A New Integration Approach to support the Development of Build-time Micro Frontend Architecture Applications

Fernando Rodrigues de Moraes
São Paulo State University – UNESP
Rio Claro, Brazil; and
VR Software
Limeira, Brazil
fr.moraes@unesp.br

Frank José Affonso
São Paulo State University – UNESP
Rio Claro, Brazil
f.affonso@unesp.br

ABSTRACT

The Micro Frontend Architecture (MFA) is an innovative, scalable, and flexible architectural model supporting frontend software development. Despite the advantages of this architectural model, technologies that depend on packaging applications in compile time benefit from MFA using a build-time approach by splitting Micro Frontend Applications (MFApps) in packages, which makes this application incapable of performing a runtime interpretation of other MFApps. This paper introduces a new integration approach to support the development of build-time MFApps to overcome limitations related to the coupling of these applications. The proposed approach employs the concept of remote component rendering and the Backend for Frontend pattern to solve continuous delivery issues, provide runtime integration, and help teams scale their development process of MFApps. As a result, we observed that our integration approach enables the developers to work with MFA in a build-time approach as a runtime integration of MFApps.

CCS CONCEPTS

• **Software and its engineering → Software creation and management**; *Designing software*;

KEYWORDS

Micro Frontend, Integration Approach, Build-time

1 INTRODUCTION

The Micro Frontend Architecture (MFA) is a microservices-inspired architecture for frontend applications, which enables this type of application to be developed and deployed independently. MFA is an innovative, scalable, and flexible architectural model that aims to support the development of web, desktop, and mobile applications. This architectural style facilitates efficient independent creation, deployment, and maintenance of decoupled applications for frontend [3, 8]. In this direction, it can be said that MFA has been revealed as a feasible alternative, especially in web applications, where it is possible to communicate with different micro applications hosted remotely and render them in runtime. Furthermore, the integration of MFA in web applications is empowered, because of their capacity to offer a dynamic frontend, capitalizing on the functionalities of browser runtime interpretation [7, 11]. According to a study conducted by Moraes et al. [11], the development of MFApps can be guided by three integration approaches, namely: build-time; frameworkless, and framework-based. In this article, only the build-time approach will be addressed because of its nature and development constraints.

Despite the multiple benefits that MFA offers for the development of frontend applications, the dependency of certain technologies in compiled applications, which generates a coupling between bundle applications known as Micro Frontend Applications (MFApps) has hindered the adoption of this architectural style within the software industry [8, 14, 17]. In short, the development of MFApps based on the build-time approach makes the main application unable to interpret other MFApps at runtime. Since mobile and desktop applications are usually compiled in a final bundle application (i.e. platform's native applications), which demands a natural coupling of MFApps in a build-time approach, teams avoid the adoption of MFA in this scenario because the missing benefits from MFA as decoupling, independent deploy, and fault isolation.

Based on the presented context, this paper presents a new integration approach to support the development of build-time MFApps to minimize the impacts related to the aforementioned limitations. The proposed approach, referred to from this point forward as RSBF (Runtime Service-Based Frontend), is based on concepts of Remote Component Rendering (RCR) associated with the Backend For Frontend (BFF) pattern [16] to solve continuous delivery issues, provide runtime integration, and help teams scale their development process of MFApps. In short, the power of components/widgets remote rendering provided by some frameworks or libraries (e.g., Remote Flutter Widget [15], React Native Remote Components [9]) to help teams adopt MFA with similar benefits in a build-time approach as well. A component/widget remote rendering is a capability of integrating remote responses (e.g., HTTP[1], socket) and interpreting as a portion of a User Interface (UI) in a frontend application, rendering this UI at runtime. BFF is a microservice pattern that dictates specific services to be developed oriented to frontend applications, with easier-to-integrate backend responses that fit better to users and with only required data [1, 16].

The remainder of this paper is organized as follows. Section 2 presents the background and related work. Section 3 provides a description of our approach. Section 4 presents a case study to show the applicability of the approach proposed in this paper. A brief discussion of the results and limitations of our study is presented in Section 5. Finally, Section 6 summarizes our conclusions and perspectives for further research.

[1]Hypertext Transfer Protocol

2 BACKGROUND AND RELATED WORK

This section presents the background (i.e., concepts and definitions) and related work that contributed to the development of our proposal. Initially, concepts of micro frontends and their development approaches are reported. Next, a general view of the main issues (i.e., BFF and RCR) that integrate our proposal is addressed. Finally, related work on our proposal is presented.

