

O'REILLY®

Second  
Edition

# Building Micro-Frontends

Distributed Systems for the Frontend



Luca Mezzalira

# **Building Micro-Frontends**

SECOND EDITION

Distributed Systems for the Frontend

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

**Luca Mezzalira**

# **Building Micro-Frontends**

by Luca Mezzalira

Copyright © 2026 Luca Mezzalira. All rights reserved.

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com).

Acquisitions Editor: Louise Corrigan

Development Editor: Angela Rufino

Production Editor: Gregory Hyman

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

November 2025: Second Edition

## **Revision History for the Early Release**

- 2024-08-09: Third Release
- 2024-09-27: Fourth Release
- 2024-10-31: Fifth Release
- 2024-12-12: Sixth Release
- 2025-02-17: Seventh Release

- 2025-03-24: Eighth Release
- 2025-05-12: Ninth Release
- 2025-06-20: Tenth Release
- 2025-08-12: Eleventh Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098170783> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Building Micro-Frontends*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-17078-3

# Brief Table of Contents (Not Yet Final)

---

Chapter 1: Micro-Frontends Principles (available)

Chapter 2: Micro-Frontend Architectures and Challenges (available)

*Chapter 3: Discovering Micro-Frontends Architectures* (unavailable)

Chapter 4: Client-Side Rendering Micro-Frontends (available)

Chapter 5: Server-Side Rendering Micro-Frontends (available)

Chapter 6: Automation of Micro-Frontends (available)

*Chapter 7: Discover and Deploy Micro-Frontends* (unavailable)

Chapter 8: Automation Pipeline for Micro-Frontends: A Case Study (available)

Chapter 9: Backend Patterns For Micro-Frontends (available)

Chapter 10: Common Anti-Patterns in Micro-Frontend Implementations (available)

Chapter 11: Migrating to Micro-Frontends (available)

Chapter 12: From Monolith to Micro-Frontends: A Case Study (available)

Chapter 13: Introducing Micro-Frontends in Your Organization (available)

*Chapter 14: AI and Micro-Frontends: Augmenting Not Replacing* (unavailable)

# Chapter 1. Micro-Frontends Principles

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the first chapter of the final book. Please note that the GitHub repo will be made active later on.

If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the author at [\*building.microfrontends@gmail.com\*](mailto:building.microfrontends@gmail.com).

At the beginning of my career, I remember working on many software projects where small or medium-size teams were developing a monolithic application with all the functionalities of a platform available in a single artifact, the product produced during the development of a software, and deployed to a web server.

When we have a monolith, we write a lot of code that should harmoniously work together. In my experience, we tend to pre-optimize or even over-engineer our application logic more often than not. Abstracting reusable parts of our code can create a more complex codebase and sometimes the effort of maintaining a complex logic doesn’t pay off in the long run. Unfortunately, something that looked straightforward at the time could look very unwieldy a few months later.

In the past decades, public cloud providers like Amazon Web Services (AWS) or Google Cloud started to gain traction. Nowadays they are popular

for delegating what is increasingly becoming a commodity, freeing up organizations to focus on what really matters in a business: the services offered to the final users.

While cloud systems offer easier scalability compared to on-premise infrastructure, monolithic architectures require us to scale either horizontally adding more containers or virtual machines or vertically increasing the configuration of the machine where our application is running.

Furthermore, working on a monolith codebase with distributed teams and co-located ones could be challenging as well. Particularly after reaching medium or large team sizes because of the communication overhead and centralized decisions where a few people decide for everyone.

In the long run, companies with large monoliths usually slow down all the operations needed to release any new feature, losing the great momentum they had at the beginning of a project where everything was easier and smaller with few complications and risks. Also, with monolithic applications, we have to test and deploy the entire codebase every single time, which comes with a higher chance of breaking the APIs in production, introducing new bugs, and making more mistakes, especially when the codebase is not rock solid or extensively tested.

Solving these and many other challenges its staff faces, a company might move from complex monolith codebases to multiple smaller codebases and scoped domains called *microservices*.

Nowadays microservices architecture is a well-known, established and popular pattern used by many organizations across the world.

Microservices split a unique codebase into smaller parts, each of them with a subset of functionalities compared to a monolith. This business logic is embraced by developers because the problem solved by a microservice is simpler than looking at thousands of lines of code. Moreover a developer can maintain a clear picture of the code base and related functionality implemented, considering the cognitive load is by far less than working on a monolithic system.

Another significant advantage is that we can scale part of the application and use the right approach for a microservice instead of a one-size-fits-all approach similar to a monolith.

There are also some pitfalls to working with microservices. The investment on automation, observability, and monitoring has to be completed to have a distributed system under control. Another pitfall is the wrong definition of a microservice's boundary, for instance, having a microservice that is too small for completing an action inside a system relying on other microservices causing a strong coupling between services and forcing them to be deployed together every time. When this phenomenon is extended across multiple services we risk ending up with a big ball of mud or a system that is so complex that it is hard to extend.

Microservices bring many benefits to the table but could bring many cons as well. In particular, when we are embracing them in a project, the complexity of having a microservice architecture could become more painful than beneficial. Considering the options available in software architecture, we should pick microservices only when needed and not choose them recklessly just because it is the latest and greatest approach.

Micro-frontends have gained more traction in the frontend community and enterprise organizations thanks to the great fit they have when aligned to other distributed architectures like microservices. Keep in mind, however, that just like how microservices aren't a universal answer to all software decomposition, neither are micro-frontends. To understand where they fit in and what they are, let's look at some of the forces that are pushing us in this direction.

## Monolith to Distributed Systems

When we start a new project or even a new business offering a service online, the first iteration should be used to understand if our project or business could succeed or not.

Usually, we start by identifying a tech stack, a list of tech services used to build and run a single app, that is familiar to our team. By minimizing the bells and whistles around the system and concentrating on the bare minimum we're able to quickly gather information about our business idea directly from our users. This is also called a *minimum viable product (MVP)*.

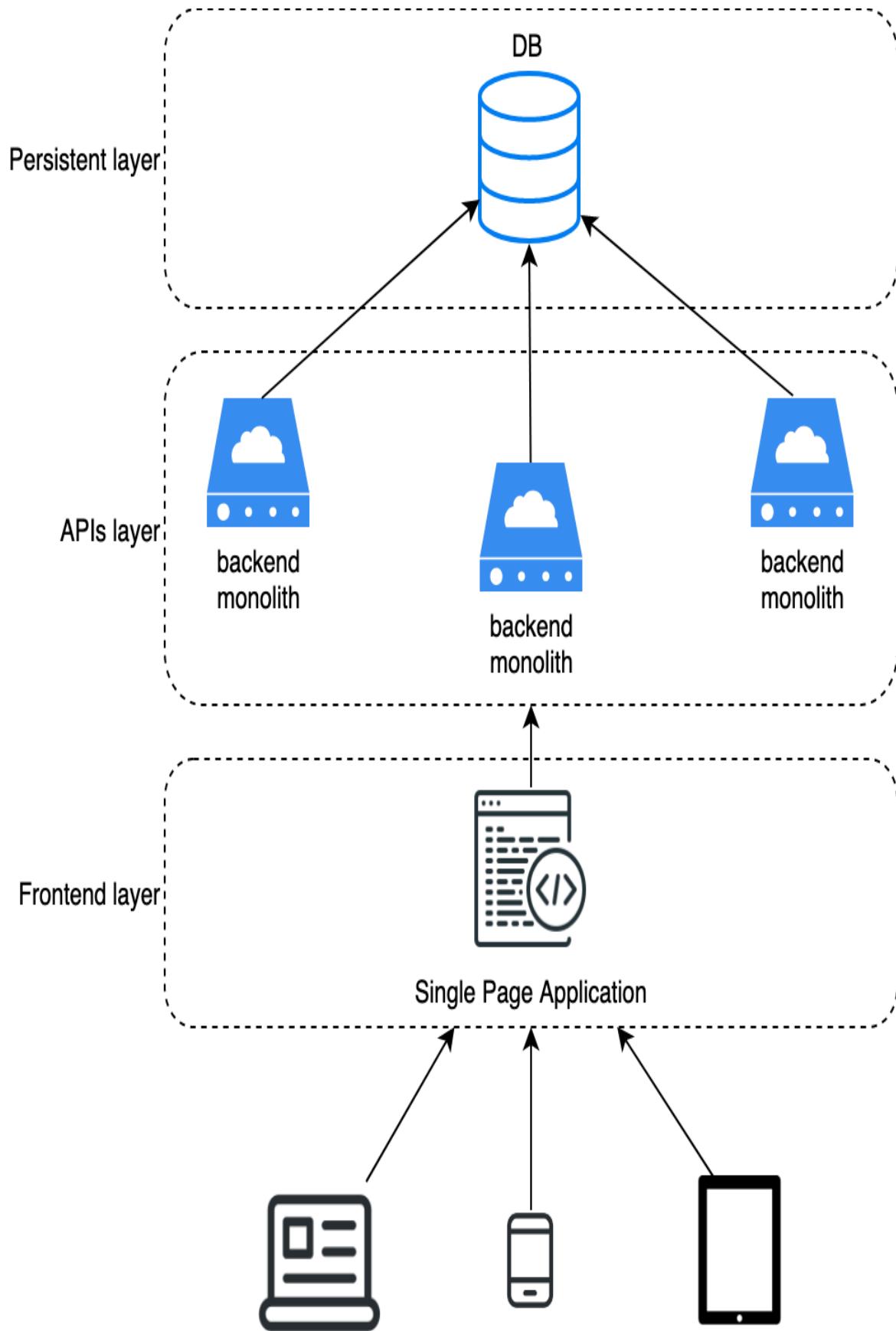
Often we design our API layer as a unique codebase (monolith) so we need to set up a single continuous integration or continuous delivery pipeline for the project. Integrating observability in a monolith application is quite easy; we just need to run an agent per virtual machine or container to retrieve the health status of our application servers. The deployment process is trivial, considering we need to handle one automation strategy for the entire APIs layer, one deployment and release strategy and when the traffic starts to increase we can scale our machine horizontally, having as many application servers as needed to fulfill the users' requests.

That's also why monolithic architecture are often a good choice for new projects considering we can focus more on the business logic of an application instead of investing too much effort on other aspects such as automation for instance.

Where are we going to store our data? We have to decide which database better suits our project needs—a graph, a NoSQL, or a SQL database? Another decision that must be made is whether we want to host our database on a cloud service or on-premises. We should select the database that will fit our business case better.

Finally, we need to choose a technology for representing our data, such as within a desktop or mobile browser, or even a mobile application. We can pick the best-known JavaScript framework available or our favorite programming language; we can decide to use server-side rendering or a Single Page Application architecture; then we define our code conventions, linting, and CSS rules.

At the end, we should end up with what you can see in [Figure 1-1](#):



*Figure 1-1. 3-tiers architecture composed by a presentation layer (frontend), an application layer (APIs layer) and a persistent layer (database)*

Hopefully, the business ideas and goals behind our project will be validated and more users will subscribe to our online service or buy the products we sell.

## Moving to Microservices

Now imagine that thanks to the success of our system, our business decides to scale up the tech team, hiring more engineers, QAs, scrum masters, and so on.

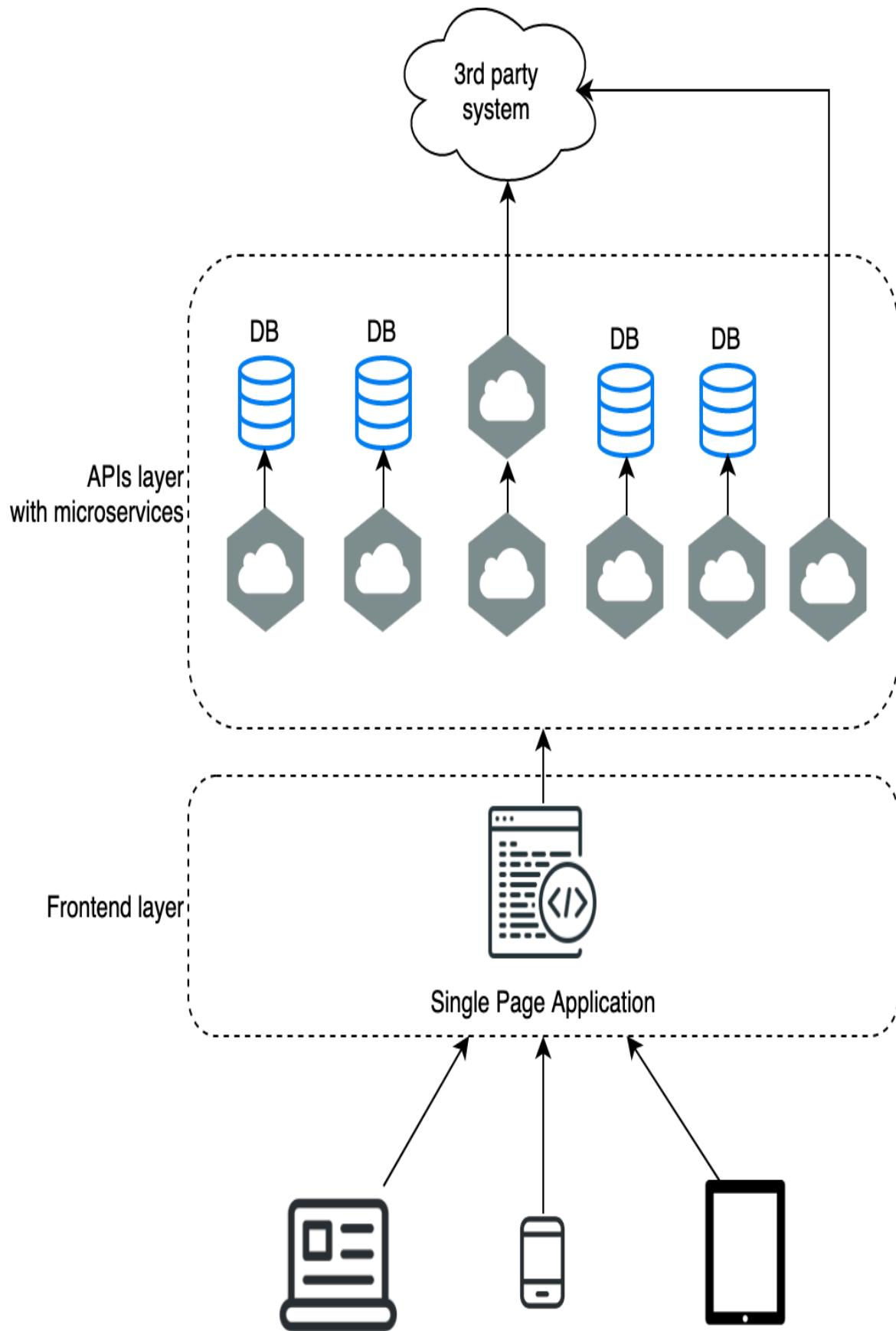
While monitoring our logs and dashboards, we realize not all our APIs are scaling organically. Some of them are highly cacheable, so the content delivery networks (CDNs) are serving the vast majority of the clients. Our origin servers are under pressure only when our APIs are not cacheable. Luckily enough, they're not all our APIs, just a small part of them.

Splitting our monolith starts to make more sense at this point, considering the internal growth and our better understanding of how the system works.

Embracing microservices also means reviewing our database strategy and, therefore, having multiple databases that are not shared across microservices; if needed, our data is partially replicated, so each microservice reduces the latency for returning the response.

Suddenly we are moving toward a decentralized ecosystem with many moving parts that are providing more agility and less risk than before.

Each team is responsible for its set of microservices. Team members can make decisions on the best database to choose, the best way to structure the schemas, how to cache some information for making the response even faster, and which programming language to pick for the job. Basically, we are moving to a world where each team is entitled to make decisions and be responsible for the services they are running in production, where a generic solution for the entire system is not needed besides the key decisions, like logging and monitoring, as we can see from [Figure 1-2](#).



*Figure 1-2. Microservices with Single Page Application*

However, we are still missing something here. We are able to scale our APIs layer and our persistent layers with well-defined patterns and best practices, but what happens when our business is growing and we need to scale our frontend teams, too?

## Introducing Micro-Frontends

So far on the frontend, we didn't have many options for scaling our applications, for several reasons. Up to a few years ago, there wasn't a strong need to do so because having a fat server, where all the business logic runs, and a thin client, for displaying the result of the computation made available by the servers, was the standard approach.

This has changed a lot in the past few years. Our users are looking for a better experience when they are navigating our web platforms, including more interactivity and better interactions.

Companies have arisen providing services with a subscription model, and many people are embracing those services. Now it's normal to watch videos on demand instead of on a linear channel, to listen to our favorite music inside an application instead of buying CDs, to order food from a mobile app instead of calling a restaurant.

This shift of behaviors requires us to improve our users' experience and provide a frictionless path to accomplish what a user wants without forgetting quality content or services.

In the past we would have approached those problems by dividing parts of our application in a shared components library, abstracting some business logic in other libraries so they could be reused across different parts of the application. In general, we would have tried to reuse as much code as possible.

I'm not advocating against solutions that are still valid and fit perfectly with many projects, but we might encounter quite a few challenges when embracing them.

For instance, when we have multiple development teams, all the rules applied to the codebase are often decided once, and we stick with them for months or even years because changing a single decision would require a lot of effort across the entire codebase and be a large investment for the organization without providing any value for the customers or the company.

Also, many decisions made during the development could result in trade-offs due to lack of time, ideal consistency, or simply laziness. We must consider that a business, like technology, evolves at a certain pace and it's unavoidable.

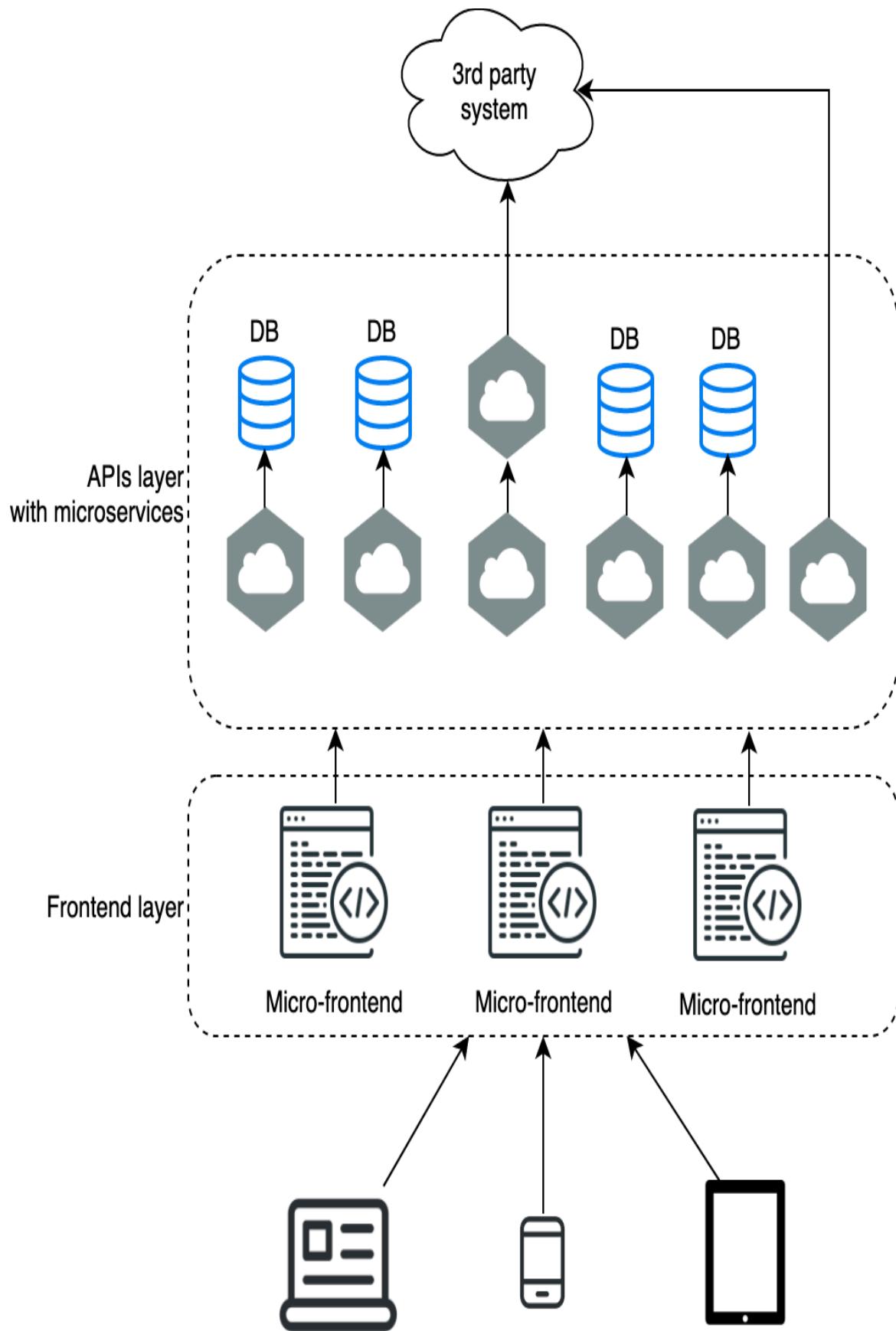
Code abstraction is not a silver bullet either; prematurely abstracting code for reuse often causes more problems than benefits. I have frequently seen abstractions make code thousands of times more complicated than necessary to be reused just twice inside the same project. Many developers are prone to over-engineering some solutions, thinking they will reuse them tens of times, but in reality, they use them far fewer times. Using libraries across multiple projects and teams could end up producing more complexity than benefits such as making the codebase more complex or requiring more effort on manual testing or adding overhead in communications.

We also need to consider the monolith approach on the frontend. Such an approach won't allow us to improve our architecture in the long run, particularly if we are working on platforms meant to be available for our users for many years or if we have distributed teams in different time zones.

Asking any business to extensively revise the tech it uses will cause a large investment upfront before it gets any results.

Now the question becomes quite obvious: **Do we have the opportunity to use a well-known pattern or architecture that offers the possibility of adding new features quickly, evolving with the business, and delivering part of the application autonomously without big-bang releases?**

I picture something like [Figure 1-3](#):



*Figure 1-3. Micro-architectures combined, this is a high-level diagram showing how Microservices and Micro-frontends can live together*

The answer is YES!

We can definitely do it and it's where micro-frontends come to the rescue.

This architecture makes more sense when we deal with mid-large companies and during the following chapters, we are going to explore how to successfully structure our micro-frontends architectures.

However, first we need to understand what the main principles are behind micro-frontends to leverage as guidance during the development of our projects.

## Microservices Principles

At the beginning of my journey into micro-frontends in 2016, there wasn't any guidance on how to structure such architecture, therefore I had to take a step back from the technical implementation and look at the principles behind other architectures for scaling a software project. Would those principles be applicable to the frontend too?

Microservices' principles offer quite a few useful concepts. Sam Newman has highlighted these ideas in his book - [Building Microservices \(O'Reilly\)](#). I've summarized the theories in [Figure 1-4](#):

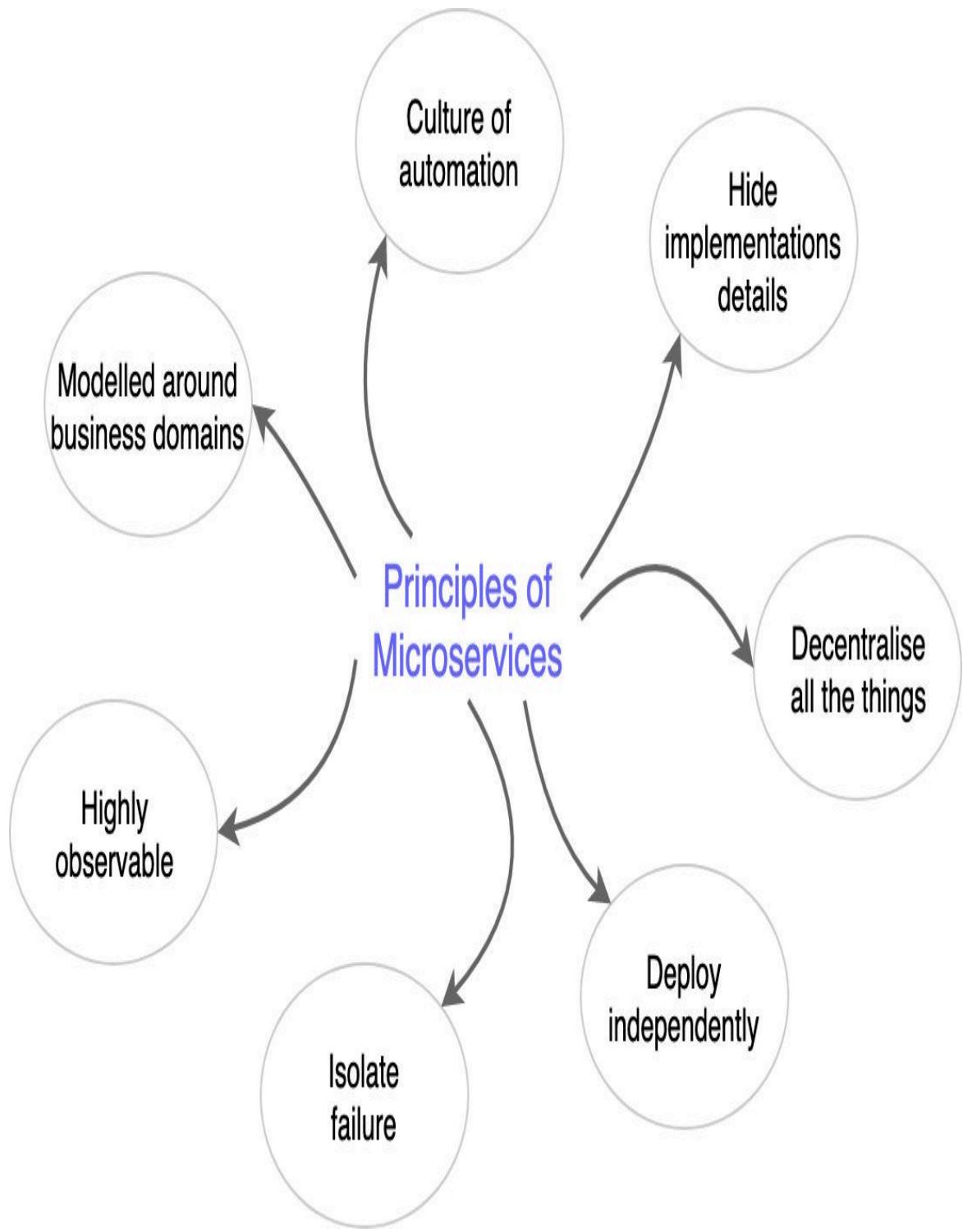


Figure 1-4. Microservices principles

Let's discuss the above principles and see how they apply to the frontend.

## **Modeled Around Business Domains**

Modeling around business domains is a key principle brought up by domain-driven design (DDD). It starts from the assumption that each piece of software should reflect what the organization does and that we should design our architectures based on domains and subdomains, leveraging ubiquitous languages shared across the business.

When working from a business point of view, this provides several benefits, including a better understanding of the system, an easier definition of a technical representation of a business domain, and clear boundaries on which a team should operate.

## **Culture of Automation**

Considering that microservices are a multitude of services that should be autonomous, we need a robust culture of automating the deployment of independent units in different environments. In my experience, this is a key process for leveraging microservices architecture; having a strong automation culture allows us to move faster and provide a better feedback loop for developers that will relay to all the capabilities offered by the company in terms of security and performance guardrails that are part of the continuous integration process.

## **Hide Implementation Details**

Hiding implementation details when releasing autonomously is crucial. If we are sharing a database between microservices, we won't be able to change the database schema without affecting all the microservices relying on the original schema. DDD teaches us how to encapsulate services inside the same business domain, exposing only what is needed via APIs and hiding the rest of the implementation. This allows us to change internal logic at our own pace without impacting the rest of the system. Very often, we call this approach API-First. We begin by defining the APIs, which serve as the contract binding the producer and consumer(s) teams. This allows them to work in parallel, focusing on either producing or consuming the

specified contract. By focusing on the API early in the development process, teams can enhance collaboration, scalability, and adaptability, making it easier to integrate and extend functionalities as the project evolves.

## **Decentralize All the Things**

Decentralizing the governance empowers developers to make the right decision at the right stage to solve a problem. With a monolith, many key decisions are often made by the most experienced people in the organization. These decisions, however, frequently lead to trade-offs alongside the software lifecycle. Decentralizing these decisions could have a positive impact on the entire system by allowing a team to take a technical direction based on the problems they are facing, instead of creating compromises for the entire system. Bear in mind that in distributed systems a team has less cognitive load to carry, therefore each team member quickly becomes a domain expert in a portion of the system and can provide the best decision to evolve its own domain.

## **Deploy Independently**

Independent deployment is key for microservices. With monoliths, we are used to deploying the entire system every time, with a greater risk of live issues and longer times for deploying and rolling back our artifacts. With microservices, however, we can deploy autonomously without increasing the possibility of breaking our entire API layer. Furthermore, we have solid techniques, like blue-green deployment or canary releases that allow us to release a new version of a microservice with even less risk, which clears the path for new or updated APIs.

## **Isolate Failure**

Because we are splitting a monolith into tens, if not hundreds, of services, if one or more microservices becomes unreachable due to network issues or service failures, the rest of the system should be available for our users.

