Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

O papel da documentação no desenvolvimento de softwares *open source*: Uma análise e um estudo de caso

Victor Hugo Miranda Pinto

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Orientador: Prof. Dr. Alfredo Goldman vel Lejbman

São Paulo
Janeiro de 2021

Resumo

Victor Hugo Miranda Pinto. **O papel da documentação no desenvolvimento de softwares *open source*: Uma análise e um estudo de caso**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2021.

A documentação de um software é valiosa tanto para a sua manutenção quanto evolução. Ela é vista como a principal forma de comunicação de informações sobre um software. Neste trabalho, buscamos entender melhor como a documentação é percebida por desenvolvedores, quais os problemas mais comuns encontrados, e como projetos *open source* de sucesso lidam com suas documentações. Após um levantamento bibliográfico sobre o tema e estudo de alguns dos projetos *open source* mais proeminentes do GitHub, aplicamos o que aprendemos em um projeto de software real. Implementamos um sistema para apoiar a documentação do VTEX IO, incluindo um novo portal de documentação. Junto à implementação deste sistema, foram implementadas mudanças culturais com um time dentro do VTEX IO. Tais mudanças tinham como objetivo trazer a documentação para o fluxo de desenvolvimento, elevando sua importância.

Palavras-chave: Documentação. Desenvolvimento de Software. *Open source.*

Abstract

Victor Hugo Miranda Pinto. **O papel da documentação no desenvolvimento de softwares *open source*: Uma análise e um estudo de caso**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2021.

A software's documentation is a valuable resource both for its maintenance and evolution. It is often recognized as the main way of communicating information regarding a piece of software. In this work, we studied how is documentation perceived by software developers, what are the most common documentation issues they face and how big open source projects deal with their documentation. After a bibliographic study to answer those questions, we applied what we've learned in a real production open source software. We've implemented a software system to support the documentation for VTEX IO, including a new documentation portal. Along with this software system, we've also implemented cultural changes along with a team inside of VTEX IO. These changes were introduced to bring documentation to the development process, elevating its importance.

Keywords: Documentation. Software Development. *Open Source.*

Sumário

Capítulo 1

Introdução

1.1 Motivação

Documentação, e sua qualidade, são elementos relevante tanto para evolução quanto para utilização de qualquer software, segundo Garousi *et al.*, 2013. Desenvolvedores consultam diariamente documentação de softwares que utilizam, sejam eles *frameworks*, manuais de referência da linguagem na qual estão programando, ou do software que estão desenvolvendo.

Forward e Lethbridge, 2002 e Lethbridge *et al.*, 2003 mostraram que, apesar dessa importância da documentação, ela em geral não é atualizada com a frequência necessária. Há uma certa assincronia entre mudanças no código e em sua documentação, com o potencial de afetar negativamente a produtividade e a qualidade do trabalho de desenvolvedores que a consomem.

Há uma percepção na comunidade de que o trabalho em melhoras de documentação é custoso e não traz tanto benefício quanto a implementação de novas funcionalidades ou resolução de *bugs*, por isso elas não são feitas frequentemente, Plosch *et al.*, 2014. De maneira geral, documentação é vista como algo que sempre pode ser feito depois do desenvolvimento. O problema está no fato de que em muitos projetos o "depois" acaba se transformando em "nunca", em geral associando a falta de documentação com a falta de tempo para escrevê-la, como mostrado por Forward e Lethbridge, 2002.

Neste trabalho vamos investigar a relevância da documentação de software — em particular a documentação técnica, voltada para desenvolvedores — para a sua utilização e manutenção, por meio de uma revisão da literatura sobre o assunto. Em seguida, vamos estudar a documentação de projetos de software ***open source***, que vêm se tornando cada vez mais relevantes na indústria e formando comunidades de usuários cada vez maiores, Anthes, 2016. Por sua natureza de código aberto, este tipo de projeto nos permite uma investigação mais detalhada de suas soluções e processos.

Vamos também apresentar uma experiência prática que tivemos aplicando o que descobrimos em um projeto *open source* mantido por uma empresa de tecnologia brasileira. Durante essa experiência prática, implementamos um sistema para suportar a documenta-

ção desse projeto, e também acabamos encontrando a necessidade de mudanças culturais nos times envolvidos.

1.2 Objetivos

Os principais objetivos desse trabalho são os seguintes:

1. Investigar — através de um levantamento bibliográfico — o estado atual da documentação técnica de softwares e como ela é utilizada por desenvolvedores.
2. Analisar como é mantida a documentação de alguns projetos *open source* de sucesso.
3. Aplicar em um projeto *open source* real, no contexto da indústria de *software*, o que aprendemos para melhorar sua documentação.

1.3 Organização dos capítulos

O próximo capítulo apresenta o objeto de interesse do trabalho, a documentação de software voltada para desenvolvedores. Serão apresentados resultados de uma revisão da literatura disponível sobre o assunto. Vamos discutir sobre a sua importância e relevância e quais os principais pontos de resistência encontrados para sua criação e manutenção.

No terceiro capítulo, é apresentada uma breve definição sobre o desenvolvimento de projetos *open source* e algumas de suas características que os diferem de outros tipos de *software*. Em seguida, apresentamos resultados de uma pesquisa sobre como alguns dos maiores projetos *open source* — segundo o relatório anual do GitHub, *The State of the OCTOVERSE*, disponível em https://octoverse.github.com/ — produzem e mantêm suas documentações.

Em seguida, no quarto capítulo, apresentamos nossa experiência prática, aplicando nossos aprendizados em um software *open source* real. Esse capítulo descreve a implementação de um sistema para apoiar a documentação do projeto.

No quinto capítulo descrevemos as mudanças culturais que acabamos introduzindo no time responsável pelo software em que trabalhamos, e o que aprendemos com elas. Por fim, o sexto capítulo apresenta as conclusões do trabalho.

Capítulo 2

Documentação e sua importância

Na literatura sobre o assunto, encontramos diversas definições para documentação de software, variando em sua abrangência, qualidade esperada e público-alvo. Para este trabalho, vamos adotar uma definição mais próxima a utilizada por Plosch *et al.*, 2014, onde documentação de software irá se referir ao conjunto de documentos criado com o objetivo de comunicar informações técnicas sobre o software a qual pertence. Com esta definição, podemos limitar o nosso escopo a documentação produzida para desenvolvedores. Ao mesmo tempo, esta definição nos permite tratar de diferentes tipos de documentação, utilizadas em diferentes momentos ao longo do ciclo de vida de um software.

Neste capítulo vamos discutir a aparente contradição entre como profissionais da área de desenvolvimento de software percebem e reconhecem a importância da documentação, e como contribuem para sua manutenção e produção. Vamos também pontuar as principais dificuldades encontradas em ambos os processos referentes à documentação, buscando entender quais são os desafios para se reverter essa situação.

2.1 Tipos de documentação de software

Qualquer projeto de software, independente do seu domínio de aplicação, gera algum tipo de documentação associada, seja ela organizada e intencional ou não. Em particular, projetos de média e grande escala podem possuir grandes quantidades de documentação associadas a eles. Segundo Sommerville *et al.*, 2005, uma série de requisitos está diretamente associada com a documentação de um projeto de software:

- A documentação deveria servir como meio de comunicação entre membros do time de desenvolvimento;
- Ser um repositório de informação que seria utilizada para manutenção do software;
- Ser uma fonte de informação para gerenciamento do projeto, apoiando seu planejamento;

- Alguma parte dessa documentação deveria orientar usuários do sistema em sua utilização e organização.

Para cumprir com todos estes requisitos, diferentes tipos de documentação são necessárias. Desde rascunhos em quadros brancos, feitos durante uma discussão técnica, até documentos profissionais que são considerados parte integrante do produto de software desenvolvido.

Entre os diferentes tipos de documentação produzidos, podemos fazer mais uma distinção:

1. Documentação de processo. Consiste em documentos que descrevem o processo de desenvolvimento e manutenção, incluindo planejamento, requisitos, padrões, etc. Esta documentação é produzida com o intuito de orientar o gerenciamento do desenvolvimento de um sistema.

2. Documentação do produto. Consiste em documentos que descrevem o software sendo desenvolvido. Normalmente inclui documentação **interna**, que descreve o software do ponto de vista de seus desenvolvedores, e documentação **externa**, voltada para o usuário final.

No escopo deste trabalho, vamos focar na documentação do produto, tanto interna quanto voltada para o usuário final. Como vamos também focar em softwares **para desenvolvedores**, há uma grande intersecção entre esses dois tipos de documentação, já que o seu público-alvo é muito similar.

2.2 Percepção de importância e relevância

A documentação de um software é muitas vezes vista como a "porta de entrada" para seus usuários. Podendo até mesmo ser o primeiro contato de alguém com tal software, o que se torna especialmente relevante no contexto atual em que uma busca pelo nome do software em um navegador é uma prática cada vez mais comum.

Como desenvolvedores, e usuários de softwares criados por outros desenvolvedores, frequentemente nos apoiamos na documentação desses softwares para utilizá-los de maneira relevante. Um software que não possui documentação, ou possui uma documentação considerada ruim, tende a nos direcionar para outros caminhos.

A documentação é vista como a principal forma de comunicação de informações sobre um software, mesmo quando desatualizada ou inconsistente, segundo Lethbridge *et al.*, 2003. Documentação pode ser consumida sob demanda, é de fácil distribuição quando se encontra em meio digital—uma prática que vem se tornando cada vez mais o padrão da indústria—e tende a não depender do conhecimento de um único indivíduo, servindo como poderoso mecanismo de difusão de conhecimento previamente acumulado.

Plosch *et al.*, 2014 e Lethbridge *et al.*, 2003 mostram que apesar dos problemas encontrados em documentações de software — dos quais vamos tratar nas próximas sessões — engenheiros de software acreditam no valor que esses artefatos possuem, e os consideram importantes tanto nos processos de manutenção quanto na criação de novas funcionalidades em sistemas existentes. Outra percepção identificada nos estudos é que

muitos engenheiros tendem a superestimar a quantidade de tempo que eles passam de fato utilizando documentação, o que nos leva a acreditar mais ainda na importância que eles atribuem à documentação.

2.3 Uso e manutenção na prática

Na prática, o uso da documentação varia de acordo com o seu conteúdo, o quanto os desenvolvedores confiam nesse conteúdo, e o estágio do ciclo de vida em o tal software se encontra. Em geral, a documentação utilizada durante o processos de manutenção é diferente daquela utilizada durante o desenvolvimento e criação.

Em Garousi *et al.*, 2013, são apresentados resultados obtidos em um estudo de caso onde podemos observar a diferença na maneira com que os diversos tipos de documentação são utilizadas em diferentes momentos. Durante o processo de desenvolvimento, documentações mais distantes do código, como análises de requisitos, ou descrições da arquitetura do software são as mais utilizadas, principalmente por desenvolvedores iniciantes. No processo de manutenção, foi observada uma diferença estatisticamente significativa na frequência de uso da documentação, que se mostrou menos utilizada frente ao próprio código-fonte.

Em Lethbridge *et al.*, 2003 vemos que, de maneira geral, documentações que são percebidas como as mais corretas e mais confiáveis, e portanto as mais consultadas, são as que estão mais próximas do código-fonte. Documentações referentes a testes e qualidade ou detalhes de implementação de certas funcionalidades, são vistas como mais confiáveis e são as mais consultadas. O inverso é observado para documentações mais distantes da implementação do software, como análises de requisitos, documentos de especificações, etc. Essa diferença na percepção destes diferentes tipos de documentação podem justificar os resultados citados no parágrafo anterior. Como documentações mais distantes do código-fonte são consideradas menos "confiáveis", elas podem ser vistas como obstáculos nas tarefas de manutenção ao invés de recursos valiosos. Desenvolvedores realizando a manutenção de um software iriam direto ao código-fonte pois ele não "mente", mesmo com a dificuldade extra que o processo de entender o código de outro desenvolvedor pode apresentar. Os desenvolvedores que responderam às perguntas dos autores classificaram como "raramente atualizadas" todos os tipos de documentação, com exceção da referente a processos de teste e controle de qualidade, que seriam atualizados "alguns dias" após mudanças no código-fonte.

Apesar de ser vista como um meio importante de transmitir informações sobre um software, a documentação apresenta diversas falhas, aparentando ser insuficiente para acompanhar a velocidade com que o próprio software evolui. Em trabalhos como Satish e Anand, 2016 e Aghajani *et al.*, 2019, temos um resumo dos problemas que foram levantados na literatura. Esse trabalho nos permite ter uma visão clara dos principais problemas que usuários de documentações técnicas enfrentam na prática. Os problemas que mais foram apontados por esses usuários — em grande parte desenvolvedores — foram os seguintes:

- Documentação desatualizada;

- Falta de uma ligação clara entre a documentação produzida e as mudanças no código que a tornaram necessárias;
- Falta de um padrão na estrutura da documentação, o que torna o processo de atualização muito custoso;
- Engenheiros de software demonstraram pouco entusiasmo com tarefas relacionadas a documentação;
- Falta de boas ferramentas para a manutenção e monitoramento de alterações.

Sobre o processo de manutenção na prática, não temos muitos estudos que o investigam, podendo essa ser uma causa para os problemas observados na documentação resultante. Como é pontuado por Dagenais e Robillard, 2010, sabemos da existência de documentações muito grandes (mais de 200.00 palavras, por exemplo) que sustentam usuários de softwares amplamente utilizados pela indústria, mas não sabemos muito sobre o processo de criação e manutenção dessa documentação. Podermos inferir que esse é um processo custoso e que diversas decisões tiveram que ser tomadas durante a sua criação. Os processos adotados por diferentes projetos, com diferentes tamanhos e times possuem requisitos e processos de manutenção de documentação diferentes entre si. No capítulo seguinte, vamos investigar como grandes projetos *open source* lidam com esse problema em mais detalhes.

Com base nos resultados comentados até aqui podemos afirmar que, na prática, a documentação de softwares de diversos tamanhos deixa a desejar em diversos atributos considerados importantes pelos seus usuários. Engenheiros de software tendem a enxergar a produção e futura manutenção de documentação como tarefas que agregam pouco valor e de alto custo, como citado em Plosch *et al.*, 2014. Essa percepção negativa pode se tornar a causa de grande parte dos problemas encontrados pelos próprios desenvolvedores quando precisam de documentação. Schreck *et al.*, 2007, apresenta resultados que podem reforçar essa ideia, como o fato de que muitas vezes documentações ruins não são resultados de processos ruins de documentação, mas sim o fato de muitos desenvolvedores não aderirem a tais processos.

2.4 Dificuldades na manutenção e produção de documentação

Diante dos problemas levantados na sessão anterior quanto à qualidade e utilização da documentação de software na prática, podemos ter a impressão de que a razão desses problemas é simplesmente devido à indisposição dos desenvolvedores em criar documentação. Não acreditamos que atribuir toda a culpa da deficiência na documentação aos desenvolvedores nos ajuda a chegar numa solução para os problemas. Vamos buscar entender quais as principais barreiras encontradas pelos desenvolvedores nos processos de criação e manutenção.

Em Parnas, 2011, o autor busca entender a deficiência na documentação de software e propor um modelo de como deveríamos documentar. Um primeiro argumento levantado pelo autor é o de que a falta de motivação por parte dos desenvolvedores em escrever

documentação faz com que ela seja escrita por outras pessoas, que podem ou não ter o conhecimento técnico necessário para compreender totalmente o que estão documentando. Essa terceirização pode gerar documentos incorretos ou incompletos, o que reduz a qualidade da documentação produzida. Problemas de qualidade na documentação alimentam um ciclo de feedback negativo:

1. A redução na qualidade dos documentos desencoraja o seu uso.
2. Uma redução no uso tende a reduzir ainda mais a motivação na criação de mais documentação, e os recursos investidos naturalmente diminuem.
3. Por fim, com a diminuição de motivação e recursos direcionados a documentação, a sua qualidade tende a cair ainda mais.

Outros resultados apresentados na mesma pesquisa revelam um problema mais profundo na maneira com que os praticantes do desenvolvimento de software enxergam a documentação. Alguns chegam a acreditar que o código-fonte é a sua própria documentação. É fácil ver como essa ideia não se sustenta com qualquer sistema de software grande o suficiente para ser dividido em diversos arquivos e possuir relações complexas de dependências.

Resultados obtidos por Forward e Lethbridge, 2002 mostram que mesmo em softwares grandes com grande volume de documentação, os usuários têm dificuldade de navegar pelo conteúdo e entender sua estrutura, oferecendo uma resistência à sua consulta e até a produção de novos conteúdos. Enquanto projetos de pequena ou média escala sofrem com a quantidade insuficiente de documentação, apesar do fato de que os engenheiros que trabalham no software reconhecerem a importância de se criar documentação, mas dizem não ter tempo para escrevê-la.

Com base nos resultados citados até então, fica clara como a manutenção de documentação se torna cada vez mais custosa ao longo do ciclo de vida de um software. Isto nos leva a acreditar que o planejamento e organização prévia de documentação, durante o período inicial de desenvolvimento de um software é crucial para determinar se ele será bem ou mal documentado. Também acreditamos que soluções *ad hoc* para criação de documentação tendem a gerar desordem e aumentar a barreira para contribuição por parte dos desenvolvedores.

Em Dagenais e Robillard, 2010, podemos ver a importância de algumas decisões-chave que são tomadas ao longo do processo de criação e manutenção da documentação em softwares. Em especial, as decisões de em qual tipo de plataforma hospedar a documentação — a **infraestrutura** da documentação — e quais ferramentas seriam utilizadas para escrevê-la foram identificadas como as mais impactantes no futuro processo de manutenção. Portanto, essas decisões devem ser tomadas com cuidado, sendo que podem causar grandes dificuldades no futuro. Um exemplo comum citado pelos autores é a escolha entre utilizar uma *Wiki*, tentando se alavancar da possibilidade da comunidade de usuários criar a própria documentação, ou uma plataforma que dá mais controle aos mantenedores, mas tira o poder da comunidade. A primeira opção, sendo uma escolha atraente principalmente para times pequenos, já que terceiriza a responsabilidade de criação de conteúdo para a comunidade, pode trazer custos altos de manutenção e revisão do conteúdo sendo produzido com o crescimento dessa comunidade. Já a segunda opção oferece mais controle sobre o conteúdo

por parte dos desenvolvedores e mantenedores do software, mas potencialmente diminui a contribuição por parte da comunidade de usuários, colocando mais pressão no time do projeto em criar sua documentação.

Vale ressaltar a conclusão apresentada por Satish e Anand, 2016 de que, projetos de software diferentes podem possuir necessidades muito diferentes quanto a sua documentação. Não existe um *único* processo de documentação — como o descrito rigorosamente em livros de engenharia de software, ou a abordagem mais "leve" para documentação proposta pelo manifesto ágil (Fowler *et al.*, 2001) — que irá funcionar para qualquer software. Portanto, não vamos buscar um processo único que resolva os problemas apresentados até o momento, mas sim propor mudanças na maneira com que times de desenvolvimento pensam sobre documentação.

