindObjectProps
  target._v = createTextVNode
  target._e = createEmptyVNode
  target._u = resolveScopedSlots
  target._g = bindObjectListeners
  target._d = bindDynamicKeys
  target._p = prependModifier
}
```

Na função **installRenderHelpers()** é possível observar ver quais são as funções auxiliares e para que método de **Vue.prototype** elas são definidas.

```
Vue.prototype.$nextTick = function (fn: (...args: any[]) => any) {
  return nextTick(fn, this)
}
```

Após a chamada para **installRenderHelpers()**, é definido o método **\$nextTick()** de **Vue.prototype**, com um parâmetro **fn** de tipo função que recebe uma quantidade N de argumentos de qualquer tipo e retorna qualquer tipo. Essa função retorna uma chamada para a função **nextTick()**, executando a função **fn** na próxima rodada de execuções.

```
Vue.prototype._render = function (): VNode
```

Após a definição de **\$nextTick()**, há a definição de **_render()**, a última definição de método de **Vue.prototype**. A sua definição não recebe parâmetros e retorna um valor de tipo **Vnode**.

```
const vm: Component = this
const { render, _parentVnode } = vm.$options
```

É então definido uma constante **vm** com o valor da instância de *Vue* que chamou este método, e duas constantes **render** e **_parentVnode** com os valores desconstruídos de **render** e **_parentVnode** de **vm.\$options**.

```
if (_parentVnode && vm._isMounted) {
  vm.$scopedSlots = normalizeScopedSlots(
    vm.$parent!,
    _parentVnode.data!.scopedSlots,
    vm.$slots,
    vm.$scopedSlots
  )
  if (vm._slotsProxy) {
    syncSetupSlots(vm._slotsProxy, vm.$scopedSlots)
  }
}
```

É então verificado se o valor de **_parentVnode** é definido e se o valor da propriedade **_isMounted** de **vm** é definido. Caso essa verificação seja avaliada como verdadeira, o valor da propriedade **\$scopedSlots** é definido com o valor da chamada da função para **normalizeScopedSlots()**. Após isso, é verificado se o valor da propriedade **_slotsProxy** é definido, e caso seja, é feita uma chamada para **syncSetupSlots()** com os valores de **vm._slotsProxy** e **vm.\$scopedSlots**.

```

export function syncSetupSlots(to: any, from: any) {
  for (const key in from) {
    to[key] = from[key]
  }
  for (const key in to) {
    if (!(key in from)) {
      delete to[key]
    }
  }
}

```

A função **syncSetupSlots()** toma como parâmetros os valores **to** e **from**, ambas de tipo qualquer e não retorna nenhum valor. No início da função, é definido uma estrutura de repetição que faz uma iteração sobre os valores de **from** usando **key** para capturar o valor atual. Essa estrutura define um valor na chave **key** de **to** com o valor da chave **key** de **from**. Após isso, há a definição de uma estrutura de repetição que faz uma iteração sobre os valores de **to** usando **key** para capturar o valor atual do iterador. Em cada repetição, é definido uma estrutura condicional que verifica se **key** não está definido em **from**, e caso essa avaliação seja verdadeira, o valor de **key** é removido de **to**.

```

.$vnode = __parentVnode!
const prevInst = currentInstance
const prevRenderInst = currentRenderingInstance
let vnode

```

Voltando a **__render()**, a definição do valor da propriedade **\$vnode** é feita com o valor de **__parentVnode** caso este não seja nulo ou indefinido. Depois são definidas duas constantes **prevInst** com o valor de **currentInstance** e **prevRenderInst** com os valores de **currentInstance** e **currentRenderingInstance** respectivamente, e uma variável **vnode** sem a definição de um valor.

```

try {
  setCurrentInstance(vm)
  currentRenderingInstance = vm
  vnode = render.call(vm.__renderProxy, vm.$createElement)
}

```

Após isso é criado uma estrutura de tratamento de erros que tenta executar **setCurrentInstance()** para definir o valor de **vm** como a instância a ser avaliada a seguir, define o valor de **currentRenderingInstance** com o valor de **vm** e define o valor de **vnode** com o valor da chamada de **render.call()** com os valores das propriedades **__renderProxy** e **\$createElement** de **vm**.

```

catch (e: any) {
  handleError(e, vm, `render`)
  if (__DEV__ && vm.$options.renderError) {
    try {
      vnode = vm.$options.renderError.call(
        vm._renderProxy,
        vm.$createElement,
        e
      )
    } catch (e: any) {
      handleError(e, vm, `renderError`)
      vnode = vm._vnode
    }
  } else {
    vnode = vm._vnode
  }
}

```

Caso a estrutura capture exceções, elas serão definidas em `e`, e uma chamada para **handleError()** será realizada. Opcionalmente, se o perfil de desenvolvimento do *Vue* for confirmado, e o valor de **vm.\$options.renderError** for definido, haverá a criação de outra estrutura de tratamento de erros, que tentará definir o valor de **vnode** com o valor da chamada para **vm.\$options.renderError.call()**. Caso essa chamada falhe, as exceções serão capturadas em uma variável `e`, e haverá a tentativa de apaziguar as exceções com outra chamada para **handleError()**, e o valor de **vnode** será definido com o valor de **vm._node**. Caso o *Vue* não esteja em perfil de desenvolvimento, o valor de **vnode** será definido com o valor de **vm._vnode**.

```

finally {
  currentRenderingInstance = prevRenderInst
  setCurrentInstance(prevInst)
}

```

Ao final da estrutura de tratamento de erros, é definido o valor de **currentRenderingInstance** com o valor de **prevRenderInst**, e é feita uma nova chamada para **setCurrentInstance()** para definir o valor da instância a ser avaliada com o valor de **prevInst**.

```

if(isArray(vnode) && vnode.length === 1) {
  vnode = vnode[0]
}

```

Após isso, é criada uma estrutura condicional que checa se **vnode** é um **Array** e se o tamanho de **vnode** é igual a 1, definindo o valor de **vnode** com o valor do índice zero dele mesmo caso a expressão seja avaliada como verdadeira.

```
if (!(vnode instanceof VNode)) {
  if (__DEV__ && isArray(vnode)) {
    warn(
      'Multiple root nodes returned from render \
function. Render function ' +
      'should return a single root node.',
      vm
    )
  }
  vnode = createEmptyVNode()
}
```

Após essa checagem, é verificado se o valor de **vnode** não é uma instância de **Vnode**. Caso esse seja o caso, o perfil de desenvolvimento do *Vue* for confirmado e **vnode** for um **Array**, é emitido um aviso ao desenvolvedor de que múltiplos nós raiz foram gerados pela função de renderização. Após isso é definido o valor de **vnode** com o valor de retorno da função **createEmptyVNode()**.

```
export const createEmptyVNode = (text: string = "") => {
  const node = new VNode()
  node.text = text
  node.isComment = true
  return node
}
```

A função **createEmptyVNode()** é definida com um parâmetro **text** de tipo **string** que é definido como uma **string** vazia por padrão, para o caso do parâmetro ser omitido. A função cria uma constante **node** com o valor da criação de um novo **Vnode**, define os seus valores **text** com o valor do parâmetro de mesmo nome e o valor de **isComment** com um valor verdadeiro e devolve este **vnode** como valor de retorno da função.

```
vnode.parent = __parentVnode
return vnode
```

Ao final da função, o valor da propriedade **parent** de **vnode** é definido com o valor de **__parentVnode**, e o valor de **vnode** é retornado pela função, marcando o final do *Mixin* de renderização.

A.7 Um resumo do funcionamento do Vue

Por fim, será feito um resumo simples do funcionamento do *Vue*, com base em todo o código analisado nas seções anteriores.

A.7.1 Inicialização

No momento que uma instância do *Vue* é criada, há a inicialização dos dados, propriedades computadas, a orquestração do sistema de reatividade, a criação do sistema de vigias para dependências, a declaração dos inscritos para vigias e a implementação do sistema de dados dinâmicos.

A.7.2 Compilação de Templates

Em seguida, o *Vue* compila os *templates* em funções de renderização, que convertem um formato de HTML declarativo em funcionalidades *JavaScript* que alteram o documento HTML de uma página diretamente.

A.7.3 Reatividade e o Documento Virtual

O *vue* rastreia as mudanças nos dados de um documento e utiliza um documento virtual, também chamado de *vdome*, para calcular de maneira simples e eficiente o menor número de mudanças necessárias para atualizar o documento real de uma página, assegurando assim, uma alta performance.

A.7.4 Ciclos de Vida

A partir do que é chamado de anzóis de ciclos de vida, tais como as funcionalidades *created*, *mounted*, *updated* e *destroyed*, o *Vue* permite aos desenvolvedores a execução de lógica customizada dentro de diferentes partes do ciclo de vida de um componente em *Vue*, permitindo assim maior controle durante o desenvolvimento de aplicações.

APÊNDICE B – FUNCIONAMENTO INTERNO DO HTMX

Neste trabalho, será feita a análise do funcionamento interno da biblioteca HTMX, com a linguagem TypeScript. O código está disponível em formato de repositório Git na URL <<http://github.com/bigskysoftware/htmx>>. Todos os códigos apresentados neste trabalho são cópias dos códigos apresentados neste repositório, ora idênticas ao material original, outrora formatados para melhor leitura deste trabalho, mas em momento nenhum sem alterar a compreensão ou o funcionamento ideal do código.

B.1 Ponto de Entrada

Ao abrir o arquivo 'package.json', diferentemente do Vue, não é possível encontrar um *script* com nome de 'build'. Isso se deve ao fato que o HTMX é desenvolvido com a linguagem JavaScript ao invés da linguagem TypeScript.

Por esse motivo, ao analisar os arquivos contidos no diretório 'dist' ou em 'src', será notado que todas os arquivos relacionados ao código fonte possuem o mesmo conteúdo, alguns com uma estrutura diferente.

Para essa leitura, será analisado o arquivo 'src/htmx.js'. O código fonte do HTMX é organizado apenas em um único arquivo, que possui aproximadamente 5000 linhas, com todas as suas funcionalidades agrupadas, e o uso do *NPM* é feito apenas para a miniaturização do código, ou seja, o processo de remover espaços em branco e substituir caracteres especiais de novas linhas por caracteres de ponto e vírgula, para garantir o funcionamento do código.

Essa miniaturização é feita para garantir uma execução mais rápida do HTMX, uma vez que o interpretador de *JavaScript* do navegador não precisará se preocupar em analisar a sintaxe de expressões se baseando nos casos onde uma expressão ocupa uma nova linha sem a delimitação utilizando chaves ou parênteses para limitar esta expressão, e também para reduzir o tamanho final do arquivo, diminuindo a carga do tráfego deste arquivo do servidor para o cliente.

```
var htmx = (function() {
  'use strict'
```

No preciso começo do arquivo, há a definição da variável **htmx**, que delimita o **namespace** da biblioteca.

```
const htmx = {
  onLoad: null,
  process: null,
```

Após essa definição, é feita uma definição de uma constante **htmx** interna que age como a *API* pública do **HTMX**, e são inicializadas várias propriedades com valores nulos ou de tipo primitivo ou com uma função já declarada dentro deste objeto. Essa inicialização é feita, pois caso algum desenvolvedor use o TypeScript para utilizar a API do HTMX, o compilador irá indicar erros de definição das funções e valores, portanto esta devida inicialização é feita após essa definição.

```
htmx.onLoad = onLoadHelper
htmx.process = processNode
htmx.on = addEventListenerImpl
htmx.off = removeEventListenerImpl
htmx.trigger = triggerEvent
htmx.ajax = ajaxHelper
htmx.find = find
htmx.findAll = findAll
htmx.closest = closest
htmx.remove = removeElement
htmx.addClass = addClassToElement
htmx.removeClass = removeClassFromElement
htmx.toggleClass = toggleClassOnElement
htmx.takeClass = takeClassForElement
htmx.swap = swap
htmx.defineExtension = defineExtension
htmx.removeExtension = removeExtension
htmx.logAll = logAll
htmx.logNone = logNone
htmx.parseInterval = parseInterval
htmx._ = internalEval
```

Aqui, são finalmente definidos os valores das funções da *API* pública do **HTMX**. Para o funcionamento interno do **HTMX**, não é necessário uma análise profunda das funcionalidades, pois a maioria não será utilizado nas provas de conceito deste trabalho. As funções que serão observadas diretamente são: **on()**, **off()**, **onLoad()**, **addClass()**, **removeClass()**, **toggleClass()**, **parseInterval()** e **swap()**, mas não será limitado a apenas estas, uma vez que funções que são executadas diretamente dentro destas também serão analisadas. Uma dessas funções é **triggerEvent()**, responsável diretamente pela parte de assinalamento de eventos para componentes em uma página *HTML*.

Mesmo com o planejamento das funcionalidades que serão observadas, é necessário observar também como o **HTMX** é inicializado pelo navegador no contexto da página *HTML*. Após a definição do **HTMX**, é iniciada a definição das utilidades do **HTMX**, tanto

da *API* interna quanto da externa. Isso indica que a inicialização do *HTMX* provavelmente acontecerá ao final do arquivo *JavaScript*.

```
return htmx
})();
```

Ao chegar ao final da definição da constante *HTMX* e ao final da definição da variável *HTMX*, ocorre o retorno da constante como valor para a variável. Logo acima deste retorno é executada uma função **ready()**, fora do escopo de uma função, o que indica que ela será executada quando o *HTMX* for inicializado, indicando que é nesta função em que o *HTMX* será iniciado.

B.2 Inicialização do HTMX

```
ready(function()
```

A chamada para a função **ready()** usa um valor de função como parâmetro único. Antes de analisar o corpo da função declarada dentro desta chamada, será analisada a definição de **ready()**.

```
var isReady = false
getDocument().addEventListener('DOMContentLoaded', function() {
    isReady = true
})
```

```
function ready(fn) {
    if (isReady || getDocument().readyState === 'complete') {
        fn()
    } else {
        getDocument().addEventListener('DOMContentLoaded', fn)
    }
}
```

A definição de **ready()** toma como parâmetro um único valor **fn** de função, mas fora de seu corpo uma variável é definida uma variável **isReady** com o valor inicial definido como falso, e uma inicialização de escuta de evento **DOMContentLoaded** no documento, utilizado **getDocument()**, que define o valor da função **isReady** para verdadeiro quando o evento é executado. O corpo de **ready()** verifica se o valor de **isReady** é verdadeiro ou se o valor da propriedade **readyState** do documento, invocado através da chamada de **getDocument()** tem o valor de completo. Caso essa expressão seja verdadeira, é feita uma chamada para a função passada como parâmetro, caso a expressão seja falsa, é criado

uma escuta de eventos, similar a que foi definida acima desta função, escutando o evento **DOMContentLoaded** e executando a função **fn** quando o evento for disparado. A checagem adicional para o valor de **readyState** do documento é feita como uma prevenção ao acaso em que a definição da **tag** de **script** do *HTMX* foi introduzida ao documento em qualquer momento após a inicialização completa do documento.

```
function getDocument() {
    return document
}
```

A função **getDocument()** é uma função que simplesmente retorna o valor do documento *HTML*.

```
mergeMetaConfig()
insertIndicatorStyles()
```

Ao voltar para a execução de **ready()** há a definição da função que é passada como parâmetro. No seu início, duas funções são executadas, **mergeMetaConfig()** e **insertIndicatorStyles()**.

```
function mergeMetaConfig() {
    const metaConfig = getMetaConfig()
    if (metaConfig) {
        htmx.config = mergeObjects(htmx.config, metaConfig)
    }
}
```

A definição de **mergeMetaConfig()** define uma função sem parâmetros, e se inicia criando uma constante **metaConfig()** com o valor de retorno da função **getMetaConfig()**. Após isso, é verificado se o valor de **metaConfig** é nulo, e caso não seja, é definido o valor da propriedade **config** da constante global **htmx** com o valor de retorno da função **mergeObjects()**.

```
function getMetaConfig() {
    const element = getDocument().querySelector('meta[name="htmx-config"]')
    if (element) {
        return parseJSON(element.content)
    } else {
        return null
    }
}
```

A função **getMetaConfig()** é definida sem parâmetros, e no seu início é definida uma constante **element** que recebe o valor retornado pelo método **querySelector()** de **document**, com o valor das configurações meta que tem o nome definido como 'htmx-config'. Após isso é verificado se o valor de **element** é definido, retornando o valor de retorno da função **parseJSON()** com o valor de **element.content** como retorno, ou retornando um valor nulo caso o valor de **element** seja indefinido ou nulo.

```
function parseJSON(jString) {
  try {
    return JSON.parse(jString)
  } catch (error) {
    logError(error)
    return null
  }
}
```

A definição da função **parseJSON()** é feita com um único parâmetro **jString**. No seu início é definida uma estrutura de tratamento de erros que executa a função **JSON.parse()**, da *API* padrão do *JavaScript*, e tenta retornar diretamente seu valor pela função. Caso a execução de **parse()** gere alguma exceção, é retornado um valor, capturado por **error** e esse valor é passado para a função **logError()** como parâmetro, e um valor nulo é retornado pela função.

```
function logError(msg) {
  console.error(msg)
}
```

A função **logError()** recebe um único parâmetro **msg**, e chama **console.error()**, da *API* padrão do *JavaScript*.

```
function mergeObjects(obj1, obj2) {
  for (const key in obj2) {
    if (obj2.hasOwnProperty(key)) {
      obj1[key] = obj2[key]
    }
  }
  return obj1
}
```

Por fim, para compreender o funcionamento de **mergeMetaConfig()**, falta apenas compreender o funcionamento de **mergeObjects()**. A sua definição é feita com os parâmetros **obj1** e **obj2**. No corpo da função, é definida uma estrutura de repetição que itera pelas

chaves de propriedade de **obj2**, e verifica se este objeto possui a chave **key**, definindo o valor da chave **key** em **obj1** com o valor da mesma chave em **obj2**. Após a execução da estrutura de repetição, é retornado o valor de **obj1**.

```
function insertIndicatorStyles() {
  if (htmx.config.includeIndicatorStyles !== false) {
    const nonceAttribute = htmx.config.inlineStyleNonce ?
    `nonce="${htmx.config.inlineStyleNonce}"` : ""
    const indicator = htmx.config.indicatorClass
    const request = htmx.config.requestClass
    getDocument().head.insertAdjacentHTML('beforeend',
      `

```

Após a execução de **mergeMetaConfig()**, é realizada a execução de **insertIndicatorStyles()**. Esta função não possui parâmetros, e no seu início verifica se o valor da configuração **includeIndicatorStyles** de **htmx** possui qualquer valor diferente de falso. Caso a avaliação seja verdadeira, é criada uma constante **nonceAttribute** que recebe dois possíveis valores: uma **string** formatada com o valor da configuração **inlineStyleNonce**, e uma **string** vazia, sendo o valor de configuração **inlineStyleNonce** verdadeiro ou falso, respectivamente.

É então feita uma adição ao documento *HTML* utilizando **insertAdjacentHTML()**, antes do final do documento, uma estilização de *CSS*, que recebe os valores de **nonceAttribute** como valor para a propriedade **nonce** da *tag style*, um valor **indicatorClass** que tem a sua opacidade definida com o valor de 0, e repetidamente os valores **requestClass** e **indicatorClass** que tem a sua opacidade definida com o valor de 1 e tem uma animação de transição definida com a mudança de opacidade suavizada com duração de 200 milissegundos.

```
let body = getDocument().body
processNode(body)
```

Após a execução de **insertIndicatorStyle()** é definida uma variável **body** com o valor da propriedade **body** do documento, e é feita uma execução da função **processNode()** com o valor de **body** como parâmetro.

```
function processNode(elt) {
  elt = resolveTarget(elt)
```

A função **processNode** toma como parâmetro um valor **elt**. No início da função, é definido o valor de **elt** com o valor da chamada para **resolveTarget()**.

```
function resolveTarget(eltOrSelector, context) {
  if (typeof eltOrSelector === 'string') {
    return find(asParentNode(context) || document, eltOrSelector)
  } else {
    return eltOrSelector
  }
}
```

A função **resolveTarget()** tem como parâmetros os valores **eltOrSelector** e **context**. No seu início é verificado se o tipo de **eltOrSelector** é **string**, retornando o valor de **find()** com os parâmetros de **asParentNode()** ou **document** e **eltOrSelector** caso este seja uma **string**, ou retornando o próprio valor de **eltOrSelector** caso contrário.

```
function asParentNode(elt) {
  return elt instanceof Element ||
    elt instanceof Document ||
    elt instanceof DocumentFragment ? elt : null
}
```

A definição de **asParentNode()** define uma função com um parâmetro **elt** que retorna um valor baseado na verificação do valor de **elt**: caso **elt** seja de tipo **Element**, **Document** ou **DocumentFragment**, o valor de **elt** é retornado, caso **elt** não tenha nenhum desses valores, um valor nulo é retornado.

```
function find(eltOrSelector, selector) {
  if (typeof eltOrSelector !== 'string') {
    return eltOrSelector.querySelector(selector)
  } else {
    return find(getDocument(), eltOrSelector)
  }
}
```

A função **find()** define uma função com os parâmetros **eltOrSelector** e **selector**. Na função, é verificado se o valor de **eltOrSelector** é diferente de um tipo **string**, retornando o valor do método **querySelector()** de **eltOrSelector** e usando o valor de **selector** como parâmetro. Caso a expressão anterior seja falsa, é feita uma chamada recursiva que retorna o valor de **find()**. Estas duas definições de função delimitam o funcionamento de **resolveTarget()**.

```
if (eltIsDisabled(elt)) {
    cleanUpElement(elt)
    return
}
```

Ao voltar para **processNode()**, há a verificação do valor de retorno da função **eltIsDisabled()** com o uso de **elt** como parâmetro, e caso o valor seja verdadeiro, é feita uma chamada para **cleanUpElement()** com o valor de **elt**.

```
function eltIsDisabled(elt) {
    return closest(elt, htmx.config.disableSelector)
}
```

A função **eltIsDisabled()** tem como parâmetro um valor **elt** e retorna o valor de chamada para a função **closest()**, utilizando o valor de **elt** e da configuração **disableSelector** da constante **htmx**.