Micro frontends. MFA is a modern architectural style based on microservice principles. In this architecture, an application (i.e., Desktop, Web, or Mobile) is developed as a collection of independent frontend applications (i.e., MFApps), each representing a specific part of the business domain. Each part of the UI is treated as a separate component or page, which can be developed, tested, deployed, and containerized independently. This modular organization enables small autonomous teams, with different technology skills (i.e., technology stacks and implementation pipelines) to work independently. This feature reduces dependency between teams, increases governance, and simplifies business complexity. Additionally, the MFA has the potential to encourage innovation and optimize application maintenance activity in the future [4, 11, 13]. Figure 1 illustrates the organization and layout of MFApps. As seen in (A), teams can be organized to work in three layers (i.e., frontend, backend, and database), where each column represents a development team working in full-stack mode. The layout decisions are represented in (B), which can be based on two options: *horizontal*, which enables multiple micro frontends per page; and *vertical*, which enables one micro frontend per page.

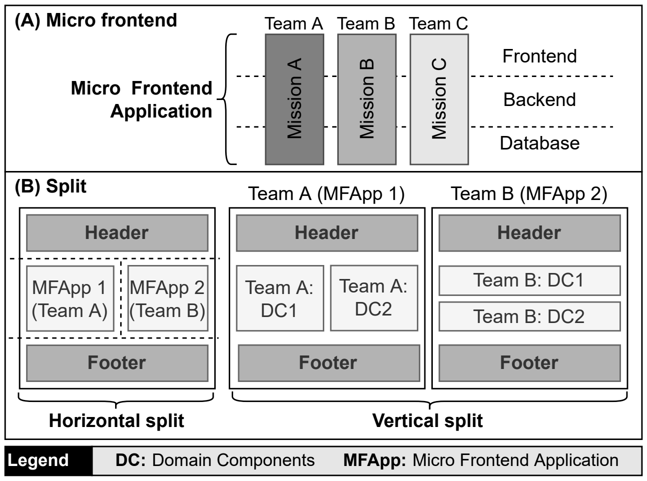


Figure 1: MFApps organization and layout

As reported in Section 1, an MFApp can be developed according to guidelines of three integration approaches: build-time, frameworkless, and framework-based [11]. Build-time is the simplest approach to the architectural implementation of micro frontends because it enables the development of MFApps as packages [17]. Frameworkless is a type of approach that does not require frameworks for the development of MFApps [5]. Framework-based is a pragmatic and successful approach that follows guidelines established within an architectural framework used to implement integration between MFApps [7].

Remote Component Rendering. RCR refers to the ability of a frontend technology to interpret remote data (e.g., HTTP and socket) and dynamically render UIs during runtime execution. This feature enables teams to write UI code (e.g., components, widgets) directly in the backend and promotes data-oriented customization of UI and faster changes to frontend applications since only the backend has to be deployed. In parallel, when MFApps are built using the BFF pattern, combining these concepts suggests that an optimized and integrated UI model design will be delivered to consuming applications. Since web applications can rely on other supported runtime integration approaches based on web components, iFrames, and module federation, RCR becomes suited to architectures developed using a build-time integration approach. Among the aforementioned technologies, iFrame has been used most to enable runtime rendering in desktop and mobile applications. In short, the iFrame does not render the components natively, causing performance degradation, usability, and accessibility problems arising from technology limitations, resulting in low adoption by the scientific community and practitioners [6, 18]. Finally, it is worth highlighting that this concept is an integral part of the proposal presented in this paper. To do so, scientific evidence was gathered [2, 10], compiled, and associated with the professional experience of the authors of this paper.

Backend for Frontend. BFF is a design pattern for developing backend services to satisfy frontend needs with a more optimized communication interface, providing custom-specific queries and responses to be integrated by a frontend application [12, 16]. This pattern facilitates integration between the backend and frontend applications and expands possibilities for data or interfaces visualized by the users because the backend is closer to user data. For instance, the backend can determine (e.g., user location, preferences) what will be sent to the frontend application. This strategy is largely used by companies in different use cases in the industry aiming to solve problems in frontend communication with microservices and to deliver a better user experience by providing a customized interface based on data [1]. Based on this context, the BFF showcases a pluggable pattern for MFA because it promotes vertical full-stack teams, runtime rendering, and easier frontend integration. Moreover, the adoption of this pattern may reduce bottlenecks on development integration and teams' cross-system implementation concerns, since a frontend developer in a vertical full-stack position would maintain an abstracted service layered on top of other microservices guided to frontend instead of directly maintaining the backend.

As **related work**, to the best of our knowledge, there is no solution regarding the design of MFApps based on a build-time approach that supports the integration of other MFApps at runtime. Thus, we present in this section relevant studies that have addressed some type of investigation related to the build-time approach to MFA. Next, a description of each study is addressed.

