Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Bachelor Thesis

Bachelor of Science

Evaluation of Micro Frontends in Web Development with Piral Framework

from Nam Phuong Nguyen
Department of Computer Science

First Examiner: Prof. Dr. Manfred Kaul

Second Examiner: Prof. Dr. Sascha Alda

Supervisor: Bernd Böllert

Submitted on: 04. July 2022

Declaration of Academic Integrity

“Ich versichere hiermit, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Mir ist bewusst, dass sich die Hochschule vorbehält, meine Arbeit auf plagiierte Inhalte hin zu überprüfen und dass das Auffinden von plagiierten Inhalten zur Nichtigkeit der Arbeit, zur Aberkennung des Abschlusses und zur Exmatrikulation führen kann.”

“I hereby declare that I have independently written the work that I have submitted. All passages, taken literally or paraphrased from published or unpublished works of others, I have appropriately cited as such. All sources and resources I have used for the submitted work are given. The work has not yet been submitted to any other examination body with the same content or in substantial parts.

I am aware that the university reserves the right to check my work for plagiarized content, and that finding plagiarized content can lead to the dismissal of the work, to the disqualification of the conclusion and to exmatriculation.”

Location, Date

Signature

Abstract

Frontend engineering on the web significantly growing more complex over the years with respect to projects' scope, codebase size, and extensive features requires that application architecture function as the precondition for maintaining sustainable software development. Micro Frontends applying micro architecture on the Frontend layer has drawn considerable attention for all the benefits this design offers: fast delivery, development optimization and developer's autonomy, of which the most dramatic and influential changes taking place are those in feature expansion and scaling development. The decision of Frontend architecture, broadly conceived, come under the influence of business reasons, legacy issues, technological advancement, resource optimization and people management. On the basis of software architecture in web application and Microservices, the study aims to present Micro Frontends major ideas, principles, best practices and investigates the opportunities, challenges of Micro Frontends application. In particular, the thesis focuses theoretically and practically on client-side composed Micro Frontends with sample application using Piral. Empirical analysis and evaluation are based on code snippets and an exemplary, quality-assured application with unit testing. Based on the implemented and measured results, the study set target to provide an extensive summary of the pros and cons of Micro Frontends as well as the selected solution.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Problem

Thanks to technological developments and advancements, the tech industry did see profound changes in Frontend development, and the need for efficient tools to maintain Frontend projects naturally arose. Of all changes, one worth being mentioned is Micro Frontends architecture style with benefits that resemble those of Microservices as a means of maintaining large web applications. Some underlying reasons for the solution are continual feature development and extension, autonomous and distributed teams' collaboration and maintenance of smaller code bases.

Decent architectures have made it much easier than ever for developers to work with a project, which is also the case in Frontend development. So far, a great variety of projects have been implemented in the architecture of monolith applications, which means a large coupling system governed by a single team in the majority of cases. This approach can be increasingly problematic if the business involves improving and adding new features on a regular basis. The company's low investment and concern in Frontend architecture, shown by the gigantic size of the code base or long release cycles, may fail to reach its goal of fast delivery in production or even hurt its rankings in the market competitiveness. Besides, any system with strong ties to a fixed technology stack can accumulate legacy code throughout the years and suffer from this issue in the future when it comes to feature extensions and software migration. Adopting current and modern technologies can serve as an important factor to, for instance, attract more developers to consider the hiring position a future work or avoid facing recruiting challenges for highly skilled occupations in the market.

This challenge also applies to the backend where a decent solution with a thriving community is identified with the name Microservices. In place of a single monolith application, the system is divided into different and independent services, which are later integrated together for scaling, maintenance, and deployment purpose. This architecture style on the backend offers a wide range of benefits such as better failure isolation, autonomous teamwork, independent deployments, clear architectural boundaries, decentralizing decisions, a faster release cycle, and faster time to market. Ultimately, a system with independent services that represent one coherent application comes to being.

The thesis aims to analyze and evaluate Micro Frontends, which follow the architecture pattern of Microservices on the Frontend side. Micro Frontends architecture is a young technology with many implementations. However, an industry-standard has not come to exist thus far. Comparison in great detail of different solutions and practical evaluation of Micro Frontends might also not yet exist. The concepts and motivations for adopting Micro Frontends as a Frontend solution will be explained theoretically and practically in the study.

1.2 Goals and Targets

The thesis will present the fundamentals and principles of Micro Frontends with a focus on dynamic Micro Frontends on the client-side and their impact on software development either architecturally or organiza-

tionally. Topics like integration and coupling of Micro Frontends, performance, routing, slot and extension, sharing dependencies, cross-framework components, distributed and vertical team structure, unit testing, best practices, and practical techniques will be addressed in detail.

In addition, the thesis not only presents to readers the diversity of Micro Frontend architecture styles but also introduces Piral as a solution. Using the study helps us picture this architecture style and assists readers with Piral learning, a relatively crucial component of Micro Frontend solutions. Piral, as a fully-fledged framework for Micro Frontends, presents the dynamic integration of Micro Frontend in client-side composition. Readers can naturally take it in by observing the Micro Frontends live beside each other.

A simple webshop system as a proof of concept will be built and investigated in detail using the Piral framework. Technology stacks used are HTML, CSS, JavaScript, React, and Piral. Requirements for studying the thesis are a basic understanding of HTML, CSS, JavaScript, and Web development. Empirical research is based on the analysis of code snippets and the sample application, which is quality-assured with unit testing.

1.3 Roadmap of Thesis

The study is structured as follows. Chapter 1 is the introductory chapter. Chapter 2 begins the study by presenting the context of this work. Chapter 3 provides a theoretical background of software architecture and centers around Microservices major principles and characteristics. Chapters 4, 5, and 6 discuss the establishment, characteristics, common problems, advantages and disadvantages of Micro Frontends, and popular application architectures. Chapter 7 is meant to examine client-side composition Micro Frontends theoretically and practically with a prototype using Piral. Chapter 8 provides a thorough walk-through of the results of the study, including a sample application, applications use cases and workflows, detailed images of the final results, and lists out the Micro Frontends components that shape the final app. Chapter 9 dives deep into implementation details and discusses technical implications of the prototypes and Micro Frontends in general, investigates them in detail, provides insight into the actual realization of the architecture, and discusses their issues. Chapter 10 highlights the selling points of Piral regarding the development process, and analyzes and evaluates the sample application empirically with unit testing. The final chapter, 11, draws the conclusions.

Chapter 2

Origin of work

This chapter provides the context of the research work.

The company that actively promotes participation in the subject is Adesso, one of the leading IT service providers in the German-speaking area with around 6.300 employees. Adesso works with a customer named Sartorius, active in the field of the biopharmaceutical industry. Sartorius adopts Piral as their Micro Frontends framework solution and collaborates with Adesso on the project. In this context, Adesso realized while Microservices has become a long-established approach, many portals or single page app (SPA) are still delivered as a monolith and therefore promoted some research on the subject matter. The micro architecture should somehow extend to the Frontend layer. In practice, Micro Frontends projects devoid of decent architecture consideration will lead to unwanted redundancy, problematic integration, and a bad user experience. Furthermore, to implement true self-contained systems with this architecture style, the UI might only be connected to each other with links for the desired true decoupling. An interesting approach, yet not widespread, is Micro Frontends framework Piral. Could the solutions make use of any number of technologies, which are Frontend frameworks like React, Vue, Svelte and their different versions? Which are the best practices and practical techniques to accomplish Micro Frontends integration and communication?

Chapter 3

Software Architecture and Microservices

This chapter provides a theoretical background of software architecture, major ideas, principles, and characteristics of Microservices for the study.

3.1 Software architecture

Software architecture refers to the fundamental structure of a software system, a broad agreement on the software's key principles, and according to Martin Fowler [6], architecture is about the important stuff. Architecture is crucial for the developer's understanding of the system, ongoing maintenance jobs, and high-quality deliveries of the product. In practice, architects design software following the primary objectives of minimizing the lifetime cost of the system in deployment and operation as well as maximizing programmer productivity in maintenance and ongoing development. Good architecture facilitates the process of performing these jobs and keeping as many options as possible for improvement, extension, and updates, according to Robert C. Martin. [7, pp. 135–137]

Development

The architecture of software should make it easy for developers to develop. In reality, there are primarily two popular cases with small and large teams. For example, projects with a handful of developers enable them to swiftly and efficiently together build a monolithic system without well-defined components and interfaces while still ensuring that everyone is on the same track. Rules imposed on developers at the beginning of the project were an impediment to such organization; therefore historically, many systems began with no architecture. On the contrary, a system with a larger size of five different teams, each consist of seven developers, is likely to make no progress without proper team structure, clear boundaries, and reliable architecture. [7, pp. 137–138]

Deployment

Deployment strategy describes the shipping process of software to production. In the past, this concern had been often neglected in the early stage of development. On current trends, deployment plays an increasingly important role in software delivery, and easy deployment with a single action is considered a goal of the software architecture. [7, pp. 138]

Maintenance

Of all strategic goals of software architecture, the worthiest being mentioned, also the most costly, is maintenance with respect to ongoing new features, defects repair, and correction execution on a regular basis, not to mention the effort of planning, tracking, and analyzing the system as a whole. A well-designed architecture should be taken as a long-lasting, productive method concerning features development optimization, risk of creating unintended impact minimization, and effective enterprise resources management.

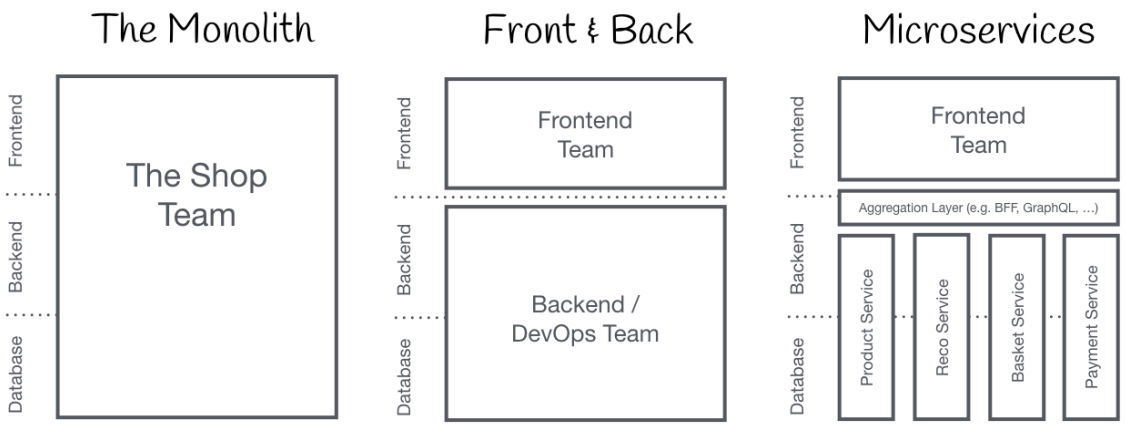


Figure 3.1. Web application architecture of an e-commerce site. Source: [1]

Operation

Operation implies the management of software as well as hardware in organizations. Though having a lower, less considerable impact on operation than in development, deployment, and maintenance, the architecture of software, if well-designed, should reveal operation and reflect the use cases, features, and required behaviors of the system for better understanding and support for developers. [7, p. 139]

As for the web, recent history has witnessed inventions and discoveries leading to great changes in the way developers think about software in the tech industry.

3.1.1 Web Application Layer

At its heart, a web application revolves around information. The information is stored in a database layer, consisting of usually relational database management systems for optimal reading, writing, and persisting data. The data are managed, queried, and updated by the second backend layer, a server-side application handling network requests, executing business, domain logic, and populating views sent back to the client. The Frontend, or presentation layer, consisting of HTML pages, CSS styling, and JavaScript for rich UI, is the user-facing interface and interacts with the client users. Changes such as input from users, form submission, or specific business use cases are persisted throughout the Backend and database layer and updated on the Frontend. Figure 3.1 shows an example of an application in different structures.

3.1.2 Monolith

At the bright dawn of the digital revolution, the monolith was the original way of developing software. A monolith is an application in which a database, server-side application, and client-side user interface are built as a single unit, is collectively owned by a handful of teams for development and deployment, and is composed of coupling layers, even subsystems that normally have complex inter-dependencies. In other words, a system with parts tightly coupled and released together is the definition of a monolith. The primary trait of a monolith is that the decision on the technology stack has to be made from the beginning of the project; in other words, the whole system functions on the same resource, including platforms, languages, libraries, communication patterns, and technical environment. Historically, monolith architecture is the natural approach to building a system. Essentially, the whole application logic and presentation, including handling HTTP resource API, processing user interface, or constructing deployment pipeline and testing scenarios, run in a single process. Consequently, changes are tied together; any change in the system, minor or major, requires the entire application to be rebuilt and published. These internal affairs have made it practically inconvenient to make changes or scale processes according to business needs.

3.1.3 Separation of Backend and Frontend

Over time, in place of a monolith that handles everything, two central teams are formed around Backend and Frontend capabilities. The traditional approach tends to favor equipping a server performing most of the rendering logic with a lightweight implementation of Frontend. The emergence of JavaScript and AJAX made it possible for the client to be less dependent on the server and created asynchronous, dynamic behavior of the web. Since then, client machines starting to possess greater computing power gives birth to client interaction advancement and feature-rich applications. As for the Backend, the data boundary was provided and contained in the application programming interface (API) layer used for page generation or handling database queries. As for the Frontend, it is completely responsible for rendering UI, handling interactions, and updating web pages with necessary resources, and the term Single Page Application (SPA) was coined.

In short, the section goes through a brief history of how software, specifically web applications, was originally built as a monolith, and the reasons behind the separation of Backend and Frontend layers, which brought much of a web application functionalities to the client-side.

3.2 Microservices

3.2.1 Characteristics and Advantages of Microservices

As more applications are being deployed on the cloud and the tech industry has become more dynamic with explosive innovation of computational technology, it is evident that modular structure and scaling development have now become, to a certain extent, universal, making it practical for applying micro architecture. In particular, some companies have been employing a systematic approach to carve up their applications into smaller parts, individual systems with independent development cycles, build process, release planning, and even internal architecture. In fact, it is more reasonable for solution architects to adopt the idea when it comes to considerable advantages of scaling parts of the system, parallel development, and organization of business capabilities subject to frequent change.

The concept of Microservices can be defined as follows, according to James Lewis and Martin Fowler:

> *"Microservices architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies."* [8, para. 3]

The primary aim of Microservices is generally for systems that come with a size reasonably large enough, well-defined organizational boundaries with explicit responsibilities [9, p. 31] and skillful teams organized for distributed systems. Some significant drivers and characteristics of Microservices are a project model with a product mentality for an ongoing relationship between developers and users, strong modularization through the pattern of change, and decentralized data management. Microservices have a variety of advantages, of which the worthiest being mentioned are strong modularization, independent deployment, and technology diversity, according to Martin Fowler. [10, para. 1–2]

Strong Modularization

In the first place, it is the modularization characteristic that defines Microservices. A software application, whether monolith or not, has always been known to developers as a large collection of modules: software building blocks and pieces that form the final system. Firstly, decoupling modules have made it much easier to work with software. For instance, as part of the job, when changing segments of the system, developers need not understand it completely other than the small part responsible for the assigned task. Secondly, as software grows in size, a good modular structure is necessary or even crucial for the project's success. Specifically, in light of Conway's Law [11], as more and more developers from different backgrounds and different locations join a project, less formal meetings and less frequent inter-team communications should be the goals, making

Microservices a perfect fit for allowing one team more or less works independently within their organizational boundary. Thirdly, module separation has always protected the software from unintentional mistakes as well as damaging workarounds of developers and enforced clear architectural boundaries. Many argue that monolith software can be built and maintained well with a good modular structure. It is understandable that a monolith can be constructed in a modular manner under conditions that good management, strict discipline, and an appropriate level of expertise are maintained. However, in reality, it is very hard to keep them, especially as team size grows, it gets progressively harder to maintain discipline and module boundaries; in fact, the pattern of Big Ball of Mud [12], and Integration Database [13] problem become more commonplace. As for monolithic software, it is a very common case that developers find it easy to cross-reference modules to each other as a careless workaround for an assigned job or apply anti-patterns due to the lack of technical competence, hence introducing coupling of services, damaging the modular structure and blurring architectural boundaries. As for distributed software, considering the fact that there are inexperienced developers of average technical ability in the team, Microservices guards against these problems of one's mistake dragging down the whole team's productivity or careless workarounds, thus re-enforces modules separation. One worth mentioning point is that in the scenario of giant projects with a rapid influx of developers, Microservices make it efficient for developers to join and get productive, easy to incorporate new people, and leverage large teams in the software than a typical monolith. [10, para. 2–11]

Independent Deployment

The second characteristic is independent deployment, which means much for a dynamic DevOps [14] culture, which has revolutionized the role of releasing software to production. Firstly, advancement in software release with DevOps concepts like continuous integration, continuous delivery has greatly facilitated code shipping. It is a very common case that in the past, the release to production phase was a painful process that took place manually, a rare event foreign to a significant number of developers, while nowadays, deployment is acknowledged to play an increasingly influential role in enterprise cooperation with skillful teams practicing continuous delivery and shipping features multiple times on a daily basis. Since many find it hard to deal with the difficulty of deploying large monoliths, where any change in part of the system causes the whole application to be deployed, Microservices serve as the medium for products to be independently deployable, where a service or component can be individually updated without interrupting the running app. Moreover, design for failure helps reduce harm to the system when a service crashes. All in all, the mentioned points help ensure a good transition from an idea to running software, swift action to market change or customer needs and are good trade-offs for the complexities of distributed systems. [10, sec. Independent Deployment]

Technology Diversity

The third characteristic is freedom of technology. Distributed systems can be written with different languages, libraries, and data stores, making it possible for teams to opt for specialized technology and supporting tools for each kind of problem. While many emphasize the need to use different tools for different jobs, Microservices, indeed, shine in dealing with the issue of versioning. In the case of one single version of a library in a monolith, upgrades are problematic in that upgrades in one part lead to incompatibility in others or break them. As the code base gets bigger, this issue becomes more devastating. Microservices allow incremental upgrades. While any decision on programming languages and technology frameworks in a monolithic system is fairly final and difficult to be dramatically altered, Microservices make it possible for teams to experiment with new tools step by step, migrate a system to superior technology, and adopt new, popular coding environments. [10, sec. Technology Diversity]

It is evident from the above interpretation that the idea of a micro ecosystem connecting individual services among a variety of business reasons does not only make significantly systematic shift in terms of decentralized governing model, strong module boundaries and technologies options, but also offer the numerous opportunities for developers to control change without slowing down software's changes, take full responsibility for the software in production and do rapid application deployment.

