探索微前端：一个电子商务中的案例研究应用

Ricardo Hideki Hangai Kojo[1][0009−0007−5614−9671],
Luiz Fernando Corte Real[1],
Renato Cordeiro Ferreira[1,2][0000−0001−7296−7091],
Thatiane de Oliveira Rosa[1,3][0000−0002−3980−0051], and
Alfredo Goldman[1][0000−0001−5746−4154]

[1] Instituto de Matemática e Estatística (IME), University of São Paulo (USP), Brazil
[2] Jheronimus Academy of Data Science (JADS), Tilburg University (TiU) and Technical University of Eindhoven (TUe), The Netherlands
[3] Federal Institute of Tocantins (IFTO), Brazil
[4] {renatocf,gold}@ime.usp.br
thatiane@ifto.edu.br

摘要 在微前端架构风格中，前端被划分为更小的组件，这些组件可以从一个简单的按钮到整个页面不等。目标是提高可扩展性、弹性和团队独立性，尽管这会增加复杂性和基础设施需求。本文旨在了解何时采用微前端是值得的，尤其是在工业背景下。为此，我们基于学术和灰色文献对微前端的最新状态进行了调查。然后我们在一个使用微服务的手工艺品市场中实现了这种架构风格。最后，我们通过一份半开放问卷对开发人员进行了实施评估。在被研究的市场公司中，由于其主系统（Java 单体）与专用前端系统之间的紧密耦合，需要进行架构变更。此外，还有废弃的技术和较差的开发者体验。为了解决这些问题，该公司采用了微前端架构以及 API 网关和后端用于前端 模式，并使用了 Svelte 和 Fastify 等技术。虽然采用微前端是成功的，但这并不是严格需要来满足公司的需求。根据混合问卷调查结果的分析，其他替代方案，如单体前端，可以实现类似的结果。使公司在其背景下选择微前端最便捷的原因是通过基础设施重用和团队间知识共享简化了实施过程，并且有利于单体系统剥离和采用微服务。

Keywords: 微前端 · 软件架构 · 案例研究 · 实验软件工程

1 介绍

微服务架构大约在 2012 年 [7] 出现，受到了学术界和工业界的广泛欢迎，并被亚马逊、Netflix 和 Uber 等公司采纳 [12] 。这种架构将应用程序分解为多个小的独立服务，这些服务协同工作。这些服务之间的松耦合带来了可维护性、可扩展性和弹性等优点，这些都是大型系统需要具备的关键特性，要求系统可靠、始终可用，并且尽量减少故障发生的可能性。然而，这种方法仅关注应用的后端部分，而忽略了前端 [10] 。

在此背景下，微前端架构应运而生，旨在将微服务的概念应用于应用程序的展示层 [10] 。在这种方法中，前端系统被划分为小的部分，这些部分可能从一个简单的按钮到一个完整的页面不等。这些部分可以在客户端（例如，在用户的浏览器中）、边缘端（例如，CDN 或内容分发网络）或服务器端（例如，云服务器）进行组合，最终产生统一且一致的输出 [9] 。

微服务和微前端之间存在显著的重叠利益与挑战。因此，考虑同时采用这两种架构是合理的，可以利用现有的知识和技术基础。然而，微前端的概念相对较新（2016 年引入 [4]），并且在该主题上的学术研究尚且不足。与受益于成熟技术和设计模式的微服务不同，微前端仍存在工具支持有限的问题。因此，许多探索的机会仍然有待发掘。

因此，本研究旨在确定采用微前端架构是否值得，特别是在软件行业背景下。为此，开展了理论和实践两方面的活动。

为了实现这一目标，第一步是通过回顾学术文献（如文章和书籍）以及灰色文献（包括博客文章和录制的演讲），来了解这种架构的现状。通过批判性分析，能够识别出最全面的知识来源、该主题的主要作者以及现有材料中的空白。除了文献综述外，微前端架构还在 A 公司（虚构名称）进行了实施，该公司是一个已经使用微服务的市场平台。这一目标还包括记录和评估实施情况：理解架构变更的原因、概述开发历史、包括新系统，并解释所做决策的理由。最后，通过向参与团队成员发放问卷来分析采用微前端的影响。