There are several patterns for providing graceful failures with microservices and the fact that they are autonomous and independent just reinforces the concept of isolating failure.

## Highly Observable

One reason that you would favor monolithic architecture in comparison to microservices is that it is easier to observe a single system than a system split in multiple services. Microservices provide a lot of freedom and flexibility, but this doesn't come for free; we need to keep an eye on everything through logs, monitors, and so on. For example, we must be ready to follow a specific client request end to end inside our system. Keeping the system highly observable is one of the main challenges of microservices.

Embracing these principles in a microservices environment will require a shift in mindset not only for your software architecture but also for how your company is organized. It involves moving from a centralized to a decentralized paradigm, enabling cross-functional teams to own their business domains end to end. This can be a particularly huge change for medium to large organizations.

## Applying Principles to Micro-frontends

Now that we've grasped the principles behind microservices, let's find out how to apply them to a frontend application.

## Modeled Around Business Domains

Modeling micro-frontends to follow DDD principles is not only possible but also very valuable. Investing time at the beginning of a project to identify the different business domains and how to divide the application will be very useful when you add new functionalities or depart from the initial project vision in the future. DDD can provide a clear direction for managing backend projects, but we can also apply some of these techniques

on the frontend. Granting teams full ownership of their business domain can be very powerful, especially when product teams are empowered to work with technology teams. The primary difference between a micro-frontend and a component lies in their modularization approach. A micro-frontend completely owns a business domain, whereas a component focuses on addressing a technical challenge, often characterized by code duplication or the creation of complex, configurable components used across multiple domains. The component approach exposes an API that is frequently coupled with its container. Therefore, any modification made to the component is likely to impact its containers as well, creating an unwanted coupling that prevents it from reaching the principles behind distributed systems. With micro-frontends, we streamline the API surface to the essential minimum required for comprehending the user's context. Typically, micro-frontends require little beyond accessing a session token and other pertinent information such as a product ID. This approach effectively diminishes the coupling between elements of the frontend application and enhances team autonomy by reducing the need for coordination across teams, owing to the infrequent changes in the minimal API exposed.

## Culture of Automation

As for the microservices architecture, we cannot afford to have a poor automation culture inside our organization; otherwise any micro-frontends approach we are going to take will end up a pure nightmare for all our teams. Considering that every project contains tens or hundreds of different parts, we must ensure that our continuous integration and deployment pipelines are solid and have a fast feedback loop for embracing this architecture. Investing time in getting our automation right will result in the smooth adoption of micro-frontends and will solve common challenges like aligning shared libraries to a specific version, enforcing budget size per micro-frontend or forcing to update every micro-frontend to the latest design system version. Moreover, automation is not important only for generating technical artifacts, more importantly it provides a fast feedback

loop for developers. Creating fast and helpful feedback loops for developers will foster the right behaviors inside the teams enforcing important architecture characteristics across the distributed system.

## Hide Implementation Details

Hiding implementation details and working with contracts are two essential practices, especially when parts of our application need to communicate with each other. It's crucial to define upfront an API contract that is shared across the teams who need to interact with different micro-frontends. Also, strong encapsulation is required to avoid domain leaks in other parts of the application. In this way each team will be able to change the implementation details without impacting other teams unless there is an API contract change. These practices allow a team to focus on the internal implementation details without disrupting the work of other teams. Each team can work at its own pace, drastically reducing external dependencies and creating more effective collaboration.

## Decentralization over Centralization

Decentralizing a team's decisions finally moves us away from a one-size-fits-all approach that often ends up being the lowest common denominator. Instead, the team will use the right approach or tool for the job. As with microservices, the team is in the best position to make certain decisions when it becomes an expert in the business domain. This doesn't mean each team should take its own direction but rather that the tech leadership (architects, principal engineers, CTOs) in conjunction with the developers and practices applied in the field, should *provide guardrails* between which teams can operate without needing to wait for a central decision. This leads to a sharing culture inside the organization becoming essential for introducing successful practices across teams.

## **Deploy Independently**

Micro-frontends allow teams to deploy independent artifacts at their own speed. They don't need to wait for external dependencies to be resolved before deploying. Achieving independence in micro-frontends means not reducing the user interface to mere components. We need to reduce the external dependencies for a team, in this way we optimize for a fast flow that will enable a team to run their operations independently.

When we combine this approach with microservices, a team can own a business domain end to end, with the ability to make technical decisions based on the challenges inside their business domain rather than finding a one-size-fits-all approach.

## **Isolate Failure**

Isolating failure in SPAs, for instance, isn't a huge problem due to their architecture, but it is with micro-frontends. In fact, micro-frontends require composing a user interface at runtime, which may result in network failures or 404 errors for one or more parts of the UI. To avoid impacting the user experience, we must provide alternative content or hide a specific part of the application. This might result in gracefully hiding non-essential micro-frontends from the interface if they fail or return a 500 error, in case the main micro-frontend of a page is not loaded.

## **Highly Observable**

Frontend observability is becoming more prominent every day, with tools like Sentry, New Relic or LogRockets providing great visibility for every developer. Using these tools is essential to understanding where our application is failing and why. As Werner Vogels, Amazon's CTO, used to say: "everything fails all the time", therefore being able to resolve issues quickly is far more important than preventing problems. This moves us toward a paradigm where we can better invest our resources by remaining ready to address system failures rather than trying to prevent them

completely. As with all microservices' principles, this is applicable to the frontend, too.

The exciting part of recognizing these principles on the frontend and backend is that, finally, we have a solution that will empower our development teams to own the entire range of a business domain, offering a simpler way to divide labor across the organization and iterate improvements swiftly into our system.

When we start this journey into the *micro-world* we need to be conscious of the level of complexity we are adding to a project, which may not be required for any other projects.

There are plenty of companies that prefer using a monolith over microservices because of the intrinsic complexity they bring to the table. For the same reason, we must understand when and how to use micro-frontends properly, as not all projects are suitable for them.

## Micro-frontends are not a silver bullet

It's very important that we use the right tool for the right job. I cannot stress this point enough. Too often I have seen projects failing or drastically delayed due to poor architectural decisions.

We need to remember that:

### NOTE

**Micro-frontends are not appropriate for every application** because of their nature and the potential complexity they add at the technical and organizational levels.

Micro-frontends are a sensible option when we are working on software that requires an iterative approach and long-term maintenance, when we have projects that require a large development team, in multi-tenant applications, or when we want to replace a legacy project in an iterative way.

However, they are not suitable for all frontend applications, they are an additional available option of frontend architecture for our projects. Micro-frontends architecture has plenty of benefits but also has plenty of drawbacks and challenges. If the latter exceed the former, micro-frontends are not the right approach for a project. As Neal Ford and Mark Andrew Richards have described in their book *Software Architecture*, “Don’t try to find the best design in software architecture, instead, strive for the least worst combination of trade-offs.” This should be your mantra from now on!

## Summary

In this chapter we introduced what micro-frontends are, what their principles are, and how those principles are linked to an architecture like microservices that was created for solving similar challenges.

Next, we will explore how to structure a micro-frontend project from an architectural point of view and the key technical challenges to understand when we design our frontend applications using them.

# Chapter 2. Micro-Frontend Architectures and Challenges

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the second chapter of the final book. Please note that the GitHub repo will be made active later on.

If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the author at [\*building.microfrontends@gmail.com\*](mailto:building.microfrontends@gmail.com).

A micro-frontend represents a business subdomain that is autonomous, independently deliverable, with same or different technology, with low degree of coupling and owned by a single team. We can summarize the key takeaways in this description with the following characteristics:

- Business domain representation
- Autonomous codebase
- Independent deployment
- Low coupling
- Optimized for fast-flow
- Single-team ownership

Micro-frontends offer many opportunities. Choosing the right one depends on the project requirements, the organization structure, and the developer's experience.

In these architectures, we face some specific challenges to success bound by similar questions, such as how we want to communicate between micro-frontends, how we want to route the user from one view to another, and, most importantly, how we identify the size of a micro-frontend.

In this chapter, we will cover the key decisions to make when we initiate a project with a micro-frontends architecture. We'll then discuss some of the companies using micro-frontends in production and their approaches.

## Micro-frontends Decisions Framework

There are different approaches for architecting a micro-frontends application. To choose the best approach for our project, we need to understand the context we'll be operating in.

Some architectural decisions will need to be made upfront because they will direct future decisions, like how to define a micro-frontend, how to orchestrate the different views, how to compose the final view for the user, and how micro-frontends will communicate and share data.

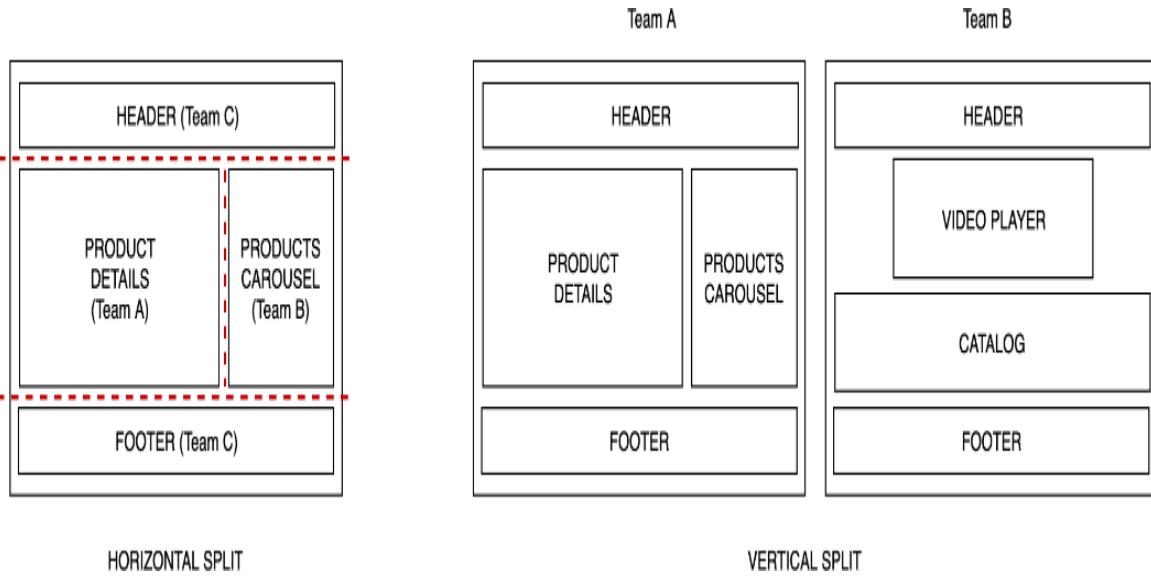
These types of decisions are called the **micro-frontends decisions framework**. It is composed of four key areas:

- defining what a micro-frontend is in your architecture
- composing micro-frontends
- routing micro-frontends
- communicating between micro-frontends

## Define Micro-frontends

Let's start with the first key decision, which will have a heavy impact on the rest. We need to identify how we consider a micro-frontend from a technical point of view.

We can decide to have multiple micro-frontends in the same view or having only one micro-frontend per view ([Figure 2-1](#)).



*Figure 2-1. Horizontal vs. vertical split*

With the horizontal split, multiple micro-frontends will be on the same view. Multiple teams will be responsible for parts of the view and will need to coordinate their efforts. This approach provides greater flexibility considering we can even reuse some micro-frontends in different views, although it also requires more discipline and governance for not ending up with hundreds of micro-frontends in the same project. Very often, higher granularity would end up with higher coupling and the risk of creating a distributed monolith.

## DISTRIBUTED MONOLITH

A distributed monolith in software architecture refers to a system that, despite being distributed across multiple servers or nodes, exhibits characteristics commonly associated with a monolithic architecture. In this context, the term “monolith” implies a single, tightly-coupled unit with interconnected components that lack clear separation of concerns. The distributed nature of the system may introduce complexities in terms of communication between components spread across different locations, but the overall structure remains monolithic in its design and interdependencies. This can hinder the independent nature of micro-frontends, risking having several external dependencies that will nullify the effort of building such architecture.

In the vertical split scenario, each team is responsible for a business domain, like the authentication or the catalog experience. In this case, domain-driven design (DDD) comes to the rescue. It's not often that we apply DDD principles on frontend architectures, but in this case, we have a good reason to explore it.

DDD is an approach to software development that centers the development on programming a domain model that has a rich understanding of the processes and rules of a domain.

Applying DDD to the frontend is slightly different from the approach taken on the backend. Certain concepts are not applicable, while others are fundamental for designing a successful micro-frontends architecture.

When examining the system holistically, you might wonder how to identify different areas that are independent. Various techniques exist, and one of my favorites by far is event storming ([Figure 2-2](#)). Event storming is a workshop that brings together individuals from the same company with different backgrounds, including product managers, testers, and developers. The focus of the workshop is on the business side, rather than the technical side.

By assembling people from various roles in the same room, you can create a timeline that describes the system end-to-end or, at least, a portion of it. This approach allows you to identify potential independent parts of the system by examining the vocabulary defined during the session.

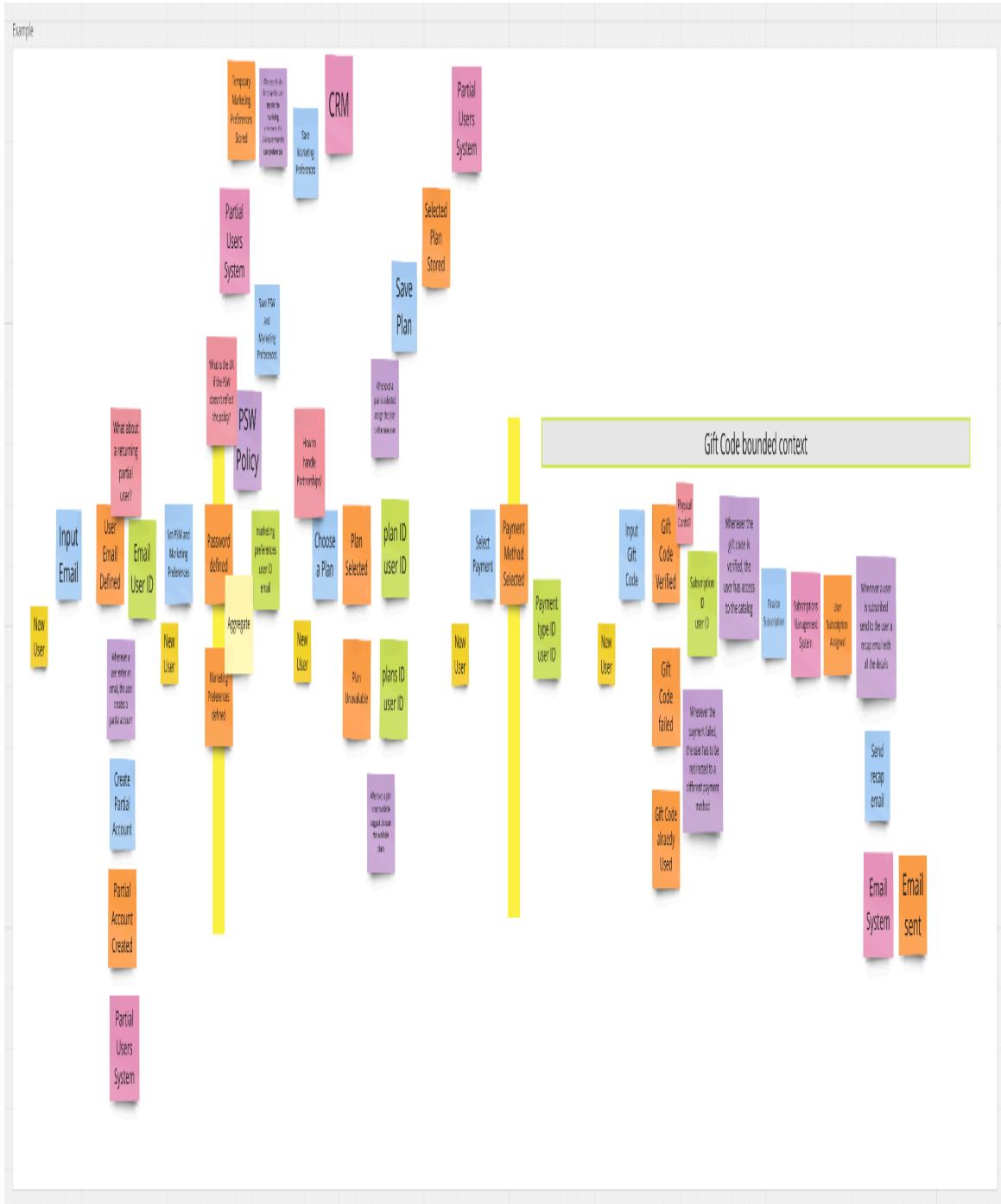


Figure 2-2. An example of Event Storming outcome for the on-boarding experience of a subscription service

Thanks to this workshop, that works for a system end-to-end not only for the frontend side, you can visualize your system, having a better understanding, but more importantly recognising the different parts that compose it, or as DDD would call them: subdomains.

In the context of DDD, subdomains refer to distinct and isolated components within a larger business domain. Each subdomain represents a specialized area with its own unique set of responsibilities, business logic, and models. The purpose of identifying subdomains is to facilitate a modular and organized approach to software development, allowing teams to focus on specific aspects of the overall business functionality. Subdomains are delineated based on clear and cohesive boundaries, enabling more effective management, development, and maintenance of complex systems by addressing individual business concerns in a targeted manner.

## EVENT STORMING

It's out of the scope of this book teaching you Event Storming, however I highly encouraging you to read the chapter on this subject from [Learning Domain Driven Design book by O'reilly](#).

DDD provides three subdomain types, but I want to provide a concrete example for you to understand better what they refer to:

- *Core subdomains*: These are the main reasons an application should exist. Core subdomains should be treated as a premium citizen in our organizations because they are the ones that deliver value above everything else. The video catalog would be a core subdomain for Netflix.
- *Supporting subdomains*: These subdomains are related to the core ones but are not key differentiators. They could support the core subdomains but aren't essential for delivering real value to users. One example would be the voting system on Netflix's videos.

- *Generic subdomains*: These subdomains are used for completing the platform. Often companies decide to go with off-the-shelf software because they're not strictly related to their domain. With Netflix, for instance, the payments management is not related to the core subdomain (the catalog), but it is a key part of the platform because it has access to the authenticated section.

Let's break down Netflix into these categories ([Table 2-1](#)).

*T  
a  
b  
le  
2  
-  
l  
.S  
u  
b  
d  
o  
m  
a  
i  
n  
s  
e  
x  
a  
m  
p  
le  
s*

---

**Subdomain type   Example**

---

Core subdomain

Catalog

---

Supportive subdomain Voting system

Generic subdomain Sign in or sign up

Why categorize subdomains, you may wonder? The answer is straightforward: you can apply different characteristics to each subdomain through this categorization.

For instance, a core domain is the essence behind your system's functionality. Therefore, investing in developer seniority, code quality, and a fast feedback loop will likely yield the best outcomes.

On the contrary, a generic domain lacks a competitive advantage. In such cases, opting for an off-the-shelf solution with integration into your system may suffice for achieving its objectives, the changes on this part of the system won't be as frequent as in other parts, and the complexity of the code to write might not be very high, therefore you can take another strategy for assembling a development team compared to other subdomains.

In essence, DDD offers more than just a rich vocabulary for system description. It introduces heuristics and techniques that guide organizations in the right direction concerning both technology and organizational structure for the first time.

## Domain-Driven Design with Micro-Frontends

After identifying subdomains, DDD introduces another concept: the *bounded context*. It's a logical boundary that hides the implementation details, exposing an application programming interface (API) contract to consume data from the model present in it.

Usually, the bounded context translates the business areas defined by domains and subdomains into logical areas where we define the model, our code structure, and potentially, our teams. Bounded context defines the way

different contexts are communicating with each other by creating a contract between them, often represented by APIs. This allows teams to work simultaneously on different subdomains while respecting the contract defined upfront.

Often in a new project, subdomains overlap bounded context because we have the freedom to design our system in the best way possible. Therefore, we can assign a specific subdomain to a team for delivering a certain business value defining the contract. However, in legacy software, these lines might be more blurred due to lack of analysis during the project lifecycle.

Too often we identify early on a technical solution without gathering the architecture characteristics we have to optimize for.

Think about this scenario: three teams, distributed in three different locations, working on the same codebase.

These teams may go for a horizontal split using iframes or web components for their micro-frontends. After a while, they realize that micro-frontends in the same view must communicate somehow. One of those teams will then be responsible for aggregating the different parts inside the view. The team will spend more time aggregating different micro-frontends in the same view and debugging to ensure everything works properly.

Obviously, this is an oversimplification. It could be worse when taking into consideration the different time zones, cross-dependencies between teams, knowledge sharing, or distributed team structure for example.

All those challenges could escalate very easily to low morale and frustration on top of delivery delays. Therefore we need to be sure the path we are taking won't let our teams down.

Approaching the project from a business point of view, however, allows you to create an independent micro-frontend with less need to communicate across multiple subdomains.

Let's re-imagine our scenario. Instead of working with web components or iframes, we are working with single page applications (SPAs) and single

pages.

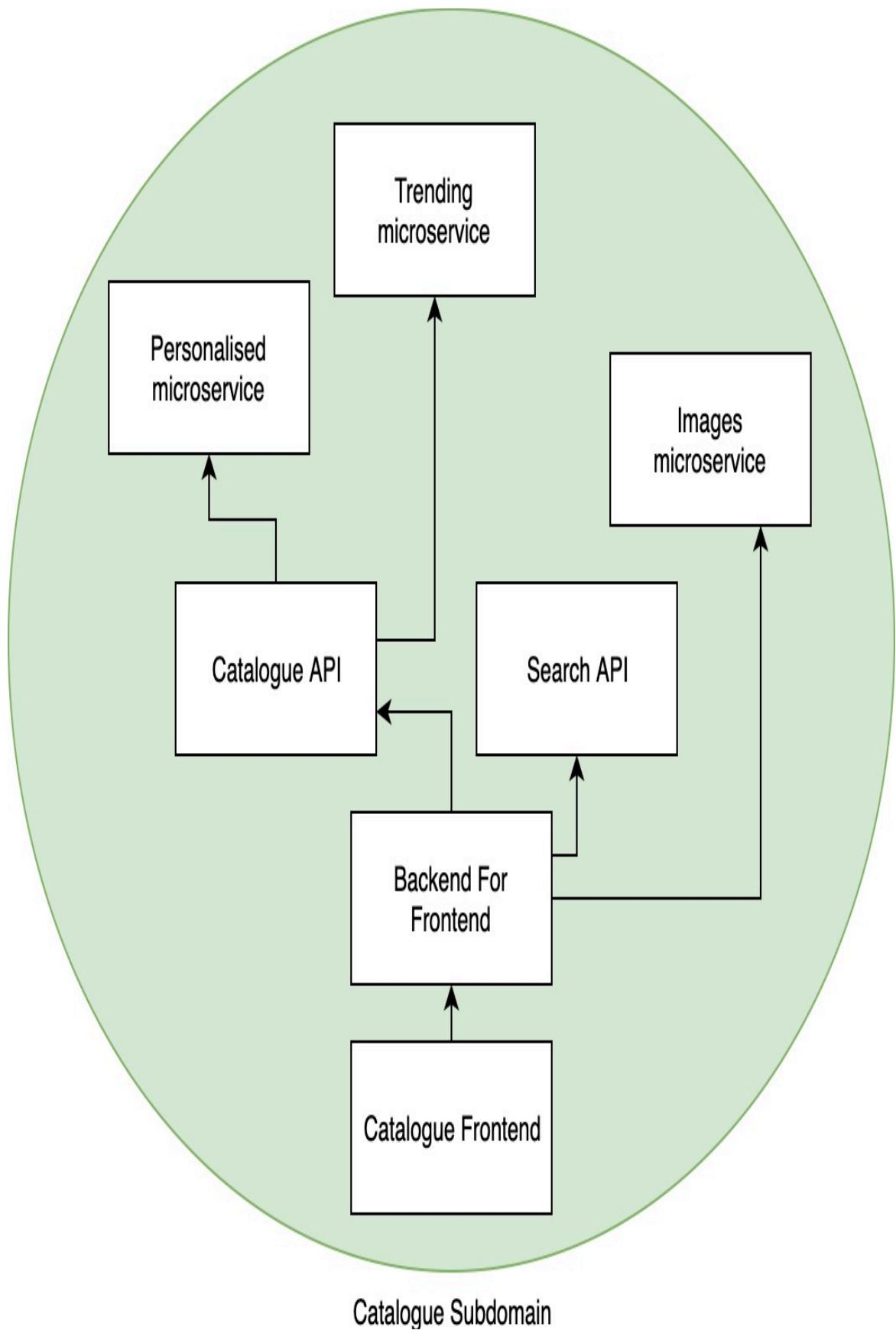
This approach allows a full team to design all the APIs needed to compose a view and to create the infrastructure needed to scale the services according to the traffic. The combination of micro-architectures, microservices, and micro-frontends provides independent delivery without high risks for compromising the entire system for release in production.

The bounded context helps design our systems, but we need to have a good understanding of how the business works to identify the right boundaries inside our project.

Developers, tech leads or architects have to come closer to the product teams, investing enough time with them and understanding the customers' needs so they can identify the different domains and subdomains, working collaboratively with the product teams. Once again, event storming could be a natural fit in these cases.

After defining all the bounded contexts, we will have a map of our system representing the different areas that our system is composed of. In [Figure 2-3](#) we can see a representation of bounded context. In this example the bounded context contains the catalogue micro-frontends that consume APIs from a microservices architecture via a unique entry point, a backend for frontend, we will investigate more about the APIs integration in chapter 9.

In DDD, the frontend is not taken into consideration but when we work with micro-frontends with a vertical split we can easily map the frontend and the backend together inside the same bounded context.



*Figure 2-3. This is a representation of bounded context*

I've often seen companies design systems based on their team's structure (*Conway's Law* states "organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations."). Instead, they needed their team structure to be flexible enough to adapt to the best possible solution for the organization in order to reduce friction and move faster toward the final goal: having a great product that satisfies customers (*Inverse Conway's Maneuver* recommends evolving your team and organizational structure to promote your desired architecture.)!

Both approaches to structure your teams and design the system architecture are fine, as long as it becomes clear the coupling between the organization structure and software architecture. Very often a change in one of these two areas will affect the other indirectly.

## **How to define a bounded context**

Premature optimization is always around the corner, which can lead to our subdomains decomposing where we split our bounded contexts to accommodate future integrations. Instead, we need to wait until we have enough information to make an educated decision.

Because our business evolves over time, we also need to review our decisions related to bounded contexts and subdomain type.

Sometimes we start with a larger bounded context. Over time the business evolves and eventually, the bounded context becomes unmanageable or too complex. So we decided to split it. Deciding to split a bounded context could result in a large code refactor but could also simplify the codebase drastically, speeding up new functionalities and development in the future.

To avoid premature decomposition, we will make the decision at the last possible moment. This way we have more information and clarity on which direction we need to follow. We must engage upfront with the product team or the domain experts inside our organization as we define the subdomains.

They can provide you with the context of where the system operates. Always begin with data and metrics.

For instance, we can easily find out how our users are interacting with our application and what the user journey is when a user is authenticated and when they're not. Data provides powerful clarity when identifying a subdomain and can help create an initial baseline, from where we can see if we are improving the system or not.

If there isn't much observability inside our system, let's invest time to create it. Doing so will pay off the moment we start identifying our micro-frontends.

Without dashboards and metrics, we are blind to how our users operate inside our applications.

Let's assume we see a huge amount of traffic on the landing page, with 70% of those users moving to the authentication journey (sign in, sign up, payment, etc.). From here, only 40% of the traffic subscribes to a service or uses their credentials for accessing the service.

These are good indications about our users' behaviors on our platform. Following DDD, we would start from our application's domain model, identifying the subdomains and their related bounded context and using behavioral data to guide us on how to slice the frontend applications.

Allowing users to download only the code related to the landing page will give them a faster experience because they won't have to download the entire application immediately, and the 40% of users who won't move forward to the authentication area will have just enough code downloaded for understanding our service.

Obviously, mobile devices with slow connections only benefit from this approach for multiple reasons: less data is downloaded, less memory is used, less JavaScript is parsed and executed, resulting in a faster first interaction of the page.

It's important to remember that not all user sessions contain all the URLs exposed by our platform. Therefore a bit of research upfront will help us

provide a better user experience.

Usually, the decision to pick a horizontal split instead of vertical split depends on the type of project we have to build. In the next chapter, we will deep dive into this topic. Bear in mind, they are not mutually exclusive. You might have part of the application where a vertical split is more appropriate than a horizontal and vice versa.

Another thing to consider is the skills set of our teams, usually, a vertical split suits better teams that are new to micro-frontends, instead, the horizontal split requires an investment upfront for creating a solid and fast development experience to test their part as well as trying inside the overall view.

## **Testing your micro-frontend boundaries**

Often, I've conducted meetings with teams that have implemented a micro-frontends architecture but treated a micro-frontend as if it were a component loaded at runtime. I have developed a mental model that can assist you in determining whether your boundaries are well-established.

