Educational Administration: Theory and Practice
2024,30(11) 37 - 41
ISSN:2148-2403
https://kuey.net/

Research Article

Dissecting Micro-Frontends: A Deep Dive Into Architectural Components

Deep Hiren Kotecha[1*], Tanish Malekar[2], Shrey Parekh[3]

[1*]Scope, Vellore Insititute of Technology, deepkotecha2702@gmail.com
[2]Scope, Vellore Insititute of Technology, tanishmalekar32@gmail.com
[3]Dwarkadas J. Sanghvi College of Engineering, Mumbai University, talkshrey@gmail.com

Citation: Deep Hiren Kotecha, et.al, (2024), Dissecting Micro-Frontends: A Deep Dive Into Architectural Components, *Educational Administration: Theory and Practice, 30(11), 37 - 41*
Doi: 10.53555/kuey.v30i11.8299

ARTICLE INFO

ABSTRACT

The world of software development has advanced exponentially, transitioning from simple document representation and transport to sophisticated architectural paradigms like Micro-frontend Applications. Emerging as an evolution of Single Page Applications (SPA), Micro-frontend Applications offer a seamless user experience by breaking down the frontend monolith into smaller, manageable, and independently deployable components. This paper delves into the intricacies of Micro-frontend architectures, examining their core components and integration strategies. By analyzing these aspects, we aim to provide insights into the benefits and challenges of adopting Micro-frontend frameworks, guiding developers in optimizing their applications for modularity, scalability, and maintainability.

Index Terms— Software development, Micro-frontend applications, Single Page Applications (SPA), frontend monolith, independently deployable components, integration strategies.

I. INTRODUCTION

Micro-frontends are an innovative way to create current online apps in the rapidly changing field of web development. Micro-frontends, which take their cues from micro-services, divide the monolithic frontend into smaller, easier-to-manage components, each with a unique set of features. Teams may now work independently on various application components because to this paradigm shift, which encourages scalability, flexibility, and quick deployment. [1]

The key concept of modularization in micro-frontends requires the breaking of a monolithic application in several SPAs. These SPAs run on dynamic URLs. For example, we have 2 SPAs App-A and App-B. App-A is initialized when URL-A is requested by a browser. Same holds for App-B with URL-B. On Run-time with the help of all the components discussed in this study, App-A and App-B will appear as a single Application. However, their architecture allows them to run independently. [2]

The goal of this study is to present a thorough in-depth analysis of the architectural elements of micro-frontends. We will investigate the underlying ideas of this method, break down the various parts of a micro-frontend architecture, and look at the advantages and difficulties of putting it into practice. Developers and architects can better use micro-frontends' ability to build dependable, scalable, and maintainable online applications by being aware of their subtleties.

II. LITERATURE REVIEW

The first thing to bear in mind in micro-frontend architecture is that each micro-frontend is an application in its own right. In appearance, this is not determinable because the microfrontends are grouped within the same application interface with indistinguishable features. This poses a significant dependency on the backend counterpart of these applications. As documented by Prajwal, Parekh and Shettar, to ensure that secure and stable handshake of user state and journey is facilitated, all micro-frontends must be "Backend-Driven" applications. [3] In simpler terms, the application journey specifics must be maintained in and controlled from a backend microservice, which will act a source of truth. (as shown in Fig. 1)

Being a distinct application is beneficial for performance and service availability for a micro-frontend. As documented Fig. 2. Framework Diversity possible in Micro-frontends by Peltonen, Mezzalira and Taibi, since

all micro-frontends are individual applications, they have independent performance metrics. This means that in a group of micro-frontends if one application is citing performance issues, then this issue is not contagious to other micro-frontends. [4] The same thing can be implied in service downtime, if one micro-frontend is down, other services are still available.

This independence of application build is an opportunity for developers to leverage multiple SPA frameworks, based on the requirements of that application (as shown in Fig. 2). Often for the same application group micro-frontends utilize different frameworks, this is well explained by Gashi, Elvisa & Hyseni, Dhurate & Shabani, Isak & Cico, Betim. [5] This ¨ flexibility is a boon for developers trying to build complex flows for various Digital Businesses.