```
function closest(elt, selector) {
    elt = asElement(resolveTarget(elt))
    if (elt) {
        return elt.closest(selector)
    }
    return null
}
```

A função **closest()** tem como parâmetros os valores **elt** e **selector**, e no seu início é definido o valor de **elt** com o valor da chamada para **asElement()**. É então verificado se o valor de **elt** é definido, retornando o valor da chamada para seu método **closest()** com o uso de **selector** como parâmetro caso seja definido, ou retornando um valor nulo caso o valor seja indefinido.

```
function asElement(elt) {
    return elt instanceof Element ? elt : null
}
```

A função **asElement()** toma como parâmetros o valor de **elt**, e retorna um valor baseado na verificação do tipo de **elt**: caso seja uma instância de **Element**, o seu próprio valor é retornado, caso contrário um valor nulo é retornado.

```
function cleanUpElement(element) {
  triggerEvent(element, 'htmx:beforeCleanupElement')
  delInitNode(element)
  forEach(element.children, function(child) { cleanUpElement(child) })
}
```

Por fim, é feita uma chamada para **cleanUpElement()**. A função toma como parâmetro o valor de **element**, e executa: **triggerEvent()**, **delInitNode()** e para cada valor da propriedade **children** de **element**, é feita uma execução recursiva de **cleanUpElement()**.

```
function triggerEvent(elt, eventName, detail) {
```

A definição de **triggerEvent()** toma como parâmetros os valores **elt**, **eventName** e **detail** como parâmetros.

```
  elt = resolveTarget(elt)
  if (detail == null) {
    detail = {}
  }
  detail.elt = elt
```

É definido o valor de **elt** com o valor de retorno da chamada para **resolveTarget()**. Após isso é verificado se o valor de **detail** é nulo, definindo o valor de **detail** como um objeto vazio caso seja. Após isso, é definido o valor da propriedade **elt** de **detail** com o valor do parâmetro **elt**.

```
const event = makeEvent(eventName, detail)
```

Após isso, é criada uma constante **event** com o valor da chamada para **makeEvent()**, utilizando **eventName** e **detail** como parâmetros.

```
function makeEvent(eventName, detail) {
  return new CustomEvent(eventName,
    { bubbles: true, cancelable: true, composed: true, detail })
}
```

A função **makeEvent()** toma como parâmetros os valores **eventName** e **detail**, e retorna um novo valor da classe **CustomEvent**, utilizando o valor de **eventName** e de **detail** dentro de um objeto como parâmetros.

```
if (htmx.logger && !ignoreEventForLogging(eventName)) {
  htmx.logger(elt, eventName, detail)
}
```

Após isso, é verificado se o valor de **htmx.logger** é definido e se o valor da chamada para **ignoreEventForLogging()** com o valor de **eventName** como parâmetro é falso. Caso a expressão seja verdadeira, é feita uma chamada para o valor de **htmx.logger()** com os valores de **elt**, **eventName** e **detail**.

```
function ignoreEventForLogging(eventName) {
  return eventName === 'htmx:afterProcessNode'
}
```

A função **ignoreEventForLogging()** toma como parâmetro um valor **eventName** e checka se o valor de **eventName** é igual a **string** **'htmx:afterProcessNode'**, retornando o valor desta checagem. Como o valor de **logger()** ainda está a ser definido pelo desenvolvedor, não há um valor definido, portanto a chamada para **logger()** não será executada neste ponto.

```
if (detail.error) {
  logError(detail.error)
  triggerEvent(elt, 'htmx:error', { errorInfo: detail })
}
```

Após isso é verificado se o valor da propriedade **error** de **detail** é definido, chamando a função **logError()** com este valor e fazendo uma chamada recursiva para **triggerEvent()**.

```
let eventResult = elt.dispatchEvent(event)
const kebabName = kebabEventName(eventName)
```

É criado uma variável **eventResult** com o valor da chamada para o método **dispatchEvent()** de **elt** com o valor de **event** como parâmetro, e é criada uma constante **kebabName** com o valor da chamada para a função **kebabEventName()** com o valor de **eventName**.

```
function kebabEventName(str) {
  return str.replace(/[a-z0-9]+([A-Z])/g, '$1-$2').toLowerCase()
}
```

A função **kebabEventName()** toma como parâmetro um valor de **str** e retorna o valor da chamada para o seu método **replace()** que procura um valor através de um *regex*,

definindo o valor de dois grupos: uma lista que tem caracteres de 'a' a 'z' e zero a nove, e outro que tem uma lista de caracteres de 'a' a 'z', porém maiúsculos. Após a delimitação dos dois grupos, é definido que o valor desses grupos será formatado separando os dois grupos por um caractere '-'. Para melhorar a demonstração deste *regex*, ao tomar como exemplo uma **string** 'issoÉUmTexto', esta chamada para **replace()** transformaria o seu valor para 'isso-é-um-texto'.

```
if (eventResult && kebabName !== eventName) {
  const kebabedEvent = makeEvent(kebabName, event.detail)
  eventResult = eventResult && elt.dispatchEvent(kebabedEvent)
}
```

Após isso, em **triggerEvent()**, é verificado se o valor de **eventResult** é definido e se o valor de **kebabName** é diferente de **eventName**. Caso seja, é criada uma constante **kebabedEvent** com o valor da chamada para **makeEvent()** e é definido o valor de **eventResult** com o valor da chamada para **dispatchEvent()** caso **eventResult** seja definido.

```
withExtensions(asElement(elt), function(extension) {
  eventResult = eventResult &&
    (extension.onEvent(eventName, event) !== false &&
      !event.defaultPrevented)
}))
```

Após isso é feita uma chamada para **withExtensions()** com os valores de **asElement()** e uma função anônima, que define o valor de **eventResult** com o valor da seguinte expressão: caso **eventResult** tenha valor que não seja indefinido ou nulo e se o valor da chamada para **onEvent()** é diferente de falso e se o valor da propriedade **defaultPrevented()** é verdadeiro, é retornado um valor verdadeiro, caso contrário, é retornado um valor falso.

```
function withExtensions(elt, todo, extensionsToIgnore) {
  forEach(getExtensions(elt, [], extensionsToIgnore), function(extension) {
    try {
      todo(extension)
    } catch (e) {
      logError(e)
    }
  })
}
```

A definição de **withExtensions()** toma como parâmetros os valores **elt**, **todo** que terá um tipo de função e **extensionsToIgnore**. Na função, é feita uma iteração sobre os

valores da chamada para **getExtensions()** capturados pela variável **extension**, executando uma função anônima que tenta executar a função **todo()** com o valor de **extension**, e caso alguma exceção seja gerada, é executada a função **logError()**.

```
function getExtensions(elt, extensionsToReturn, extensionsToIgnore) {
  if (extensionsToReturn == undefined) {
    extensionsToReturn = []
  }
  if (elt == undefined) {
    return extensionsToReturn
  }
  if (extensionsToIgnore == undefined) {
    extensionsToIgnore = []
  }
}
```

A função **getExtensions()** toma como parâmetros as variáveis **elt**, **extensionsToReturn** e **extensionsToIgnore**. No seu corpo, é criada uma estrutura condicional que verifica se o valor de **extensionsToReturn** é um valor indefinido e caso seja, o valor de **extensionsToReturn** é definido com um **Array** vazio. Em seguida, outra estrutura condicional verifica se o valor de **elt** é indefinido, e caso seja, o valor de **extensionsToReturn** é definido como valor pela função. Logo após, é verificado se o valor de **extensionsToIgnore** é indefinido, e caso seja, o valor de **extensionsToIgnore** é definido como um **Array** vazio.

```
const extensionsForElement = getAttributeValue(elt, 'hx-ext')
if (extensionsForElement) {
  forEach(extensionsForElement.split(','), function(extensionName) {
    extensionName = extensionName.replace(/ /g, "")
  })
}
```

Logo após, é criada uma constante **extensionsForElement** que recebe o valor de **getAttributeValue()**. Uma estrutura condicional verifica se o valor de **extensionsForElement** é um valor definido, e caso seja, uma iteração sobre os valores de **extensionsForElement** separados nos caracteres de vírgula, capturados em uma variável **extensionName** dentro de uma função anônima. Dentro da função anônima o valor de **extensionName** recebe o valor de **extensionName** sem nenhum caractere de barra.

```
if (extensionName.slice(0, 7) == 'ignore:') {
  extensionsToIgnore.push(extensionName.slice(7))
  return
}
if (extensionsToIgnore.indexOf(extensionName) < 0) {
  const extension = extensions[extensionName]
}
```

```

    if (extension && extensionsToReturn.indexOf(extension) < 0) {
      extensionsToReturn.push(extension)
    }
  }
}

```

Em seguida, é verificado se o valor do pedaço entre o primeiro e o sexto caractere de **extensionName** têm o valor da **string** 'ignore:', e caso tenha, o valor de **extensionName** a partir do sétimo caractere é adicionado ao final de **extensionsToIgnore** e a função retorna sem valores. Após isso, é checado se o número de instâncias de **extensionName** em **extensionsToIgnore** é menor que zero, e caso seja, uma constante **extension** é criada com o valor da instância de **extensionName** em **extensions**. Em seguida, é verificado se o valor de **extension** é definido e se o número de instâncias de **extension** em **extensionsToReturn** é menor que 0, e caso seja, o valor de **extension** é adicionado ao final de **extensionsToReturn**.

```

return getExtensions(asElement(parentElt(elt)),
  extensionsToReturn,
  extensionsToIgnore)

```

Ao final da função, o valor da chamada recursiva para **getExtensions()** é retornado com o valor do elemento pai de **elt**.

```

function delInitNode(element)

```

Observando a implementação de **delInitNode()**, a função possui apenas um parâmetro **element**.

```

const internalData = getInternalData(element)
if (internalData.timeout) {
  clearTimeout(internalData.timeout)
}

```

No início da função é criado uma constante **internalData** com o valor de retorno da chamada para a função **getInternalData()**, utilizando **element** como parâmetro. Após isso, é verificado se o valor da propriedade **timeout** de **internalData** é definido, chamando a função **clearTimeout()** utilizando essa propriedade como parâmetro caso seja verdadeira.

```

function getInternalData(elt) {
  const dataProp = 'htmx-internal-data'
  let data = elt[dataProp]
  if (!data) {
    data = elt[dataProp] = {}
  }
}

```

```

    }
    return data
}

```

A função **getInternalData()** toma como parâmetro um valor **elt** e verifica a existência de um valor 'htmx-internal-data' dentro do elemento providenciado. Caso ele encontre algum valor, ele será retornado pela função, caso não, a função definirá o valor dessa chave como um objeto vazio e retorna esse valor.

declare **function** clearTimeout(id: number | **undefined**): **void**;

Já a função **clearTimeout()** é uma função da biblioteca padrão do *JavaScript*, que remove qualquer valor de **timeout** de uma propriedade fornecida como parâmetro.

```

if (internalData.listenerInfos) {
    forEach(internalData.listenerInfos, function(info) {
        if (info.on) {
            removeEventListenerImpl(info.on, info.trigger, info.listener)
        }
    })
}

```

Voltando para **delInitNode()**, é criada uma estrutura condicional que verifica se a propriedade **listenerInfos** de **internalData** possui um valor definido, fazendo uma verificação da definição de um valor da propriedade **on** de cada informação das escutas, e executando a função **removeEventListenerImpl()** para cada uma delas.

```

function removeEventListenerImpl(arg1, arg2, arg3) {
    ready(function() {
        const eventArgs = processEventArgs(arg1, arg2, arg3)
        eventArgs.target.removeEventListener(eventArgs.event,
            eventArgs.listener)
    })
    return isFunction(arg2) ? arg2 : arg3
}

```

A função **removeEventListenerImpl()** toma como parâmetros os valores **arg1**, **arg2** e **arg3**. Esses valores são, em ordem, o elemento que deverá ter a escuta removida, o nome do evento que deve ser removido da escuta, e a escuta a ser removida. A função faz então uma chamada para a função **ready()**, passando uma função anônima que cria uma constante **eventArgs** que recebe o valor de retorno da chamada para **processEventArgs()**. Após isso é feita uma chamada para **removeEventListener()** na propriedade **target** de

eventArgs, e por fim, a função retorna o valor da **arg2** ou de **arg3** a depender do valor de **arg2**: caso seja uma função, o seu próprio valor é retornado, caso contrário o valor de **arg3** é retornado.

```
function processEventArgs(arg1, arg2, arg3, arg4) {
    if (isFunction(arg2)) {
        return {
            target: getDocument().body,
            event: asString(arg1),
            listener: arg2,
            options: arg3
        }
    } else {
        return {
            target: resolveTarget(arg1),
            event: asString(arg2),
            listener: arg3,
            options: arg4
        }
    }
}
```

A função **processEventArgs()** é uma função que simplesmente verifica se o valor de **arg2** é uma função, e retorna o valor de um objeto com os valores de **target**, **event**, **listener** e **options** de acordo com as checagens.

```
delInitOnHandlers(element)
forEach(Object.keys(internalData),
function(key) { if (key !== 'firstInitCompleted') delete internalData[key] })
```

Após a execução de **removeEventListenerImpl()**, é feita a execução da função **delInitOnHandlers()**, e para cada chave existente em **internalData**, é executada uma função anônima que verifica se o valor da chave é diferente do valor 'firstInitCompleted', removendo o valor dessa chave de **internalData**.

```
function delInitOnHandlers(elt) {
    const internalData = getInternalData(elt)
    if (internalData.onHandlers) {
        for (let i = 0; i < internalData.onHandlers.length; i++) {
            const handlerInfo = internalData.onHandlers[i]
```

```

        removeEventListenerImpl(elt, handlerInfo.event,
                                handlerInfo.listener)
    }
    delete internalData.onHandlers
}
}

```

A função **delInitOnHandlers()** coleta os valores dos dados de um elemento **elt**, passado como parâmetro para a função, utilizando **getInternalData()**. É então verificado se a propriedade **onHandlers** de **internalData** é definida, e é feita uma estrutura de repetição que tem a duração definida pela propriedade **onHandlers** e coleta o valor de informação desta e faz uma chamada para **removeEventListenerImpl()**. Após essa execução, o valor de **internalData.onHandlers** é removido. A explicação desta função encerra o entendimento da função **cleanUpElement()**, o que nos permite voltar para **processNode()**.

```

const elementsToInit = []
if (maybeDelInitAndHash(elt)) {
    elementsToInit.push(elt)
}

```

Após a possível execução de **cleanUpElement()**, é criada uma constante **elementsToInit** que recebe uma lista vazia, e é verificado se o valor de retorno para **maybeDelInitAndHash()** é verdadeiro, utilizando **elt** como parâmetro. Caso essa avaliação seja verdadeira, o valor de **elt** é adicionado ao final de **elementsToInit**.

```

function maybeDelInitAndHash(elt) {
    if (!(elt instanceof Element)) {
        return false
    }

    const nodeData = getInternalData(elt)
    const hash = attributeHash(elt)
    if (nodeData.initHash !== hash) {
        delInitNode(elt)
        nodeData.initHash = hash
        return true
    }
    return false
}

```

A definição de **maybeDelInitAndHash()** toma como parâmetro um valor **elt**. No seu início é verificado se o valor de **elt** é uma instância de **Element**, e caso não seja, um

valor falso é retornado imediatamente pela função, indicando que o valor passado como parâmetro é um elemento de um documento virtual. Após isso, são criadas as constantes **nodeData** e **hash** com os valores de **getInternalData()** e **attributeHash()** respectivamente, e é verificado se o valor da propriedade **initHash** de **nodeData** é diferente de **hash**. Caso seja, é feita uma execução para **delInitNode()** com o valor de **elt**, e o valor da propriedade **initHash** de **nodeData** é definido com o valor de **hash**, e um valor verdadeiro é retornado pela função. Caso a expressão seja falsa, no seu final é retornado um valor falso.

```
function attributeHash(elt) {
  let hash = 0
  for (let i = 0; i < elt.attributes.length; i++) {
    const attribute = elt.attributes[i]
    if (attribute.value) {
      hash = stringHash(attribute.name, hash)
      hash = stringHash(attribute.value, hash)
    }
  }
  return hash
}
```

Das funções executadas em **maybeDelInitAndHash()**, apenas **attributeHash()** ainda não foi vista. A sua implementação toma como parâmetro um valor **elt**. No seu início é definida uma variável **hash** com um valor 0. É criada uma estrutura de repetição que tem a duração definida pelo tamanho da propriedade **attributes** de **elt**. Para cada valor, é criada uma constante **attribute** que captura esse valor, e é verificado se a propriedade **value** de **attribute** é definida, atribuindo o valor de hash com a chamada para a função **stringHash()** duas vezes, uma com o nome do atributo, e outra com o valor deste atributo. Após isso, o valor de **hash** é retornado pela função.

```
function stringHash(string, hash) {
  let char = 0
  while (char < string.length) {
    hash = (hash << 5) - hash + string.charCodeAt(char++) | 0
  }
  return hash
}
```

A função **stringHash()** possui como parâmetros os valores **string** e **hash()**. Essa função irá criar um valor de **hash** utilizando o valor providenciado pelos parâmetros **string** e **hash**. Durante uma estrutura condicional, e enquanto o número de **char** for menor que o tamanho de **string**, o valor de **hash** é atribuído com a seguinte expressão: **hash** com

um **bitshift** à esquerda em 5 casas, menos o valor de **hash**, mais o valor do código do caractere na posição de **char**. Após essa expressão, o valor é operado utilizando o operador *bitwise or* com o valor de 0, para garantir que o valor final seja um inteiro de 32 *bits*. A compreensão de **stringHash()** completa o entendimento de **attributeHash()**, que por sua vez completa a compreensão de **maybeDeInitAndHash()**, nos permitindo voltar para **processNode()**.

```
forEach(findElementsToProcess(elt), function(child) {
    if (eltIsDisabled(child)) {
        cleanUpElement(child)
        return
    }
    if (maybeDeInitAndHash(child)) {
        elementsToInit.push(child)
    }
})
```

Após a provável execução de **maybeDeInitAndHash()**, haverá a verificação do valor **child** de cada elemento, onde é verificado se o valor de **child** é desabilitado e caso seja verdade, é feita uma chamada para **cleanUpElement()** e é verificado se o valor da chamada para **maybeDeInitAndHash()** com o valor de **child** é verdadeiro, adicionando os valores iterativos de **child** ao final de **elementsToInit**. A adição desses valores não irá interferir na ordem de inicialização dos elementos, uma vez que os elementos pais já foram adicionados na lista anteriormente, e os elementos filhos estão sendo adicionados apenas agora.

```
forEach(findHxOnWildcardElements(elt), processHxOnWildcard)
forEach(elementsToInit, initNode)
```

Ao final da função, é feita a iteração sobre dois valores: um sobre os valores de retorno de **findHxOnWildcardElements()** com o valor de **elt**, executando **processHxOnWildcard()** para cada um dos elementos, e uma iteração sobre os valores de **elementsToInit**, executando **initNode** para cada um dos valores.

```
function findHxOnWildcardElements(elt) {
    const elements = []
    if (elt instanceof DocumentFragment) {
        for (const child of elt.childNodes) {
            processHXOnRoot(child, elements)
        }
    } else {
```

```

    processHXOnRoot(elt, elements)
  }
  return elements
}

```

Primeiramente será analisado o funcionamento de **findHxonWildcardElements()**. A função toma como parâmetro um valor **elt**, e no seu início, uma constante **elements** é criada com um valor vazio. Após isso, é verificado se o parâmetro **elt** é uma instância de **DocumentFragment**. Caso seja, é feita uma execução para **processHXOnRoot()** para cada elemento **child** de **elt**. Caso contrário, é feita apenas uma execução para **processHXOnRoot()**. Após essas execuções, o valor de **elements** é retornado pela função.

```

function processHXOnRoot(elt, elements) {
  if (shouldProcessHxOn(elt)) {
    elements.push(asElement(elt))
  }
  const iter = HX_ON_QUERY.evaluate(elt)
  let node = null
  while (node = iter.iterateNext()) elements.push(asElement(node))
}

```

A função **processHXOnRoot()** toma como parâmetros os valores **elt** e **elements**, e no seu início, verifica se o valor de **shouldProcessHXOn()** com o valor de **elt** é verdadeiro, adicionando o valor de **elt** como um elemento caso seja. Após isso é criada uma constante **iter** com o valor da avaliação de **elt** utilizando **HX_ON_QUERY**.

```

const HX_ON_QUERY = new XPathEvaluator()
  .createExpression('//*[ @*[ starts-with(name(),
    "hx-on:") or starts-with(name(), "data-hx-on:") or ' +
    ' starts-with(name(), "hx-on-") or starts-with(name(), "data-hx-on-") ]])'

```

A expressão constante **HX_ON_QUERY** é uma avaliação *JavaScript* feita acerca de um valor de texto. Ela verifica se o valor de cada propriedade possui um valor 'hx-on', 'data-hx-on', 'hx-on-', ou 'data-hx-on-'.

Após essa avaliação é criada uma constante **node** que recebe um valor nulo, e é criada uma estrutura condicional que é executada enquanto o valor de **node** for verdadeiro, e para cada execução, o valor de **iter** recebe o próximo elemento de **node**. Para cada execução, o valor de **node** como um elemento é adicionado ao final de **elements**.

Essa análise conclui o entendimento de **findHXOnWildcardElements()**, portanto agora será analisada a definição de **processHXOnWildcard()**.