Apesar de processos de documentação não funcionarem em todos os projetos devido a suas necessidades específicas, os resultados que discutimos até o momento indicam problemas relacionados a esses processos. Processos ruins, ou muito custosos tendem a se apresentar como barreiras para os desenvolvedores, que naturalmente associam o processo de documentação com algo caro e menos atrativo do que continuar desenvolvendo. Isso nos leva a acreditar na importância da criação de ferramentas de software que sejam capazes de facilitar tais processos, tornando-os eficientes e fáceis de serem seguidos. Vamos explorar mais as possibilidades e as ferramentas sendo utilizadas por softwares *open source* para resolver os problemas apresentados neste capítulo a seguir.

Capítulo 3

Documentação em software *open source*

Software *open source* vem ganhando cada vez mais espaço na indústria, criando comunidades cada vez maiores de desenvolvedores e usuários. Artigos como Anthes, 2016 deixam clara a importância desse tipo de software, e até afirmam que o modelo *open source* "ganhou".

A ideia de distribuição de software grátis moderna, que deu origem a o que hoje chamamos de *open source*, ganhou força quando Richard Stallman, em 1974, escreveu e disponibilizou para o mundo o editor de texto *open source* Emacs. Em 1983, Stallman criou o projeto GNU para desenvolver e distribuir um sistema operacional baseado no Unix e alguns programas utilitários, ainda muito utilizados hoje. A partir de 1983, o número de usuários e projetos *open source* sendo criados e mantidos cresce consistentemente ano após ano.

Na indústria, o uso de softwares *open source* tem crescido especialmente durante a década passada. Com empresas de software cada vez mais utilizando projetos *open source* desenvolvidos por desenvolvedores externos, ou mesmo criando e mantendo públicos seus próprios projetos *open source*. Alguns exemplos que podemos citar de empresas gigantes que colaboram e mantém seus próprios projetos *open source*, utilizados por milhões de desenvolvedores, são a Microsoft, com o TypeScript, e a Google, com o Tensor Flow.

No último relatório anual publicado pelo GitHub — principal plataforma de hospedagem de software *open source* no mundo — disponível em The State of the Octoverse vemos que durante o período entre 1 de outubro de 2018 e 30 de setembro de 2019, impressionantes 10 milhões de novos desenvolvedores se cadastraram na plataforma, contribuindo para mais de 44 milhões de projetos. Esses números podem nos dar uma ideia de como a comunidade *open source* mundial é grande e está crescendo rápido.

Dada a relevância cada vez maior de software *open source* na indústria e na comunidade de desenvolvedores, vamos investigar como esses projetos se organizam, como coordenam comunidades de desenvolvedores em diversos locais do mundo e por fim como realizam a manutenção e criação da sua documentação.

3.1 Organização

Projetos *open source* possuem características singulares quanto a sua organização. Principalmente porque esses projetos precisam lidar com *comunidades* de desenvolvedores, em contraste com o modelo tradicional de apenas um time trabalhando em um mesmo software. Aberdour, 2007, cita algumas diferenças entre os processos de controle e certificação de qualidade em softwares *open source* comparados com softwares considerados "tradicionais", ou seja, proprietários e fechados. A impressão tirada pelo autor é de que, mesmo desviando consideravelmente do modelo tradicional de desenvolvimento de software, projetos *open source* conseguem crescer sem comprometer sua qualidade. Algumas práticas ainda são observadas nesse modelo de desenvolvimento, como gerenciamento centralizado, responsabilidades bem definidas quanto ao código produzido, distribuição de tarefas a serem feitas, planejamento estratégico, entre outros. Porém, a execução de cada uma delas é diferente.

Segundo Berglund e Priestley, 2001, Aberdour, 2007, Bayati, 2018 entre outros estudos sobre software *open source*, é clara a necessidade de uma comunidade sustentável e engajada para o sucesso e longevidade desses projetos. A busca por tal comunidade deve sempre ser uma preocupação do principal time (ou às vezes do) responsável.

Um modelo organizacional para um comunidade sustentável de software *open source* proposto por Aberdour, 2007 é muito utilizado nos dias de hoje, principalmente por projetos *open source* que contam com o apoio de empresas. Esse modelo é chamado de *The onion model*, e sua base é a seguinte: uma comunidade sustentável de desenvolvedores é constituída por um pequeno time *core* de desenvolvedores e números crescentes de desenvolvedores contribuidores, apontadores de *bugs* no software, e usuários. Essa comunidade seria sustentada por indivíduos buscando progressivamente aumentar seu envolvimento, se movendo na direção do time *core*. Esse modelo é observado em projetos *open source* de grande impacto hoje, como: a biblioteca para desenvolvimento de interfaces React; a linguagem de programação TypeScript, o editor de texto Visual Studio Code, entre outros.

Esse modelo organizacional é relevante para nossa pesquisa porque nos dá uma base para entender a divisão de responsabilidades em comunidades *open source*. Sob os olhos desse modelo, um time pequeno de desenvolvedores, o chamado *core*, é responsável por quase toda a produção de código e sua manutenção. Além de também assumir tarefas administrativas do projeto, como sua direção estratégica e planejamento de auto nível. Esse time também é naturalmente o grupo de pessoas que possui mais contexto sobre o projeto. Dito isto, entendemos que o time *core* é o principal time responsável pela produção e manutenção da documentação de um software *open source*. Sendo também encarregado de tomar decisões como as citadas em Dagenais e Robillard, 2010 e no capítulo anterior quanto a estrutura e processos dessa documentação.

3.2 Práticas de documentação

Vamos agora estudar exemplos reais de como alguns dos maiores projetos *open source* do mundo lidam com sua documentação. Os projetos a seguir foram escolhidos com base

na última edição do relatório The State of the Octoverse, pelo GitHub. Esses projetos foram os listados como *Top and trending projects*, para o período o período entre 1 de outubro de 2018 e 30 de setembro de 2019, e serão os analisados nesta sessão (com exceção do projeto firstcontributions/first-contributions, que não é um projeto de software, e está além do escopo da pesquisa):

	Projeto open source	Contribuidores
1	microsoft/vscode	19.1 mil
2	MicrosoftDocs/azure-docs	14 mil
3	flutter/flutter	13 mil
4	firstcontributions/first-contributions	11.6 mil
5	tensorflow/tensorflow	9.9 mil
6	facebook/react-native	9.1 mil
7	kubernetes/kubernetes	6.9 mil
8	DefinitelyTyped/DefinitelyTyped	6.9 mil
9	ansible/ansible	6.8 mil
10	home-assistant/home-assistant	6.3 mil

3.2.1 Visual Studio Code (microsoft/vscode)

Visual Studio Code é um editor de texto muito popular no momento entre desenvolvedores, principalmente para o desenvolvimento de aplicações para a Web. Foi criado e é mantido pela Microsoft, enquanto seu código-fonte é aberto e disponível no GitHub, sob a licença MIT.

Informações sobre o editor e instruções para sua utilização podem ser encontrados em seu site oficial e sua em central de documentação.

Ao entrar na documentação do projeto, vemos uma primeira sessão de *overview*, onde podemos encontrar *links* para um *Getting Started*, uma galeria das extensões mais populares para o editor de texto, uma sessão de primeiros passos que contém uma série de *links* úteis para tutoriais (artigos) sobre as funcionalidades básicas do editor para ajudar um usuário novo, e mais alguns *links* para *download* e informações relacionadas à privacidade.

As primeiras considerações sobre a interface que podem ser feitas com base na navegação do site são as seguintes: o uso de um processo de *build* que gera páginas estáticas de conteúdo, o que explica a velocidade no carregamento e navegação pelo site; uso de *Server-Side Rendering* (SSR), que pode ser verificado com a análise das requisições HTTP sendo feitas ao carregar a página, que já chega pronta para o navegador, ou seja, como HTML; uso de *React*, evidenciado pela estrutura HTML da página e por *React Developer Tools*.

O principal ponto de destaque na documentação do VSCode é visível quando navegamos pela barra lateral para qualquer artigo, como por exemplo Setup Overview:

Um botão no canto direito-superior, ao lado do título do artigo que diz *Edit*, é um *link* para um arquivo de texto escrito com *Markdown*, em um repositório *open source*, também no GitHub. Isso significa que os artigos visíveis na documentação do Visual Studio Code são abertos e colaborativos. Com isso, o time responsável por documentar o editor de texto

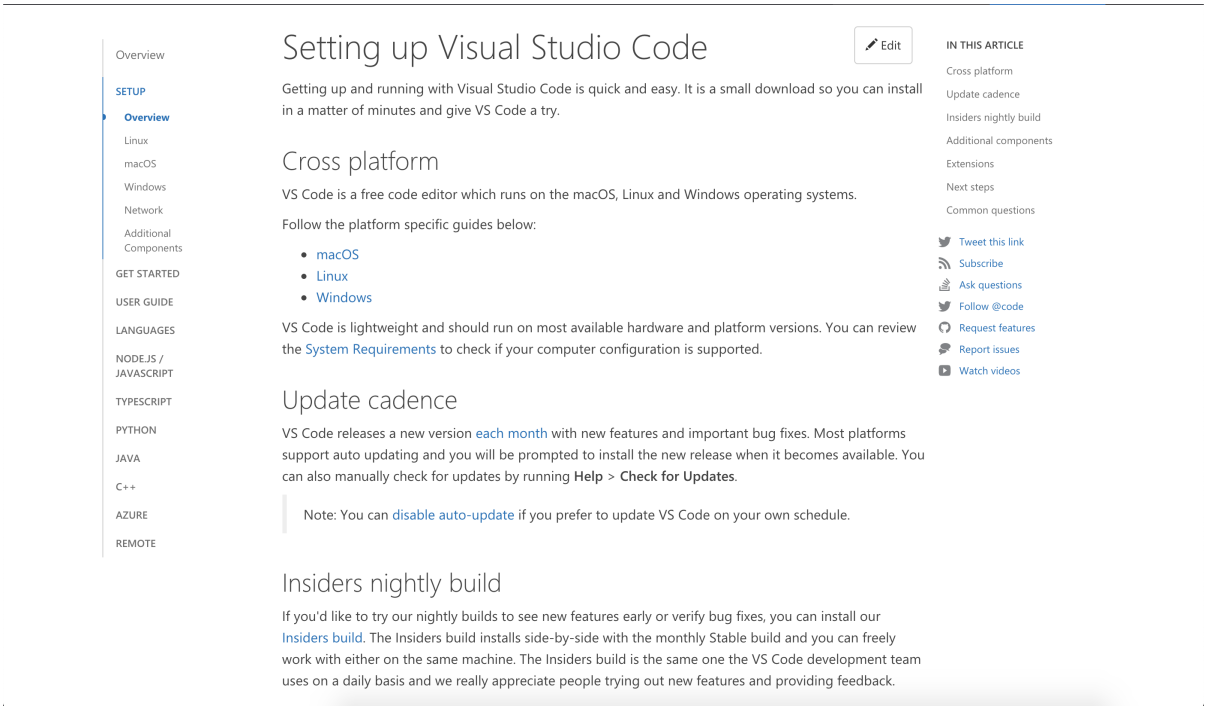

Figura 3.1: *Artigo na documentação do Visual Studio Code*

pode se alavancar do poder da comunidade *open source* para manter essa documentação atualizada e revisada constantemente. Vamos analisar mais de perto esse repositório de documentação em https://github.com/Microsoft/vscode-docs/.

Explorando o repositório de documentação do VS Code, vemos uma estrutura interessante de diretórios, onde a localização dos artigos na árvore de diretórios desse repositório é equivalente à localização de tal artigo em VS Code Docs. Isso indica que durante o processo de *build* do site essa estrutura se mantém para gerar o HTML das páginas. Também significa que, como o site é estaticamente gerado, cada modificação no conteúdo desse repositório requer um novo *build* do site, que processa o *Markdown* e as imagens para cada artigo. No mesmo repositório também estão os artigos de *release notes* do produto, que também devem ser processados pelo processo de *build* estático. Outra característica que vale ver destacada é o uso de *metadata* nos arquivos *Markdown*, que provém informações relevantes sobre cada um deles e são utilizadas no processo de *build* para determinar o títulos das páginas HTML dos respectivos arquivos, onde esse arquivo se encaixa no índice de navegação do site, entre outras.

Infelizmente não podemos investigar o processo de *build* da documentação do projeto em mais detalhes, já que o código da interface e os *scripts* responsáveis pelo *build* do site estão em um repositório privado.

3.2.2 Azure Docs (MicrosoftDocs/azure-docs)

O repositório Azure Docs é análogo ao que vimos anteriormente para o Visual Studio Code, mas para a documentação da plataforma Azure, também da Microsoft. O ponto mais interessante é como esse repositório se alavanca do poder da documentação *open*

source, sendo o segundo com mais contribuidores do GitHub. Sua estrutura é análoga à do *Microsoft/vscode-docs*, porém com alguns arquivos particulares que indicam um processo de *build* diferente do visto anteriormente.

Não temos acesso à muita informação sobre o processo de *build* que transforma os arquivos *markdown* e suas imagens em páginas HTML, pois novamente o repositório que contém essa informação é privado. Apesar disso, encontramos um arquivo chamado docfx.json, que contém informações que devem ser utilizadas pelo processo de *build* nos dá informações sobre a *engine* utilizada para processar o *markdown* vindo do repositório, a *markdig*, uma *engine* feita para projetos *.NET*. Quanto à maneira com que a documentação é produzida, mantida e consumida, esse projeto e o Visual Studio Code seguem as mesmas práticas.

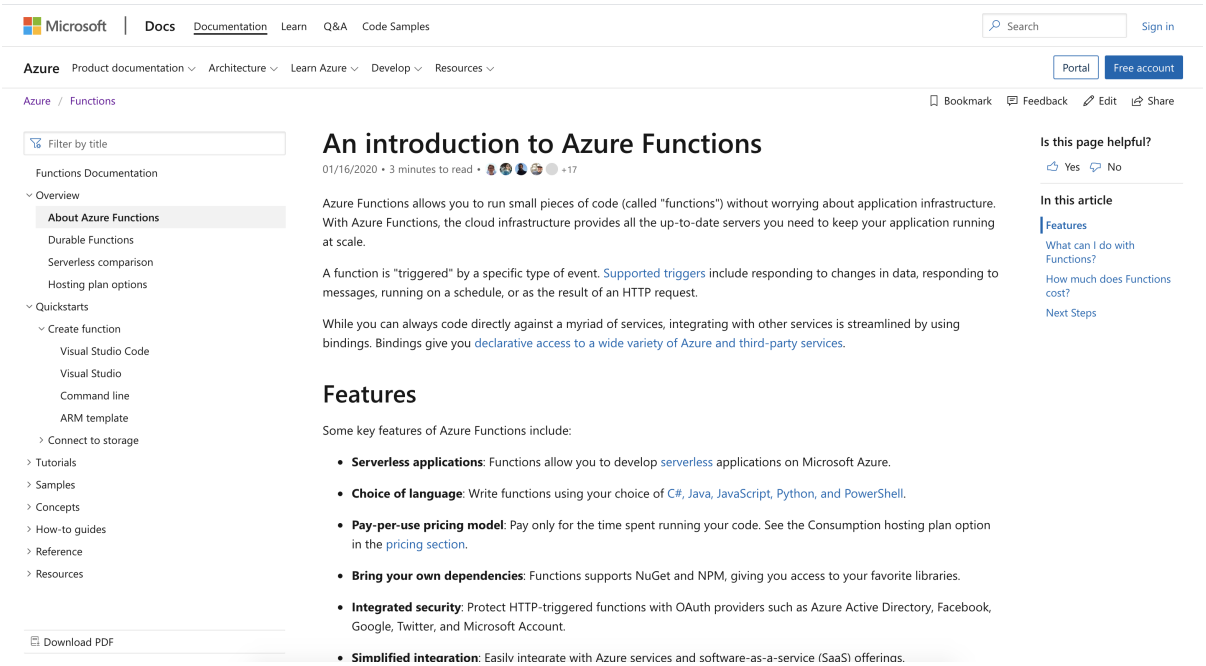

Figura 3.2: *Artigo na documentação da Azure*

O *site* oficial da documentação que utiliza o conteúdo desse repositório pode ser encontrado em https://docs.microsoft.com/azure, e sua estrutura é similar ao *site* de documentação do Visual Studio Code. Navegando pelos diferentes tópicos e artigos é notável a velocidade de navegação muito boa pelo *site*, o que evidencia que o processo de *build* deste também serve páginas estáticas para o navegador, e também utiliza *Server-Side Rendering*. A estrutura da página de artigos é muito similar à vista na documentação do VS Code, e também temos um botão que nos leva para o conteúdo da página no GitHub, onde pode ser editado.

3.2.3 Flutter (flutter/flutter)

Flutter é um *toolkit* para construção de interfaces, desenvolvido pela Google para criação de aplicações para Web, *desktop*, iOS e Android que são nativamente compiladas, podendo até mesmo compartilhar o mesmo código. Sua documentação pode ser encontrada em Flutter Developer Docs.

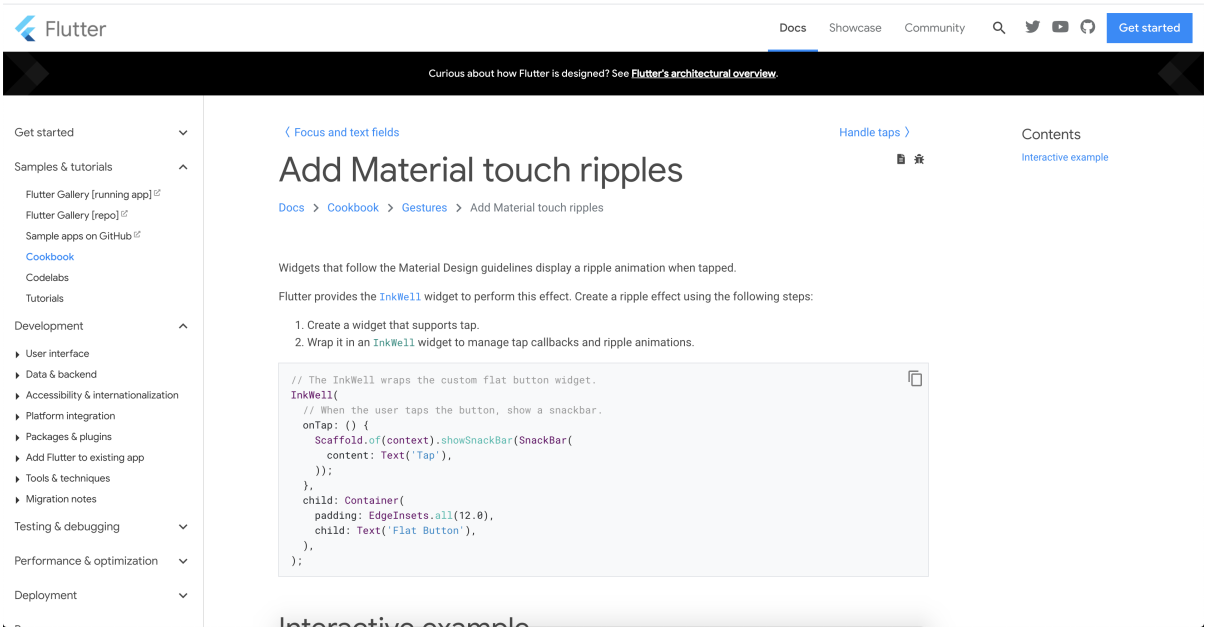

Figura 3.3: *Artigo na documentação do Flutter*

No *site* de documentação, temos uma página inicial com uma série de *links* e vídeos, além de uma sessão dedicada às atualizações mais recentes ao site, algo similar aos *release notes* feitos pelo time do VS Code. Além disso temos um menu lateral onde podemos encontrar os artigos disponíveis. Nas páginas de artigos, no canto superior direito, ao lado dos títulos de cada artigo, temos dois botões discretos que nos levam para o *markdown* da página no repositório que contém a documentação, escrita em *markdown*, e uma página de criação de *Issues* do GitHub também no repositório de documentação. Novamente, o time que mantém o projeto se alavancando do poder da comunidade *open source* para manter sua documentação atualizada e revisada. Em termos organizacionais, o *site* de documentação é muito similar ao que foi visto no VS Code. Podemos observar novamente uma navegação muito rápida pelos artigos e diferentes sessões, também justificada pela geração estática do *site* no processo de *build* e *Server-Side Rendering*, entregando a página pronta para o navegador.