2 背景

本节旨在介绍理解本研究所需的基本概念，假设读者具备软件架构和架构风格的基础知识，尤其是微前端。

2.1 软件架构与架构风格

软件架构可以定义为“需要推理系统的一组结构，其中包括软件元素、它们之间的关系以及两者的属性”[1]。它涵盖了指导系统设计的基础、属性、规则和约束。

定义软件架构的目的是为了便于开发、部署、操作和维护 [8]，此外还支持可扩展性、弹性、安全性和可维护性等定性属性的分析。因此，架构决策对系统的演化至关重要。为了识别最合适的架构来解决特定的质量属性，采用了各种架构风格和模式 [11]。

一种架构风格规定了一组受限的元素和关系，可以用来定义系统的结构 [11]。符合这些约束条件的系统可以被视为该架构风格的一个实例。需要注意的是，一个单一的应用程序可能会采用多种风格。

在选择特定的架构风格时，必须考虑决策背后的动机、预期的好处以及潜在的挑战。架构决策往往具有重大影响，并且正如 Brooks 著名所说 [2]，“软件开发中没有一劳永逸的解决方案”；没有任何单一的架构风格能够提供普遍的解决方案或只有优势。

本研究探讨了单体式、基于微服务的以及微前端架构风格，这些特别有助于理解所提出的科研。

2.2 整体式与基于微服务的架构风格

在单体架构中，整个应用程序被构建成一个包含所有必要逻辑的单一层次，以实现系统的全部功能，通常会生成一个单独的可执行文件。这种架构较为简单，因此常作为新项目的选择。然而，随着团队和代码库的增长，可能会出现技术及组织问题，如维护困难、bug 数量增加、特性开发缓慢以及扩展性限制等问题。在这种情况下，微服务架构可能成为一个可行的替代方案 [14]。

根据 Lewis 和 Fowler [7], “微服务架构风格是一种开发单个应用程序的方法，该方法将应用程序视为一组小型服务，每个服务在其自身的进程中运行，并通过轻量级机制进行通信，通常是 *HTTP* 资源 *API*。这些服务围绕业务能力构建，并可通过完全自动化的部署工具独立部署。这些服务的集中管理程度极低，可能使用不同的编程语言编写，并采用不同的数据存储技术。”

在文献中，微服务常被描述为一种替代单体系统的方案，而单体系统面临组织和技术上的限制。从单体到微服务的过渡可以遵循 STRANGLER FIG 模式 [3]，在这个模式中，现有的功能会被提取成独立的服务，直到单体变得

过时并被完全取代。这一过程是渐进的，因为对单体进行全面重写通常由于其规模和复杂性而不可行。

2.3 微前端架构

微前端是“一种架构风格，其中独立可交付的前端应用程序被组合成一个更大的整体”[6]。应用程序被分解为更小、更简单的部分，这些部分可以独立开发、测试和部署，同时保持统一的用户体验。

根据 Peltonen 等人。[10]，“微前端扩展了微服务架构的思想，并且微服务中的许多原则也适用于微前端。”。因此，通常会并行实现这两种架构。由于后端和前端是不同的领域，现有的分布式系统知识和基础设施可以被充分利用而不至于服务之间产生冲突。出于这些原因，在采用微服务之后，微前端被认为是自然而然的下一步 [15]。

Peltonen 等。[10] 和 Geers [4] 识别了采用微前端架构的几个动机。这些包括前端系统的复杂性增加、代码库的增长以及组织挑战。可扩展性也是一个反复出现的问题，特别是关于独立部署和团队的需求、更快的功能交付、创新支持以及减少新开发人员的入职时间。在某些情况下，采用还受到市场趋势的影响，公司采纳微服务和微前端以符合行业最佳实践。

此外，微前端提供了类似于微服务的好处。这些包括技术灵活性，允许系统每一部分使用最合适的工具，并能够形成专注于特定业务领域的跨职能团队。这种架构促进了团队和项目的自主性，便于开发、测试和持续交付。它还支持在应用程序和组织层面的可扩展性。另一个相关方面是韧性：微前端中的故障仅影响那个服务，避免了单体架构中常见的全系统宕机。

关于采用微前端的挑战，最重要的包括由架构分布性质引入的复杂性，这涉及多个项目和多种技术。确保服务间的视觉一致性也至关重要，因为最终产品必须对用户呈现一致且统一的外观。治理构成了另一个挑战，尤其是在不同团队共同负责同一服务或在同一业务领域内运营时。最后，性能问题必须得到解决，由于潜在的代码重复、共享依赖以及接口组合过程中传输的数据量 [10,4]。

在实现微前端之前，需要定义将采用哪种方法。主要有两种方法：水平方法，其中每个微前端代表页面的一部分；垂直方法，其中每个微前端对应一个业务上下文或领域。在后者中，单个微前端可能包含页面的部分、整个页面，甚至多个页面 [9]。