3.2.2 Disadvantages and Problems of Microservices

Commonly observed disadvantages of Microservices are the associated complexities of distributed systems, multiple points of failure that result in a higher chance of failure in run-time, increased orchestration complexity on the system level, difficult management in case of a large number of services, difficulties in debugging and testing, and inconsistencies between services and versioning hell.

In particular, it is the operational complexity of distributed systems and multi-services architecture that enables application failure at multiple points, increases orchestration complexity, and makes debugging and testing more difficult. Because of its nature, communication between services over networks is not instantaneous, leading to latency, extra failure handling effort due to error or unavailability of the participating services [9, pp. 6-7]. In this case, determining the major cause of an application crash, navigating buggy code, and figuring out solutions to mitigate these situations require a high level of technical expertise and familiarity with the system. For example, a developer needs to know which code resides in which repository, in which business domain a problem occurs, or have excellent technical skills to handle the failure of multiple dependent services. In another case, it is commonly observed that freedom of choice in technologies results in higher maintenance of much more frameworks and programming languages as well as difficulty in scaling development.

In addition, developers need to focus attention on the increasing number of remote calls, HTTP resources, and API requests over time or the consequence of data replication and eventual inconsistency. If multiple services are scattered, hosted, and expanded remotely, it is difficult to refactor application boundaries because multiple layers are involved, backward compatibility needs to be guaranteed, and testing requires more work, not to mention the difficulties of designing clear and correct architectural boundaries.

3.3 Summary

Web applications, which originated as monoliths, have been continuing to evolve into smaller components in response to technological advances as well as developers' and users' needs. On the foundation of the traditional approach with two central Frontend and Backend teams, greater modularization in software with Microservices has now become a significant trend indicating the need for easier maintenance, independent deployment, and freedom of technology. The chapter further provides an explanation of Microservices architecture, clarifies its distinctive characteristics in comparison with monoliths, and discusses its advantages and disadvantages of it in further detail.

Chapter 4

Basics of Micro Frontends

This chapter explores the idea of Microservices on the Frontend layer, puts together the common problem of Micro Frontends, and goes through some of their distinguishing characteristics.

4.1 Micro Architecture on the Frontend and Problems of Micro Frontends

While much has been said on the Backend, micro architecture on the Frontend appears to be at the outset. Micro Frontends, a term coined by ThoughtWorks, describes a web application as a combination of features and business domains that are owned by independent teams. A team comes into contact with the product in production with customers, owns the complete development cycle from the database to the user interface, and is cross-functional. Ultimately, systems composed of individual, self-contained and micro applications come into being on the Frontend. [1, para. 1–2]

Many may maintain that Micro Frontends solely and logically imply assembling HTML composed of different fragments from multiple Micro Frontends. However, what if the application needs to deliver assets such as JavaScript, CSS, or image files? As for the JavaScript part, how can developers add dynamic behavior to web pages among different Micro Frontends? How does the application reflect desired changes in the URL path? What if two global variables in two Frontend codes have the same name? As for the CSS part, what is the best style isolation technique? How should developers implement CSS scoping? As for hypermedia contents, what does bundle splitting look like? How should the application deliver hypermedia content? Ultimately, Micro Frontend projects should have the capabilities of delivering small chunks of data and fragments of HTML or UI components to the Frontend without interrupting the main application and propagating changes across different modules.

In contrast to the Backend, Micro Frontends have some more problems other than those from the Microservices world. In the first place, performance considerations will caution corporates and companies against the adoption of Micro Frontends during the course of doing research and developing their products. In this case, the Backend offers a variety of solutions to give every service extensive computing resources while all we have on the Frontend is the client machine, for instance, a desktop with great computing power or a small smartphone. Therefore, resource exhaustion is possible. Moreover, Micro Frontends require not only code isolation but also sharing resources such as the technology of choice and run-time framework. Micro Frontends should make style isolation possible to avoid style leaks, conflicts, and different CSS styles, as well as have HTML fragments translate into the final Document Object Model [15].

The strength of Micro Frontends, universally clarified by strong modularization and great flexibility, will lead to a number of disadvantages in the way that micro architecture is realized, which is also the case of Microservices. All in all, the advantages of Micro Frontends as well as Microservices can significantly outweigh the disadvantages depending on each specific use case, not to mention the available tooling, active community, and best practices that come with it.

4.2 Characteristics of Micro Frontends

Among fundamental conditions concerning software solutions in terms of innovation and entrepreneurship, strategic management, and technical know-how, the most influential mentioned involves the delivery of products. In particular, companies often face such emergent problems as feature extension and upgrade upon customer's request, and actually need guidelines to conform more easily to this pattern of change. Evidently, Micro Frontends lend assistance to the developers by componentization of services, independent deployment, and cross-functional teams.

Firstly, the componentization of different Micro Frontends leads to great changes in the code base structure. They tend to be simpler, easier to understand business as well as technical contexts, and continuously reinforce the commitment to clear architecture boundaries, business functionalities, and intentional decoupling between services. In addition, this characteristic allows developers to upgrade and deliver features, new or existing, independently, automatically, and freely without breaking the present working API contracts or being disturbed by the legacy monolith. Finally, incremental upgrades of internal architecture, growing dependencies, and user experience are isolated and straightforward.

Secondly, as more and more systems are being deployed to the cloud, Micro Frontends have a crucial role in creating individual continuous integration, continuous delivery pipelines. When a user interface element or subsection of a web page finishes its latest development iteration, it is allowed to be published to the production environment in no time, resulting in a much shorter release cycle and dynamic behavior in product delivery.

Thirdly, cross-functional teams having a shorter decision-making process, possessing a piece of the software, and owning a complete product development cycle promote a design around business capabilities and users. Teams will find it easier to become accustomed to a new working environment and collaborate better in teamwork. Decentralized governing model, technology freedom, and autonomous teams also adopt agile methodologies and embrace distributed work culture in companies instead of a monolithic one.

All in all, Frontend to be arranged for micro architecture benefits developers in the way they interact with others and give them a complete orientation to their job in terms of decoupled code bases, considerable autonomy, and effective management of change.

4.3 An example of Micro Frontends

Excerpt from the final sample application, figure 4.1 is a web page consisting of four teams; each has a separate mission. Team red is responsible for presenting the products and product details. Team blue provides a good checkout experience with shopping cart functionality. Team yellow uses an artificial intelligence mechanism to help customers discover new products. In this case, the UI elements or fragments from different Micro Frontends such as the purchase button, recommendation products, and further updates handling mechanisms come from the remote servers hosting them. Finally, all three Micro Frontends live in an application shell, the gateway entry to access the web application. The application shell is also a Micro Frontend, responsible for the overall design and general functionalities, including navigation, header, menu items, and footer.

Figure 4.1. First look of the incomplete integration of Micro Frontends. Source: own illustration

4.4 Summary

This chapter helps readers understand the defining characteristics of Micro Frontend, how they fit in the latest trend of slicing a web app into micro applications, and differentiates problems of Micro Frontends on Frontend from those of Microservices on Backend in aspects such as running environment in client's browser, assets management, performance and integration challenges. Distinguishing characteristics of micro architecture on Frontend are discussed in detail and wrapped up with an exemplary model, which helps picture this design pattern in practice.

Chapter 5

Micro Frontends Application Architecture

This chapter presents existing architectural options for building Micro Frontends.

Micro Frontends break an application into smaller code bases and have them integrated into one final piece in production. Though monolith application has become the industry standard, dividing the application this way still offers remarkable benefits in scaling Frontend development, scalable organizations, and code maintenance or features extension ability. Following Micro Frontends approaches will help clarify the architecture's nature.

5.1 Static versus Dynamic Micro Frontends

Surprisingly, the static approach is a fully static usage of Micro Frontends. It breaks down the application into several packages, which are subsequently merged at build time. The obvious advantage the static approach enjoys is that all information is known at build time. All Micro Frontends are made available during build time by static imports. This leads to early capture of potential errors, deeper integration with few run-time dependencies, and the assurance that we have a robust, working, and executable code. The main disadvantage, also a defining characteristic, of a static approach is that changes in any part of a Micro Frontend, or package, require a rebuild of the whole application, resulting in central and dependent deployments. [2, pp. 68–71]

> *"The primary use cases of static micro frontend solutions are slowly changing websites or smaller web applications. One example framework here is Bit [16]."* - Florian Rappl

On the other hand, a dynamic approach leads to great changes in the development cycle. Of the most influential changes taking place in the application are those in publishing Micro Frontends to a source, their source updates, and connecting an application to a Micro Frontend. This complexity and loose coupling are the underlying cause of a much more fragile application. Additional tooling and error boundary handling further increase the complexity on the infrastructure level. However, the dynamic approach starts to shine in that it selects and updates each Micro Frontend independently during run-time or on a per-request level without interrupting the core running application, hence giving the developers a lot of freedom. [2, pp. 68–71]

> *"The primary use cases of dynamic micro frontend solutions are personalized websites or larger web applications. One example framework here is Webpack's Module Federation [17]. A bundler such as Webpack produces one or more JavaScript files for each micro frontend."* - Florian Rappl

It is evident from the above interpretation that the static approach has made it practically convenient for a Micro Frontends implementation which only requires some boilerplate code, no lengthy infrastructure configuration, or sophisticated integration. In contrast, a dynamic approach will be of concern for their independence, flexibility, and a lot of freedom at a reasonable corresponding cost of difficult implementation. Looking at the matter from different angle, decision on final architecture solution can be made by considering the anticipated development team structure.

5.2 Horizontal- versus Vertical-composed Micro Frontends

Continually increasing the project's scope and team size has made it best practice to divide the software into horizontal layers with a Frontend team and one or more teams on the Backend. In reality, Micro Frontends not only mean an unchanged technology architecture but also define an alternative organizational approach: dividing the application into vertical slices where each slice has a dedicated development cycle built from the database in the Backend to the Frontend user interface. The long-established horizontal and relatively new vertical approaches are discussed as follows.

The following screenshot illustrates a typical horizontal approach:

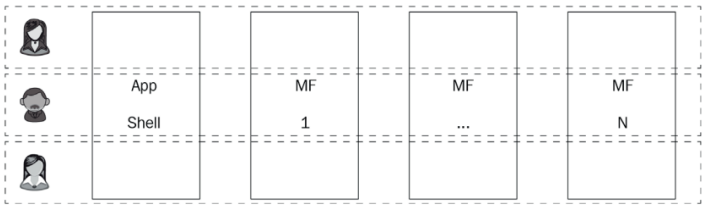


Figure 5.1. Horizontal teams setup. Source: [2, p. 210]

The figure shows that software is divided into multiple pieces where multiple Micro Frontends are developed, tested, and deployed in isolation per page like an isolated web application, with teams formed around technical capabilities. Many maintain that focusing on providing ready-made pages with this structure is the best policy to reason with and picture the whole project. Besides, it is reasonable that a single page of the application with teams formed around technical or horizontal concerns like styling, forms, validation, pure API, or operation teams will do justice in terms of technology specialty. However, what really matters is how well and to what extent a horizontal approach can scale. The answer is it is not the case: a horizontal approach requires close coordination of teams for every web view and does not advocate multiple Micro Frontends to share and reuse certain parts among one another. [2, pp. 71–73]

> *"The primary use cases of horizontal micro frontend solutions are content-heavy websites or single-use-case pages. One example framework here is Podium [18]."* - Florian Rappl

In contrast, as one Micro Frontend and one team are formed with respect to business domains rather than technical capabilities, vertically arranged software systems have cross-functional teams solve their problems, perform their tasks requiring only knowledge of a single subdomain and make them responsible for the complete software's stack from top to bottom. Then, different Micro Frontends are assembled in the client's browser to form a final web page, and one business domain is owned by one team; for instance, authentication process or shopping experience adhered to the principles of Domain Driven Design [19]. Consequently, multiple Micro Frontends can be displayed on the same web page; one Micro Frontend can also serve as a web component extension inside others. The very first characteristic to clarify this nature lies in how the teams develop, upgrade and extend features. For instance, when opting for a certain feature extension, the responsible team already has all the specialists it needs to perform the task without any advanced agreements with other Frontend or Backend teams. In addition, code upgrades no longer remain problematic. Since each team owns its complete stack, they can independently decide on their software updates, technology switch, and resolve framework or dependency version. Moreover, teams acquire end-to-end ownership of everything they need to deliver customer value, making user-centered design an integral part of the vertical architecture. For example, every team encapsulates a product's feature, a customer journey, or a single page of the application that the end-users see from ideation throughout the production process and ships the feature directly to the customer, hence removing the need for pure Frontend or deployment teams. The typical vertical approach offers remarkable benefits in terms of features development optimization, splitting problem domain into smaller parts, and customer focus with respect to project management and organizational structure. [3, pp. 6–22], [2, pp. 71–73]

One major drawback of a vertical approach is the direct impact on software developers. As the application is broken down in pieces, developers have to carry out debugging tasks of multiple Micro Frontends simultaneously and may have trouble visualizing the final product. In this case, providing a system that can be debugged and extended well is the most challenging task. [2, p. 73]

> *"The primary use cases of vertical micro frontend solutions are large web applications and web portals. One example framework here is Piral [20]."* - Florian Rappl

The following figure illustrates vertical approach:

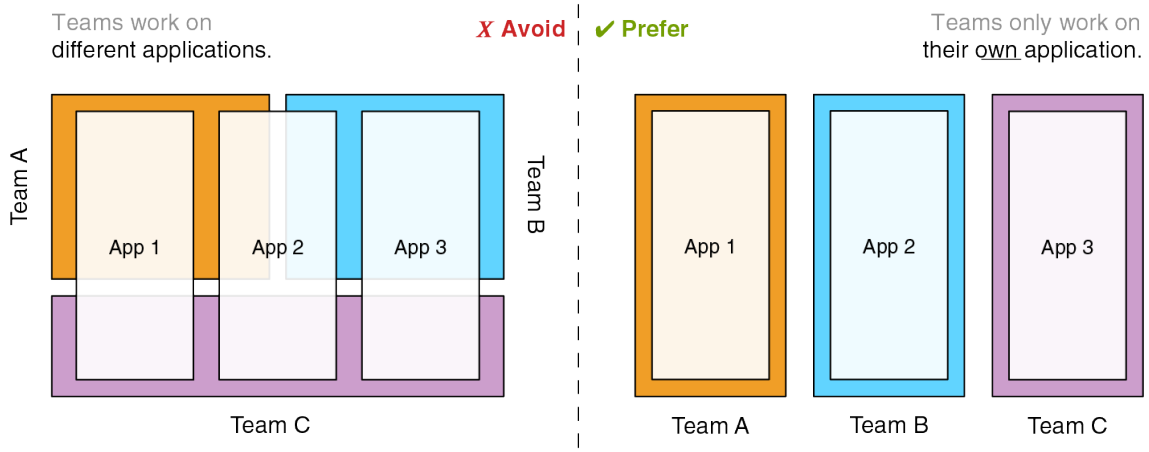


Figure 5.2. Vertical teams. Source: [3, p. 238]

It is reasonably assumed that vertical-composed Micro Frontends have a crucial role in user-focused missions and cross-functional teams. In particular, it helps promote agile and independent teamwork, ensure a plausible interpretation of the related domain of the application and create a dedicated continuous integration, continuous delivery pipeline, while the horizontal design has been widespread as a long-established approach with teams formed around technical specialties and perspective.

5.3 Backend- versus Frontend-driven Micro Frontends

Micro Frontends differ from each other in build time, run-time, per-view, in-view, etc. among these properties, the key factor that makes the most fundamental difference among them is the decision between Backend and Frontend as the area of composition. As a matter of fact, many nowadays like to see Micro Frontends as a real Frontend game-changer without any changes in the backend, while others tend to stick to the server-side-composed solution, enjoying a familiar architecture and transferable knowledge coming from the Microservices world.

Server-Side Rendering (SSR) was the original technique for making dynamic websites possible. Naturally, server-side Micro Frontends were one of the first Micro Frontends implementations. The primary reason for this is that necessary technology has long existed, that is, Server Side Includes (SSI [21]) and its successor Edge Side Includes (ESI [22]). The main advantage of backend-driven Micro Frontends is the fast and seamless delivery of Micro Frontends in the context of server-to-server communication. In this case, the client browser receives ready-made web pages without additional integration and assembling techniques. Moreover, this approach does not require JavaScript to be active on the client and is friendly for search engines and crawlers. Finally, debugging is trouble-free, and developers might make use of existing Microservices infrastructure knowledge. The challenging task the backend approach faces is building complicated infrastructure to help ensure scaling development and high reliability. One worth mentioning point is that deployment takes place during build-time due to the server-side rendering nature. [2, pp. 73–76]

> *"The primary use cases of backend micro frontend solutions are e-commerce websites and content portals. One example framework here is Mosaic 9 [23]."* - Florian Rappl

On the other hand, many argue that it is the client-side Micro Frontends that can help developers accomplish the most flexibility. It is reasonable that any UI components and pages can be used regardless of framework, technology, server- or client-side strategy. However, this statement dwarfs the importance of employing a Backend that comes with some powerful capabilities of optimizations and enhancements. In this case, mixing some Backend capabilities into the Frontend may be a solution. The daunting task of this approach is performance consideration due to the integration, composition, and assembly of different Micro Frontends in the user's browser. It always takes time to work and requires a high level of technical expertise. [2, pp. 73–76]

> *"The primary use cases of frontend micro frontend solutions are tool-like experiences and web applications. One example framework here is Single-spa [24]."* - Florian Rappl

On balance, frontend-driven Micro Frontends have reinforced the advantages of being flexible and independent regardless of any chosen UI framework and rendering mechanism, even though backend Micro Frontends may exclusively allow a few great optimizations as well as enhancements by nature.