1. To enhance the robustness of your architecture, consider reducing the API surface exposed to the containers. When you expose too many properties of a micro-frontend, the risk of coupling increases significantly. This is because you allow the container of the micro-frontend to own the context instead of the micro-frontend itself. This leads to accidental complexity that becomes evident when deploying a micro-frontend and constant coordination efforts across teams.
2. Micro-frontends are inherently context-aware. Typically, a micro-frontend requires a minimal amount of information to function properly. They are designed with awareness of the context. For example, passing a product ID or enabling the retrieval of a session token to consume an API are common characteristics of the horizontal split approach. More common properties shared from

the micro-frontend container should lead you to question the implementation and revisit the API contract or the micro-frontends boundaries.

3. In contrast to components, micro-frontends are less extensible. In designing a component, the focus is on high reusability and code abstraction. Micro-frontends, however, are designed for independence and minimal external dependencies. Due to their context-aware nature, they are less likely to be extensible or composed with each other. A sign of wrong boundaries is a proliferation of micro-frontends per view or deep nesting between micro-frontends.
4. Furthermore, micro-frontends are more coarse-grained than components. While a classic component, such as a button, are small and highly flexible to be composed with other components, micro-frontends are highly specialized in their functionality. This specialization limits their reusability, and they are unlikely to be extensible for creating “larger micro-frontends”. It is recommended to avoid fine-grained micro-frontends, as they tend to result in a higher degree of coupling, external dependencies, and context leakage towards their containers.

Having gained a comprehensive understanding of micro-frontends and their identification, let's now delve into the robust mental models widely embraced within the frontend community – the Micro-Frontends Decisions Framework.

## **Micro-frontends composition**

There are different approaches for composing a micro-frontends application ([Figure 2-4](#)).

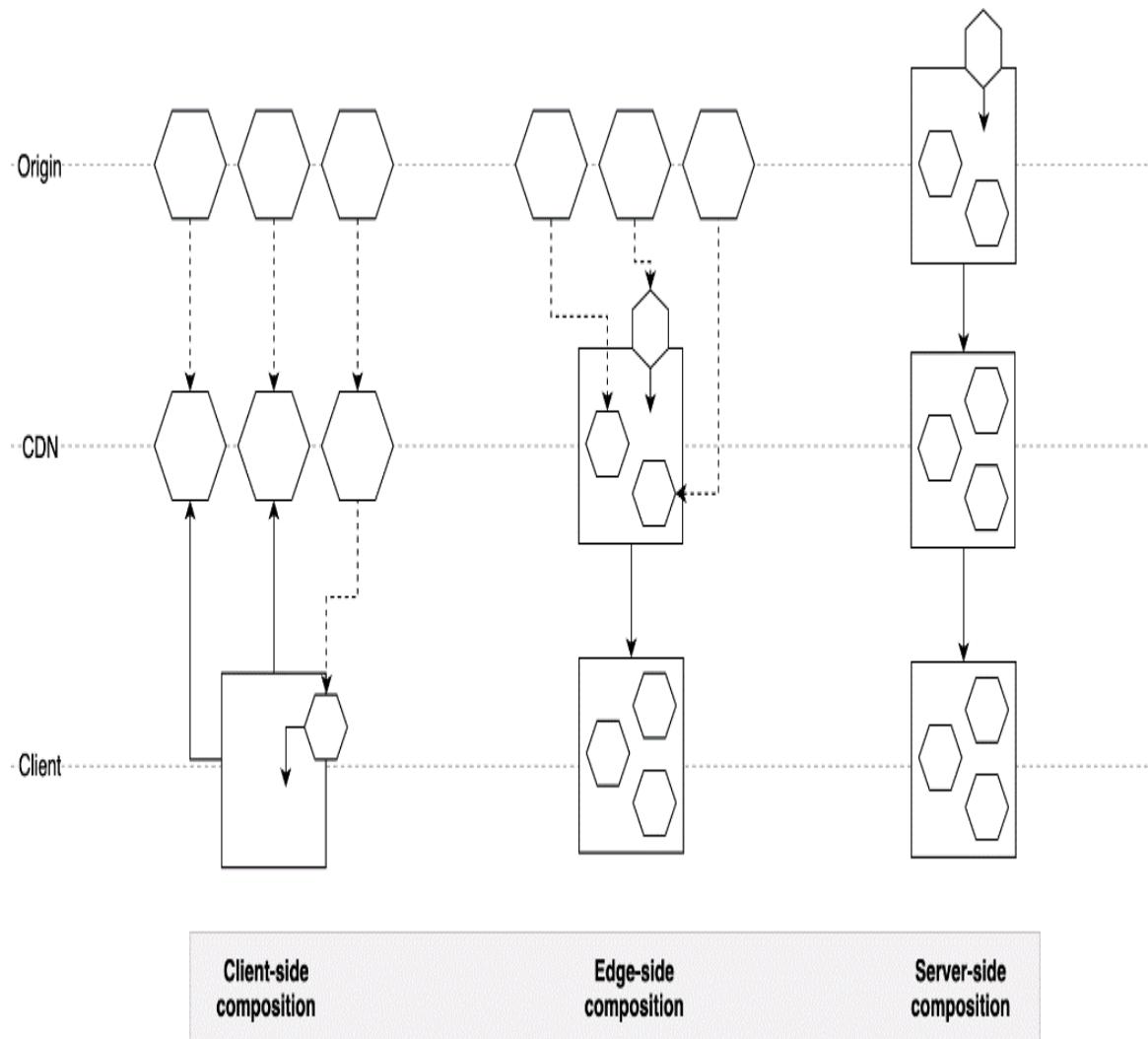


Figure 2-4. Micro-frontends composition diagram

In this diagram we can see three different ways to compose a micro-frontends architecture:

- Client-side composition
- Edge-side composition
- Server-side composition

Starting from the left of our diagram, we have a client-side composition, where an application shell loads multiple micro-frontends directly from a content delivery network (CDN), or from the origin if the micro-frontend is not yet cached at the CDN level. This composition is beneficial either for

horizontal or vertical split micro-frontends. In the middle of the diagram, we compose the final view at the CDN level, retrieving our micro-frontends from the origin and delivering the final result to the client. The right side of the diagram shows a micro-frontends composition at the origin level where our micro-frontends are composed inside a view, cached at the CDN level, and finally served to the client. For edge-side and server-side composition, we mainly use a horizontal split approach.

Let's now observe how we can technically implement this architecture.

## Client-Side Composition

In the client-side composition case, where an application shell loads micro-frontends inside itself, the micro-frontends should have a JavaScript or HTML file as an entry point so the application shell can dynamically append the document object model (DOM) nodes in the case of an HTML file or initializing the JavaScript application with a JavaScript file or an EcmaScript module.

In the beginning, we also used a combination of iframes to load different micro-frontends, or a transclusion mechanism on the client side via a technique called client-side include. Client-side include lazy-loads components, substituting empty placeholder tags with complex components. For example, a library called *h-include* uses placeholder tags that will create an AJAX request to a URL and replace the inner HTML of the element with the response of the request.

This approach gives us many options, but using client side includes has a different effect than using iframes. From 2019 onwards, more micro-frontends solutions started to gain traction for building a successful client-side composition such as Module Federation or Single SPA. In the next chapters we will explore this part in detail.

## NOTE

According to [Wikipedia](#), in computer science, *transclusion* is the inclusion of part or all of an electronic document into one or more other documents by hypertext reference. Transclusion is usually performed when the referencing document is displayed and is normally automatic and transparent to the end user. The result of transclusion is a single integrated document made of parts assembled dynamically from separate sources, possibly stored on different computers in disparate places.

An example of transclusion is the placement of images in HTML. The server asks the client to load a resource at a particular location and insert it into a particular part of the DOM.

## Edge-Side Composition

With edge-side composition, we assemble the view at the CDN level. Many CDN providers give us the option of using an XML-based markup language called Edge Side Include (ESI). [ESI](#) is not a new language; it was proposed as a standard by Akamai and Oracle, among others, in 2001. ESI allows a web infrastructure to be scaled in order to exploit the large number of points of presence around the world provided by a CDN network, compared to the limited amount of data center capacity on which most software is normally hosted. One drawback to ESI is that it's not implemented in the same way by each CDN provider; therefore, a multi-CDN strategy, as well as porting our code from one provider to another, could result in a lot of refactors and potentially new logic to implement. It's important to highlight that this practice is not embraced massively by organizations world-wide. The recommendation is to use mainly client-side or server-side composition.

## Server-Side Composition

The last possibility we have is the server-side composition. In this case, the origin server is composing the view by retrieving all the different micro-frontends and assembling the final page. If the page is highly cacheable, the CDN will then serve it with a long time-to-live policy. However, if the page is personalized per user, serious consideration will be required regarding the scalability of the eventual solution, when there are many requests coming from different clients. When we decide to use server-side composition we

must deeply analyze the use cases we have in our application. If we decide to have a runtime composition, we must have a clear scalability strategy for our servers in order to avoid downtime for our users.

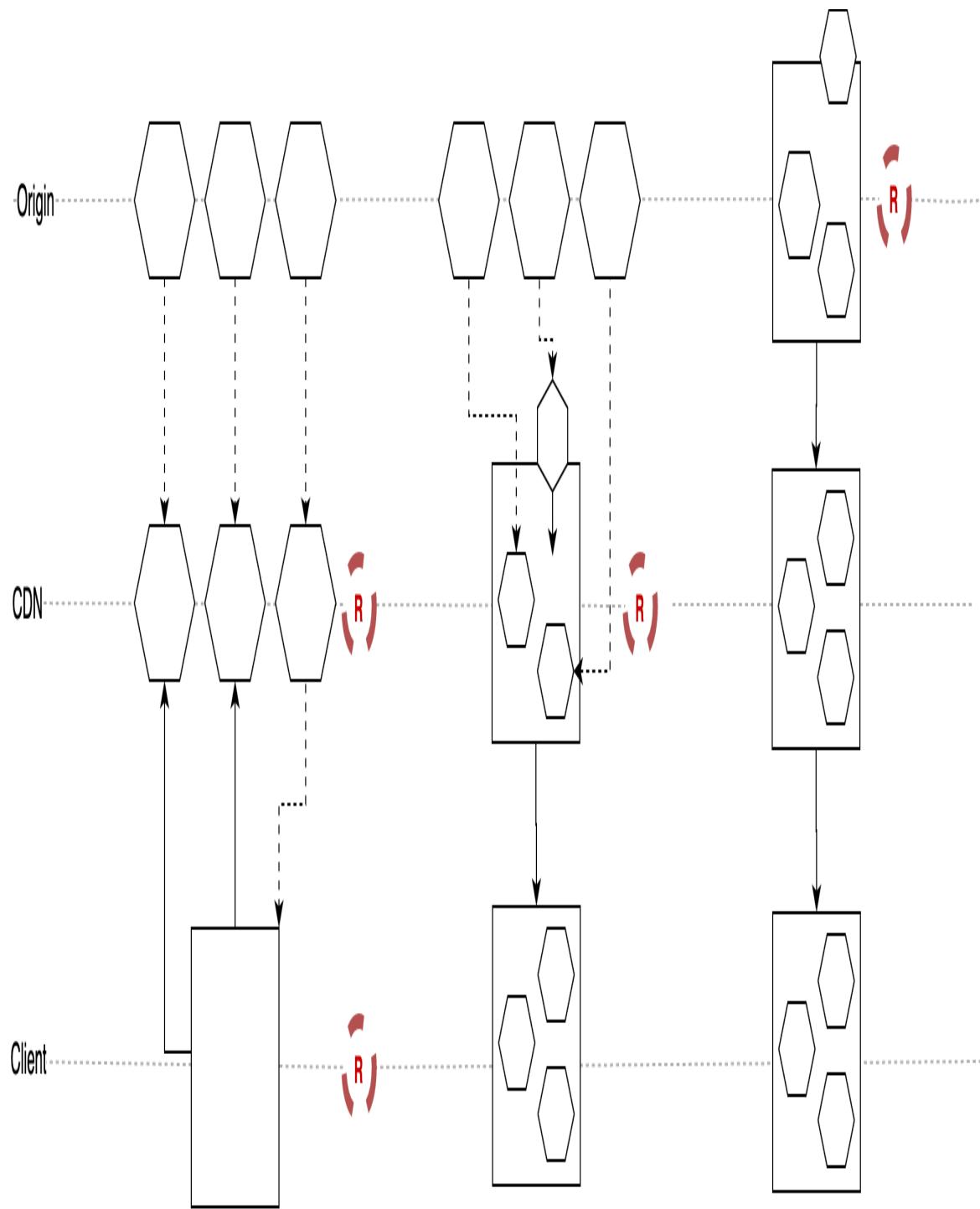
From these possibilities, we need to choose the technique that is most suitable for our project and the teams' knowledge. As we will learn later on in this journey, we also have the opportunity to deploy an architecture that exploits both client-side and server-side composition—that's absolutely fine as long we understand how to structure our project.

## Routing micro-frontends

The next important choice we have is how to route the application views.

This decision is strictly linked to the micro-frontends composition mechanism we intend to use for the project.

We can decide to route the page requests in the origin, on the edge, or at client-side ([Figure 2-5](#)).



Client-side  
composition

Edge-side  
composition

Server-side  
composition

*Figure 2-5. Micro-frontends routing diagram*

When we decide to compose micro-frontends at the origin, see the server-side composition on the right of [Figure 2-5](#), we are forced to route the requests at origin considering the entire application logic lives in the application servers.

However, we need to consider that scaling an infrastructure could be nontrivial, especially when we have to manage burst traffic with many requests per second (RPS). Our servers need to be able to keep up with all the requests and scale horizontally very rapidly. Each application server then must be able to retrieve the micro-frontends for the composing page to be served.

We can mitigate this problem with the help of a CDN. The main downside is that when we have dynamic or personalized data, we won't be able to rely extensively on the CDN serving our pages because the data would be outdated or not personalized.

When we decide to use edge-side composition in our architecture, the routing is based on the page URL and the CDN serves the page requested by assembling the micro-frontends via transclusion at edge level.

In this case, we won't have much room for creating smart routing—something to remember when we pick this architecture.

The final option is to use client-side routing. In this instance, we will load our micro-frontends according to the user state, such as loading the authenticated area of the application when the user is already authenticated or loading just a landing page if the user is accessing our application for the first time.

If we use an application shell that loads a micro-frontend, the application shell is responsible for owning the routing logic, which means the application shell retrieves the routing configuration first and then decides which micro-frontend to load.

This is a perfect approach when we have complex routing, such as when our micro-frontends are based on authentication, geo-localization, or any

other sophisticated logic. When we are using a multipage website, micro-frontends may be loaded via client-side transclusion. There is almost no routing logic that applies to this kind of architecture because the client relies completely on the URL typed by the user in the browser or the hyperlink chosen on another page, similar to what we have when we use edge-side include approach. We won't have any scalability issues in either case.

Those routing approaches are not mutually exclusive, either. As we will see later in this book, we can combine those approaches using CDN and origin or client-side and CDN together.

The important thing is determining how we want to route our application. This fundamental decision will affect how we develop our micro-frontends application.

## **Micro-frontends communication**

When we have multiple micro-frontends on the same page, the complexity of managing a consistent, coherent user interface for our users may not be trivial. This is also true when we want communication between micro-frontends owned by different teams. Bear in mind that each micro-frontend should be decoupled from the others on the same page, otherwise we are breaking the principle of independent deployment.

New teams approaching this paradigm might be tempted to use a “global state manager” for sharing the state across micro-frontends, however this is considered an anti-pattern in distributed systems. We will deep dive into this and other anti-patterns later in the book.

In this case, we have a few options for notifying other micro-frontends that an event occurred. In general, we have to maintain the micro-frontends decoupled from each other, avoiding sharing a global state across them. We can inject an eventbus, a mechanism that allows decouple components to communicate with each other via events sent via a bus, in each micro-frontend and notify the event to every micro-frontend. If some of them are

interested in the event dispatched, they can listen and react to it (Figure 2-6).

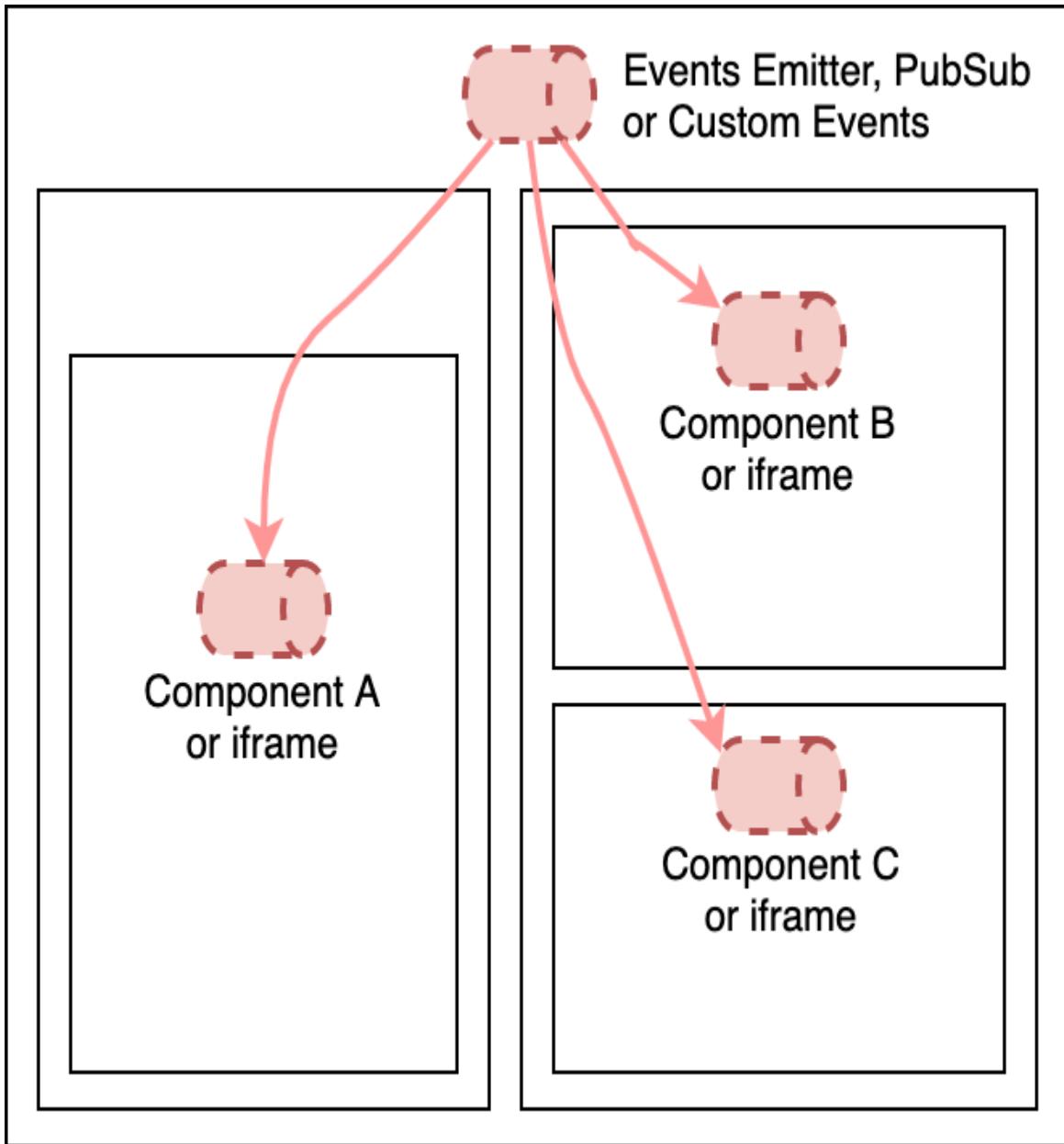


Figure 2-6. Event emitter and custom events diagram

To inject the eventbus, we need a micro-frontend container to instantiate the eventbus and inject it inside all of the page's micro-frontends, alternatively is having the application shell applying this logic and injecting or exposing the event bus to every micro-frontend.

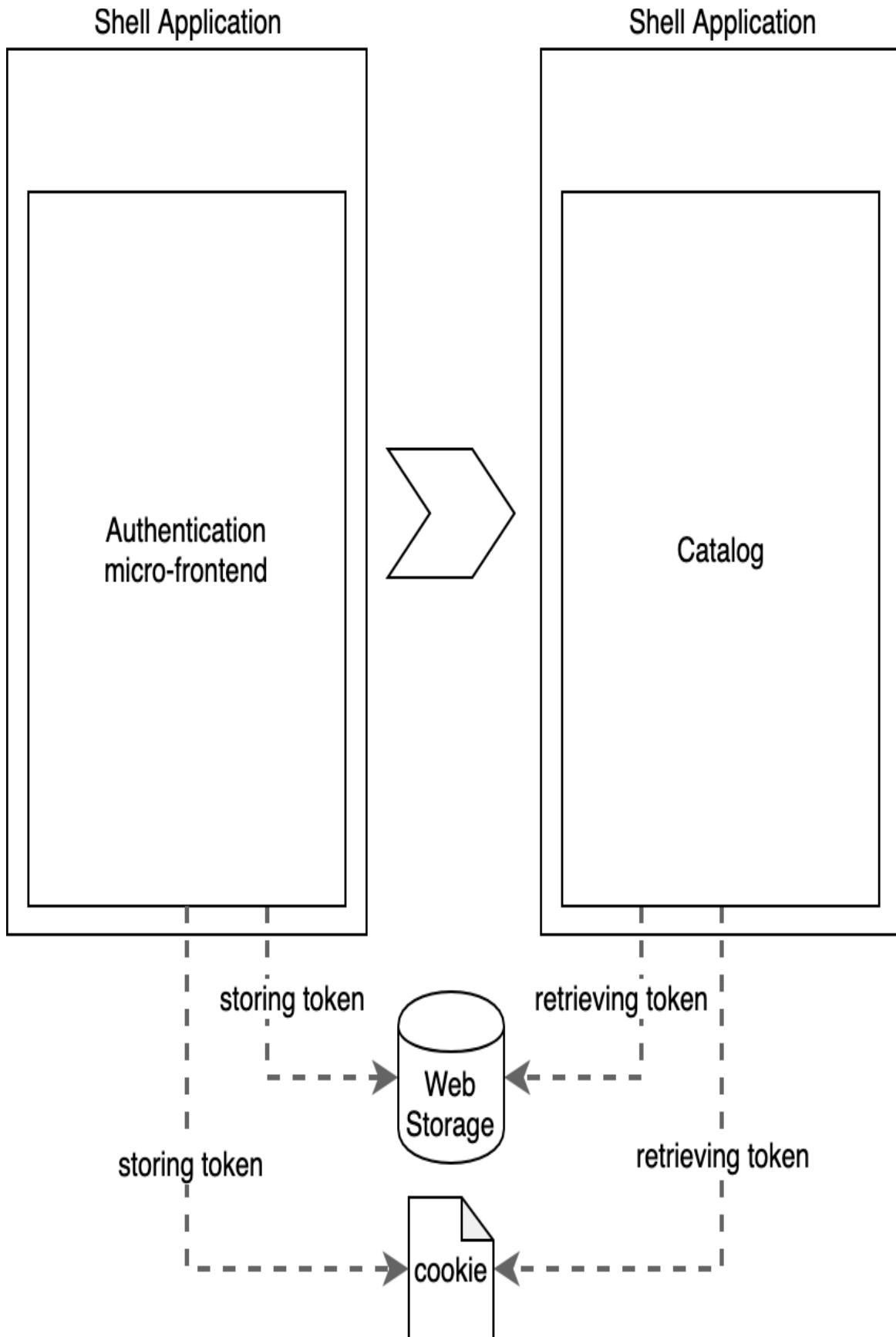
Another solution is to use **Custom Events**. These are normal events but with a custom body, in this way, we can define the string that identifies the event and an optional object custom for the event. Here's an example:

```
new CustomEvent('myCustomEvent', {  
    detail: {  
        someData: 12345  
    }  
});
```

The custom events should be dispatched via an object available to all the micro-frontends, such as the window object, the representation of a window in a browser. If you decide to implement your micro-frontends with iframes, using an eventbus would allow you to avoid challenges like which window object to use from inside the iframe, because each iframe has its own window object. No matter whether we have a horizontal or a vertical split of our micro-frontends, we need to decide how to pass data between views. Moreover, a custom event propagates to the window object traversing the elements tree expressed in the DOM. Imagine if a team accidentally stops the propagation of the custom event before reaching the DOM. This might cause more than a headache, so the recommendation is using an event emitter as the first choice.

Now, imagine we have one micro-frontend for signing in a user and another for authenticating the user on our platform. After being successfully authenticated, the sign-in micro-frontend has to pass a token to the authenticated area of our platform. How can we pass the token from one micro-frontend to another? We have several options.

We can use a web-storage-like session, local storage, or cookies ([Figure 2-7](#)). In this situation, we might use the local storage for storing and retrieving the token independently. The micro-frontend is loaded because the web storage is always available and accessible, as long as the micro-frontends live in the same subdomain.



*Figure 2-7. Sharing data between micro-frontends in different views*

For ephemeral data however, you could pass some them via query strings - for example, [www.acme.com/products/details?id=123](http://www.acme.com/products/details?id=123) the text after the question mark represents the query string, in this case the ID 123 of a specific product selected by the user - and retrieves the full details to display via an API ([Figure 2-8](#)). Remember, using query strings is not the most secure way to pass sensitive data, such as passwords and user IDs, however.

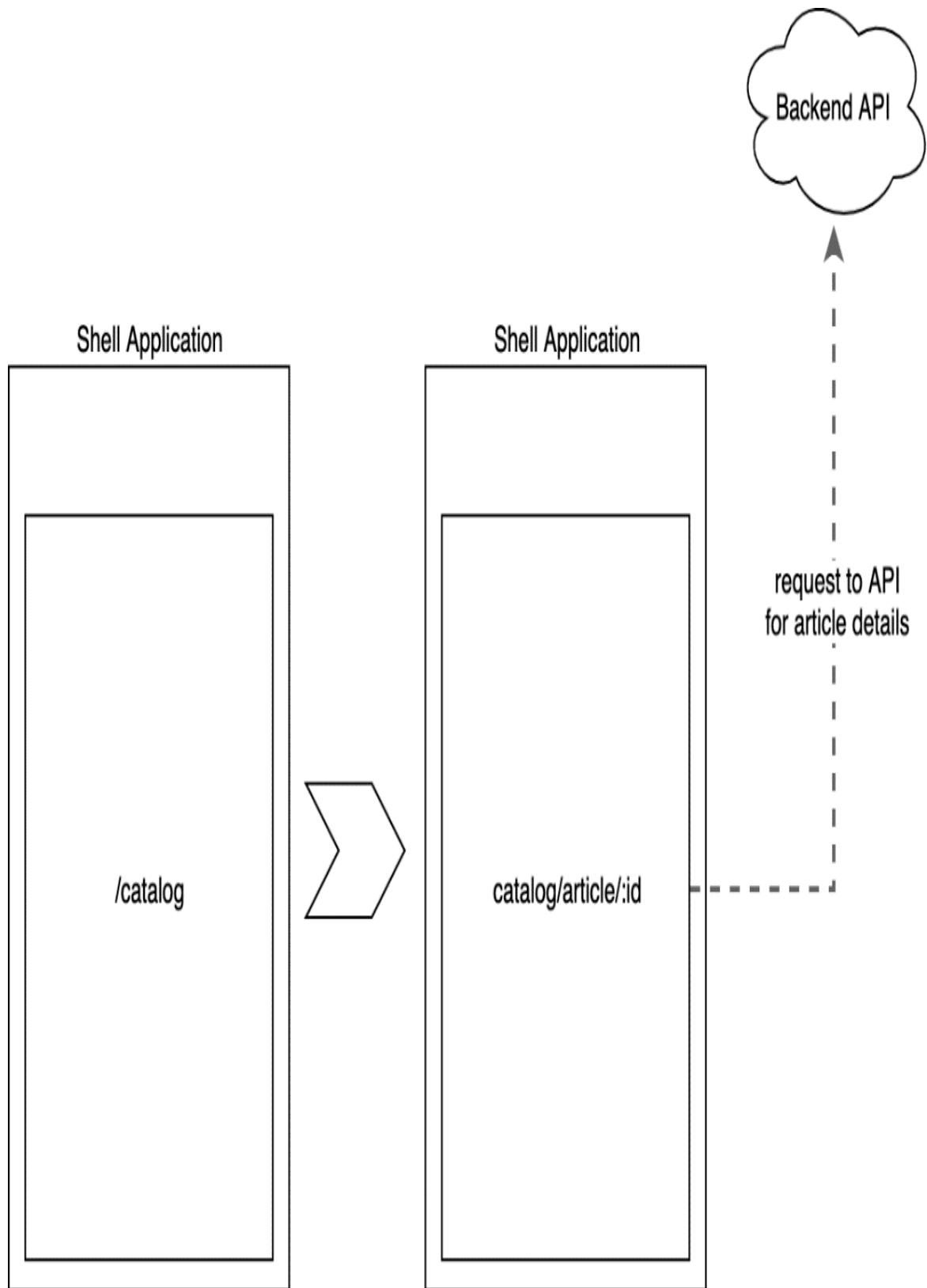


Figure 2-8. Micro-frontends communication via query strings or URL

To summarize, the micro-frontends decisions framework is composed of four key decisions: identifying, composing, routing, and communicating.

In [Table 2-2](#) you can find all the combinations available based on how you identify a micro-frontend.