Stefanovska & Trajkovik [17] conducted a study to evaluate code reuse in an MFA. In short, this study can be summarized in two phases: the decomposition of a frontend application; and the case reuse by integrating the decomposed components in a new MFApp. In the same direction as Jackson [8], the aforementioned authors mentioned that the build-time integration approach is limited because it is necessary to re-compile and release every single MFAppp to release a change to any individual part of the final application.

The development of MFApps in Single-Page Applications (SPA) was investigated in the study by Pavlenko et al. [13]. To do so, the aforementioned authors conducted a case study focused on the domain of online courses, which investigated the applicability of microservices architecture in the design and frontend development of a web application. The evidence of this study revealed that the main drawback of the build-time approach is the compilation time of an application because all the MFApps should be recompiled if one of them is changed.

Despite these relevant initiatives, there is no similar proposal to ours, which can deal with the development of MFApps based on the build-time integration approach and the MFApp integration at runtime. In summary, these initiatives were constrained to adhering to established integration approaches because of apprehension for issues that have already been overcome by the inherent features of MFA. Therefore, it can be said that our proposal bypasses the limitations of the build-time integration approach, making it behave similarly to the integration approaches at runtime.

3 THE PROPOSED APPROACH

As reported in Section 2, MFA requires that MFApps be decoupled and integrated at runtime to enable the full benefits and capabilities of the MFA architectural style. Drawing a parallel between integration approaches, applications based on a build-time approach are considered easier to implement than those based on a runtime approach, since the application code is grouped at compile-time, but comes with the cost of the limitations mentioned in Section 1. According to evidence reported in the study conducted by Moraes et al. [11], the development of MFApps based on compiled technologies presents limitations regarding continuous software delivery, which restricts several benefits of the MFA architectural style. In parallel, this study revealed that the developers are restricted/limited by developing MFApps for mobile applications (e.g., Android and IOS) or Desktop native based on compiled technologies because there is no alternative to deliver the final application. Based on this context, the integration approach proposed in this paper combines the rendering capabilities of remote components with the BFF pattern to support the development of MFApps based on the build-time integration approach. This combination enables the development of this type of application to overcome the natural limitations imposed by development based on compiled technologies, making it behave similarly to the development of technologies based on runtime. The Figure 2 illustrates a layered architectural view of the RSBF integration approach.

Regarding the architectural model, the proposed approach was organized into three layers interconnected by two distinct gateway services, the first for RSBF services and the second for microservices. As illustrated in **top layer**, the development of MFApps can adopt different division strategies. Although only the vertical and horizontal divisions have been depicted in this figure, the hybrid division, which combines the previous two, can also be used. According to our proposal, the frontend team should be responsible for developing MFApps and RSBF services (**intermediate layer**), integrating both

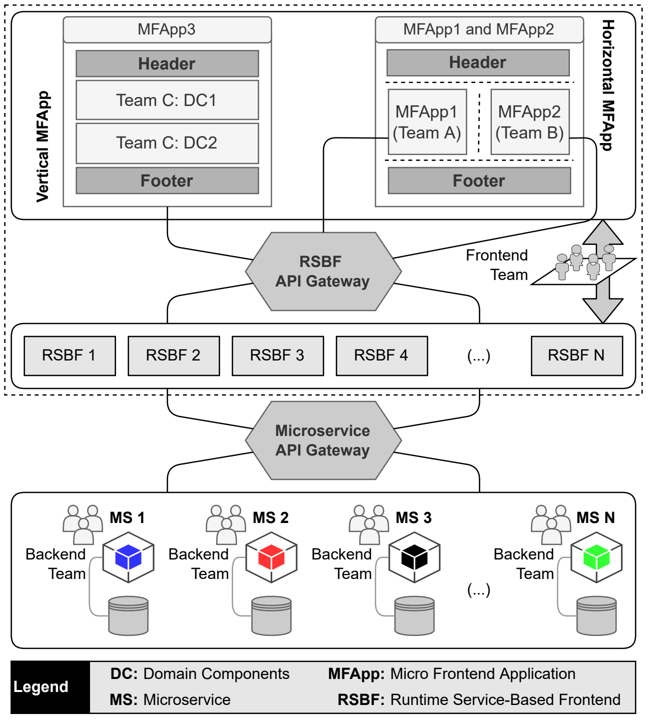


Figure 2: An architectural view of the RSBF approach

through an "`RSBF API Gateway`". API Gateway is a design pattern for microservice architecture, which creates a gateway between services and the frontend application. In this sense, developers must create a specific API gateway for each frontend, encapsulating the RSBF services for each MFApp. This abstraction enables the frontend team greater flexibility when making changes to their RSBF services, making deployment independent of the backend teams and enabling faster delivery of frontend changes. By employing this strategy, the developers can inject remote code in a frontend so that it can be interpreted and rendered into a user interface, thereby enabling the development of MFApps with runtime integration. Thus, development teams can operate in full-stack mode, abstracting the backend (i.e., microservices, APIs, and databases) and maintaining only their services (i.e., RSBF) for frontend applications. Finally, in the **lower layer** the development of microservices that will make up an application (`MS-1` to `MN-N`) takes place by the backend team. RSBF services utilize the "`Microservice API Gateway`" to access the microservices within this layer.