Diferente dos *sites* dos projetos da Microsoft, o *site* do Flutter é completamente open source, permitindo uma investigação sobre o seu processo de *build*. O código-fonte pode ser encontrado em flutter/website, e vemos que nesse mesmo repositório temos toda a documentação escrita em *markdown* junto ao resto do código-fonte do *site*. Ao abrir um dos artigos vemos que no *markdown* temos alguns *tokens* sendo utilizados como variáveis nos textos e cabeçalhos com *metadata* que serão utilizados no processo de *build*, o que indica que esses arquivos serão processados como *templates*. As informações referentes ao *build* do *site* estão podem ser encontradas em um sub-diretório Git Dart Site-Shared. Podemos ver também que o *markdown* é processado e as páginas estáticas são geradas pelo Jekyll, um gerador de *sites* estáticos escrito em *Ruby*.

3.2.4 Tensorflow (tensorflow/tensorflow)

TensorFlow é um solução *open source* criada e mantida pela Google para *machine learning*, contando com um vasto ecossistema de ferramentas, bibliotecas e diversos recursos criados pela sua comunidade para permitir que pesquisadores e desenvolvedores possam facilmente se alavancar dos modelos computacionais desse campo. Sua documentação pode ser encontrada em TensorFlow.

O time responsável pelo TensorFlow trata sua documentação dividindo-a em duas categorias de acordo com o seu conteúdo, como explicado em Contribute to the TensorFlow documentation, sendo cada uma delas gerada de uma maneira:

- *API Reference* - essa documentação é gerada por *docstrings* escritas ao longo do código-fonte do projeto, ou seja, a documentação que é exibida no *site* referente à API vem da própria implementação referente ao conteúdo sendo consumido. Tal código-fonte é *open source* e pode ser encontrado no repositório oficial do projeto;
- *Narrative Documentation* - a documentação narrativa é composta por tutoriais e guias diversos, similar ao conteúdo encontrado na documentação do VS Code. Esse conteúdo também é *open source* e escrito em *markdown*, podendo ser encontrado e editado em tensorflow/docs. Algo singular nessa documentação é a presença de diversos *Jupyter Notebooks*, que constituem um artigo completo de documentação.

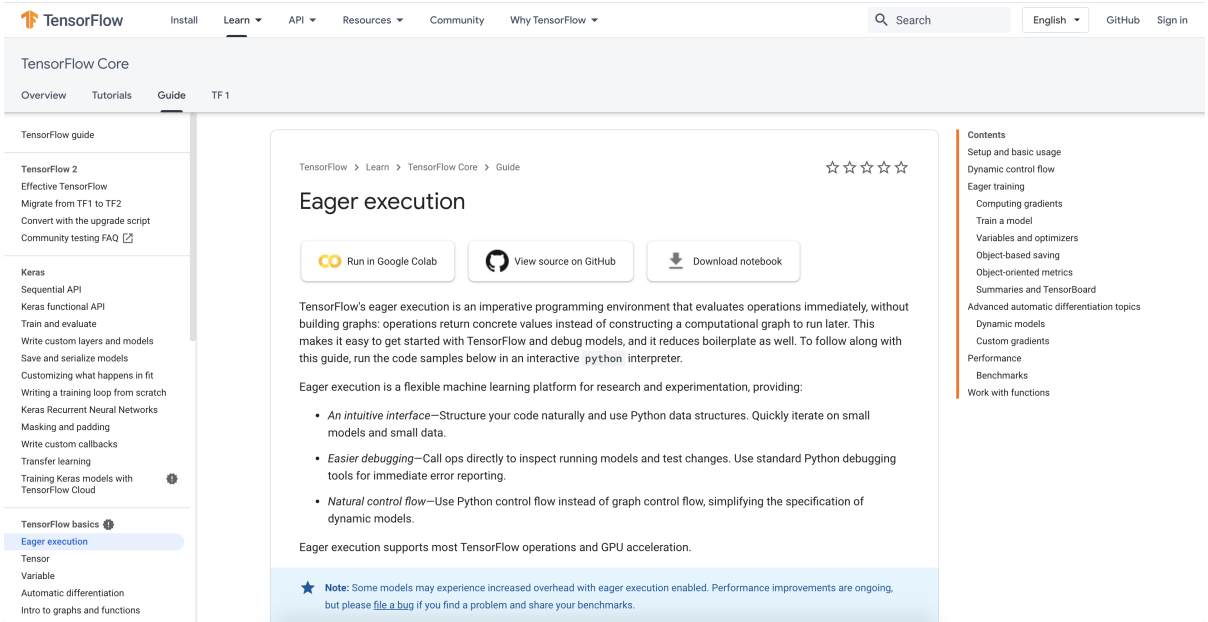

Figura 3.4: *Artigo na documentação do TensorFlow*

Navegando pelo *site*, podemos identificar as mesmas características observadas até aqui, ou seja, as páginas são estaticamente geradas para cada artigo e *Server-Side rendering*, o que garante uma experiência rápida de navegação, sem estados de carregamento. Um ponto de destaque na documentação do TensorFlow é a quantidade de oportunidades para interação do usuário. Apesar da página ser estaticamente gerada, o que impossibilita interatividade, o usuário é constantemente incentivado a testar o conteúdo que está lendo, e muitos trechos

de código estão presentes para serem copiados e testados. Esse também é o primeiro projeto que analisamos que utiliza documentação dentro de seu código-fonte.

Outros pontos que não vimos até agora, mas merecem serem citados são: internacionalização e avaliação do conteúdo da documentação pelos seus usuários. A documentação do TensorFlow está disponível em seu *site* em impressionantes 21 idiomas, com traduções sendo impulsionadas pela comunidade *open source* do projeto. Quanto a avaliação do conteúdo da documentação por parte de seus usuários, vemos em todos os artigos um espaço para avaliação, que deve ser utilizado pelo time responsável para orientar esforços de melhora.

3.2.5 React Native (facebook/react-native)

React Native é uma biblioteca *JavaScript* criada e mantida pela Facebook com o objetivo de combinar as melhores partes do desenvolvimento de aplicações nativas com a experiência de desenvolvimento de aplicações para Web usando React. Seu principal ponto de destaque é possibilitar que aplicações sejam desenvolvidas usando React, para a Web, e possam ser facilmente utilizadas em iOS e Android. Seu site pode ser encontrado em React Native.

A documentação do projeto começa com uma página de *Getting Started* e um menu de navegação lateral. Ao navegar por diferentes artigos observam-se elementos comuns com os demais projetos analisados, como: todos os artigos contém um *link* que leva o usuário ao conteúdo *Markdown* que pode ser editado pela comunidade; navegação rápida devido ao uso de um processo de *build* que serve páginas estáticas, apesar de não ser total nesse caso, pois existem elementos interativos nos artigos, o que requer processamento no cliente; uso de *Server-Side Rendering*.

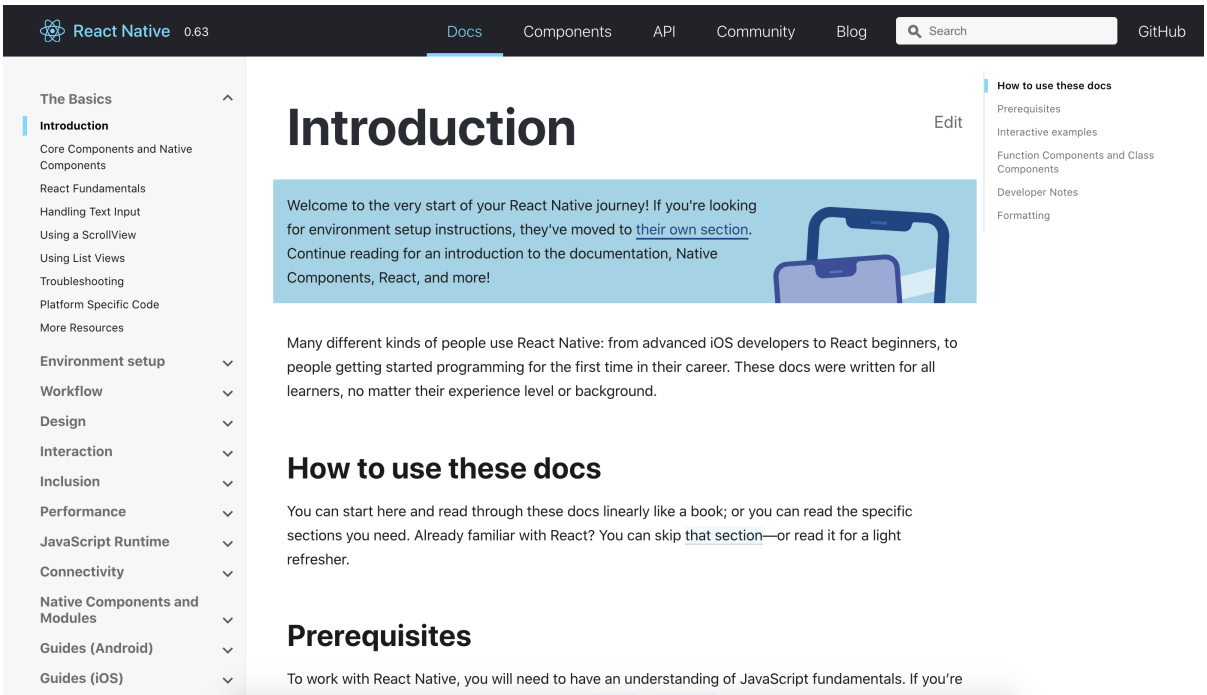

Figura 3.5: *Artigo na documentação do React Native*

O código-fonte do *site* inteiro é aberto, não apenas seu conteúdo *markdown*, e pode ser encontrado em facebook/react-native-website. Ao analisar o código, podemos ver que a documentação em *markdown* é processada por um *software* chamado Docusaurus, também desenvolvido pelo Facebook, que utiliza React para gerar árvores de componentes a partir de *markdown*.

3.2.6 Kubernetes (kubernetes/kubernetes)

Kubernetes é uma plataforma extensível e *open source* para gerenciamento de *workloads* e serviços conteinerizados. Sua documentação pode ser encontrada em Kubernetes Documentation.

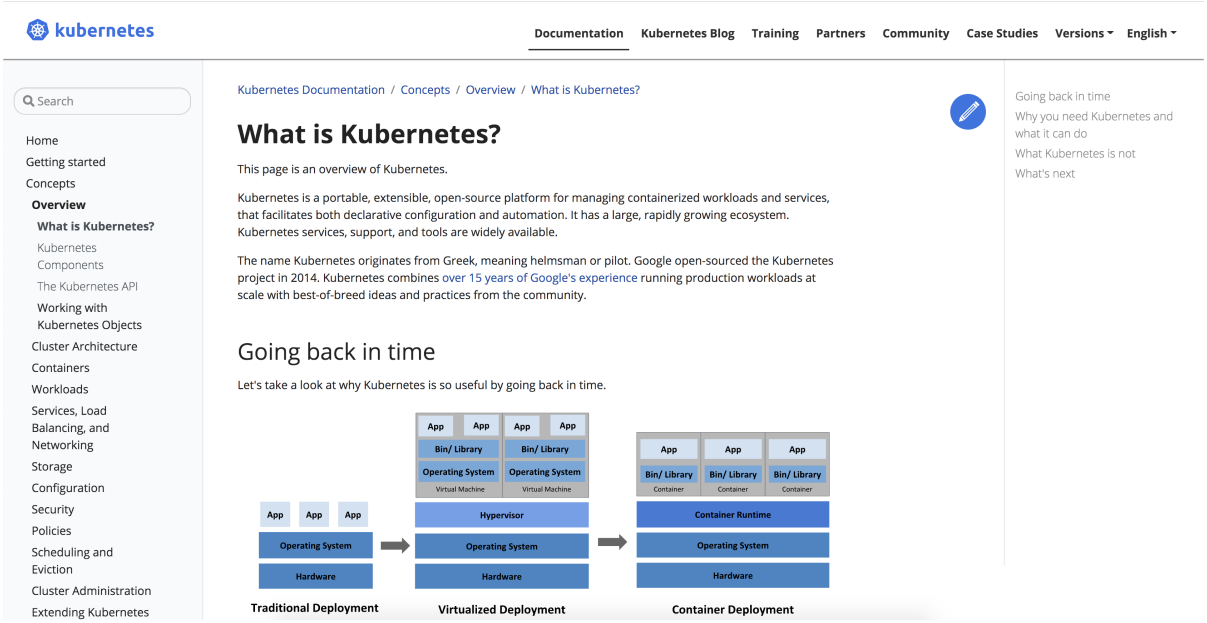

Figura 3.6: *Artigo na documentação do Kubernetes*

A documentação encontrada no site é escrita em *markdown*, processada e servida usando Hugo, um gerador de sites estáticos *open source*. Como os demais projetos vistos anteriormente, a documentação do Kubernetes também é gerada estaticamente a partir de arquivos *markdown* que estão abertos a colaboração da comunidade. Um ponto de destaque na documentação do projeto é o grande foco em internacionalização do conteúdo. O diretório contento o conteúdo *markdown* do site em kubernetes/website é dividido em 15 idiomas.

3.2.7 Definitely Typed (DefinitelyTyped/DefinitelyTyped)

Esse projeto é muito diferente dos demais, pois não é exatamente um *software* em forma de produto, mas sim um repositório com definições de tipos para *TypeScript* de diversos módulos escritos em *JavaScript* e publicados no NPM.

Esse repositório possui uma abordagem mais simples de documentação, dado que a única forma de contribuir para o repositório é com definições de tipos. Sua documentação é constituída pelo seu README, que contém informações gerais sobre o projeto e instruções

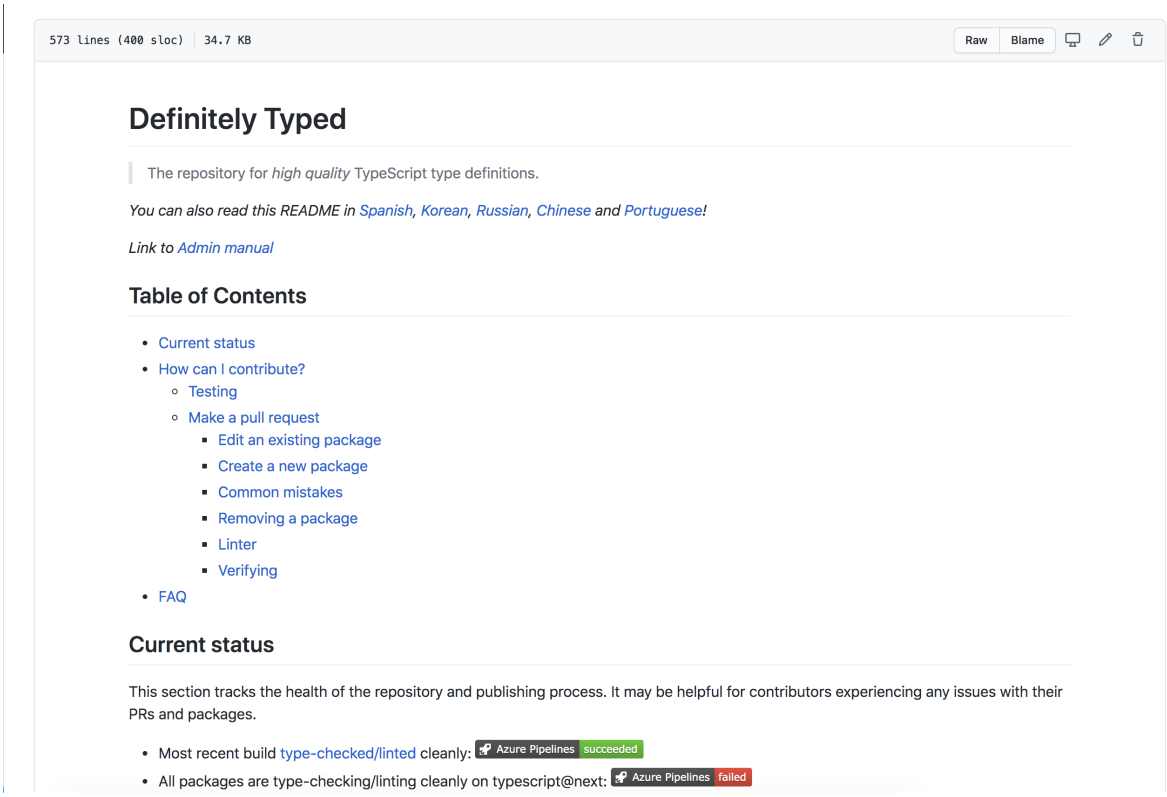

Figura 3.7: *README.md no repositório do projeto Definitely Typed*

para contribuir. Um ponto de destaque é a preocupação do time que mantém o repositório com internacionalização, tendo seu README em 5 idiomas.

3.2.8 Ansible (ansible/ansible)

Ansible é uma plataforma *open source* para automatização de processos de *deploy* de serviços, criada e mantida pela Red Hat. Sua documentação pode ser encontrada em Ansible Docs.

A documentação desse projeto é mais simples do que a dos demais grandes projetos de software, muito focada no conteúdo de texto. Da mesma maneira que as demais, esse conteúdo é escrito em uma linguagem de marcação e passa por um processo de *build* para gerar páginas estáticas e um índice para navegação, ambos sendo enviados prontos para o navegador, o que torna a navegação rápida. O principal ponto de destaque é a utilização de uma linguagem de marcação diferente do markdown que vimos até então. O conteúdo da documentação é *open source*, e faz parte do repositório principal do projeto em ansible/ansible, e é escrito em *ReStructuredText*, ou *rst*. Esse conteúdo é então processado pela ferramenta de documentação Sphinx, geradora de documentação de projetos em Python.