```

function processHxOnWildcard(elt) {
  delInitOnHandlers(elt)

  for (let i = 0; i < elt.attributes.length; i++) {
    const name = elt.attributes[i].name
    const value = elt.attributes[i].value
    if (startsWith(name, 'hx-on') || startsWith(name, 'data-hx-on')) {
      const afterOnPosition = name.indexOf('-on') + 3
      const nextChar = name.slice(afterOnPosition, afterOnPosition + 1)
      if (nextChar === '-' || nextChar === ':') {
        let eventName = name.slice(afterOnPosition + 1)
        if (startsWith(eventName, ':')) {
          eventName = 'htmx' + eventName
        } else if (startsWith(eventName, '-')) {
          eventName = 'htmx:' + eventName.slice(1)
        } else if (startsWith(eventName, 'htmx-')) {
          eventName = 'htmx:' + eventName.slice(5)
        }

        addHxOnEventHandler(elt, eventName, value)
      }
    }
  }
}

```

A definição da função toma como parâmetro um valor **elt**, e no seu início é feita uma chamada para **delInitOnHandlers()**. Após a execução de **delInitOnHandlers()**, é realizada a execução de um grande bloco de código. Para cada elemento de atributo em **elt**, é verificado se o atributo tem o nome de 'hx-on' ou de 'data-hx-on' no seu começo. Caso possua, é então verificado se o próximo valor é um caractere de '-' ou de ':'. Caso essa expressão seja verdadeira, é então criada uma variável **eventName** que recebe o valor do nome do evento que vem após a definição do valor anterior. É então feita uma formatação com várias estruturas condicionais de tal modo que o valor de **eventName** possua a estrutura de 'htmx:eventName', substituindo **eventName** pelo nome do evento, e é feita uma execução à função **addHxOnEventHandler()** com os valores de **elt**, **eventName** e **value**.

```

function addHxOnEventHandler(elt, eventName, code) {
  const nodeData = getInternalData(elt)
  if (!Array.isArray(nodeData.onHandlers)) {

```

```
nodeData.onHandlers = []
}
```

A definição de **addHxOnEventHandler()** toma como parâmetros os valores **elt**, **eventName** e **code**. No seu início é criada uma constante **nodeData** com o valor da chamada para **getInternalData()** utilizando o valor de **elt** como parâmetro. Após isso, é verificado se o valor da propriedade **onHandlers** de **nodeData** não é um **Array**, definindo o seu valor como uma lista vazia caso a avaliação seja verdadeira.

```
let func
const listener = function(e) {
  maybeEval(elt, function() {
    if (eltIsDisabled(elt)) {
      return
    }
    if (!func) {
      func = new Function('event', code)
    }
    func.call(elt, e)
  })
}
```

Após essa verificação, é criada uma variável **func** sem valor de inicialização, e uma constante **listener** com o valor de uma função anônima. Essa função anônima toma como parâmetro um valor **e**. Dentro desta função é feita uma chamada para **maybeEval()**, que usa os valores de **elt** e de uma outra função anônima como parâmetro. Esta função anônima interna verifica se o valor de **elt** está desabilitado, retornando imediatamente caso a avaliação seja verdadeira, e verifica se o valor de **func** é definido, alterando seu valor para uma nova função com o valor de 'event' e de **code** como construtores da função. Após isso a função anônima interna se encerra com a chamada para a função **func()**, utilizando **elt** e **e** como parâmetros, encerrando também a função anônima externa.

```
elt.addEventListener(eventName, listener)
nodeData.onHandlers.push({ event: eventName, listener })
```

Após essa definição, é criado uma escuta de evento para **elt**, utilizando o evento em **eventName** e chamando o valor de **listener** como função. Após isso, esses valores são adicionados como objetos no final da lista de **onHandlers**, propriedade de **nodeData**, concluindo a execução de **addHxOnEventHandler()** que conclui a explicação de **processHxWildcard()**.

A última definição que precisa ser analisada em **processNode()** é a de **initNode()**.

```
function initNode(elt) {
  triggerEvent(elt, 'htmx:beforeProcessNode')
```

A função **initNode()** toma como parâmetro o valor **elt**, e imediatamente faz uma chamada de evento com o valor de 'htmx:beforeProcessNode' através de **triggerEvent()**.

```
const nodeData = getInternalData(elt)
const triggerSpecs = getTriggerSpecs(elt)
const hasExplicitHttpAction = processVerbs(elt, nodeData, triggerSpecs)
```

Após isso, são criadas 3 constantes: **nodeData** com o valor da chamada para **getInternalData()**, **triggerSpecs** com o valor de **getTriggerSpecs()** e **hasExplicitHttpAction** que recebe o valor de **processVerbs()**.

A análise de **getInternalData()** já foi realizada, portanto faltam apenas as análises de **triggerSpecs()** e **processVerbs()**.

```
function getTriggerSpecs(elt) {
  const explicitTrigger = getAttributeValue(elt, 'hx-trigger')
  let triggerSpecs = []
  if (explicitTrigger) {
    const cache = htmx.config.triggerSpecsCache
    triggerSpecs = (cache && cache[explicitTrigger]) ||
      parseAndCacheTrigger(elt, explicitTrigger, cache)
  }

  if (triggerSpecs.length > 0) {
    return triggerSpecs
  } else if (matches(elt, 'form')) {
    return [{ trigger: 'submit' }]
  } else if (matches(elt,
    'input[type="button"],
    input[type="submit"]')) {
    return [{ trigger: 'click' }]
  } else if (matches(elt, INPUT_SELECTOR)) {
    return [{ trigger: 'change' }]
  } else {
    return [{ trigger: 'click' }]
  }
}
```

A função **getTriggerSpecs()** toma como parâmetro um valor **elt**. No seu início é criada uma constante **explicitTrigger** que recebe o valor de **getAttributevalue()**, que é

utilizado para coletar o valor do atributo definido em 'hx-trigger'. Após isso, é definido uma variável **triggerSpecs** com uma lista vazia como valor. Após isso, é verificado se o valor de **explicitTrigger** é definido, e caso seja, é definido o valor de uma constante cache que recebe o valor de configuração **triggerSpecsCache** de **htmx** e define o valor de **triggerSpecs** com o valor da chave **explicitTrigger** em cache ou com o valor de **parseAndCacheTrigger()**. Após essa definição, é verificado se o tamanho de **triggerSpecs** é maior que 0, retornando o seu valor pela função. Caso essa seja falsa, é verificado se o tipo do elemento é um formulário, retornando uma lista com um **trigger submit**, ou se o tipo de **elt** é um **input** de tipo **button** ou **submit**, retornando um **trigger click**, ou se **elt** é um **INPUT_SELECTOR**, retornando um valor de **trigger change** e por fim, caso nenhuma dessas avaliações sejam verdadeiras, é retornado um valor de **trigger click**.

```
const INPUT_SELECTOR = 'input, textarea, select'
```

A constante **INPUT_SELECTOR** define os valores de elementos input, textarea e select.

```
function processVerbs(elt, nodeData, triggerSpecs) {
  let explicitAction = false
  forEach(VERBS, function(verb) {
    if (hasAttribute(elt, 'hx-' + verb)) {
      const path = getAttributeValue(elt, 'hx-' + verb)
      explicitAction = true
      nodeData.path = path
      nodeData.verb = verb
      triggerSpecs.forEach(function(triggerSpec) {
        addTriggerHandler(elt,
          triggerSpec,
          nodeData,
          function(node, evt) {
            const elt = asElement(node)
            if (eltIsDisabled(elt)) {
              cleanUpElement(elt)
              return
            }
            issueAjaxRequest(verb, path, elt, evt)
          })
      })
    }
  })
}
```

```

    return explicitAction
}

```

Por ultimo, é definida a função **processVerbs()**, que toma como parâmetros os valores **elt**, **nodeData** e **triggerSpecs**. No início da função, é criada uma variável **explicitAction**, com um valor falso, e então é definida uma iteração sobre o valor da constante **VERBS**. Para cada elemento de **VERBS** é executada uma função anônima que captura esse valor em uma variável **verb**.

```
const VERBS = ['get', 'post', 'put', 'delete', 'patch']
```

É então verificado na função anônima se o valor de 'hx-' com o valor do verbo atual no final está dentro dos atributos do elemento avaliado pela função. Caso essa condição seja verdadeira, é criada uma constante **path** com o valor do atributo 'hx-VERB', e é definido o valor das propriedades **path** e **verb** de **nodeData** com os valores de **path** e **verb** respectivamente e da variável **explicitAction** como verdadeiro. Após isso, é feita uma iteração sobre os valores de **triggerSpecs**: para cada valor **triggerSpec** é chamada uma execução à função **addTriggerHandler()** com os parâmetros **elt**, **triggerSpec**, **nodeData** e uma função anônima com os parâmetros **node** e **evt**. Essa função anônima cria uma constante **elt** com o valor do elemento passado para a função como um **Element**. Após isso é verificado se o valor de **elt** está desabilitado, e caso esteja, é feita uma chamada para **cleanUpElement()** e a função tem seu retorno dentro da estrutura. Caso a estrutura condicional não avalie a expressão como verdadeira, é feita uma chamada para a função **issueAjaxRequest()**.

```
function issueAjaxRequest(verb, path, elt, event, etc, confirmed)
```

A função **issueAjaxRequest()** toma como parâmetros os valores **verb**, **path**, **elt**, **event**, **etc** e **confirmed**. Esta função é responsável por realizar o envio de requisições *HTTP*, porém é um arquivo muito grande, portanto, apenas a análise de certos trechos desta função serão feitos.

```

    let resolve = null
    let reject = null
    etc = etc !== null ? etc : {}
    if (etc.returnPromise && typeof Promise !== 'undefined') {
        var promise = new Promise(function(_resolve, _reject) {
            resolve = _resolve
            reject = _reject
        })
    }
}

```

```

if (elt == null) {
    elt = getDocument().body
}

```

No início da função, são criadas as variáveis **resolve** e **reject**, ambas com valores nulos. Em seguida é atribuído ao valor de **etc** o resultado da seguinte expressão: se **etc** for diferente de um valor nulo, a variável mantém seu valor, caso contrário um objeto vazio é assinalado à função.

Em seguida é criada uma estrutura condicional que verifica se o valor da propriedade **returnPromise** de **etc** é definido e se o tipo de **Promise** é diferente de indefinido. Caso a expressão seja avaliada como verdadeira, é criada uma variável **promise** com o valor de uma nova instância de **Promise**. Em seguida, é verificado se o valor de **elt** é nulo, onde neste caso o valor de **elt** recebe o valor do corpo do documento *HTML*.

Em seguida no arquivo, são feitas várias checagens acerca de valores de configuração e de validade dos dados necessários para realizar a requisição **HTTP**.

```

const xhr = new XMLHttpRequest()
eltData.xhr = xhr
eltData.abortable = abortable
const endRequestLock = function() {
    eltData.xhr = null
    eltData.abortable = false
    if (eltData.queuedRequests != null &&
        eltData.queuedRequests.length > 0) {
        const queuedRequest = eltData.queuedRequests.shift()
        queuedRequest()
    }
}

```

Em seguida é criado uma constante **xhr** com o valor de uma nova instância de **XMLHttpRequest()**. Esta será a constante utilizada para realizar a requisição *HTTP*. Em seguida são atribuídos valores para a configuração da requisição.

```

if (verb !== 'get' && !usesFormData(elt)) {
    headers['Content-Type'] = 'application/x-www-form-urlencoded'
}

```

Depois de algumas verificações, é então verificado se o tipo da requisição *HTTP* é diferente de 'get' e se **elt** não usa dados de formulários através de **usesFormData()**. Caso essa avaliação seja verdadeira, o tipo de conteúdo da requisição é definido como 'application/x-www-form-urlencoded'

```
xhr.send(params)
return promise
```

Ao final da função, a requisição é feita através do método **send()** de **xhr** e o valor da promessa contida em **promisse** é retornado como valor. Para concluir o entendimento de **processVerbs()**, falta compreender a função **addTriggerHandler()**.

```
function addTriggerHandler(elt, triggerSpec, nodeData, handler) {
```

A função toma como parâmetros os valores **elt**, **triggerSpec**, **nodeData** e **handler**.

```
if (triggerSpec.trigger === 'revealed') {
  initScrollHandler()
  addEventListener(elt, handler, nodeData, triggerSpec)
  maybeReveal(asElement(elt))
```

No seu início, é verificado se o valor da propriedade **trigger** de **triggerSpec** é igual à 'revealed'. Caso a verificação seja verdadeira, é feita uma chamada para as funções **initScrollHandler()**, **addEventListener()** e **maybeReveal()**.

```
let windowIsScrolling = false
let scrollHandler = null
function initScrollHandler() {
  if (!scrollHandler) {
    scrollHandler = function() {
      windowIsScrolling = true
    }
    window.addEventListener('scroll', scrollHandler)
    window.addEventListener('resize', scrollHandler)
    setInterval(function() {
      if (windowIsScrolling) {
        windowIsScrolling = false
        forEach(getDocument()
          .querySelectorAll("[hx-trigger*='revealed'],
            [data-hx-trigger*='revealed']"),
          function(elt) {
            maybeReveal(elt)
          })
      }
    }, 200)
  }
}
```

A função **initScrollHandler()** não toma parâmetros, mas faz uso das variáveis globais **windowIsScrolling**, definida com um valor falso e **scrollHandler** que recebe um valor nulo. No início da função é verificado se o valor de **scrollHandler** não é definido, e caso isso seja verdade, o valor de **scrollHandler** é definido como uma função anônima que define o valor de **windowIsScrolling** como verdadeiro. Após isso, são criados dois eventos para a janela da página, utilizando os eventos de redimensionamento e rolagem, ambas chamando a função contida em **scrollHandler**. Logo depois é feita uma chamada para **setInterval()** que recebe uma função anônima como parâmetro, e tem sua duração definida para cada 200 milissegundos. A função anônima verifica se o valor de **windowIsScrolling** é verdadeiro, e caso seja, define seu valor como falso e verifica todas as instâncias de elementos que possuem 'hx-trigger=revealed' ou 'data-hx-trigger=revealed' como propriedades de **tags HTML**, e executa **maybeReveal()** com o valor de **elt**, que nesse caso é o valor capturado pela iteração.

```
function maybeReveal(elt) {
  if (!hasAttribute(elt, 'data-hx-revealed') && isScrolledIntoView(elt)) {
    elt.setAttribute('data-hx-revealed', 'true')
    const nodeData = getInternalData(elt)
    if (nodeData.initHash) {
      triggerEvent(elt, 'revealed')
    } else {
      elt.addEventListener('htmx:afterProcessNode',
        function() { triggerEvent(elt, 'revealed') }, { once: true })
    }
  }
}
```

A função **maybeReveal()** toma como parâmetro um valor **elt**, e no início de sua função verifica se o valor de **elt** não possui um atributo 'data-hx-revealed' e se o valor da chamada para **isScrolledIntoView()** com o valor de **elt** como parâmetro é falso. Caso essa expressão seja verdade, o valor de 'data-hx-revealed' no elemento é definido como verdadeiro, e é criada uma constante **nodeData** com o valor da chamada para **getInternalData()**. É então verificado se o valor da propriedade **initHash** de **nodeData** é definido, e caso seja, é feita uma chamada para **triggerEvent()**, passando o valor 'revealed' como parâmetro. Caso contrário, é adicionado uma escuta de eventos que é executada após 'htmx:afterProcessNode', que executa uma função anônima, que chama **triggerEvent()** passando o valor de 'revealed'. Após isso é definido que esta escuta de evento executa a sua função apenas uma vez, e é descartada.

```
function isScrolledIntoView(el) {
  const rect = el.getBoundingClientRect()
```

```

const elemTop = rect.top
const elemBottom = rect.bottom
return elemTop < window.innerHeight && elemBottom >= 0
}

```

A função **isScrolledIntoView()** toma como parâmetro um valor **el** e rapidamente verifica se a posição do elemento passado como parâmetro está dentro do foco na janela.

```
function addEventListener(elt, handler, nodeData, triggerSpec, explicitCancel)
```

A função toma como parâmetros os valores de **elt**, **handler**, **nodeData**, **triggerSpec** e **explicitCancel**.

```

const elementData = getInternalData(elt)
let eltsToListenOn
if (triggerSpec.from) {
  eltsToListenOn = querySelectorAllExt(elt, triggerSpec.from)
} else {
  eltsToListenOn = [elt]
}

```

No início da função, é definido uma constante **elementData** com o valor de retorno de **getInternalData()** e uma variável **eltsToListenOn** sem definição de valor. Após isso, é criada uma estrutura condicional que verifica se o valor da propriedade **from** de **triggerSpec** é definido, guardando o valor de retorno de **querySelectorAllExt()** em **eltsToListenOn** caso seja, ou guardando o valor de **elt** dentro de um Array e guardando esse valor em **eltsToListenOn**. O funcionamento de **querySelectorAllExt()** será analisado logo em seguida desta função, por agora, o interesse é o funcionamento de **addEventListener()**.

```

if (triggerSpec.changed) {
  if (!('lastValue' in elementData)) {
    elementData.lastValue = new WeakMap()
  }
  eltsToListenOn.forEach(function(eltToListenOn) {
    if (!elementData.lastValue.has(triggerSpec)) {
      elementData.lastValue.set(triggerSpec, new WeakMap())
    }
    elementData.lastValue
      .get(triggerSpec)
      .set(eltToListenOn, eltToListenOn.value)
  })
}

```

Após essa checagem, é verificado se o valor da propriedade **changed** de **triggerSpec** é definido. Após isso, é feita outra verificação para averiguar se o valor 'lastValue' está contido dentro de **elementData**. Caso essa avaliação seja verdadeira, o valor de **elementData.lastValue** recebe uma nova estrutura **WeakMap**. Um valor **WeakMap** define um dicionário cuja chave deve ser um objeto ou símbolos não registrados. Após isso, é feita uma iteração sobre os valores de **eltsToListenOn**, e é criada uma função anônima que toma como parâmetro o valor **eltToListenOn**. Dentro da iteração, é criada uma estrutura condicional que verifica se o valor de **triggerSpec** está presente em **lastValue**, propriedade de **elementData**. Caso o valor não esteja contido em **lastValue**, é definido o valor de **triggerSpec** com uma nova instância de um **WeakMap**. Após isso, o valor da propriedade **triggerSpec** dentro de **lastValue** é definido com o valor de **eltToListenOn** e seu valor é definido como sua propriedade **value**.

```
forEach(eltsToListenOn, function(eltToListenOn)
```

Fora da iteração sobre os valores de **eltsToListenOn** e fora da estrutura condicional que verifica **triggerSpec.changed**, é criada uma outra iteração sobre os valores de **eltsToListenOn**, capturando os valores individualmente no parâmetro **eltToListenOn** da função anônima a ser executada.

```
if (!bodyContains(elt)) {
    eltToListenOn.removeEventListener(triggerSpec.trigger, eventListener)
    return
}
if (ignoreBoostedAnchorCtrlClick(elt, evt)) {
    return
}
if (explicitCancel || shouldCancel(evt, elt)) {
    evt.preventDefault()
}
if (maybeFilterEvent(triggerSpec, elt, evt)) {
    return
}
```

Dentro da função anônima, diversas checagens e definições são feitas. No início das checagens, é definido uma constante **eventListener** que define uma função anônima, com um parâmetro **evt**, e é verificado se o valor de **elt** não está contido no corpo do documento através da função **bodyContains()**. Caso a expressão seja verdadeira, o valor da escuta de evento definido em **trigger** dentro de **triggerSpec** é removido e a função é encerrada. Após isso é verificado se o valor de **ignoreBoostedAnchorCtrlClick()** é verdadeiro, terminando a execução da função caso seja.

```
function ignoreBoostedAnchorCtrlClick(elt, evt) {
    return getInternalData(elt).boosted &&
    elt instanceof HTMLAnchorElement &&
    evt.type === 'click' &&
    (evt.ctrlKey || evt.metaKey)
}
```

A função **ignoreBoostedAnchorCtrlClick()** é definida com os parâmetros **elt** e **evt**, e verifica se o valor de **boosted** nos dados internos de **elt** é verdadeiro, se **elt** é uma instância de um elemento âncora do documento *HTML*, se o tipo de **evt** é 'click' e se o evento faz uso da tecla 'ctrl' ou tecla 'meta', retornando o valor dessa expressão como um valor booleano.

Em seguida, é verificado se o valor de **explicitCancel** é verdadeiro ou se o valor da chamada para **shouldCancel()** é verdadeiro, chamando o método **preventDefault()** para não permitir a execução do evento.

```
function shouldCancel(evt, elt) {
    if (evt.type === 'submit' || evt.type === 'click') {
        elt = asElement(evt.target) || elt
        if (elt.tagName === 'FORM') {
            return true
        }
        if (elt.form && elt.type === 'submit') {
            return true
        }
        if (elt instanceof HTMLAnchorElement && elt.href &&
            (elt.getAttribute('href') === '#' ||
            elt.getAttribute('href').indexOf('#') !== 0)) {
            return true
        }
    }
    return false
}
```

A função **shouldCancel()** toma como parâmetros os valores de **evt** e **elt**, e verifica se o valor do tipo de evento é 'submit' ou 'click'. Caso o tipo do elemento seja um formulário, o valor retornado é verdadeiro, se o tipo de formulário é definido e o tipo de evento for 'submit', é retornado um valor verdadeiro, e se o tipo do elemento é um elemento de âncora que possui uma propriedade **href** definida com o valor "#", o valor retornado é verdadeiro. Caso nenhuma das expressões sejam avaliadas, o valor retornado é falso.

```

const eventData = getInternalData(evt)
eventData.triggerSpec = triggerSpec
if (eventData.handledFor == null) {
    eventData.handledFor = []
}

```

Depois, ao voltar para **addEventListener()**, é definida uma constante **eventData** com o valor da chamada para **getInternalData()**, e o valor da propriedade **triggerSpec** dentro de **eventData** é definido com o valor de **triggerSpec**. Após isso, é verificado se o valor da propriedade **handledFor** de **eventData** é nulo, definindo seu valor com um **Array** vazio caso verdadeiro.

```

if (eventData.handledFor.indexOf(elt) < 0) {
    eventData.handledFor.push(elt)
    if (triggerSpec.consume) {
        evt.stopPropagation()
    }
}

```

Em seguida, é checado se o valor da posição de **elt** em **eventData.handledFor** é menor que zero, adicionando seu valor para o final de **handledFor** caso verdadeiro. Em seguida, é verificado se o valor da propriedade **consume** de **triggerSpec** é verdadeiro, chamando o método **stopPropagation()** de **evt**.