5.4 Summary

This chapter presents different approaches to designing Micro Frontends applications in terms of build time, run-time, per-view, in-view, server, and client composition. The first section presents static and dynamic Micro Frontends, which determine construction either in build time or run-time of the final application. The second section focuses more on the development team's structure with horizontal-composed in comparison with vertical-composed Micro Frontends. The third section discusses server and client as the area of composition where multiple Micro Frontends are integrated into one single application. From the sections as mentioned earlier, it is evident that the exact implementation of Micro Frontends will be defined differently to different concerns wearing different viewpoints: either taking one or another will depend mainly on each and every individual case, situation, and circumstance projects are in; still, the primary area of use for each specific type is clearly categorized in the chapter.

Chapter 6

Micro Frontends Advantages and Disadvantages

This chapter presents Micro Frontends' advantages and disadvantages in greater detail.

6.1 Advantages of Micro Frontends

Apart from the close resemblance to Microservices (section 3.2.1), Micro Frontends come with the following significant advantages.

6.1.1 Decrease On-boarding Time

The onboarding process should always take place, whatever the cost. Incorporating new developers has become not only essential but also impacted the overall progress of the project productively and required a significant amount of enterprise resources in terms of time, money, and workforce.

> *"20 years ago, most developers that entered a company stayed for quite a while. Over time this decreased, to be around 2 years for most companies. If a standard developer gets productive in a larger code base after 6 months, this means that a large fraction of the investment (about 25%) is not really fulfilled."* - Florian Rappl

It is reasonable that incorporating new employees effectively and efficiently with a generally high hiring bar and often large code base should be a realistic aim. Then, strong modularization with independent repositories in Micro Frontends helps increase the cognitive ability of developers during the onboarding process to understand the code base. Needless to say, for a new developer, working with a smaller repository aids them in the learning and comprehension of the whole project, bug fixing, and troubleshooting of unusual situations. Besides, working with a smaller commit history in each repository lowers the entry level because of well-maintained and up-to-date documentation. [2, p. 16]

As for large monolith applications, according to James Lewis and Martin Fowler, they can also be modularized in alignment with business capabilities. However, the whole system will turn out to be organized around too many contexts, hence making team members struggle to fit all of these into their short-term memory. In addition, the modularization strategy requires a great deal of discipline in the case of a monolith, while it can be effectively enforced by the componentization of services in a micro architecture pattern. [8, sec. Organized around Business Capabilities]

One worth mentioning pitfall is that complex bugs involved multiple points of failure scenarios require in-depth technical expertise to solve. Extensive experience with the system, familiar with business use cases and knowing which code lives in which repository are the prerequisites for Micro Frontends debugging. Dividing teams with clear architectural boundaries, dedicated responsibilities, and shared infrastructure knowledge can effectively solve this problem.

6.1.2 Isolated Features and Optimization for Feature Development

The ability to ship independent features will help ensure that engineers develop features upon customers' requests in an optimal way.

In the first place, using Micro Frontends will enable teams to roll out features progressively while any new extension in a layered architecture comes out as a result of an all-sided consideration, including a series of alignments, as complicated as how meetings are arranged, changes discussed, and specification is written. In other words, only one team, instead of multiple teams, is involved in building a new feature, no communication or alignment effort is desired among different teams, and there is no need for optimization or prioritization discussions. For example, assuming in a monolith Frontend with Angular, if many customers request upon feature extension, a feature can be built and compiled separately, then, developers find a way to integrate the new feature later to the main app with little or without inter-teams agreements, alignment and coordination, either upfront or before publishing to production.

Micro Frontends architecture turns out to be robust in this particular aspect of isolating different parts of the application where the application still technically works even though a Micro Frontend is out of service. Consequently, this allows one Micro Frontend to interchange with another.

6.1.3 Independent Upgrades

Since each team owns its complete stack, they can independently decide on their software updates, technology switch, and resolve framework or dependency versions without coordination with other teams or with minor adjustments to inter-team conventions. As for monolith application, this switch is conducted on a much larger scale with a higher risk of incompatibility and inadvertent breakage, significant expenses in terms of time and money, and lots of business and technical agreements. Micro Frontends enabling the splitting of a monolith into smaller micro applications makes it possible for a specific part of the software to evolve over time.

6.1.4 Autonomous Teams

In the context of Micro Frontends, autonomous teams help achieve three important goals.

Firstly, one system is owned by one team. During the course of development, teams come up with ideas, implement them actively, respond swiftly in unexpected situations and circumstances, and learn to be committed to themselves and their own actions regarding their independent tasks. This leads to more straightforward teamwork and, in turn, cuts out the need for central Backend or Frontend teams. One worth mentioning point is that for instance, in place of a monolith Frontend that aggregates different services from the backend, a vertical architecture with the introduction of full-stack teams covering one backend service and one Frontend module can prove to be a good pattern for establishing clear architectural boundaries and reinforcing domain driven design [19] principles. The beauty of this approach is strong modularization of the system, avoidance of the infamous spaghetti code, decrease in cognitive load of code for a programmer, and alignment requirements among teams.

Secondly, in production, each system can function when the neighboring systems are down. Each sub-system can maintain its own data, hence removing the need for much communication on the system-to-system level or exchanging API requests.

Thirdly, facing such emergent problems as developing new features quickly upon customer's request while maintaining employment stability of new developers, there have been calls among corporates to acquire an enduring and costly-effective means of efficiently leveraging human resources. By making strong modularization architecture the key, incorporating new developers into a part of the system should have positive impacts on project management, resource distribution, and business expenses. In addition, teams' ownership of code repository, build process, and release schedule help solve problems deriving from challenges of collaborating with external teams or freelancers, and recruiting new talents in the tech industry, which has become absurdly difficult in recent years.

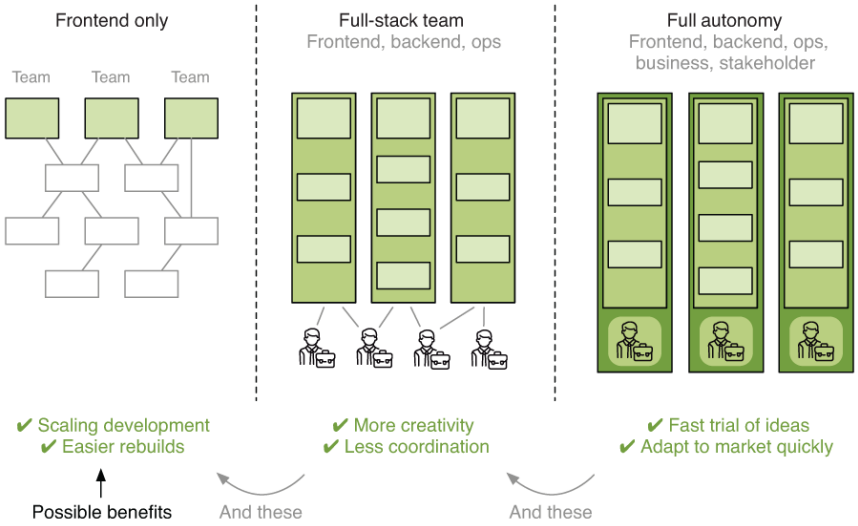


Figure 6.1. Autonomous full-stack teams. Source: [3, p. 240]

6.1.5 Faster Time To Market

From the above-mentioned points, Micro Frontends architecture makes it much easier than ever for teams owning the complete life cycle of a product, including the development process, release schedule, and customer collaboration. Products in place of projects forming a binding agreement with developers and user-centered design gives faster time to market concerning business reasons.

6.1.6 A/B Testing

Modularization architecture has made it practically convenient for A/B testing scenarios, greatly facilitating user research process. In this case, only certain parts of the application are informed about changes to reorganize their structure, activate and switch specific features for user feedback gathering, while in a monolith, code changes may occur repeatedly and scatteredly across the software. [2, p. 18]

6.2 Disadvantages of Micro Frontends

Apart from the close resemblance to Microservices (section 3.2.2), Micro Frontends come with the following significant disadvantages.

6.2.1 Increased Payload Size

Assembling web pages, HTML fragments, and UI components from multiple teams ends up in a larger download size of Frontend code for client users. A universal pattern of lazy loading and sharing dependencies should be applied to avoid downloading untouched resource and decreasing web performance, and can reduce redundancy without introducing tight coupling between teams.

6.2.2 Code Duplication in the Frontend

Micro Frontends based websites typically introduce redundancy as more isolated UI components will require more JavaScript and separate styling CSS code.

6.2.3 Redundancy

Multiple teams always mean collective redundancy. Every team takes care of their own stack, including setting up the application server, maintaining the build process, and probably shipping duplicate scripts, styling to the

browser. Moreover, developers will find it hard to synchronize with each other in exchanging best practices, knowledge, or troubleshooting. For instance, a security breach discovered in a hugely popular library makes it a practical reason for development teams to all fix and update their source themselves as this bug cannot be fixed centrally in a distributed system. Additionally, suppose one team manages to optimize their build process for rapid execution with fewer resources. In that case, they have to share this information with other teams so that they can repeat the process in their settings to gain from the enhancement. Practically, impacts created by independent teamwork and inter-team loosely coupling play a greater part in building products than those costs associated with redundancy. [3, pp. 17–18]

6.2.4 Consistency

In the settings of a distributed system and independent databases, problems arise when one team needs to request data from another. One common use case is an e-commerce site where products are the data that are shared universally, often replicated using an event bus or a feed system among different services. Each service possesses its own storage of shared data. The data replication scenario draws attention to issues such as considerable latency when a service is down or guaranteed data consistency. [3, p. 18]

6.2.5 Heterogeneity

When all teams opt for the same stack, communication pattern is less of a hassle, developers can switch teams occasionally, and exchanging knowledge is straightforward. If the freedom of technologies leads to practical usage of a significantly large number of Frontend frameworks and tools, advantages in a one-technology-stack setting are hard to keep. Accurate assessment and evaluation depend on each and every individual case. [3, p. 18]

6.3 Summary

This chapter examines the advantages and disadvantages of working with Micro Frontends theoretically in great detail, along with discussions in many individual cases on which readers can consider the value and adoption of Micro Frontends as an architecture solution.

Chapter 7

Siteless UI Micro Frontends with Piral

In this chapter, Micro Frontends solution Piral is presented in the context of Siteless UI. Siteless UI is the official term and name used by Smapiot [20] referring to vertical-composed Micro Frontends. The final solution is Piral, a framework suitable for building portal applications.

7.1 Basics of Siteless UI Micro Frontends

Siteless UI is based on a plugin architecture. The plugin architecture is a pattern that allows the addition of external components or third-party plugins, consequently creating boundaries across which changes cannot propagate [7, pp. 170–173], one excellent example is Visual Studio Code [25]. As a plugin architecture, Siteless UI has the characteristics of flexibility and consistency and provides run-time-driven Micro Frontends. At its heart, Siteless UI uses an application shell as the gateway to the application to orchestrate the whole communications, activities, and interactions of other Micro Frontends. Optimally, the app-shell should not know how to resolve the Micro Frontends in favor of delegating this task to a dedicated service to load these modules remotely. The integration of each module is encapsulated in each Micro Frontend. The following figures picture the architecture of the Piral framework as a Siteless UI pattern.

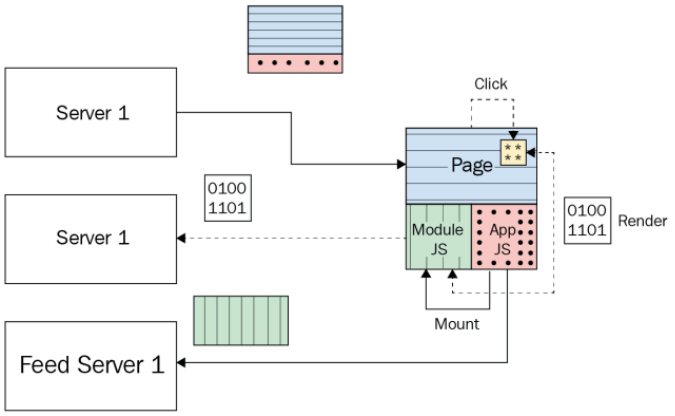


Figure 7.1. Siteless UI architecture. Source: [2, p. 75]

The significant difference in this pattern is the separate, independent feed server from all other modules used to discover the individual Micro Frontends. As seen in the figure above, the app-shell first fetches, loads Micro Frontends modules from the feed server (”Feed Server 1”), which is different than the data sources (”Server 1”), and can initialize (mount) and destroy (unmount) during run-time. Instead of behaving like a completely independent micro application, the Micro Frontends are integrated into the main running app as modules (”Module JS”) within the execution context of the app-shell. Further interactions will be passed down to the involved Micro Frontends and dedicated services to communicate with respective sources, for instance, ”Server 1”. In

Piral architecture, the whole life cycle of each service is realized by the setup of a plugin. In particular, each Micro Frontend exposes a `setup` function. For example, Micro Frontend `pilet-recommendation` exposes its `setup` function with `app.registerExtension` for extension registration, that is providing a reusable UI component to others.

```
import * as React from "react";
import { PiletApi } from "app-shell";
import { Recommendations } from "./components/Recommendations";

interface RecommendationExtension {
  // ...
}

export function setup(app: PiletApi) {
  const emitEventData = (id) => {
    // ...
  };

  app.registerExtension<RecommendationExtension>(
    "recommendations",
    ({ params }) => (
      <Recommendations category={params.category} navigate={emitEventData} />
    )
  );
}

// ...
```

Listing 7.1. Extension registration in Piral

Every Micro Frontend exclusively provides its own integration. The app-shell is the aggregated service and reasonably provides dedicated functionalities and universal UI parts such as registration of navigation or main layout. During the setup phase, apart from special works such as singleton components or shared dependencies, developers rarely see any interaction between different Micro Frontends.

7.2 Advantages of Siteless UI

Siteless UI offers a universal pattern to deliver micro applications with built-in API for crucial Micro Frontends functionalities, standard interfaces for different modules, user experience formed around the layouts, and great flexibility.

7.2.1 Local Development

Local development has a positive impact on developer experience. Going for Siteless UI, without acquiring excessive support from a senior, understanding techniques to resolve different URLs or inject modules into a running instance in a special environment, with Piral, a developer is able to clone the repository and start a local debugging session with `piral debug` (section 8.3.3) and `emulator` package[1].

7.2.2 Publishing Modules

DevOps landscape continually adopts new ways of deploying the application. Siteless UI goes for application delivery in the form of zipped tarball or tar files, using `tgz` extension. Its distinctive benefit is the format's adoption by `npm` tool, one of the most important tooling in JavaScript community. Furthermore, in the case of Piral, it has a production-ready implementation of the feed server called Piral Feed Service[2] for hosting Micro Frontends pieces. Developers can quickly use, deploy and test their applications, especially for the purpose of evaluating Micro Frontends solution or trying out a proof of concept. [2, pp. 188–189]

[1] Reference: `https://docs.piral.io/reference/documentation/emulator`
[2] Reference: `https://portal.piral.cloud/`

7.2.3 Built-in Run-time

A run-time has a crucial role in local development as well as in production. It orchestrates the application and takes care of aspects like automatic provisioning, caching rules, and run-time optimizations. In the case of Piral, the framework providing this technical solution makes it easier for the development process and for developers to focus on the actual Frontend development.

7.2.4 Development Cycle

Piral makes it convenient for developers to develop and publish micro applications as well as provides mature solutions of feed service and debugging tools. Detailed setup and techniques will be discussed throughout the sample application later in the study.

7.3 Disadvantages of Siteless UI

While monolith comes with great debugging tools, Siteless UI is hard to debug, not to mention the development difficulties. The trouble lies in the proper setup, and local development. It requires special development mode on a live instance or even with other modules in the production environment.

This pattern leads to a strong coupling between the application shell and other modules. The worse will become the worst when changes in the API are not informed to the consumers, and problems will not arise until the deployment of the app-shell, as the system is integrated at run-time and developers easily neglect the matter during compile time.

Due to its nature, client-side composition or Siteless UI is difficult to render on the server, hence unfit for information-driven sites such as e-commerce or newspaper. Its primary use case is large interactive systems and web portals.

7.4 Introduction to Piral

7.4.1 What is Piral?

Piral is a development framework, primarily built on React, Typescript, and Node.js for building Micro Frontends solutions. As a framework, Piral provides the framework for distributed web applications with modularized structure, a collection of libraries covering a wide variety of features, and a suite of developer tools helping to develop, update and deploy code.

7.4.2 Piral Technologies

Important statement regarding Piral technologies is defined as follows.

> *"Piral does not start from zero. The stack that is used by Piral is React-based. Nevertheless, the API supports any kind of framework, as long as it can work with an arbitrary element to render it into. Piral itself is based on React and its eco-system, e.g., React DOM (to render on a website), React Router (for routing), React Atom (global state management using React.Context) and a React independent building block Piral Base (which allows loading modules at runtime)."*
> [4] - Smapiot

7.4.3 Piral Concepts

Piral Instance, Application Shell

An application shell is the entry point of the whole application, structures the basic layout (including header, navigation, footer, etc.), hosts shared components, and defines the loading and integration process of the Micro Frontends (Pilets).