O uso de ReStructuredText é muito comum entre projetos de software escritos em Python. Até mesmo a documentação oficial da linguagem em Python Docs é escrita em *rst* e processada pela Sphinx. Sua sintaxe sacrifica a facilidade de leitura do markdown em troca de formas mais avançadas, com foco na especificação técnica da saída esperada.

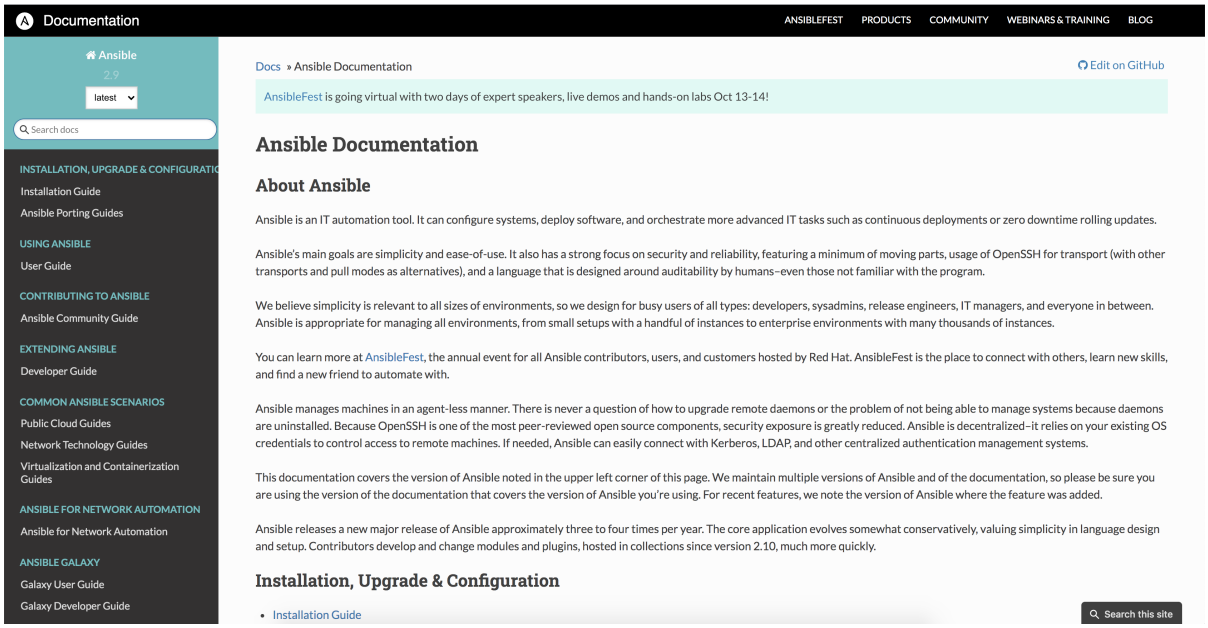

Figura 3.8: *Página inicial na documentação do Ansible*

Além disso, o uso com Sphinx permite a geração automática de documentação com o uso de *docstring*, o que também é utilizado no TensorFlow.

3.2.9 Home Assistant (home-assistant/home-assistant)

Home Assistant é um software de automação doméstica com foco em total controle e privacidade que conta com uma comunidade entusiasta de projetos *faça você mesmo*, ou *DIY* em inglês. Sua documentação pode ser encontrada em Home Assistant Docs.

Navegando pela documentação do projeto podemos verificar os mesmos elementos básicos vistos até aqui: navegação rápida graças a páginas estáticas chegando prontas para o navegador, aliadas ao uso de *Server-Side rendering*; conteúdo aberto e colaborativo, escrito em markdown, que pode ser encontrado em home-assistant/home-assistant.io; uso de *metadata* no markdown para ser consumido no processo de *build*. Ao mesmo tempo vemos uma abordagem simples empregada no *design* do site, que utiliza o *jekyll* no seu processo de *build*.

3.2.10 Comentários

Com essa pesquisa conseguimos identificar alguns pontos em comum e algumas particularidades dos projetos *open source* mais populares do GitHub quanto a sua documentação. Isso nos permite entender melhor como, na prática, esses projetos buscam resolver alguns dos problemas que vimos até agora. Algumas considerações sobre as soluções apresentadas:

- Dos 9 projetos estudados, 8 utilizam sites estáticos gerados por um processo de *build* no servidor, o que garante excelente velocidade na navegação e menos trabalho por parte do navegador. A documentação do React Native utiliza essa técnica parcialmente pois utiliza amostras de código interativas em alguns de seus artigos, e isso

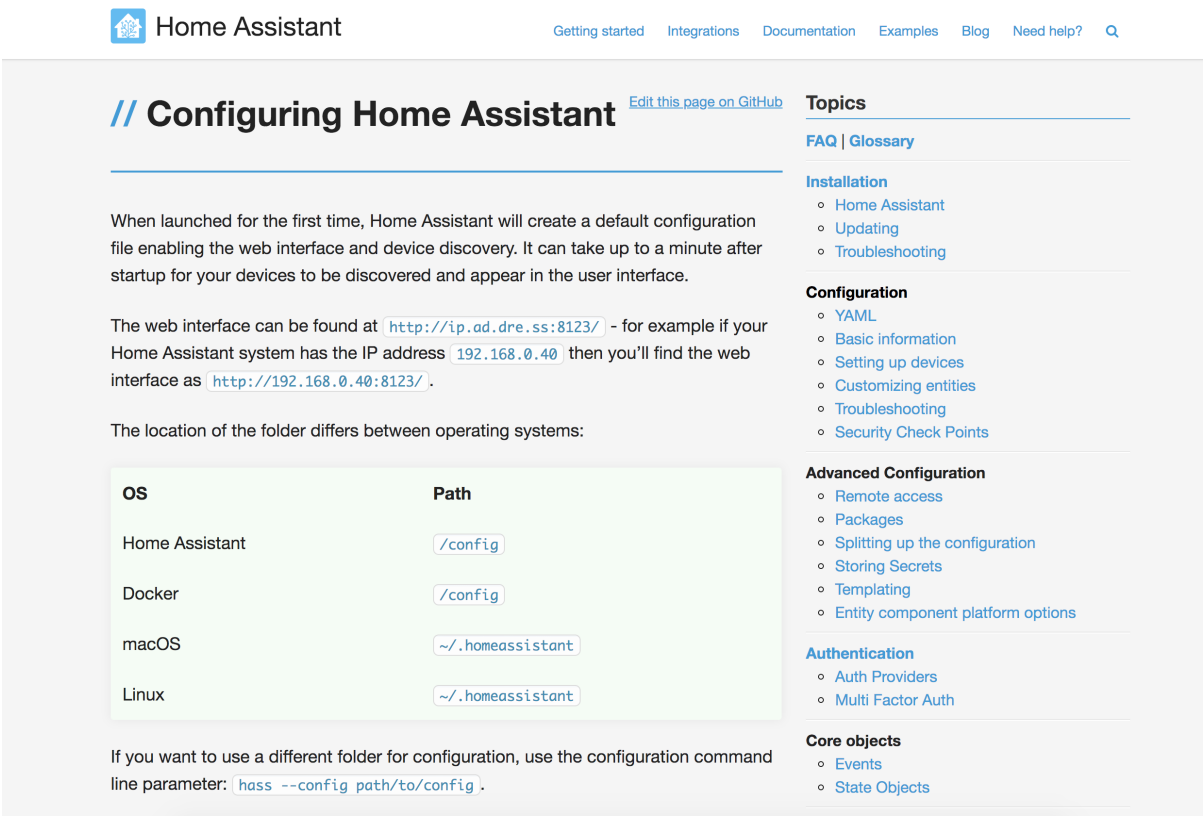

Figura 3.9: *Artigo na documentação do Home Assistant*

necessariamente precisa ser feito no cliente.

- Todos os projetos possuem conteúdo aberto e colaborativo, o que permite que se a comunidade *open source* contribua para a documentação dos projetos que utiliza.

- Todos os projetos, com exceção do Ansible, utilizam *markdown* como linguagem de *markup* para o conteúdo de sua documentação, uma linguagem muito popular entre a comunidade *open source*, principalmente no desenvolvimento Web. O Ansible utiliza ReStructuredText, muito comum entre projetos feitos em Python.

- Os projetos que utilizam *docstrings*, comentários no código-fonte para gerar documentação, são aqueles que expõem uma API bem definida e utilizada como base para criação de novas aplicações com base nessas APIs, no caso TensorFlow, Kubernetes e Ansible, todas aplicações *back-end*.

- Quanto à internacionalização, 5 dos 9 projetos discutidos dão suporte para ao menos 2 idiomas, sendo o inglês o padrão em todos os projetos. Um ponto de destaque é que os três projetos criados e mantidos pela Google que foram discutidos estão nessa categoria, o que possibilita que um grupo maior de pessoas no mundo tenha acesso aos produtos.

3.3 Problemas mais frequentes

Em Aghajani *et al.*, 2019, são apresentados resultados de uma pesquisa realizada em grande parte em repositórios de projetos *open source*, no GitHub. Consideramos essa pesquisa uma boa referência para quais seriam os problemas mais comuns encontrados na documentação de softwares *open source*. Seus resultados mostraram que a maioria dos problemas que usuários encontram com a documentação são referentes ao conteúdo propriamente dito, ou seja, documentação incompleta ou desatualizada.

O segundo grupo de problemas mais comuns são os relacionados a como o conteúdo da documentação está escrito e organizado. Metade das reclamações de usuários nesse grupo mencionava a dificuldade em não encontrar o conteúdo que estavam buscando, mesmo ele estando na documentação. Outro problema citado, que foi destacado na nossa pesquisa da na sessão anterior, foi a dificuldade por parte dos usuários na colaboração com a documentação. Esses problemas nos levam a concluir que a organização do conteúdo da documentação, junto à uma solução de busca seriam muito importantes para projetos *open source* com muita documentação são importantes para garantir uma boa experiência de uso. Acreditamos que a solução utilizada pelos projetos que vimos na sessão anterior (adicionar em todos os arquivos um *link* para uma página de edição) seja suficiente para evitar a maioria das dúvidas quanto à colaboração.

O terceiro grupo de problemas mais frequentes foram os relacionados com as ferramentas utilizadas para manter e produzir documentação, escolhidas pelos mantenedores do projeto, em geral o time *core*. Um resultado interessante foi que as dúvidas mais comuns nesse grupo demonstravam problemas na documentação das próprias ferramentas utilizadas. O principal risco que enxergamos nesse tipo de problema é a possibilidade dessas ferramentas se tornarem obstáculos para a colaboração da comunidade. Como pontuado por Dagenais e Robillard, 2010, contribuidores estão, em geral, **menos** dispostos a contribuir com documentação do que com código. Logo, o processo de contribuição deve ser fácil e com o mínimo de barreiras possível.

Por fim, o quarto grupo de problemas mais comuns diz respeito ao processo de manutenção e criação de documentação. Dentre os problemas desse grupo, um que também apareceu na discussão da sessão anterior foi a internacionalização do conteúdo da documentação. Este não é um problema fácil de resolver, pois requer um grande esforço por parte do time *core* do projeto em incentivar a comunidade a traduzir o conteúdo que eles produzem. Além disso, também precisam de membros dessas comunidades que saibam ler e escrever em pelo menos 2 idiomas, o que acaba restringindo o número de potenciais colaboradores. Manter a qualidade do conteúdo produzido em múltiplos idiomas também é um problema difícil, pois requer revisão por falantes nativos de cada idioma.

3.4 Próximos passos

Agora que já conduzimos uma revisão da literatura sobre documentação de software e seus problemas, estudamos como alguns dos maiores projetos de software open source lidam com ela e quais os problemas mais frequentemente enfrentados em projetos dessa natureza, vamos falar sobre os próximos passos deste trabalho.

Nos próximos capítulos vamos apresentar uma experiência prática que tivemos durante a avaliação e criação de uma solução de documentação em uma empresa de tecnologia brasileira, a VTEX. Foi com base em tudo o que vimos até agora que identificamos quais deveriam ser as características fundamentais de um sistema de documentação que não gerasse os problemas levantados. O sistema implementado é capaz de centralizar a documentação–o que foi um desafio devido à natureza distribuída do seu conteúdo–e automatizar sua publicação e distribuição.

Introduzimos também uma série de mudanças culturais–em específico em um time dentro da empresa que trabalha com um produto *open source*–que consideramos importantes para garantir que a manutenção e criação de documentação se tornasse uma responsabilidade compartilhada entre desenvolvedores e *technical writers*. Buscamos também encontrar fluxos de trabalho que facilitassem contribuições vindas da comunidade *open source* de usuários do projeto.

Capítulo 4

Uma aplicação real

Buscando aplicar o que aprendemos até o momento em um ambiente real na indústria de software, tivemos a oportunidade de trabalhar com a VTEX, em um produto *open source* chamado Store Framework. Durante essa colaboração, estudamos qual era o estado da documentação já existente sobre o produto e buscamos identificar pontos de melhora. Então, entendendo as necessidades particulares do produto e dos seus desenvolvedores, implementamos um sistema de software capaz de centralizar toda a documentação em um único endereço, além de automatizar o processo de publicação e atualização de conteúdo.

Ao longo do processo de estudo do Store Framework e da implementação do sistema, percebemos que mudanças culturais seriam necessárias. Conversando com o time de desenvolvedores e a *technical writer* responsável pela documentação do *framework*, fomos capazes de definir e testar essas mudanças. Elas serão o foco do próximo capítulo. Neste capítulo vamos apresentar rapidamente o produto em que colaboramos, o estado da documentação antes de começarmos nosso trabalho, e o sistema implementado.

4.1 VTEX IO e Store Framework

O produto em que trabalhamos é o Store Framework da VTEX, mas como ele faz parte de outro produto, o VTEX IO, vamos descrevê-lo primeiro.

VTEX IO é uma plataforma de desenvolvimento que busca abstrair as partes mais complexas da configuração de aplicações feitas para a Web. Um grande diferencial da plataforma é que ela é a sua infraestrutura nativa em nuvem, ou seja, usuários do VTEX IO nunca desenvolvem nada localmente, em seus computadores, mas todo o código é executado e testado em nuvem, em tempo real. Com isso, o VTEX IO resolve problemas de compatibilidade entre times que trabalham em diferentes ambientes de desenvolvimento, ou até permite que ambientes sejam compartilhados através de uma URL.

Uma característica do VTEX IO, que será especialmente relevante mais adiante, é a existência de *apps*. Cada "pacote" de funcionalidades que um usuário do VTEX IO deseja usar pode ser acessado através de *apps*, de maneira semelhante ao que é feito em *smartphones* Android ou iOS. Para desenvolver algo usando a plataforma, o usuário

deve acessar uma conta, e nessa conta ele pode configurar diferentes opções, e instalar as funcionalidades que necessita. Veremos mais adiante que, pensando de acordo com este modelo de *apps*, a documentação deve ser feita com base em **cada** *app*, ou seja, cada *app* é responsável pela sua própria documentação.

Store Framework, é um *framework* para o desenvolvimento de lojas de *ecommerce* no VTEX IO. Este *framework* pode ser dividido em dois conjuntos de funcionalidades: um conjunto de componentes, chamados de "blocos", que devem ser compostos para criar a experiência de compra de um cliente final. Estes blocos são distribuídos como *apps* do VTEX IO; Suporte para uma linguagem declarativa onde desenvolvedores declaram quais blocos devem ser renderizados na loja, e como eles devem ser configurados.

Os usuários do Store Framework são desenvolvedores ou possuem certo conhecimento prévio sobre desenvolvimento de aplicações para a Web. Portanto, é este o público com o qual nos preocupamos quando pensamos sobre a documentação do VTEX IO e do Store Framework.

Com esta primeira descrição sobre o VTEX IO e o Store Framework, pudemos perceber que a documentação poderia ser dividida em dois tipos, de maneira análoga à divisão do Store Framework. Portanto teríamos documentações individuais, para cada *app* e seus "blocos", e documentações que descrevem a experiência de desenvolvimento com o *framework*.

4.2 Estado inicial da documentação

Antes de começarmos nosso trabalho com o time do Store Framework para melhorar sua documentação, foi importante entender o que já existia, e quais eram os problemas enfrentados naquele momento. Começamos entendendo o contexto do VTEX IO e do Store Framework dentro da VTEX como um todo. Assim poderíamos entender melhor qual o objetivo da melhora na documentação, e qual o seu público-alvo.

Antes de começarmos nossos trabalhos, ambos os produtos eram novos no contexto da empresa, e o número de clientes e lojas sendo criadas com o *framework* era pequeno comparado ao número de novas lojas sendo criadas na VTEX. Os clientes que estavam utilizando o VTEX IO e o Store Framework tinham sido escolhidos cuidadosamente para serem parceiros no desenvolvimento desses dois produtos. O VTEX IO como um todo ainda estava em processo de desenvolvimento e amadurecimento, com mudanças grandes acontecendo frequentemente. Como o número de usuários da plataforma ainda era pequeno e facilmente acessível, mantendo contato direto com os desenvolvedores, as informações sobre mudanças e instruções de uso eram muitas vezes passadas em conversas ou mensagens privadas. Não existia ainda a necessidade de uma comunicação de amplo alcance e a documentação de ambos os produtos não foi o foco dos desenvolvedores até aquele momento.

Com o amadurecimento do VTEX IO e do Store Framework, ambos os produtos foram ganhando números cada vez maiores de usuários. Novas lojas na VTEX tinham a opção de serem construídas desde o começo com base no VTEX IO. Com mais lojas sendo construídas no VTEX IO, mais desenvolvedores tiveram contato diário com o Store Framework, e o

método de comunicação direta com os desenvolvedores do *framework* demonstrou sinais de que não seria escalável. Foi então que a necessidade de uma documentação escrita, que pudesse ser facilmente compartilhada com desenvolvedores ficou clara. Tal solução, quando começamos o nosso trabalho, era um documento criado no Google Docs, com o nome de "VTEX IO Master Documentation". Neste documento, documentação escrita de maneira colaborativa pelos desenvolvedores de diferentes times que compunham o VTEX IO estava disponível. O acesso ao documento era controlado individualmente, sendo compartilhado com usuários do VTEX IO através do contato direto com algum membro do time de desenvolvimento. Para o Store Framework, existia ainda outro tipo de documentação, referente às *apps* que continham "blocos" para construção de lojas, como descrito na sessão anterior. Essa documentação estava distribuída em repositórios no GitHub, junto ao código-fonte *open source* dessas *apps*.