```

if (triggerSpec.target && evt.target) {
    if (!matches(asElement(evt.target), triggerSpec.target)) {
        return
    }
}
if (triggerSpec.once) {
    if (elementData.triggeredOnce) {
        return
    } else {
        elementData.triggeredOnce = true
    }
}
}

```

Após isso, é verificado se os valores da propriedade **target** em **triggerSpec** e em **evt** são definidos, e é verificado se os elementos são iguais. Caso sejam, a função termina a sua execução. Logo após, é verificado se o valor da propriedade **once** em **triggerSpec** é verdadeiro, verificando se o valor de **elementData.triggeredOnce** é verdadeiro, e retornando da função caso seja, ou definindo o seu valor como verdadeiro caso não.

```

(triggerSpec.changed) {
  const node = evt.target
  const value = node.value
  const lastValue = elementData.lastValue.get(triggerSpec)
  if (lastValue.has(node) && lastValue.get(node) === value) {
    return
  }
  lastValue.set(node, value)
}

```

Em seguida, é verificado se o valor da propriedade **changed** de **triggerSpec** é definido. Caso seja, são criadas as constantes **node** com o valor de **evt.target**, **value** com o valor de **node.value** e **lastValue** com o valor de **elementData.lastValue.get()**. Depois é criada uma estrutura condicional que verifica se o valor de **node** está contido em **lastValue** e se o valor de **node** em **lastValue** é igual a **value**. Caso essa expressão seja verdadeira, a função é encerrada, mas caso não, o valor de **node** em **lastValue** é definido com o valor de **value**.

```

if (elementData.delayed) {
  clearTimeout(elementData.delayed)
}
if (elementData.throttle) {
  return
}

```

Então é verificado se o valor de **delayed**, propriedade de **elementData** é definido, chamando a função **clearTimeout()**, caso seja, e é verificado se o valor da propriedade **throttle** de **elementData** é definido, encerrando a função caso seja.

```

if (triggerSpec.throttle > 0) {
  if (!elementData.throttle) {
    triggerEvent(elt, 'htmx:trigger')
    handler(elt, evt)
    elementData.throttle = getWindow().setTimeout(function() {
      elementData.throttle = null
    }, triggerSpec.throttle)
  }
} else if (triggerSpec.delay > 0) {
  elementData.delayed = getWindow().setTimeout(function() {
    triggerEvent(elt, 'htmx:trigger')
    handler(elt, evt)
  }, triggerSpec.delay)
}

```

```

    }, triggerSpec.delay)
  } else {
    triggerEvent(elt, 'htmx:trigger')
    handler(elt, evt)
  }
}

```

Por fim, é verificado se o valor da propriedade **throttle** de **triggerSpec** é maior que zero. Caso seja, é verificado se o valor de **throttle** em **elementData** é indefinido, chamando **triggerEvent()**, **handler()** e definindo o valor de **throttle** com o valor de **setTimeout()** com uma função que define o seu próprio valor para nulo após o tempo definido em **throttle**. Em seguida, é verificado se o valor de **delay** em **triggerSpec** é maior que 0, guardando o valor de uma chamada para **setTimeout()** que chama **triggerEvent()** e **handler()**, com a duração definida pelo próprio valor de **delay**. Por fim, caso nenhuma das expressões sejam verdadeiras, é feita uma execução para **triggerEvent()** e **handler()** diretamente.

```

if (nodeData.listenerInfos == null) {
  nodeData.listenerInfos = []
}
nodeData.listenerInfos.push({
  trigger: triggerSpec.trigger,
  listener: eventListener,
  on: eltToListenOn
})
eltToListenOn.addEventListener(triggerSpec.trigger, eventListener)

```

Ao final de **addEventListener()**, é verificado se o valor de **listenerInfos**, propriedade de **nodeData** é nulo, definindo o seu valor para um **Array** vazio caso seja, e ao final, é adicionado um valor de objeto com as propriedades de **triggerSpec.trigger**, **eventListener** e **eltToListenOn**. Ao final da função, é adicionado uma escuta de evento para o evento de **trigger** em **triggerSpec** e é chamada a função **eventListener** quando o evento é acionado. Para concluir o funcionamento de **addEventListener()**, basta ver agora a função **querySelectorAllExt()**.

```
function querySelectorAllExt(elt, selector, global)
```

A função toma como parâmetros os valores de **elt**, **selector** e **global**.

```

if (selector.indexOf('global ') === 0) {
  return querySelectorAllExt(elt, selector.slice(7), true)
}

```

No seu início, é imediatamente verificado se a posição da **string** 'global' dentro de **selector** é no seu início. Caso seja, é feita uma chamada recursiva para **querySelectorAllExt()** que passa **selector** com o seu início removido através do método **slice()**, removendo os sete primeiros caracteres.

```
elt = resolveTarget(elt)
```

```
const parts = []
{
  let chevronsCount = 0
  let offset = 0
  for (let i = 0; i < selector.length; i++) {
    const char = selector[i]
    if (char === ',' && chevronsCount === 0) {
      parts.push(selector.substring(offset, i))
      offset = i + 1
      continue
    }
    if (char === '<') {
      chevronsCount++
    } else if (char === '/' &&
      i < selector.length - 1 &&
      selector[i + 1] === '>') {
      chevronsCount--
    }
  }
  if (offset < selector.length) {
    parts.push(selector.substring(offset))
  }
}
```

Após isso, o valor de **elt** é definido pelo retorno da função **resolveTarget()**, e é criada uma constante **parts** que recebe um **Array** vazio. Após isso, é criado um novo escopo dentro da função que cria duas variáveis: **chevronsCount** e **offset**, ambas com o valor inicial de zero, e cria uma estrutura de repetição que tem a duração definida pelo tamanho de **selector**. Dentro da estrutura de repetição, é criada uma constante **char** com o valor do caractere atual sendo iterado pela estrutura, e é verificado se o valor deste caractere é igual à ',' ou se o valor de **chevronsCount** é igual a zero. Caso qualquer uma das duas afirmações seja verdadeira, o valor do **caractere** é adicionado a **parts**, e o valor de **offset** recebe o valor de **i** somado à 1, e a estrutura de repetição passa para o próximo valor da

iteração. Após isso, é verificado se o valor de **char** é igual a '<', incrementando o valor de **chevronsCount** caso seja verdade, ou verificando se o valor de **char** é igual a '/' e se **i** é menor que o valor do tamanho de **selector** menos 1 e se o valor de **selector** somado a 1 é igual a '>', executando um decremento no valor de **chevronsCount** caso a expressão seja avaliada como verdadeira. Ao final da iteração, é verificado se o valor de **offset** é menor que o tamanho de **selector**, e caso isso seja verdade, o valor no espaço de índice **offset** em **selector** é adicionado ao final de **parts**.

```
const result = []
const unprocessedParts = []
```

Em seguida, são criadas duas constantes, **result** e **unprocessedParts** com os valores de estruturas de **Array** vazias.

```
while (parts.length > 0) {
  const selector = normalizeSelector(parts.shift())
  let item
```

É então criada uma estrutura de repetição que dura enquanto o tamanho de **parts** for maior que zero. No seu início, é criada uma constante **selector** que recebe o primeiro valor removido de **parts** e é processado pela função **normalizeSelector()**, e uma variável **item** é criada.

```
function normalizeSelector(selector) {
  const trimmedSelector = selector.trim()
  if (startsWith(trimmedSelector, '<') &&
    endsWith(trimmedSelector, '/>')) {
    return trimmedSelector.substring(1, trimmedSelector.length - 2)
  } else {
    return trimmedSelector
  }
}
```

A função **normalizeSelector()** tem apenas um valor **selector** como parâmetro. No seu início, é criada uma constante **trimmedSelector()** que recebe o seu próprio valor com espaços em branco de caracteres de barra removidos do começo e do final da **string**. Após isso, é verificado se o valor de **trimmedSelector** começa com '<' e termina com '/>'. Caso a expressão seja verdadeira, o valor de **trimmedSelector** é enviado sem esses caracteres como valor de retorno pela função, caso contrário, o valor é retornado diretamente sem nenhum tratamento.

```

if (selector.indexOf('closest ') === 0) {
    item = closest(asElement(elt), normalizeSelector(selector.slice(8)))
}

```

É então criada uma série de estruturas condicionais para a checagem do estado de **selector**. Primeiramente, é verificado se **selector** possui um valor 'closest', definindo o valor de item com o valor da chamada para **closest()** caso não tenha.

```

function closest(elt, selector) {
    elt = asElement(resolveTarget(elt))
    if (elt) {
        return elt.closest(selector)
    }
    return null
}

```

A função **closest()** recebe como parâmetros os valores **elt** e **selector()**. No seu corpo, define o valor de **elt** com uma chamada para **resolveTarget()** que é transformado em um **Element** com outra chamada para **asElement()**. Após isso, é verificado se o valor de **elt** é definido, e caso seja o valor do **selector** mais próximo de **elt** é retornado. Caso não seja, um valor nulo é retornado pela função.

```

else if (selector.indexOf('find ') === 0) {
    item = find(asParentNode(elt), normalizeSelector(selector.slice(5)))
}

```

Após isso, é verificado se **selector** possui algum valor 'find', e caso não tenha, o valor de item é definido com uma chamada para **find()**, que tenta encontrar o valor do **selector** dentro do elemento pai **elt**.

```

else if (selector === 'next' || selector === 'nextElementSibling') {
    item = asElement(elt).nextElementSibling
}

```

Em seguida, é verificado se **selector** é igual à **next** ou **nextElementSibling**, e caso seja, o valor de **item** é definido com o valor do próximo **Element** irmão à **elt**.

```

else if (selector.indexOf('next ') === 0) {
    item = scanForwardQuery(elt,
        normalizeSelector(selector.slice(5)), !!global)
}

```

É então verificado se o valor de **selector** possui um valor 'next', e caso tenha, é definido o valor de **item** com o valor de retorno de **scanForwardQuery()**.

```
var scanForwardQuery = function(start, match, global) {
  const results = asParentNode(getRootNode(start, global))
    .querySelectorAll(match)
  for (let i = 0; i < results.length; i++) {
    const elt = results[i]
    if (elt.compareDocumentPosition(start) ===
      Node.DOCUMENT_POSITION_PRECEDING) {
      return elt
    }
  }
}
```

A função contida em **scanForwardQuery()** é definida com três parâmetros: **start**, **match** e **global**. No seu corpo, é definida uma constante **results** que recebe o valor da chamada de **querySelectorAll()**, recebendo os valores que se enquadram dentro de **match**. Após isso, é criada uma estrutura de repetição que tem a duração definida pelo tamanho de **results**. No seu corpo é capturado o valor iterado atualmente com a constante **elt** e é verificado se o valor de **compareDocumentPosition()** é igual à **Node.DOCUMENT_POSITION_PRECEDING**. Caso isso seja avaliado como verdade, o valor de **elt** é retornado.

```
else if (selector === 'previous' ||
  selector === 'previousElementSibling') {
  item = asElement(elt).previousElementSibling
}
```

Voltando à **querySelectorAllExt()**, é então verificado se o valor de **selector** é igual à 'previous' ou à 'previousElementSibling'. Caso seja, o valor de **item** é definido com o valor do **Element** anterior à **elt**.

```
else if (selector.indexOf('previous ') === 0) {
  item = scanBackwardsQuery(elt,
    normalizeSelector(selector.slice(9)), !!global)
}
```

Em seguida, é checado se o valor de **selector** possui um valor 'previous', e caso tenha, é definido o valor de **item** com o valor de retorno para **scanBackwardsQuery()**.

```

var scanBackwardsQuery = function(start, match, global) {
  const results = asParentNode(getRootNode(start, global))
    .querySelectorAll(match)
  for (let i = results.length - 1; i >= 0; i--) {
    const elt = results[i]
    if (elt.compareDocumentPosition(start) ===
        Node.DOCUMENT_POSITION_FOLLOWING) {
      return elt
    }
  }
}

```

A função **scanBackwardsQuery()** tem uma estrutura praticamente idêntica a de **scanForwardQuery()**, mas ele procura os valores em **results** decremento o valor do iterador, ou seja, pelo final primeiramente, e compara o valor da posição do documento com **Node.DOCUMENT_POSITION_FOLLOWING**.

```

else if (selector === 'document') {
  item = document
} else if (selector === 'window') {
  item = window
} else if (selector === 'body') {
  item = document.body
} else if (selector === 'root') {
  item = getRootNode(elt, !!global)
}

```

É então verificado se **selector** é igual à 'document', 'window', 'body' ou 'root', definindo o valor de item de acordo com o resultado, respectivamente: **document**, **window**, **document.body** ou **getRootNode()**.

```

function getRootNode(elt, global) {
  return elt.getRootNode ?
    elt.getRootNode({ composed: global }) : getDocument()
}

```

A função **getRootNode** procura o nóculo raiz do elemento que chama a função, definido como a propriedade **getRootNode**. Caso a propriedade seja definida, o valor da chamada para **getRootNode()** é feito, caso contrário é devolvida uma chamada para **getDocument()**.

```

else if (selector === 'host') {
    item = ((elt.getRootNode())).host
} else {
    unprocessedParts.push(selector)
}

```

Ao final das checagens é verificado se **selector** é de valor 'host', e caso seja, define o valor de item com a propriedade **host** de **elt.getRootNode()**. Ao final, caso nenhuma das condições sejam avaliadas como verdadeiras, o valor de **selector** é adicionado ao final de **unprocessedParts()**.

```

if (item) {
    result.push(item)
}

```

É então declarado que se **item** for definido, o valor de **item** é adicionado ao final de **result**.

```

if (unprocessedParts.length > 0) {
    const standardSelector = unprocessedParts.join(',')
    const rootNode = asParentNode(getRootNode(elt, !!global))
    result.push(...toArray(rootNode.querySelectorAll(standardSelector)))
}

```

```

return result

```

Por fim, é verificado se o tamanho de **unprocessedParts** é maior que 0, criando as constantes **standardSelector** que recebe os valores de **unprocessedParts** transformados em uma **string** separados por um caractere ',' e **rootNode** que recebe o valor da chamada para **asParentNode()** com **getRootNode()** de **elt** como parâmetro. Ao final o valor da chamada para o método **querySelectorAll()** é adicionado ao final de **result** como um **Array**, e o valor de **result** é retornado pela função.

Assim, é finalizada a leitura de **querySelectorAllExt()**, finalizando o entendimento de **addEventListener**, que por sua vez, conclui o entendimento de **addTriggerHandler()** e por consequência **processVerbs()**.

```

if (!hasExplicitHttpAction) {
    if (getClosestAttributeValue(elt, 'hx-boost') === 'true') {
        boostElement(elt, nodeData, triggerSpecs)
    } else if (hasAttribute(elt, 'hx-trigger')) {
        triggerSpecs.forEach(function(triggerSpec) {

```

```

        addTriggerHandler(elt, triggerSpec, nodeData, function() {
        })
    })
}
}

```

Ao voltar para **initNode()**, é verificado se o valor de **hasExplicitActions** é falso, e neste caso, será checado se o valor de 'hx-boost' possui um valor verdadeiro definido, através de **getClosestAttributeValue()**. Caso a expressão seja verdadeira, é feita uma chamada para **boostElement()**. Caso seja falsa, é então checado se o elemento possui uma propriedade 'hx-trigger', então executando uma iteração sobre o valor de **triggerSpecs** e chamando a função **addTriggerHandler()** para cada um deles, passando uma função vazia como um dos parâmetros.

```

function getClosestAttributeValue(elt, attributeName) {
    let closestAttr = null
    getClosestMatch(elt, function(e) {
        return !!closestAttr =
            getAttributeValueWithDisinheritance(elt,
            asElement(e), attributeName)
    })
    if (closestAttr !== 'unset') {
        return closestAttr
    }
}

```

A função **getClosestAttributeValue()** toma como parâmetros os valores de **elt** e **attributeName**, e no seu corpo, cria uma variável **closestAttr** com um valor inicial nulo. É então feita uma chamada para **getClosestMatch()** com os parâmetros **elt** e uma função anônima que recebe e como parâmetro e retorna um valor que define **closestAttr** com o valor de **getAttributeValueWithDisinheritance()**. Ao final da função, se o valor de **closestAttr** for diferente de 'unset', o valor é retornado pela função.

```

function getClosestMatch(elt, condition) {
    while (elt && !condition(elt)) {
        elt = parentElt(elt)
    }

    return elt || null
}

```

A função **getClosestMatch()** toma como parâmetros os valores **elt** e **condition**, e cria uma estrutura de repetição que dura enquanto o valor de **elt** for definido e enquanto a condição passada como parâmetro for falsa, definindo o valor de **elt** como o elemento parente de **elt**. Ao final da função o valor de **elt** é retornado caso este não seja nulo.

```
function getAttributeValueWithDisinheritance(initialElement,
ancestor, attributeName) {
  const attributeValue = getAttributeValue(ancestor, attributeName)
  const disinherit = getAttributeValue(ancestor, 'hx-disinherit')
  var inherit = getAttributeValue(ancestor, 'hx-inherit')
  if (initialElement !== ancestor) {
    if (htmx.config.disableInheritance) {
      if (inherit && (inherit === '*' ||
        inherit.split(' ').indexOf(attributeName) >= 0)) {
        return attributeValue
      } else {
        return null
      }
    }
    if (disinherit && (disinherit === '*'
      || disinherit.split(' ').indexOf(attributeName) >= 0)) {
      return 'unset'
    }
  }
  return attributeValue
}
```

A função **getAttributeValueWithDisinheritance()** tem três parâmetros: **initialElement**, **ancestor** e **attributeName**. No seu corpo, define duas constantes: **attributeValue** com o valor de **getAttributeValue()** utilizando **attributeName** como parâmetro e **disinherit** com o valor de **getAttributeValue()** utilizando 'hx-disinherit' como parâmetro, e é criada uma variável **inherit** com o valor da chamada para **getAttributeValue()** utilizando 'hx-inherit' como parâmetro. Após isso, é verificado se o valor de **initialElement** diferente de **ancestor**, e após isso, se o valor de configuração **disableInheritance** de **htmx** é verdadeiro. Caso seja, é verificado se o valor de **inherit** é definido e se o valor de **inherit** é '*' ou o valor de **attributeName** não está contido em **inherit**. Caso essa expressão seja verdadeira, o valor de **attributeValue** é retornado, caso contrário um valor nulo é retornado. Após isso, ainda na estrutura condicional mais externa, é checado se o valor de **disinherit** é definido e se o valor de **disinherit** é igual a '*' ou se o valor de **attributeName** está contido em **disinherit**, retornando 'unset'. Ao final da função, o valor de **attributeValue** é retornado pela função.

```
function boostElement(elt, nodeData, triggerSpecs)
```

A função **boostElement()** toma como parâmetros os valores **elt**, **nodeData** e **triggerSpecs()**.

```
if ((elt instanceof HTMLAnchorElement
    && isLocalLink(elt)
    && (elt.target === "" ||
    elt.target === '_self')) ||
    (elt.tagName === 'FORM' &&
    String(getRawAttribute(elt, 'method')).toLowerCase() !== 'dialog'))
```

É então verificado se o valor de **elt** é uma instância de **HTMLAnchorElement** e se a chamada para **isLocalLink()** é verdadeira, e se **elt.target** tem como valor "" ou '_self' ou se **elt.tagName** é igual a 'FORM' e se o atributo 'method' de **elt** é diferente de 'dialog'.

```
function isLocalLink(elt) {
    return location.hostname === elt.hostname &&
        getRawAttribute(elt, 'href') &&
        getRawAttribute(elt, 'href').indexOf('#') !== 0
}
```

A definição de **isLocalLink()** define como parâmetro um valor **elt**, e retorna uma expressão booleana a ser avaliada: se **location.hostname** é igual à **elt.hostname** e se **getRawAttribute()** para **element** procurando por um valor 'href' existe e se possui um valor '#'.

```
nodeData.boosted = true
let verb, path
if (elt.tagName === 'A') {
    verb = (('get'))
    path = getRawAttribute(elt, 'href')
} else {
    const rawAttribute = getRawAttribute(elt, 'method')
    verb = ((rawAttribute ? rawAttribute.toLowerCase() : 'get'))
    path = getRawAttribute(elt, 'action')
    if (path === null || path === "") {
        path = location.href
    }
    if (verb === 'get' && path.includes('?')) {
        path = path.replace(/\?[^\#]+/, "")
    }
}
```

```

    }
}

```

É então definido o valor de **nodeData.boosted** como verdadeiro, e são criadas duas variáveis: **verb** e **path**, sem valores de inicialização. É então verificado se a **tag** de **elt** é um 'a', e caso seja, o valor de **verb** é definido como 'get' e o valor de **path** é definido com o valor de 'href'. Caso o valor da **tag** não seja um 'a', é criada uma constante **rawAttribute** que recebe o valor da propriedade **method** de **elt** e o valor de **verb** é definido de acordo com a checagem de **rawAttribute**: se for definido, o seu próprio valor é definido, caso contrário o valor 'get' é retornado. O valor de **path** é definido com a chamada para **getRawAttribute()** coletando o valor da propriedade 'action' de **elt**, e é verificado se **path** é nulo ou uma **string** vazia, definindo seu valor como **location.href** caso seja verdade. Após isso, é verificado se o valor de **verb** é igual a 'get' e se **path** inclui um caractere '?', definindo o valor de **path** com o valor de uma expressão regular que remove qualquer valor com *queries* no final.

```

function getRawAttribute(elt, name) {
    return elt instanceof Element && elt.getAttribute(name)
}

```

A função **getRawAttribute()** tem como parâmetros os valores de **elt** e **name** e em seu corpo, retorna o valor do atributo definido por **name** dentro de **elt** com o método **getAttribute()** da classe **Element** caso **elt** seja uma instância de **Element**.