Based on the presented context, it can be said that MFApps based on the RSBF approach provide an abstraction layer when using BFF concepts, making teams specialize in maintaining frontend-oriented applications by knowing the communication interface between the frontend and the backend. Therefore, the proposed integration approach can provide several benefits for the development of MFApps based on the build-time approach, facilitating its adoption for those interested in MFA development with compiled technologies. Next, a brief description of these benefits is addressed:

- *Optimized maintenance.* Specific services designed for MFApps become easier to maintain, since both (frontend and backend) were designed to "speak the same language". Stated differently, the backend was designed specifically to satisfy the individual needs of the frontend.
- *Promoted full stack teams.* The architectural style of micro frontends favors the organization of development teams in full stack mode because developers work from the design of the BFF service to the frontend of an MFApp.
- *Easy integration.* The development of BFF services tends to minimize integration efforts between such services and an MFApp. In summary, these services should be developed to deliver specific data to the frontend, which will then be responsible for incorporating this data into a designated portion of the user interface.
- *Service-oriented data capabilities.* Based on data, services can perform transformations in the user interface and provide data-oriented capabilities directly in the backend; and
- *Faster software delivery.* A compiled frontend application must be fully deployed to make changes to user interfaces. By following the proposed approach, it is possible to make changes to the user interfaces independently while implementing the backend service.

Figure 3 illustrates a comparison between the integration approaches, build-time, runtime, and RSBF. It can be observed that in the build-time approach, applications (`App` element) are naturally coupled when built with the Package Manager (`PM` element) forming a single package, significantly harming the advantages of MFA [8]. At runtime, applications can be served in different runtime applications and interpreted independently, becoming fully decoupled, enabling independent deployment and development of MFApps. Based on this context, it can be said that build-time using RSBF is an approach for compiled applications where runtime interpretation is still required. To accomplish this, MFApps are bundled during the build process, while the UI portion is served by independent microservices. Thus, following BFF concepts and applying RCR capability, these RSBF applications are integrated and empower the front-end application with runtime interpretation, delivering a new UI interpreted at runtime to the client (i.e., mobile or Desktop applications).

4 CASE STUDY

To evaluate the applicability, strengths, and weaknesses of our integration approach, this section presents a case study that we conducted in the mobile application course platform domain, which will be referred to from this point forward as `AppCourse`. Our interest is to show that the proposed integration approach can support the development of MFApps based on compiled technologies, overcoming the limitations mentioned in Sections 1 and 3.

As **subject application** of our empirical analysis, we selected an application aimed at course management, as illustrated in Figure 4. This application provides its students with a set of courses organized into several modules, each with specific content and can be present in more than one course. As courses are offered remotely, students can enroll in multiple courses simultaneously. Furthermore, as courses can have intersecting modules, the student can