在实现微前端架构时需要考虑的另一个重要方面是采用架构模式。在这种情况下，最相关的模式是 API 网关 和后端用于前端（BFF）。

API 网关 是一个作为其他服务入口点的服务。所有客户端（如移动应用或桌面网站）发出的请求都会经过网关，网关除了将这些请求路由到适当的服务外，还可以处理授权检查和协议转换。这种模式在基于微服务的架构中特别有用，在这种架构中，多个客户端访问各种服务。与每个服务的通信可以通过不同的协议进行，如 HTTP 和 gRPC[13]。

后端用于前端（BFF）模式是 API 网关 模式 [13] 的一种变体，其中每种客户端类型都有自己的专用网关。这允许特定客户端需求的隔离，并使一个团队能够对客户端应用程序及其对应的 BFF 承担责任。

3 架构概述 Company A

作为电子商务平台，前端是 **Company A**（虚构名称）成功的关键要素。网站必须具有表现力、用户友好，并且能够在桌面和移动设备上访问。在采用微前端之前，**Company A** 遵循了 STRANGLER FIG 模式过程来构建其单体系统，这里称为 `Marketplace` 。虽然一些后端功能已经迁移到独立的微服务中，但前端仍然集中在单体中。此外，某些 UI 组件由 `Aquarelle`（虚构名称）提供，这是一个为满足特定需求而创建的单独项目。

`Marketplace` ，于 2012 年启动，负责后端和前端服务，并作为所有系统的单一入口点，因此成为单点故障。使用 Java、JSP、Sass 以及内部开发的 JavaScript 库构建的单体架构促进了早期的快速开发。然而，随着时间推移，由于复杂性增加、部署时间长和架构退化，它变成了瓶颈。作为 STRANGLER FIG 结为绞杀榕 模式的一部分，产品搜索、运费计算和类别管理等核心功能被迁移到了微服务。

`Aquarelle` 基于 Node.js，出现以支持买家和卖家之间的一种新聊天功能，需要一个能够显示动态后端数据（如订单状态和可用用户操作）的响应式界面。由于现代框架在 2016 年初不可用，解决方案涉及开发内部库。

尽管最初取得了成功，`Aquarelle` 面临了技术与架构挑战。其关键库之一在发布后不久就被弃用，导致维护困难、缺乏文档以及由于技术过时而减少了开发者兴趣。此外，开发限制，如有限的 Node.js 经验和不足以解耦特性的时间，导致与 `Marketplace` 高度耦合。

该单体结构仍负责认证、数据编排和格式化，迫使前端开发者与后端代码进行交互。这种僵化的架构加上使用过时的技术，最终导致了采用微前端架构的决定。

4 在 Company A 实现微前端

由于上一节讨论的问题，制定了一个基于采用现代技术的新前端架构实施计划。选定的架构是微前端，这有助于职责的隔离，允许将界面分割成更小的部分，并能够使用不同的技术。为此，需要创建一个充当 API 网关 的服务，以便单体不再作为中央入口点，并且仅在需要时才被访问。基于此计划，创建了项目 `Fortelle` 、`Veloura` 、`Nuvelle` 和 `Brindle`（虚构名称），每个项目都在构建新架构中发挥着战略作用。

`Fortelle` 是 API 网关 模式的实现，作为 **Company A** 系统的入口点。所有用户请求都会通过它，在这里处理路由、验证和安全功能等其他服务。此实现对于启用微前端接管之前集中在单体中的用户界面服务职责至关重要。同时，团队开始评估框架和库以确保良好的性能和积极的开发者体验。

`Veloura` 是 **Company A** 的设计系统，包括一套构建用户界面的标准——包括颜色、间距、字体和图标——这些都强化了公司的视觉形象 [5]。`Nuvelle` 项目最初是作为选定前端技术的证明概念而构思的。其开发使团队能够验证诸如文件组织、支持库以及代码约定等实践。从 `Nuvelle` 起，首个 `Brindle` 被创建，目标是从单体应用迁移到新架构的一个特定手工艺材料搜索页面。此页面被选中是因为它在视觉上与主搜索页相似，但流量较低。