```

triggerSpecs.forEach(function(triggerSpec) {
    addEventListener(elt, function(node, evt) {
        const elt = asElement(node)
        if (eltIsDisabled(elt)) {
            cleanUpElement(elt)
            return
        }
        issueAjaxRequest(verb, path, elt, evt)
    }, nodeData, triggerSpec, true)
})

```

Ao final de **boostElement()** é feita uma iteração sobre os valores de **triggerSpecs** e para cada, é feita uma execução de **addEventListener()** que define uma função que cria uma constante **elt** que transforma o valor de **node** em um **Element**, e é verificado se o valor de **elt** é desativado, fazendo uma chamada para **cleanUpElement()** e retornando da função, ou executando **issueAjaxRequest()** caso contrário. Isso conclui o funcionamento de **boostElement()**.

```

if (elt.tagName === 'FORM' ||
    (getRawAttribute(elt, 'type') === 'submit' &&
     hasAttribute(elt, 'form'))) {
    initButtonTracking(elt)
}

```

```

nodeData.firstInitCompleted = true
triggerEvent(elt, 'htmx:afterProcessNode')

```

Ao final de **initNode()**, é verificado se o valor da **tag** de **elt** é 'form' ou se o valor do atributo 'type' é igual a 'submit' e se o elemento possui um atributo 'form', executando uma chamada para **initButtonTracking()** caso seja verdadeiro. Ao final da função, é definido o valor da propriedade **firstInitCompleted** de **nodeData** como verdadeiro, e é feita uma chamada para **triggerEvent()** com um valor 'htmx:afterProcessNode'.

```

function initButtonTracking(elt) {
    elt.addEventListener('click', maybeSetLastButtonClicked)
    elt.addEventListener('focusin', maybeSetLastButtonClicked)
    elt.addEventListener('focusout', maybeUnsetLastButtonClicked)
}

```

A função **initButtonTracking()** define três escutas de evento para **elt** que espera pelas ações de clique, entrada de foco e saída de foco do mouse, executando **maybeSetLastButtonClicked()** para os dois primeiros e **maybeUnsetLastButtonClicked()** para o último.

```

function maybeSetLastButtonClicked(evt) {
    const elt = getTargetButton(evt.target)
    const internalData = getRelatedFormData(evt)
    if (internalData) {
        internalData.lastButtonClicked = elt
    }
}

```

A função **maybeSetLastButtonClicked()** toma como parâmetro um valor **evt**. No seu corpo, cria as constantes **elt** com o valor de **getTargetButton()** e **internalData** com o valor de **getRelatedFormData()**. Após isso, é verificado se o valor de **internalData** é definido, e caso seja, o valor de **lastButtonClicked** propriedade de **internalData** é definido com o valor de **elt**.

```
function getTargetButton(target) {
  return (closest(asElement(target), "button, input[type='submit']"))
}
```

A função **getTargetButton()** recebe um parâmetro **target** como parâmetro e retorna o valor do botão ou **input** de tipo 'submit' mais próximo.

```
function getRelatedFormData(evt) {
  const elt = getTargetButton(evt.target)
  if (!elt) {
    return
  }
  const form = getRelatedForm(elt)
  return getInternalData(form)
}
```

A função **getRelatedFormData()** toma como parâmetro o valor **evt** e no seu corpo cria uma constante **elt** que recebe o valor de **getTargetButton()**. É então verificado se o valor de **elt** é indefinido, encerrando então a função. Caso **elt** seja definido, é criada uma constante **form** com o valor da chamada para **getRelatedForm()** e é retornado o valor de **getInternalData()** de **form**.

```
function getRelatedForm(elt) {
  return elt.form || closest(elt, 'form')
}
```

A função **getRelatedForm()** tem como parâmetro um valor **elt**, e retorna o valor da propriedade **form** de **elt** ou o valor do elemento **form** mais próximo de **elt**.

```
function maybeUnsetLastButtonClicked(evt) {
  const internalData = getRelatedFormData(evt)
  if (internalData) {
    internalData.lastButtonClicked = null
  }
}
```

Por fim, a função **maybeUnsetLastButtonClicked()** recebe como parâmetro um valor **evt** e cria uma constante **internalData** com o valor de **getRelatedFormData()**. É então verificado se **internalData** é definido, definindo o valor de sua propriedade **lastButtonClicked** como nulo, caso seja. Isso, por fim, completa o entendimento de **initNode()**, que encerra a análise de **processNode()**, nos permitindo voltar para a função de inicialização do *HTMX*.

```

const restoredElts = getDocument().querySelectorAll(
  "[hx-trigger='restored'],[data-hx-trigger='restored']"
)
body.addEventListener('htmx:abort', function(evt) {
  const target = evt.target
  const internalData = getInternalData(target)
  if (internalData && internalData.xhr) {
    internalData.xhr.abort()
  }
})

```

Ao voltar para a chamada de **ready()** na inicialização de *HTMX*, é criada uma constante **restoredElts** que recebe o valor de todos os elementos que possuem as propriedades 'hx-trigger=restored' ou 'data-hx-trigger=restored'. Após isso, é criada uma escuta de eventos para o evento 'htmx:abort' que executa uma função anônima que captura o evento dentro da variável **evt**. Dentro desta função são criadas as constantes **target** e **internalData** com os valores de **evt.target** e **getInternalData()** utilizando **target** como parâmetro, respectivamente. Após isso, é verificado se os valores de **internalData** e de sua propriedade **xhr** são definidos, chamando o método **abort()** de **xhr** caso a avaliação seja verdadeira.

abort(): void;

Segundo a definição de **abort()**, a função cancela qualquer atividade na rede existente.

```

const originalPopstate = window.onpopstate ?
  window.onpopstate.bind(window) : null

```

Após isso, é criada uma constante **originalPopState** que recebe um de dois valores, a partir da checagem do valor de **window.onpopstate**: caso o valor seja definido, é enviado o valor da chamada para **bind()**, método de **window.onpopstate**, utilizando **window** como parâmetro, caso o valor seja indefinido é retornado um valor nulo. A definição de **onpopstate** define que esse valor carrega a história de *URLs* da navegação do usuário.

```

window.onpopstate = function(event) {
  if (event.state && event.state.htmx) {
    restoreHistory()
    forEach(restoredElts, function(elt) {
      triggerEvent(elt, 'htmx:restored', {
        document: getDocument(),

```

```

        triggerEvent
    })
})
} else {
    if (originalPopstate) {
        originalPopstate(event)
    }
}
}
}

```

É então definido o valor de **window.onpopstate** com uma função anônima que recebe um valor **event** como parâmetro. No seu início, é verificado se o valor de **state** e de sua propriedade **htmx**, que são propriedades de **event**, são definidos. Caso a expressão seja verdadeira, é feita uma chamada para **restoreHistory()** e é feita uma iteração sobre os valores de **restoredElts**, executando outra função anônima capturando os valores de **restoreElts** em uma variável **elt**. Dentro desta função é feita uma chamada para **triggerEvent()**, que aciona um evento 'htmx:restored', e passa um objeto com os valores de **getDocument()**, e **triggerEvent**. Caso a expressão seja falsa, é então verificado se o valor de **originalPopState** é definido, e caso seja, é feita uma chamada para **originalPopstate()**, utilizando **event** como parâmetro.

```
function restoreHistory(path)
```

A definição de **restoreHistory()** toma como parâmetro um valor **path**.

```

saveCurrentPageToHistory()
path = path || location.pathname + location.search
const cached = getCacheHistory(path)

```

No seu início é feita uma chamada para **saveCurrentPageToHistory()**, o valor de **path** é definido com seu próprio valor caso este não seja indefinido, mas caso seja, recebe o valor da concatenação de **pathname** e **search**, ambas propriedades de **location**, e por fim é criada uma constante **cached** com o valor de **getCacheHistory()**.

```
function saveCurrentPageToHistory()
```

A função **saveCurrentPageToHistory()** não toma nenhum parâmetro.

```

const elt = getHistoryElement()
let path = currentPathForHistory
if (canAccessLocalStorage()) {
    path = sessionStorage.getItem('htmx-current-path-for-history')
}

```

```

}
path = path || location.pathname + location.search

```

No seu início é definida uma constante **elt** com o valor de **getHistoryElement()** e uma variável **path** com o valor de **currentPathForHistory**. Após isso, é verificado se o valor da chamada para **canAccessLocalStorage()** é verdadeiro, definindo o valor de **path** com o valor do item 'htmx-current-path-for-history', utilizando o método **getItem()** de **sessionStorage**. Após isso, é verificado se o valor de **path** é definido, e caso ele não seja, o valor de **path** recebe os valores de **pathname** e **search**, ambas propriedades de **location**.

```

function getHistoryElement() {
    const historyElt = getDocument()
        .querySelector("[hx-history-elt],[data-hx-history-elt]")
    return historyElt || getDocument().body
}

```

A definição de **getHistoryElement()** não toma parâmetros, e cria uma constante **historyElt** com o valor dos elementos que possuem as propriedades 'hx-history-elt' ou 'data-hx-history-elt'. Ao final da função, se o valor de **historyElt** for definido, ele é retornado ao fim da função, caso contrário o valor de **body** do documento é retornado.

```

function canAccessLocalStorage() {
    const test = 'htmx:sessionStorageTest'
    try {
        sessionStorage.setItem(test, test)
        sessionStorage.removeItem(test)
        return true
    } catch (e) {
        return false
    }
}

```

A definição da função **canAccessLocalStorage()** não toma parâmetros, e cria uma constante **test** com uma **string** 'htmx:sessionStorageTest', e tenta adicionar e remover esse valor do armazenamento da sessão do usuário, através dos métodos **setItem()** e **removeItem()** de **sessionStorage**. Caso a execução destes não levante exceções, um valor verdadeiro é retornado, mas caso alguma exceção seja capturada, um valor falso é retornado.

```

const disableHistoryCache = getDocument()
    .querySelector("[hx-history="false" i]

```

```

    ,[data-hx-history="false" i])

if (!disableHistoryCache) {
    triggerEvent(getDocument().body,
        'htmx:beforeHistorySave',
        { path, historyElt: elt })
    saveToHistoryCache(path, elt)
}

if (htmx.config.historyEnabled) history.replaceState(
    { htmx: true },
    getDocument().title, location.href
)

```

É então criada uma constante **disableHistoryCache** que recebe todos os valores de elementos que possuem uma propriedade 'hx-history=false' ou um valor 'data-hx-history=false' definidos. Após isso, é verificado se o valor de **disableHistoryCache** é indefinido, e caso seja, é feita uma chamada para **triggerEvent()**, acionando o evento 'htmx:beforeHistorySave' e chamando **saveToHistoryCache()**. Ao final da função é verificado se o valor de configuração **historyEnabled** de **htmx** é definido, chamando o método **replaceState** de **history** para devolver o estado anterior da janela atual no navegador.

```

function getCachedHistory(url) {
    if (!canAccessLocalStorage()) {
        return null
    }

    url = normalizePath(url)

    const historyCache = parseJSON(sessionStorage.
        getItem('htmx-history-cache')) || []
    for (let i = 0; i < historyCache.length; i++) {
        if (historyCache[i].url === url) {
            return historyCache[i]
        }
    }
    return null
}

```

A função **getCachedHistory()** é definida com um parâmetro **url**, e em seu corpo no início é verificado se o valor da chamada para **canAccessLocalStorage()** é falso, e caso

seja, um valor nulo é retornado pela função. Caso contrário, o valor de **url** é definido com uma chamada para **normalizePath()**, utilizando **url** como parâmetro. Após isso, é definida o valor da constante **historyCache()** com o valor da chamada de **parseJSON()** utilizando os valores de 'htmx-history-cache' e retornando o valor em formato de **Array** ou uma **Array** vazia caso não haja resultados. Ao final da função é criado uma iteração que tem a duração definida pelo tamanho de **historyCache**, e durante sua execução, verifica se o valor da propriedade **url** de **historyCache** é estritamente igual ao valor do parâmetro **url** da função, e caso seja, este valor é retornado. Caso a iteração termine sem retornar qualquer valor, um tipo nulo é retornado pela função.

```
function normalizePath(path) {
  const url = new URL(path, 'http://x')
  if (url) {
    path = url.pathname + url.search
  }
  if (path !== '/') {
    path = path.replace(/\/+$/, "")
  }
  return path
}
```

A função **normalizePath()** toma um valor **path** como parâmetro, e em seu início, irá iniciar uma constante **url** com o valor de uma nova instância do objeto *URL* da biblioteca padrão do *JavaScript* com um valor inicial **placeholder** definido com a rota 'http://x' e o caminho é definido com o valor de **path**. Após isso, é verificado se o valor de **url** é existente, para o caso de **path** ser nulo, e é então definido o valor de **path** com o valor das propriedades **pathname** e **search** de **url** concatenadas nesta ordem. Após isso é verificado se o valor de **path** não é apenas um único caractere de barra, e caso seja, o valor de **path** terá essa única barra removida do **url**. Ao final da função, o valor de **path** é retornado.

```
if (cached) {
  const swapSpec = { swapStyle: 'innerHTML',
    swapDelay: 0, settleDelay: 0, scroll: cached.scroll }
  const details = { path, item: cached,
    historyElt: getHistoryElement(), swapSpec }
```

Ao voltar para **restoreHistory()** é verificado se o valor de **cached**, definido com a chamada para **getCachedHistory()** é um valor válido. Caso seja, o valor de **swapSpec** é definido com um objeto, definindo suas propriedades **swapStyle** com o valor de 'innerHTML', **swapDelay** com um valor 0, **settleDelay** com um valor 0 e **scroll** com o valor da propriedade **cached** de **scroll**. Após isso, é criada uma constante **details** com um objeto que possui as

propriedades definidas com o valor de **path**, uma propriedade **item** com o valor de **cached**, uma propriedade **historyElt** com o valor da chamada para **getHistoryElement()**, e uma propriedade **swapSpec** com o valor da constante definida anteriormente.

```
if (triggerEvent(getDocument().body, 'htmx:historyCacheHit', details)) {
    swap(details.historyElt, cached.content, swapSpec, {
        contextElement: details.historyElt,
        title: cached.title
    })
    setCurrentPathForHistory(details.path)
    triggerEvent(getDocument().body, 'htmx:historyRestore', details)
}
```

Após isso, ainda dentro da estrutura condicional anterior, é verificado se o valor de **triggerEvent()** retorna um valor válido, e caso seja, são feitas chamadas para as funções **swap()**, **setCurrentPathForHistory()** e **triggerEvent()**. A funcionalidade de **swap()** será vista a fundo em outro capítulo desta análise, pois se trata da funcionalidade básica do método **swap** de **htmx**. A função **triggerEvent()** já foi explicada, portanto em seguida será analisado o funcionamento de **setCurrentPathForHistory()**.

```
let currentPathForHistory = location.pathname + location.search
```

```
function setCurrentPathForHistory(path) {
    currentPathForHistory = path
    if (canAccessLocalStorage()) {
        sessionStorage.setItem('htmx-current-path-for-history', path)
    }
}
```

A função **setCurrentPathForHistory()** requer um parâmetro **path**. Dentro dela, o valor da variável global **currentPathForHistory**, que recebe o valor de **path**. Caso o navegador permita o uso do armazenamento de sessão, essa função também irá definir o valor de uma chave 'htmx-current-path-history' com o valor de **path** no armazenamento de sessão.

```
else {
    if (htmx.config.refreshOnHistoryMiss) {
        htmx.location.reload(true)
    } else {
        loadHistoryFromServer(path)
    }
}
```

Ao voltar para **restoreHistory()**, ao final da função, caso a estrutura condicional anterior não seja executada, o **htmx** irá verificar se o valor de da propriedade **refreshOnHistoryMiss** de **htmx.config** é verdadeiro, executando uma chamada para o método **reload()** de **htmx.location**. Caso contrário, é feita uma chamada para **loadHistoryFromServer()** e a função é encerrada.

```
function loadHistoryFromServer(path) {
  const request = new XMLHttpRequest()
  const swapSpec = { swapStyle: 'innerHTML',
    swapDelay: 0, settleDelay: 0 }
  const details = { path, xhr: request,
    historyElt: getHistoryElement(), swapSpec }
  request.open('GET', path, true)
  if (htmx.config.historyRestoreAsHxRequest) {
    request.setRequestHeader('HX-Request', 'true')
  }
}
```

A função **loadHistoryFromServer()** toma como parâmetro um valor **path**, e em seu início declara três constantes: **request**, com o valor de uma nova instância de **XMLHttpRequest**, **swapSpec** com um objeto que servirá para definir o escopo de uma chamada para **swap()** e **details**, que declara alguns detalhes de uma requisição **HTTP**. Após isso, é chamado o método **open()** de **request**, declarando uma requisição 'GET' para o caminho declarado em **path**. Em seguida, é verificado se o valor da propriedade **historyRestoreAsHxRequest** de **htmx.config** é definido como verdadeiro, chamando o método **setRequestHeader()** de **request** e definindo o valor de **HxRequest** como verdadeiro.

```
request.setRequestHeader('HX-History-Restore-Request', 'true')
request.setRequestHeader('HX-Current-URL', location.href)
request.onload = function() {
  if (this.status >= 200 && this.status < 400) {
    details.response = this.response
    triggerEvent(getDocument().body,
      'htmx:historyCacheMissLoad', details)
    swap(details.historyElt, details.response, swapSpec, {
      contextElement: details.historyElt,
      historyRequest: true
    })
    setCurrentPathForHistory(details.path)
    triggerEvent(getDocument().body,
      'htmx:historyRestore',
```

```

    { path, cacheMiss: true, serverResponse: details.response })
  } else {
    triggerErrorEvent(getDocument().body,
      'htmx:historyCacheMissLoadError', details)
  }
}

```

Após isso, são feitas duas chamadas para **setRequestHeader()**, definindo os valores de 'HX-History-Restore-Request' como verdadeiro e 'HX-Current-URL' com o valor de **href** de **location**. Após isso, é passada uma função para o parâmetro **onload** de **request**. Esta função anônima se inicia verificando se o status da requisição é maior ou igual a 200 ou menor que 400, indicando que houve sucesso entre uma transação ou redirecionamento pelo servidor. Caso o código esteja entre este limite, o valor da propriedade **response** de **details** é definido com o valor da resposta dessa requisição, e é feita uma chamada para **triggerEvent()** para definir um evento 'htmx:historyCacheMissLoad'. Após isso, é feita uma chamada para **swap()**, uma chamada para **setCurrentPathForHistory()** e outro **triggerEvent()** que declara um evento 'htmx:historyRestore'. Caso o código da requisição não esteja entre a faixa de 200 à 400, isso indica um erro de comunicação, e é feita uma chamada para **triggerErrorEvent()** que declara um erro 'htmx:historyCacheMissLoadError'.

```

if (triggerEvent(getDocument().body, 'htmx:historyCacheMiss', details)) {
  request.send()
}

```

Ao final da função, é verificado se é possível enviar um evento 'htmx:HistoryCacheMiss', e caso possa, é feita a requisição que foi construída ao longo da função.

```

getWindow().setTimeout(function() {
  triggerEvent(body, 'htmx:load', {})
  body = null
}, 0)
})

return htmx
})();

```

Por fim, ao voltar para a chamada de inicialização do *HTMX* com **ready()**, é feita uma chamada para **setTimeout()** da janela atual, passando uma função anônima que aciona um evento 'htmx:load' e é definido o valor de **body** como nulo para auxílio do coletor de lixo do *JavaScript*, terminando assim a chamada para **ready()**. Ao final do *script*, o valor da constante **htmx** definida no início do *script* é retornado como valor.

B.3 As funções do HTMX

A seguir neste trabalho, serão analisadas as funções delimitadas no capítulo anterior, que compõem um papel crucial no funcionamento interno do HTMX. Estas funções são:

- **on()**.
- **off()**.
- **onLoad()**.
- **addClassToElement()**.
- **removeClassFromElement()**.
- **toggleClassFromElement()**.
- **parseInterval()**.
- **swap()**.

B.3.1 A função *on()*

```
function addEventListenerImpl(arg1, arg2, arg3, arg4) {
  ready(function() {
    const eventArgs = processEventArgs(arg1, arg2, arg3, arg4)
    eventArgs.target
      addEventListener(
        eventArgs.event,
        eventArgs.listener,
        eventArgs.options
      )
  })
  const b = isFunction(arg2)
  return b ? arg2 : arg3
}
```

A função **addEventListenerImpl()** quatro parâmetros, **arg1**, **arg2**, **arg3** e **arg4**. No seu início faz uma chamada para a função **ready()**, que declara uma função anônima. Dentro desta função anônima, é criada uma constante **eventArgs**, que recebe o valor de retorno da chamada para **processEventArgs()** e é feita uma chamada para o método **addEventListener()** da propriedade **target** de **eventArgs()**. Por fim, é criada uma constante **b** com o valor de chamada para **isFunction()** utilizando **arg2** como parâmetro, e é retornado o valor de **arg2**, caso este seja uma função, ou de **arg3** caso contrário. É necessário notar a diferença entre **addEventListener()** e **addEventListenerImpl()**, pois ambas têm funcionalidades diferentes: **addEventListener()** é a função que propriamente adicionar eventos em uma página, enquanto **addEventListenerImpl()** é um **wrapper** que executa o anterior dentro de si.

B.3.2 A função *off()*

```
function removeEventListenerImpl(arg1, arg2, arg3) {
  ready(function() {
    const eventArgs = processEventArgs(arg1, arg2, arg3)
    eventArgs.target
      .removeEventListener(eventArgs.event, eventArgs.listener)
  })
  return isFunction(arg2) ? arg2 : arg3
}
```

A função **off()**, denominada como **removeEventListenerImpl()** pelo *HTMX* já foi vista anteriormente no capítulo de inicialização do *HTMX*. Esta função toma como parâmetros os valores de **arg1**, **arg2** e **arg3**. No seu corpo, faz uma chamada para **ready()**, que possui uma função anônima interna. Dentro desta função, é definida uma constante **eventArgs** com o valor de retorno da chamada para **processEventArgs()**. É então feita uma chamada para o método **removeEventListener()** da propriedade **target** de **eventArgs**, e é retornado o valor de **arg2**, caso este seja uma função, ou de **arg3** caso contrário ao final da função. Da mesma maneira que a função **on()**, **removeEventListener()** e **removeEventListenerImpl()** são duas funções distintas, onde uma é incorporada pela outra, respectivamente.

B.3.3 A função *onLoad()*