O VTEX IO e o Store Framework continuaram crescendo, com novas lojas sendo criadas, e cada vez mais desenvolvedores utilizando ambos os produtos. O Store Framework seria em breve anunciado como a solução padrão para a implementação de novas lojas na VTEX. Diante disso, a falta de uma solução de documentação unificada, de qualidade, e acesso público se tornou uma prioridade. Foi então que começamos o nosso planejamento e trabalho na solução de documentação que criamos para VTEX IO.

4.3 Principais requisitos

O primeiro passo no desenvolvimento de uma nova solução de documentação para o VTEX IO, com foco no Store Framework, foi listar alguns pré-requisitos, dos quais não abriríamos mão na implementação final. Nos baseamos na revisão da literatura sobre o assunto e no que observamos em grandes projetos *open source* hospedados no GitHub. Alguns desses requisitos foram os seguintes:

1. A documentação deveria estar o mais próxima possível do código que ela documenta. Reduzindo o esforço necessário por parte do desenvolvedor para atualizar ou criar documentação no projeto em que já está trabalhando.

2. Os desenvolvedores não devem ter que se preocupar com a publicação da documentação. Eliminando potenciais atritos no processo de atualização de documentação, tornando-o o mais simples possível.

3. Deveríamos ter uma única fonte de verdade para nossa documentação. Um único local onde qualquer usuário consiga encontrar toda a documentação que ele precise. Assim tornamos o processo de busca por informação mais simples.

4. A comunidade *open source* deveria ser capaz de contribuir com a documentação.

5. A linguagem padrão para toda a documentação seria *markdown*. *Markdown* é a linguagem padrão para a escrita de READMEs no GitHub e no GitLab, com suporte para imagens e trechos de código devidamente formatados, e muito comum em projetos *open source*.

6. A solução não deveria impedir a internacionalização do seu conteúdo. O VTEX IO e o Store Framework são utilizados por usuários em diversos países, e não podemos

ter uma solução que não seja capaz de suportar todos eles.

7. *Technical writers* deveriam ter total poder sobre a estrutura da documentação e como o seu conteúdo é apresentado ao usuário final. Como descrito por Sapir *et al.*, 2016, *technical writers* assumem um papel muito importante nos times de produtos de empresas de tecnologia como a VTEX. Esses profissionais são essenciais para evitar o problema da má organização e falta de estrutura da documentação, apontado como um dos mais frequentes nos capítulos anteriores.

O design e a implementação final da solução de documentação teve como base esses pré-requisitos. Todos eles foram levantados por nós e discutidos com o time responsável pelo Store Framework, e com o time de Technical Solutions da VTEX, que também nos ajudou ao longo do processo.

4.4 Implementação da solução

Uma vez que tínhamos listados nossos critérios de satisfação, antes mesmo de começar o processo de implementação da solução, conseguimos uma boa visão sobre quais características o produto final deveria ter. Ao final de algumas reuniões com o time do Store Framework, e de Technical Solutions da VTEX, chegamos à uma solução de duas partes:

- Um serviço que implementaria uma API GraphQL, rodando no próprio VTEX IO. Este serviço seria responsável por encontrar toda a documentação que estivesse disponível e devesse ser mostrada ao usuário final.
- Um cliente Web, também desenvolvido com o VTEX IO, que seria o *front end* do novo portal de documentação. Este seria o ponto de contado por usuários finais com a documentação do Store Framework, seja ela referente a *apps* específicas, ou ao fluxo de desenvolvimento.

Tendo essa divisão clara desde o começo facilitou o processo de planejamento. Vamos abordar cada uma dessas partes individualmente a seguir.

4.4.1 Serviço GraphQL (`docs-graphql`)

O serviço, que serviria como o *back end* da solução de documentação, foi a parte do projeto mais integrada com o VTEX IO, e envolveu um estudo mais profundo sobre o funcionamento da plataforma. É neste serviço que resolvemos grande parte dos requisitos explicados na sessão anterior.

Primeiro, para evitar a mudança de contexto do lado do desenvolvedor e também remover sua preocupação com a distribuição da documentação, decidimos que iríamos tornar a documentação uma parte integrante de cada *app* desenvolvida com VTEX IO. Assim, conseguimos manter a documentação próxima do código, removendo o atrito que um desenvolvedor encontraria para produzir documentação. Esta foi uma decisão que se mostrou especialmente importante mais adiante, pois permitiu que a documentação de uma *app* fosse versionada junto com a própria *app*.

A partir deste momento em que decidimos que a documentação faria parte de qualquer *app* publicada, o fluxo de publicação dessa documentação já estava resolvido. Bastava integrar utilizar o mesmo fluxo de publicação de *apps* que já fazia parte do VTEX IO e do fluxo de desenvolvimento que os usuários já conheciam.

Apps no VTEX IO passam por um processo de *build* ao serem publicadas, e o produto final desse *build* é armazenado em um *registry*, semelhante ao NPM (JavaScript) ou o PyPI (Python). Essas *apps* então podem ser instaladas em ambientes de desenvolvimento do VTEX IO. Logo, deveríamos integrar a publicação da documentação da *app* neste processo de *build*, para que ela fizesse parte do produto final.

De maneira simplificada, o *build* de uma *app* é um processo no qual um serviço recebe os arquivos de uma *app* na sua forma original, exatamente como foram escritos pelos seus desenvolvedores, e transforma-os de alguma forma. O produto final dessa transformação é enviado ao *registry* e pode ser consumido por outras *apps* de alguma forma, ou simplesmente instalado em ambientes de desenvolvimento.

No VTEX IO, o *build* de uma *app* é dividido em diversas partes dependendo do seu conteúdo. Por exemplo, uma *app* que implementa uma interface Web e ao mesmo tempo um serviço para o *back end*, terá seu *build* dividido em duas partes, uma para cada parte da *app*. Ainda neste exemplo, o código refente à interface Web seria processado por um *builder* de React, enquanto o código referente ao serviço *back end* seria processado pelo *builder* de Node.js. Para manter o código das *apps* organizado, os arquivos a serem processados por diferentes *builders* devem estar em diretórios com os nomes dos seus respectivos *builders*, como na imagem abaixo.

Entendendo como o processo de *build* funciona, ficou claro que uma solução simples que atendia aos nossos pré-requisitos seria criar um novo *builder*, capaz de processar a documentação de *app* e torná-la disponível para o consumo. Nosso serviço GraphQL, que recebeu o nome de `docs-graphql`, seria o consumidor do resultado do *build*, e trataria de expor pela API GraphQL este conteúdo.

Todos os *builders* desenvolvidos no VTEX IO fazem parte de uma grande *app* chamada `builder-hub`, então o primeiro passo da implementação foi criar o novo *builder*, que recebeu o nome de `Docs Builder`, no `builder-hub`. Este *builder* foi implementado para identificar aquivos no formato *markdown*, e incluí-los no produto final do *build* de uma *app*. A escolha pelo formato *markdown* veio tanto da pesquisa descrita no Capítulo 3 deste trabalho, que mostra o quanto *markdown* é comum em projetos *open source*, quanto pela facilidade de escrita e manutenção de documentos neste formato. Agora, cada *app* no VTEX IO teria a sua documentação incluída.

Uma vez que modificamos o processo de *build* das *apps*, tornando a documentação uma parte integrante de toda *app* desenvolvida no VTEX IO, avançamos para a implementação de um serviço que iria expor uma API GraphQL voltada ao uso dessa documentação. O serviço recebeu o nome de `docs-graphql`. Sua principal função é ter acesso ao *registry* do VTEX IO, conseguir ler os arquivos *markdown* que o `Docs Builder` separou, processá-los de alguma maneira e enviá-los para um cliente. A imagem abaixo mostra uma visualização criada com a ferramenta *open source* GraphQL Voyager do *schema* da API GraphQL que criamos.

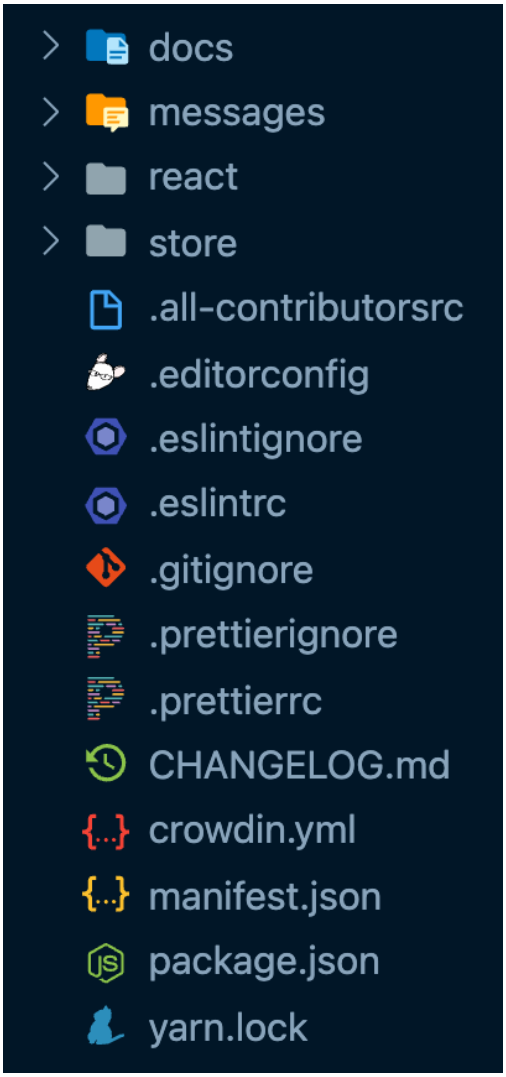

Figura 4.1: *Estrutura de diretórios comum de uma app do VTEX IO*

Durante o desenvolvimento dessa API contamos com a colaboração da *technical writer* do time do Store Framework, que nos ajudou a determinar quais seriam os diferentes tipos de documentação a serem levados em consideração. Citando alguns deles, podemos ver no *schema* acima menções a "*recipes*", que são pequenos tutoriais orientados a guiar o leitor do começo ao fim de uma tarefa que ele precisa cumprir, e "*getting started*", os artigos que compõe o guia de primeiros passos no Store Framework. Vemos também "*release notes*", outro tipo de conteúdo que também seria considerado pelo time como documentação.

4.4.2 Cliente Web (`docs-ui`)

Enquanto estávamos desenvolvendo o *builder* e a API GraphQL, também desenvolvíamos um cliente Web, que seria o portal de documentação oficial do VTEX IO como um todo, incluindo e focando no Store Framework. Esse portal foi desenvolvido usando o próprio VTEX IO, com React e TypeScript, e a *app* recebeu o nome de `docs-ui`.

Durante a primeira iteração do `docs-ui`, implementamos uma interface simples, capaz

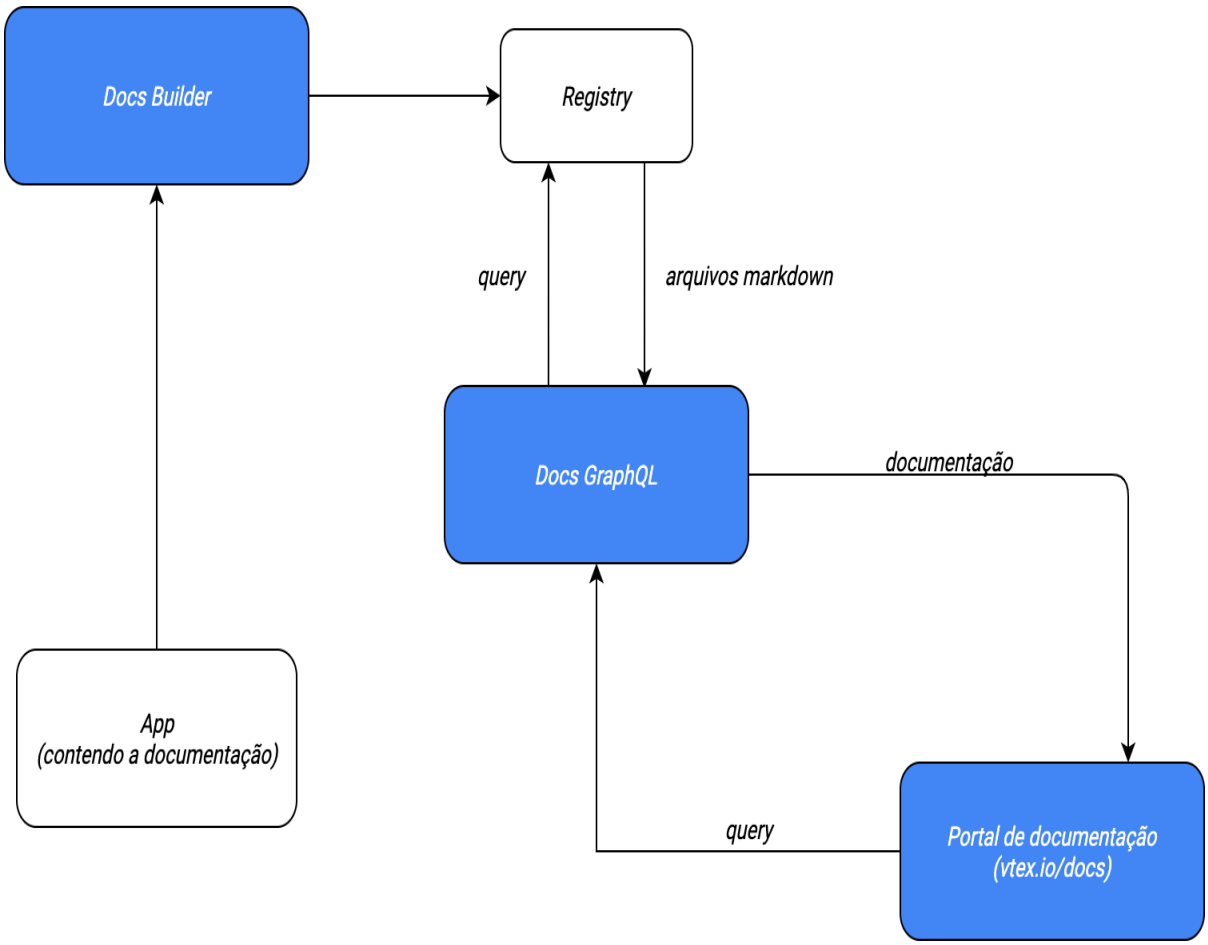

Figura 4.2: *Fluxo que a documentação segue até chegar ao vtex.io/docs*

de se comunicar com o `docs-graphql`, fazendo uma *query* pela documentação de uma certa *app*, e renderizando o *markdown* que era recebido usando componentes React. O principal é que tivéssemos controle sobre o processo de renderização, para conseguir fazer ajustes de design, mas também não tivéssemos grandes penalidades no tempo de carregamento. A primeira versão funcional do `docs-ui` pode ser vista na imagem abaixo.

A interface era simples, composta por "cartões" apenas com o nome das *apps*, e ao clicar no nome de uma delas, o usuário era levado para uma página que renderizava a documentação da última versão lançada da *app*. A lista de *apps* que apareciam nessa interface era escrita manualmente no código ainda, mas este serviu como um protótipo e nos permitiu obter *feedback*.

Ao longo das próximas iterações, experimentamos com diversas ideias, entre elas uma é especialmente relevante, apesar de terem sido modificada até chegar ao produto final, pois definiu como iríamos entregar uma experiência boa ao usuário final. A primeira foi a possibilidade de uma *app* conter um arquivo *markdown* especial na sua documentação, esse arquivo seria um `Summary.md`, e ele seria um índice para a documentação da *app*. Usando esse índice, uma *app* com documentação mais complexa, separada em múltiplos arquivos, poderia descrever essa estrutura e a interface mostraria uma navegação para o usuário, seguindo o índice. Conseguimos implementar uma versão funcional dessa navegação, mas

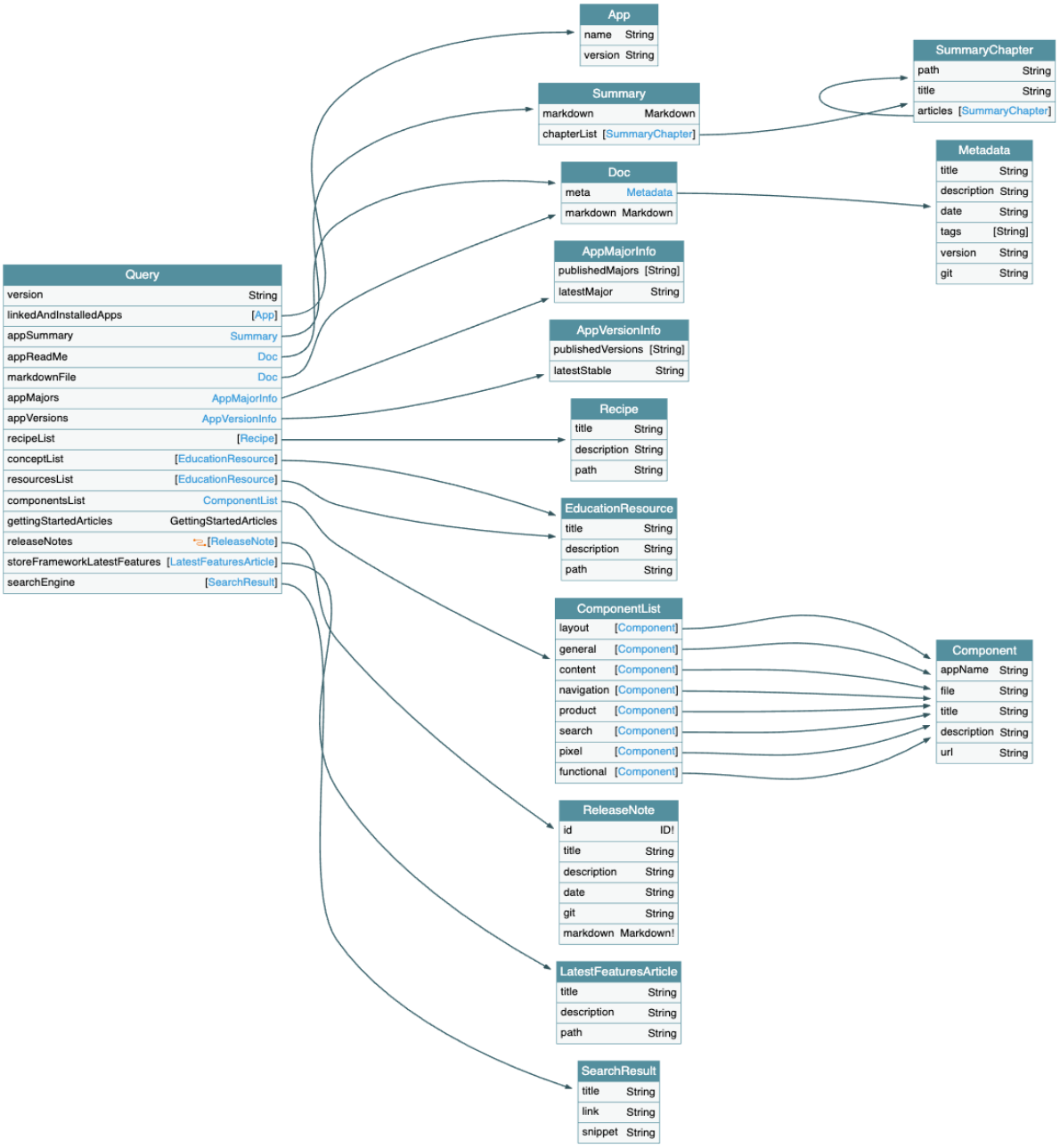

Figura 4.3: *Visualização da API GraphQL do docs-graphql gerada com GraphQL Voyager*