$T$

$a$

$bl$

$e$

$2\text{-}$

$2.$

$M$

$ic$

$r$

$o\text{-}$

$fr$

$o$

$nt$

$e$

$n$

$d$

$s$

$d$

$e$

$ci$

$si$

$o$

$n$

$s$

$fr$

$a$

$m$

$e$

$w$

$o$

$rk$

$s$

$u$

$m$

$m$

a  
ry

Micro-frontends definition	Composition	Routing	Communication
Horizontal	Client side Server side Edge side	Client side Server side Edge side	Event emitter Custom events Web storage Query strings
Vertical	Client side Server side	Client side Server side Edge side	Web storage Query strings

## Micro-Frontends in Practice

Although micro-frontends are a fairly new approach in the frontend architecture ecosystem, they have been used for a few years at medium and large organizations. and many well-known companies have made micro-frontends their main system for scaling their business to the next level.

### Zalando

The first one worth mentioning is Zalando, a European fashion and e-commerce company. I attended a conference presentation made by their technical leads, and I have to admit I was very impressed by what they have

created and released open source, considering it was still early days and not many companies were talking about microapps.

More recently, Zalando has replaced the well-known OSS project called Tailor.js with [Interface Framework](#). Interface Framework is based on concepts similar to Tailor.js but is more focused on components and GraphQL than on Fragments.

## Formula One

[Formula 1 Digital Technology](#), responsible for the Formula1.com website and apps, aimed to increase web traffic, content consumption, and subscriptions. Faced with user expectations for fast loading times, they turned to micro-frontends to improve website performance and scalability. By migrating from a monolithic architecture to a micro-frontend-based system, Formula 1 achieved a 34% increase in subscriptions and sign-ups, a 26% reduction in platform costs, and significant improvements in Lighthouse performance scores (30% on web, 56% on mobile web). This transformation allowed for independent testing and deployment, faster delivery of changes, full integration with their design system, and more granular caching policies, all contributing to a faster and more engaging user experience. They adopted a test-and-learn iterative approach using the strangler fig pattern, gradually migrating functionality to the new micro-frontend architecture while extracting reusable components into separate systems for use by other domains like F1 TV and Fantasy.

## Dunelm

Dunelm is a well-known e-commerce company in the United Kingdom. They have embraced micro-frontends to allow multiple teams working together in a server-side rendering composition. They started with Next.js, but they are moving towards a simpler implementation with React.js due to the realization that Next.js was used mainly for routing and not much for all the features offered by the framework.

They worked on their implementation using a serverless approach fully in AWS, I highly encourage you to hear more about their story on [this episode](#) of “Micro-frontends in the trenches”.

## Netflix

In the Revenue and Growth department of Netflix, the engineers decided to embark in a micro-frontends approach creating an internal framework called [Lattice](#).

They discovered prevalent design patterns and architectures dispersed among different tools, with potential duplicating efforts among teams. Their goal was to streamline these tools in a manner that aligns with the scalability of the supported teams. The solution needed to embody the flexibility of a micro-frontend and the adaptability of a framework, enabling the stakeholders to enhance our tools effectively. They used Module Federation as well and this helped them to solve many challenges like dependencies management and runtime load of micro-frontends.

## PayPal

If you are a PayPal user and you log into the web application, you are interacting with a micro-frontends architecture. Thanks to this approach, they have shifted their mindset on how to build their web application. Moreover, they have started to share their approaches at scale. We have to remember that to build the web interface, multiple teams work together to generate one of the best payment experiences out there. Finally, PayPal team members started to contribute to the community by sharing what they had learned while building their application. I recommend watching this fantastic talk on [micro-frontends communication](#).

## BMW

BMW implemented a B2B portal that collects several applications under the same umbrella. The main rationale was reducing the cognitive load of their users when they have to perform an action across multiple portals.

Their approach is heavily based on maximum flexibility with a few constraints. In fact, any framework or JavaScript library can be loaded inside their Angular shell that uses Module Federation to manage the runtime loading of micro-frontends and the external dependencies. Here is a [great demo made by one of their engineers](#)

## SAP

Another company that is using iframes for its applications is SAP. SAP released *luigi framework*, a micro-frontends framework used for creating an enterprise application that interacts with SAP. Luigi works with Angular, React, Vue, and SAPUI—basically the most modern and well-adopted frontend frameworks, plus a well-known one, like SAPUI, for delivering applications interacting with SAP. Since enterprise applications are B2B solutions, where SEO and bandwidth are not a problem, having the ability to choose the hardware and software specifications where an application runs makes iframes adoption easy. If we think of the memory management provided by the iframes is out of the box, the decision to use them makes a lot of sense for that specific context.

## OpenTable

Another interesting approach is OpenTable's [Open Components](#) project, embraced by Skyscanner and other large organizations and released open source.

Open Components uses a really interesting approach to micro-frontends: a registry similar to the Docker registry gathers all the available components encapsulating the data and UI, exposing an HTML fragment that can then be encapsulated in any HTML template.

A project using this technique receives many benefits, such as the team's independence, the rapid composition of multiple pages by reusing components built by other teams, and the option of rendering a component on the server or on the client.

When I have spoken with people who work at OpenTable, they told me that this project allowed them to scale their teams around the world without creating a large communication overhead. For instance, using micro-frontends allowed them to smooth the process by repurposing parts developed in the United States for use in Australia—definitely a huge competitive advantage.

## DAZN

Last but not least is DAZN, a live and video-on-demand sports platform that uses a combination of SPAs and components orchestrated by a client-side agent called boot-strap.

DAZN's approach focuses on targeting not only the web but also multiple smart TVs, set-top boxes, and consoles.

Its approach is fully client side, with an orchestrator always available during the navigation of the video platform to load different SPAs at runtime when there is a change of business domain. Max Gallo, a distinguished engineer at DAZN, who followed the creation of the platform from day 1, shares his insights and the reason to embrace this approach in an episode of [“Micro-Frontends in the trenches”](#).

These are just some of the possibilities micro-frontends offer for scaling up our co-located and/or distributed teams. More and more companies are embracing this paradigm, including New Relic, Starbucks, Amazon, and Microsoft.

## Summary

In this chapter, we discovered the different high-level architectures for designing micro-frontends applications. We dove deep into the key decisions to make: *define, compose, orchestrate, and communicate*.

We also defined a heuristic to test micro-frontends boundaries after defining them.

Finally, we discovered that many organizations are already embracing this architecture in production, with successful software not merely available inside the browsers but also in other end uses, like desktop applications, consoles, and smart TVs.

It's fascinating how quickly this architecture has spread across the globe. In the next chapter, I will discuss how to technically develop micro-frontends, providing real examples you can use within your own projects.

# Chapter 3. Client-Side Rendering Micro-Frontends

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the fourth chapter of the final book. Please note that the GitHub repo will be made active later on.

If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the author at [\*building.microfrontends@gmail.com\*](mailto:building.microfrontends@gmail.com).

In this chapter, we will use the micro-frontend decisions framework to build a basic ecommerce website using Module Federation for a client-side rendering approach. As we’ve discussed, there isn’t a one-size-fits-all solution when it comes to architecture. The project’s goals, the organization’s structure and communications, and the technical skills available with the company are some of the factors we have to consider when we need to choose an approach.

After identifying the context we’ll operate in, we can use the micro-frontend decisions framework to help define the key pillars for our architecture’s technical direction. Instead of creating the same example in multiple frameworks, I’ll focus on helping you build the right mental model, which will allow you to master any micro-frontend framework rather than memorizing only one or two of the options available.

We will definitely explore some code, but I will stress the importance of understanding *why* a decision is made. This way, despite the approach and framework you use in your next project, you will be able to decide what the right direction is, independent of how familiar you are with a specific micro-frontend framework.

Remember the old saying “Give a man a fish, and you feed him for a day. Teach a man to fish, and you feed him for a lifetime”? Let’s learn to fish.

## The Project

Our project is an internal t-shirts ecommerce website for an enterprise organization. The site is composed of several subdomains, including:

- Homepage
- Login
- Payment
- Catalog
- Account management
- Employee support
- FAQ

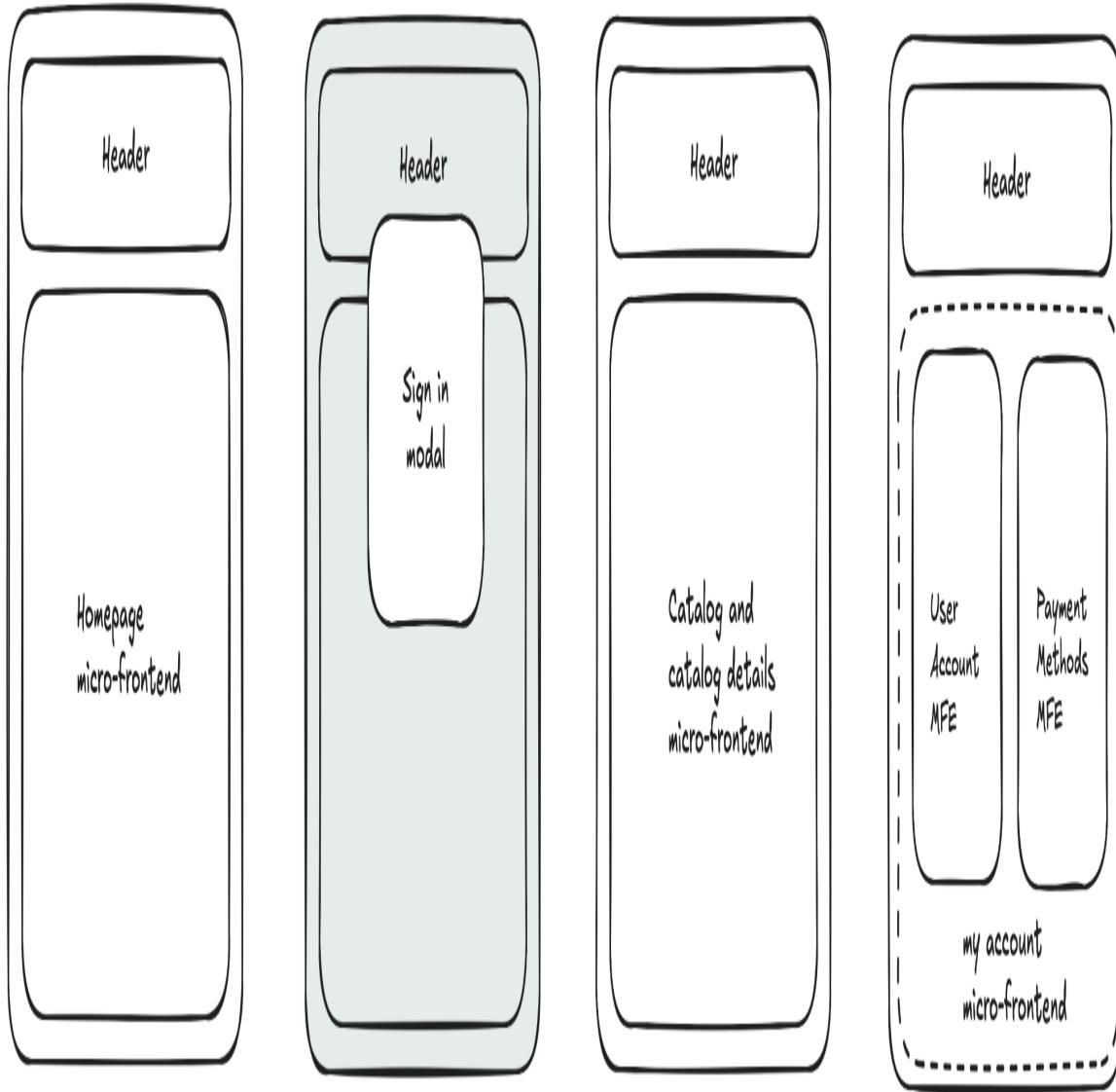
For our example in this chapter, we’ll use only some of them: homepage, authentication, catalog, and account management. The ecommerce site must have a consistent user interface so that users will have a cohesive experience while they shop for their favorite t-shirt. We will have several teams responsible for delivering this project. To hit the project deadline, the tech department decides to reuse an internal engine developed for its B2C ecommerce solutions. It’s a monolithic backend architecture that’s been battle-tested after several years of development and hardening in production environments. However, the tech department wants to move away from siloing the frontend and backend expertise, so it decides to use micro-

frontends and to set up independent teams responsible for a subdomain of the new ecommerce site. It will also use backend developers to rearchitect the backend using microservices and incorporate agility at the business and technical levels.

The next step is to assign the teams responsible for the different subdomains:

- *Team Sashimi* will be responsible for the homepage subdomains. Because this is an internal ecommerce site, the team will implement the sign-in form using the centralized authentication system available, which employees use to access every system inside the organization. It will also be responsible for the user authentication and personal details for the account details micro-frontend. One team member will be a full stack developer, and the rest will focus on the backend integration with Microsoft **Active Directory (AD)**.
- *Team Maki* will own the core domain—the catalog. It's the largest team and will be responsible for the main user experience. The team will be split between frontend and backend developers.
- *Team Nigiri* will cover the payments subdomain. It will integrate different payment methods, such as credit cards and PayPal.

The flow we'll implement is composed of three sections. [Figure 3-1](#) shows that a user is presented with the homepage, then she authenticates to see the full catalog. Catalog micro-frontends will be a vertical-split micro-frontend, while the my account section will be composed of two horizontal-split micro-frontends. We can have just one developer working on the authentication frontend because the heavier part of the implementation is on the backend. For the catalog, however, we want a richer user experience, so we will have a team with deep knowledge of frontend practices. Finally, because account management is an intersection of different subdomains, the two teams responsible for those subdomains will help develop this view.



*Figure 3-1. The t-shirts ecommerce sections: homepage, sign-in, catalog with product details, and account management*

Following the micro-frontend decisions framework and testing their assumptions with proof of concepts, the teams have decided to use:

#### *A hybrid approach for identifying micro-frontends*

Instead of using either a horizontal or vertical split for the whole project, the teams decided to use the right approach for each subdomain. A vertical split is more suitable for achieving the business requirements for authentication and the catalog considering those are assigned to the same team and they will be capable of managing the entire subdomain

without many external dependencies. While a horizontal split for the account management subdomain fulfills the business need to have multiple subdomains.

### *A client-side composition*

A client-side composition supports the requirements of the internal ecommerce site and is within the team's skill sets. This composition also allows future evolutions to other platforms, like a desktop application and even a progressive web application.

### *Client-side routing*

Once we decide to use client-side composition, the decisions framework helps us easily decide that the routing should happen on the client side as well. We also have to consider that there will be two types of routing: a global routing handled by the micro-frontend container (also called the application shell), which will be responsible for routing between micro-frontends, and a local routing inside the catalog subdomains, where the Maki team will develop micro-frontends with multiple views.

### *Communication between micro-frontends embracing decisions framework suggestions*

Again following the decisions framework suggestions, Team Sashimi will store the session token in a cookie and create a middleware to augment every request to an API decorating the header with a bearer token. On the other hand, the account management view will have two micro-frontends and could have more in the future, we want to maintain the teams and the artifacts independently from each other. As a result, we'll use an event emitter to communicate between micro-frontends and the application shell, defining up front the events triggered by every micro-frontend and the related payload.

As said at the beginning of this chapter, the technology chosen for the ecommerce site is Module Federation in conjunction with React. After several proofs of concept, the teams felt that Module Federation would

provide everything they needed to successfully release this project. The main reasons for embracing Module Federation over other solutions are:

### *Existing webpack knowledge*

Webpack is widely used inside the organization. Many developers have used this JavaScript bundler for other projects, so they don't have to learn a new framework. Module Federation fits nicely in their technology stack, considering it's just a plug-in of a well-known tool for the company.

### *Client-side composition*

With the micro-frontend composition based on the client side, Module Federation will provide a simple way to asynchronously load JavaScript bundles. It was developed initially for this specific use case and then extended to server-side rendering, so if in the future the requirements change, the teams will be able to change the micro-frontend implementation while maintaining Module Federation as stable assets for the evolution of their platform.

### *A seamless developer experience*

The teams have significant expertise with webpack. As well, the implementation in the automation pipeline and the local development tools remain the same, so it's a great way for the teams to be immediately productive.

Module Federation was chosen for specific reasons for this project. For other projects, Module Federation may or may not be the right choice. When we are designing an architecture, we have to think about the trade-offs before blindly selecting a technology. Analyze your team structures, developers' skills, the tech stack used in other projects by the company, and the project's business goals before deciding which micro-frontend architecture is suitable for your use case. After analyzing the context you're

working in, use the micro-frontend decisions framework to create a solid foundation for future decisions for your project.

## Module Federation 101

Before jumping into the technical implementation, we need to understand a few basic concepts to appreciate the reasoning behind some technical decisions. Module Federation allows a JavaScript application to dynamically load and run code from another bundle, micro-frontends are the perfect use case for this capability. Version 2.0 made a massive step forward compared to the previous version. In fact Module Federation 2.0 is available for webpack as well as Rollup, Vite, RSPack and more to come. The previous coupling to Webpack is gone and it works nicely for client and server side applications.

Module Federation provides two key concepts that we have to understand before working with it:

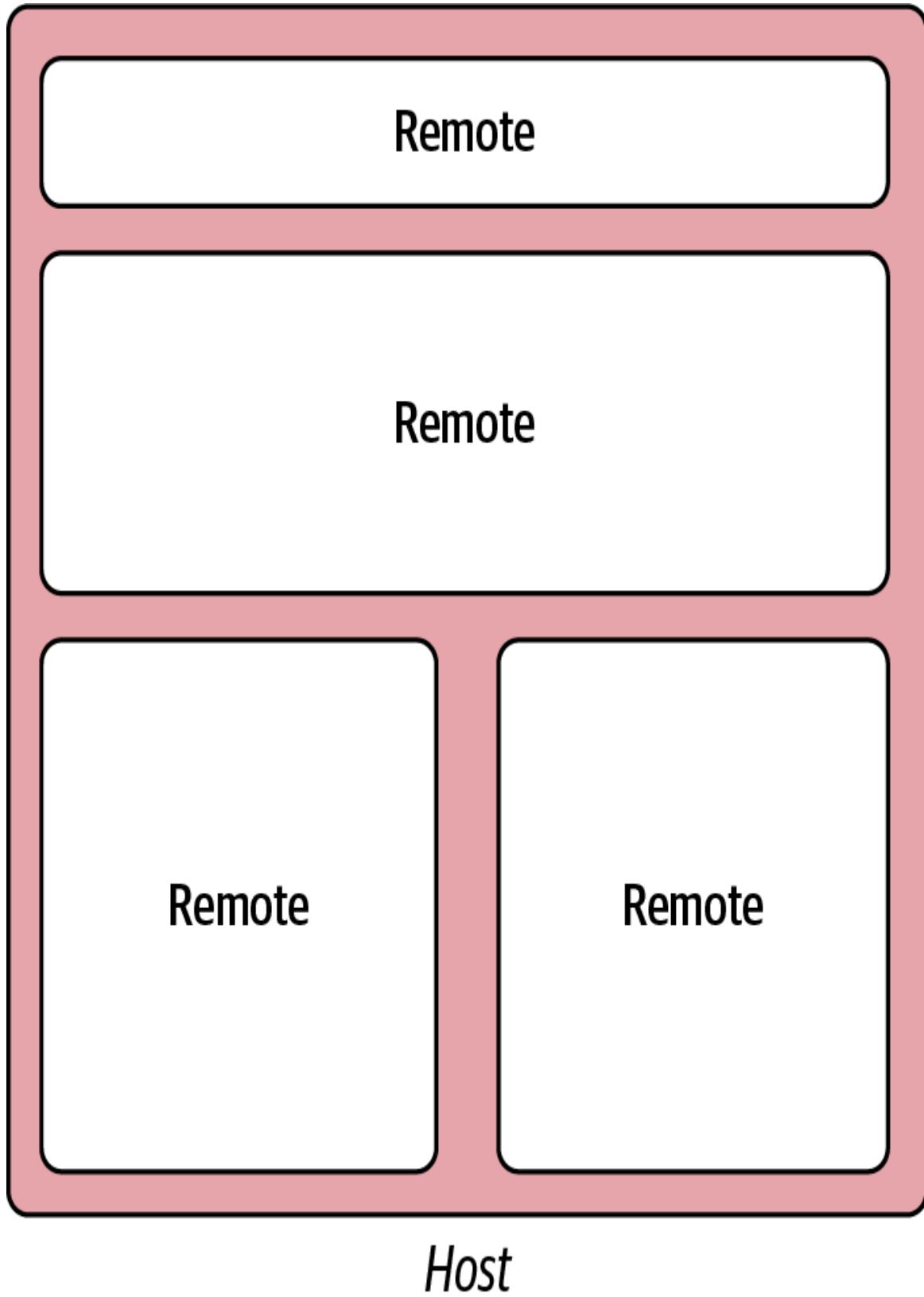
### *Host*

The container that loads shared libraries, micro-frontends, or components at runtime.

### *Remote*

The JavaScript bundle we want to load inside a host.

As we can see in [Figure 3-2](#), a host can load multiple remotes. In our case, the host represents the application shell, while a remote represents a micro-frontend.



*Figure 3-2. Module Federation is composed of two key elements: the host, which is responsible for loading some JavaScript bundles at runtime, and the remote, which is responsible for any type of*

*JavaScript bundles, such as shared libraries, micro-frontends, or even components*

With Module Federation, sharing can be bidirectional, allowing a remote to share parts or the whole bundle with a host and vice versa. However, bidirectional sharing can complicate your architecture very quickly. The best approach is sharing unidirectionally, so that a host never shares anything with remotes. This makes debugging easier and will reduce the potential for domain leaks to the host, which could cause design coupling between hosts and remotes.

Leveraging this architecture allows us to not only specify remotes in a selected bundler configuration but also load them using JavaScript in our code. For instance, we can fetch the routes from an API and generate a dynamic view of remotes based on the user's country or role. This is the right implementation for multi-environments systems like the ones we are used to working on a daily basis.

Because Module Federation has plug-ins for multiple bundlers, we can use other bundler capabilities to optimize the code in the best way for our project. For instance, Module Federation creates many JavaScript chunk files by default, but we may prefer a less chatty implementation for our remote, loading just two or three files. Let's assume the project bundler is webpack, we could use the *MinChunkSizePlugin* that forces webpack to slice the chunks with a minimum of kilobytes per file. We could also use the *DefinePlugin* to replace variables in your code with other values or expressions at compile time. Using this plug-in, we can easily create some logic to provide the right base path when we are testing code locally or when it's running on our development environments. Combined with other plug-ins available in the bundler ecosystem, Module Federation can be a powerful, suitable way to tweak your outputs for your context.

We should have enough Module Federation knowledge to dive deeply into the implementation details. The project we're exploring here shares many of the configurations available for Module Federation in a micro-frontend project. For more details, check out the [official documentation](#).

# Technical Implementation

Now that we've reviewed the context where the application will be developed and applied the decisions framework and selected the technical strategy, it's time to look at the implementation details. The t-shirts ecommerce repository is [available on GitHub](#), so you can review the entire project or clone and play with it. I intentionally developed the example without any server interaction so that you can run it locally without any external dependencies.

Just to recap, the application is composed of an application shell that is available during the entire user's session, loading different micro-frontends, such as the authentication, the catalog, and the account management micro-frontends. For the technology stack, the teams chose React with webpack and Module Federation, enabling every team to create independent micro-frontends. Using Module Federation, they can share common dependencies and load them only once during a user's session. This creates a seamless experience for users without compromising developer experience. Let's dive deeply into the key aspects of the main parts of this application.

## Project Structure

Before going ahead with the case study, I want to say a few words on the structure I created for the t-shirts ecommerce project. When you clone the repository, you will see multiple folders, as shown in [Figure 3-3](#).

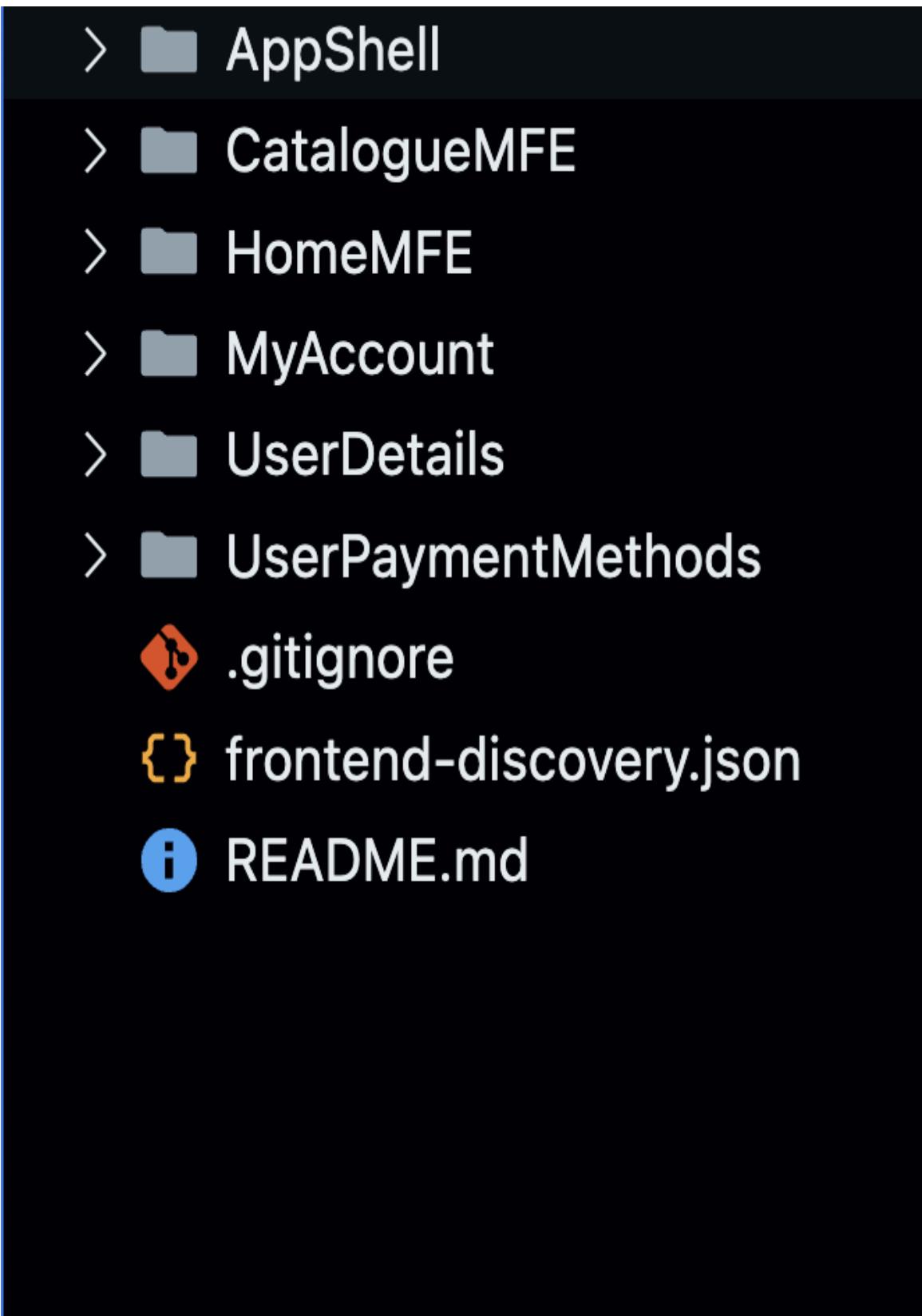
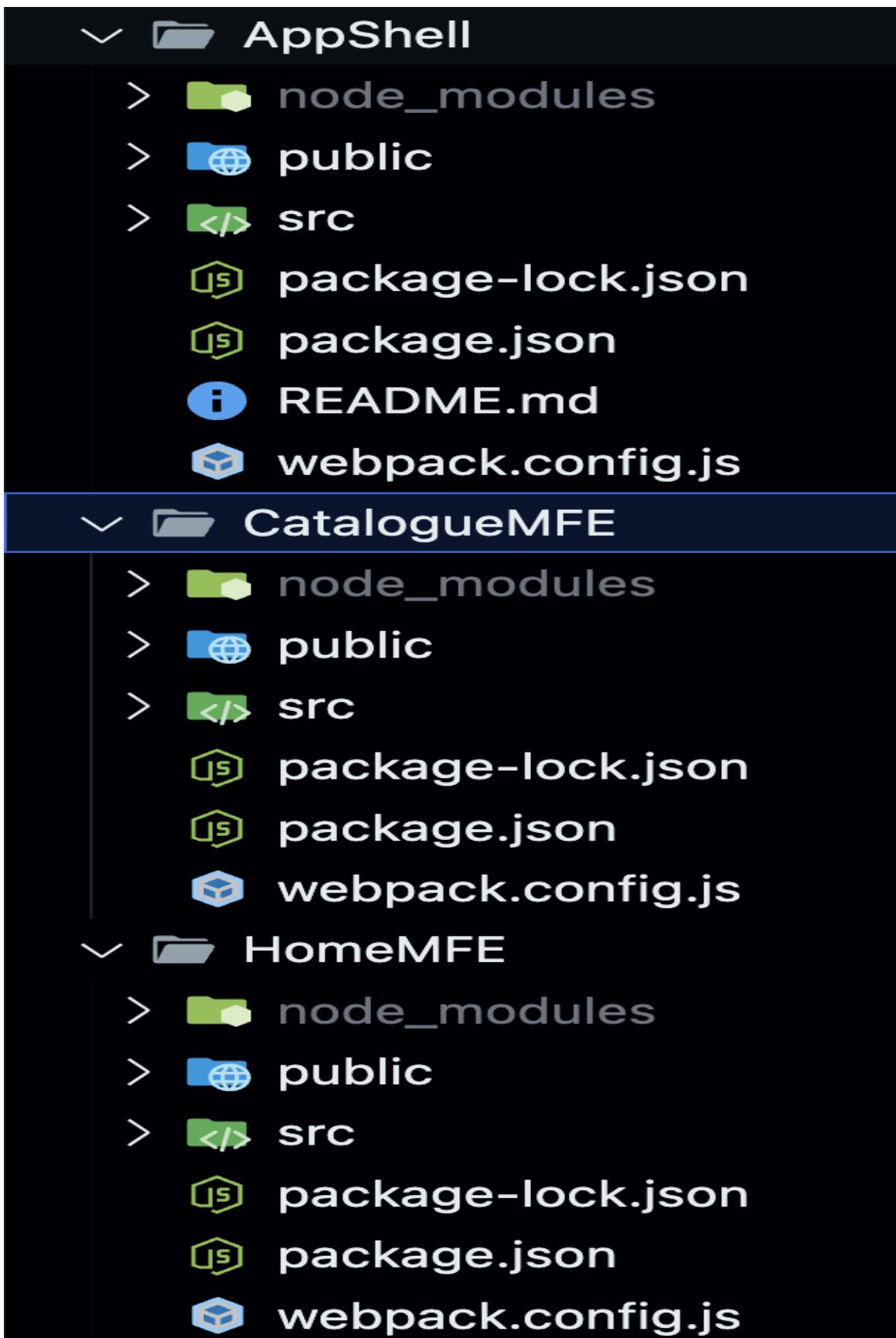


Figure 3-3. The t-shirts ecommerce project folder structure

Every folder represents an independent project. Because they are present in the same repository, this is a monorepo approach, but they could easily be extracted in a polyrepo approach.