```
function onLoadHelper(callback) {
  const value = htmx.on('htmx:load', function(evt) {
    callback(evt.detail.elt)
  })
  return value
}
```

A função **onLoad()**, declarada como **onLoadHelper()** é definida com um parâmetro **callback**, que deve ser uma função. Na função, é criada uma constante **value** com o valor de retorno da chamada para **on()**, que escuta um evento 'htmx:load', e executa uma função anônima com um evento **evt**, que executa **callback()**, utilizando **evt.detail.elt** como parâmetro de chamada. Ao final da função, o valor de **value** é retornado pela função.

B.3.4 A função *addClassToElement()*

```
function addClassToElement(elt, clazz, delay) {
  elt = asElement(resolveTarget(elt))
  if (!elt) {
```

```

        return
    }
    if (delay) {
        getWindow().setTimeout(function() {
            addClassToElement(elt, clazz)
            elt = null
        }, delay)
    } else {
        elt.classList && elt.classList.add(clazz)
    }
}

```

A função **addClass()**, definida como **addClassToElement()** toma como parâmetros os valores **elt**, **clazz** e **delay**, e no seu início tentar resolver **elt** em seu formato de elemento através da chamada para **asElement()**. Caso o valor de **elt** seja indefinido, o que significa que o elemento não pode ser encontrado, a função encerra sua execução, caso contrário, é verificado se o valor de **delay** é definido. Caso seja, é definido um **timer** com o valor de **delay** utilizando a chamada para **setTimeout()**, fazendo uma chamada recursiva para **addClassToElement()** sem o valor de **delay** e definindo o valor de **elt** como nulo. Caso **delay** seja indefinido, é verificado se o valor da propriedade **classList** de **elt** é definido e caso seja, é feita uma chamada para o método **add()** de **classList** que adiciona a classe **clazz** para o elemento.

B.3.5 A função *removeClassFromElement()*

```

function removeClassFromElement(node, clazz, delay) {
    let elt = asElement(resolveTarget(node))
    if (!elt) {
        return
    }
    if (delay) {
        getWindow().setTimeout(function() {
            removeClassFromElement(elt, clazz)
            elt = null
        }, delay)
    } else {
        if (elt.classList) {
            elt.classList.remove(clazz)
            if (elt.classList.length === 0) {
                elt.removeAttribute('class')
            }
        }
    }
}

```

```

    }
  }
}

```

A função **removeClass()**, definida como **removeClassFromElement()**, toma como parâmetros os valores **node**, **clazz** e **delay**, similarmente à **addClass()**. No seu início é definido um valor **elt** que tenta resolver um elemento com o valor de **node**, e caso essa resolução não seja bem sucedida, a função termina sua execução. É então verificado se o valor de **delay** é definido, chamando o método **setTimeout()** da página atual e fazendo uma chamada recursiva para **removeClassElement()** sem o valor de **delay()**. Caso o valor de **delay** não seja definido, é verificado se o valor da propriedade **classList** de **elt** é definido, e caso seja, é feita uma chamada para seu método **remove()**, removendo a classe **clazz**. Ao final desta checagem, caso o valor do tamanho de **classList** seja 0, o atributo 'class' é removido de **elt**.

B.3.6 A função *toggleClassOnElement()*

```

function toggleClassOnElement(elt, clazz) {
  elt = resolveTarget(elt)
  elt.classList.toggle(clazz)
}

```

A função **toggleClass()**, denominada como **toggleClassOnElement()** toma como parâmetros **elt** e **clazz**. No seu corpo, define o valor de **elt** com a resolução de **elt** através da função **resolveTarget()**, e é feita uma chamada para o método **toggle()** de **classList**, propriedade de **elt** utilizando **clazz** como parâmetro. Este método faz parte da biblioteca padrão do *JavaScript*.

B.3.7 A função *parseInterval()*

```

function parseInterval(str) {
  if (str == undefined) {
    return undefined
  }

  let interval = NaN
  if (str.slice(-2) == 'ms') {
    interval = parseFloat(str.slice(0, -2))
  } else if (str.slice(-1) == 's') {
    interval = parseFloat(str.slice(0, -1)) * 1000
  }
}

```

```

    } else if (str.slice(-1) == 'm') {
        interval = parseFloat(str.slice(0, -1)) * 1000 * 60
    } else {
        interval = parseFloat(str)
    }
    return isNaN(interval) ? undefined : interval
}

```

A função **parseInterval()** toma um parâmetro **str** e em seu início verifica se seu valor é indefinido, retornando indefinido como valor caso seja. Caso contrário, a função cria uma variável **interval**, com o valor padrão definido como **NaN**. É então verificado se o valor de **str** possui 'ms', 's' ou 'm' ao final da **string**, indicando uma duração de milissegundos, segundos ou minutos, respectivamente. Caso qualquer um desses valores sejam encontrados dentro de **str**, o valor de **interval** receberá o valor transformado para milissegundos, caso contrário, o programa só tentará transformar **str** para um valor com ponto decimal. Ao final da função, é verificado se o valor de **interval** é igual a **NaN**. Caso seja, um valor indefinido é retornado, caso contrário o próprio valor de **interval** é retornado.

B.3.8 A função *swap()*

```

function swap(target, content, swapSpec, swapOptions) {
    if (!swapOptions) {
        swapOptions = {}
    }
}

```

A última função a ser analisada dentro do código fonte do *HTMX* é a função **swap()**. Sua definição vem com os parâmetros **target**, **content**, **swapSpec** e **swapOptions**. Em seu início, é verificado se o valor de **swapOptions** é definido, e caso não seja, um objeto vazio é atribuído à essa variável.

```

let settleResolve = null
let settleReject = null

```

Após isso, duas variáveis são criadas com valores nulos: **settleResolve** e **settleReject**.

```

let doSwap = function() {
    maybeCall(swapOptions.beforeSwapCallback)

    target = resolveTarget(target)
    const rootNode = swapOptions.contextElement ?
        getRootNode(swapOptions.contextElement, false) : getDocument()
}

```

```

const activeElt = document.activeElement
let selectionInfo = {}
selectionInfo = {
  elt: activeElt,
  start: activeElt ? activeElt.selectionStart : null,
  end: activeElt ? activeElt.selectionEnd : null
}

```

É criada então uma variável **doSwap()** que guarda uma função anônima sem parâmetros. Dentro desta função, é feita uma chamada para **maybeCall()**, e é definido o valor de **target** com o retorno da função **resolveTarget()**. Em seguida, uma constante **rootNode** é criada a partir da checagem do valor da propriedade **contextElement** de **swapOptions**. Caso o valor seja definido, o valor da chamada para **getRootNode()** é retornado, caso contrário, a chamada para **getDocument()** é retornada como valor para a constante. Após isso é criada uma constante **activeElt** com o valor da propriedade **activeElement** de **document**, e é criada uma variável **selectionInfo** que é instanciada com o valor de um objeto que possui os valores de **activeElt**, denominado como **elt**, a propriedade **selectionStart** de **activeElt** ou nulo caso esta seja indefinida, denominado como **start** e o valor da propriedade **selectionEnd** de **activeElt** ou nulo caso esta seja indefinida, denominada como **end**.

```

function maybeCall(func) {
  if (func) {
    func()
  }
}

```

A função **maybeCall()** é definida com um parâmetro **func**, e em seu corpo, verifica se o valor de **func** é definido, e caso seja, executa a função incorporada à **func**.

```

const settleInfo = makeSettleInfo(target)

if (swapSpec.swapStyle === 'textContent') {
  target.textContent = content
}

```

Ao voltar para **swap()**, é criada uma constante **settleInfo**, com o valor de retorno de **makeSettleInfo()**, utilizando **target** como parâmetro, e é verificado se o valor da propriedade **swapStyle** de **swapSpec** é estritamente igual à **string** 'textContent', definindo o valor da propriedade **textContent** de **target** com o valor de **content**.

```
function makeSettleInfo(target) {
  return { tasks: [], elts: [target] }
}
```

A função **makeSettleInfo()** toma como parâmetro um valor **target**, e retorna um objeto com os campos **tasks**, um **Array** vazio e **elts**, um **Array** populado pelo valor de **target**.

```
else {
  let fragment = makeFragment(content)

  settleInfo.title = swapOptions.title ||
  fragment.title
```

Voltando para **swap()**, caso a estrutura condicional anterior não seja executada, uma variável **fragment** é criada com o valor de retorno da função **makeFragment()**, utilizando **content** como parâmetro, e em seguida, o valor da propriedade **title** de **settleInfo** é definido com o valor da propriedade **title** de **fragment** caso o valor da propriedade **title** de **swapOptions** seja definido, ou nulo caso contrário.

```
makeFragment(response) {
  const responseWithNoHead = response
  .replace(/<head(\s[^>]*)?>[\s\S]*?</head>/i, "")
  const startTag = getStartTag(responseWithNoHead)
```

A função **makeFragment()** toma como parâmetro uma variável **response**. Em seu início, é criada uma constante **responseWithNoHead** que recebe o valor de uma chamada para **replace()**, e utilizando uma expressão regular que remove completamente uma **tag HTML 'head'**. Após isso, é criada uma constante **startTag** que recebe o valor de retorno da chamada para **getStartTag()**.

```
function getStartTag(str) {
  const tagMatcher = /<([a-z][^\\0>\x20\t\r\n\f]*)/i
  const match = tagMatcher.exec(str)
  if (match) {
    return match[1].toLowerCase()
  } else {
    return ""
  }
}
```

A função **getStartTag()** tem um parâmetro **str**. Em seu início, cria uma constante **tagMatcher** com o valor de um **regex** que procura qualquer **tag HTML**, e uma constante **match** que recebe o valor da execução do **regex** de **tagMatcher** no valor de **str**. Caso o valor de **matcher** seja definido, o valor da segunda instância definida em **match** é retornado como valor, transformado para uma **string** em caixa baixa, caso contrário, uma **string** vazia é retornada pela função.

```
if (startTag === 'html') {
  fragment = (new DocumentFragment())
  const doc = parseHTML(response)
  takeChildrenFor(fragment, doc.body)
  fragment.title = doc.title
}
```

Após a execução de **getStartTag()** em **makeFragment()**, é criada uma variável **fragment** sem nenhum valor declarado inicialmente. É então feita uma avaliação do valor de **startTag**, e caso essa **tag** tenha o valor de 'html', é então definido o valor de **fragment** com um novo objeto de **DocumentFragment**, uma constante **doc** com o valor de **parseHTML()**, uma chamada para **takeChildrenFor()** e é definido o valor da propriedade **title** de **fragment** com o valor da propriedade **title** de **doc**.

```
function takeChildrenFor(fragment, elt) {
  while (elt.childNodes.length > 0) {
    fragment.append(elt.childNodes[0])
  }
}
```

A função **takeChildrenFor** toma como parâmetros os valores **fragment** e **elt**, e durante sua execução, executa um laço de repetição que tem sua duração definida pelo valor do tamanho da propriedade **childNodes** de **elt**, e para cada execução do laço, é adicionado ao final de **fragment** o valor no índice 0 da propriedade **childNodes** de **elt**.

```
function parseHTML(resp) {
  const parser = new DOMParser()
  return parser.parseFromString(resp, 'text/html')
}
```

A função **parseHTML()** toma como parâmetro um valor **resp**, e em seu corpo, cria uma constante **parser** que recebe uma nova instância de **DOMParser** e retorna o valor da chamada para o método **parseFromString()** utilizando o valor de **resp**, transformando o fragmento da resposta sem a **tag** 'head' em um documento **HTML**.

```

else if (startTag === 'body') {
  fragment = (new DocumentFragment())
  const doc = parseHTML(responseWithNoHead)
  takeChildrenFor(fragment, doc.body)
  fragment.title = doc.title
}

```

Ao voltar para **makeFragment()**, caso a estrutura condicional anterior seja falsa, é então verificado se o valor de **startTag** é igual ao valor de 'body'. Caso seja, o valor de **fragment** é definido com a criação de uma nova instância do valor de **DocumentFragment**, o valor de **doc** recebe o valor de **parseHTML()**, é feita uma chamada para **takeChildrenFor()** e o valor da propriedade **title** de **fragment** recebe o valor da propriedade **title** de **doc**.

```

else {
  const doc = parseHTML('<body><template \
class="internal-hmx-wrapper">' +
responseWithNoHead +
'</template></body>')
  fragment = (doc.querySelector('template').content)
  fragment.title = doc.title

  var titleElement = fragment.querySelector('title')
  if (titleElement && titleElement.parentNode === fragment) {
    titleElement.remove()
    fragment.title = titleElement.innerText
  }
}

```

Caso nenhuma das expressões anteriores seja verdadeira, uma última parte da estrutura é executada, onde é inicialmente criada uma constante **doc** com o valor da chamada de **parseHTML()** criando um documento com o valor da variável **responseWithNoHead** cercado por **tags** 'body' e 'template'. Em seguida o valor de **fragment** é definido com o valor da chamada do método **querySelector()** no elemento **doc**, e o valor da propriedade **title** de **fragment** é definido com o valor da propriedade **title** de **doc**. Após isso, é criada uma variável **titleElement** que recebe o valor de retorno da chamada para o método **querySelector()** de **fragment**, procurando por uma **tag** 'title' no documento. Caso haja uma **tag** 'title' no documento, este é removido de **fragment**, definindo o valor de sua propriedade **title** com o valor de **innerText** de **titleElement**.

```

if (fragment) {
  if (htmx.config.allowScriptTags) {

```

```

        normalizeScriptTags(fragment)
    } else {
        fragment.querySelectorAll('script')
            .forEach((script) => script.remove())
    }
}
return fragment

```

Por fim, na função **makeFragment()**, é verificado se o valor de **fragment** é um valor definido. Caso seja, é então verificado se o valor de configuração do *HTMX* **allowScriptTags** é verdadeiro. Caso seja, é feita uma chamada para **normalizeScriptTags()**, mas caso não, as **tags** 'script' do documento são removidas através de uma função anônima dentro de um iterador nos valores de **fragment**. Ao final da função, o valor de **fragment** é retornado.

```

function normalizeScriptTags(fragment) {
    Array.from(fragment
        .querySelectorAll('script'))
        .forEach( (script) => {
            if (isJavaScriptScriptNode(script)) {
                const newScript = duplicateScript(script)
                const parent = script.parentNode
                try {
                    parent.insertBefore(newScript, script)
                } catch (e) {
                    logError(e)
                } finally {
                    script.remove()
                }
            }
        })
}

```

A função **normalizeScriptTags()** toma como parâmetro um valor **fragment**. No seu início, é criado um **Array** com o valor do **script** contido na **tag** 'script' e é feita uma iteração sobre este valor, e é executada uma função anônima com o valor individual da iteração mapeado para a variável **script**. Dentro da função anônima, é verificado se o valor de retorno da função **isJavaScriptScriptNode()** é verdadeiro, e caso seja, são criadas as constantes **newScript** com o valor da chamada para **duplicateScript()** e **parent** com o valor da propriedade **parentNode** de **script**. É então criada uma estrutura de tratamento de erros, que tenta adicionar o valor de **newScript** antes do valor de **script** em **parent** através

do método **insertBefore()**. Caso esta execução gere um erro, o erro é desviado para a função **logError()** e o valor de **script** é removido do **Array** definido no começo.

```
function isJavaScriptScriptNode(script) {
    return script.matches('script') &&
    (script.type === 'text/javascript' ||
    script.type === 'module' ||
    script.type === '')
}
```

A função **isJavaScriptScriptNode()** toma como parâmetro um valor **script** e retorna uma expressão booleana que verifica se: a **tag** 'script' é definido com o nome de 'script' e se **script** é definido como um tipo 'text/javascript', "module" ou uma **string** vazia.

```
function duplicateScript(script) {
    const newScript = getDocument().createElement('script')
    forEach(script.attributes, function(attr) {
        newScript.setAttribute(attr.name, attr.value)
    })
    newScript.textContent = script.textContent
    newScript.async = false
    if (htmx.config.inlineScriptNonce) {
        newScript.nonce = htmx.config.inlineScriptNonce
    }
    return newScript
}
```

A função **duplicateScript()** toma como parâmetro um valor **script**. Em seu início, cria uma constante **newScript**, que recebe o valor da chamada do método **createElement()** de **DOM** para criar uma **tag** 'script'. Após isso, para cada atributo definido em **script**, é executada uma função anônima que mapeia este valor para o nome de **attr**. Nesta função é feita uma chamada para o método **setAttribute()** de **newScript()** com os valores das propriedades **name** e **value** de **attr**. Após a função anônima, o valor da propriedade **textContent** de **newScript** recebe o valor da propriedade **textContent** de **script**, e o valor da propriedade **async** de **newScript** recebe um valor falso. Por fim, caso o valor de configuração **inlineScriptNonce** seja verdadeiro, o valor da propriedade **nonce** de **newScript** recebe o valor deste valor de configuração, e o valor de **newScript** é retornado ao final da função.

```
if (swapOptions.historyRequest) {
    fragment = fragment
```

```

        .querySelector(['hx-history-elt'],[data-hx-history-elt'])
        || fragment
    }
}

```

Ao voltar para **swap()**, é declarada uma estrutura condicional que verifica se a propriedade **historyRequest** de **swapOptions** é um valor definido. Caso seja, o valor de **fragment** é definido com o valor de **tags** que possuem os atributos 'hx-history-elt' ou 'data-hx-history-elt' ou o próprio valor de **fragment** caso estes não existam dentro de **fragment**.

```

if (swapOptions.selectOOB) {
    const oobSelectValues = swapOptions.selectOOB.split(',')
    for (let i = 0; i < oobSelectValues.length; i++) {
        const oobSelectValue = oobSelectValues[i].split(':', 2)
        let id = oobSelectValue[0].trim()
        if (id.indexOf('#') === 0) {
            id = id.substring(1)
        }
        const oobValue = oobSelectValue[1] || 'true'
        const oobElement = fragment.querySelector('#' + id)
        if (oobElement) {
            oobSwap(oobValue, oobElement, settleInfo, rootNode)
        }
    }
}

```

Após isso, é criada uma estrutura condicional que verifica se o valor da propriedade **selectOOB** de **swapOptions** é definido. Caso seja, é criado uma constante **oobSelectValues** com o valor de um **Array** que recebe os valores a partir da divisão do valor da **string** contida em **selectOOB** pelas vírgulas. Após isso, é criada uma estrutura de repetição que tem sua duração definida pelo tamanho de **oobSelectValues**. Dentro da iteração, é criado uma constante **oobSelectValue** com o valor de cada item ordenado de **oobSelectValues**, dividido em 2 por um caractere ':', e em seguida é criada uma variável **id** que recebe o valor da primeira posição de **oobSelectValue** com o seu valor sem espaços em branco antes ou depois da **string**. Após isso, é verificado se o valor de **id** não possui nenhum caractere '#', e caso não haja, o valor de **id** recebe o valor da chamada para o método **substring** que utiliza todo o conteúdo em **id** com exceção do primeiro caractere. Após isso, é criada uma constante **oobValue** com o valor da segunda posição de **oobSelectValue** ou com um valor verdadeiro caso esta não exista, e é criado uma constante **oobElement** com o valor da chamada para o método **querySelector()** utilizando o valor de **id** como parâmetro. Após isso, caso o valor de **oobElement** seja definido, é feita uma chamada para **oobSwap()**.

```
function oobSwap(oobValue, oobElement, settleInfo, rootNode) {
  rootNode = rootNode || getDocument()
  let selector = '#' + CSS.escape(getRawAttribute(oobElement, 'id'))
  let swapStyle = 'outerHTML'
```

A função **oobSwap()** é definida com os parâmetros **oobValue**, **oobElement**, **settleInfo** e **rootNode**. No seu início é definido o valor do parâmetro **rootNode** com o seu próprio valor, caso exista, ou com o valor da chamada para **getDocument()**, que retornará todo o documento *HTML*. Em seguida é criada uma variável **selector** que recebe o valor de uma **string** com o primeiro caractere definido como '#' e em seguida o valor é definido com o valor do atributo **id** dentro do elemento ou fragmento *HTML* contido em **oobElement** através da função **getRawAttribute()**, e uma variável **swapStyle** é definida com o valor de uma **string** com o valor de 'outerHTML'.

```
if (oobValue === 'true') {
} else if (oobValue.indexOf(':') > 0) {
  swapStyle = oobValue.substring(0, oobValue.indexOf(':'))
  selector = oobValue.substring(oobValue.indexOf(':') + 1)
} else {
  swapStyle = oobValue
}
```

Em seguida, é criada uma estrutura condicional que verifica se o valor de **oobValue** é estritamente igual à **string** 'true', executando nada caso seja verdadeiro. Em seguida, caso a condição anterior não seja verdadeira, é verificado se o valor do índice do primeiro caractere ':' dentro de **oobValue** é maior que zero, e caso a checagem seja verdadeira o valor de **swapStyle** é definido pelo corte da **string** em **oobValue** desde o início até o primeiro caractere ':', e o valor de **selector** é definido com o valor do corte da **string** em **oobValue** começando pela posição do primeiro caractere ':' somado à um até o final da **string**. Caso nenhuma das duas estruturas condicionais anteriores seja executada, o valor de **swapStyle** é definido com o valor de **oobValue**.

```
oobElement.removeAttribute('hx-swap-oob')
oobElement.removeAttribute('data-hx-swap-oob')
```

```
const targets = querySelectorAllExt(rootNode, selector, false)
```

É então feito duas chamadas para o método **removeAttribute()** de **oobElement** removendo as classes 'hx-swap-oob' e 'data-hx-swap-oob', e em seguida uma constante **targets** é criada com o valor de retorno da chamada para **querySelectorAllExt()**.