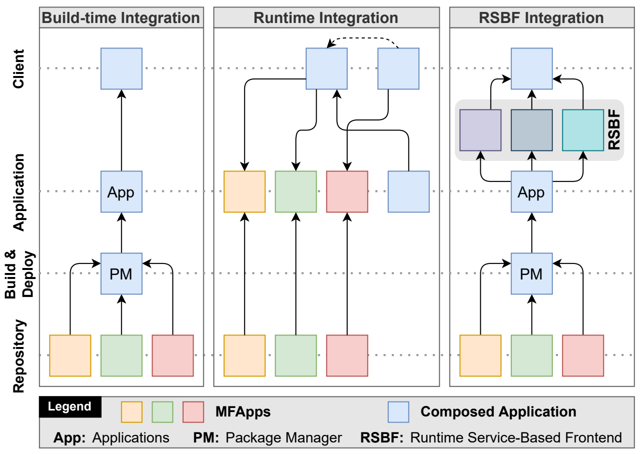


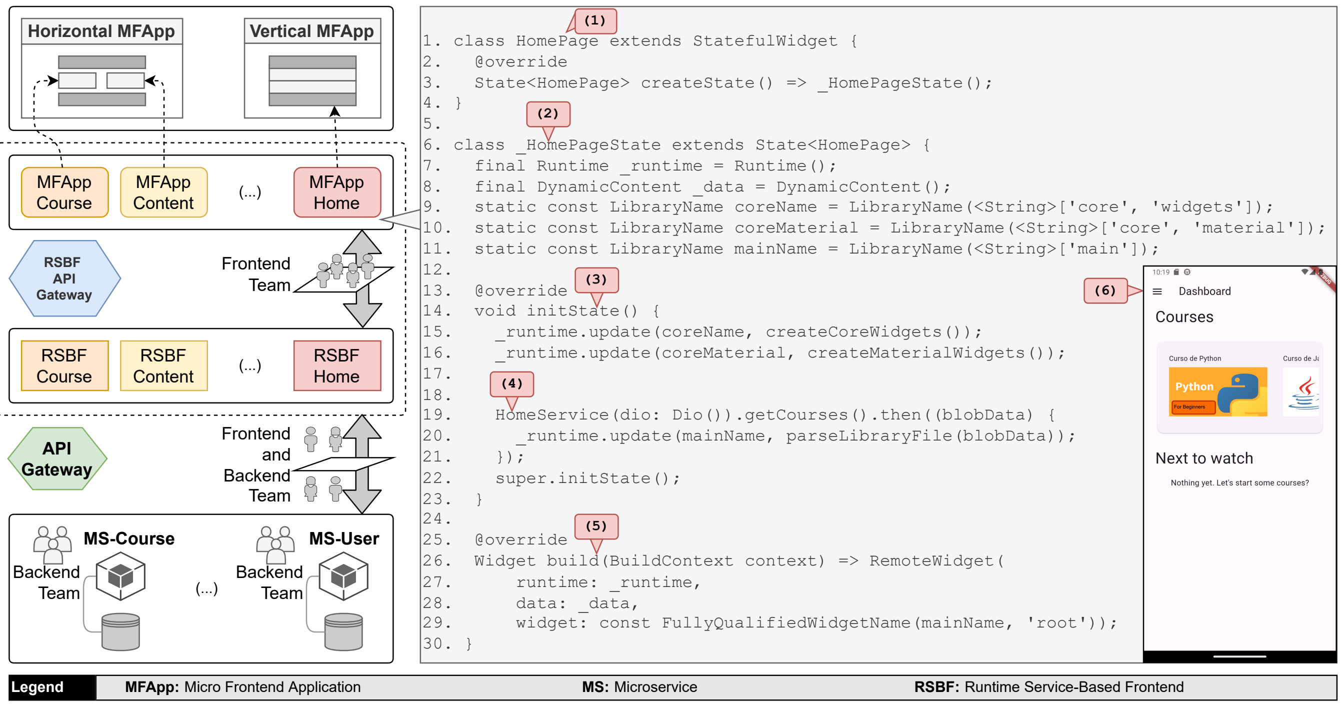
Figure 3: Comparative between integration approaches

take advantage of modules so as not to take the same module more than once. Next, we outlined a concise description of our empirical techniques employed for this case study.

Empirical research strategy. Concerning the development of the `AppCourse` application[2], the Flutter[3] framework was used as frontend technology together with the RFW library[4], which is responsible for remote interpretation of the RSBF-based UI. In this direction, it is worth highlighting that the technological choice can directly influence the limitations of the approach proposed in this paper, particularly regarding the ability to render application components remotely.

As illustrated in Figure 4, the `AppCourse` application is fully compliant with the same architectural organization presented in Figure 2. For space and scope reasons, only elements related to the approach proposed in this paper will be detailed in this section. At the bottom, the development of the microservices that will be part of the `AppCourse` application (e.g., `MS-Course` and `MS-User`) is illustrated, which will be accessed by the application's RSBF services via `API Gateway`. Here, it is worth highlighting the communication between the backend and frontend teams to align needs and standardize APIS (i.e., input and output parameters). Therefore, it can be stated that the responsibility for developing RSBF services that deliver UI source code in blob format (i.e., binary data) via HTTP requests lies with the frontend team. It is worth highlighting that the choice of blob is because this format is ideal for moving files between APIs and MFApps. From an operational viewpoint, it can be said that it is up to an MFApp to request to consume data from one (or more) API(s) and render the result on the screen. The source code listing numbered from 1 to 30 exemplifies the integration between `Home` RSBF service and MFApps. For instance, the `HomePage` class, referenced by the **Balloon (1)**, is a Flutter component of the `Home` MFApp, rendering the initial screen after user authentication. This MFApp queries the `Home` RSBF service, which returns the content of the mentioned page based on the data retrieved from the equivalent microservice.