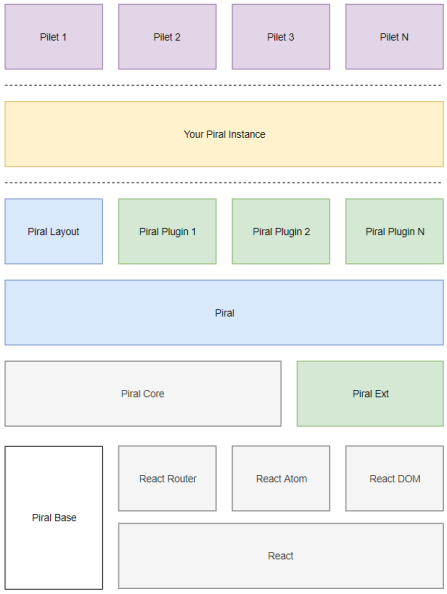


Figure 7.2. Piral building blocks. Source:
`https://docs.piral.io/reference/documentation/architecture`

"A Piral instance builds the application shell and as such the foundation for executing pilets. All central and shared functions like layout, navigation menus or notification handling will be configured in the Piral instance. In the end, the app shell is the foundation for the whole frontend. The app shell is the top layer, which may (later on) hold other shared libraries or the shared UI components. The modules are then built later." [4] - Smapiot

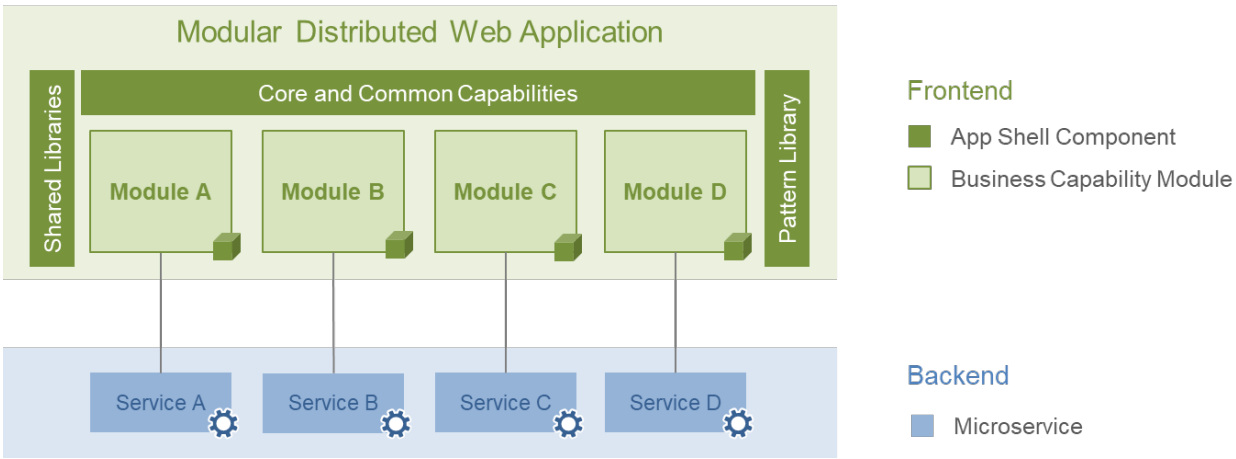


Figure 7.3. Application shell and the micro applications in the distributed system. Source: [4]

Pilets, Feature Modules or Micro Frontends Entities

Pilets are the Micro Frontends responsible for their functionalities, use cases, etc., including their own assets and dedicated dependencies, and provide their components to the app-shell or use available components made available by the app-shell.

Pilet Setup Function

The `setup` function is the default configuration for a Pilet, defined with an example as follows

> *"There is a single function, which controls the configuration of a pilet - it is the setup method in the config file index.tsx. The scaffolding process will add the setup function with some configurations: "* [4] - Smapiot

```
export function setup(app: PiletApi) {
  app.showNotification('Hello from Piral!');
  app.registerMenu(() =>
    <a href="https://docs.piral.io" target="_blank">Documentation</a>);
  app.registerTile(() => <div>Welcome to Piral!</div>, {
    initialColumns: 2,
    initialRows: 1,
  });
}
```

Listing 7.2. Default setup function in a Pilet

> *"Every Pilet receives an object called the PiletAPI, which provide it access to the common Piral instance in the app-shell. The PiletAPI provides a series of useful methods for setting up and configuring a pilet. For example, the scaffolding registers with the method registerTile a tile for the dashboard and a menu entry with the method registerMenu."* [4] - Smapiot

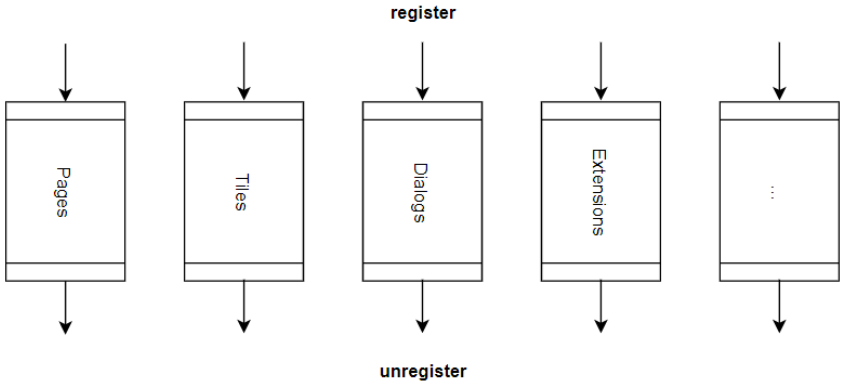


Figure 7.4. Piral standard API. Source: [4]

Piral Feed Service

The Pilets, or different Micro Frontends can be published to a common feed, from which the application can load modules into the local development for developing and testing purposes as well as into production. This is the feed server in figure 7.1. In Piral, there is a mature solution for the feed service: Piral Cloud Feed Service[3].

In this case, the web application including the app-shell and Pilets can be debugged in the emulator on development machine. All steps are built-in and made available in Piral with dedicated tooling as well as documentation. The following diagram illustrates the overall workflow of setup process in Piral:

[3]Reference: `https://portal.piral.cloud/`

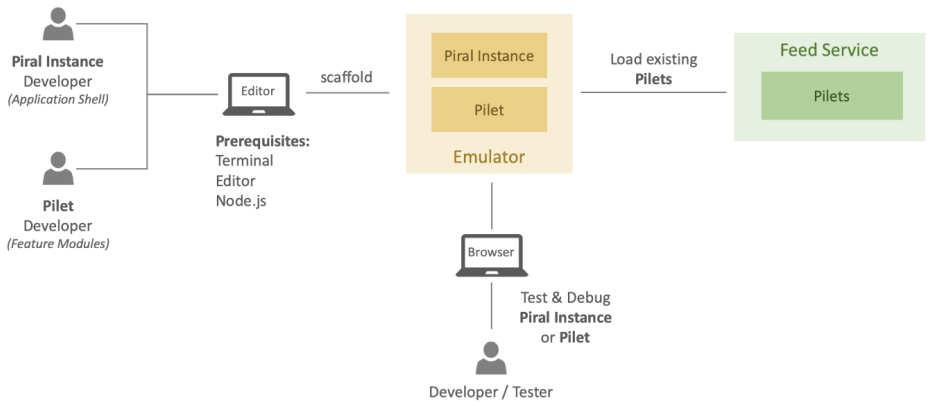


Figure 7.5. Piral setup process. Source:
`https://docs.piral.io/guidelines/tutorials/01-introduction`

7.5 Siteless UI implementations with Piral

The primary trait of Siteless UI Micro Frontends is the requirement of an application shell to bootstrap and assemble the whole application in the browser. There are, essentially, three issues to deal with: routing mechanism for activating pages, sharing dependencies and sharing components, for instance, UI extensions or global state.

7.5.1 Pilets Details

The implementation of Piral is realized by the sample application, which consists of four Micro Frontends, including the application shell and three Pilets. Each Micro Frontend is responsible for specific segments of the final application, listed as follows.

Table 7.1. Pilets and their UI elements

Pilet	Frontend Fragments
app-shell	1) Overall layout 2) Header 3) Menu items
pilet-products	1) ProductsPage (page/component) 2) ProductDetailsPage (page/component)
pilet-recommendation	1) Recommendations (component extension)
pilet-carts	1) BuyButton (component extension) 2) CartPage (page)

For a complete and in-depth understanding of the sample application, comprehensive analysis and coverage of the code base, including activating pages, client-side routing, sharing dependencies, cross-framework components, communication patterns, and integration techniques, are presented in subsequent chapter 8 and 9.

7.6 Summary

The first part of this chapter discusses Siteless UI, a client-side composition Micro Frontends where the micro apps are dynamically loaded, assembled, and managed within the browser context. The second part presents Piral as a solution to this approach, its concepts, terminologies, available suite of development as well as production tools and how it greatly fits in the concept of dynamic Micro Frontends on client side. The last part briefly goes over the implementation of Piral with its four micro apps, which are discussed in detail in subsequent chapters 8 and 9.

Chapter 8

Micro Frontends Sample Application

This chapter is a walk-through of a simple web shop system, the result of the study, built for empirical evaluation and analysis of Micro Frontends.

8.1 Introduction to Wild Orchard Store

Before diving deeper into the complete implementation of Micro Frontends architecture, it is essential to have a common, shared understanding of the "Wild Orchard Store", a prototype serves as a proof of concept for the study with the aim to illustrate Micro Frontends techniques in a practical fashion. "Wild Orchard Store" [1], an imaginary company, is originally, dedicatedly, and solely created for the thesis. While there are a few demos of Micro Frontends public on the Internet such as Netflix clone[2], the famous Tractor Store[3] and Piral Tractor Store[4], a detailed tutorial of Micro Frontends in practice in the scale like the one of the study appears to have yet to exist.

"Wild Orchard Store" manufactures organic, regenerative and high-quality teas of various categories, including green, red, black teas and matcha. The "Wild Orchard Store" is their customer-facing e-commerce site delivering all their products. The company adopts Micro Frontends architecture and builds the web application from scratch. "Wild Orchard Store" website starts with four multiple teams:

- App-shell team
- Products team
- Recommendation products team
- Cart team

and three web pages:

- List of products page
- Product details page
- Cart page

All images used in the prototype originate from the original site[5], CSS styling was mostly taken and copied from the Piral version of Tractor Store[6] and footer UI from Netflix clone demo[7]. Though having similar

[1] A real company from the US and South Korea, active in the field of organic teas products. Source: `https://wildorchard.com/`
[2] Source: `https://netflixclone.deruwe.me/`
[3] Source: `https://micro-frontends.org/0-model-store/`
[4] Source: `https://mife-demo.florian-rappl.de/products`
[5] Source: `https://wildorchard.com/collections/all-products-new`
[6] Source: `https://github.com/piral-samples/piral-microfrontend-demo`
[7] Source: `https://github.com/DanteDeRuwe/netflix-piral`

names, the sample application and the original site are completely unrelated.

To illustrate the techniques of Micro Frontends, the site will be built in Siteless UI with Piral. Following figure presents the heart of the sample application, a product details page.

Figure 8.1. Product details page. Source: own illustration

8.2 User Stories and Use cases

8.2.1 Use Case Diagram

Figure 8.2 illustrates the use case diagram of the prototype.

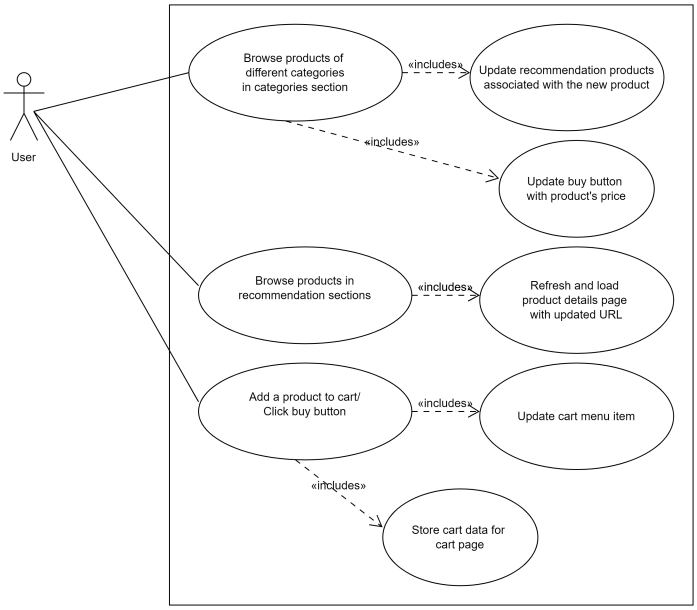


Figure 8.2. Three application use cases. Source: own illustration

8.2.2 User Stories

1. Browse products of different categories in categories section: when users click to an image in other categories section, related products section and buy button section are updated
2. Browse products in recommendation sections: when users click to an image in related products section, users are redirected to the new product details page
3. Add a product to cart/ Click buy button: when users click buy button, the cart menu item is updated, if users moves to cart page, the stored cart data are showed

8.3 Piral Sample Application Walk-through

8.3.1 Software Description

The Wild Orchard Store has four repositories:

- One repository for the app-shell orchestrating the Micro Frontends: app-shell
- One repository for the products Micro Frontend providing list of products page, product details pages, components and functions: pilet-products
- One repository for the products Micro Frontend providing cart pages, components and functions: pilet-carts
- One repository for the products Micro Frontend providing related products components and functions: pilet-recommendation

All three Pilets are published on Piral Cloud Service (section 7.4).

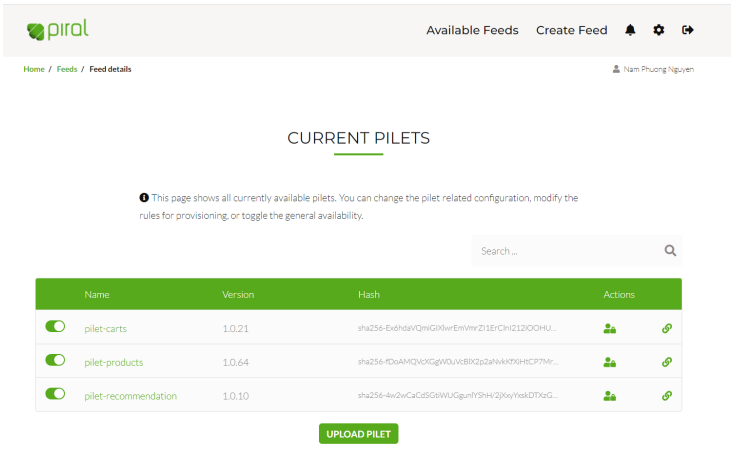


Figure 8.3. Piral cloud feed service. Source: own illustration

8.3.2 Software Documentaion

The repository was shared and uploaded on Github. As part of the thesis, a dedicated public Github monorepository was created and accessible on `https://github.com/namphuong2217/micro-frontends-webshop-piral-sample`

8.3.3 Software Installation

After successfully bootstrapping the repository, we could start the application with command line as follows.

```
# Install the Piral CLI
npm i piral-cli -g
# Check version of the Piral CLI
piral --version
# clone repository
git clone https://github.com/namphuong2217/micro-frontends-webshop-piral-sample
# change to app-shell directory
cd app-shell
# install dependencies if node_modules not installed
npm install
# starting app-shell
piral debug
```

Listing 8.1. Installation step by step

8.3.4 Architecture Diagram of the Sample Application

The app-shell is the gateway of the application. App-shell initially and consequently uses, loads and updates different Micro Frontends parts, they are `pilet-products, pilet-recommendation and pilet-carts`.

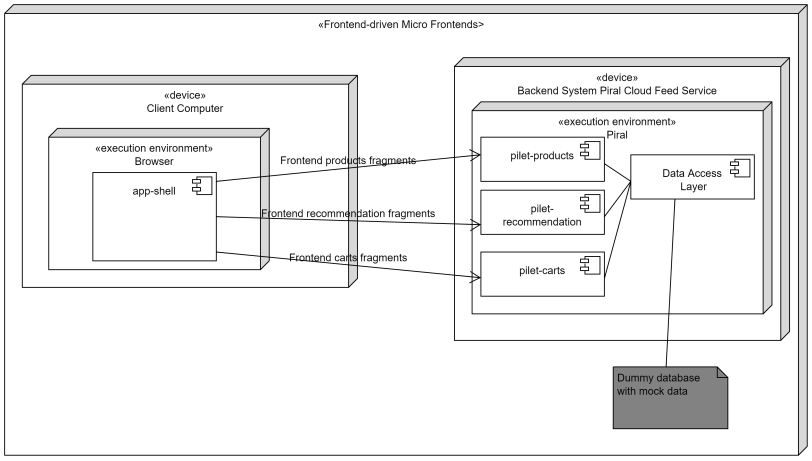


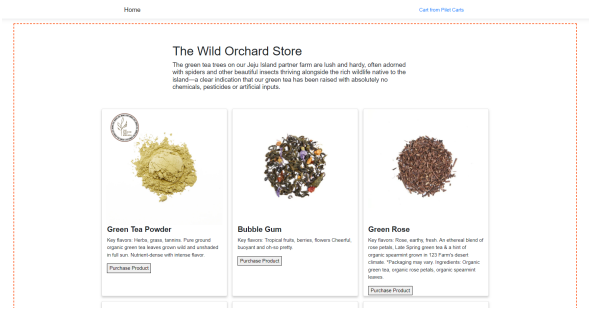
Figure 8.4. Wild Orchard Store composition. Source: own illustration

8.3.5 Results of the Sample Application Implementation

Following table and figures help readers visualizing the whole application with each Pilet's area and detailed images of the final results of the prototype.