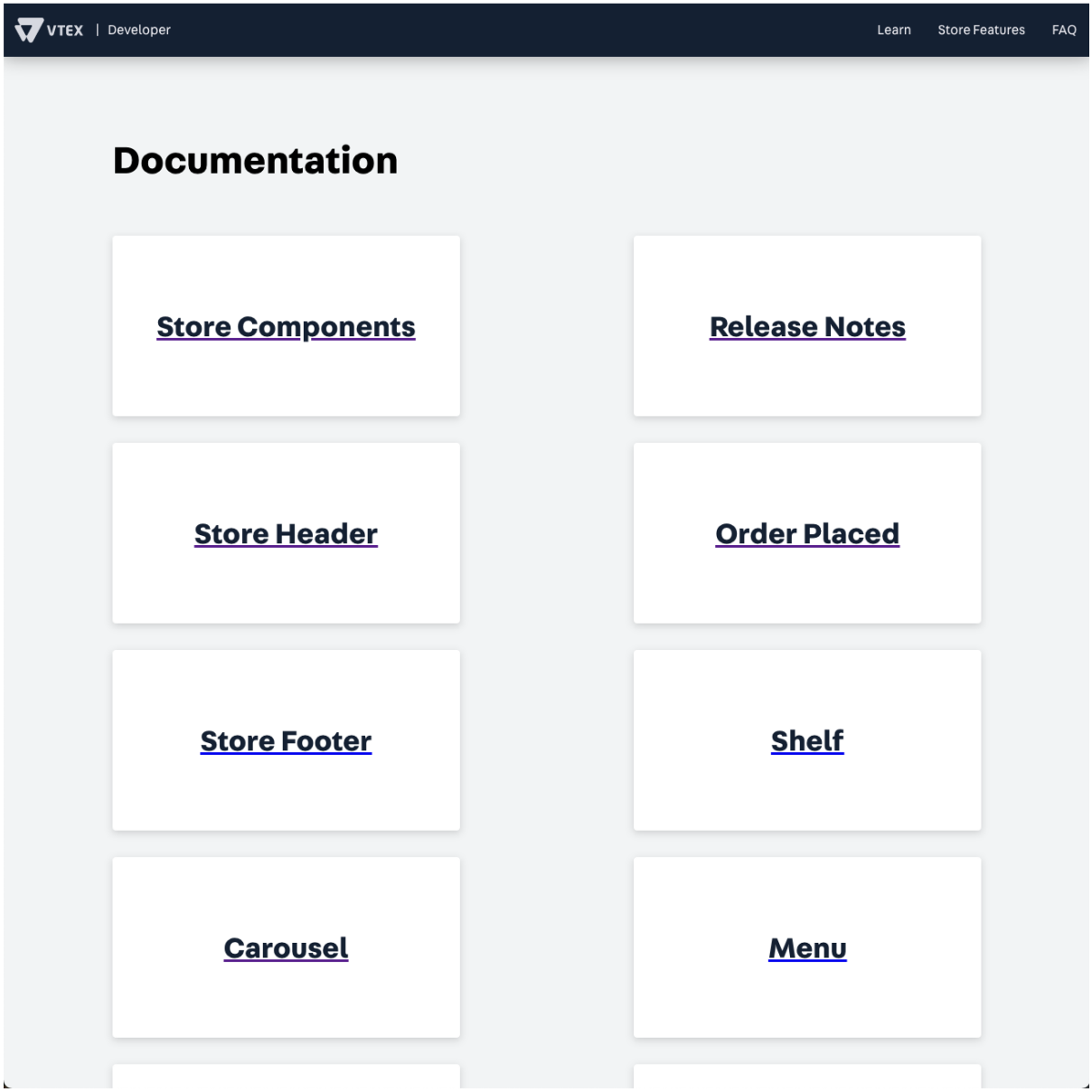

Figura 4.4: *Primeira versão funcional do portal de documentação do VTEX IO*

descartamos a ideia depois de conversas com o designer do time do Store Framework e com outros desenvolvedores, pois prejudicaria a experiência de uso para o usuário final. Foi então que optamos por ter um *único* menu de navegação, do lado esquerdo da interface, que não mudaria dentro da mesma sessão de um usuário.

Uma preocupação constante que tivemos durante o desenvolvimento foi como poderíamos dar à *technical writer* do time o controle sobre a organização e a navegação pela documentação. Era claro para nós que essa era uma responsabilidade dela, e portanto não deveríamos dificultar seu controle. Como demonstrado por Dagenais e Robillard, 2010, é comum que haja atritos entre desenvolvedores e *technical writers* quando eles não fazem parte do mesmo time, e buscamos criar uma ferramenta que evitasse esse tipo de problema. Sapir *et al.*, 2016 mostra que é de grande importância o papel desses profissionais para a criação de documentações de alta qualidade, e fáceis de se utilizar.

Pensando nesse controle, decidimos que o menu de navegação, e a classificação de todo o conteúdo que fosse mostrado no `docs-ui` pudessem ser configurados através de uma *app* do VTEX IO chamada de `io-documentation`. Essa *app* tem um papel central no conteúdo que é mostrado no portal de documentação. Além dela configurar elementos críticos da interface, como o menu de navegação, ela também é o repositório com toda a documentação do VTEX IO que não está contida em outras *apps*. Essa documentação é referente ao fluxo de desenvolvimento, apresentando conceitos sobre a plataforma, tutoriais, entre outros tipos de documentação.

A criação da *app* `io-documentation`, nos permitiu criar um fluxo simples para a *technical writer* do Store Framework conseguir fazer alterações na organização ou publicar novos conteúdos na documentação. Quando decidimos criar essa *app*, também decidimos que ela seria *open source*, e estaria hospedada no GitHub com as demais *apps* do Store Framework, em vtex-apps/io-documentation. Dessa maneira, abriríamos espaço para a comunidade de usuários contribuírem com a documentação, através do fluxo de *pull requests* do GitHub, o que seria acessível por ser um fluxo familiar a desenvolvedores *open source*, e ao mesmo tempo exigiriam revisões e aprovações pela *technical writer* do time. Como discutido em Dagenais e Robillard, 2010, queríamos evitar os problemas que poderiam ser causados por falta de moderação do conteúdo da documentação, mas ao mesmo tempo não impor barreiras muito grandes aos colaboradores.

Ao longo do projeto, sempre tivemos o apoio e a colaboração dos times do Store Framework e de Technical Solutions da VTEX, e mais perto do lançamento oficial do portal de documentação, tivemos o apoio especial de um designer da VTEX. Com a ajuda dele, chegamos ao design que usaríamos para o lançamento da Docs UI 1.0.0. Abaixo estão algumas das telas da interface, começando com a página inicial, seguida da página da documentação de uma dada *app*, e por fim uma página de listagem de *recipes* (tutoriais) sobre um determinado tópico.

Figura 4.5: *Screenshot da primeira versão pública do vtex.io/docs (v1.0.0)*

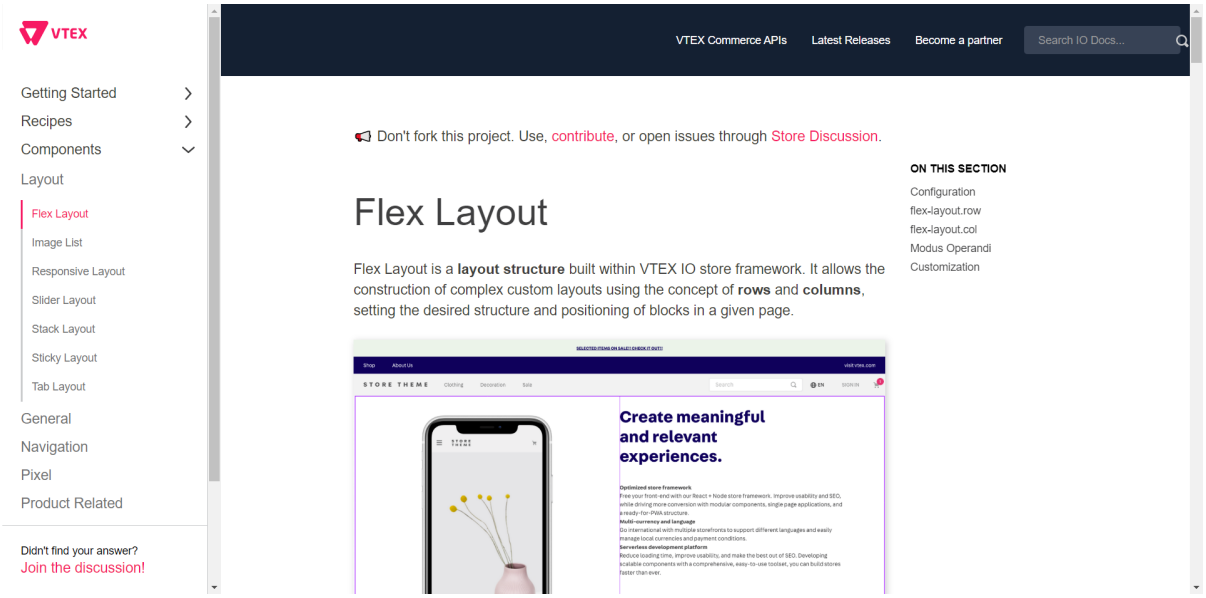

Figura 4.6: *Screenshot da primeira versão pública do vtex.io/docs (v1.0.0)*

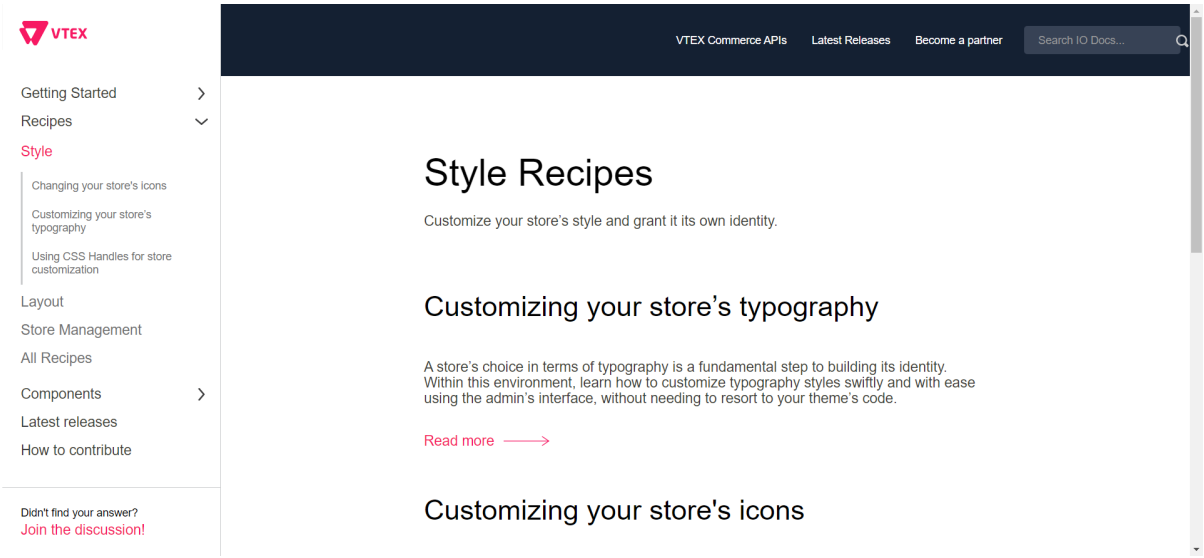

Figura 4.7: *Screenshot da primeira versão pública do vtex.io/docs (v1.0.0)*

4.5 Resultados e próximos passos

Depois de aproximadamente 2 meses e meio de trabalho *full-time* no projeto, conseguimos lançar a primeira versão das três *apps*, `docs-graphql`, `docs-ui` e Docs Builder. Lançamos também um *bot* do GitHub, que chamamos de `docs-bot`. Esse *bot* foi instalado em todos os repositórios *open source* da VTEX, e serviria como um lembrete para todos os desenvolvedores de que, mudanças no código de uma *app*, poderiam implicar em mudanças na documentação dessa *app*.

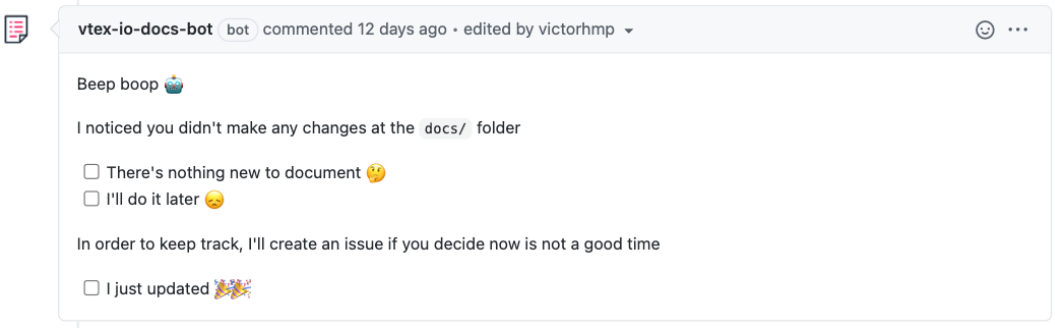

Figura 4.8: *Mensagem do bot em uma pull request quando nenhuma mudança na documentação é detectada*

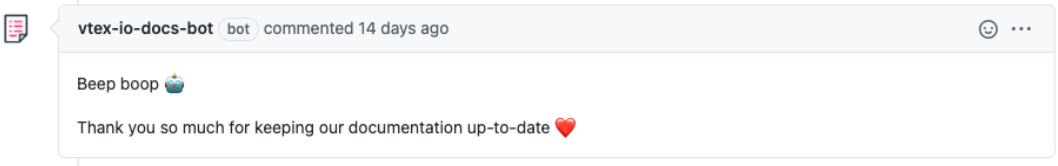

Figura 4.9: *Mensagem do bot em uma pull request quando a documentação foi alterada*

Para oficializar o lançamento do portal de documentação, em vtex.io/docs, fizemos uma

apresentação para toda a VTEX em um evento semanal chamado Demo Friday. Logo após a apresentação, recebemos muitos *feedbacks* positivos, e a recepção dos usuários internos do VTEX IO foi positiva em geral.

Nas semanas seguintes recebemos mais *feedbacks*, principalmente dos times de suporte da VTEX e dos desenvolvedores de *apps* do VTEX IO, que não eram do Store Framework. Para o time de suporte, a existência de um portal de documentação facilitou o fluxo de trabalho, já que não precisavam buscar por informações que poderiam estar em diferentes fontes. Para os desenvolvedores de *apps*, o principal ponto que recebeu elogios foi a facilidade com que poderiam escrever e publicar documentações de suas *apps*. Graças à solução envolvendo o Docs Builder, conseguimos fazer com que, qualquer *app* publicada no VTEX IO que tivesse documentação, estivesse automaticamente disponível no portal de documentação, seguinte o formato de URL: `https://vtex.io/docs/app/<nome-da-app>@<versão-da-app>`. Se a versão da *app* não fosse incluída, a versão mais atual seria buscada.

Retomando os pré-requisitos que estabelecemos no começo do projeto, vemos que conseguimos atender a todos eles.

1. *"A documentação deveria estar o mais próxima possível do código que ela documenta"* e *"Os desenvolvedores não devem ter que se preocupar com a publicação da documentação"* - Conseguimos atingir ambos os requisitos com a criação do Docs Builder, permitindo que a documentação passasse a fazer parte das próprias *apps*. A publicação da documentação de uma *app* é feita automaticamente junto com a publicação do seu código. Assim também fizemos com que a documentação passe a ser naturalmente versionada, acompanhando o versionamento da *app*.

2. *"Deveríamos ter uma única fonte de verdade para nossa documentação. Um único local onde qualquer desenvolvedor consiga encontrar toda a documentação que ele precise"* - Cumprimos este requisito com a criação do portal de documentação em vtex.io/docs. Essa deveria ser a única fonte de documentação que um usuário teria para encontrar o que procura. Seja pela navegação do portal, planejada e controlada pela *technical writer* do Store Framework, ou acessando diretamente a documentação de uma *app*, através do padrão de URL `https://vtex.io/docs/app/<nome-da-app>@<versão-da-app>`.

3. *"A comunidade open source deveria ser capaz de contribuir com a documentação"* - Cumprimos este requisito fazendo com que todo o conteúdo de documentação sobre o VTEX IO, não sendo específico para *apps* individuais, fosse colocado em uma *app open source*, a io-documentation. Qualquer usuário da documentação pode contribuir com o seu conteúdo, seguindo o fluxo familiar de contribuições pelo GitHub.

4. *"A solução não deveria impedir a internacionalização do seu conteúdo"* - Apesar de no lançamento do portal de documentação seu conteúdo só estar disponível em inglês, tanto o serviço GraphQL quanto o `docs-ui` são capazes de lidar com internacionalização. Sabendo qual o idioma selecionado no momento, o `docs-ui` faz a *query* correta para o `docs-graphql`, que busca a documentação escrita na linguagem desejada. Do ponto da pessoa que escreve a documentação, sua única preocupação deve ser, dentro de um diretório `/docs`, onde está a documentação da *app*, criar

diretórios individuais para cada idioma. No `io-documentation`, por exemplo, já deixamos o conteúdo dividido por idioma, para facilitar o suporte no futuro.

5. *"Technical writers deveriam ter total poder sobre a estrutura da documentação e como o seu conteúdo é apresentado ao usuário final"*. Também cumprimos este requisito através da criação do `io-documentation`. Através dessa *app technical writers* poderiam controlar a navegação pelo portal e o seu conteúdo.

Apesar de termos cumprido todos os pré-requisitos que definimos, e termos recebido feedbacks positivos, algumas funcionalidades de uma solução completa de documentação não foram entregues no lançamento. Uma funcionalidade que consideramos importante, mas escolhemos não incluir na versão 1.0.0 foi a busca por conteúdo. Decidimos focar nos pré-requisitos que tínhamos definido no começo do projeto antes de priorizar a implementação da funcionalidade de busca. Este seria o mais importante próximo passo do projeto.

Um feedback que recebemos dos desenvolvedores de *apps* do IO, e não antecipamos, é que há casos em que o versionamento da documentação estar em compasso com o versionamento da *app* se torna um incomodo. Quando um desenvolvedor faz mudanças apenas na documentação da sua *app*, sem mudanças no código, ter que lançar uma nova versão da *app* como um todo pode confundir os usuários.