Coming onto the deployment architectures for microfrontends, we are faced with a variety of choices. The choice among us is to decide where to stage the distinction between micro-frontends. We can do so at the web server level, where based on regular expressions matching within the Request URL, we contact various micro-frontends. An alternate approach could differentiate the micro-frontend deployment service and make it configurable. Based on the observations of Cheripurathu and Kulkarni, and the prevailing sentiment in the industry towards the first approach, this paper will focus on the same. [6]

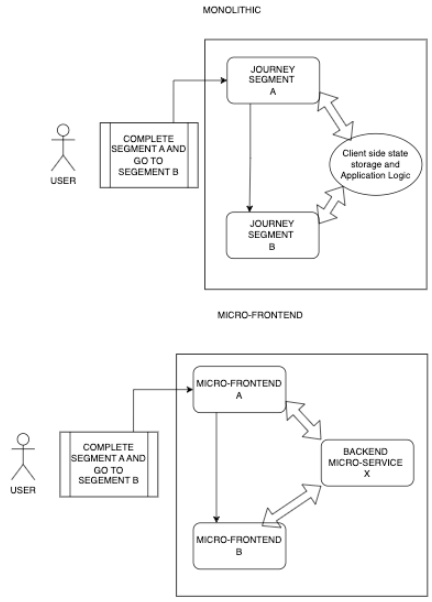

Fig. 1. State Management in Micro-frontend and Monolithic Applications

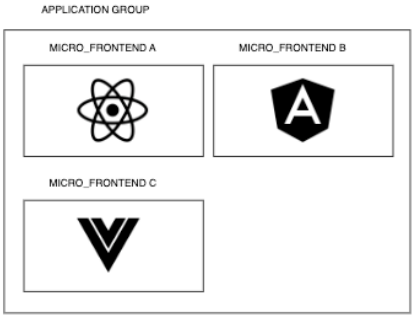

Fig. 2. Framework Diversity possible in Micro-frontend

III. PROPOSED ARCHITECTURE AND ARCHITECTURAL COMPONENTS

For any software system, the major stages of its development and release are – Build-Time and Run-Time. For each of these stages we shall outline all components and their usage. For our purposes Build Time refers to the stage where the application instance is built and deployed onto its container. Whereas Run-time refers to the process in which a user requests a Web Application, and our architecture handles the serving of the specific micro-frontend.

A. Build Time

1) Docker (Containerization): Docker is an open-source platform that streamlines the deployment and management of applications through containerization. Containerization is a technology that involves packaging an application and its dependencies together into a single, lightweight container. This container includes everything the application needs to run, such as code, libraries, and system tools, which ensures consistency

across different computing environments. [7] Unlike traditional virtualization, which uses virtual machines with their own operating systems, containers share the host system's OS kernel but operate in isolated user spaces. This makes them more efficient and portable, facilitating faster deployment, scalability, and simplified management of applications.

2) Helm (Deployment Package Manager): For Kubernetes, an open-source framework for managing containerized applications, Helm is a package management tool. Helm uses "charts," which are pre-configured Kubernetes resources, to make the deployment, management, and scaling of these applications easier. Helm is an effective tool for automating complicated Kubernetes activities and supporting best practices in continuous deployment and infrastructure as code since charts are simple to share and version. Helm's approach for templating enables users to dynamically manage their Kubernetes manifests, making application deployments predictable and uniform.

3) Pods (Kubernetes Container Registry): The lowest and most basic unit that may be created or deployed in the Kubernetes object model is called a pod. A pod can hold one or more containers (like Docker containers) and represents a single instance of a process that is currently operating in your cluster. A pod's containers can share storage volumes in addition to the network namespace, which includes the IP address and port space. Several intricately connected application containers that require resource sharing are supported by pods. [8] They provide consistent and effective resource management and scaling within a Kubernetes cluster by offering a high degree of abstraction for the deployment and management of containerized applications.

4) Build Time Architecture: As described in Fig. 3, we must build our application locally or through a CI (Continuous Integration) interface and push it onto Docker. Docker will perform containerization onto the built application image it receives. On completion, Docker will return the container compliant application image for further consumption. Next, we will ask Helm to fetch the containerized image from Docker and create replicas for load balanced deployments and aid Kubernetes in packaging individual Pods for Deployment. These pods will be stored into the Kubernetes Cluster and will be catalogued based on configurations. For our use case, they will package individual pods for individual microfrontends and create proxy-pass identifiers for run-time requests.