Table 8.1. Pilets and their locations

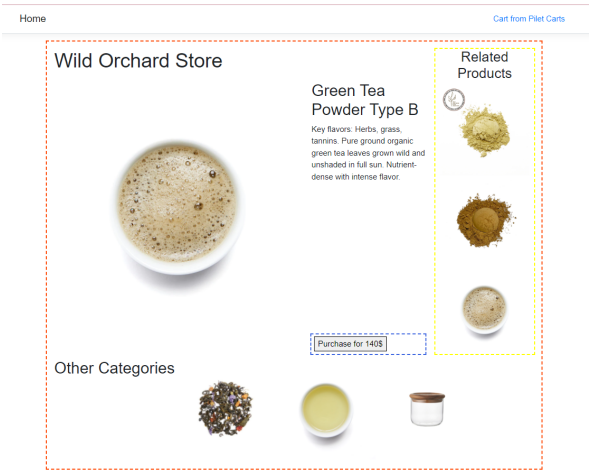
Pilet	Pilet's area
app-shell	container and navigation with menu items
pilet-products	marked with red border
pilet-recommendation	marked with yellow border
pilet-carts	marked with blue border



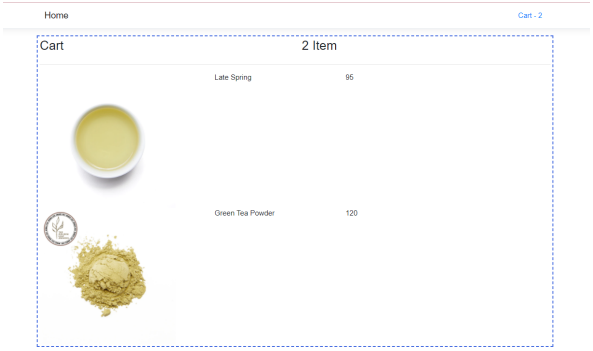
(a) List of products page

(b) Product details page

Figure 8.5. Pilet-products. Source: own illustration

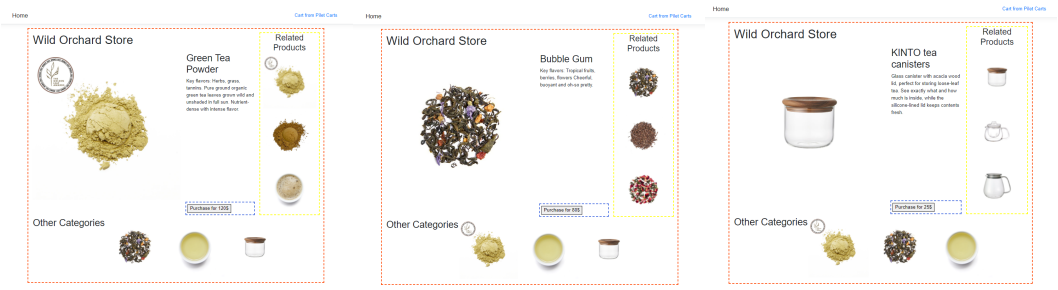


(a) pilet-recommendation product



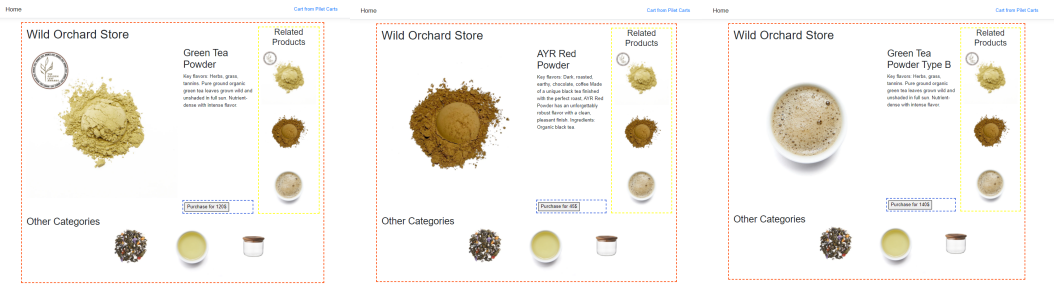
(b) pilet-carts cart page

Figure 8.6. Pilet-recommendation and pilet-carts. Source: own illustration



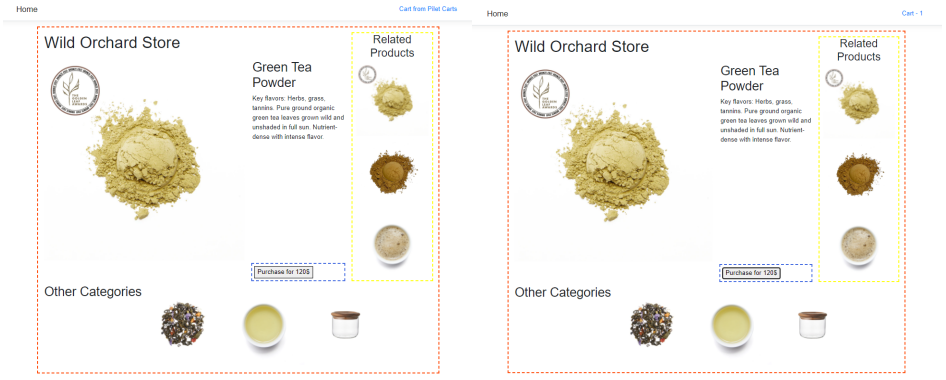
(a) Starting point **(b)** Navigate to new category **(c)** Navigate to another new category

Figure 8.7. Customer Journey of the first use case. Source: own illustration



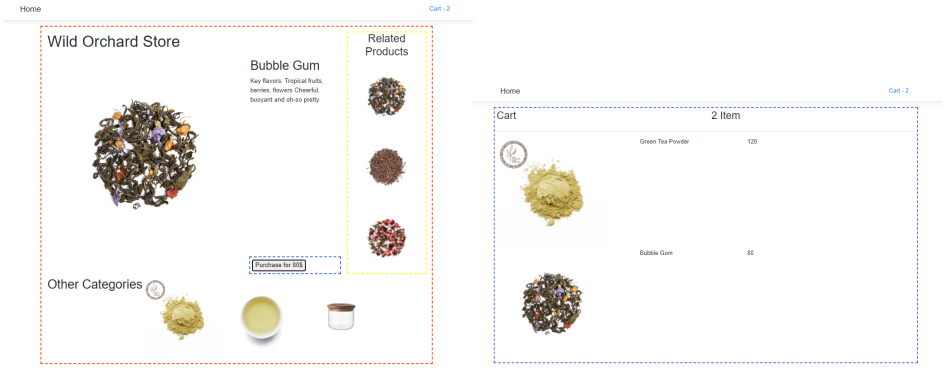
(a) Starting point **(b)** Navigate to new item in related products section **(c)** Navigate to another new item in related products section

Figure 8.8. Customer Journey of the second use case (related products section comes from pilet-recommendation, other categories section comes from pilet-products). Source: own illustration



(a) Starting point **(b)** Purchase a product

Figure 8.9. Customer Journey of the third use case. Source: own illustration



(a) Purchase another product **(b)** Navigate to cart page

Figure 8.10. Customer Journey of the third use case (cont)

8.4 Summary

This chapter presents the results of the study: a Piral sample application with a complete concept, customer journey, and user interface, preparing readers for a thorough understanding of Micro Frontends implementation in detail in the subsequent chapter.

Chapter 9

Micro Frontends Technical Implementation

This chapter examines Micro Frontends implementations and the results of the study in greater detail with code snippets and practical examples.

9.1 Micro Frontends Integration

9.1.1 Link and Iframe

A simple and intuitive strategy of implementing Micro Frontends is page-to-page transition via link and composition of fragments using iframe [26].

As for link transition, firstly, the single coupling between teams is the knowledge of other teams' URL patterns. For instance, a team need only know the exact URL path to a Micro Frontend and hard code it to its web page to make the integration work via a simple click from the client. Secondly, link transition meets the expectation of high robustness and low coupling. For instance, if a Micro Frontend goes down, other parts in the whole application still work. In other words, the web application shares nothing. A Micro Frontend has everything it needs to do its task regardless of technical frameworks, programming languages, and deployment strategy other teams use or any error they cause.

However, developers cannot embed a Micro Frontend into another, and more sophisticated integration techniques cannot be achieved using links.

As for iframe transition, it is possible to place one Micro Frontend inside another yet still preserve loose coupling and robustness properties. Firstly, iframes provide strong technical isolation. For instance, styles in one iframe cannot leak into other components, and script execution is regulated as they include crucial security features to encapsulate the different Micro Frontends.

However, iframe integration is a document-in-document or websites-in-websites approach, while direct assembling of HTML fragments is desired in most cases, which requires a Micro Frontend to provide or expose snippets of HTML or valid web components later translated into DOM elements within the final DOM context. Iframe also has some major disadvantages. Firstly, the layout of an iframe's content is fixed and determined in advance. For example, the current host document needs to know the exact height of the including iframe's content coming from a remote Micro Frontend. Moreover, responsive design is hard to achieve. In this case, the child iframe's content has no way to responsively adjust height or width as its dimension is defined by the parent document. Teams end up with a more complicated technical agreement of not only the URLs but also CSS properties. Secondly, iframe is resource-intensive and leads to poor performance for the browser. The browser generates a new context with computing resources such as RAM and CPU for every iframe. Thirdly, iframe is bad for accessibility and search engines.

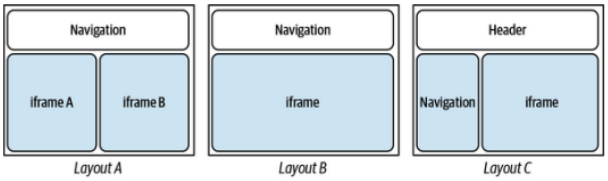


Figure 9.1. Micro Frontends with iframe. Source: [5, ch. 4]

9.1.2 Integration with Piral

Activating pages, client-side routing

One of the core features of Piral is routing. The app-shell assumes the task of determining what pages are shown based on the URL path. The following figure illustrates the standard pattern of routing engines. The `app`, or `PiletApi` object, is globally installed and used to register and unregister pages of all Micro Frontends. Consequently, URL changes are handled by routing engines. This enables orchestrating multiple pages from one or multiple Micro Frontends in the application.

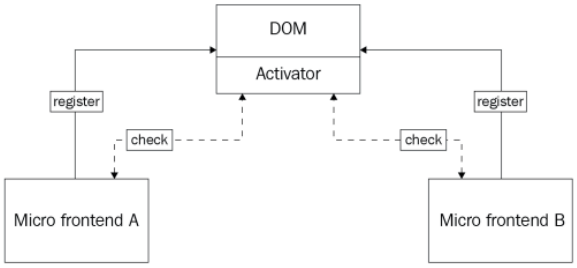


Figure 9.2. A common activator checks all registered Micro Frontends for their status [2, p. 159]

In Piral, each module carries out the page registration step itself via `app.registerPage` and `app.unregisterPage` functions. This takes place within the exposed `setup` function of each Pilets, often in the root config file `index.tsx`.

Let's look at an example of the config file `index.tsx` in `pilet-products (pilet-products\src\index.tsx)`.

```
// ...
import * as React from "react";
import { PiletApi } from "app-shell";
import { ProductsPage } from "./components/ProductsPage";
// ..

const ProductDetailsPage = React.lazy(
  // ...
);

export function setup(app: PiletApi) {
  // page resgistrations
  app.registerPage("/landing", ({ history }) => (
    <ProductsPage history={history} />
  ));

// ...
}
```

Listing 9.1. pilet-products index.tsx

Here the URL path `${host}/landing` is associated with `ProductsPage` React component. When a webpage with URL path `${host}/landing` is requested, the products page will be shown.

The PiletApi keeps track, orchestrates, and decides which Micro Frontend is currently loaded, should be loaded next, and need to be mounted or unmounted.

Extending, sharing components, or Piral extension

One of the selling points of Piral is Piral extension, that is, using components provided by the Pilets.

In the sample application, the product details page from `pilet-products` has and uses the buy button from `pilet-carts`.

The code looks like this, in `pilet-carts\src\index.tsx`:

```
  // ...
  import * as React from "react";
  import { PiletApi } from "app-shell";
  import { BuyButton } from "./components/BuyButton";
  import { CartPage } from "./components/CartPage";

  interface BuyButtonExtension { // ...  }

  export function setup(app: PiletApi) {
    // page resgistrations
    app.registerPage("/cart", () => ( // ... ));

    app.setData("cart-data", []);

    const addToCart = (item) => { // ...  }

    app.registerExtension<BuyButtonExtension>("buy-button", ({ params }) => (
      <BuyButton addToCart={addToCart} product={params.product} />
    ));
    // ...
  }
```

Listing 9.2. pilet-carts extension registration

The code look like this in `pilet-products\src\index.tsx`:

```
  // ...

const ProductDetailsPage = React.lazy(
  () => import("./components/ProductDetailsPage")
);

export function setup(app: PiletApi) {
  // page resgistrations
  app.registerPage("/landing", ({ history }) => ( // ... ));

  app.registerPage("/products/:id?", ({ history, match, piral }) => (
    <ProductDetailsPage
      id={match.params.id || "1"}
      history={history}
      BuyButton={({ product }) => (
        <piral.Extension
          name="buy-button"
          params={{ product }}
          empty={() => (
            <div className="blue-buy" id="buy">
              {" "}
              Buy button from (Micro Frontend) Pilet Carts{" "}
            </div>
          )}
        />
      )}
```

```
      Recommendations={({ category }) => ( // ... )}
    />
  ));
  // ...
  }
```

Listing 9.3. pilet-products uses pilet-carts extension

Here the `pilet-carts` register an extension name `buy-button` which hosts the `Recommendation` React component. In `pilet-products`, the `ProductDetailsPage` use the extension via `piral.Extension` API. In this case, the `piral.Extension` with name `buy-button` is passed down to `ProductDetailsPage`, all in the root config `index.tsx` file. This approach decouples the actual React component from Piral and improves readability and testability of the code; `empty` property specifies the fallback element if the main application cannot resolve this extension due to error, failure or out of service status.

Moreover, during the registration step of a shared component, the name of the Micro Frontend producer (`pilet-carts`) need not be mentioned. Alternatively, developers can give the extension an appropriate name (`buy-button`). As a result, the application and developers derive the name of an extension by its actual name instead of directly referencing the name of the responsible service.

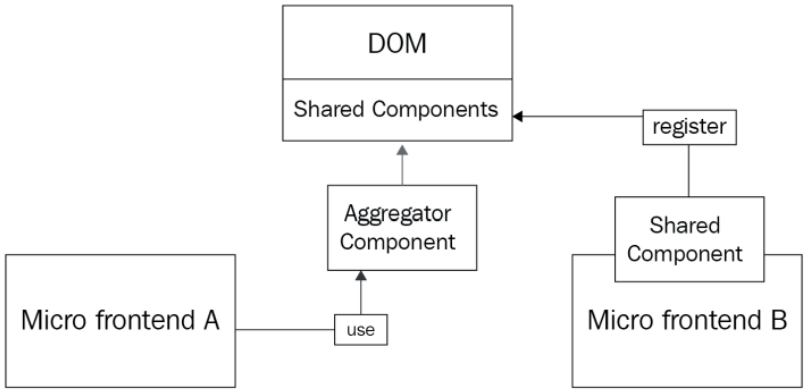


Figure 9.3. The general flow for components extension [2, p. 171]

9.2 Communication Patterns

In many real-world applications, interaction is the norm. Hence, communication patterns are essential and required in most cases.

9.2.1 Iframe Communication

In the case of iframe, according to Florian Rappl, communication can take place between reusable Frontend pieces by using the `window.postMessage` function since `HTML5`. The first characteristic of `window.postMessage` lies in the way that it only allows strings to be passed. It is obvious that true objects or JavaScript functions cannot be delivered even though strings can be translated into quite complex `JSON-serialized` objects. [2, pp. 13–14]

Sending a message

For example, the following code represents the code for Micro Frontend products; the iframe sends a message data to the target origin `http://localhost:1234/pilet-products`

```
<iframe src="http://localhost:1234/app-shell" name="app-shell">

<script>
  let appShellFrame = window.frames.app-shell;
  appShellFrame.postMessage("message", "http://localhost:1234/pilet-products");
</script>
}
```

Listing 9.4. Iframe script of Micro Frontend products

Receiving a message

A handler is responsible for listening to a new event when `postMessage` is called. The `event` object has `event.data` property to access the transferred data, and `origin` property represents the sender, for instance, `http://localhost:1234/app-shell`

```
  <script>
    window.addEventListener('message', function(event) {
      alert(`Received ${event.data} from ${event.origin}`);
    });
  </script>
}
```

Listing 9.5. An iframe script can be properly set up for receiving the message in other iframes

9.2.2 Piral Communication

Top-down Communication

The first use case (8.7) is governed by `pilet-products`. On the current product details page, users can navigate and click around different categories of chosen tea, that is, matcha, green tea leaves, or spring tea series. Consequently, relevant sections, that is, related products and buy button with price sections, are updated.

For instance, by clicking on the spring tea category, a new spring tea product page is loaded, and the product recommendation section from `pilet-recommendation` as well as the buy button with price from `pilet-carts` are updated accordingly. In this case, `pilet-products` initializes a request, passes data, communicates new changes to and notify `pilet-recommendation` and `pilet-carts`, the two Pilets consumes these events.

The code look like this in `pilet-products\src\components\ProductDetailsPage.tsx`:

```
import * as React from "react";
import { History } from "history";
import { products } from "../data/products";
import { productsUniqueCategories } from "../data/productsUniqueCategories";

export interface ProductDetailsPageProps {
  id: string;
  history: History;
  BuyButton: React.ComponentType<{
    product: any;
  }>;
  Recommendations: React.ComponentType<{
    category: string;
  }>;
}

const ProductDetailsPage: React.FC<ProductDetailsPageProps> = ({
  id,
  history,
  BuyButton,
  Recommendations,
}) => {
  const [currentDisplayedProduct] = products.filter(
```

```
    (product) => id === product.id
  );

  return (
    currentDisplayedProduct && (
      <>
        <div id="webhop-main">
          <h1 id="store">Wild Orchard Store</h1>
          <h2 id="name"> </h2>
          <p id="description">
            // ...
          </p>
          <div id="imageMain">
            // ...
          </div>
          <BuyButton product={currentDisplayedProduct} />
          <div id="options">
            // ...
          </div>
          <Recommendations category={currentDisplayedProduct.category} />
        </div>
      </>
    )
  );
};
```

Listing 9.6. ProductDetailsPage passes product data directly to extended components

In short, `pilet-products` communicate and deliver requests to `pilet-recommendation` and `pilet-carts`, that is a top-down communication.

Top-up Communication

For the second use case (8.8), at first glance, this situation seems to be identical to the first one. The main difference is that browsing, clicking, and navigating inside the related products section involves another Micro Frontend, that is `pilet-recommendation`.