[2]https://github.com/fernandormoraes/the-bob-project/tree/main/bob_mobile
[3] https://flutter.dev
[4]https://pub.dev/packages/rfw



```
1. class HomePage extends StatefulWidget {
2.   @override
3.   State<HomePage> createState() => _HomePageState();
4. }
5.
6. class _HomePageState extends State<HomePage> {
7.   final Runtime _runtime = Runtime();
8.   final DynamicContent _data = DynamicContent();
9.   static const LibraryName coreName = LibraryName(<String>['core', 'widgets']);
10.  static const LibraryName coreMaterial = LibraryName(<String>['core', 'material']);
11.  static const LibraryName mainName = LibraryName(<String>['main']);
12.
13.  @override
14.  void initState() {
15.    _runtime.update(coreName, createCoreWidgets());
16.    _runtime.update(coreMaterial, createMaterialWidgets());
17.
18.
19.    HomeService(dio: Dio()).getCourses().then((blobData) {
20.      _runtime.update(mainName, parseLibraryFile(blobData));
21.    });
22.    super.initState();
23.  }
24.
25.  @override
26.  Widget build(BuildContext context) => RemoteWidget(
27.      runtime: _runtime,
28.      data: _data,
29.      widget: const FullyQualifiedWidgetName(mainName, 'root'));
30. }
```

Figure 4: Overview of RSBF integration approach

By examining the source code of the mentioned figure, **Balloons 1 to 5** illustrate the execution sequence, emphasizing the key steps for implementing the approach outlined in this paper (see Section 3) and demonstrated in this case study. In **(1)** a class called `HomePage` is declared, which inherits from a predefined class from the Flutter framework called `StatefulWidget`. This Flutter class defines methods for executing components that contain state (i.e. changes in screen rendering). As can be seen in Line 3, this class contains the `createState` method that must be overridden to return an instance of `State<T>`.

In **(2)**, `_HomePageState` is declared as a private class[5] at Line 6. In short, this class represents the state of screen rendering, inheriting from the pre-defined `State<T>` class, where `T` represents a generic type. Here, `HomePage` is the class that instantiates the `_HomePageState` screen state in the `createState` method call (see Line 3). Subsequently, in Lines 7 to 11, the objects of the RFW library (e.g., `Runtime`, `DynamicContent`, `LibraryName`) are declared and instantiated to render the remote components.

As can be seen in **(3)**, `initState` (Line 14) is the method inherited from the `State` class (Line 6), which must be called at the beginning of the rendering of a UI component. To do so, this method was overridden so that `_HomePageState` can request data from the RSBF services and render the new state on the screen with the UI returned from the service. The private attribute named `_runtime` is responsible for building the library of Core and Material components provided by the RFW library (see Lines 9 and 10). Regarding these component libraries, other ones can also be used according to developers' customization.

`HomeService` is the class responsible for receiving the instance of a library called `Dio`[6] so that HTTP requests can be achieved **(4)**. For instance, this class requests an RSBF service by calling the `getCourses` method, which returns data in blob format in `blobData` parameter (see Line 19). Then, the `update` method (Line 20) of the `Runtime` class (see declaration in Line 7) is called, receiving two parameters. The first is a `LibraryName` object (i.e., `mainName` object – see Line 11) and the second is the return from the `parseLibraryFile` function, which deserializes the returned code into `blobData` parameter and interprets it to be rendered. It is worth mentioning that this function belongs to the RFW library.

In **(5)**, `RemoteWidget` is a component of the RFW library that receives an object from the `Runtime` class on Line 27 (see declaration on Line 7), an object from the `DynamicContent` class on Line 28 (see declaration on Line 8), and an instance of a component used to reference the component library that will be rendered on Line 29. The `build` method, inherited from the `State` class, returns a `Widget` component to be rendered on the screen. To do so, this method was overridden so that it returns a `RemoteWidget` component, so the return to be rendered on screen becomes the interpreted remote component. The screen illustrated in **(6)** represents the remote component already rendered for the end user.

Based on the content presented in this section, it can be said that the approach proposed in this paper was implemented in this case study through the relationship between the `HomePage` component **(1)**, which instantiates the RFW library objects (see Lines 7 to 11), and the interpretation of the request's response made by `HomeService` **(4)** to RSBF services, rendering a UI code snippet at

[5] In Flutter, a class or attribute private must be declared with "_" character at begin.

[6] https://pub.dev/packages/dio

runtime. In short, this relationship was detailed in three steps (i.e., 3, 4, and 5). In **(3)** and **(4)** the state of the `HomePage` component was initialized and a request for an RSBF was called (Line 19), returning a blob file for interpretation of the returned code (Line 20). The update of the component state for an interpreted UI in the component rendering occurs in **(5)** by calling the build method with a return from a `RemoteWidget`. Next, we briefly discuss the results and limitations of the proposed new integration approach and the case study conducted in this paper.