B. Run Time

1) Kubernetes (Cluster): A Kubernetes cluster is a robust and scalable system designed to automate the deployment, scaling, and management of containerized applications. It consists of a set of node machines, which can be either physical or virtual, orchestrated to run these applications efficiently. The Fig. 3. Build-Time Architecture architecture of a Kubernetes cluster is divided into two main types of nodes: master nodes and worker nodes. The master node, or control plane, manages the cluster and handles tasks such as scheduling applications, maintaining the desired state, scaling applications, and rolling out updates. Key components of the control plane include the API Server, which serves as the central management entity, etc., a distributed key-value store for configuration data and cluster state, the Controller Manager, which maintains the desired state by regulating various tasks, and the Scheduler, which assigns work to worker nodes based on resource availability.

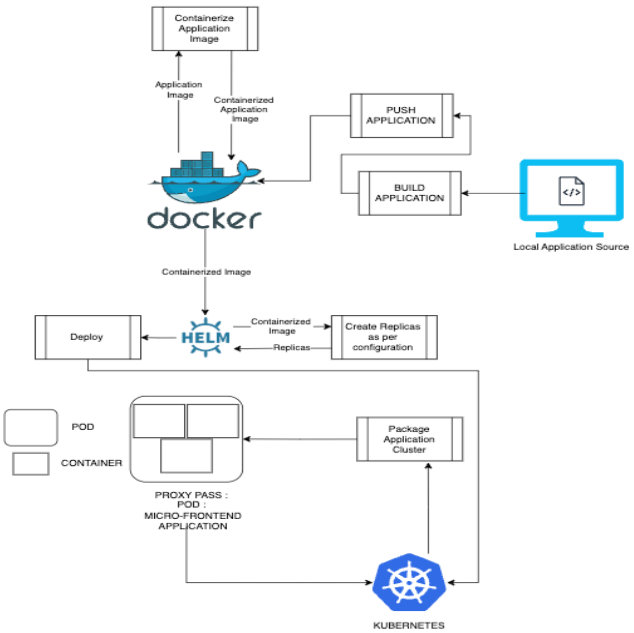

Fig. 3. Build-Time Architecture

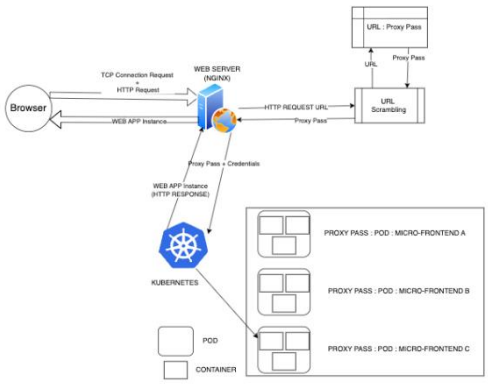

Fig. 4. Run-Time Architecture

The use of Kubernetes clusters has several advantages. Because of their inherent redundancy and self-healing characteristics, they enable high availability while providing scalability, enabling applications to scale up or down in response to traffic and resource usage. Through the optimization of underlying infrastructure consumption, Kubernetes also improves resource efficiency and lowers expenses. It also gives apps portability, allowing them to function reliably in a variety of settings, including public clouds and on-premises. Multi-cloud and hybrid settings, microservices designs, pipelines for continuous integration and deployment (CI/CD), batch processing or large data workloads—all make Kubernetes clusters helpful in a variety of scenarios. Developers may concentrate on creating apps while Kubernetes handles the operational complexity for enterprises by utilizing the extensive features of Kubernetes to Fig. 4. Run-Time Architecture achieve elevated levels of automation, efficiency, and dependability.

2) NGINX (Web server): Renowned for its impressive performance, stability, and low resource usage, NGINX is an open-source web server that can also be used as an HTTP cache, load balancer, and reverse proxy. NGINX is a crucial part of contemporary web infrastructures because of its ability to manage many concurrent connections with little memory consumption. It was first developed by Igor Sysoev. Serving static files (HTML, CSS, JavaScript, and pictures) and handling requests dynamically by serving as a gateway to several backend services, it does so with efficiency. Web applications perform better and are more scalable with this feature. Incoming network traffic can also be split up across several backend servers by NGINX, which ensures load balancing and raises the dependability and availability of web services.
In NGINX, route serving is achieved by use of a comprehensive configuration file that specifies the appropriate handling of various request kinds. Server blocks, which define the settings for domains or IP addresses, including the port and root directory for the files, are included in the configuration file. These are like virtual hosts in Apache. [9] Prefix matching and regular expressions are used by location blocks within these server blocks to route requests to different areas of the application based on the URI, deciding which block to apply. The NGINX “proxy pass” directive passes client requests to designated backend servers when operating as a reverse proxy.