```

if (targets.length) {
  forEach(
    targets,
    function(target) {
      let fragment
      const oobElementClone = oobElement.cloneNode(true)
      fragment = getDocument().createDocumentFragment()
      fragment.appendChild(oobElementClone)
    }
  )
}

```

Uma estrutura condicional é criada que verifica se o valor do tamanho de **targets** é definido. Caso seja verdadeiro, uma iteração sobre os valores de **targets** é feita, criando uma função anônima que mapeia cada elemento iterado para a variável **target**. Dentro desta iteração é criado uma variável **fragment** sem um valor definido, e uma constante **oobElementClone** é definida com uma chamada para o método **cloneNode()** de **oobElement**. Em seguida, é definido o valor de **fragment** com o valor do documento *HTML* transformado em um fragmento do documento, e em seguida, é feita uma chamada para o método **appendChild()** com o valor de **oobElementClone**.

```

if (!isInlineSwap(swapStyle, target)) {
  fragment = asParentNode(oobElementClone)
}

```

É então checado se o valor da chamada para **isInlineSwap()** com os valores de **swapStyle** e **target** é verdadeira, definindo o valor de **fragment** com a chamada para **asParentNode()** utilizando o valor de **oobElementClone**.

```

function isInlineSwap(swapStyle, target) {
  const extensions = getExtensions(target)
  for (let i = 0; i < extensions.length; i++) {
    const extension = extensions[i]
    try {
      if (extension.isInlineSwap(swapStyle)) {
        return true
      }
    } catch (e) {
      logError(e)
    }
  }
  return swapStyle === 'outerHTML'
}

```

A definição de **isInlineSwap()** toma como parâmetros os valores **swapStyle** e **target**. Em seu corpo, é criada uma constante **extensions** que recebe o valor de retorno para a função **getExtensions()**. Em seguida, é criada uma estrutura de repetição que tem a duração definida pelo tamanho da propriedade **length** de **extensions**. Dentro da estrutura de repetição é criada uma constante **extension** com o valor do índice atual de **extensions**. Em seguida, é criada uma estrutura de tratamento de erros que tenta realizar a execução do método **isInlineSwap()** para cada iteração de **extension**. Caso haja erros durante a execução da estrutura de tratamento de erros, é feita uma chamada para **logError()** que trata o erro adequadamente. Ao final da função, é retornado o valor da comparação de **swapStyle** com a **string** 'outerHTML': caso sejam iguais é retornado verdadeiro, caso contrário, o valor retornado é falso.

```
const beforeSwapDetails = { shouldSwap: true, target, fragment }
if (!triggerEvent(target, 'htmx:oobBeforeSwap', beforeSwapDetails)) return

target = beforeSwapDetails.target
if (beforeSwapDetails.shouldSwap) {
  handlePreservedElements(fragment)
  swapWithStyle(swapStyle, target, target, fragment, settleInfo)
  restorePreservedElements()
}
```

Ao voltar para **oobSwap()**, é criada uma constante **beforeSwapDetails** que recebe o valor de um objeto com os campos **shouldSwap**, com um valor verdadeiro, **target** com o valor da variável **target** e **fragment** com o valor da variável **fragment**. Após isso é verificado se o valor da chamada para **triggerEvent()** produz algum valor de erro, e caso produza, a função é encerrada precocemente. Em seguida, o valor de **target** é definido com o valor da propriedade **target** de **beforeSwapDetails**. Depois o valor da propriedade **shouldSwap** de **beforeSwapDetails** é avaliado, e caso seja verdadeiro, as funções **handlePreservedElement()**, **swapWithStyle()** e **restorePreservedElements()** são executadas na ordem mencionada.

```
function handlePreservedElements(fragment) {
  forEach(findAll(fragment,
    '[hx-preserve],
    [data-hx-preserve]'),
    function(preservedElt) {
      const id = getAttributeValue(preservedElt, 'id')
      const existingElement = getDocument().getElementById(id)
      if (existingElement !== null) {
```

```

    if (preservedElt.moveBefore) {
      let pantry = find('#--htmx-preserve-pantry--')
      if (pantry == null) {
        getDocument()
        .body
        .insertAdjacentHTML('afterend',
          "<div id='--htmx-preserve-pantry--'></div>")
        pantry = find('#--htmx-preserve-pantry--')
      }
      pantry.moveBefore(existingElement, null)
    } else {
      preservedElt.parentNode
      .replaceChild(existingElement, preservedElt)
    }
  }
})
}

```

A função **handlePreservedElements()** toma como parâmetro um valor **fragment**. Em seu corpo, a função irá procurar todos os valores dentro do trecho *HTML* em **fragment** que possuem os atributos 'hx-preserve' ou 'data-hx-preserve' e irá criar uma iteração sobre eles, capturando o valor dentro da variável **preservedElt**. Dentro da iteração, uma constante **id** é criada com o valor do atributo de 'id' de **preservedElt** e uma constante **existingElement** é criada com o valor do elemento que possui o valor de **id** em **id**. Em seguida é verificado se o valor de **existingElement** é nulo, e caso não seja, é então checado se o valor da propriedade **moveBefore** de **preservedElt** é definido, e uma variável **pantry** é criada com o valor de um fragmento *HTML* que possui o id '--htmx-preserve-pantry--'. Caso não haja um elemento com esse **id**, a função criará uma **tag** 'div' com este **id** que será usado como armazenamento para preservar os elementos escolhidos. Por fim, é feita uma chamada para o método **moveBefore()** de **pantry**. Caso o valor da propriedade **moveBefore** de **preservedElt** seja indefinido, a função apenas troca os elementos atuais do nó pai de **preservedElt** pelo seu valor.

```

function swapWithStyle(swapStyle, elt, target, fragment, settleInfo) {
  switch (swapStyle) {
    case 'none':
      return
    case 'outerHTML':
      swapOuterHTML(target, fragment, settleInfo)
      return
  }
}

```

A função **swapStyle()** é definida com os parâmetros **swapStyle**, **elt**, **target**, **fragment** e **settleInfo**. No seu corpo, é definido um **switch case** que avalia o valor de **swapStyle**. De acordo com o seu valor, uma função diferente é chamada, com exceção do caso padrão. O primeiro caso testado é o valor 'none', que encerra a função com um retorno vazio. Em seguida, o caso testado é o valor 'outerHTML', que executa **swapOuterHTML()**.

```
function swapOuterHTML(target, fragment, settleInfo) {
  if (target.tagName === 'BODY') {
    return swapInnerHTML(target, fragment, settleInfo)
  }

  let newElt
  const eltBeforeNewContent = target.previousSibling
  const parentNode = parentElt(target)
  if (!parentNode) {
    return
  }
  insertNodesBefore(parentNode, target, fragment, settleInfo)
```

A função **swapOuterHTML()** é definida com os parâmetros **target**, **fragment** e **settleInfo**. No seu começo é feita uma checagem sobre o valor da propriedade **tagName** de **target**, verificando se o valor desta propriedade é igual à 'BODY'. Caso seja, isto indica que o valor da **tag HTML** é uma **tag 'body'**, e uma chamada para **swapInnerHTML()** é feita. Após essa checagem uma variável **newElt** é criada sem valor definido, uma constante **eltBeforeNewContent** é criada com o valor da propriedade **previousSibling** de **target** e uma constante **parentNode** é criada com o valor de retorno da chamada para **parentElt()**. Logo após, é verificado se o valor de **parentNode** é definido, retornando a função precocemente neste ponto, caso este seja o caso. Caso contrário, é feita uma chamada para a função **insertNodesBefore()**.

```
function insertNodesBefore(
  parentNode,
  insertBefore,
  fragment,
  settleInfo) {
  handleAttributes(parentNode, fragment, settleInfo)
  while (fragment.childNodes.length > 0) {
    const child = fragment.firstChild
    addClassToElement(asElement(child), htmx.config.addedClass)
    parentNode.insertBefore(child, insertBefore)
```

```

    if (child.nodeType !== Node.TEXT_NODE &&
        child.nodeType !== Node.COMMENT_NODE) {
        settleInfo.tasks.push(makeAjaxLoadTask(child))
    }
}
}

```

A função **insertNodesBefore()** toma como parâmetros as variáveis **parentNode**, **insertBefore**, **fragment** e **settleInfo**. No seu início é feita uma chamada para a função **handleAttributes()**, e em seguida é criada uma estrutura de repetição que é executada enquanto o valor do tamanho da propriedade **childNodes** de **fragment** seja maior que zero. Dentro desta estrutura, uma constante **child** é criada e recebe o valor da propriedade **firstChild** de **fragment**, e é adicionado uma classe para o elemento contido em **child**, através de **addClassToElement()**. Logo após, é feita uma chamada para o método **insertBefore()** de **parentNode**, e é verificado se o tipo da propriedade **nodeType** de **child** é diferente de uma **string** e de um comentário em formato *HTML*, e caso este seja o caso, o valor de retorno da função **makeAjaxLoadTask()** é adicionado como valor na propriedade **tasks** de **settleInfo**.

```

function handleAttributes(parentNode, fragment, settleInfo) {
    forEach(fragment.querySelectorAll('[id]'), function(newNode) {
        const id = getRawAttribute(newNode, 'id')
        if (id && id.length > 0) {
            const normalizedId = id.replace(/"/g, '\\"')
            const normalizedTag = newNode
            .tagName
            .replace(':', '\\:')
            const parentElt = asParentNode(parentNode)
            const oldNode = parentElt &&
            parentElt.querySelector(normalizedTag +
                "[id='" + normalizedId + "']")
            if (oldNode && oldNode !== parentElt) {
                const newAttributes = newNode.cloneNode()
                cloneAttributes(newNode, oldNode)
                settleInfo.tasks.push(function() {
                    cloneAttributes(newNode, newAttributes)
                })
            }
        }
    })
}

```

```

    })
  }

```

A função **handleAttributes()** toma como parâmetro os valores **parentNode**, **fragment** e **settleInfo**. No seu início é feita uma iteração sobre todas as **tags HTML** em **fragment** que possuem um atributo **id** definido, capturando essas **tags** como fragmentos em uma variável **newNode** dentro de uma função anônima. Dentro desta função, é criada uma constante **id** que recebe o valor do atributo 'id' da **tag HTML** dentro de **newNode**, e é verificado se o valor de **id** é definido e se o valor de **id** tem tamanho maior que zero. Caso este seja o caso, são criadas as constantes **normalizedId()**, com o valor de **id** formatado com os caracteres de apóstrofes transformados em uma sequência com um caractere de barra e um caractere de apóstrofe, uma constante **normalizedTag** que recebe o valor da propriedade **tagName** de **newNode** com o seu valor formatado com os caracteres de dois pontos transformados em uma sequência com um caractere de barra e um caractere de dois pontos, uma constante **parentElt** que recebe o valor de retorno da chamada para a função **asParentNode()**, e uma constante **oldNode** que recebe o valor da **tag** contida em **parentElt** que possui o **id** definido em **normalizedId** e tem o nome da **tag** **normalizedTag**, caso o valor de **parentElt** seja definido. É então verificado se o valor de **oldNode** é definido e se ele é diferente de **parentElt()**, e caso esta condição seja verdade, é criada uma constante **newAttributes** com o valor clonado de **newNode**, uma chamada para **cloneAttributes()** é feita para atribuir os atributos de **oldNode** para **newNode**, e é adicionado uma função anônima na propriedade **tasks** de **settleInfo** que executa mais uma vez a função **cloneAttributes()**, desta vez clonando os atributos de **newAttributes** e adicionando-os em **newNode**.

```

function cloneAttributes(mergeTo, mergeFrom) {
  forEach(mergeTo.attributes, function(attr) {
    if (!mergeFrom.hasAttribute(attr.name) &&
        shouldSettleAttribute(attr.name)) {
      mergeTo.removeAttribute(attr.name)
    }
  })
  forEach(mergeFrom.attributes, function(attr) {
    if (shouldSettleAttribute(attr.name)) {
      mergeTo.setAttribute(attr.name, attr.value)
    }
  })
}

```

A função **cloneAttributes()** tem como parâmetros as variáveis **mergeTo** e **mergeFrom**. Em seu corpo, é criada uma iteração sobre os valores da propriedade **attributes** de

mergeTo, capturando-os em uma função anônima que verifica se o valor de **mergeFrom()** não possui o atributo checado e se o valor de retorno da função **shouldSettleAttribute()** é verdadeiro, executando uma chamada para o método **removeAttribute()** caso a condição seja avaliada como verdade. Em seguida, é feita uma segunda iteração, desta vez sobre os valores da propriedade **attributes** de **mergeFrom()**, capturando-os em uma função anônima e verificando se o valor de retorno da função **shouldSettleAttribute()** é verdadeiro, e caso a condição seja avaliada como verdadeira, é feita uma chamada para o método **setAttribute()** de **mergeTo**.

```
function shouldSettleAttribute(name) {
    return htmx.config.attributesToSettle.includes(name)
}
```

A função **shouldSettleAttribute()** toma como parâmetro único a variável **name**, e em seu corpo, retorna o valor do método **includes()** da propriedade **attributesToSettle** da configuração do *HTMX*.

Por fim, para terminar a compreensão de **insertNodesBefore()**, falta a análise de **makeAjaxLoadTask()**.

```
function makeAjaxLoadTask(child) {
    return function() {
        removeClassFromElement(child, htmx.config.addedClass)
        processNode(asElement(child))
        processFocus(asParentNode(child))
        triggerEvent(child, 'htmx:load')
    }
}
```

A função **makeAjaxLoadTask()** toma como parâmetro único a variável **child**. Em seu corpo, retorna uma função anônima que realiza chamada para as funções **removeClassFromElement()**, **processNode()**, **processFocus()** e **triggerEvent()**. A única destas funções não vista até aqui é **processFocus()**.

```
function processFocus(child) {
    const autofocus = '[autofocus]'
    const autoFocusedElt = asHtmlElement(matches(child, autofocus) ?
    child : child.querySelector(autofocus))
    if (autoFocusedElt != null) {
        autoFocusedElt.focus()
    }
}
```

A função **processFocus()** toma como parâmetro único a variável **child** e em seu corpo, cria as constantes **autofocus**, com o valor da **string** '[autofocus]' e **autoFocusedElt** que recebe o próprio valor de **child** caso este possua o atributo **autofocus** ou recebe o valor dos elementos encontrados dentro de **child** que possuam o atributo **autofocus**. Em seguida, caso **autoFocusedElt** não seja nulo, é chamado o método **focus()** de **autoFocusedElt**, e a função se encerra.

```
if (eltBeforeNewContent == null) {
    newElt = parentNode.firstChild
} else {
    newElt = eltBeforeNewContent.nextSibling
}
settleInfo.elts = settleInfo.elts.filter(function(e) { return e !== target })
```

Ao voltar para **swapOuterHTML()**, é verificado se o valor de **eltBeforeNewContent** é nulo. Caso seja, a variável **newElt** recebe o valor da propriedade **firstChild** de **parentNode**. Caso contrário, **newElt** recebe o valor da propriedade **nextSibling** de **eltBeforeNewContent**. Em seguida, o valor da propriedade **elts** de **settleInfo** recebe o valor de retorno da chamada para **filter()** da propriedade **elts** de **settleInfo**, utilizando uma função anônima que verifica se cada valor contido é diferente de **target**.

```
while (newElt && newElt !== target) {
    if (newElt instanceof Element) {
        settleInfo.elts.push(newElt)
    }
    newElt = newElt.nextSibling
}
cleanUpElement(target)
target.remove()
```

Por fim, é criada uma estrutura de repetição que dura enquanto **newElt** for definido e enquanto **newElt** ter um valor diferente de **target**. Dentro desta estrutura, é verificado se o valor de **newElt** é uma instância de **Element**, e caso seja, o valor de **newElt** é adicionado ao final da propriedade **elts** de **settleInfo**, e ao final do laço o valor de **newElt** recebe o valor da propriedade **nextSibling** de **newElt**. Ao final da função, são feitas as chamadas para a função **cleanUpElement()** e para o método **remove()** de **target**.

```
case 'afterbegin':
    swapAfterBegin(target, fragment, settleInfo)
    return
case 'beforebegin':
```

```

    swapBeforeBegin(target, fragment, settleInfo)
    return
  case 'beforeend':
    swapBeforeEnd(target, fragment, settleInfo)
    return
  case 'afterend':
    swapAfterEnd(target, fragment, settleInfo)
    return

```

Em seguida, na ordem de checagem do **switch case** em **swapWithStyle()**, há a verificação por ‘afterbegin’, ‘beforebegin’, ‘beforeend’ e ‘afterend’, executando, em ordem, as funções **swapAfterBegin()**, **swapBeforeBegin()**, **swapBeforeEnd()** e **swapAfterEnd()**.

```

function swapAfterBegin(target, fragment, settleInfo) {
  return insertNodesBefore(
    target,
    target.firstChild,
    fragment,
    settleInfo)
}

```

```

function swapBeforeBegin(target, fragment, settleInfo) {
  return insertNodesBefore(
    parentElt(target),
    target,
    fragment,
    settleInfo)
}

```

```

function swapBeforeEnd(target, fragment, settleInfo) {
  return insertNodesBefore(
    target,
    null,
    fragment,
    settleInfo)
}

```

```

function swapAfterEnd(target, fragment, settleInfo) {
  return insertNodesBefore(

```

```

    parentElt(target),
    target.nextSibling,
    fragment,
    settleInfo)
}

```

As quatro funções possuem os mesmos parâmetros, **target**, **fragment** e **settleInfo** e o mesmo corpo, neste caso, o retorno do valor da chamada para **insertNodesBefore()**, com uma pequena diferença entre eles. Em **swapAfterBegin()** o primeiro parâmetro é **target** e o segundo é a propriedade **firstChild** de **target**, em **swapBeforeBegin()** o primeiro parâmetro é o elemento pai de **target** e o segundo parâmetro é **target**, em **swapBeforeEnd()** o primeiro parâmetro é **target** e o segundo é um valor nulo e em **swapAfterEnd()** o primeiro parâmetro é o elemento pai de **target** e o segundo parâmetro é a propriedade **nextSibling** de **target**.

```

case 'delete':
    swapDelete(target)
    return

function swapDelete(target) {
    cleanUpElement(target)
    const parent = parentElt(target)
    if (parent) {
        return parent.removeChild(target)
    }
}

```

A função **swapDelete()** toma como parâmetro único a variável **target**. Em seu corpo, é feita uma chamada para **cleanUpElement()**, uma constante **parent** é criada com o valor da chamada para **parentElt()** utilizando **target** como parâmetro, e é verificado se **parent** tem um valor definido, retornando uma chamada para o método **removeChild()** caso esta condição seja verdadeira.

```

default:
    var extensions = getExtensions(elt)
    for (let i = 0; i < extensions.length; i++) {
        const ext = extensions[i]

```

Por fim, existe o caso padrão, executado na hipótese de que nenhum dos casos anteriores seja executado. Dentro dele, é criado uma variável **extensions** com o valor de retorno da chamada para **getExtensions()**, e é criada uma estrutura de repetição que tem

a duração definida a partir do tamanho de **extensions**. Dentro da estrutura de repetição, para cada iteração um valor **ext** captura o valor do índice **i** de **extensions**.

```
try {
  const newElements = ext
    .handleSwap(swapStyle, target, fragment, settleInfo)
  if (newElements) {
    if (Array.isArray(newElements)) {
      for (let j = 0; j < newElements.length; j++) {
        const child = newElements[j]
        if (child.nodeType !== Node.TEXT_NODE &&
            child.nodeType !== Node.COMMENT_NODE) {
          settleInfo
            .tasks
            .push(makeAjaxLoadTask(child))
        }
      }
    }
    return
  }
} catch (e) {
  logError(e)
}
```

Em seguida, é criada uma estrutura de tratamento de erros. Dentro dela, é criada uma constante **newElements** que recebe o valor de retorno do método **handleSwap()** de **ext**. É então verificado se o valor de **newElements** é definido e caso seja, é verificado se o valor de **newElements** é do tipo **Array**. Caso ambas as expressões anteriores sejam verdadeiras, é criada uma estrutura de repetição que possui a duração definida pelo tamanho de **newElements**. Na estrutura de repetição, é criada uma constante **child** que recebe o valor do índice **j** de **newElements**, e é verificado se o valor da propriedade **nodeType** de **child** é diferente dos tipos de **string** e comentário *HTML*. Caso a expressão anterior seja verdadeira, o valor de retorno da função **makeAjaxLoadTask()** é adicionado como valor para a propriedade **tasks** de **settleInfo**, e a função termina sua execução. Caso algum erro seja gerado durante a execução dessa função, será tratado redirecionado para a função **logError()**, onde será tratado.

```
if (swapStyle === 'innerHTML') {
  swapInnerHTML(target, fragment, settleInfo)
} else {
```

```

    swapWithStyle(
    htmx.config.defaultSwapStyle,
    elt,
    target,
    fragment,
    settleInfo)
  }

```

Por fim, fora da estrutura de repetição mais exterior dentro do caso padrão, é verificado se o valor de **swapStyle** é igual à **string** 'innerHTML'. Caso essa expressão seja verdadeira, a função **swapInnerHTML()** será executada, caso contrário, ocorrerá uma execução recursiva da função **swapWithStyle()**, marcando assim o fim da função **swapWithStyle()**.

Para concluir o entendimento de **swapWithStyle()**, basta observar o funcionamento de **swapInnerHTML()** agora.

```

function swapInnerHTML(target, fragment, settleInfo) {
  const firstChild = target.firstChild
  insertNodesBefore(target, firstChild, fragment, settleInfo)
  if (firstChild) {
    while (firstChild.nextSibling) {
      cleanUpElement(firstChild.nextSibling)
      target.removeChild(firstChild.nextSibling)
    }
    cleanUpElement(firstChild)
    target.removeChild(firstChild)
  }
}

```

A função **swapInnerHTML()** toma como parâmetros as variáveis **target**, **fragment** e **settleInfo**. Em seu corpo, é criada uma constante **firstChild** com o valor da propriedade **firstChild** de **target**, e é feita uma chamada para a função **insertNodeBefore()**. Em seguida, é verificado se o valor de **firstChild** é definido, criando uma estrutura de repetição que dura enquanto o valor da propriedade **nextSibling** de **firstChild** for definido. Dentro desta estrutura, são feitas as chamadas para a função **cleanUpElement()** e para o método **removeChild** de **target**. Já fora da estrutura de repetição, são feitas mais uma chamada para **cleanUpElement()** e mais uma chamada para o método **removeChild()** de **target**.