For instance, when a user clicks on the `AYR Red Powder` image, the current page is updated with information about the new product. In particular, there is no smooth and app-like experience like that of single page application rerouting like `react-router-dom`; instead, the browser refreshes the page with an updated URL path. The reason for this is simple: `pilet-recommendation` is a separate component, cannot properly refresh the page with new content, and can only notify the `pilet-products` to refresh the page for new contents. In fact, `pilet-recommendation` emits an event with the desired product ID, `pilet-products` listens to this event, receives the product ID from `pilet-recommendation`, and carries out the actual navigation on its own. In other words, `pilet-recommendation` does not do the navigation itself; instead, it delegates the task to `pilet-products`.

The code look like this in `pilet-products\src\index.tsx`:

```
// ...
// Event from Recommendation pilet
// Pilet Products carries out the actual navigation
app.on("recommendation-click-event", (data) => {
  const navigateToNewPage = () => {
    history.push(`/products/${data.id}`);
    window.location.reload();
  };
  return navigateToNewPage();
});
// ...
}
```

Listing 9.7. pilet-products listens to custom event and carries out the navigation

The code look like this in `pilet-recommendation\src\index.tsx`:

```
import * as React from "react";
import { PiletApi } from "app-shell";
import { Recommendations } from "./components/Recommendations";

interface RecommendationExtension {
  category: string;
}

export function setup(app: PiletApi) {
  const emitEventData = (id) => {
    app.emit("recommendation-click-event", {
      id,
    });
  };

  app.registerExtension<RecommendationExtension>(
    "recommendations",
    ({ params }) => (
      <Recommendations category={params.category} navigate={emitEventData} />
    )
  );
}
```

Listing 9.8. pilet-recommendation emits custom event and transmits the product Id

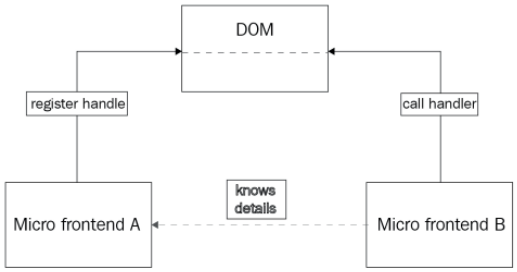


Figure 9.4. The standard DOM event interface can be used to exchange events [2, p. 169]

This case is a top-up communication via events between Micro Frontends, from `pilet-recommendation` upward to `pilet-products`.

Sharing data between Pilets

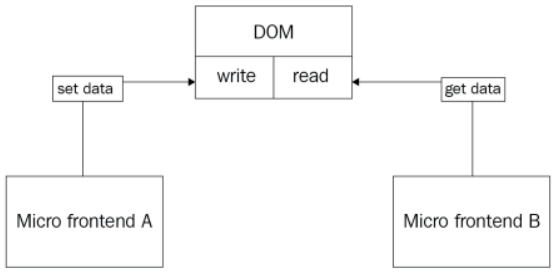


Figure 9.5. A central API provides read and write access to shared data [2, p. 170]

The last use case is the cart function (8.10). Users click on the buy button, the cart navigator on the top-right corner is updated accordingly, and cart data can be seen on the cart page.

The use case essentially involves `pilet-carts`, `app-shell`, and requires `pilet-products`. When

users click the buy button, the product data are saved (in memory) and later can be retrieved on the cart page. In this case, `pilet-carts` uses, updates, and keeps track of the state of cart data. The state of cart data is universally shared and accessed by any Pilet and the app-shell with `setData` and `getData` methods of Pilet API. It then updates the Cart menu item in the app-shell every time new data arrives via the Piral global state management mechanism. Eagle eye readers may notice that, especially in the second use case, when a page reload takes place, the cart data is reset. This is inevitable due to in-memory storage and a lack of proper and persistent state management. This is acceptable for the purpose of the study.

The code look like this in `pilet-carts\src\index.tsx`:

```
import * as React from "react";
import { Link } from "react-router-dom";
import { PiletApi } from "app-shell";
import { BuyButton } from "./components/BuyButton";
import { CartPage } from "./components/CartPage";

interface BuyButtonExtension {
  product: any;
}

export function setup(app: PiletApi) {
  // page resgistrations

  app.registerPage("/cart", () => ( // ... ));

  app.setData("cart-data", []);

  const addToCart = (item) => {
    var cart = app.getData("cart-data");
    cart.push(item);
    app.setData("cart-data", cart);
    console.log("Cart: ", cart);
  };

  app.registerExtension<BuyButtonExtension>("buy-button", ({ params }) => ( // ...));

  // Cart event
  app.registerMenu("cart-menu", () => { // ...});

  app.on("store-data", ({ name }) => {
    if (name === "cart-data") {
      app.registerMenu("cart-menu", () => (
        <Link to="/cart">Cart - {app.getData("cart-data").length} </Link>
      ));
    }
  });
}
```

Listing 9.9. pilet-carts handles cart data

9.3 Lazy Loading

Instead of downloading all information on initial loading to a web application, a component is loaded at its first use case. Then, lazy loading reduces initial load time and page weight leading to quicker page load time and conserving bandwidth by avoiding requesting unused data and resources.

The code look like this in `pilet-products\src\index.tsx`:

```
  // ...
const ProductDetailsPage = React.lazy(
  () => import("./components/ProductDetailsPage")
);

export function setup(app: PiletApi) {
```

```
// page resgistrations
app.registerPage("/landing", ({ history }) => (
  <ProductsPage history={history} />
));

app.registerPage("/products/:id?", ({ history, match, piral }) => (
  <ProductDetailsPage
    id={match.params.id || "1"}
    history={history}
    BuyButton={({ product }) => ( // ... )}
    Recommendations={({ category }) => ( // ... )}
  />
));
// ...
}
```

Listing 9.10. Lazy loading of a React component in pilet-products index.tsx file

On initial load, a user is navigated to the list of all products page. Then, only when moving to the product details page does the browser download the code for `ProductDetailsPage` React component. Lazy loading is implemented with `React.lazy()` [27] in this case.

9.4 Dependencies Management

Sharing dependencies is an appropriate setting in Micro Frontends. In practice, developers can take either way of sharing dependencies: sharing no dependencies as is the case in Microservices or sharing all inter-team dependencies. To get the best of both worlds, however, developers have to decide which dependencies to share.

Firstly, in the beginning, it is not necessary for every Micro Frontend to share all dependencies; instead, they should always share nothing. It is reasonable that so doing will make it easy for developers to familiarize themselves with an unfamiliar coding environment and avoid configuration constraints, potential bugs, and inessential complexities. However, including every required dependency in each Micro Frontend in this manner makes the whole application slow and bloated, finally worsening its performance. Then, what really matters is how developers should decide which dependencies to share. In the Micro Frontends world, this decision is justified on the basis of performance-based assessment and evaluation. According to Florian Rappl [2], some criteria are:

- *They should be of a significant size (from at least 15 KB to 30 KB; even more appropriate is 100 KB)*
- *They should be used by at least two micro frontends.*
- *They should not have multiple commonly used versions with breaking changes.*
- *They should play an important role in rendering components*
- *They should be technical, not domain specific*

For clarity, the amount of for instance, 15KB or 100KB is the size of the shared dependencies, not the final download size for clients. A dependency that is not to be shared is bundled in the respective Micro Frontend, increasing the size of a Micro Frontend and potentially producing duplicates. In particular, some dependencies may rely on global values in the code and might only be used as a singleton, consequently demanding these dependencies to be shared.

One worth mentioning point is that increasing demand for domain-specific dependencies often indicates that domain decomposition and architectural boundaries have developed faults or undergone some incorrect steps beforehand, for instance, during the requirement engineering process. In this case, developers should fall back to previous software engineering stages and improve them before eventually adding more shared dependencies. Once a shared dependency is included, it is coupled to the system and generally impossible to reverse.

Secondly, how should this functionality be realized?

Sharing dependencies in Webpack Module Federation

Webpack Module Federation offers Micro Frontends a way to exchange dependencies freely:

```
// ...
const ModuleFederationPlugin = require("webpack/lib/container/ModuleFederationPlugin")
    ;
// ...

const devConfig = {
  // ...
  plugins: [
    new ModuleFederationPlugin({
      name: "app-shell",
      // ...
      // sharing dependencies
      shared: packageJson.dependencies,
    }),
  ],
};

// ...
```

A selling point of Module Federation is the ability to add additional constraints such as dependencies versions. This allows finer control at selecting only dependencies with match versions.

Sharing dependencies in Piral

Considering three Micro Frontends `pilet-products`, `pilet-recommendation` and `pilet-carts`. All of them use the same framework that is React. Moreover, they can use additional libraries such as React Router DOM. In this case, developers need a way to allow them use one single dependency of React.

In Piral, sharing dependencies is straightforward. The easiest way to share dependencies from the app shell is to declare them in the externals section of the `package.json`. For instance, if we want to share vue dependency, we can extend `pilet.external` in the app shell's `package.json`:

```
// ...
"pilets": {
    "files": [],
    "externals": [
      "vue"
    ],
    // ...
}
// ...
```

One worth mentioning point is that the following dependencies are shared and default dependencies used in Piral, hence always added:

- react
- react-dom
- react-router
- react-router-dom
- history
- tslib
- path-to-regexp

- @libre/atom
- @dbeining/react-atom

Any other dependency needs to be added to the externals list. Here is an example of sharing dependencies of pilet-products where they are listed in `peerDependencies` property.

> *"The peerDependencies represent the list of shared dependency libraries, i.e., dependencies treated as external, which are shared by the application shell. The peerModules repesent the list of shared dependency modules, i.e., modules treated as external, which are shared by the application shell."* (Source: `https://docs.piral.io/reference/documentation/metadata`)

```
// ...
"peerDependencies": {
    "@dbeining/react-atom": "*",
    "@libre/atom": "*",
    "app-shell": "*",
    "history": "*",
    "path-to-regexp": "*",
    "react": "*",
    "react-dom": "*",
    "react-router": "*",
    "react-router-dom": "*",
    "tslib": "*"
  },
// ...
```

A real-world sharing dependencies scenario from the sample application:

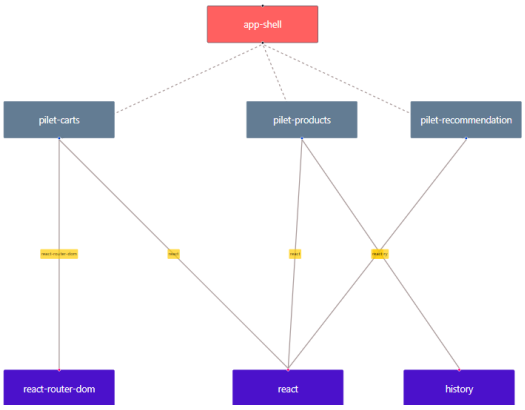


Figure 9.6. Dependencies in Piral with three Pilets use one single React package. Source: own illustration

The downside of this pattern is increased coordination and overall complexity. In the end, developers presumably get the best of both worlds: more flexibility and better performance.

9.5 Knowledge Sharing

In practice, it is the case that communicating changes such as API changes and event production will be of concern for scaling development.

Firstly, agreements, conventions, and contracts need to be formed among teams. For instance, if a Micro Frontend exposes an event with a certain name, this sharing information needs to be transmitted to all

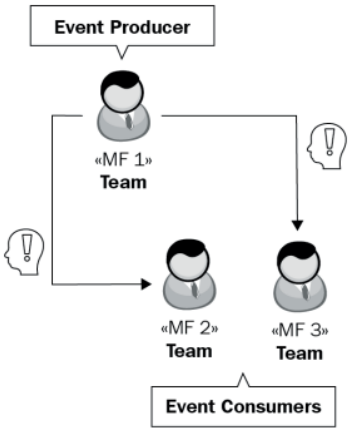


Figure 9.7. Change of event's name. Source: [2, p. 29]

consumers. In the case of event name changes, the responsible team must do the job of keeping consuming teams informed about the changes. Secondly, autonomy is good, and productivity is better where the communication factor plays an important role. Adopting shared load test scenarios or picking a common error-logging infrastructure without reinventing the wheels should always be taken as a long-lasting, productive method concerning high-quality, focused idea development, effective time management, and possible risk management. A transparent protocol for information exchange should be available in most cases.

Thirdly, ensuring visual consistency across Micro Frontends is vital for the final product. A style guide and design system, including user interfaces like form, navigation, menu item, input field, button, icon, or typography, can be made public, accessible for every party, and serve as a guideline and reference for developers.

9.6 Cross-framework Components

Piral is based on React and utilizes plugin architecture to allow components from other frameworks to be integrated. For example, in the context of one Pilet, to use `vue` components, developers need to install the Piral plugin for `vue`: `piral-vue`.

```
# Install Vue
npm i vue
# Add the Vue plugin to the Pilet by running:
npm i piral-vue
```

Listing 9.11. Setup to integrate Vue components in a Pilet

The following function for cross-frameworks integration is then added to the Pilet API: `fromVue()`

Use a vue component in a Pilet:

```
import { PiletApi } from "app-shell";
import { fromVue } from 'piral-vue/convert';
import VuePage from "./Page.vue";
import BuyButton from "./BuyButton.vue";

interface BuyButtonExtension {
  item: Object
}

export function setup(app: PiletApi) {

  const addToCart = (item) => {
    // ...
```

```
  }

  app.registerPage('/sample', app.fromVue(VuePage));

  app.registerExtension<BuyButtonExtension>(
    'buy-button',
    app.fromVue(BuyButton, { addToCart: addToCart}})
  )
}
```

Listing 9.12. Piral docs Vue plugin. Source: `https://docs.piral.io/plugins/piral-vue/overview`

Naturally, the framework's specific modules and tools such as `vue, vue-loader, vue-template-compiler` need to be installed via `npm`. The final dependencies can be like so:

```
// ...
"dependencies": {
    "@vue/component-compiler-utils": "^3.3.0",
    "piral-vue": "^0.14.24",
    "vue": "^2.0.2",
    "vue-loader": "^17.0.0",
    "vue-template-compiler": "^2.0.2"
  }
// ...
```

Listing 9.13. Vue realted dependencies in a Pilet

Incompatible framework's version of components, for instance, `vue@\^2.0.0` and `vue@\^3.0.0` plays no role in this case, as long as the setup is properly handled.

9.7 Reliability

As a consequence of utilizing distributed systems, applications need to handle the failure of services properly if one of them is unavailable due to some reason, introducing an additional layer of complexity yet making the software in one way or another more resilient to failure. Established approaches such as setting small timeouts, implementing retries for high-priority requests, and detecting a connection loss so as to handle it gracefully can be applied [2, p. 31]. Still, developers need to handle exceptions and errors properly. In this case, error boundary can be used. Prominent examples are React Error Boundaries [28] and Piral fallback mechanism (section 9.8).

In particular, practitioners may struggle to find a way of making the Piral instance (application shell) more resilient to handle transient failures when loading from the feed service. In this aspect, as Piral is unopinionated, no such handling has existed yet. One simple technique is assembling the response of feed service upfront into the HTML page within a dynamic web server upfront, which makes it resilient and better for performance.

9.8 Cancellation of a Micro Frontend Service

Cancellation of a Micro Frontend service can support testing and help ensure scaling development and loose coupling. In the context of Piral apps, let's try disabling the two Pilets `pilet-carts` and `pilet-recommendation`:

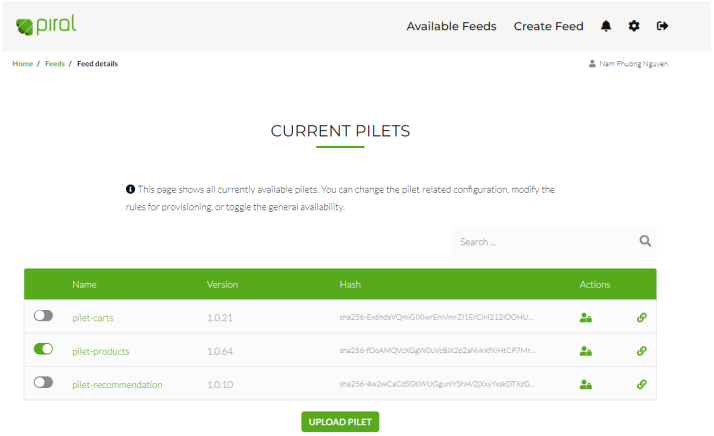


Figure 9.8. Deactivate the Pilets. Source: own illustration

Piral provides a fallback mechanism for an extension with property `empty` out of the box, in `pilet-products\src\index.tsx`, the code looks like this:

```
export function setup(app: PiletApi) {
// ...

app.registerPage("/products/:id?", ({ history, match, piral }) => (
  <ProductDetailsPage
    id={match.params.id || "1"}
    history={history}
    BuyButton={({ product }) => (
      <piral.Extension
        name="buy-button"
        params={{ product }}
        empty={() => (
          <div className="blue-buy" id="buy">
            {" "}
            Buy button from (Micro Frontend) Pilet Carts{" "}
          </div>
        )}
      />
    )}
    Recommendations={({ category }) => (
      <piral.Extension
        name="recommendations"
        params={{ category }}
        empty={() => (
          <div className="green-recos" id="reco">
            {" "}
            Recommendation products from (Micro Frontend) Pilet Recommendation{" "}
          </div>
        )}
      />
    )}
  />
));

// ...
}
```

Listing 9.14. pilet-products index.tsx

Failure in one Micro Frontend can be properly handled to avoid affecting other parts of the application. Moreover, in light of feature development optimization (section 6.1.2), feature extension and significant improvement can take place without the consequence of editing the same user interface simultaneously. The end-result:

Figure 9.9. pilet-carts and pilet-recommendation are unavailable. Source: own illustration

9.9 CSS Scoping

Micro Frontends with independent teams make it possible for CSS classes to have duplicate names, leading to some weird behavior on the overall layout. One simple yet resilient and widely adopted practice is prefixing the Micro Frontends name to CSS classes. For example, in styling a product card in `pilet-products`:

```
  .card {}
  .card__image {}
  .card__image--active {}
  .card__image--inactive {}

  // to
  .products_card {}
  .products_card__image {}
  .products_card__image--active {}
  .products_card__image--inactive {}
}
```

Listing 9.15. pilet-products CSS prefixing

9.10 Request Optimizations for Performance

Micro Frontends leading to modularization draw developers' attention to growing API requests as in place of a monolith, a much more fragmented system, both server- and client-side, comes into being. The sum total of HTTP requests increases as a consequence. In this case, if many modules query the same data, a best practice to mitigate this negative effect is micro-caching for API requests. [2, p. 25]

9.11 Performance of Multiple Frameworks Micro Frontends

According to Luca, using multiple frameworks like React, Angular, Vue, or Svelte in Micro Frontends leads to larger payload size, potential dependency clashes, and much overhead. For instance, assume having multiple versions of React and a version of Angular in the same view, the browser needs to download several versions of React and one version of Angular, resulting in a large download size for client users and potential conflicts of global variables among different libraries. However, there are cases where this approach makes sense, for example, in dealing with old or outdated framework versions and migrating a legacy application to a new one. There are several solutions for the multi-framework approach. [5, ch. 4]

> *"Iframes create a sandbox so that what loads inside one iframe doesn't clash with another iframe. Module Federation allows you to share libraries and provides a a mechanism for avoiding clashing*

> *dependencies. Import maps allow us to define scopes for every dependency so we can define different versions of the same libraries to different scopes. And web components can "hide" behind the shadow DOM the frameworks need for a micro-frontend."* - Luca Mezzalira

9.12 Mobile Performance of Micro Frontends with Piral

Considering the dependencies and payload size (section 9.4), practitioners of Micro Frontends may wonder how Micro Frontends would turn out on mobile/smartphones. In principle, nothing changes for web on mobile in terms of dependencies and payload size. In comparison to computers, Micro Frontends should not perform worse on mobile. As Micro Frontends structurally decompose an application, updates, upgrades, and changes are usually more fine-grained. In Piral, developers can either omit Micro Frontends that offer nothing but overhead or replace some Micro Frontends with smaller ones dedicated for mobile or any other platform. Therefore, for instance, with proper optimization, over time, Micro Frontends can deliver better performance for recurring users on average.