在将第一个 `Brindle` 部署到生产环境后，开发了新的克隆以满足内部需求。所有源自 `Nuvelle` 的系统都被称为 `Brindle` 项目，并使用额外的名字来区分它们，例如 `Brindle Discovery` 。目前，**Company A** 的所有微前端都是 `Brindle` 项目。Figure 1表明了微前端的基本结构：用户请求通过 API 网关 并被重定向到一个 `Brindle` 实例，在那里后端用于前端（BFF）模式处理内部路由，编排来自微服务的数据，并将其转发给由 **Company A** 开发的开源库渲染的模板。

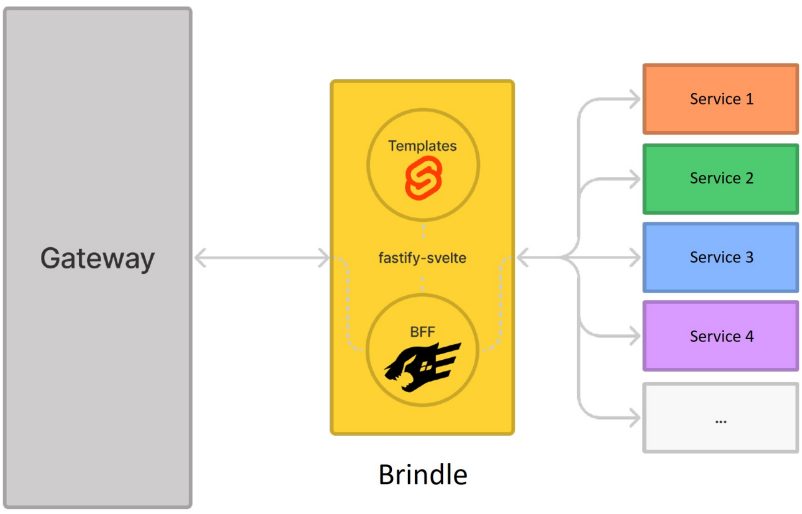

图 1. Brindle 项目的建筑设计草图。

随着三个 Brindle 项目的创建，前端团队能够识别出跨服务的通用模块。这些共享组件被提取到一个新的库 Brindle Core 中，这是一个用于构建 Brindles 的开源框架。Brindle Core 的主要优势之一是它能够抽象化内部机制，而这些机制很少发生变化，从而减少了代码重复、文件数量以及在通用组件中的重复测试需求。

5 当前架构的 Company A

随着提到的项目的概念提出，公司的前端架构经历了重大变化。引入了 API 网关 模式——如 Figure 2所示——使得除了单体之外的其他服务也能处理部分用户请求处理，分散了责任，并使微前端的采用成为可能。因此，创建新的 Brindle 项目变得简单，只需要对 API 网关 代码进行少量添加。

架构转型在 **Company A** 证明是成功的。采用垂直微前端允许将页面从单体迁移至 Brindle 项目，这些项目使用现代技术并提供改进的开发体验。此外，新结构促进了公司新产品前端系统的创建。值得注意的是，向微服务转型的趋势有利于这种采纳，因为它使得可以重用已有的技术和基础设施，因为微前端也共享了微服务的原则。

在此研究中，观察到架构的自然演进涉及将更多页面从单体迁移到新服务，并创建额外的微前端。然而，没有计划扩展服务的数量，因为

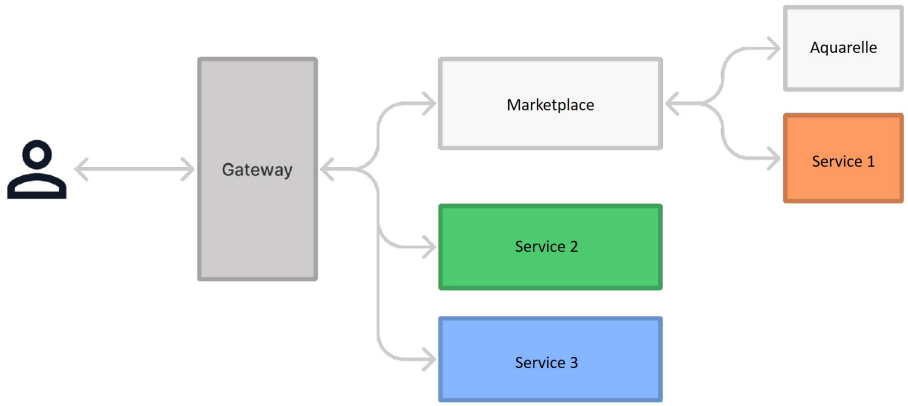

图 2. 实现微前端后请求流到前端。

Company A 已基于其产品定义了一个领域划分。因此，在构思新产品之前，团队仍专注于改进现有服务。

然而，存在一个瓶颈：要将更多页面迁移到 `Brindle` 项目中，必须将单体中的功能转换为微服务。由于后端团队目前没有可用性来开发这些服务，获取接口所需的必要数据受到损害。为了缓解这一问题，后端和前端团队必须建立契约，允许模拟对未来服务的调用并推进开发。尽管采用了微服务和微前端，**Company A** 仍在评估跨学科团队的实施情况，考虑组织影响，并质疑这种做法是否总是有益，正如文献中所讨论的。