All the folders have a similar structure, as you can see in [Figure 3-4](#).



*Figure 3-4. Micro-frontend structure*

## Application Shell

As previously described, the application shell remains present throughout the user's entire session. Since it is essential for orchestrating micro-frontends but does not belong to any specific business subdomain, the teams decided to assign its implementation to a newly formed team of principal engineers, called Sasazushi. Because building this part of the system requires minimal effort and ongoing maintenance is expected to be low—given that the application shell does not own any business domain logic—the principal engineers were chosen for this team in addition to their primary roles within their respective teams.

Sasazushi is responsible for:

- Preventing domain leakage in the application shell
- Implementing global routing between micro-frontends
- Ensuring micro-frontends are correctly mounted and unmounted
- Managing cross-domain dependencies, bundling them into one or multiple JavaScript chunks

Additionally, given that the team consists of principal engineers, it also oversees the overall performance of the system. This includes establishing recurring meetings with other teams to share optimization best practices and reviewing performance bottlenecks identified during assessments.

Now, let's analyze the webpack configuration. Since Module Federation is a webpack plug-in, we can import it just like any other JavaScript library:

```
const { ModuleFederationPlugin } = require('@module-federation/enhanced');
```

The most basic webpack configuration consists of an entry file, an output folder, and a mode that determines how the code is transpiled—typically set

to either *development* or *production*:

```
module.exports = {
  entry: "./src/index",
  mode: "development",
  output: {
    publicPath: "auto",
  },
// additional configuration
}
```

Beyond this basic setup, we usually define rules to support specific language features or frameworks. In this case, we are using *css-loader* to handle styles and Babel with the React preset to enable JSX support:

```
module: {
  rules: [
    {
      test: /\.css$/,
      use: ['style-loader', 'css-loader'],
    },
    {
      test: /\.jsx?$/,
      loader: 'babel-loader',
      exclude: /node_modules/,
      options: {
        presets: ['@babel/preset-react',
          '@babel/preset-env'],
        plugins: ['@babel/plugin-transform-runtime'],
      },
    },
  ],
},
```

However, the most critical aspect of our micro-frontend architecture is configuring Module Federation to load remote micro-frontends. Since the application shell serves as the container for these micro-frontends, it acts as the *host* in Module Federation terminology.

Many online examples demonstrate specifying remotes directly within the Module Federation plugin of the chosen bundler. However, in production environments, this approach is less common—I have rarely seen teams

implementing it this way. We will explore this point in more detail later in this chapter.

```
new ModuleFederationPlugin({
  name: 'shell',
  filename: 'remoteEntry.js',
  shared: {
    'react-router-dom': {
      singleton: true,
      requiredVersion: '6.21.3'
    },
    react: {
      singleton: true,
      requiredVersion: '18.2.0'
    },
    'react-dom': {
      singleton: true,
      requiredVersion: '18.2.0'
    }
  }
},
],
}
```

After defining the name, in this case *shell* for a host, we specify the libraries we want to share across all micro-frontends. When working with other micro-frontend frameworks, handling shared dependencies requires careful consideration.

A common approach is to create an independent repository for shared libraries and dependencies. Developers then establish an automated pipeline for building and deploying this shared code, along with governance policies covering update responsibilities, package size constraints, rollback procedures, and deployment strategies. However, with Module Federation, this process is significantly simplified. Instead of managing a separate repository, we only need to specify the shared dependencies in both the host and remotes—in our case, the application shell and all micro-frontends. Webpack and Module Federation will then generate multiple JavaScript files and ensure that each dependency is downloaded only once per user session, regardless of how many micro-frontends require it.

This may seem straightforward, but in practice, it's often more complex. The level of simplicity that Module Federation brings to optimizing micro-frontends is truly remarkable. Defining shared libraries in the Module Federation configuration is as simple as shown in the following code snippet:

```
//...

shared: {
  'react-router-dom': {
    singleton: true,
    requiredVersion: '6.21.3'
  },
  react: {
    singleton: true,
    requiredVersion: '18.2.0'
  },
  'react-dom': {
    singleton: true,
    requiredVersion: '18.2.0'
  }
}

//...
```

In the `shared` object, we specify the libraries that should be shared across micro-frontends. Module Federation also provides advanced APIs that allow us to fine-tune how these dependencies are handled.

One option is to ensure that a library is loaded only once by using the `singleton` property, as demonstrated in our example. We can also define a specific version of a library to maintain compatibility across micro-frontends. In our case, we are using a predefined set of React libraries, including `React`, `React-DOM`, and `React-Router-Dom`, to ensure consistency.

Additionally, we can configure a scope that determines where these libraries will be appended. We will explore this approach in more detail when examining some of the micro-frontends in this project.

## SHARED LIBRARY VERSIONS

When using Module Federation, if you don't specify a required version for a shared library, it defaults to the version already declared in your application's package.json file. This ensures consistency by running the application with a version it recognizes and trusts.

However, when multiple applications share the same module, things can become more complex. Module Federation evaluates the versions of the shared module across all applications. If another application uses a newer but compatible version, it will automatically use that version without issues. However, if a remote application relies on an older version, a warning will appear in the console. In such cases, Module Federation attempts to resolve the conflict by selecting the highest available version that remains compatible with all applications involved.

To gain more control over this process, additional configuration options like `requiredVersion` and `singleton` can be used. Defining a `requiredVersion` ensures that your application only accepts versions within a specified range, reducing the risk of compatibility issues. The `singleton` option guarantees that only one instance of the shared module is loaded, maintaining consistency across all micro-frontends.

By leveraging these options, you can prevent version conflicts and ensure seamless interoperability between your micro-frontends.

Module Federation allows dependencies to be loaded either synchronously or asynchronously. Synchronous loading can be enabled using the `eager` property in the plug-in configuration. However, the recommended approach is to load dependencies asynchronously. This prevents users from having to download all dependencies in a large upfront bundle, helping to maintain key performance metrics such as *Time to First Byte (TTFB)* and *Time to Interactive (TTI)*.

To enable asynchronous loading, we need to split the application's initialization into multiple files. In our setup, the application shell is divided into three main files: `index.js`, `bootstrap.js`, and `app.js`.

The `index.js` file serves as the entry point of our application and contains just a single line of code:

```
import("./bootstrap");
```

The `bootstrap.js` file is responsible for instantiating the application shell and mounting the React application inside a `<div>` element called "root", which is present in the HTML template:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import './setupFetch';

const root =
ReactDOM.createRoot(document.getElementById('root'));
root.render(
<React.StrictMode>
  <App />
</React.StrictMode>,
);
```

Let's take a moment to discuss the `setupFetch` module found in the Application Shell bootstrap. In a micro-frontend architecture, authentication and authorization can be managed in different ways. One common approach is to implement a middleware in the application shell that automatically appends a JWT token to every fetch request made by the micro-frontends. This centralizes authentication, ensuring a consistent and standardized process across all API calls.

Here's an example of how this might be implemented:

```
const originalFetch = window.fetch;

window.fetch = async (input, init = {}) => {
```

```
const jwtToken = 'your-jwt-token-here';
const headers = new Headers(init.headers || {});
headers.append('Authorization', `Bearer ${jwtToken}`);

const modifiedInit = {
  ...init,
  headers,
};

return originalFetch(input, modifiedInit);
};
```

With this middleware in place, every outgoing request from any micro-frontend is automatically enriched with the correct Authorization header containing the JWT token. This approach works particularly well in horizontally split micro-frontend architectures, where the shell is responsible for orchestrating shared logic and managing cross-cutting concerns like authentication.

However, this method isn't without its trade-offs. In a vertically split micro-frontend architecture—where each micro-frontend represents a distinct business domain and is often owned by different teams—authentication and authorization may need to be handled independently. In such cases, delegating authentication to each micro-frontend can be an alternative, when each domain needs to enforce its own security policies.

Additionally, this middleware should be considered a starting point rather than a production-ready solution. Real-world implementations need to account for complexities such as token expiration handling, error management, and compatibility with different UI frameworks or libraries.

If you choose to adopt this approach, be sure to test it thoroughly within your environment and refine it to handle these edge cases. Not all the frameworks or libraries are using fetch API as the base for their HTTP calls. Therefore make sure you properly identify the method or methods used before overriding.

Finally, `App.js` will contain two key elements. First, the `Main` component, which defines the basic structure of the user interface,

including the application's header. Second, React Router, a widely used routing solution for React applications, which will manage global navigation across micro-frontends.

Now, let's take a look at the `render` method in the `App.js` file:

```
render() {
  const { routes, isLoading } = this.state;

  return (
    <Router>
      <div>
        <Header />
        <main style={{ maxWidth: '1200px', margin: '0 auto',
padding: '2rem' }}>
          {isLoading ? (
            <Loading />
          ) : (
            <Routes>
              {routes.map((route, index) => (
                <Route
                  key={index}
                  path={route.path}
                  element={<System request={route.request} />}
                />
              ))}
              <Route
                path="*"
                element={
                  <div style={{ textAlign: 'center', padding:
'2rem' }}>
                    <h2>Page not found</h2>
                  </div>
                }
              />
            </Routes>
          )}
        </main>
        <NotificationModal emitter={emitter} />
      </div>
    </Router>
  );
}
```

As you can see, all the routes are dynamically generated. In mid-to-large-size applications, it's highly unlikely that you'll define all remotes directly

in the Webpack configuration. The main reason is that most teams work with a multi-environment strategy, where the application is composed of different endpoints depending on the deployment environment.

Even when testing micro-frontends, having the flexibility to load different versions of micro-frontends—especially those your team doesn’t own—is crucial. Imagine your team has been developing a new feature for weeks. During final testing, you need to ensure that your code doesn’t conflict with work done by other teams. To achieve this, you might test the latest version of your micro-frontend in a development environment while pulling stable versions of other micro-frontends from staging or another pre-production environment.

Another key consideration is page routing within a micro-frontend architecture. Let’s start with global routing. When using an application shell, routing happens on the client side, making the shell responsible for navigating between micro-frontends. Typically, URLs have multiple levels. For example, in

`https://www.mysite.com/catalog/product/123`, the application shell should handle only the first-level routes (e.g., `https://www.mysite.com/catalog`). Once the user lands on that route, one or more micro-frontends take over to manage deeper navigation.

This approach is critical because it ensures that the application shell remains as domain-agnostic as possible. By dynamically loading only the first-level URLs and delegating the rest of the navigation to individual micro-frontends, we prevent unnecessary dependencies between teams. This accelerates the release process, as each team remains autonomous—one of the primary benefits of adopting a micro-frontend architecture.

To load routes dynamically, we invoke a function called `initializeMFEs` when the component is mounted. This function is responsible for fetching the route configuration and registering the micro-frontends with Module Federation:

```
initializeMFEs = async () => {
  try {
```

```

        await init({
            name: 'shell',
        });

        const response = await
fetch('http://127.0.0.1:8080/frontend-discovery.json');
        const data = await response.json();

        const remotes = [];
        const routeConfigs = [];

        for (const [_, configs] of
Object.entries(data.microFrontends)) {
            const config = configs[0];
            const { name, alias, exposed, route, routes } =
config.extras;
            remotes.push({
                name,
                alias,
                entry: config.url
            });
            routeConfigs.push({
                path: route,
                request: `${name}/${exposed}`
            });
            if (routes) {
                routes.forEach(dynamicRoute => {
                    routeConfigs.push({
                        path: dynamicRoute,
                        request: `${name}/${exposed}`
                    });
                });
            }
        }

        await registerRemotes(remotes);
        this.setState({
            routes: routeConfigs,
            isLoading: false
        });
    } catch (error) {
        console.error('Failed to initialize or load micro-frontend
configuration:', error);
        this.setState({ isLoading: false });
    }
}

```

here are two key aspects of Module Federation that we need to consider in this method:

- 1. Initializing Module Federation:** We must trigger the init method with a configuration. The only mandatory field is the name, but this can be extended with additional details, such as shared library versions.
- 2. Registering Remotes at Runtime:** A remote must be registered inside the init method configuration or via the registerRemotes function to be loaded dynamically.

At the beginning of this example, we did not specify the remotes statically inside the Webpack Module Federation plugin. This is why we need to register them dynamically when initializing the application shell.

In Chapter 8, we'll explore how to use a discovery service to retrieve routes dynamically. For now, let's assume we're using a simple JSON configuration hosted within our infrastructure.

As we can see, the Route object consists of a path (representing the first-level URL) and the corresponding micro-frontend to load. To streamline this process, I've created a System function that reads the JSON configuration and dynamically loads a remote after registering it.

```
const System = ({ request }) => {
  if (!request) {
    return <h2>No system specified</h2>;
  }

  const MFE = lazy(() =>
    loadRemote(request)
      .then(module => ({
        default: module.default
      }))
  );

  return (
    <Suspense fallback={<div>Loading...</div>}>
      <MFE emitter={emitter} />
    </Suspense>
  );
}
```

```
) ;  
};
```

In this function, I'm using the `loadRemote` method from Module Federation to asynchronously load a micro-frontend. I then return a `Suspense` component with the loaded micro-frontends, injecting an emitter that will be needed for communication between micro-frontends later on.

When the user selects the new eCommerce area from the home page, the router will load the new micro-frontend in a manner similar to how we would lazy-load a regular React component in a single-page application (SPA). Module Federation will handle the process of importing the remote module for you.

## Home Micro-Frontend

The first micro-frontend to be loaded in the application shell is the homepage one. This micro-frontend doesn't have any micro-frontend-specific code, so if you navigate through the code, you won't spot any particular micro-frontend implementation. However, I decided to add a bit of spice to this domain.

One of the main benefits of micro-frontends is reducing external dependencies between teams. Imagine your team has a huge backlog of stories to deliver, and you receive a request from the application shell team to update React to version 18. With tight deadlines, you probably don't want another story to handle this update, as it could risk delaying more important features. Meanwhile, other teams may have already updated their own micro-frontends to the latest version in no time.

To avoid this risk, you can leverage another powerful feature of micro-frontends: the ability to manage different library versions in the same application.

When you look at the homepage's webpack configuration, you'll notice that it differs from the application shell's configuration:

```

new ModuleFederationPlugin({
    name: 'HomeMFE',
    filename: 'remoteEntry.js',
    exposes: {
        './MFE': './src/App'
    },
    shared: {
        'react-router-dom': {
            singleton: true,
            requiredVersion: '6.21.3'
        },
        'react17': {
            import: 'react',
            singleton: true,
            requiredVersion: '17.0.2'
        },
        'react17-dom': {
            import: 'react-dom',
            singleton: true,
            requiredVersion: '17.0.2'
        }
    }
}),

```

In this case, I have created a different scope (or container, in Module Federation terms) where this micro-frontend will append React and React-DOM libraries, avoiding a version clash with the default scope used for the rest of the libraries. In fact, React-Router-DOM is retrieved from the default scope because that version works just fine with both React 17 and React 18.

Module Federation 2.0 provides a way to inspect shared resources through the browser's developer tools. All shared resources are available in the `window.__FEDERATION__.__SHARE__`.

This object contains:

- All shared dependencies organized by micro-frontend name
- Version information for each shared resource
- The current state of shared modules

For example, you can inspect:

```
// See all shared resources for HomeMFE
window.__FEDERATION__.__SHARE__.HomeMFE
```

This is particularly useful for:

- Debugging shared dependencies
- Verifying correct version sharing
- Confirming proper isolation of React versions (e.g., React 17 vs. 18)
- Understanding how Module Federation manages shared resources at runtime

Here's an example of what you might see:

```
window.__FEDERATION__.__SHARE__.HomeMFE
{
  'react17': { /* React 17 instance details */ },
  'react17-dom': { /* ReactDOM 17 instance details */ }
}

window.__FEDERATION__.__SHARE__.UserPaymentsMFE
{
  'react': { /* React 18 instance details */ },
  'react-dom': { /* ReactDOM 18 instance details */ }
}
```

I think it's a very handy way to verify your configurations without delving too deep into the internals of Module Federation!

### NOTE

Remember, the ability to isolate libraries is also available with other approaches like using scopes with import maps or SystemJS. This capability is great because it is spread across multiple libraries leveraging web standards like Native Federation or Single SPA, for example.

Although you can handle multiple library versions in the same application, I urge you not to optimize for this approach. When I was building this demo, I encountered several incompatibilities across libraries, and it wasn't a fun experience at all. Moreover, designing an application that downloads more code than users need is far from being user-centric.

There are situations, like the one I described, that buy you time to update the libraries, but don't abuse this feature.

## Catalog Micro-Frontend

The catalog domain is probably the most complex and largest of all the micro-frontends. It's the reason users are going to the website, so it has to not only be simple to use but also provide all the information the user is looking for. Team Maki is responsible for this micro-frontend, and its goal is to implement multiple views so users can discover what's available in the catalog and get the details of each product. In the future, the team may have to add new functionalities, such as sharing product images taken by a buyer or adding a review score with comments.

The team will implement all these features and prepare the codebase in a modular fashion, such that, in the future, it will be easy to hand over part of the domain to another team if needed. Strong encapsulation and solid modularity will allow Team Maki to easily decouple part of the domain and collaborate with other teams to provide a great user experience.

This vertical-split micro-frontend's peculiarity is that we have to handle multiple views inside the same micro-frontend, a sort of SPA specific to the catalog domain. This shouldn't preclude the possibility of adding shared or domain-specific components, such as a personalized products component, implemented by other teams inside this domain. Although the application shell is responsible for the global routing, for this micro-frontend we have to implement local routing (that is, a routing implemented at a micro-frontend level) that works in conjunction with the global one. In our example, the local routing doesn't differ much from the global routing, as shown in the following code snippet:

```

class App extends React.Component {
  render() {

    return (
      <>
        <div>
        //...
        </div>
        <Routes>
          <Route path=":id" element={<ProductDetails />} />
          <Route path="/" element={<ProductList />} />
        </Routes>
      </>
    ) ;
  }
}

```

Using the React Router library, we initially retrieve the first level of URL depth. We then incrementally append the product ID to the depth level when a user selects a specific product. From the second level onward, the structure and management are usually fully handled inside the micro-frontend, so the domain can evolve autonomously, eliminating the need to coordinate its enhancements with other teams. This also prevents domains from creating overlapping first-level URLs because only one team is responsible for the global routing.

For this reason, I created a wrapper function that shares the React Router main functions as properties for the component to load in a catalogue route:

```

export function withRouter(Component) {
  return function WithRouterWrapper(props) {
    const navigate = useNavigate();
    const location = useLocation();
    const params = useParams();

    return (
      <Component
        {...props}
        navigate={navigate}
        location={location}
        params={params}
      />
    );
  };
}

```

```
) ;  
}  
}
```

And in the Catalogue.js component I have wrapped it in the export:

```
export default withRouter(Catalogue);
```

This is the same approach you will see for other micro-frontends if they require to handle the routing mechanism implemented inside the application.

When we implement the details page, we can use the product ID to request from an API the data to display:

```
const ProductDetails = () => {  
  const navigate = useNavigate();  
  const { id } = useParams();  
  //...
```

In this way, we prepare our codebase for potential future splits without too much effort. Query strings are a good way to hand over ephemeral information to another view or even to another micro-frontend. Since this type of data is consumed in the flush by another part of the system and doesn't need to be stored for long, using query strings for passing them across the system is strongly recommended.

## Account Management Micro-Frontend

Team Nigiri, responsible for the payment subdomain, and Team Sashimi, responsible for the user's account, have to collaborate for the account management view, with part of the payment information and part of the user's details converging in the same view. Because these domains are assigned to different teams, we need a different approach from the other micro-frontends. Instead of a vertical-split micro-frontend architecture, we'll use a horizontal split to compose the final view and allow the two domains to communicate with each other for specific user interactions. To

achieve this, we'll need to create a new host, that I usually call meta shell, and two remotes. We know every remote is associated with a team, but what about the new host? Together, Teams Nigiri and Sashimi define the strategy for evaluating this common container of their subdomains. The new host has a clear implementation path that consists of loading two micro-frontends: a user's details and payment details

Because of that, Team Nigiri decides to take ownership of the new host and collaborate with Team Sashimi to define mechanisms to ensure the developer experience is as smooth as possible and the releases of the host don't cause any issues with Team Sashimi's work.

You may be asking yourself where the new host will technically be presented to the user. Module Federation allows us to nest multiple hosts, and there isn't a strong hierarchical structure. In fact, there is a very thin line between remote and host because a remote can expose some libraries used by the host and vice versa. Remember, we need to pay attention to this thin line: when it's crossed and we start to share dependencies bidirectionally, we risk creating unmaintainable code that offers more problems than benefits in the long run. My recommendation is to force a hierarchical relation between host and remote where the sharing is always unidirectional so that the host will never expose any module with its remotes. This simple but effective practice simplifies a micro-frontends architecture implementation, reducing the risk of potential bugs. Moreover, it improves application debugging, which reduces the coupling between modules and avoids creating a big ball of mud in which multiple modules depend on each other. When we reduce the coupling and external dependencies in this way, each team will have the power to make the right decisions for the project, taking into account that we sometimes have to compromise to achieve the organization's business goals.

On the technical side, we have to account for some small changes to handle the horizontal approach. First, we need a container for the two micro-frontends. We'll create a host called MyAccount, which will load the two remotes and have a similar configuration of Module Federation plugin like the other micro-frontends. The main difference is that my account has to be

a host, because it concerns the user's details and payment micro-frontends, but it also must be a remote for the application shell. To do this, we add the `exposes` objects, as seen in the following code snippet:

```
// additional code before

new ModuleFederationPlugin({
  name: 'MyAccountMFE',
  filename: 'remoteEntry.js',
  exposes: {
    './MFE': './src/MyAccount'
  },
  shared: {
    react: {
      singleton: true,
      requiredVersion: '18.2.0',
    },
    'react-dom': {
      singleton: true,
      requiredVersion: '18.2.0',
    }
  }
}

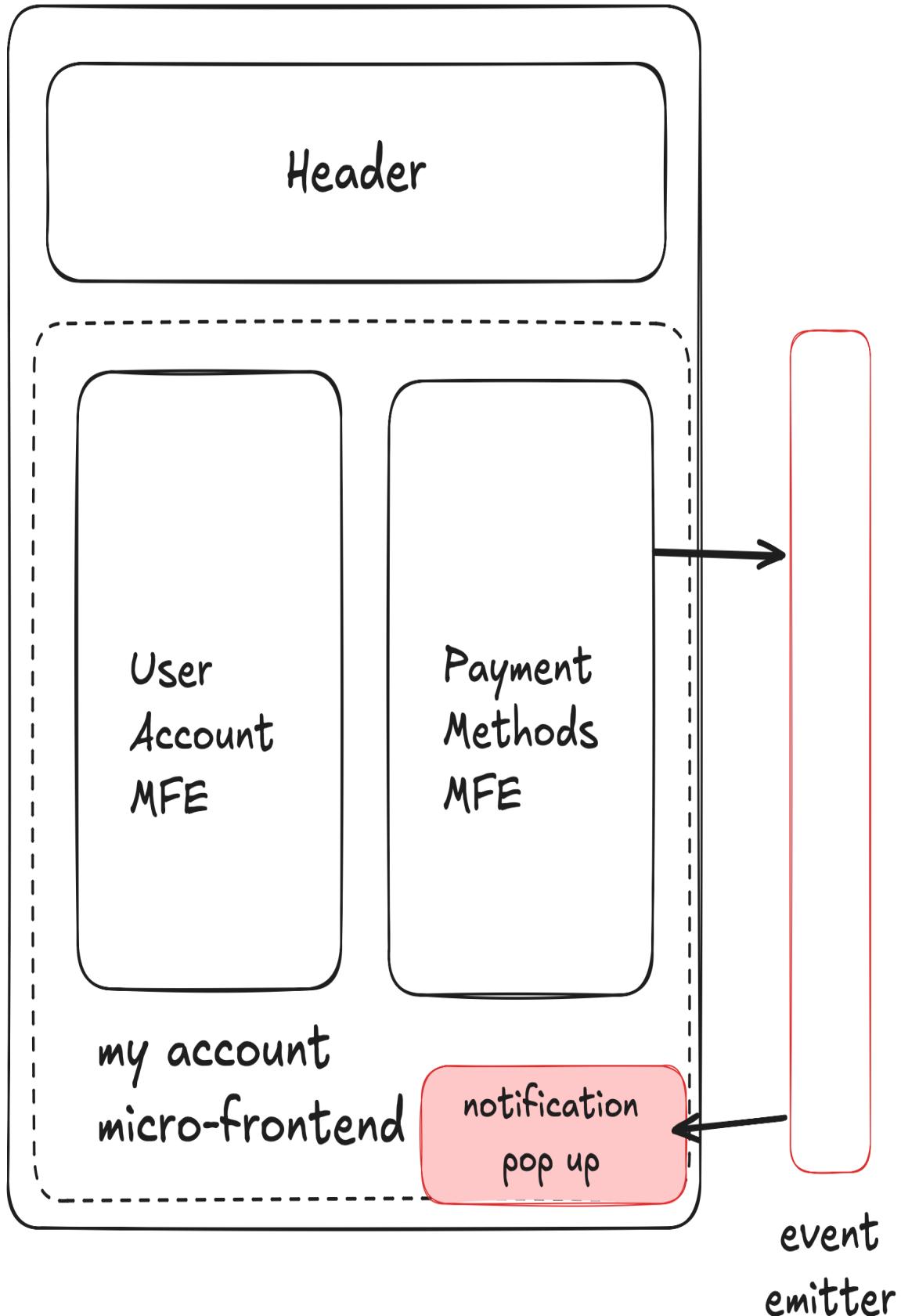
// additional code after
```

With just this change, we can have a host that is also a remote. However, we should avoid overusing this practice, as it can quickly lead to many small applications that represent individual components rather than entire business domains. Designing boundaries correctly is crucial. When we are unsure whether to create a new micro-frontend or incorporate a feature into an existing one, we should revisit the whiteboard and ensure our decision aligns with the core principles of micro-frontends.

A key aspect of this implementation is the communication between micro-frontends. As the Micro-Frontends Decision Framework states, we need to share information between micro-frontends without directly sharing code across different domains, as would be the case with a global state. An ideal approach is leveraging the publish/subscribe pattern to keep domains decoupled. When a modification occurs within a domain, the corresponding micro-frontend emits an event to notify the change. The application shell,

which includes a toast notification system accessible to all micro-frontends, listens for these events and displays notifications as needed. This ensures seamless communication without introducing unnecessary dependencies.

See [Figure 3-5](#) for reference.



*Figure 3-5. The event emitter shared between micro-frontends allows the communication across domain's boundaries maintaining their independence*

In the host, we create the same mechanism we did in the application shell but this time loading retrieving the user details and payment methods micro-frontends:

```
async componentDidMount() {
  try {

    await init({
      name: 'myAccount',
    });

    const response = await
fetch('http://127.0.0.1:8080/frontend-discovery.json');
    const data = await response.json();

    const userMfes = [
      data.microFrontends.UserDetailsMFE[0],
      data.microFrontends.UserPaymentMethodsMFE[0],
    ];

    await registerRemotes(
      userMfes.map(mfe => ({
        name: mfe.extras.name,
        entry: mfe.url,
      }))
    );
  }

  const requests = userMfes.map(mfe =>
`/${mfe.extras.alias}/${mfe.extras.exposed}`);
  console.log('Setting requests:', requests);
  this.setState({ mfeRequests: requests, isLoading: false });
} catch (err) {
  console.error('Error in loadMfes:', err);
  this.setState({ error: err.message, isLoading: false });
}
}
```

An optimization for this approach would be injecting the list of available micro-frontends into the My Account micro-frontend, retrieved by the application shell. However, let's consider the context. Making an additional call that is highly cacheable will not cause any performance issues.

Moreover, this is the only micro-frontend leveraging a meta-shell approach. Therefore, if the system evolves, we can consider making this change. Otherwise, we can maintain the separation between the application shell and the My Account data, keeping them independent and avoiding the need for team coordination.

After retrieving the information for the two micro-frontends, we render them inside the My Account view.