9.13 Summary

Starting from the simplest approach of realizing Micro Frontends with links and iframe, this chapter presents the results of the study: actual implementations as well as technical implications of Piral as a Micro Frontends solution, covering significant aspects like communication patterns, integration techniques, dependencies management, lazy loading, preliminary assessment of reliability, multiple framework usages and performance on mobile. Eventually, practical examples and implementations are explained and discussed in detail, greatly assisting readers in understanding and picturing how Micro Frontends would be developed in reality.

Chapter 10

Evaluation of Piral Sample Application

This chapter evaluates the sample application, results of the study, with respect to development, testing and debugging.

10.1 Developer Experience in Piral

10.1.1 Creation of Application Shell and Pilets

The whole development cycle is clarified with Piral development workflow (section 7.5). The following code snippets illustrate the typical setup and scaffolding process via the command line:

```
# Install the Piral CLI
npm i piral-cli -g

# Scaffold an application shell
piral new --target app-shell

# Create a npm package (app-shell-1.0.0.tgz) of the app shell referenced by the pilets
npx piral build

# Scaffold a new pilet with referenced path to the tarball of the app-shell
pilet new ./app-shell/dist/emulator/app-shell-1.0.0.tgz --target pilet-products
pilet new ./app-shell/dist/emulator/app-shell-1.0.0.tgz --target pilet-recommendation
pilet new ./app-shell/dist/emulator/app-shell-1.0.0.tgz --target pilet-carts

# Start the Piral instance in debug mode
npx piral debug
```

Listing 10.1. Complete scaffolding process in Piral

The scaffolding process is provided out of the box in Piral, and by default, the app-shell is started on `http://localhost:1234` with the content of the pilets.

10.1.2 Micro Frontends Development Cycle

For every Pilet, a developer can more or less make changes independently without causing unnecessary and unwanted effects on other parts of the system.

```
# the URL of Piral cloud feed service
const feedUrl =
  "https://feed.piral.cloud/api/v1/pilet/micro-frontends-thesis-nguyen-feed";

# debug a Pilet locally
pilet debug

# publish a Pilet to the feed service
npx pilet publish --fresh --url https://feed.piral.cloud/api/v1/pilet/micro-frontends-
    thesis-nguyen-feed --api-key <the-api-key>
```

Listing 10.2. Publishing Pilets

The API key is available and can be taken from the Piral Feed Service (section 7.4.3). In principle, every Pilet can be developed, built, and published independently only with some coordination with the app-shell.

10.2 Testing of Pilets

Unit testing was carried out using `piral-jest-utils` dependency, which provides React Typescript unit testing with Jest [29].

```
npm install --save-dev piral-jest-utils@^0.14.24 jest@^26.0.0 ts-jest@^26.0.0 @types
  /jest@^26.0.24
```

Listing 10.3. piral-jest-utils installation with recommended versions for the study

Due to incompatibility issues with the latest version of React 17 at the time of writing and the fact that both `Jest` and `piral-jest-utils` are a bit outdated, readers might need to install the dependencies in a forceful manner. The following code worked fine with the sample application and should have no major trouble in other settings.

```
npm install --save-dev piral-jest-utils@^0.14.24 jest@^26.0.0 ts-jest@^26.0.0 @types
  /jest@^26.0.24 --force
```

Listing 10.4. piral-jest-utils installation in force mode

For an example of `pilet-products` unit testing, the testing code look like this in `pilet-products\src\test\ProductDetailsPage.test.tsx`:

```
import * as React from "react";
import { History } from "history";
import ProductDetailsPage from "../components/ProductDetailsPage";
import { shallow, render } from "enzyme";

const TestDummyComponent = () => <div />;

describe("ProductDetailsPage", () => {
  it("renders", () => {
    render(
      <ProductDetailsPage
        id={"1"}
        history={{} as any as History}
        BuyButton={TestDummyComponent}
        Recommendations={TestDummyComponent}
      />
    );
  });

  it("calls history push", () => {
    const push = jest.fn();
    const wrapper = shallow(
      <ProductDetailsPage
        id={"1"}
        history={{ push } as any as History}
        BuyButton={TestDummyComponent}
        Recommendations={TestDummyComponent}
      />
    );

    wrapper.find("button").first().simulate("click");
    expect(push).toHaveBeenCalled();
  });
});
```

Listing 10.5. ProductDetailsPage.test.tsx

The above tests verify the correct rendering of the main component `ProductDetailsPage` and mocks button click for navigation within the pilet. After running the test with the command `npm test` as in the root directory `pilet-products\package.json` like so:

```
 // ...
"scripts": {
    "start": "pilet debug",
    "build": "pilet build",
    "upgrade": "pilet upgrade",
    "test": "jest --passWithNoTests"
  },
   // ...
```

Listing 10.6. pilet-products package.json file

The test results look like this:

```
> pilet-products@1.0.64 test
> jest --passWithNoTests

 PASS  src/test/ProductsPage.test.tsx
  ProductsPage
        renders (18 ms)

 PASS  src/test/ProductDetailsPage.test.tsx
  ProductDetailsPage
        renders (10 ms)
        calls history push (8 ms)

-------------------------------|---------|----------|---------|---------|-----------
File                           | % Stmts | % Branch | % Funcs | % Lines | Uncovered
    Line #s
-------------------------------|---------|----------|---------|---------|-----------
All files                      |      95 |       75 |    87.5 |   94.73 |
 components                    |   94.44 |       75 |    87.5 |   94.11 |
  ProductDetailsPage.tsx       |     100 |       75 |     100 |     100 | 58
  ProductsPage.tsx             |   85.71 |      100 |   66.66 |   83.33 | 38
 data                          |     100 |      100 |     100 |     100 |
  products.tsx                 |     100 |      100 |     100 |     100 |
  productsUniqueCategories.tsx |     100 |      100 |     100 |     100 |
-------------------------------|---------|----------|---------|---------|-----------
Test Suites: 2 passed, 2 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        5.786 s
Ran all test suites.
```

Listing 10.7. pilet-products unit testing results

Finally, we have a running sample application verified by proper unit testing.

10.3 Debugging with Piral Inspector

One main problem of Micro Frontends debugging is the absence of elements from other Micro Frontends. As discussed in section 5.2, providing a solution that can be debugged and extended well is a challenging task in vertically arranged software. Piral provides Piral Inspector (`https://docs.piral.io/tooling/piral-inspector`), a browser extension to enhance the debugging experience of Piral instances. With Piral Inspector, developers have a debugging tool for the Piral instance (e.g., `app-shell`) that works against the feed service and all Pilets (e.g., `pilet-products, pilet-recommendation, pilet-carts`).

The most useful features of Piral Inspector are the overview of loaded Pilets, adding/removing Pilets in real-time, visualization of Pilets origin, display of registered routes, events, and state container. With Piral Inspector extension added to the Chrome browser, some demos are given as follows. The visualization of the dependencies map of figure 9.6 also comes from Piral Inspector.

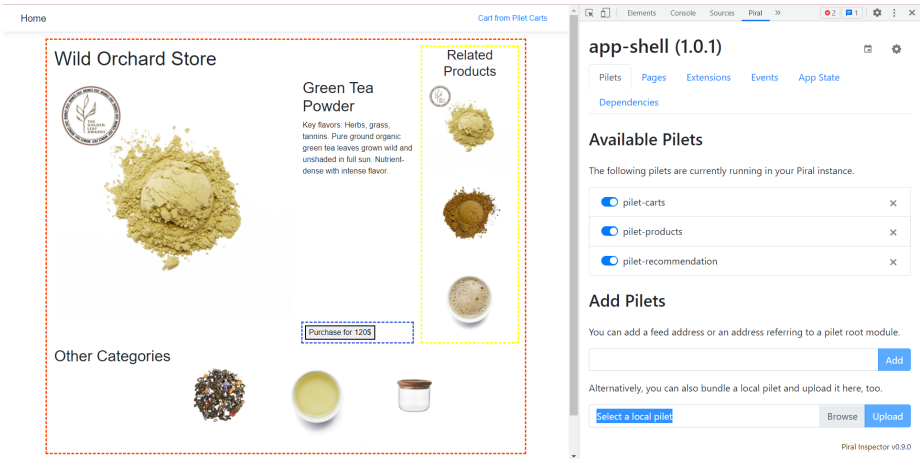


Figure 10.1. Piral Inspector at first glance. Source: own illustration

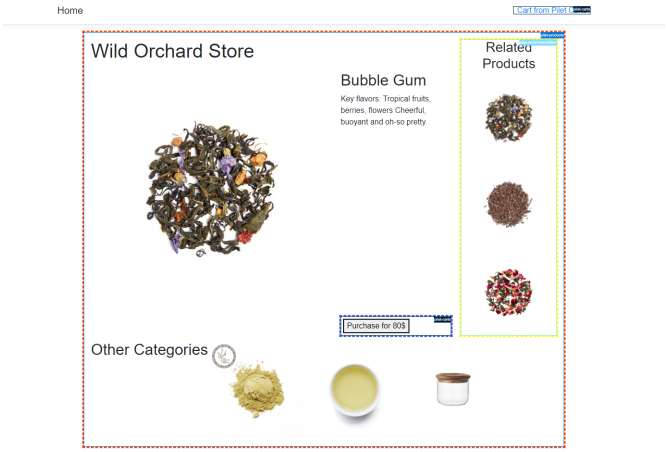


Figure 10.2. Visualization of components origin. Source: own illustration

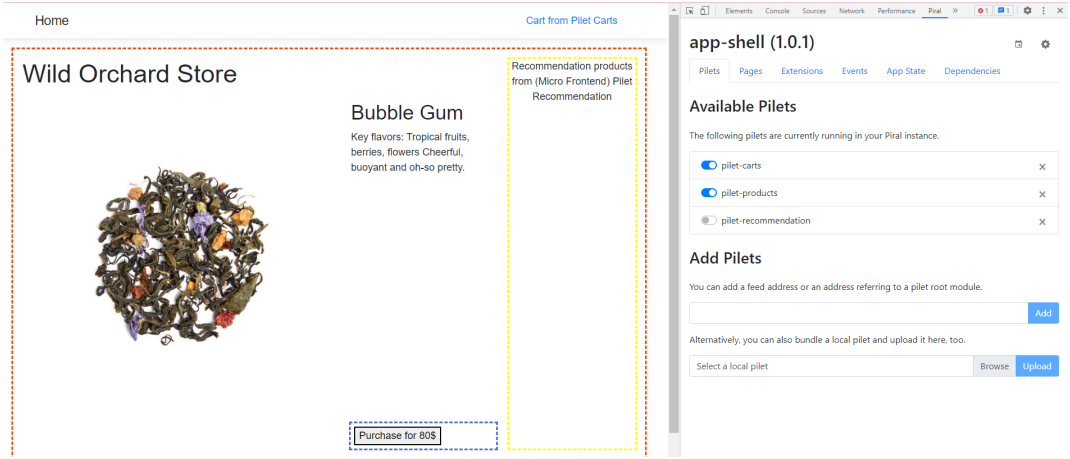


Figure 10.3. Real-time adding and removing of a Pilet. Source: own illustration

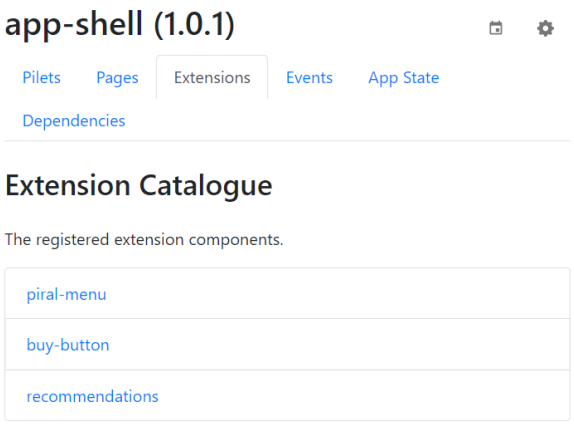
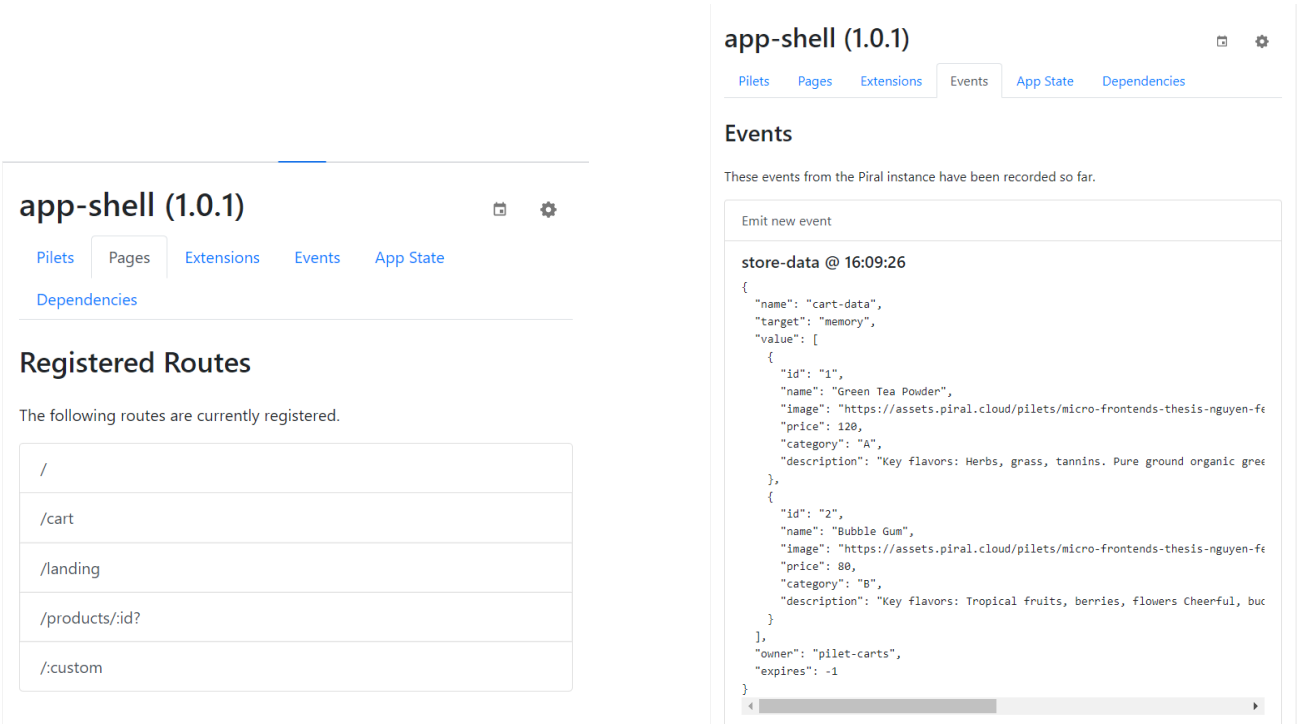


Figure 10.4. Recommendations extension of pilets-recommendation and buy-button extension of pilet-carts. Source: own illustration



(a) All registered routes

(b) Tracking of events and state

Figure 10.5. Routes and events. Source: own illustration

10.4 Summary and Discussion

The first section documents the development process from scaffolding to publishing a Micro Frontends and how it fits into the concept of Siteless UI with Piral. The second section offers some insights into the testing of Piral app. The last section highlights Piral Inspector, the debugging tool that greatly assists developers in developing Micro Frontends applications.

The selling points of Piral, also major challenges in Micro Frontends, are local development, common

run-time implementation, and debugging experience.

Firstly, with Piral, it is possible to replicate developer experiences like the one in a monolith. Developers can directly clone the repository, start a debugging session, and consequently write, build and publish their modules from the local environment.

Secondly, building a run-time for Micro Frontends solution orchestrating running in production is significant and often a painful process. Moreover, enabling a working system in local development, integrating a feed service, and ensuring the reliability of Micro Frontends in packaging are all non-trivial. Piral provides an established framework and the technical basis to start a running system swiftly.

Lastly, Piral provides Piral Inspector as the debugging tool, which tackles some common challenges raised by Micro Frontends practitioners.