6 开发者调查

为了更好地了解架构变更带来的影响，一份问卷被应用于直接或间接参与前端工作的 **Company A** 的员工。本节的目的是展示所获得的结果。

本研究的一个目标是评估公司实施的微前端。从这个意义上说，转向直接参与其中的人士——开发者是合理的。因此，准备了一份半开放式的问卷，以识别他们面临的问题、对变化的期望以及受访者所感知的影响。

问卷设计的目的是尽可能地将采用新技术的影响与新架构带来的影响分开。由于这两种复杂的转型同时发生，预计开发者在没有这种方法论预防措施的情况下可能会难以区分他们的回答。制定了十九个问题，其中十五个是开放式问题，四个是多项选择题。对于多项选择题，使用了李克特量表。

这些问题被分为五个部分。第 1 部分涉及受访者的信息，如专业经验以及与 **Company A** 的前端相关的工作。第 2 部分涵盖了在采用微前端之前已有的架构。第 3 部分和第 4 部分分别关注新技术的使用和架构变更。最后，第 5 部分旨在了解参与者对新架构的理解。

多项选择题要求受访者考虑技术与建筑影响的七个方面：招聘和入职；部署；跨多个团队的发展；项目理解；新功能的实施；可测试性；以及开发速度。这些方面之所以被选中，是因为它们与公司相关，并且在审查过的文献中经常出现。

总共获得了八个回应。只有一名女性回答了问卷。大多数参与者（62.5%）在科技领域有超过 10 年的经验。一半的人担任前端开发人员。工程经理和前端团队负责人也参与了研究。

当被问及他们在前端方面的参与情况，无论是开发还是架构决策时，受访者对自己的评价介于中等到高水平之间。这一结果与预期一致，考虑到开发者直接处理前端问题，而工程经理则有更广泛的视角，并且间接参与更多。

当被问及新架构时，一些受访者回答了“我不知道如何回答”，而没有人选择“无影响”。这表明对感知到的架构影响缺乏清晰度。在分析的方面中，可测试性和招聘/入职流程受到了最大的负面影响。关于可测试性，尽管测试不再与单体耦合，但在基于微服务和微前端等分布式系统中的端到端测试本质上更为复杂。至于招聘和入职，服务碎片化可以在必要时促进分阶段引入；然而，理解整个系统的难度显著增加，需要更多的适应时间。

对于开放性问题，逐个分析了回答，并对相关内容进行了编码。经过全面分析后，将代码归类为主题，如旧架构、新架构和微前端等。每个主题都进一步细分以更好地组织出现的主题。

对于旧架构，目标是识别其主要优势和劣势，并验证受访者期望看到哪些方面的改进，也就是说，重点在于与变化相关的预期。对于新架构，目标是了解采用微前端后的感知优势和劣势，重点关注对参与者日常活动的影响。最后，研究调查了参与者对微前端的理解、他们对新架构的看法以及是否将其视为该方法的合法实现。

注意到大量缺点。然而，重要的是要强调，提出的一些观点可能与所使用的技术更密切相关，而不是架构本身。预计在响应中技术影响和架构影响之间存在混淆。在提到的优点中，大多数都与 `Aquarelle` 有关，尽管它实现了目标，但仍受到技术和架构限制的影响。最常被提及的缺点是前端系统之间的紧密耦合，这正是架构变更的主要原因以及预期改进的最大方面。

关于新架构，提到的优点多于缺点。有些观点被同时从正反两方面提及：使用新兴架构能够吸引开发人员，并且将系统拆分成更小的服务可能会简化入职流程。然而，分布式系统的内在复杂性可能延长入职过程。此外，尽管实施速度有所提高，代码量也相应增加。

一个值得注意的发现是，一半的参与者并未将新架构识别为微前端实现。可以得出结论，这两个主要因素促成了这一结果：文献中的现有定义和缺乏对垂直切片方法的明确引用。许多定义表明，单个页面必须被分割成多个可视组件才能被视为微前端，但这并不一定是真的。正如微服务一样，构成服务的定义并非一成不变。因此，可能存在根本不涉及可视界面的微前端。