5 DISCUSSION OF RESULTS

This section provides a summary of the main findings and discusses the relevance of the study. The investigation presented a novel integration approach surrounding MFA, which surpassed the known limitations on build-time implementations of the aforementioned architecture.

As evidenced by the case study presented in Section 4, it can be stated that the novel integration approach proposed in this paper is a promising alternative for overcoming the inherent limitations of compiled applications when adopting MFA. In short, our approach employs runtime rendering capabilities to address the shortcomings of the build-time approach, enabling the adoption of principles analogous to those employed in runtime integration approaches for MFA. Because of this feature, we can infer that applications developed using compiled technologies can also benefit from the principles of MFA [11]. To achieve this, the proposed approach provides a means of decoupling these compiled applications, thereby enabling independent deployment and the isolation of failures. Consequently, the benefits gained from the use of RSBF in conjunction with MFA are enhanced, with previously coupled MFApps now consuming UI code from decoupled RSBFs that are deployed and isolated as microservices.

With regards to the final application, the integration of a frontend application with RSBF simplifies the process of modifying the user interface, eliminating the need for recompiling and redeploying the application. This is a noteworthy feature, particularly for applications that are required to be deployed in application stores (e.g., Play Store, App Store, Microsoft Store). Given that these stores have a review process that may result in delays in time to market, the approach proposed in this paper can effectively addresses this challenge by enabling direct changes to the UI in RSBF.

Concerning the limitations, it is important to note that the approach proposed in this paper relies on RCR capabilities. Therefore, it is recommended to conduct a thorough analysis of each technology, such as the frontend framework, to identify potential limitations. Our case study revealed that the RFW library for Flutter presented some limitations concerning UI rendering. For instance, this library does not provide the capability to create animations, page transitions, or drag-and-drop components. Nevertheless, it is possible to overcome these limitations using pre-built components (e.g., reusing components already created on the frontend side), since the RFW library can interpret the pre-built component consumed from a remote source. A preliminary investigation into the impact of our approach on application performance suggests that it may have a slightly negative effect. In a build-time application, it is possible to rely on immutable components or widgets. In contrast, when using RCR, the components are rendered at runtime.

6 FINAL REMARKS AND FUTURE WORK

According to the investigation conducted by Moraes et al. [11], there are several research gaps related to MFApps development, namely: (i) establishment of solid and complete concepts; (ii) management of the development of MFApps; and (iii) frameworks for decision-making regarding the adoption of this architectural style. Therefore, it can be said that establishing any initiative on this research topic is a non-trivial and challenging activity, especially when there is no literature support to denote guiding parameters for this type of work. Therefore, aiming to contribute to this research topic, this paper presented an integration approach called RSBF, developed based on BFF services and the RCR concept. This approach enables build-time MFApps development (i.e., compiled technologies) to behave similarly to runtime MFApps development. Although build-time is an integration approach that has limitations that violate the MFA principles (see Sections 1 and 3), applications targeted at native platforms (e.g., Mobile, Desktop) depend on their packaging at build-time. This feature reinforces the interest of the scientific community and practitioners in directing their efforts to promote advances in this area of research (i.e., integration approach).

Although the approach proposed in this paper has been previously evaluated through a case study, it is worth highlighting that the work in progress is accomplished by researchers who have been investigated in this research area, as well as professionals with relevant experience in this development area (i.e., MFA). Regarding future work on the integration approach proposed in this paper, at least two activities are intended: (i) conducting further case studies or proofs of concepts to fully evaluate the proposed approach, including different software domains and different technologies for component rendering and UI interpretation; (ii) instantiate our approach in other programming languages and frontend frameworks in order to evaluate its applicability and behavior in relation to its main purpose; and (iii) evaluation of the behavior of the proposed approach in a broader development and execution scenario. Therefore, based on the content presented in this paper, a positive research scenario can be idealized, since it is envisaged that the proposed approach can become an effective contribution to the software engineering area as a feasible alternative for problems related to the development of MFApps for applications that require the build-time approach (e.g., Mobile and Desktop).

ACKNOWLEDGMENTS

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES).