Outro ponto que percebemos posteriormente que poderia ser melhorado é como a estrutura das páginas de listagens de *recipes* e componentes é configurada. Optamos por utilizar arquivos em *JSON*, localizados no `io-documentation`, para configurar essas páginas. Um exemplo desses arquivos pode ser visto em `componentList.json`. O problema que notamos com o tempo é que o formato *JSON*, quando escrito à mão, é muito suscetível a erros. Alguns desses erros causaram problemas no portal de documentação em mais de uma ocorrência.

Ao longo de todo o projeto, conversando com o time do Store Framework, com outros times do VTEX IO e com o time de Technical Solutions, notamos que apenas a existência de um sistema como esse que estávamos desenvolvendo não seria suficiente. Precisávamos introduzir mudanças culturais nos times, para garantir que o esforço sendo feito para criar conteúdo e um sistema de documentação não fossem perdidos. A manutenção da documentação de uma certa *app* deveria ser responsabilidade de todos os desenvolvedores dessa *app*. E a documentação do VTEX IO deveria ser responsabilidade de todos os desenvolvedores da VTEX que usam ou criam o VTEX IO. Do lado técnico, o `docs-bot` foi criado com essa intenção em mente. No próximo capítulo vamos discutir em mais detalhe as mudanças culturais que introduzimos nos times de desenvolvimento, em especial no time do Store Framework.

Capítulo 5

Cultura de documentação: como priorizamos a documentação no desenvolvimento de software

Além do sistema descrito no capítulo anterior para suportar a documentação do VTEX IO, também percebemos ao longo do nosso trabalho com o time do Store Framework que, sem uma cultura que incentivasse a criação e manutenção de documentação, nossos esforços não seria suficientes. Em Dagenais e Robillard, 2010, é levantado que, se a documentação fosse deixada apenas para um time de *technical writers*, separado do time de desenvolvimento, diversos problemas naturalmente aparecem. O mais comum deles é o atraso na produção de documentação. É fácil criar um padrão em que a documentação sempre está tentando se manter atualizada, mas ao mesmo tempo mais código está sendo escrito. Queríamos evitar isso no Store Framework.

Vamos apresentar a base de um modelo para uma cultura de documentação, que busca simplificar os processos envolvendo documentação e mudar a percepção de um time diante destes processos. Acreditamos que a melhor maneira de obter melhores resultados quanto a produção e manutenção de documentação é mudar a maneira com que desenvolvedores pensam sobre estas práticas e até quando elas deveriam ser feitas.

5.1 Problemas a serem considerados

Os problemas que consideramos quando estávamos pensando sobre a cultura de documentação foram os levantados nos capítulos anteriores deste trabalho. Nossos objetivos foram dedicados a resolver os seguintes problemas:

- Assincronia com que mudanças ao código fonte e na documentação acontecem. O que naturalmente resulta em documentação desatualizada, um dos problemas mais comuns que citamos neste trabalho, e incompleta.
- Falta de consistência na organização do conteúdo de documentação produzido. Que faz com que usuários não consigam encontrar o que procuram, mesmo que a solução para seus problemas já tenha sido documentada.

- Complexidade do processo de manutenção e contribuições com o uso de ferramentas desnecessariamente complexas. Consequentemente reduzir o custo associado à tarefas ligadas a documentação, que pode ser um ponto de resistência por parte dos desenvolvedores de um projeto.
- Evitar que o ciclo de feedback citado na sessão 2.4 e descrito por Parnas, 2011 se estabeleça dentro de um projeto de software.
- Incentivar a colaboração da comunidade *open source*.

A descrição das mudanças introduzidas no projeto, e a cultura que descrevemos neste capítulo, foi aplicada no desenvolvimento do Store Framework, um software em grande parte *open source*. Por isso a preocupação com a comunidade de usuários esteve sempre presente.

5.2 Trazendo a documentação para o processo de desenvolvimento

Projetos diferentes podem seguir processos de desenvolvimento muito diferentes, mas vamos considerar o ciclo utilizado pelo time do Store Framework, que se encaixa na maioria dos projetos *open source* modernos:

1. Escrita do código que implementa uma nova funcionalidade;
2. Implementação de testes de unidade, uma vez que o código funciona para casos simples de uso;
3. Submissão do código escrito à revisão por outro membros do time de desenvolvimento, e da comunidade no caso de projetos *open source*. Essa submissão é feita na forma de um *pull request*. É neste estágio que comumente aparecem integrações de *Continuous Integration*(CI), responsáveis por de fato testar o código em ambientes similares ao de produção.
4. Aprovação do código e seu *merge* no código em produção, seguido da distribuição desta nova versão do código. Nesta etapa é comum a presença de integrações que automatizam o processo de distribuição (*Continuous Deployment* ou CD).

O que observamos na grande maioria dos projetos é que, depois do passo final de distribuição do novo código, os desenvolvedores não precisam mais se preocupar com a funcionalidade que implementaram e vão para a próxima. Salvos eventuais *bug fixes*. A documentação do código recém-implementado não é feita junto deste mesmo código, como testes de unidade por exemplo, mas depois que ele já foi revisado e lançado. É comum que a pessoa que documenta uma nova funcionalidade não é a mesma que a desenvolveu, o que pode resultar em perda de informação pela falta de contexto da pessoa documentando.

Diante do ciclo de desenvolvimento que estamos considerando, vemos dois problemas em potencial quanto a documentação: a documentação estará constantemente desatualizada, já que sempre é produzida **depois** do código-fonte; a pessoa que escreveu o código não ser a mesma pessoa que o documenta pode introduzir erros na documentação

ou a insuficiência de informação. Introduzimos uma pequena mudança no processo de desenvolvimento que foi capaz de lidar com ambos os problemas, eliminando-os ou pelo menos reduzindo sua ocorrência.

Nossa mudança no processo de desenvolvimento busca mudar como os desenvolvedores pensar sobre a documentação. Incluindo um passo extra no processo descrito no início desta sessão: "Escrita ou atualização da documentação do software". Parece ser uma mudança sutil no processo, mas este é o passo mais importante de todo o modelo cultural que estamos propondo. A escrita da documentação precisa estar **dentro** do processo de desenvolvimento. Sua posição não precisa estar fixa no processo, podendo ocorrer antes, durante ou até depois da mudança feita, contando que ainda faça parte do processo. O pensamento que acreditamos ser o mais valioso desta cultura é o de que "Nada está pronto se ainda não foi documentado".

5.2.1 Documentação antes da mudança

Vamos considerar o primeiro caso onde incluímos a documentação no processo de desenvolvimento: "documentar antes de desenvolver". Esta abordagem pode parecer contraintuitiva ou até ser vista como uma maneira de tornar o processo de desenvolvimento desnecessariamente lento, tirando sua agilidade. É importante deixar claro que estamos nos referindo a uma documentação simples e efêmera neste primeiro momento, não a produção de documentação final.

Antes de começar a implementar uma nova funcionalidade, o desenvolvedor (ou o time todo) escreve um documento **enxuto**, em um formato fácil de ser editado (como *markdown*) descrevendo a motivação da mudança e alguns detalhes de sua implementação. Esse documento pode ser visto como uma *proposta* de mudança a ser feita no software. Neste cenário, o principal papel desta documentação é de tornar o mais claro possível qual vai ser o resultado final da implementação que o time está pensando. É durante a escrita desde documento de proposta que muitas decisões são tomadas quanto a APIs que outros desenvolvedores irão consumir. Também durante a escrita que ficam mais evidentes quais partes da solução proposta o time ainda não tem clareza, e quais são os possíveis problemas que podem enfrentar. Por fim, esta escrita também tende a forçar os membros do time envolvidos à pensar em soluções alternativas, o que pode salvar horas de trabalho futuro implementando algo que rapidamente seria modificado.

Em projetos de software *open source* a utilidade destes documentos de proposta se mostra ainda maior, sendo uma forma do time *core* do projeto comunicar à comunidade o que pretende fazer. Através desta comunicação, o time recebe feedback de sua comunidade, antes mesmo de terem implementado as mudanças no software. Em Dagenais e Robillard, 2010 é cunhado o termo *embarrassment-driven development* para se referir ao processo de documentar algo antes mesmo de implementar em projetos *open source*. Ao escrever uma documentação em forma de *proposta*, os desenvolvedores do projeto rapidamente identificam problemas na solução e buscam corrigi-los antes de apresentar à comunidade.

Alguns projetos *open source* com grandes comunidades utilizam deste processo de documentação antes de fazer uma mudança buscando o feedback de seus contribuidores por meio de documentos chamados de *RFCs*, ou *Request for comments*. Exemplos de tais

projetos são React (cerca de 1500 colaboradores no repositório do projeto) e Yarn (cerca de 550 colaboradores no repositório do projeto). RFCs são uma solução muito conhecida no desenvolvimento de software aberto, sendo utilizado desde os primeiros desenvolvimentos da Arpanet, como descrito em H. Flanagan, 2019 e Coyle, 2002.

RFCs

RFCs devem ser de fácil consumo pela comunidade *open source* envolvida no projeto, mas também devem ser fáceis de escrever. Com estes dois objetivos em mente, utilizamos *markdown* para escrevê-los.

O primeiro passo foi criar um RFC modelo, que pudéssemos utilizar para descrever qualquer proposta. Assim o time sempre encontra uma estrutura lógica de ideias no início da escrita. Um modelo adequado deve ser feito de acordo com a natureza do software sendo desenvolvido. Segue o modelo [1] que utilizamos como base para os RFCs no Store Framework:

1. **Resumo da ideia**. Uma rápida explicação da alteração a ser feita.
2. **Exemplo básico**. Caso a alteração proposta envolva mudanças ou adição de APIs, esta sessão deve incluir um exemplo de uso simples. Pode ser omitida caso contrário.
3. **Motivação**. O motivo desta mudança, quais problemas ou novos casos de uso são resolvidos por ela. Esta sessão deve ser escrita em detalhes porque caso a proposta do RFC não avance, sua motivação pode ser usada para a criação de soluções alternativas. Com isso em mente, o autor da proposta deve evitar o acoplamento das limitações que está tentando resolver com a solução que está propondo.
4. ***Design* detalhado**. Esta é a sessão mais importante do RFC. É nesta sessão que o autor explica a proposta em detalhes suficientes para alguém já familiarizado com o software entender, e alguém com experiência em contribuir com o código conseguir implementá-la. Aqui devem ser levados em consideração casos mais específicos e de borda. Além de mais exemplos de uso. Qualquer nova terminologia também deve ser apresentada nesta sessão.
5. **Potenciais efeitos negativos (*drawbacks*)**. Nesta sessão é onde o autor deve pensar em quais os potenciais efeitos negativos da proposta. Alguns efeitos a serem considerados: custo de implementação muito elevado, dificuldade para ensinar os usuários do software, como esta alteração se integra com as demais alterações planejadas, etc.
6. **Alternativas**. Descrever brevemente outras soluções que foram consideradas. Caso não seja aplicável, a sessão pode ser omitida.
7. **Como ensinamos aos usuários**. Descrever novos conceitos ou termos que seriam introduzidos, e qual o custo dessas mudanças do ponto de vista de ensinar aos usuários atuais como tirar proveito do que será implementado. Esta sessão deve ser

[1] Baseado nos modelos utilizados pelos projetos: React (biblioteca para o desenvolvimento de interfaces em JavaScript), Ember (*framework full-stack* para o desenvolvimento de aplicações Web com JavaScript), Yarn (gerenciador de pacotes para Node.js) e Rust (linguagem de programação).

analisada e complementada se necessário por um *technical writer* ou alguém que possua uma visão mais abrangente de como o software está documentado.

8. **Questões em aberto**. Esta sessão visa deixar claro para o leitor do RFC quais são os detalhes da proposta que ainda não estão muito bem definidos. É uma ótima maneira de receber opiniões da comunidade inclusive em como resolvê-las.

Uma vez escrito um documento de RFC, este deve ser revisado primeiro pelo time *core* do projeto, o que já servirá como primeira rodada de *feedback*, e então tornado público para a comunidade *open source*. Em cima das ideias apresentadas nele, deve acontecer uma discussão entre o time do projeto e a comunidade, até que cheguem em uma solução que satisfaz ambas as partes. O resultado final é uma documentação prévia da mudança a ser implementada no software. Antes mesmo de ser implementada, esta mudança já está documentada e já foi discutida pela comunidade, esse é o poder do RFC.

Apesar dos benefícios que discutimos até o momento para a prática de se escrever documentação antes de implementar uma mudança, o processo de escrita e discussão de um RFC é custoso e portanto não é adequado em todos os casos. Este modelo é especialmente útil para mudanças grandes, com custo de implementação relativamente alto ou que introduzam novos conceitos aos usuários. O time *core* do projeto deve ser capaz de determinar quando um RFC é necessário ou não. O React, por exemplo, em seu repositório de RFCs recomenda que eles sejam criados apenas para sugerir mudanças "substanciais" ao React ou sua documentação.

5.2.2 Documentação antes do lançamento

Este processo é mais tradicional, onde a documentação é escrita logo após a implementação da mudança no software, mas antes de sua distribuição. Neste cenário, a etapa de escrita da documentação se encaixa entre os passos de "submissão do código para revisão"e o passo de lançamento. É neste caso em que o `docs-bot`, descrito no capítulo anterior, faz o seu papel.

Escrevendo a documentação antes do lançamento e distribuição de mudanças no software sendo desenvolvido, garantimos que usuários não estão utilizando algo que ainda não foi documentado. Também garantimos que a documentação está correta, pois o mesmo desenvolvedor que escreveu o código consegue facilmente escrever ou pelo menos revisar a documentação que será lançada junto à esse código. Idealmente, a documentação seria escrita pelo próprio desenvolvedor e revisada pelos demais membros do time e um *Technical writer*.

A principal dificuldade na implementação deste processo é de que em muitos projetos a documentação não está localizada no mesmo repositório que o código-fonte sendo alterado. Vimos este padrão em alguns dos projetos já citados nesta pesquisa como o Visual Studio Code, o React, o Flutter e o React Native. Esta divisão não é algo ruim, e de certa forma facilita contribuições da comunidade, que não precisa baixar o repositório inteiro de código destes projetos só para corrigir algo na documentação. Mas acreditamos que é neste cenário que a importância de ideias como "nada está pronto enquanto não for documentado" se mostra. O trabalho "extra" por parte do desenvolvedor de ter que fazer modificações em mais de um repositório é justificado pelo benefício de garantir que a documentação do

software se mantém atualizada e correta.

Atualização da documentação se tornou um pré-requisito para aprovação de qualquer mudança no código que a necessite. Ou seja, não devem ser aprovadas e lançadas mudanças no código-fonte sem a devida atualização ou criação de documentação que cubra tal mudança. Em projetos *open source*, essa mesma restrição deve ser imposta para contribuições da comunidade. Como discutido em Dagenais e Robillard, 2010, apesar de ser uma barreira extra à contribuições, os benefícios da prática são sentidos tanto pelo time *core* do projeto quanto os seus usuários.

Este modelo complementa bem os RFCs citados anteriormente. Um RFC em seu estado final é um documento completo o suficiente para servir de base para a documentação a ser escrita *após* a implementação e antes do lançamento da alteração descrita.

5.2.3 Documentação depois do lançamento

Em alguns casos a documentação pode de fato se tornar um fator limitante à entregas do time, dependendo de sua complexidade e nível de detalhe. Especialmente em cenários em que uma mudança é feita em uma parte do software que interage com outras de tal maneira que cria casos de uso complexos para os usuários. Estes casos muitas vezes requerem documentações mais completas e detalhadas, tocando em mais de uma parte do software. Portanto documentações mais demoradas de se escrever.

Nestes casos em que a documentação que deve ser produzida se mostra complexa e muito custosa, ela não precisa ser o fator limitante do lançamento e distribuição da mudança feita no projeto. A mudança em questão pode ser lançada e distribuída, seguindo normalmente o processo de desenvolvimento exemplo dado no início da sessão. Mas a necessidade de sua documentação **não** pode ser esquecida. Podemos considerá-la um passo extra no processo de desenvolvimento, depois da distribuição da nova versão do software mas ainda dentro do mesmo processo. Este é um detalhe importante, pois não queremos que a documentação seja tratada como um processo separado, mas sim uma parte do mesmo processo de desenvolvimento. O processo completo de desenvolvimento e entrega só será considerado completo quando a documentação estiver atualizada de acordo.

O `docs-bot` apresenta aos desenvolvedores uma opção *"I'll do it later."*, que permite ao dono da *pull request* sinalizar que não irá atualizar a documentação naquele momento, mas sim após o lançamento. Nesse caso, o *bot* cria uma *issue* no repositório do projeto no GitHub, lembrando os desenvolvedores de que ainda precisam escrever a documentação.

5.2.4 Mudanças que não necessitam de documentação

É importante considerar também os casos em que alterações na documentação não são necessários. De maneira geral, as mudanças que se enquadram nesta categoria são as que não geram alterações no uso do software ou corrigem problemas de usabilidade. Alguns exemplos são: correções de *bugs*, adição de novos testes e atualização de ferramentas de *linting* e padronização de código em geral.

Quanto ao processo de desenvolvimento no caso destas mudanças, não seria necessária a adição de um passo de documentação. Mas é importante que o desenvolvedor possa

facilmente indicar que considera desnecessária qualquer mudança na documentação do software. Assim, mesmo que não sejam feitas alterações na documentação, o desenvolvedor conscientemente sinaliza que não é necessário. É pensando neste caso que o `docs-bot` também oferece uma opção *"There's nothing new to document"*, sinalizando que as alterações feitas no código não precisam de documentação.

5.3 Distribuição de responsabilidades

Como estamos sugerindo que a documentação faça parte do processo de desenvolvimento e, em muitos casos, seja escrita pelos próprios desenvolvedores do time de um projeto, precisamos levar em conta as responsabilidades que essa mudança trás para um time. Outro ponto é o destaque da importância da presença de *technical writers* como membros dos times de desenvolvimento. A experiência que tivemos com o time do Store Framework, contando com uma *technical writer*, foi muito valiosa e colaborativa. Vamos descrever as responsabilidades que foram delegadas de acordo com os papeis dos membros do time *core* do Store Framework. Com base nos resultados apresentados por Dagenais e Robillard, 2010, não recomendamos a separação do time de desenvolvedores e de *technical writers*, mas sim a sua integração, formando times multidisciplinares como o do Store Framework.

5.3.1 *Technical writers*

Como descrito em Grice, 1997, o trabalho de *technical writers* vem se tornando cada vez mais abrangente com o tempo. Por volta da década de 1960, esses profissionais escreviam em grande parte documentação de produtos de software prontos, que já possuíam um manual técnico escrito pelos seus criadores. Sua principal função era transformar esses manuais técnicos em algo que os usuários finais pudessem consumir. Desde de a década de 1990, a natureza e o escopo do trabalho desses profissionais mudou, se tornando cada vez mais amplo. Hoje, como descrito por Sapir *et al.*, 2016, *technical writers* assumem papeis importantes nos times de produto de empresas de tecnologia, responsáveis por produzir, organizar e revisar a documentação de produtos *durante* o seu desenvolvimento.