分析还提供了证据，表明尽管微前端的成功实施，但这并不是 **Company A** 上下文中唯一可行的架构替代方案。例如，一个单体前端也可以工作，前提是接口解耦，并使用现代技术。支持这一论点的一点是只有一位参与者提到了可扩展性，这是基于微服务架构的主要优势之一，因此也是微前端的优势。

7 结论

本文旨在了解在行业中采用微前端是否值得。最初，通过对学术文献和灰色文献来源（如博客文章和会议演讲）的研究来探讨该架构的现状。接下来，在 **Company A** 实施了微前端，并最终通过对其开发者的调查评估了采用情况。

关于该主题的文献仍然有限，特别是关于 Mezzalira 所描述的垂直或基于领域的微前端概念 [9]，其简洁性有助于实施。此外，现有的定义缺乏灵活性，并且通常将架构与页面分解成组件绑定在一起，这在公司内部造成了混淆。

尽管采用了垂直方法，但由于缺乏组合这一核心关注点（如 Geers [4] 所述），开发者们犹豫将它归类为微前端架构。因此，有必要使该架构的定义与其前身——微服务——更紧密地对齐，并记录不带用户界面的替代形式如微前端。

在 **Company A** 处，所有先决条件均已满足，并认为采用是成功的。API 网关 模式实现了前端职责的分配，而后端用于前端（BFF）模式支持与后端服务的通信。尽管如此，采用微前端并不是严格必要的，因为其他架构，例如单体前端，也能达到类似的结果。使过渡更加可行的是公司正在进行向微服务迁移的过程，这使得团队能够重复使用基础设施并共享现有知识。因此，虽然不是唯一可能的解决方案，但微前端在特定情况下证明是最方便的选择。人们认为本文通过结合理论与实践，提供了电子商务环境中实施的历史记录，并对文献提出了批判性视角，可能会支持研究者和从业者。此外，本研究为 **Company A** 的架构讨论做出了贡献，并作为当前和未来开发者的文档。

参考文献

1. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison Wesley, 3a edn. (2013)
2. Brooks, F.: The Mythical Man-Month: Essays on Software Engineering. Addison Wesley (1995)
3. Fowler, M.: Stranglerfigapplication. https://martinfowler.com/bliki/StranglerFigApplication.html (2004), Último acesso em 14/08/2021
4. Geers, M.: Micro Frontends in Action. Manning, 1a edn. (2020)
5. Hacq, A.: Everything you need to know about design systems. https://uxdesign.cc/everything-you-need-to-know-about-design-systems-54b109851969 (2018), Último acesso em 23/12/2021
6. Jackson, C.: Micro frontends. https://martinfowler.com/articles/micro-frontends.html (Jun 2019), Último acesso em 21/06/2021
7. Lewis, J., Fowler, M.: Microservices. https://www.martinfowler.com/articles/microservices.html (Mar 2014), Último acesso em 21/06/2021
8. Martin, R.C.: Arquitetura Limpa: O Guia do Artesão para Estrutura e Design de Software. Alta Books Editora (2019)
9. Mezzalira, L.: Building Micro-Frontends. O'Reilley Media, Inc. (2021)

10. Peltonen, S., Mezzalira, L., Taibi, D.: Motivations, benefits, and issues for adopting micro-frontends: A multivocal literature review. Information and Software Technology **136**, 106571 (2021). https://doi.org/https://doi.org/10.1016/j.infsof.2021.106571, https://www.sciencedirect.com/science/article/pii/S0950584921000549
11. Richards, M.: Software Architecture Patterns. O'Reilley Media, Inc., 1a edn. (2015)
12. Richardson, C.: Microservices Patterns. Manning, 1a edn. (2018)
13. Richardson, C.: Pattern: Api gateway / backends for frontends. https://microservices.io/patterns/apigateway.html (2018), Último acesso em 27/10/2021
14. Richardson, C.: Pattern: Monolithic architecture. https://microservices.io/patterns/monolithic.html (2019), Último acesso em 23/12/2021
15. Yang, C., Liu, C., Su, Z.: Research and application of micro frontends. IOP Conference Series: Materials Science and Engineering **490**, 062082 (Apr 2019). https://doi.org/10.1088/1757-899x/490/6/062082, https://doi.org/10.1088/1757-899x/490/6/062082