3) Run-time Architecture: As described in Fig. 4, we use Kubernetes Cluster to host multiple pods, each pod identified by a unique “proxy pass” and maintaining containers of a specific micro-frontend. When the web server (Nginx) requests a particular proxy pass, Kubernetes responds with the relevant micro-frontend application instance. The process flow goes thus – the browser requests for URL: www.example.com/route-to-micro-frontend-c/extendedroute. The web server receives the request and initiates Scrambling of the URL. Based on the configs provided to the web server, it performs regular expression and pattern matching to determine the proxy pass relevant to micro-frontend-c. The web server passes this proxy pass and credentials onto Kubernetes. Kubernetes validates the credentials provided and retrieves the relevant application image for micro-frontend-c from the proxy pass. This instance is served to the web server and consequently to the browser. The micro-frontend renders on the browser and the website activities begin.

IV. CONCLUSION

We have carefully investigated the architectural elements of micro-frontends in this work, offering a thorough examination of their benefits, implementation, and structure. Micro-frontends, with their modular approach that improves scalability, maintainability, and flexibility, mark a significant evolution in front-end development. The examination of several architectural elements, such as state management, routing, and inter-micro-frontend communication, has brought to light the challenges and factors that must be considered for a successful deployment. The integration tactics, communication patterns, and frameworks selected all significantly impact how effective a micro-frontend architecture is. We also looked at the difficulties in

managing shared dependencies and providing consistent user experiences that come with micro-frontends, and we talked about several ways to address these problems. In the end, the use of micro-frontends is in line with the increasing demand for frontend architectures that are more durable and adaptable in the face of dynamic and sophisticated web applications. Micro-frontends present a promising means of achieving user-centric design and quick innovation, which are priorities for companies. It is anticipated that future work in this area will concentrate on improving performance, developing new integration strategies, and broadening the toolbox accessible to developers. By delving deeply into the architectural elements of micro-frontends, we intend to equip developers with the fundamental knowledge they need to effectively utilize this paradigm and advance the web development industry's future generation.

REFERENCES

1. Savani, Nilesh. (2023). The Future of Web Development: An Indepth Analysis of Micro-Frontend Approaches. International Journal of Computer Trends and Technology. 65-69.10.14445/22312803/IJCTTV71I11P109.
2. Yuma Nishizu, Tetsuo Kamina, Implementing Micro Frontends Using Signal-based Web Components, Journal of Information Processing, 2022, Volume 30, Pages 505-512, Released on J-STAGE August 15, 2022, Online ISSN 1882-6652, https://doi.org/10.2197/ipsjjip.30.505, https://www.jstage.jst.go.jp/article/ipsjjip/30/0/30 505/ article/char/en
3. Prajwal, Y. R., Parekh, J. V., & Shettar, R. (2021). A Brief Review of Micro-frontends. United International Journal for Research & Technology, 2(8), 123-126. Retrieved from https://uijrt.com/articles/v2/i8/UIJRTV2I80017.pdf
4. Peltonen, S., Mezzalira, L., & Taibi, D. (2020). Motivations, Benefits, and Issues for Adopting Micro-Frontends: A Multivocal Literature Review. Information and Software Technology. In Press. Retrieved from https://arxiv.org/abs/2007.00293v3
5. Gashi, Elvisa & Hyseni, Dhurate & Shabani, Isak & Cico, Betim. ¨ (2024). The advantages of Micro-Frontend architecture for developing web application.
6. Cheripurathu, Kurian & Kulkarni, Sanjeev. (2024). Integrating Microservices and Microfrontends: A Comprehensive Literature Review on Architecture, Design Patterns, and Implementation Challenges. Journal of Scientific Research and Technology. 1-12. 10.61808/jsrt115.
7. Mukaj, Jon. (2023). Containerization: Revolutionizing Software Development and Deployment Through Microservices Architecture Using Docker and Kubernetes. 10.13140/RG.2.2.23804.51841.
8. Abhishek, Manish & Rao, D. & Subrahmanyam, K.. (2022). Framework to Deploy Containers using Kubernetes and CI/CD Pipeline. International Journal of Advanced Computer Science and Applications. 13. 10.14569/IJACSA.2022.0130460.
9. Kunda, Douglas & Chihana, Sipiwe & Muwanei, Sinyinda. (2017). Web Server Performance of Apache and Nginx: A Systematic Literature Review. 8. 43-52.