Along with the previous chapters, this chapter clarifies some of the most common questions about Micro Frontends in the development and deployment process, consequently presenting Piral's implementation as a case study.

Chapter 11

Summary and Conclusion

11.1 Summary

As for the evaluation of Micro Frontends, the study presents central ideas, characteristics of software architecture, web application, Microservices and Micro Frontends, motivations leading to this specific architecture adoption, the results of client-side composition Miro Frontends with a prototype using Piral, covers from pros and cons, implementation principles, scalability to organizational aspects. On the other hand, Micro Frontends do come with significant costs. Potential performance issues, redundancy, consistency, communication overheads, cross-cutting concerns, integration patterns, download size, debugging difficulty, and overall complexity need to be thoroughly investigated, require careful planning and appropriate technical expertise.

As for the practical part with Piral, the approach of client-side composition, the focus of the study, provides a functional, dynamic and powerful application with explicit techniques and sheds light on the architecture's impact on development, deployment, and operation processes. This part covers implementation details, technical implications, and challenges that need to be tackled, and presents the sample application, which is flexible and straightforward to develop while ensuring a working system. As for implementation code in the prototype, in total, approximately 515 lines of React/TypeScript were written, including bootstrapping/importing code and excluding code for mock data, among which there are approximately 185 lines of code for the app-shell repository, 175 for the pilet-products repository, 50 for pilet-recommendation, and 105 for pilet-carts. As for testing code, in total, approximately 100 lines of React/TypeScript were written, including bootstrapping/importing code, among which there are approximately 40 lines of code for the pilet-products repository, 20 for pilet-recommendation, and 40 for pilet-carts.

11.2 Conclusion

Considering the current industry's landscape, many large-scale systems originally developed as monoliths have been rewritten, migrated or extended with micro architecture pattern of Microservices. This has become a long-established approach to solve downsides of monolithic solution such as strong coupling of components, complex deployment scenarios and large code base managed by a single development team. Over time, this pattern has been extended in the Frontend layer for large projects where team's structure and alignment efforts productively impact on efficiency and application scaling.

Micro Frontends, describing an alternative, experimental method of Frontend development, have reached a wider audience in recent years. Throughout the study, it becomes apparent that Micro Frontends make it possible to provide considerable benefits of Microservices on the Frontend side, especially scaling development processes, mean much for developers in dealing with major problems of monolithic systems and attract interest in Micro Frontends as an alternative organizational approach. Moreover, the architecture pattern can be applied for good reasons concerning business rules with respect to release planning, distributed deployments and independent teams. In the case of a consistent and complex pattern of change in development phase, dynamic systems as guiding principles of the product, teams with great autonomy which are deemed necessary, Micro

Frontends can affect the final outcome, help enhance the overall performance and meet strategic objectives of the project on the foundation of accurate measurements, established practices, and critical review. The study should be of service to practitioners and researchers, help them become accustomed to the idea of Micro Frontends, examine this design as advanced systems capable of giving added advantages over its counterpart monoliths, clarify the common architecture problems with solution guidelines, distinguish the pros and cons in different cases, and sorting primary areas of use for a certain type of application architecture.

In addition, as new tools continuously become readily available, no single technology has yet dominated the market of Micro Frontends. In the thesis, Piral, a Micro Frontends framework taken as the case study, has made itself distinguishing with respect to a number of advantages from dynamic modules integration into a single web application within the client browser's context, built-in Micro Frontends scaffolding process, sound design patterns and principles, and extensive documentation. A complete example, originally created using Piral from scratch in dedicated repositories with simple setup and required infrastructure as well as development tools and guiding principles, greatly assists readers in implementing client-side rendered Micro Frontends solution. We have seen how Piral realizes transmitting and delivering Micro Frontends pieces from different sources dynamically, discussion on its pros and cons, challenges that come with this pattern, developer experience, and how changes take place. As a framework, internal complexity and complicated setup are abstracted away, establishing a common foundation to bootstrap an application with little effort.

Piral makes the most sense for web portals, where many micro apps run independently from each other, deliver information from diverse sources, and can live on their own. It is best used to extend existing applications or migrate them to Micro Frontends over time, a realistic alternative to realize large-scale Micro Frontend solutions, and lightweight because of its design principles with simplicity and decoupling between the micro applications. Piral ensures loose module coupling in scaling development and requires reasonable costs for integration. The major drawback is that the application shell is quite complicated, and the fact that dependency of the modules on Piral API significantly increases.

The major achievement of the study is the author's efforts in implementing a well-designed sample application, especially ensuring a working system during the whole process. Appropriately discerning distinguishing characteristics and common problems between micro architecture on the Backend and Frontend also requires a relatively great amount of effort, not to mention difficulties in performing research on unfamiliar technologies such as Microservices, the initial training, and first-hand experience with foreign frameworks and programming languages. While good documentation and public demos are accessible, detailed tutorials of building Micro Frontends with Piral appear to have not existed; the thesis provides a comprehensive guide of the framework and a sound presentation of extensive theoretical background on micro architecture in web development. The results presented in this study should help readers, as well as developers, justify their decisions against or in favor of Piral, assist them in selecting a scalable application architecture leveraging Micro Frontends, and serve as a starting point for successfully implementing their first solution on the foundation of an established framework with technical principles, precise requirements, proper governance, and design awareness. In light of web development, the study promotes the adoption of Micro Frontends in general and Piral in particular for appropriate circumstances and, more importantly, establishes a sound foundation for the pattern by providing extensive guidelines to successfully bootstrap and launch Micro Frontends applications.

Bibliography

[1] M. Geers, "Microfrontends extending the microservice idea to frontend development." [Online]. Available: https://micro-frontends.org/

[2] F. Rappl, *The art of Micro Frontends: Build websites using compositional UIs that grow naturally as your application scales*. Packt Publishing, 2021.

[3] M. Geers, *Micro Frontends in Action*. Manning Publications, 2020.

[4] Smapiot, "Piral getting started." [Online]. Available: https://docs.piral.io/guidelines/tutorials/02-getting-started

[5] L. Mezzalira, *Building Micro-Frontends: Scaling Teams and Projects Empowering Developers*. O'Reilly Media, 2021.

[6] M. Fowler, "Software architecture guide," 2019. [Online]. Available: https://martinfowler.com/architecture/

[7] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. USA: Prentice Hall Press, 2017.

[8] J. Lewis, "Microservices," 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html

[9] S. Newman, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, Incorporated, 2019.

[10] M. Fowler, "Microservice trade-offs," 2015. [Online]. Available: https://martinfowler.com/articles/microservice-trade-offs.html#boundaries

[11] S. Newman, "Demystifying conway's law," 2014. [Online]. Available: https://www.thoughtworks.com/insights/articles/demystifying-conways-law

[12] B. Foote and J. Yoder, "Big ball of mud," 06 1999. [Online]. Available: http://www.laputan.org/mud/

[13] M. Fowler, "Integration database," 2004. [Online]. Available: https://martinfowler.com/bliki/IntegrationDatabase.html

[14] Atlassian, "Devops: Breaking the development-operations barrier." [Online]. Available: https://www.atlassian.com/devops

[15] Mozilla, "Introduction to the dom." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

[16] Bit, "Component-driven development platform," 2015. [Online]. Available: https://bit.dev/

[17] Webpack, "Module federation." [Online]. Available: https://webpack.js.org/concepts/module-federation/

[18] Podium, "Podium: Easy server side composition of microfrontends," 2019. [Online]. Available: https://podium-lib.io/

[19] M. Fowler, “Domain driven design,” 2014. [Online]. Available: https://martinfowler.com/bliki/DomainDrivenDesign.html

[20] Smapiot, “Piral,” 2014. [Online]. Available: https://piral.io/

[21] Apache, “Introduction to server side includes.” [Online]. Available: https://httpd.apache.org/docs/current/howto/ssi.html

[22] Oracle, “Edge side includes (esi) language tags.” [Online]. Available: https://docs.oracle.com/cd/B14099_19/caching.1012/b14046/esi.htm

[23] Zalando, “Project mosaic: Microservices for the frontend,” 2016. [Online]. Available: https://www.mosaic9.org/

[24] Canopy, “Single-spa: A javascript router for front-end microservices,” 2015. [Online]. Available: https://single-spa.js.org/

[25] Microsoft, “Visual studio code.” [Online]. Available: https://code.visualstudio.com/

[26] W. Standards, “iframe: The inline frame element.” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe

[27] Meta, “React: Code-splitting.” [Online]. Available: https://reactjs.org/docs/code-splitting.html

[28] ——, “Error boundaries.” [Online]. Available: https://reactjs.org/docs/error-boundaries.html

[29] F. O. Source, “Jest: A delightful javascript testing framework.” [Online]. Available: https://jestjs.io/

Appendix A

Source Codes Structure

Following illustrations present the structure of the sample application. Universally, main contents of a Pilet are `components` directory contains the UI elements, `test` directory contains the testing files, a `tarball` or `tgz` extension file is the production ready version of the Pilet for publishing to feed service, `dist` directory is the release bundle, `node_modules` is dependencies file managed by `npm`, `coverage` directory and `jest.config.js` file are used by `Jest` for testing, finally `data` and `assets` directories contain mock data for the sample application.

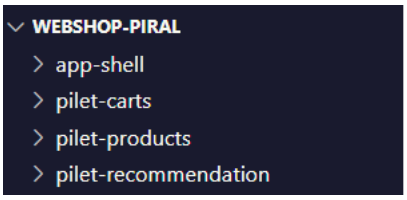
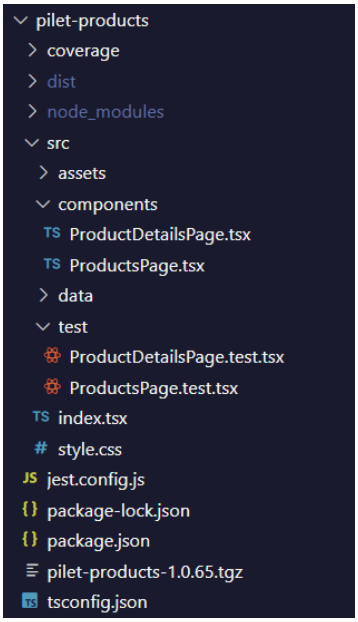
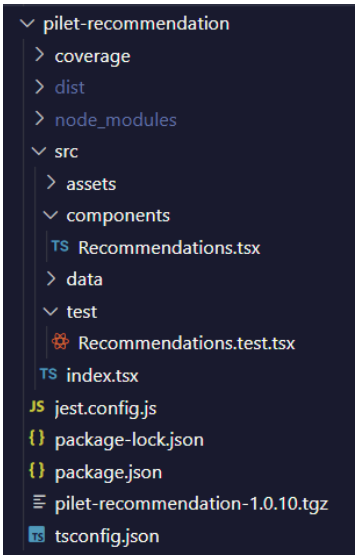


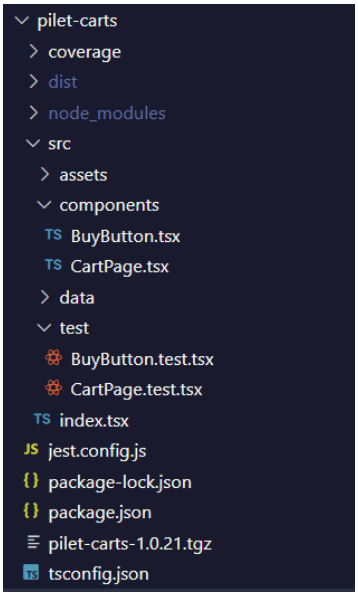
Figure A.1. The monorepository consists of four repositories. Source: own illustration



(a) pilet-products repository



(b) pilet-recommendation repository



(c) pilet-carts repository

Figure A.2. Repository structure of three Pilets. Source: own illustration

Appendix B

Testing Results

Following figures present the testing results of the sample application.

Figure B.1. pilet-products testing results. Source: own illustration

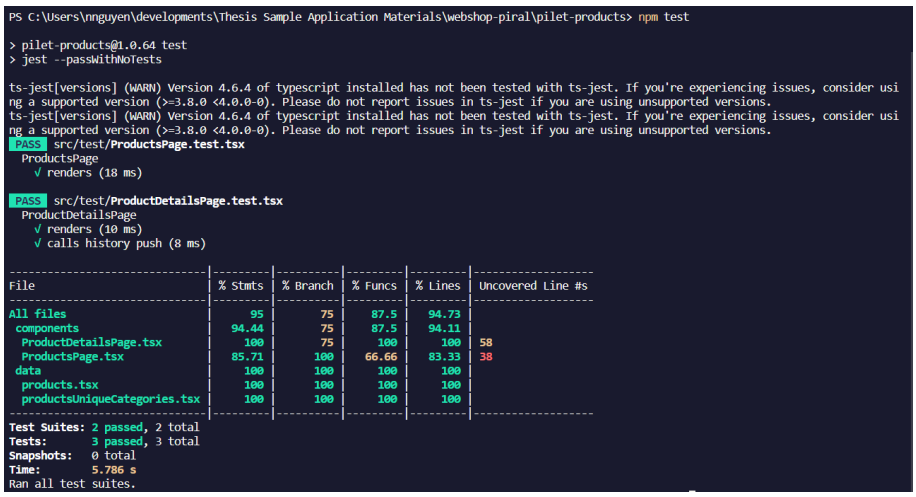


Figure B.2. pilet-recommendation testing results. Source: own illustration

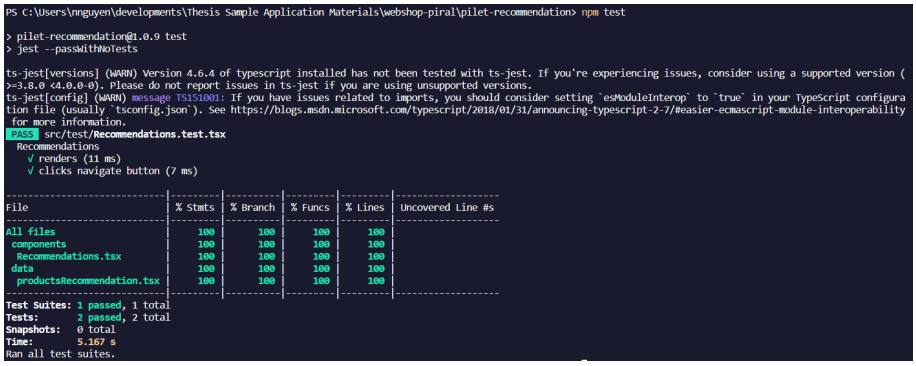
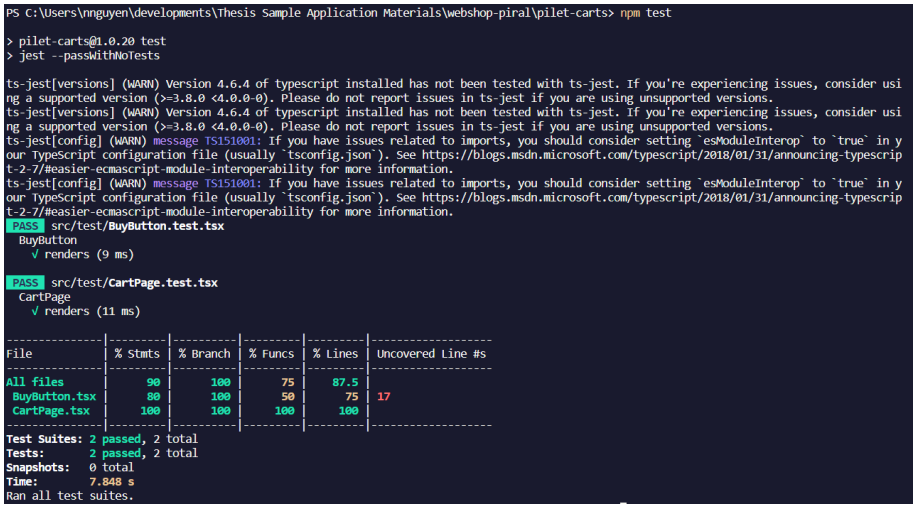


Figure B.3. pilet-carts testing results. Source: own illustration



```
PS C:\Users\nnguyen\developments\Thesis Sample Application Materials\webshop-piral\pilet-carts> npm test

> pilet-carts@1.0.20 test
> jest --passWithNoTests

ts-jest[versions] (WARN) Version 4.6.4 of typescript installed has not been tested with ts-jest. If you're experiencing issues, consider usi
ng a supported version (>=3.8.0 <4.0.0-0). Please do not report issues in ts-jest if you are using unsupported versions.
ts-jest[versions] (WARN) Version 4.6.4 of typescript installed has not been tested with ts-jest. If you're experiencing issues, consider usi
ng a supported version (>=3.8.0 <4.0.0-0). Please do not report issues in ts-jest if you are using unsupported versions.
ts-jest[config] (WARN) message TS151001: If you have issues related to imports, you should consider setting `esModuleInterop` to `true` in y
our TypeScript configuration file (usually `tsconfig.json`). See https://blogs.msdn.microsoft.com/typescript/2018/01/31/announcing-typescrip
t-2-7/#easier-ecmascript-module-interoperability for more information.
ts-jest[config] (WARN) message TS151001: If you have issues related to imports, you should consider setting `esModuleInterop` to `true` in y
our TypeScript configuration file (usually `tsconfig.json`). See https://blogs.msdn.microsoft.com/typescript/2018/01/31/announcing-typescrip
t-2-7/#easier-ecmascript-module-interoperability for more information.
 PASS  src/test/BuyButton.test.tsx
  BuyButton
    √ renders (9 ms)

 PASS  src/test/CartPage.test.tsx
  CartPage
    √ renders (11 ms)

---------------|---------|----------|---------|---------|-------------------
File           | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
---------------|---------|----------|---------|---------|-------------------
All files      |      90 |      100 |      75 |    87.5 |
 BuyButton.tsx |      80 |      100 |      50 |      75 | 17
 CartPage.tsx  |     100 |      100 |     100 |     100 |
---------------|---------|----------|---------|---------|-------------------
Test Suites: 2 passed, 2 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        7.848 s
Ran all test suites.
```