Technical writers são essenciais na implantação da cultura de documentação que estamos descrevendo neste capítulo. É sua responsabilidade a organização e revisão da documentação sendo produzida tanto pelo time *core* quanto pela comunidade *open source*. Esses profissionais devem conhecer a documentação do projeto como um todo, sendo capazes de garantir sua consistência, o que resulta em uma melhor experiência para o usuário final. Além disso, *technical writers* também devem ditar os padrões a serem seguidos na produção da documentação, em seu momento inicial. Uma das formas de se fazer isso é através da criação de modelos a serem seguidos para cada tipo de documentação a ser escrita. Esses modelos então devem ser seguidos por todos os contribuidores do projeto.

No Store Frmework, a *technical writer* do time criou modelos para a documentação de *apps* e também para *pull requests* do time e da comunidade. Estes modelos tiveram um papel crítico em tornar toda documentação produzida pelo time mais uniforme, além de introduzir uma estrutura que se tornou familiar para os desenvolvedores e para os usuários do *framework*.

5.3.2 Desenvolvedores

A partir do momento em que a documentação começa a fazer parte do processo de desenvolvimento, um desenvolvedor não é mais responsável apenas por implementar funcionalidades, mas também documentá-las. Esta é a principal responsabilidade dos desenvolvedores na cultura que introduzimos: documentar o que está sendo desenvolvido. Os desenvolvedores devem fornecer aos *technical writers* ao menos a base de uma documentação com que eles possam trabalhar.

Desenvolvedores também devem estar disponíveis para ajudar *technical writers* a entender de fato o funcionamento e o impacto do que desenvolveram. Em geral, desenvolvedores do time terão mais contexto do que os *technical writers* sobre o que eles mesmo implementaram, e a documentação escrita por eles pode não estar clara o suficiente. É importante que os desenvolvedores estejam abertos a colaborar com os *technical writers* para que a documentação resultante sempre esteja correta e completa o suficiente para seus leitores. No Store Framework por exemplo, a *technical writer* do time não possuía formação técnica, então era de extrema importância que os desenvolvedores do time à ajudassem a entender mudanças que exigissem mais contexto técnico para serem documentadas.

Outra responsabilidade de cada desenvolvedor do time é a manutenção da documentação que ele mesmo escreveu. De maneira análoga a implementação da funcionalidade pelo código, o desenvolvedor que escreveu a documentação possui mais contexto sobre o seu conteúdo. Por isso, ao serem encontrados erros ou problemas com parte dessa documentação, o próprio desenvolvedor é responsável por resolvê-los ou ao menos ajudar o *technical writer* do time a resolver.

5.4 Facilitando a manutenção

Dado que estamos propondo uma cultura de desenvolvimento que dá a devida importância à documentação de software, mas ao mesmo tempo aumenta a responsabilidade dos membros do time *core* do projeto, precisamos abordar uma maneira de facilitar o trabalho de criação e manutenção proposto. Este modelo que descrevemos neste capítulo até então não foi testado em isolamento, mas sim aliado ao sistema de documentação descrito no capítulo anterior.

É importante que o time *core* sinta-se confortável em iterar rapidamente sobre a documentação do projeto, com ferramentas simples focadas no **conteúdo** e não na sua forma ou organização. Como vimos no capítulo 3, uma solução comum em projetos *open source* é a utilização de *Markdown* para escrita da documentação, independente de como esse conteúdo é apresentado aos usuários finais. O amplo suporte de ferramentas e existência de bibliotecas para *front-end* capazes de transformar *markdown* em páginas HTML pode justificar essa ampla adoção. Além disso, *markdown* é suportado pelos principais clientes de Git do mercado—GitHub, GitLab e Bitbucket—em *README*s de repositórios até discussões e comentários.

Além de uma linguagem de escrita simples e amplamente conhecida, é importante que o fluxo de edição e criação de nova documentação não esteja distante do fluxo de desenvolvimento. Como discutido neste capítulo, queremos trazer a documentação para o

processo de desenvolvimento, e portanto não devemos tornar esse processo mais custoso do que o necessário. Com isso em mente, e a necessidade de manter um registro de mudanças na documentação e conectá-las ao código, acreditamos que o uso do mesmo controle de versão utilizado no código-fonte do projeto deve ser o utilizado na documentação. Assim o trabalho "extra" de documentar o que está sendo implementado está apenas na produção do conteúdo, e não em processo desnecessariamente complicados.

Nosso sistema de documentação, composto pelo Docs Builder, Docs GraphQL e Docs UI, tornou a implementação dessa cultura de documentação possível. Promovendo a colaboração entre *technical writers* e desenvolvedores, e fornecendo a eles um sistema que permitisse a iteração rápida da documentação, sem grandes desvios do ciclo já familiar de desenvolvimento.

5.5 Participação da comunidade *open source*

Da mesma maneira que projetos *open source* se alavancam da sua comunidade para o seu crescimento e implementação de funcionalidades, essa mesma comunidade pode estar engajada na manutenção da documentação do projeto. Cabe ao time *core* do projeto possibilitar que a comunidade assuma esse papel. O caminho que seguimos com o Store Framework para incentivar a participação da comunidade foi ampliar a **comunicação** entre o time *core* e os times que utilizam o *framework* para desenvolver lojas no ecommerce da VTEX.

Implementamos duas novas formas de comunicação com a comunidade *open source* que foram muito bem recebidas: *release notes* e um fórum no GitHub para discussões sobre o *framework*, chamado de Store Discussion. Os *release notes* foram uma iniciativa liderada pela *technical writer* e pela *product manager* do Store Framework. Eles servem como atualizações mensais para a comunidade sobre mudanças e novas funcionalidades que o time *core* implementou ao longo do mês anterior. Um exemplo pode ser encontrado no *release notes* de Outubro de 2020. Já o Store Discussion, foi criado para ser um espaço aberto de discussão e colaboração da comunidade com o time do Store Framework. Ele é aberto e se encontra em Store Discussion.

Como mostrado em Dagenais e Robillard, 2010, diferentes soluções de documentação que permitem contribuições por parte da comunidade oferecem diversos pontos negativos e positivos. A escolha de como permitir que a comunidade *open source* contribua com a documentação de um projeto deve levar em consideração qual a barreira inicial de uma contribuição *versus* o controle sobre o conteúdo que o time *core* deseja. Um resultado interessante da pesquisa em questão foi que **todos** os projetos analisados que começaram sua documentação através de uma *wiki*, acabaram desistindo desse formato com o crescimento da comunidade. Isso pode ser explicado pela necessidade de maior controle do conteúdo que estava sendo publicado. Uma *wiki* possui uma barreira de contribuição baixa para qualquer membro da comunidade de usuários do projeto, mas abre mão desse controle por parte do time *core*, em especial dos *technical writers* do time.

Buscando um equilíbrio entre a dificuldade de contribuições por parte da comunidade e o controle sobre o conteúdo que o time *core* do projeto precisa, tratamos contribuições à documentação da mesma maneira que contribuições de código. A documentação é colocada

em repositórios públicos, e contribuições devem ser feitas por meio de *pull requests*, onde podem ser revisadas pelos *technical writers*, desenvolvedores do projeto e outros membros da comunidade. Aghajani *et al.*, 2019 e Dagenais e Robillard, 2010 apontam como obstáculo para contribuições da comunidade na documentação a complexidade associada ao processo, seja ela causada por ferramentas ou por falta de clareza deste processo. Esta foi a nossa justificativa para seguir o fluxo de contribuições via *pull requests*.

Capítulo 6

Conclusões

Neste trabalho, nossos objetivos eram: entender o estado atual da documentação de software para desenvolvedores e como ela é vista por desenvolvedores de software, por meio de uma revisão bibliográfica sobre o assunto; estudar como projetos *open source* de grande escala lidam com a sua documentação; e por fim aplicar o que aprendemos em um projeto real, no contexto de uma empresa de software.

Com a revisão bibliográfica conduzida no Capítulo 2, notamos que as pesquisas existentes sobre a documentação de software de maneira geral apontavam para uma série de problemas generalizados, que se mostravam comuns em muitos projetos diferentes. Lethbridge *et al.*, 2003 mostrou que a documentação era vista como a principal forma de comunicação de informações sobre um dado software, mesmo estando desatualizada ou inconsistente. Resultado que foi também obtido por Plosch *et al.*, 2014, além de também mostrar que desenvolvedores tendem a superestimar a quantidade de tempo que eles passam de fato utilizando documentação. Resultados como estes nos levam a acreditar que desenvolvedores de software valorizam a documentação e a consideram importante.

Apesar dessa percepção positiva da documentação que consomem, na prática foram observados diversos problemas em documentações de diferentes softwares, pequenos ou grandes, *open source* ou não. Plosch *et al.*, 2014 chega a citar que, apesar de consumirem documentação e considerá-la importante, desenvolvedores se mostram pouco dispostos a produzi-la. Trabalhos como Satish e Anand, 2016 e Aghajani *et al.*, 2019 nos fornecem uma visão clara sobre os problemas mais comuns encontrados em documentações de diferentes softwares, *open source* ou não. Alguns dos problemas mais comuns são: documentação desatualizada; falta de uma estrutura da documentação como um todo, que torna fácil a busca por um determinado conteúdo; falta de um padrão entre os diferentes documentos que compõem a documentação; falta de entusiasmo ou motivação por parte dos desenvolvedores, que muitas vezes são as pessoas mais qualificadas para escrever a documentação.

No Capítulo 3, apresentamos nossas observações sobre como os 10 projetos da lista *Top and trending projects*[1] do GitHub de 2019 lidam com as duas documentações.Encontramos notáveis similaridades entre os projetos, como: o uso de linguagens de *markup* simples,

[1]The State of the Octoverse - GitHub: https://octoverse.github.com/

Markdown em 8 deles, repositórios de conteúdo *open source*, onde a comunidade de usuários poderia facilmente colaborar com a documentação por meio de *Pull requests*; documentação acessível através de um portal que consolidava todo o seu conteúdo, apresentando-o de maneira organizada.

Por fim, nos capítulos 4 e 5 apresentamos nossas experiências práticas, buscando aplicar o que aprendemos com a etapa de pesquisa do trabalho em um projeto *open source* real. Nossa experiência prática foi dividida na implementação de um sistema de software para lidar com a documentação do projeto e em uma série de mudanças culturais com ao lado do time com que estávamos trabalhando.

Durante a implementação do sistema, que incluiu a criação do portal VTEX IO Docs, conseguimos elencar os principais problemas que queríamos resolver ou evitar com a documentação do VTEX IO. Ao final do trabalho, com o lançamento e apresentação da versão 1.0.0 do portal, conseguimos satisfazer todos os objetivos principais que tínhamos, como: dar controle da estrutura do conteúdo para *technical writers*; permitir que desenvolvedores conseguissem produzir documentação sem precisar de grandes alterações nos seus fluxos de desenvolvimento; permitir que a documentação mantivesse a sua estrutura distribuída, entre outros.

Ao longo do processo de desenvolvimento e trabalho junto ao time do Store Framework, notamos que precisaríamos implementar mudanças culturais no time para garantir que o esforço feito para melhorar a documentação não fosse perdido. Nossa principal preocupação era evitar o ciclo descrito por Parnas, 2011, e conseguimos resultados satisfatórios no final do projeto. A cultura de documentação implementada no Store Framework chegou a ser apresentada para os outros times da VTEX e recebeu *feedbacks* positivos do time de *Education* da empresa.

Próximos passos

Com o passar do tempo, e com mais desenvolvedores utilizando o sistema que criamos para documentar suas *apps* do VTEX IO, recebemos algumas sugestões de melhorias para a experiência que criamos. O ponto mais citado é que, como a nossa solução faz com que a documentação seja versionada junto ao código da *app*, desenvolvedores sempre precisam lançar novas versões de suas *apps*, mesmo que só tenham feito alterações na documentação. Acreditamos que esse problema poderia ser resolvido com a evolução da nossa solução.

Outro ponto que poderia ser melhorado, visando facilitar o trabalho de usuários menos técnicos, é fornecer uma interface que abstraia o fluxo do Git e de lançamento da *app* que contém a documentação conceitual do VTEX IO.

Por fim, uma melhoria que beneficiaria tanto *technical writers* como desenvolvedores é a implementação de uma solução simples de *preview*. Atualmente, a pessoa escrevendo um arquivo para documentação, que estará no portal, não consegue ver este arquivo em nenhum ambiente de testes sendo renderizado no portal. A documentação é publicada, e só então a pessoa que a criou pode vê-la no portal.

Referências

[Aberdour 2007] Mark Aberdour. *Achieving quality in open-source software*. Jan. de 2007. doi: 10.1109/MS.2007.2 (citado na pg. 10).

[Aghajani *et al.* 2019] Emad Aghajani *et al.* "Software Documentation Issues Unveiled". Em: *Proceedings - International Conference on Software Engineering*. Vol. 2019-May. IEEE Computer Society, mai. de 2019, pgs. 1199–1210. isbn: 9781728108698. doi: 10.1109/ICSE.2019.00122 (citado nas pgs. 5, 21, 46, 47).

[Anthes 2016] Gary Anthes. "Open source software no longer optional". Em: *Communications of the ACM* 59.8 (ago. de 2016), pgs. 15–17. issn: 15577317. doi: 10.1145/2949684. url: https://dl.acm.org/doi/10.1145/2949684 (citado nas pgs. 1, 9).

[Bayati 2018] Sharab Bayati. "Poster: Understanding Newcomers Success in Open Source Community". Em: *2018 IEEE/ACM 40th International Conference on Software Engineering*. 2018, pgs. 224–225 (citado na pg. 10).

[Berglund e Priestley 2001] Erik Berglund e Michael Priestley. "Open-source documentation". Em: *Proceedings of the 19th annual international conference on Computer documentation - SIGDOC '01*. New York, New York, USA: Association for Computing Machinery (ACM), 2001, pg. 132. doi: 10.1145/501516.501543. url: http://portal.acm.org/citation.cfm?doid=501516.501543 (citado na pg. 10).

[Coyle 2002] Karen Coyle. "Open source, open standards". Em: *Information Technology and Libraries* 21.1 (mar. de 2002), pgs. 33–36. url: https://search.proquest.com/docview/215833413/abstract/93169E0E05E4CF8PQ/1?accountid=14643 (citado na pg. 40).

[Dagenais e Robillard 2010] Barthélémy Dagenais e Martin P. Robillard. "Creating and evolving developer documentation: Understanding the decisions of open source contributors". Em: *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. New York, New York, USA: ACM Press, 2010, pgs. 127–136. isbn: 9781605587912. doi: 10.1145/1882291.1882312. url: http://portal.acm.org/citation.cfm?doid=1882291.1882312 (citado nas pgs. 6, 7, 10, 21, 32, 37, 39, 42, 43, 45, 46).

[Forward e Lethbridge 2002] Andrew Forward e Timothy C. Lethbridge. "The relevance of software documentation, tools and technologies". Em: *Proceedings of the 2002 ACM symposium on Document engineering - DocEng '02*. New York, New York, USA: ACM Press, 2002, pg. 26. isbn: 1581135947. doi: 10.1145/585058.585065. url: http://portal.acm.org/citation.cfm?doid=585058.585065 (citado nas pgs. 1, 7).

[Fowler *et al.* 2001] Martin Fowler *et al. Manifesto para Desenvolvimento Ágil de Software*. 2001. url: https://agilemanifesto.org/iso/ptbr/manifesto.html (citado na pg. 8).

[Garousi *et al.* 2013] Golara Garousi, Vahid Garousi, Mahmoud Moussavi, Guenther Ruhe e Brian Smith. "Evaluating usage and quality of technical software documentation: An empirical study". Em: *ACM International Conference Proceeding Series*. New York, New York, USA: ACM Press, 2013, pgs. 24–35. isbn: 9781450318488. doi: 10.1145/2460999.2461003. url: http://dl.acm.org/citation.cfm?doid=2460999.2461003 (citado nas pgs. 1, 5).

[Grice 1997] Roger A Grice. "Professional roles: Technical writer". Em: *Foundations for teaching technical communication: Theory, practice, and program design* (1997), pgs. 209–220 (citado na pg. 43).

[H. Flanagan 2019] Ed. H. Flanagan. *Fifty Years of RFCs*. Rel. técn. Dez. de 2019. doi: 10.17487/RFC8700. url: https://www.rfc-editor.org/info/rfc8700 (citado na pg. 40).

[Lethbridge *et al.* 2003] Timothy C. Lethbridge, Janice Singer e Andrew Forward. *How Software Engineers Use Documentation: The State of the Practice*. Nov. de 2003. doi: 10.1109/MS.2003.1241364 (citado nas pgs. 1, 4, 5, 47).

[Parnas 2011] David Lorge Parnas. "Precise documentation: The key to better software". Em: *The Future of Software Engineering*. Springer Berlin Heidelberg, 2011, pgs. 125–148. isbn: 9783642151866. doi: 10.1007/978-3-642-15187-3{_}8. url: https://link.springer.com/chapter/10.1007/978-3-642-15187-3_8 (citado nas pgs. 6, 38, 48).

[Plosch *et al.* 2014] Reinhold Plosch, Andreas Dautovic e Matthias Saft. "The value of software documentation quality". Em: *Proceedings - International Conference on Quality Software*. IEEE Computer Society, nov. de 2014, pgs. 333–342. isbn: 9781479971978. doi: 10.1109/QSIC.2014.22 (citado nas pgs. 1, 3, 4, 6, 47).

[Sapir *et al.* 2016] Adi Sapir, Israel Drori e Shmuel Ellis. "The Practices of Knowledge Creation: Collaboration Between Peripheral and Core Occupational Communities". Em: *European Management Review* 13.1 (mar. de 2016), pgs. 19–36. issn: 17404754. doi: 10.1111/emre.12064. url: http://doi.wiley.com/10.1111/emre.12064 (citado nas pgs. 26, 32, 43).

[Satish e Anand 2016] C. J. Satish e M. Anand. “Software documentation management issues and practices: A survey”. Em: *Indian Journal of Science and Technology* 9.20 (mai. de 2016). issn: 09745645. doi: 10.17485/ijst/2016/v9i20/86869 (citado nas pgs. 5, 8, 47).

[Schreck *et al.* 2007] Daniel Schreck, Valentin Dallmeier e Thomas Zimmermann. “How documentation evolves over time”. Em: *International Workshop on Principles of Software Evolution (IWPSE)*. New York, New York, USA: ACM Press, 2007, pgs. 4–10. isbn: 9781595937223. doi: 10.1145/1294948.1294952. url: http://portal.acm.org/citation.cfm?doid=1294948.1294952 (citado na pg. 6).

[Sommerville *et al.* 2005] Ian Sommerville, Richard H. Thayer e M. I. Christensen. “Software Documentation”. Em: *Software Engineering, vol 2: The supporting Processes*. Wiley-IEEE Press, 2005. isbn: 978-0985270711 (citado na pg. 3).