```

forEach(settleInfo.elts, function(elt) {
  triggerEvent(elt, 'htmx:oobAfterSwap', beforeSwapDetails)
})

```

Ao voltar para **oobSwap()**, é feita uma iteração sobre os valores da propriedade **elts** de **settleInfo**, e seus valores são capturados na variável **elt** dentro de uma função anônima que executa uma chamada para **triggerEvent()**.

```
oobElement.parentNode.removeChild(oobElement)
} else {
  oobElement.parentNode.removeChild(oobElement)
  triggerErrorEvent(getDocument().body, 'htmx:oobErrorNoTarget', { content: oobElement })
}
return oobValue
```

Ao final de **oobSwap()**, é feita uma chamada para o método **removeChild()** da propriedade **parentNode** de **oobElement**. Caso a condição da estrutura condicional anterior seja falsa, neste caso, se o tamanho de **targets** for indefinido, serão feitas duas chamadas para o método **removeChild()** da propriedade **parentNode** de **oobElement** e para a função **triggerErrorEvent()**. Por fim, a função retorna como valor a variável **oobValue**.

```
findAndSwapOobElements(fragment, settleInfo, rootNode)
forEach(findAll(fragment, 'template'), function(template) {
  if (template.content && findAndSwapOobElements(template.content, settleInfo, rootNode)) {
    template.remove()
  }
})
```

Ao voltar para **swap()**, há a execução da função **findAndSwapOobElements()** e uma iteração sobre os valores dentro de **fragment** que são **tags** 'template' capturados pela variável **template** dentro de uma função anônima. Nesta função anônima, é verificado se o valor da propriedade **content** de **template** é definido e se a chamada para **findAndSwapOobElements()** é definida. Caso a expressão seja avaliada como verdade, o método **remove()** de **template** é executado.

```
function findAndSwapOobElements(fragment, settleInfo, rootNode) {
  var oobElts = findAll(fragment,
    '[hx-swap-oob],
    [data-hx-swap-oob]')
  forEach(oobElts, function(oobElement) {
    if (htmx.config.allowNestedOobSwaps ||
      oobElement.parentNode === null) {
      const oobValue = getAttributeValue(
        oobElement,
        'hx-swap-oob')
    }
  })
}
```

```

        if (oobValue !== null) {
            oobSwap(oobValue,
                oobElement,
                settleInfo,
                rootNode)
        }
    } else {
        oobElement
        .removeAttribute('hx-swap-oob')

        oobElement
        .removeAttribute('data-hx-swap-oob')
    }
})
return oobElts.length > 0
}

```

A função **findAndSwapOobElements()** toma como parâmetros as variáveis **fragment**, **settleInfo** e **rootNode()**. Em seu início, é criada uma variável **oobElts** que recebe todos os valores dentro de **fragment** que possuem propriedades com o nome de 'hx-swap-oob' ou 'data-hx-swap-oob' e em seguida é criada uma iteração sobre os valores de **oobElts**, capturados na variável **oobElement** dentro de uma função anônima. Dentro da iteração, é verificado se o valor de configuração do *HTMX* **allowNestedOobSwaps** é verdadeiro e se o valor da propriedade **parentElement** de **oobElement** é nulo. Caso a expressão seja verdadeira, uma constante **oobValue** é criada com o valor de retorno de **getAttributeValue()**, e é verificado se o valor de **oobValue** é diferente de nulo, executando **oobSwap()** caso seja. Caso a estrutura condicional exterior não seja executada, o método **removeAttribute()** de **oobElement** é executado duas vezes para remover os atributos 'hx-swap-oob' e 'data-hx-swap-oob', e por fim, a função retorna o valor da comparação se o tamanho de **oobElts** é maior que zero.

```

if (swapOptions.select) {
    const newFragment = getDocument().createDocumentFragment()
    forEach(fragment.querySelectorAll(swapOptions.select), function(node) {
        newFragment.appendChild(node)
    })
    fragment = newFragment
}
handlePreservedElements(fragment)
swapWithStyle(

```

```

swapSpec.swapStyle,
swapOptions.contextElement,
target,
fragment,
settleInfo)
restorePreservedElements()

```

Ao voltar para **swap()**, há uma estrutura condicional que verifica se o valor da propriedade **select** de **swapOptions** é definido, criando uma constante **newFragment** com o valor de um fragmento *HTML* novo e executando uma iteração sobre os valores de todos os valores que possuem a configuração de **select** dentro de **fragment**, capturando os fragmentos na variável *node* que é parâmetro de uma função anônima. Dentro desta função, é executado o método **appendChild()** de **newFragment**. Após isso, **fragment** recebe o valor de **newFragment**, e são feitas as chamadas para as funções **handlePreservedElements()**, **swapWithStyle()** e **restorePreservedElements()**.

```

if (selectionInfo.elt &&
!bodyContains(selectionInfo.elt) &&
getRawAttribute(selectionInfo.elt, 'id')) {
    const newActiveElt = document
        .getElementById(getRawAttribute(selectionInfo.elt, 'id'))
    const focusOptions = {
        preventScroll: swapSpec.focusScroll !== undefined ?
        !swapSpec.focusScroll : !htmx.config.defaultFocusScroll
    }
    if (newActiveElt) {
        if (selectionInfo.start && newActiveElt.setSelectionRange) {
            try {
                newActiveElt.setSelectionRange(
                    selectionInfo.start,
                    selectionInfo.end)
            } catch (e) {}
        }
        newActiveElt.focus(focusOptions)
    }
}

```

Em seguida, é verificado se os valores da propriedade **elt** de **selectionInfo** e o valor de retorno de **getRawAttribute()** são definidos e se o valor de retorno de **bodyContains()** é falso. Caso essa expressão seja verdadeira, é criada uma constante **newActiveElt**

com o valor do elemento contido na propriedade **elt** de **selectionInfo**, e uma constante **focusOptions** é criada com um objeto que possui os campos **preventScroll** com o valor da negação de **focusScroll** de **swapSpec** caso a propriedade **focusScroll** de **swapSpec** seja diferente de indefinida ou com a negação do valor da propriedade **defaultFocusScroll** da configuração do *HTMX*. Em seguida, é verificado se o valor de **newActiveElt** é definido, verificando em seguida se o valor da propriedade **start** de **selectionInfo** e da propriedade **setSelectionRange** de **newActiveElt** são definidos. Caso a expressão anterior seja verdadeira, é criada uma estrutura de tratamento de erros. Dentro desta estrutura, é feita a chamada para o método **setSelectionRange()** de **newActiveElt**. Caso a estrutura gere algum erro, o erro é capturado em uma variável e é ignorado pelo *HTMX*. Por fim, é feita uma chamada para o método **focus()** de **newActiveElement**.

```
target.classList.remove(htmx.config.swappingClass)
forEach(settleInfo.elts, function(elt) {
    if (elt.classList) {
        elt.classList.add(htmx.config.settlingClass)
    }
    triggerEvent(elt, 'htmx:afterSwap', swapOptions.eventInfo)
})
maybeCall(swapOptions.afterSwapCallback)
```

Após isso, é feita uma chamada para o método **remove()** da propriedade **classList** de **target** e é criada uma iteração sobre os valores da propriedade **elts** de **settleInfo**, capturando os valores em uma variável **elt** dentro de uma função anônima. Nesta função anônima, é verificado se o valor da propriedade **classList** de **elt** é definida, e caso seja, é adicionado uma nova classe através do método **add()** da propriedade **classList** de **elt**. Por fim, é feita uma chamada para **triggerEvent()** e fora da iteração, uma chamada para **maybeCall()** é feita.

```
if (!swapSpec.ignoreTitle) {
    handleTitle(settleInfo.title)
}
```

Em seguida, é verificado se o valor da propriedade **ignoreTitle** de **swapSpec** é falso, e caso seja, é feita uma chamada para **handleTitle()**.

```
function handleTitle(title) {
    if (title) {
        const titleElt = find('title')
        if (titleElt) {
            titleElt.textContent = title
        }
    }
}
```

```

    } else {
      window.document.title = title
    }
  }
}

```

A função **handleTitle()** toma como parâmetro único a variável **title**. Em seu corpo, é verificado se o valor de **title** é definido, e caso seja, uma constante **titleElt** é criada com o valor do elemento de título da página *HTML*, e é verificado se este elemento é definido. Caso o valor de **titleElt** seja definido, o valor de sua propriedade **textContent** recebe o valor de **title**, caso contrário, a propriedade **title** do documento *HTML* recebe diretamente o valor de **title**.

```

const doSettle = function() {
  forEach(settleInfo.tasks, function(task) {
    task.call()
  })
  forEach(settleInfo.elts, function(elt) {
    if (elt.classList) {
      elt.classList.remove(htmx.config.settlingClass)
    }
    triggerEvent(elt, 'htmx:afterSettle', swapOptions.eventInfo)
  })

  if (swapOptions.anchor) {
    const anchorTarget = asElement(
      resolveTarget('#' + swapOptions.anchor))
    if (anchorTarget) {
      anchorTarget
        .scrollIntoView({ block: 'start', behavior: 'auto' })
    }
  }
  updateScrollState(settleInfo.elts, swapSpec)
  maybeCall(swapOptions.afterSettleCallback)
  maybeCall(settleResolve)
}

```

Após isso, é criada uma constante **doSettle** que recebe uma função anônima. Dentro desta função é criada uma iteração sobre os valores da propriedade **tasks** de **settleInfo** que são capturadas em uma variável **task** dentro de uma função anônima, e

nesta função anônima é realizada uma chamada para o método **call()** de **task**. Em seguida, uma iteração sobre os valores da propriedade **elts** de **settleInfo**, capturando o valor na variável **elt** dentro de uma função anônima. Dentro desta função é verificado se o valor da propriedade **classList** de **elt** é definido, e caso seja, é feita uma chamada para o método **remove()** da propriedade **classList** de **elt**. Em seguida, fora da estrutura condicional, é feita uma chamada para **triggerEvent()**. Logo após, é verificado se o valor da propriedade **anchor** de **swapOptions** é definido. Caso seja, uma constante **anchorTarget** é criada com o valor de retorno de **asElement()** e é verificado se o valor de **anchorTarget** é definido e caso seja é executada o método **scrollIntoView()** de **anchorTarget**. Por fim são feitas chamadas para as funções **updateScrollState()** e **maybeCall()**.

```
function updateScrollState(content, swapSpec) {
  const first = content[0]
  const last = content[content.length - 1]
  if (swapSpec.scroll) {
    var target = null
    if (swapSpec.scrollTarget) {
      target = asElement(querySelectorExt(
        first,
        swapSpec.scrollTarget))
    }
    if (swapSpec.scroll === 'top' && (first || target)) {
      target = target || first
      target.scrollTop = 0
    }
  }
}
```

A função **updateScrollState()** toma como parâmetros as variáveis **content** e **swapSpec**. No seu início são criadas as constantes **first** com o valor do conteúdo na primeira posição de **content** e **last** com o valor do conteúdo na última posição de **content**. Em seguida, é criada uma estrutura condicional que verifica se o valor da propriedade **scroll** de **swapSpec** é definida. Caso a condição seja verdadeira, é criada uma variável **target** com um valor nulo, e é criada uma estrutura condicional que verifica se o valor da propriedade **scrollTarget** de **swapSpec** é definida, definindo o valor de **target** com o valor de retorno da chamada para **asElement()** que incorpora o valor de retorno para **querySelectorExt()**. Em seguida, é verificado se o valor da propriedade **scroll** de **swapSpec** é igual à string 'top' e se **first** ou **target** são valores definidos. Caso a condição seja verdadeira, o valor de **target** é definido com o valor de **target** caso este seja definido, caso contrário o valor de **first** é utilizado, e o valor da propriedade **scrollTop** de **target** é definida como zero.

```
if (swapSpec.scroll === 'bottom' && (last || target)) {
  target = target || last
}
```

```

    target.scrollTop = target.scrollHeight
  }
  if (typeof swapSpec.scroll === 'number') {
    getWindow().setTimeout(function() {
      window.scrollTo(0, (swapSpec.scroll))
    }, 0)
  }

```

Em seguida, é verificado se a propriedade **scroll** de **swapSpec** é definida com o valor de uma **string** 'bottom' e se o valor de **last** ou de **target** são definidos. Caso a expressão seja verdadeira, o valor de **target** recebe o próprio valor de **target**, caso contrário o valor de **last** é assinalado, e o valor da propriedade **scrollTop** de **target** é definida com o valor da propriedade **scrollHeight** de **target**. Logo após, uma outra estrutura condicional verifica se o valor da propriedade **scroll** de **swapSpec** é um valor numérico, e caso seja, assinalar uma chamada para a função **setTimeout()**, definido a execução de uma função anônima que moverá a página para a posição zero depois de 0 segundos de espera.

```

if (swapSpec.show) {
  var target = null
  if (swapSpec.showTarget) {
    let targetStr = swapSpec.showTarget
    if (swapSpec.showTarget === 'window') {
      targetStr = 'body'
    }
    target = asElement(
      querySelectorExt(first, targetStr))
  }
}

```

Após isso, uma estrutura condicional verifica se o valor da propriedade **show** de **swapSpec** é verdadeiro. Então, uma variável **target** é criada com um valor nulo, e é verificado se o valor da propriedade **showTarget** de **swapSpec** é verdadeiro. Caso seja, é criado uma variável **targetStr** com o valor da propriedade **showTarget** de **swapSpec**, e uma estrutura condicional verifica se o valor de **showTarget** em **swapSpec** tem o valor de uma **string** 'window', e caso tenha, o valor de **targetStr** é definido com a **string** 'body'. Por fim, o valor de **target** é definido com o valor de retorno de **querySelectorExt()**, incorporado pela função **asElement()**.

```

if (swapSpec.show === 'top' &&
    (first || target)) {
  target = target || first
}

```

```

    target.scrollToView({
      block: 'start', behavior: htmx.config.scrollBehavior
    })
  }

  if (swapSpec.show === 'bottom' &&
    (last || target)) {
    target = target || last
    target.scrollToView({
      block: 'end', behavior: htmx.config.scrollBehavior
    })
  }

```

Por fim, existem duas estruturas condicionais similares. Em ambas, é verificado se o valor da propriedade `show` de **swapSpec** é definido com o valor de uma **string**, na primeira com o valor de 'top', na segunda com o valor de 'bottom'. Em seguida, é verificado se o valor de uma variável ou de **target** é definido, **first** na primeira estrutura, **last** na segunda. Dentro do corpo o valor de **target** é definido com seu próprio valor caso este seja definido, ou com o valor de **first** na primeira estrutura ou com de **last** na segunda estrutura. Por fim, é feita uma chamada para o método **scrollIntoView()** de **target**.

```

if (swapSpec.settleDelay > 0) {
  getWindow().setTimeout(doSettle, swapSpec.settleDelay)
} else {
  doSettle()
}

```

Ao voltar para **swap()**, é verificado se o valor da propriedade **settleDelay** de **swapSpec** é maior a zero e caso seja, o **script** força o navegador a esperar o tempo definido na propriedade **settleDelay** de **swapSpec** enquanto espera para executar **doSettle()**. Caso a propriedade **settleDelay** de **swapSpec** não seja definida, a função **doSettle()** é executada.

```

let shouldTransition = htmx.config.globalViewTransitions
if (swapSpec.hasOwnProperty('transition')) {
  shouldTransition = swapSpec.transition
}

```

Em seguida uma variável **shouldTransition** é criada com o valor da propriedade **globalViewTransitions** da configuração do *HTMX*, e é verificado se o valor de retorno do método **hasOwnProperty()** de **swapSpec** é verdadeira, e caso seja o valor de **shouldTransition** é definido com o valor da propriedade **transition** de **swapSpec**.

```

if (shouldTransition &&
triggerEvent(
    elt,
    'htmx:beforeTransition',
    swapOptions.eventInfo
) &&
typeof Promise !== 'undefined' &&
document.startViewTransition) {
    const settlePromise = new Promise(
        function(_resolve, _reject
    ) {
        settleResolve = _resolve
        settleReject = _reject
    })
    const innerDoSwap = doSwap
    doSwap = function() {
        document.startViewTransition(function() {
            innerDoSwap()
            return settlePromise
        })
    }
}

```

Logo após, é verificado se o valor da propriedade **shouldTransition** é verdadeiro, se o valor de retorno de **triggerEvent()** é definido, se o tipo de **Promise** é definido e se o valor da propriedade **startViewTransition()** de **document** é definido. Caso essa expressão seja verdadeira, uma constante **settlePromise** é criada com o valor de uma nova instância de **Promise** que recebe uma função anônima como parâmetro no **contrutor**, e tem como parâmetros uma variável **_resolve** e uma variável **_reject**. No corpo da função anônima, o valor de **settleResolve** é definido com o valor de **_resolve** e o valor de **settleReject** é definido com o valor de **_reject**. Em seguida, é criada uma constante **innerDoSwap** com o valor da função **doSwap()** ao invés do valor de retorno da função, e o valor de **doSwap** recebe uma nova função que em seu corpo define o valor da propriedade **startViewTransition** de **document** com o valor de uma função anônima que chama **innerDoSwap()** e retorna o valor de **settlePromise**.

```

try {
    if (swapSpec?.swapDelay && swapSpec.swapDelay > 0) {
        getWindow().setTimeout(doSwap, swapSpec.swapDelay)
    } else {

```

```

        doSwap()
    }
} catch (e) {
    triggerErrorEvent(elt, 'htmx:swapError', swapOptions.eventInfo)
    maybeCall(settleReject)
    throw e
}

```

Ao final de **swap()**, é criada uma estrutura de tratamento de erros, onde é verificado se o valor da propriedade de **swapDelay** dentro da promessa de **swapSpec** é definido e se o valor de **swapDelay** é maior que zero. Caso essa condição seja verdadeira, é feita uma chamada para o método **setTimeout()** do documento *HTML* que executa **doSwap()** com a duração definida pela propriedade **swapDelay** de **swapSpec**. Caso a expressão anterior seja falsa, é feita apenas uma chamada para **doSwap()**. Por fim, se algum erro for levantado durante a execução deste último trecho, o erro é capturado dentro de uma variável **e**, e são feitas chamadas para **triggerErrorEvent()**, **maybeCall()** utilizando o valor de **settleReject** e lançando o erro **e**.

B.4 Um resumo do funcionamento do HTMX

Para solidificar o entendimento do HTMX, neste capítulo será apresentado um resumo do que a biblioteca faz, passo por passo, de acordo com o que foi apresentado nos últimos capítulos.

B.4.1 Inicialização

A partir da inicialização de uma página HTML com a biblioteca HTMX, o script começará suas atividades preparando um ambiente simples para a execução de requisições AJAX e inicializando todos os valores necessários. Em seguida, irá procurar todos os elementos com atributos de ações HTTP, tais como 'hx-get', 'hx-post', 'hx-put', 'hx-patch' e 'hx-delete', e irá atribuir escutas de eventos para as ações correspondentes ao atributo 'hx-trigger' definido no elemento, ou para o evento padrão do elemento caso este atributo seja omitido pelo desenvolvedor.

B.4.2 Intercepção de eventos

Quando algum evento que foi sinalizado a partir de um elemento com atributos de ação HTTP é disparado, o HTMX intercepta este evento e começa a análise e tratamento do que as ações do elemento indicam, tais como a rota para a qual o elemento está apontando, o método HTTP que o elemento está utilizando, qual o método desejado de troca de elementos no corpo do HTML e qual o alvo para a troca de elementos no documento.

B.4.3 Requisição

Em seguida, o HTMX realiza a requisição HTTP de acordo com o que foi indicado pelo elemento HTML, e caso a requisição resulte em algum erro, o HTMX tratará do erro, lançando as informações necessárias na saída padrão de erros.

B.4.4 Tratamento da resposta

A partir da aquisição da resposta pelo servidor, considerando que a resposta não seja um corpo vazio, o HTMX irá verificar se o elemento que iniciou a requisição possui um atributo 'hx-target'. Caso o elemento possua este atributo, o HTMX já terá processado o alvo da requisição previamente, então ele irá para o passo de troca de elementos, mas caso não possua, é implicitamente inferido que o elemento a ser substituído é o próprio elemento que iniciou a requisição. Em seguida o HTMX irá verificar se há um atributo 'hx-swap', que indica a estratégia de troca de elementos. Caso não haja, a estratégia padrão para troca é a inserção da resposta em HTML dentro do elemento que iniciou o processo.

B.4.5 A troca de elementos

Assim que o tratamento da resposta é concluído, o HTMX começa o processo de troca de elementos, a partir do que é indicado no atributo 'hx-swap'. As estratégias possíveis para troca de elementos no HTMX são:

- HTML interno: A resposta é inserida dentro do elemento HTML responsável pela chamada.
- HTML externo: A resposta substitui o elemento HTML responsável pela chamada.
- Antes do começo: A resposta é adicionada antes do elemento HTML responsável pela chamada.
- Depois do começo: A resposta é adicionada antes do primeiro nóculo filho dentro do elemento HTML responsável pela chamada.
- Antes do fim: A resposta é adicionada após o último nóculo filho dentro do elemento HTML responsável pela chamada.
- Depois do fim: A resposta é adicionado após o elemento HTML responsável pela chamada.
- Excluir: A resposta sempre exclui o elemento HTML responsável pela chamada, independentemente da existência de conteúdo da resposta.
- Nenhum: A resposta não substitui nenhum valor, mas ainda processa trocas fora dos limites.

B.4.6 Trocas fora dos limites

O HTMX possui uma técnica que permite a substituição de elementos que estão fora dos limites do elemento que iniciou o processo de troca. Ao utilizar o atributo 'hx-oob-

swap' com um valor verdadeiro, o elemento com um atributo 'id' procura outro elemento com o exato mesmo valor de identificador, e substituí esse elemento automaticamente.

B.4.7 Conclusão do processamento

Por fim, ao receber os novos elementos, o HTMX limpa o seu cache interno de requisições, remove o campo que é utilizado para o recebimento da requisição antes da troca de elementos e se prepara para receber novas requisições futuramente.