REFERENCES

[1] Amr S. Abdelfattah and Tomás Cerný. 2023. Filling The Gaps in Microservice Frontend Communication: Case for New Frontend Patterns. In *Proceedings of the 13th International Conference on Cloud Computing and Services Science, CLOSER 2023, Prague, Czech Republic, April 26-28, 2023*, Maarten van Steen and Claus Pahl (Eds.). SCITEPRESS, Prague, Czech Republic, 184–193. https://doi.org/10.5220/0011812500003488

[2] Markus Ast and Martin Gaedke. 2017. Self-contained web components through serverless computing. In *Proceedings of the 2nd International Workshop on Serverless Computing* (Las Vegas, Nevada) *(WoSC '17)*. Association for Computing Machinery, New York, NY, USA, 28–33. https://doi.org/10.1145/3154847.3154849

[3] Yan Bian, Dechao Ma, Qing Zou, and Weirui Yue. 2022. A Multi-way Access Portal Website Construction Scheme. In *The 5th International Conference on Artificial Intelligence and Big Data, ICAIBD 2022*. Institute of Electrical and Electronics Engineers Inc., Chengdu, China, 589 – 592. https://doi.org/10.1109/ICAIBD55127.2022.9820236

[4] Fabian Bühler, Johanna Barzen, Lukas Harzenetter, Frank Leymann, and Philipp Wundrack. 2022. Combining the Best of Two Worlds: Microservices and Micro Frontends as Basis for a New Plugin Architecture. *Communications in Computer and Information Science* 1603 CCIS (2022), 3 – 23. https://doi.org/10.1007/978-3-031-18304-1_1

[5] Frameworkless. 2024. Frameworkless movement. on-line. Avaliable: https://www.frameworklessmovement.org, acessed on July 26, 2024.

[6] E. Garcia-Lopez, A. Garcia-Cabot, A. Castillo-Martinez, A. Gutierrez-Escolar, J.A. Medina, J.M. Gutierrez-Martinez, , and J.J. Martinez-Herraiz. 2014. Mobile Usability: An Experiment to Check Whether Current Mobile Devices are Ready to Support Frames and iFrames. In *Information Systems Development: Transforming Organisations and Society through Information Systems (ISD2014 Proceedings)*, V. Strahonja, N. Vrček., D. Plantak Vukovac, C. Barry, M. Lang, H. Linger, and C. Schneider (Eds.). Information Systems Development: Transforming Organisations and Society through Information Systems (ISD2014 Proceedings), Varaždin, Croatia: Faculty of Organization and Informatics, 234–241.

[7] Michael Geers. 2020. *Micro frontends in action*. Manning Publications, New York, NY.

[8] Cam Jackson. 2019. Micro Frontends. on-line. Avaliable: https://martinfowler.com/articles/micro-frontends.html, acessed on July 26, 2024.

[9] Sarath KCM. 2024. react-native-remote-components. on-line. Avaliable: https://github.com/sarathkcm/react-native-remote-components, acessed on July 26, 2024.

[10] Microsoft. 2024. Visual Studio App Center documentation. on-line. Avaliable: https://learn.microsoft.com/en-us/appcenter/distribution/codepush/, acessed on July 26, 2024.

[11] Fernando Rodrigues Moraes, Gabriel Nagassaki Campos, Nathalia Rodrigues Almeida, and Frank José Affonso. 2024. Micro frontend-based Development: Concepts, Motivations, Implementation Principles, and an Experience Report. In *Proceedings of the 26th International Conference on Enterprise Information Systems*. INSTICC, SciTePress, Angers, France, 175–184. https://doi.org/10.5220/0012627300003690

[12] Sam Newman. 2015. Pattern: Backends For Frontends. on-line. Avaliable: https://samnewman.io/patterns/architectural/bff/, acessed on July 26, 2024.

[13] Andrey Pavlenko, Nursultan Askarbekuly, Swati Megha, and Manuel Mazzara. 2020. Micro-frontends: Application of microservices to web front-ends. *Journal of Internet Services and Information Security* 10, 2 (2020), 49 – 66. https://doi.org/10.22667/JISIS.2020.05.31.049

[14] Severi Peltonen, Luca Mezzalira, and Davide Taibi. 2021. Motivations, benefits, and issues for adopting Micro-Frontends: A Multivocal Literature Review. *Information and Software Technology* 136 (2021), 106571. https://doi.org/10.1016/j.infsof.2021.106571

[15] PUB.DEV. 2024. Remote Flutter Widgets. on-line. Avaliable: https://pub.dev/packages/rfw, acessed on July 26, 2024, version rfw: 1.0.26.

[16] Chris Richardson. 2024. Variation: Backends for frontends. on-line. Avaliable: https://microservices.io/patterns/apigateway.html, acessed on July 26, 2024.

[17] Emilija Stefanovska and Vladimir Trajkovik. 2022. Evaluating Micro Frontend Approaches for Code Reusability. *Communications in Computer and Information Science* 1740 CCIS (2022), 93 – 106. https://doi.org/10.1007/978-3-031-22792-9_8

[18] Guangliang Yang, Jeff Huang, and Guofei Gu. 2019. Iframes/Popups Are Dangerous in Mobile WebView: Studying and Mitigating Differential Context Vulnerabilities. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 977–994. https://www.usenix.org/conference/usenixsecurity19/presentation/yang-guangliang