```
render() {
  const { mfeRequests, error, isLoading } = this.state;

  if (error) {
    return <div>Error loading account components: {error}</div>;
  }

  if (isLoading) {
    return <div>Loading...</div>;
  }

  return (
    <div style={{ padding: '20px' }}>
      <h1 style={{ marginBottom: '20px' }}>My Account</h1>
      <div style={{
        display: 'grid',
        gridTemplateColumns: 'repeat(auto-fit, minmax(300px,
1fr))',
        gap: '20px'
      }}>
        {mfeRequests.map(request => (
          <System key={request} request={request} emitter=
{this.props.emitter} />
        )));
      </div>
    </div>
  );
}
```

Now, let's dive into the two micro-frontends: User Details and User Payment Methods.

The User Details micro-frontend has the same configuration as the Home micro-frontend. Since it uses React 17, it leverages the same Module

Federation configuration and reuses the same dependencies if they are already loaded by the Home micro-frontend.

```
new ModuleFederationPlugin({
  name: 'UserDetailsMFE',
  filename: 'remoteEntry.js',
  exposes: {
    './MFE': './src/UserDetails'
  },
  shared: {
    'react17': {
      import: 'react',
      singleton: true,
      requiredVersion: '17.0.2'
    },
    'react17-dom': {
      import: 'react-dom',
      singleton: true,
      requiredVersion: '17.0.2'
    }
  }
}),
```

The most important thing to remember is to use the same name for dependencies. This ensures that when Module Federation loads them, it recognizes that they are already present and avoids downloading them again.

Looking at the User Payment Methods micro-frontend, we can see the implementation of micro-frontend communication. As explained in this chapter, the chosen approach is an event emitter. The main reason for this choice is its simplicity compared to reactive streams and its independence from the DOM hierarchy, unlike custom events. It should be your go-to approach for communication within micro-frontends.

I cannot stress enough the importance of avoiding global state managers across micro-frontends. State management should be handled inside each micro-frontend to keep them independent from one another. When a micro-frontend is loaded into the application shell, an emitter is injected to facilitate communication between micro-frontends or between a micro-frontend and the application shell.

```

        return (
            <Suspense fallback={<div>Loading...</div>}>
                <MFE emitter={emitter} />
            </Suspense>
        );
    
```

That emitter is simply a singleton that cannot be extended, thanks to the use of the `Object.freeze` method.

```

import { EventEmitter } from 'tseep';

class EventEmitterSingleton {
    constructor() {
        if (EventEmitterSingleton.instance) {
            return EventEmitterSingleton.instance;
        }

        this.emitter = new EventEmitter();
        EventEmitterSingleton.instance = this;
    }

    emit(event, data) {
        this.emitter.emit(event, data);
    }

    on(event, callback) {
        this.emitter.on(event, callback);
    }

    off(event, callback) {
        this.emitter.off(event, callback);
    }
}

const emitter = new EventEmitterSingleton();
Object.freeze(emitter);

export default emitter;

```

Now, from every micro-frontend, I can emit or listen to an event without relying on a specific position inside the DOM, as is the case with custom events. In fact, every time the user changes the default payment method, we emit an event called `notification` with the message that the toast notification should display.

```

handleMakeDefault = (id) => {
  const { emitter } = this.props;

  this.setState(prevState => ({
    paymentMethods: prevState.paymentMethods.map(method => ({
      ...method,
      isDefault: method.id === id
    }))
  }), () => {
    const defaultMethod = this.state.paymentMethods.find(m =>
m.id === id);
    if (emitter) {
      const notification = {
        type: 'success',
        title: 'Payment Method Updated',
        message: `${defaultMethod.brand} ending in
${defaultMethod.last4} is now your default payment method.`
      };
      console.log('notification msg', notification);

      emitter.emit('notification', notification);
    }
  });
};


```

Bear in mind that, in this case, the event is fairly simple, consisting of just a title, a type, and a message. We keep it generic to ensure it's useful across all domains, considering the communication will be with a shared toast notification present in the application shell. However, this doesn't mean that all your events should be structured in the same way.

On the frontend, designing events for an event emitter in a micro-frontends architecture requires a lightweight, flexible, and decoupled approach to communication. Events should be namespaced to avoid collisions (e.g., `"cart:itemAdded"`, `"auth:userLoggedIn"`), and the event payloads should be consistent and predictable to ensure smooth integration across micro-frontends. The event emitter should support both global and local event handling, where global events (e.g., authentication changes) are broadcast at the application shell level, while local events (e.g., UI state changes) are scoped to individual micro-frontends. Events should be designed to avoid

tight coupling—micro-frontends should listen for events rather than directly calling functions in other micro-frontends.

Then, we created a component in the application shell that handles the notification:

```
componentDidMount() {
  const { emitter } = this.props;
  emitter.on('notification', this.handleNotification);
}

componentWillUnmount() {
  const { emitter } = this.props;
  emitter.off('notification', this.handleNotification);
}

handleNotification = ({ type, title, message }) => {
  console.log('Notification received:', { type, title, message });
  this.setState({
    isOpen: true,
    type,
    title,
    message
  });
}
```

I hope this example helps you understand that Module Federation can be a great companion for your client-side rendering applications. One of the key benefits is the ease of sharing dependencies between different micro-frontends. In this example, we have a React 17 micro-frontend coexisting on the same view as a React 18 micro-frontend, with the My Account micro-frontend using React 18. Module Federation allows both versions of React to be loaded without conflict, ensuring that each micro-frontend can operate independently while sharing common resources efficiently.

Leveraging micro-frontends is not as complex as it may seem when you have a clear understanding of what needs to be expressed within your web or mobile application. The example shared demonstrates that the code structure doesn't differ significantly from building a standard client-side application with the framework of your choice. The key difference lies in

defining clear boundaries and responsibilities for each micro-frontend and establishing an efficient communication strategy between them. This is where the real shift happens—understanding how micro-frontends interact while maintaining independence. However, this challenge is addressed by the Micro-Frontends Decision Framework, which provides a structured approach to designing, organizing, and implementing micro-frontends. By applying this framework to every micro-frontends application, teams can make informed decisions, ensuring scalability, maintainability, and seamless collaboration across different teams.

The use of event emitters, custom events, or reactive streams should be intentional and minimal, ensuring each micro-frontend remains loosely coupled and independently deployable. Additionally, careful attention must be given to routing strategies, distinguishing between global routing (handled by the application shell) and local routing (managed within each micro-frontend). By following these principles and relying on the Micro-Frontends Decision Framework, teams can design modular, adaptable frontends that align with the distributed nature of modern applications while maintaining a seamless user experience.

## Project Evolution

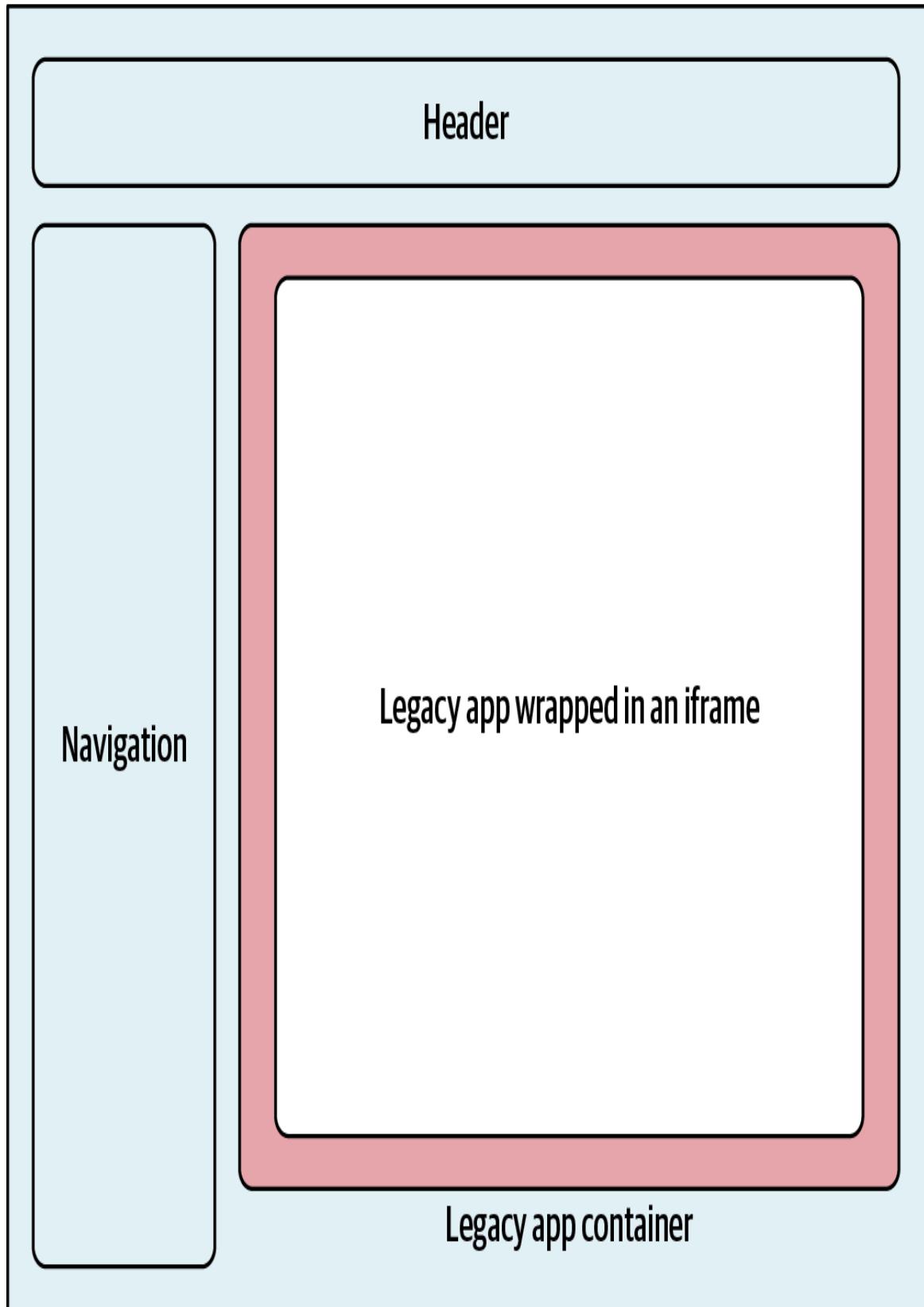
We don't want to create a project that works just for a short period of time. We want to create one that can evolve over time, eliminating the need to start from scratch in order to advance our organization's business goals. Let's explore, then, some potential ways this project could evolve and how we can create a coherent implementation across different subdomains.

### Embedding a Legacy Application

Imagine that we have to add a tool for customizing existing products, like socks, hoodies, and mugs, to our ecommerce project. The tool is a legacy application that was developed several years ago with an old version of Angular. Only one person from the team that developed this solution is still with the company, and she is keeping the lights on for this project, fixing

bugs and optimizing the codebase where possible. To reduce the feature's time to market, the business and the tech department decide to integrate the legacy tool with the existing micro-frontend architecture and ship them for a limited time. Later, a new team will take ownership of the project and revamp it to natively embrace micro-frontends. The application is well encapsulated, not requiring any particular information about the environment that is running, and we can pass the configuration needed to render a file to the configurator via a query string. Additionally, we want to minimize possible clashes with other parts of the codebase, such as with the application shell.

We can solve this problem by wrapping the legacy application inside an iframe, as in [Figure 3-6](#), which will prevent any possible clash with the existing micro-frontend system.



Application shell

*Figure 3-6. The application shell loads a micro-frontend that acts as an adapter between the new and old worlds. The legacy application is wrapped in an iframe to minimize the impact with the existing micro-frontend codebase.*

However, if we want to communicate with the legacy application and vice versa, such as by displaying errors across the entire interface instead of only in the iframe, we should create a communication bridge between the legacy application and the application shell in order to reuse the alerting system. We could directly integrate the application shell with the legacy application, but this would mean polluting the application shell codebase. We can implement a better strategy than that. Instead, we will apply an adapter pattern using a micro-frontend as a container for the iframe that contains the legacy application. The micro-frontend will be responsible for orchestrating the iframe using query strings and intercepting any messages from the legacy application, translating it into events emitted in the event bus.

### NOTE

The *adapter pattern* is a software design pattern (also known as wrapper) that allows an existing class's interface to be used as another interface. It is often used to make existing classes work with others without modifying their source code.

By using a micro-frontend as adapter, we can prepare our project for future evolutions. We can also reduce any refactoring in the application shell, first for integrating the legacy application and then for substituting with the new micro-frontend implementation. Within the application shell, we will maintain a business-unaware logic, since the communication will be translated to events via the event emitter. This process acts as an anticorruption layer between the inner and the outer systems. This pattern also comes in handy when we want to consolidate multiple applications under the same system and slowly but steadily replace every legacy application, with micro-frontends implementing a **strangler pattern**, which allows the micro-frontend application to live alongside the legacy ones.

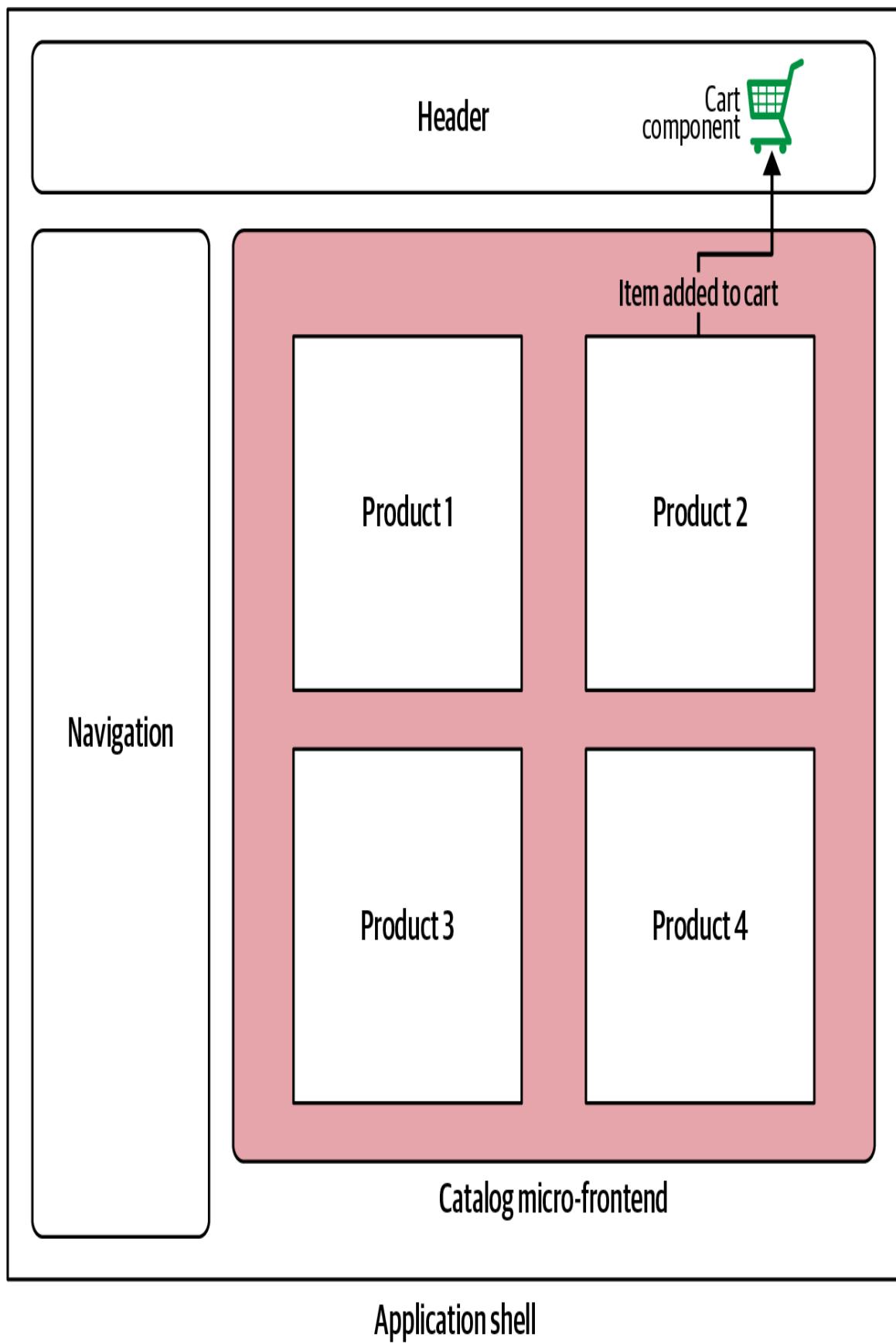
## Developing the Checkout Experience

The project is finally getting traction inside the organization and Team Nigiri is developing the checkout process. The product owner and the UX team have decided to place a cart inside the application shell's header. The cart should be shown only when a user is authenticated in the shop, so therefore that component should be visible only in certain views. When the cart button is clicked, the new checkout micro-frontend will guide the user to process the order correctly inside the system.

The cart component (see [Figure 3-7](#)) has different responsibilities:

- Hide and reveal the cart based on the area a user is navigating to
- Display the total number of items in the cart
- Start the checkout experience

Because the cart component will be present in the application shell, we have to create a logic for hiding it when a user is not authenticated and sharing it when they are. We could have the application shell orchestrate the component's visibility, but the checkout domain logic would leak into the application shell, which could pollute the codebase. Additionally, every time we want to change the visibility logic, we'd need to release a new application shell version. And because the checkout experience and the application shell are owned by different teams, creating such dependencies can only cause more troubles than benefits.



*Figure 3-7. The cart component is created and maintained by Team Nigiri. The component is loaded inside the application shell and uses an event emitter for listening for when an object should be added to the cart and showing the total number of elements the user has selected.*

A better solution is to ask the team responsible for the application shell to add the component, and the component itself will handle its own visibility based on a set of conditions, such as page URL. In this way, any logic change or improvement will happen inside the component and not leak these implementation details to the application shell. The application shell team will need to upgrade the library used for loading the component if they used a compile time implementation. In the case of Module Federation, they won't need to do anything else because the new component will be loaded at runtime.

To display the total number of items in the cart, we first need to add the product to the cart via an API exposed by the backend and then notify the cart component to update the value displayed in the interface. The best way to achieve this is by emitting an event via the event emitter instance. When the cart component receives the event, it will consume an API to retrieve the number of items currently in the cart.

Finally, when the user starts the checkout experience by clicking on the cart component, all that's required is changing the URL, notifying the application shell to move on to the checkout micro-frontend.

As you can see, investing a bit of time up front to think through the implementation of a simple element like a cart component can make a great difference in the long run. This cart component will maintain strong encapsulation despite living inside a different domain (the application shell). It will receive events from other parts of the system via the event emitter and route the user to the checkout experience. Following the principles of micro-frontends makes us reason in new ways, enhancing our developer experience and avoiding domain leaks in other application areas.

## Hosting a client-side rendering micro-frontends project

When hosting client-side rendering micro-frontends, I've observed that teams typically fall into three main approaches: using a single storage with a CDN, employing multiple storages with a unified CDN, or hosting micro-frontends within secure containers for highly regulated industries. When I mention storage, think about solutions like [AWS Simple Storage Service](#) for instance. Each approach has its own advantages and trade-offs, and the right choice often depends on the organization's structure and requirements.

The single storage with a CDN approach involves storing all micro-frontends in a centralized location, such as a cloud storage bucket, and serving them via a CDN. This setup is simple to implement and manage, as it centralizes deployment pipelines and ensures consistency across teams.

For example, imagine an e-commerce platform where the homepage, product pages, and checkout flow are separate micro-frontends. In this setup, all micro-frontends are stored in one bucket (e.g., AWS S3) and served globally through a CDN like CloudFront. This simplifies governance and makes it easy to reason about deployments while ensuring high performance through caching at edge locations. This approach works best for organizations with tightly integrated teams under unified governance.

The multiple storages with a unified CDN approach gives each team its own storage for micro-frontends while using a single CDN to deliver the content. This setup offers greater autonomy to individual teams while still leveraging the performance benefits of a shared delivery mechanism. For instance, consider a travel booking platform where separate teams manage flights, hotels, and car rentals as independent micro-frontends. Each team can maintain its own storage bucket for artifacts but rely on the same CDN for delivery. This allows teams to operate independently while maintaining consistency in user experience. However, this approach requires more coordination between teams to ensure compatibility and avoid duplication of effort in setting up and maintaining the infrastructure.

The secure container hosting approach is typically used by organizations in highly regulated industries like finance or healthcare, where accessing static

files via the public internet is not permitted. Micro-frontends are hosted in containers (e.g., Docker) within secure networks accessible only via VPNs or similar mechanisms. For example, a financial institution might use this method to host internal dashboards for account management or compliance reporting. While this ensures strict security controls, it introduces significant complexity in infrastructure management and wastes compute resources since containers are not optimized for serving static files. I strongly discourage using containers for this purpose unless absolutely necessary due to regulatory requirements.

For most organizations, the choice between single storage with a CDN and multiple storages with a unified CDN boils down to their structure and workflows. My recommendation is to adopt the single storage with a CDN approach whenever possible because it is easier to set up, simpler to manage, and reduces operational overhead. It provides clear governance and ensures consistency across teams while delivering excellent performance.

While the multiple storages with a unified CDN approach offers more flexibility for decentralized teams, it requires careful coordination to avoid duplication and maintain compatibility across artifacts. It's best suited for larger organizations or those with autonomous teams that need greater control over their micro-frontends.

I strongly discourage using containers to serve static files unless absolutely necessary (e.g., in highly regulated industries). Containers are designed for running applications rather than serving static assets, making them an inefficient and expensive choice for this purpose. They consume unnecessary compute resources and significantly increase infrastructure complexity without providing meaningful benefits compared to simpler solutions like CDNs or cloud storage.

By carefully evaluating these approaches based on your organizational structure and requirements, you can make an informed decision that balances simplicity, performance, and scalability while avoiding unnecessary complexity.

## Caching

One of the key optimizations for hosting client-side rendering micro-frontends is configuring different Time-to-Live (TTL) values for the CDN cache based on the rate of change of individual micro-frontends. This strategy allows you to balance performance and freshness effectively, ensuring that frequently updated micro-frontends remain current while less volatile ones benefit from longer caching durations.

Micro-frontends with a low rate of change can have higher TTL values. For example, an e-commerce platform might have a micro-frontend displaying 1950s Formula 1 standings or archived product details. These components rarely change, so setting a high TTL ensures they are cached for extended periods, reducing server load and improving response times for users.

On the other hand, micro-frontends with frequent updates, such as breaking news sections or live inventory displays, require lower TTL values to ensure users always see the latest information. For instance, the same e-commerce platform might have a micro-frontend showing real-time stock availability or promotional banners. These components need rapid updates, so shorter TTL values ensure the CDN fetches fresh content more often. In these cases, even 1 minute TTL can help reduce the traffic towards origin and improve the response time to the clients, so don't forget to set up the cache correctly.

## Summary

In this chapter, we have seen the micro-frontend decisions framework in action. Every team identified the right approach to achieve the requirements presented by the product teams. They maintained a decoupled approach, knowing that this path will guarantee independence and faster response time to any business shift. During this journey, we have also seen a technical implementation of a micro-frontend architecture using webpack and Module Federation.

This is one of the many approaches mentioned in Chapter 3. Every framework and technology will have its own implementation challenges; walking you through the reasoning behind certain decisions is far more valuable than evaluating each implementation. With this approach, you will have a mental model that allows you to move from one framework to another easily, just by following what you learned during this journey.

# Chapter 4. Server-Side Rendering Micro-Frontends

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the fifth chapter of the final book. Please note that the GitHub repo will be made active later on.

If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the author at [\*building.microfrontends@gmail.com\*](mailto:building.microfrontends@gmail.com).

You’ve been there. You open a website, and it takes forever to load. Or worse—content flashes, jumps around, and leaves you wondering if it’s even working. If you’re building a B2C platform, especially in e-commerce, you can’t afford that. Users bounce. SEO suffers. Trust evaporates.

When combined with micro-frontends, SSR unlocks greater scalability. However, blending these two approaches is not without its challenges.

In this chapter, we’ll explore when SSR is the right choice for your frontend, the unique hurdles it introduces in a micro-frontend landscape, and practical strategies for dividing, composing, and scaling your application. Along the way, we’ll look at real-world patterns, performance tips, and case studies that show how leading teams are building robust, modern web platforms with SSR and micro-frontends.

## When to Use Server-Side Rendering

Server-side rendering architectures shine in several key scenarios. First, they're excellent for content-heavy websites where search engine optimization (SEO) is crucial, such as news sites, blogs, and e-commerce platforms. Search engines can more effectively crawl and index server-rendered content, improving visibility and rankings. Second, SSR provides better performance for users with poor network conditions or less powerful devices, as it delivers HTML content immediately without waiting for JavaScript to execute. Lastly, SSR is valuable for applications requiring fast initial page loads to reduce bounce rates and improve user retention.

E-commerce platforms particularly benefit from SSR during high-traffic events like flash sales. The initial product information loads quickly, giving users immediate feedback while interactive elements hydrate progressively.

However, if you are building a Business-to-Business (B2B) application where the majority of the system is behind authentication, SSR should be the last architecture you consider.

## Scalability Challenges

Scaling SSR applications during traffic spikes presents unique challenges. Unlike static sites or client-side applications, SSR requires significant server resources to render HTML for each request. During events like Black Friday, an e-commerce platform might see a traffic increase 10-20x above normal levels, potentially overwhelming servers.

There are several techniques that we can employ to reduce the strain on our SSR systems as well as the APIs that these systems consume:

### *Serverless or prepositioning compute*

There are various ways to handle compute scalability, ranging from serverless solutions like AWS Lambda to pre-provisioning compute resources, such as keeping containers running in advance of predictable traffic spikes. The choice between these approaches, or a hybrid of both,

allows organizations to optimize for cost, performance, and operational control based on their specific application needs and traffic patterns.

### *Pre-computation*

Recommendation engines and content stores prepare data ahead of time, reducing processing during peak loads. For instance, during a flash sale, product pages can be pre-rendered and cached at the Content Delivery Network (CDN) level, while inventory updates are handled through smaller, targeted API calls.

We will cover a case study where these techniques were used on a well-known and large scale website later in this chapter.

## Dividing Micro-Frontends

Most successful SSR micro-frontend implementations use a coarse-grained horizontal split. The header and footer (rarely changing components) are often shared across the application, while the main content area is divided by page or domain.

Modern SSR frameworks like Next.js or Astro naturally support this division through their page-based structure. The page boundary—where content changes frequently—forms a natural split point between teams. For example, one team might manage product pages while another handles checkout flows.

Some companies attempt more granular splitting, where individual components on a page come from different micro-frontends. While this provides maximum team autonomy, it introduces significant challenges such as:

- Coordination complexity increases exponentially
- Performance can suffer from multiple service calls
- Debugging becomes more difficult across service boundaries

- Higher possibility on dependencies clashes
- Cache invalidation strategies grow more complex

## Composition Approaches

Overall, the main decision you need to make is how to compose micro-frontends. There are several approaches for composing server-rendered micro-frontends, each with its own trade-offs and ideal use cases:

- **HTML fragments** remain a classic approach, where backend services return HTML chunks that are assembled on the server before being sent to the client. This method is straightforward and works well for coarse-grained splits, but can become difficult to manage as the number of fragments and teams grows.
- **Framework-specific solutions** are increasingly popular. For example, Next.js supports a multi-zone approach, allowing different parts of your app to be handled by separate deployments, while Astro.js introduces the concept of Server Islands, letting you mix rendering strategies within a single page. Each framework brings its own conventions for composition, often influenced by the underlying UI technology or runtime.
- **Web components with shadow roots** provide a robust option for building micro-frontends. By encapsulating each micro-frontend as a web component using Shadow DOM, you ensure strong isolation of styles and logic, which helps prevent CSS and JavaScript conflicts between teams. Thanks to broad support for Declarative Shadow DOM in all major modern browsers (from 2023 onwards), this approach is now practical even with server-side rendering, making it easier to integrate micro-frontends built with different frameworks and to support progressive enhancement strategies.
- **Module federation** allows teams to independently build and deploy micro-frontends, sharing code and dependencies at runtime.

This enables true runtime composition, reducing duplication and making it easier to roll out updates across teams without redeploying the entire application.

- **Vertical split with independent infrastructure** is a simpler but effective strategy, especially for organizations with clear domain boundaries. Here, each major section of the application—often mapped to first-level URLs—is handled by a separate, independently deployed web application.

It's important to note that edge computing isn't always suitable, particularly for single-region applications with strict compliance requirements. The overhead of distributing workloads to edge locations may not justify the benefits, especially when your data remains centralized in another region. Additionally, compute power at the edge is often less than in-region resources, which can mean dealing with limitations in Node.js features or other runtime constraints.

## The main challenge

Composing server-side rendered micro-frontends is one of the most nuanced challenges in modern web architecture. Over the years, working with hundreds of teams, I've seen a wide variety of composition strategies —each with its own trade-offs in terms of performance, team autonomy, and operational complexity.

There are many options available for composing micro-frontends, ranging from simple server-side assembly of HTML fragments (transclusion) to more advanced techniques like runtime module federation, web components with shadow DOM, leveraging modern JavaScript frameworks capabilities like Next.js, Astro or Qwik, or orchestrating vertical splits via independent applications leveraging infrastructure. The choice often depends on your organization's structure, the technologies you use, and the level of independence you want to grant your teams. In the following sections, we'll analyze some of the most popular and effective composition patterns I've encountered in real-world projects. By understanding their strengths and

limitations, you'll be better equipped to select the right approach for your own architecture.

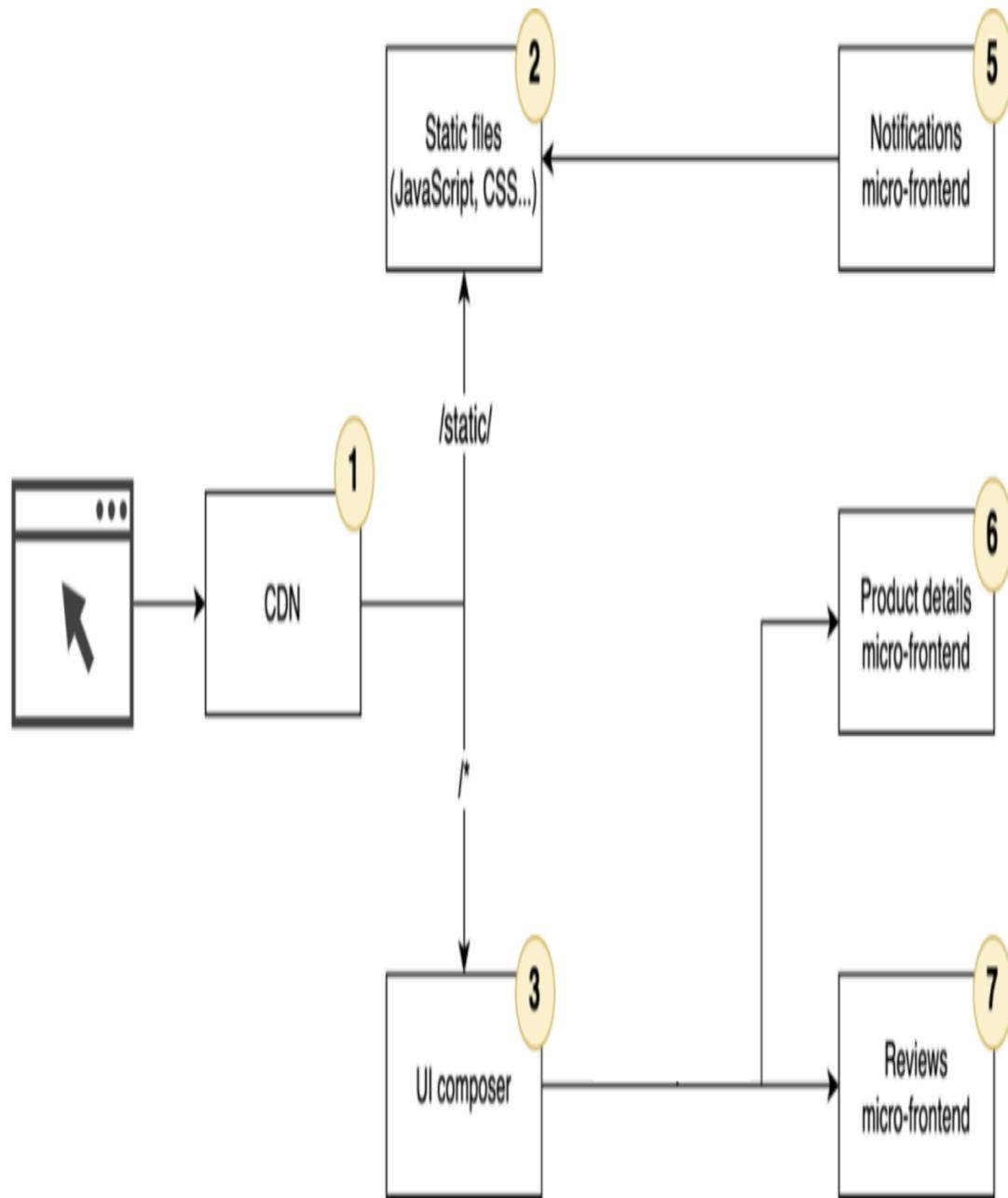
## HTML fragments

The first option we are going to explore is the HTML fragments approach. With this model, each team is responsible for a specific business capability—such as product catalog, shopping cart, or user reviews—and exposes its user interface (UI) as an HTML fragment via an HTTP endpoint. These fragments are not full standalone pages, but rather partial HTML snippets that represent a bounded context within the larger application.

To illustrate, consider an e-commerce platform. The product details page might be composed of several micro-frontends: one for product information, another for customer reviews, and a third for personalized recommendations. Each of these micro-frontends is developed by a separate team, possibly using different technologies or programming languages. For example, the product details micro-frontend could be built in Java, the reviews in Python, and the recommendations in .NET. Each team is responsible for delivering its fragment as HTML, ensuring it adheres to a contract—such as a specific placeholder or API—that the overall application expects.

Ideally, the frontend part should use the same library so that dependencies are loaded only once for the entire application, preserving user experience and performance. Libraries like HTMX provide a solid foundation for this multi-language approach.

The assembly of these fragments into a cohesive page is handled by a UI composer—a server-side orchestrator that leverages transclusion ([Figure 4-1](#)).



*Figure 4-1. An example of HTML fragments implementation*

The UI composer (no. 3) loads a base HTML template containing placeholders for each micro-frontend (no.2). At runtime, it fetches the relevant HTML fragments from the various upstream services (no.6 and 7) and injects them into the corresponding placeholders.

For instance, when a user navigates to a product page, the UI composer retrieves the product details fragment from the product team's service, the reviews fragment from the reviews team's service, and so on, and then stitches them together server-side before streaming the complete HTML page to the browser.

This approach brings several benefits. It allows teams to work autonomously, as each team can update, test, and deploy its micro-frontend independently without impacting others. The UI composer remains business-logic unaware—it simply knows where to fetch each fragment and where to place it, but not how each fragment is built or what business rules it implements. This separation of concerns enables robust testing, as teams can validate their fragments in isolation, confident that the composition mechanism will remain stable.

In the context of e-commerce, this model is particularly powerful. Imagine another Black Friday scenario: the promotions team can update the recommendations fragment to highlight flash sales, while the product team rolls out a new product details layout, all without needing to coordinate releases or risk breaking the overall page. If the reviews service experiences high load, the UI composer can gracefully degrade by showing a cached fragment or a simple placeholder, ensuring the rest of the page remains functional and performant.

This architecture is especially attractive for organizations with strong backend and full stack engineering teams, as it allows them to leverage their existing expertise in languages and frameworks beyond JavaScript, while still delivering a modern, scalable, and maintainable frontend experience.

Let's explore how the composition would look with this approach. When a user requests a page, the journey begins at the edge of the system, typically at a CDN. The CDN handles caching and security, routing the request to the appropriate backend if the content isn't already cached. If the requested page requires dynamic assembly, the CDN forwards the request to the system's entry point—a load balancer or gateway that directs traffic to the UI composer service.

The UI composer is the central orchestrator responsible for constructing the final HTML page:

```
fastify.get('/productdetails', async(request, reply) => {
  try{
    const catalogDetailspage = await
    transformTemplate(catalogTemplate)
    return reply
      .code(200)
      .header('content-type', 'text/html')
      .send(catalogDetailspage)
  } catch(err){
    request.log.error(createError('500', err))
    throw new Error(err)
  }
})
```

It starts by retrieving a static HTML template, which includes common dependencies in the head and placeholders or custom elements in the body indicating where each micro-frontend fragment should be inserted. Here's an example:

```
<!DOCTYPE html>
<html>
<head>
  <title>AWS micro-frontends</title>
  <script src="./static/nanoevents.js" type="module">
</script>
  <script src="./static/preact.min.js"></script>
  <script src="./static/htm.min.js"></script>
  <style>
    ...
  </style>
</head>
<body>
  ...
  <div id="noitificationscontainer">
    <script src="./static/notifications.js" defer></script>
  </div>
  <micro-frontend id="catalog" errorbehaviour="error"/>
  <micro-frontend id="review" errorbehaviour="hide" />
</body>
</html>
```

In the head tag, there are the common dependencies and in the body there are the placeholders or custom elements indicating where each micro-frontend fragment should be inserted.

Another interesting aspect is that the micro-frontend placeholder can also specify an error handling strategy. This signals to the UI composer how to manage error responses from a service. When you consider a view in a web application, it becomes clear that not all parts are always essential. For instance, in an e-commerce platform, the product detail page requires information about the product itself, but if, for any reason, the reviews are not available at that moment, the application can provide a degraded experience by displaying just the essential part—the product details. This flexibility allows a micro-frontend system to fail gracefully if a service returns an error, becomes unavailable, or if its response time is too high.

For example, if reviews are unavailable, the application can provide a degraded experience by displaying just the essential product details.

To fill these placeholders, the UI composer uses a service discovery mechanism—a key-value store or registry that maps logical names to the actual locations of the corresponding services. This enables teams to deploy and update their micro-frontends independently, ensuring rapid iteration within their bounded contexts. Once endpoints are resolved, the UI composer makes parallel requests to each micro-frontend service.

The discovery pattern will be covered in great detail in the next chapter. Each service generates its HTML fragment—often by querying its own data sources and applying its own rendering logic. These fragments are returned to the UI composer, which injects them into the appropriate placeholders within the template:

```
const transformTemplate = async (html) => {
  try{
    const root = parse(html);
    const mfeElements = getMfeElements(root);
    let mfeList = [];
    // generate VOs for MFEs available in a template
    if(mfeElements.length > 0) {
      mfeElements.forEach(element => {
```

```

        mfeList.push(
            getMfeVO(
                element.getAttribute("id"),
                element.getAttribute("errorbehaviour")
            )
        )
    });
} else {
    return ""
}
// Retrieves the micro-frontends endpoint from the
discovery service
mfeList = await getServices(mfeList);
// retrieve HTML fragments
const fragmentsResponses = await
Promise.allSettled(mfeList.map(element =>
element.loader(element.service)))
// analyse responses
const mfeToRender = fragmentsResponses.map((element,
index) => analyseMFEresponse(element,
mfeList[index].errorBehaviour))
// transclusion in the template
mfeElements.forEach((element, index) =>
element.replaceWith(mfeToRender[index]))
return root.toString();
} catch(error) {
    console.error("page generation failed", error)
}
}
}

```

HTML is easy to parse with JavaScript, and transclusion becomes a powerful mechanism for replacing placeholders with real HTML fragments that include HTML, styles, and JavaScript. The composition process is strictly mechanical: the UI composer does not interpret or alter the content of the fragments, preserving a clear separation of concerns and keeping the composition logic business-agnostic. In case of errors, the UI composer can apply logic to handle errors gracefully—either displaying a user error or hiding the affected micro-frontend:

```

const analyseMFEresponse = (response, behaviour) => {
    let html = "";

    if(response.status !== "fulfilled") {
        switch (behaviour) {

```

```
        case behaviour.ERROR:
            throw new Error()
        case behaviour.HIDE:
        default:
            html = ""
            break;
        }
    } else {
        html = response.value.body
    }

    return html;
}
```

After all fragments are assembled, the UI composer streams the completed HTML page back through the CDN to the user's browser. Streaming enables the system to deliver critical content as soon as it's available, improving perceived performance and reducing time-to-interactive. If any fragment is delayed or fails, the UI composer can insert a fallback or cached version, ensuring resilience and a seamless user experience.

Not all micro-frontends must be rendered on the server side; some can be fully client-side rendered, offering flexibility in how each part of the application is delivered and interacts with the user. For example, the notifications micro-frontend is loaded on the client and listens for events emitted by other micro-frontends, providing real-time feedback such as messages when an item is added to the cart. This event-driven communication pattern enables loose coupling between micro-frontends.

By thoughtfully choosing the rendering mode for each micro-frontend, teams can optimize both performance and user experience—leveraging CDN caching for static sections and dynamic updates for interactive features. This flexibility is a key strength of the micro-frontends architecture, empowering organizations to tailor their approach based on the needs of each business domain and user interaction pattern.

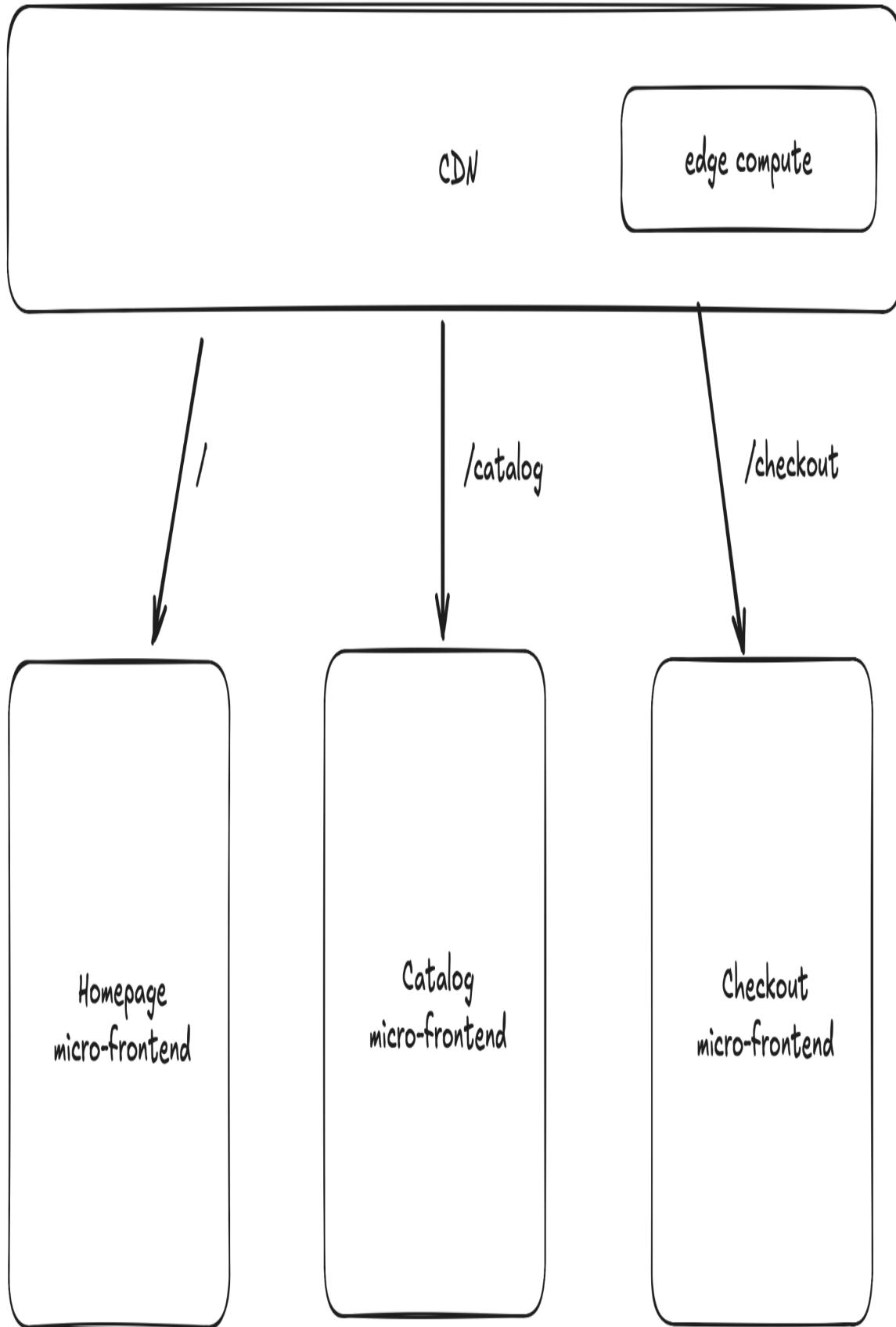
Finally, it is generally recommended to maintain a clear separation between the APIs that serve raw data and the services that perform frontend rendering. Backend APIs can evolve independently, optimized for data delivery and business logic, while frontend rendering services focus solely

on transforming that data into presentational HTML fragments. This approach was notably adopted by Zalando, German fashion e-commerce, in their early implementation with Tailor.js, and today is increasingly popular among organizations with strong backend or full stack engineering teams working in languages such as Java, Python, or .NET. By focusing on HTML fragments as the integration point, organizations can leverage their existing skills and infrastructure while reaping the benefits of modern, decoupled frontend development.

An example of this approach using AWS serverless services can be found in this [github repository](#). It's important to highlight that this approach is not AWS-centric; it can be implemented with any compute solution, from on-premises infrastructure to other cloud providers.

## Splitting by URL

This technique directs traffic to distinct, independently deployed services. Rather than composing multiple fragments on a single page, this model assigns entire top-level routes (such as /news, /standings, or /video) to specific applications, each maintained and deployed by a dedicated team ([Figure 4-2](#)).



*Figure 4-2. The CDN configuration helps you to associate different first level URL to different clusters that are taking care about server side render a specific subdomain of an system*

At the top, a CDN receives all incoming user requests. Based on the first segment of the URL—such as /, /catalog, or /checkout—the CDN routes each request directly to a dedicated micro-frontend application responsible for that part of the site. For example, requests to the root path are handled by the Homepage micro-frontend, /catalog by the Catalog micro-frontend, and /checkout by the Checkout micro-frontend.

Alternatively, you can use edge compute to intercept requests and rewrite them to specific endpoints. This approach is especially useful during migrations, as it keeps temporary redirection logic out of your main codebase. By isolating these rules at the edge, you ensure the micro-frontends logic is well encapsulated and doesn't require additional tests every time a micro-frontend changes.

This architecture offers remarkable simplicity and resilience. Each micro-frontend is a fully independent application, developed, deployed, and scaled by its own team. If one micro-frontend becomes unavailable the rest of the system continues to function without interruption. This isolation of concerns ensures that failures are contained and do not cascade across the platform, greatly improving overall system reliability.

Another key advantage is the flexibility in caching strategies. Because each micro-frontend is served independently, teams can tailor caching policies to the needs of their domain. For instance, the contact us micro-frontend, which may change infrequently, can be aggressively cached at the CDN to optimize performance and reduce backend load. In contrast, the homepage or catalog micro-frontend, which handles dynamic data, might require fresher content and thus use more conservative caching or even bypass caching entirely. This granularity allows the business to balance performance, cost, and data freshness for each area of the site.

Shared dependencies, such as utility libraries or design system components, are distributed through a private package repository. Ideally, these shared elements are implemented as web components, enabling teams to reuse and

update UI elements without rewriting each application. This approach ensures visual and functional consistency across micro-frontends, while allowing each team to evolve independently and adopt design system updates incrementally.

From a compute perspective, the deployment model is flexible and straightforward. Each micro-frontend can be hosted using containers or serverless functions, depending on the organization's infrastructure preferences and scalability requirements. This empowers teams to select the most suitable technology stack and deployment strategy for their needs.

Authentication is also streamlined in this model. By storing the authentication token in a cookie scoped to the base domain, every micro-frontend served under that domain has access to the token and can reuse it for secure API calls or session validation. This avoids the need for complex cross-application authentication mechanisms and ensures a seamless user experience as users navigate between different sections of the site.

## **F1.com website is powered by micro-frontends**

The split of the first-level URL was central to the transformation of F1.com, where the challenge was to modernize a legacy, monolithic web platform into a scalable, high-performing, and independently deployable system. The initial architecture, like many enterprise websites, was a classic three-tier monolith: a single application server generated all HTML, managed all business logic, and handled every user request. This setup, while straightforward, imposed significant constraints on scalability, release velocity, and team autonomy.

Any change—no matter how minor—required regression testing and redeployment of the entire application, leading to bottlenecks and increased risk.

The migration began by identifying clear domain boundaries within the site's structure. For F1.com, this meant recognizing that pages like the subscription journey, news listings, and statistics could be isolated and owned by different teams. Instead of rewriting the entire site in one go, the

team adopted an incremental “strangler fig” pattern: new features or domains were extracted one by one into standalone applications. A key enabler was the use of first-level URL routing at the edge or gateway layer. Incoming requests were inspected at the top-level path—such as /tv-proposition for subscription pages—and routed directly to the appropriate independent service. Pages still served by the monolith continued to be handled as before, while new or migrated pages were seamlessly served by their dedicated micro-frontend applications.

This routing strategy offers remarkable simplicity and speed for migration. There’s no need for a complex composition layer or runtime orchestration: each micro-frontend is a self-contained application, responsible for its own assets, dependencies, and rendering. Shared elements, such as the header, footer, or design system components, can be distributed as libraries and reused across applications, ensuring visual consistency without tightly coupling deployments.

According to their implementation results:

## **Business Impact**

- 34% increase in subscriptions and signups to F1 Unlocked and F1 TV (measured from January to September 2024)
- 26% reduction in platform costs by shifting workloads from Amazon EC2 to container-based solutions

These data are a fantastic testimony on how well this architecture could suit organizations to enhance their core metrics and improve business outputs. Despite this, there is more room for improvement and the most important thing is having an architecture that works for you more than the other way around. Formula One is on a good trajectory to further improve its future web applications.

If you are interested in learning more about this case study, [watch the talk here](#).

## Next.js multi-zones

Next.js multi-zones is another approach for building micro-frontends, especially for large-scale applications that need to be split into independently deployable sections. Sounds like micro-frontends doesn't it?

In this model, each “zone” is a standalone Next.js application responsible for a specific set of routes or features-like `/blog/*`, `/dashboard/*`, or the main site, while all zones are seamlessly merged under a single domain for the end user.

Navigation in a Multi-Zones setup is designed to feel smooth and unified for users. When moving between pages within the same zone, navigation is handled by Next.js, resulting in fast, soft navigations that don't require a full page reload. However, navigating from one zone to another (for example, from `/` to `/dashboard`) triggers a hard navigation, unloading resources from the current zone and loading those for the new zone.

While hard navigations are sometimes unavoidable in a multi-zone or micro-frontend architecture, there are several best practices you can adopt to minimize their impact and preserve a seamless user experience. Vercel recommends leveraging modern browser features like prefetching and prerendering to reduce perceived latency when crossing zones. By prefetching resources for likely destinations in advance, or even prerendering the next page in a hidden tab, you ensure that much of the content is already loaded and ready to display when the user initiates a hard navigation. The Speculation Rules API, for example, allows you to declaratively specify which links should be prefetched or prerendered, making transitions between zones noticeably faster.

It's also important to use plain `<a>` tags for links that cross zone boundaries, rather than the Next.js `<Link>` component. This prevents Next.js from attempting to soft-navigate to a route that belongs to a different application, and ensures the browser handles the transition as a full page load. For zones that are frequently visited together, consider grouping them within the same application to reduce the number of hard navigations required.

Finally, thoughtful routing configuration using rewrites or middleware can help optimize request handling and further streamline the user journey. By combining these techniques—prefetching, prerendering, correct link usage, and intelligent routing—you can greatly reduce the friction of hard navigations and deliver a user experience that feels fast and cohesive, even in complex multi-zone environments.

The real power of Multi-Zones comes from how rewrites are configured. Next.js enables you to set up rewrites in your `next.config.js` so that requests to certain paths are transparently proxied to the appropriate zone, even if that zone is running on a different server or port. This acts like a reverse proxy, avoiding CORS issues and allowing all the zones to appear as a single, cohesive application to the user. For example, you might rewrite `/dashboard/:path*` to `https://dashboard.example.com/:path*`, letting the dashboard team deploy and manage their app separately from the rest of the site.

Bear in mind that with this configuration, you can always create your own reverse proxy and configure it similarly to how you would in a multi-zone setup. So, if you decide to implement a multi-zone approach, switching from the built-in proxy to a custom reverse proxy requires minimal changes.

Middleware in Next.js adds another layer of flexibility. Middleware lets you run code before a request is completed, allowing you to rewrite, redirect, or even inject custom logic based on the incoming request. In a Multi-Zones architecture, middleware can be used to handle authentication, apply feature flags, or perform A/B testing, ensuring that requests are routed to the right zone or handled according to your business rules, all before the main application code runs.

Let's explore a real example. I created a [T-Shirt Shop project](#) to offer a clear, practical example of how to implement a micro-frontend architecture using Next.js 15's multi-zone capabilities. In this setup, the application is divided into three distinct zones - Home, Catalog, and Account - each represented by its own independent Next.js application.

The Home zone acts as the landing page, welcoming users and showcasing featured products. The Catalog zone manages product listings and detailed product views, handling the core shopping experience. The Account zone is responsible for user authentication and profile management, ensuring that sensitive operations are kept logically and technically separate from the rest of the application.

Each zone maintains its own routing and configuration, which means teams can make decisions inside their own area of responsibility independently without impacting the rest of the platform.

Shared components are managed through a shared library, promoting consistency and reducing duplication. Styling across all zones is unified with Tailwind CSS, ensuring a cohesive look and feel no matter which part of the site a user visits.

## **Home zone: the entry point of your web application**

Starting from the home zone, the `next.config.js` file for the Home zone in the T-Shirt Shop project is a great example of how Next.js multi-zones orchestrate independent applications into a unified experience. The most critical aspect here is the `rewrites` configuration, which acts as the traffic director for requests that don't belong to the Home zone itself.

When a user visits the site, requests for the root path or any route handled by the Home zone are processed locally. However, if a user navigates to a path that begins with `/catalog/` or `/account/`, the Home zone's Next.js server uses the `rewrites` defined in `next.config.js` to transparently forward these requests to the appropriate remote zone. This is achieved by mapping the incoming path to the destination URL and base path of the Catalog or Account zones, as specified by environment variables. The result is a seamless experience for the user, who remains on the same domain, while under the hood, the request is routed to a completely separate Next.js application.

This approach ensures that each zone can be developed, deployed, and scaled independently, while the Home zone acts as the entry point and

traffic coordinator. The rewrites are handled at the server level, so the browser is unaware that the handoff-navigation to `/catalog/shirts` or `/account/profile` appears as a natural part of the site, even though those pages are rendered by different applications.

Although this mechanism might initially seem problematic, the rewrites code actually offers enough flexibility to define the first-level URL once and require only minimal changes throughout the project's lifecycle:

```
async rewrites() {
  return {
    fallback: [
      {
        source: '/catalog/:path*',
        destination:
` ${process.env.NEXT_PUBLIC_CATALOG_URL}${process.env.NEXT_PUBLIC_CATALOG_BASE_PATH}/:path*`,
      },
      {
        source: '/account/:path*',
        destination:
` ${process.env.NEXT_PUBLIC_ACCOUNT_URL}${process.env.NEXT_PUBLIC_ACCOUNT_BASE_PATH}/:path*`,
      },
    ],
  };
},
```

To visualize how this works, consider the following sequence diagram, written in Mermaid.js syntax. This diagram illustrates what happens when a user requests the `/catalog/shirts` page ([Figure 4-3](#)):

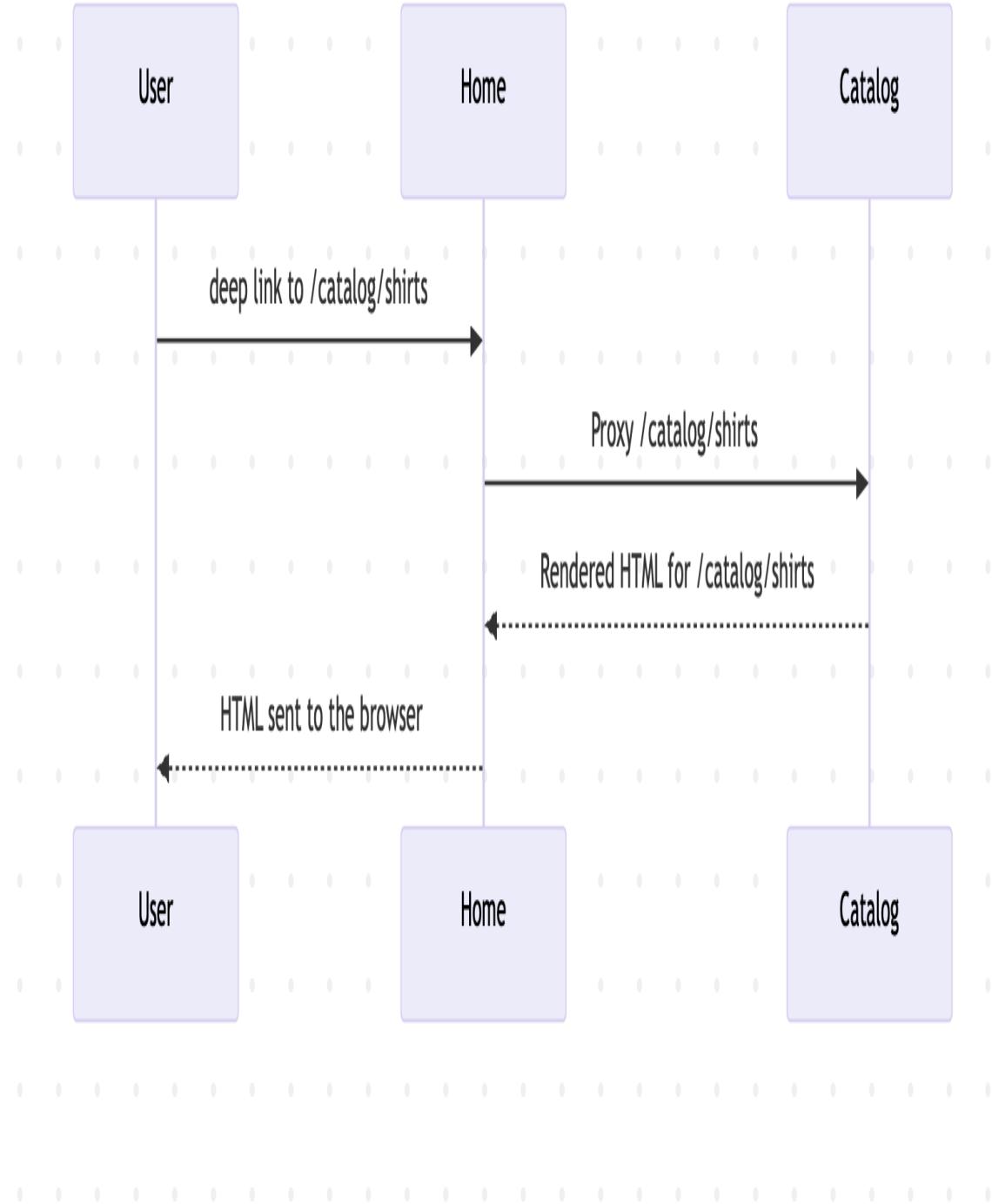


Figure 4-3. This sequence diagram highlights how the home zone acts as proxy for the other zones.

In this flow, the user's browser sends a request to the Home zone's Next.js server. The server, recognizing that the path matches the

`/catalog/:path*` pattern, proxies the request to the Catalog zone as defined in the rewrites. The Catalog zone renders the requested page and returns the HTML back to the Home zone, which then delivers it to the user. This pattern is repeated for any route that belongs to a different zone, allowing the application to scale horizontally and maintain clear separation of concerns.

## How to handle shared components

In a multi-zone micro-frontend architecture like the one used in the T-Shirt Shop project, the way you share components across zones is a crucial design decision. In this setup, shared components such as headers, or footers that span across all the different zones are included at build time rather than at runtime. This approach means that each zone—whether it's Home, Catalog, or Account imports the shared components directly from a common library as part of its build process. As a result, the components are bundled into the final output of each independent Next.js application.

Choosing build-time sharing brings several advantages. Most importantly, it avoids duplication of logic and styling at runtime, which can otherwise lead to inconsistent user experiences and unnecessarily large JavaScript bundles. By ensuring that all zones use the same version of shared components at build time, you maintain a consistent look and feel throughout the application. This also simplifies debugging and maintenance, since any updates to a shared component are picked up the next time each zone is built and deployed.

However, this approach does require some discipline around package management and versioning. Automation tools like Dependabot come very handy here because it will simplify the review of zones using shared components and they can automatically bump up a dependency and open a PR for a team review. This is a manageable trade-off for the benefits of speed and simplicity at runtime.

In this example, we updated the `layout.jsx` leveraging the shared components available in this monorepo approach:

```
import "@t-shirt-shop/shared/src/styles/globals.css";
import { Header, Footer } from "@t-shirt-shop/shared";

export default function RootLayout({
  children,
}: Readonly<{
  children: React.ReactNode;
}>) {
  return (
    <html lang="en">
      <body className={inter.className}>
        <div className="min-h-screen flex flex-col">
          <Header />
          <main className="flex-grow">{children}</main>
          <Footer />
        </div>
      </body>
    </html>
  );
}
```

Everything we've said about code reusability still holds true here. If you're unsure, just start by duplicating the code. Later, see if it really makes sense to create (and maintain!) an abstraction like a shared component. Try not to share too much too soon—it can add complexity and make life harder for other teams.

Alternatively, some organizations experiment with runtime sharing of components using technologies like module federation or injection of HTML fragments. This allows zones to load shared components dynamically at runtime, which can make it easier to roll out updates without rebuilding every zone. While this offers flexibility, it introduces additional complexity in dependency management and can lead to version mismatches or unexpected behaviors if different zones load incompatible versions of a component.

For most teams, especially those prioritizing performance and reliability, build-time sharing is the preferred approach. It keeps runtime environments lean and predictable, and it ensures that styling and behavior remain consistent across all parts of the application, no matter which zone a user is navigating.

## How to manage data

Managing ephemeral data and cross-zone communication is a subtle but vital aspect of building a robust multi-zone micro-frontend architecture. In this model, each zone is an independent application, so sharing information between them requires careful consideration of both user experience and security.

For lightweight, transient information such as search terms, product IDs, or promo codes-query, strings are an effective and straightforward solution.

For example, if a user enters a promo code on the Home page and then navigates to a specific product in the Catalog zone, the promo code can be appended to the URL as a query parameter. When the Catalog zone receives the request, it can extract the promo code from the query string and apply any relevant discounts or messaging. This approach is simple, transparent, and works well for data that doesn't need to persist beyond a single navigation or session. You could also consider using session storage, but keep its security model in mind. Session and local storage are tied to their subdomains, so the session storage on `www.mysite.com` can't be seen by `catalog.mysite.com` and vice versa.

However, not all data is suitable for this kind of ephemeral, client-side transfer. For persistent or sensitive information such as the number of items in a user's cart, authentication tokens, or user preferences, it's best to rely on backend storage or APIs to synchronize state across zones. For instance, the cart icon displayed in the header across all zones should reflect the current state of the user's shopping cart, regardless of which zone the user is in. This ensures consistency, security, and a seamless experience as users move between different parts of the application.

You might be wondering: what about a user session token?

In a multi-zone setup, where each part of the application (such as the Home, Account, and Catalog zones) is independently deployed and serves different functionality, authentication must be handled efficiently to maintain a seamless user experience.

Authentication is often managed using tokens, typically stored in cookies, which can be securely accessed by each independent zone to validate the user's session. This method enables seamless authentication as users navigate between zones while ensuring secure access to resources across the entire application.

At the core of a secure authentication system is the use of tokens, usually in the form of JWTs (JSON Web Tokens), which are issued by an authentication service when a user logs in. These tokens are set as cookies in the browser to persist the user's session across multiple requests and pages.

To ensure the security of these cookies, it's important to set several key attributes:

#### *HttpOnly*

This prevents the cookie from being accessed via JavaScript, safeguarding it from cross-site scripting (XSS) attacks.

#### *Secure*

Ensures the cookie is transmitted only over HTTPS, protecting it from man-in-the-middle attacks.

#### *SameSite*

By setting this attribute to Lax or Strict, you can prevent cross-site request forgery (CSRF) attacks by restricting when cookies are sent with cross-site requests.

With these security measures in place, whenever a user navigates between zones—whether it's moving from the Home page to the Account page or accessing personalized content on the Catalog zone—the authentication token is included automatically in the HTTP headers. This allows the server of each zone to validate the token, perform SSR checks, and ensure the user is authorized to access protected resources, such as their profile information or past order history.

When a user requests a page in a multi-zone architecture, middleware can intercept the request before it reaches the page's rendering logic. This is an effective way to ensure that:

- The user's session is valid and the authentication token is present.
- The token is verified using the secret key, ensuring it hasn't been tampered with.

By centralizing this logic in middleware, you eliminate the need to duplicate authentication and authorization code across multiple zones, leading to a more maintainable and secure system. This ensures that sensitive resources, like user profiles or admin panels, are only accessible by authorized users, while other users are redirected to login pages or access-denied pages as needed.

To ensure persistent user sessions, it's a good practice to use refresh tokens alongside short-lived access tokens. Access tokens typically have an expiration time (e.g., 15 minutes to an hour), while refresh tokens are long-lived and can be used to obtain new access tokens when the current one expires. This mechanism helps in maintaining user sessions without forcing users to log in frequently.

The refresh token can be securely stored in an `HttpOnly` cookie, and the backend API can issue a new access token when the old one expires, based on the valid refresh token. This strategy provides a balance of security and usability, preventing users from being logged out frequently while still ensuring that stolen tokens are only useful for a limited time.

Beyond using cookies, another important consideration is where and how you store tokens. `HttpOnly` cookies remain the most secure choice for session management, as they cannot be accessed by JavaScript running in the browser, protecting them from XSS attacks. It's important to never store sensitive tokens in `localStorage` or `sessionStorage`, as these are vulnerable to JavaScript access and can be easily exploited if an attacker gains control over the client-side application.

Tokens should always be transmitted over HTTPS to prevent interception by attackers on insecure networks. The use of the `Secure` cookie attribute ensures that tokens are only sent over secure connections.

Whether you are using Next.js, Astro.js, or any other SSR solution, following these principles will help you build a robust and secure authentication and authorization system in your micro-frontend architecture.

## Vercel: a glance into the future

After leveraging multi-zone architecture for vertically split micro-frontends, it's clear Vercel is pushing innovation even further. CTO Malte Ubl has confirmed micro-frontends will become first-class primitives on the platform, and recent discussions with Vercel's engineering team reveal concrete steps toward this vision. They're developing tools to streamline federated rendering, environment management, and remote component integration—key pain points in micro-frontend architectures.

A standout innovation is their approach to environment-agnostic module federation, where relative paths for remote entries allow seamless switching between production, staging, and local setups. This is managed via a centralized `microfrontends.json` configuration file, paired with a developer toolbar for dynamic environment simulation.

The beauty of this approach is that any developer can swap micro-frontends directly from the UI in any environment, including production, without having to deploy them separately in each environment. In my opinion, this will significantly simplify the development and testing of micro-frontends before running the usual automation for deploying a new version.

For SSR-specific challenges, Vercel uses rewrite rules and cookie-based routing overrides to maintain security without compromising user experience.

React Server Components (RSCs) are poised to play a pivotal role. Vercel's engineers are prototyping **Remote Components**, a powerful abstraction

that enables code transclusion into Next.js applications. This mechanism allows hosts to embed server-rendered content from external services with minimal client-side JavaScript. While currently in early stages, the long-term vision is to open-source Remote Components and make them framework-agnostic—supporting not only React, but also other frontend frameworks capable of leveraging RSC, Incremental Static Generation (ISG), or even plain HTML. This positions Vercel as a central player in enabling native integration of distributed UIs, regardless of framework choice.

As this book is being written, it's exciting to note that the discussions between Vercel and the Module Federation team have resumed. The Vercel platform's strategy is not limited to the Module Federation. Support is expanding to include **Single-SPA** and any other existing frameworks, including custom micro-frontend frameworks. This underscores Vercel's commitment to flexibility and composability across diverse technical stacks.

In the second half of 2025, Vercel is preparing a public beta rollout, with plans for conference demos and ecosystem partnerships. While full details remain under wraps, their focus on developer experience (local proxying, hybrid deployments) and observability (cache health monitoring, dependency tracking) signals a holistic approach to micro-frontends—one that could redefine how teams build and scale composable web applications in the coming years.

## API Strategies

We cannot think about server-side rendered micro-frontends without considering API strategies—where the data comes from and how teams are structured to provide and consume that data. In SSR architectures, the way you design, aggregate, and cache data is just as important as how you compose your UI.

GraphQL has become just as popular as REST in modern micro-frontend architectures. While most companies still use REST APIs—often for

historical reasons—GraphQL is increasingly appreciated, especially by fullstack and frontend developers. Its flexibility allows teams to fetch exactly the data they need, which is particularly useful for supporting different screen sizes and device types, such as mobile apps and desktop web applications.

One of the main benefits of GraphQL in a micro-frontend setup is the ability to maintain a single unified graph for the entire application, rather than splitting GraphQL into multiple endpoints for each domain. This unified approach simplifies the data layer and allows backend services to evolve independently, without forcing frontend teams to constantly update their data-fetching logic.

For teams that prefer REST, the Backend for Frontend (BFF) pattern remains a solid choice. With BFF, each micro-frontend can have its own tailored backend service that aggregates data from various domains, maintaining a clear separation between the micro-frontend user interfaces and the underlying microservices.

However, perfectly dividing a system into subdomains and assigning each to a separate team is rarely straightforward.

For example, consider an e-commerce website: the product listing, shopping cart, and user profile might seem like distinct areas, but features such as personalized recommendations or promotional banners often require data from multiple domains. This overlap means teams must collaborate and share clear API contracts, which can introduce complexity and coordination challenges.

It's important to recognize that these boundaries are rarely perfect, and some cross-domain dependencies are inevitable.

Throughout this book, we'll explore API strategies in greater depth, helping you navigate the practical realities of building scalable, maintainable micro-frontend systems.

## **Don't fear the cache: scaling SSR with smart caching strategies**

Caching doesn't need to be intimidating. Even implementing a small 5-second TTL (Time-To-Live) can dramatically reduce the strain on your backend systems during traffic spikes. When a popular sports event or breaking news story drives thousands of visitors to your site simultaneously, these brief caching windows prevent your databases and APIs from being overwhelmed.

## **Types of Caches Every Developer Should Know**

Before diving into real-world implementations, let's understand the different types of caches you should consider when building server-side rendered micro-frontends.

### **Content Delivery Networks**

CDNs store content at edge locations around the world, serving it directly to users without hitting your origin servers. CDNs excel at caching with:

- Static assets (images, CSS, JavaScript)
- Full HTML pages that don't require personalization
- API responses that can be shared across users

A CDN is your first line of defense against traffic spikes, often handling 80-90% of requests for popular content. I had customers who were able to reduce their transactions per second (TPS) from tens of thousands to dozens leveraging CDNs properly.

## In-Memory Database Cache

In-memory solutions like Redis provide microsecond response times, making them perfect for server-side rendering scenarios. The Cache-Aside pattern is most commonly used in micro-frontends applications.

This caching pattern is pretty straightforward. Your application checks the cache first before querying backends, updating the cache with fresh data when needed. This pattern works exceptionally well for read-heavy workloads like serving web pages.

For example, imagine a news website that needs to render a “Top Stories” component on multiple pages. Instead of making repeated API calls to the content service for each user request, you can cache the HTML fragment:

```
async function getTopStoriesComponent(req, res) {
  // First check if we have it in Redis
  const cachedHTML = await redisClient.get('top-stories-
fragment');

  if (cachedHTML) {
    // We found it in cache, use it directly
    return cachedHTML;
  }

  // Not in cache, call the content service
  const stories = await contentService.getTopStories();
  const htmlFragment = renderToString(<TopStoriesComponent
stories={stories} />);

  // Store in Redis with a 5-minute TTL
  await redisClient.set('top-stories-fragment', htmlFragment,
'EX', 300);

  return htmlFragment;
}
```

This approach can reduce page rendering time from hundreds of milliseconds to just a few milliseconds, significantly improving both user experience and server throughput during traffic spikes. In some projects I

worked on, I was able to achieve query times of around 5ms for gathering 1MB of HTML. Absolutely a life saver!

## Warm Cache

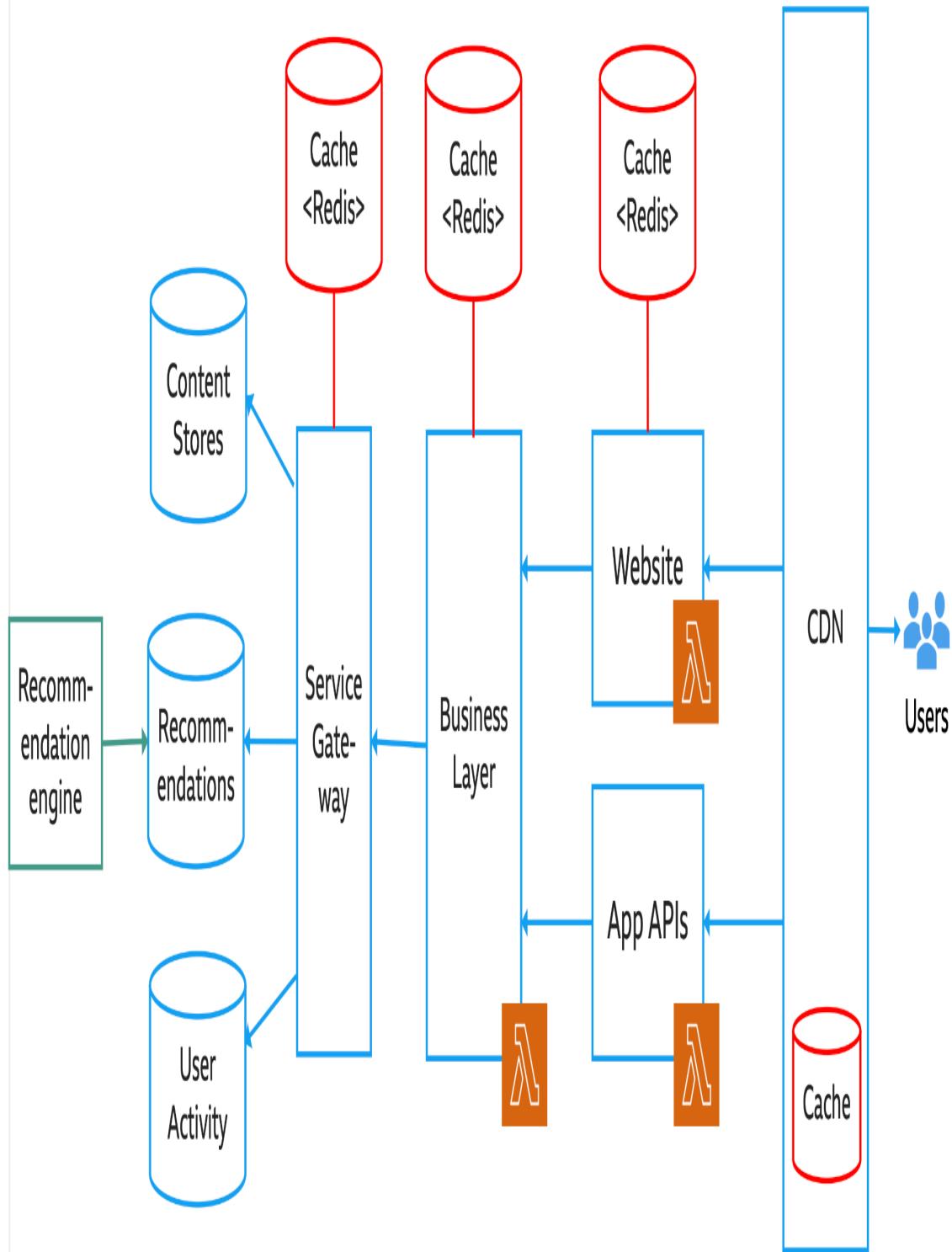
Instead of waiting for the first user request to populate a cache (creating a poor experience for that user sometimes), always-warm caches proactively load data before it's needed:

- Periodic jobs populate caches with frequently accessed data
- Background processes refresh cache entries before expiration
- Pre-computation of expensive operations during low-traffic periods

## The BBC Architecture: Caching in Action

During QCon London 2025, the BBC shared their implementations using multiple caching layers. In fact, the BBC's architecture demonstrates how these caching strategies work together in a real-world, high-scale system. Let's follow a request through their architecture to see caching in action ([Figure 4-4](#)).

# Three patterns for scale: caching, serverless, and pre-compute



*Figure 4-4. - BBC website high level architecture*

When someone wants to check the latest football scores or breaking news on the BBC, their request first hits the CDN. For popular content like major news stories or sports events, the CDN likely already has the response cached, allowing it to serve the content immediately without touching any of BBC's internal systems. This cache absorbs the majority of repeated requests, ensuring that only truly unique or uncached requests travel deeper into the system.

Moreover, this approach enables you to configure different cache TTLs based on content volatility. Breaking news pages might cache for just 30 seconds, while archived content could be cached for days. Even those brief 30-second windows dramatically reduce origin load during traffic surges.

If the CDN doesn't have the content cached (a "cache miss"), the request reaches either the Website or App APIs layer. Here, Lambda functions (shown by the orange  $\lambda$  icons) handle server-side rendering.

Notice how each of these components connects to its own Redis cache. These caches store rendered HTML fragments and API responses, creating a buffer between the rendering layer and deeper backend services.

For instance, for highly dynamic but frequently accessed components like sports scores or weather updates, BBC could set short TTLs of 5-15 seconds. This brief window helps the system handle massive concurrent requests while still keeping content relatively fresh.

The Business Layer, which aggregates and applies business logic to data from multiple sources, also leverages serverless functions for scalability and flexibility. At this stage, a dedicated Redis cache helps ensure that frequently requested business data is served quickly, reducing the need for repeated computation and further protecting backend services from spikes in demand.

Thanks to the caching approach on this layer, you can implement circuit breakers for making the system more resilient. If backend services become

slow or unresponsive, the system can serve slightly stale cached data instead of failing completely, prioritizing availability over perfect freshness.

The Service Gateway connects the business logic to the underlying data sources. This crucial layer has its own Redis cache to minimize calls to backend systems.

**Integrated Best Practice:** The BBC implements staggered cache expiration times at this layer. Instead of all cache entries expiring simultaneously (potentially causing a “thundering herd” of requests to backends), each entry gets a slightly randomized TTL, spreading the load more evenly.

When a major news event breaks or a significant sports match concludes, the BBC might see traffic increase tenfold within minutes. Without this layered caching strategy, their servers would likely collapse under the load.

Instead, the CDN handles most requests, while the various Redis caches ensure that even dynamic, personalized content can be delivered quickly without overwhelming backend systems.

Even implementing a brief 5-10 second cache for data that changes infrequently can make the difference between a smooth user experience and system failure during peak traffic. For instance, when England scores in a World Cup match, millions of users might check BBC Sport simultaneously —these brief caching windows ensure the underlying content systems don’t receive millions of identical requests.

## **Performance: The Key Reason for Server-Side Rendering**

Performance is at the heart of why organizations choose server-side rendering, and it’s especially critical when you’re building with micro-frontends. The journey to a fast, interactive web experience doesn’t end with delivering HTML quickly; it’s about what happens next—how and when your JavaScript loads, how much of it runs, and how users actually experience your site across devices and network conditions.

One of the most effective strategies is to use partial hydration or resumability. Instead of hydrating the entire page, you focus on making only the interactive parts of your UI come alive on the client. For example, a news homepage might render the article content and navigation instantly, but only hydrate the comments section or live score widgets when a user scrolls them into view. Frameworks like Preact, Astro and Qwik are leading the way here, allowing you to deliver static content immediately while deferring or even skipping hydration for parts of the page that don't need it. This approach dramatically reduces the amount of JavaScript that needs to run on initial load, which is especially important for users on slower devices or networks.

To ensure these optimizations are working, you need to measure performance continuously and objectively. Lighthouse is an invaluable tool for this. By running Lighthouse audits on your micro-frontends throughout development and as part of your CI/CD pipeline, you can track key metrics like Largest Contentful Paint (LCP), First Contentful Paint (FCP), and Time to Interactive (TTI). Setting a performance budget for each micro-frontend—such as a maximum JavaScript bundle size or a target LCP—helps teams catch regressions early. If a new feature pushes your bundle over budget or slows down the first paint, you'll know right away, before it reaches production.

A technique I've found both easy to implement and extremely valuable for improving the developer feedback loop is to generate the final micro-frontend artifact using hooks provided by your version control system. When a developer opens a pull request, the hook triggers a service that clones the project, builds it, and checks if the final artifact size exceeds a specified threshold. If it does, you can choose to block the merge or notify the team in another way. This approach helps teams catch potential performance issues early and maintain high standards for bundle size and efficiency.

But synthetic tests alone aren't enough. Real-world observability in production is essential. You need to see how your application performs for actual users, across a variety of devices, browsers, and network conditions.

Real User Monitoring (RUM) tools can capture this data, showing you not just averages but the outliers—those users on older phones or spotty connections who may be having a very different experience. With this insight, you can prioritize optimizations that matter most, like improving hydration speed on mobile or reducing JavaScript for users in regions with slower networks.

***Performance work is never “done.” It’s a cycle: optimize, measure, observe, and repeat.***

By combining SSR, smart hydration strategies, continuous Lighthouse checks, and robust observability, you can ensure your micro-frontends deliver not just fast initial loads but consistently great experiences for all your users.

## Summary

As you move forward with SSR and micro-frontends, keep your focus on what matters most: delivering a fast, seamless experience for your users while empowering your teams to work efficiently and independently.

Don’t feel pressured to chase every new framework or architectural pattern—instead, pick the tools and strategies that best fit your team’s strengths and your project’s needs. Whether you’re experimenting with Next.js multi-zones, Astro islands, or classic HTML fragment composition, remember that clear ownership, thoughtful boundaries, and regular performance checks will take you much further than any one piece of technology.

Start small, measure often, and don’t be afraid to iterate. Use real user data to guide your optimizations, and encourage your teams to share what works—and what doesn’t.

Most importantly, keep the lines of communication open between teams so your micro-frontends grow together, not apart. Building at scale is always a journey, but with the right mindset and a willingness to learn, you’ll be well-equipped to deliver robust, modern web applications that delight your users and stand the test of time.

# Chapter 5. Automation of Micro-Frontends

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the sixth chapter of the final book. Please note that the GitHub repo will be made active later on.

If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the author at  
*[building.microfrontends@gmail.com](mailto:building.microfrontends@gmail.com)*.

In this chapter, we discuss another key topic for distributed systems like micro-frontends: the importance of a solid automation strategy. The microservices architecture adds great flexibility and scalability to our architecture, allowing our APIs to scale horizontally based on the traffic our infrastructure receives and allowing us to implement the right pattern for the right job instead of having a common solution applied to all our APIs, as in a monolithic architecture. Despite these great capabilities, microservices increase the complexity of managing the infrastructure, requiring an immense number of repetitive actions to build and deploy them. Any company embracing the microservices architecture, therefore, must invest a considerable amount of time and effort on their *continuous integration* (CI) or *continuous deployment* (CD) pipelines (more on these to come). Given how fast a business can drift direction nowadays, improving a CI/CD pipeline is not only a concern at the beginning of a project; it’s a constant incremental improvement throughout the entire project life cycle. One of

the key characteristics of a solid automation strategy is that it creates confidence in artifacts' replicability and provides fast feedback loops for developers.

This is also true for micro-frontends. Having solid automation pipelines will allow our micro-frontend projects to be successful, creating a reliable solution for developers to experiment, build, and deploy. In fact, for certain projects, micro-frontends could proliferate in such a way that it would become nontrivial to manage them. One of the key decisions listed in the micro-frontend decisions framework, discussed in Chapter 2, is the possibility of composing multiple micro-frontends in the same view (horizontal split) or having just one micro-frontend at a time (vertical split). With the horizontal split, we could end up with tens or even hundreds of artifacts to manage in our automation pipelines. Therefore, we have to invest in solutions to manage such scenarios. Vertical splits also require work but are close to the traditional way to set up automation for single-page applications (SPAs). The major difference is you'll have more than one artifact and potentially different ways to build and optimize your code.

We will dive deep into these challenges in this chapter and the following ones, starting with the principles behind a solid and fast automation strategy and how we can improve the developer experience with some simple but powerful tools. Then we'll analyze best practices for continuous integration and micro-frontend deployment. Finally, we conclude with an introduction to fitness functions for automating and testing architecture characteristics during different stages of the pipelines.

## Automation Principles

Working with micro-frontends requires constantly improving the automation pipeline. Skipping this work may hamper the delivery speed of every team working on the project and decrease their confidence to deploy in production or, worse, frustrate the developers as well as the business when the project fails. Nailing the automation part is fundamental if you're

going to have a clear idea of how to build a successful continuous integration, continuous delivery, or continuous deployment strategy.

## CONTINUOUS INTEGRATION VERSUS CONTINUOUS DELIVERY VERSUS CONTINUOUS DEPLOYMENT

An in-depth discussion about continuous integration, continuous delivery, and continuous deployment is beyond the scope of this book. However, it's important to understand the differences between these three strategies. Continuous integration defines a strategy where an automation pipeline kicks in for every merge into the main branch, extensively testing the codebase before the code is merged in the release branch. Continuous delivery is an extension of continuous integration where, after the tests, we generate the artifact ready to be deployed with a simple click from a deployment dashboard. Continuous deployment goes one step further, deploying in production the artifacts built after the code is committed in the version control system. If you are interested in learning more, I recommend reading *Continuous Delivery* by David Farley and Jez Humble (Addison-Wesley Professional).

To get automation speed and reliability right, we need to keep the following principles in mind:

- Keep the feedback loop as fast as possible.
- Iterate often to enhance the automation strategy.
- Empower your teams to make the right decisions for the micro-frontends they are responsible for.
- Identify some boundaries, also called guardrails, where teams operate and make decisions while maintaining tool standardization.
- Define a solid testing strategy.

Let's discuss these principles to get a better understanding of how to leverage them.

## Keep a Feedback Loop Fast

One of the key features of a solid automation pipeline is fast execution. Every automation pipeline provides feedback for developers. Having a quick turnaround on whether our code has broken the codebase is essential for developers to create confidence in what they have written. Good automation should run often and provide feedback in a matter of seconds or minutes, at the most. It's important for developers to receive constant feedback so they will be encouraged to run the tests and checks within the automation pipeline more often. It's essential, then, to analyze which steps may be parallelized and which are serialized. A technical solution that allows both is ideal. For example, we may decide to parallelize the unit testing step so we can run our tests in small chunks instead of waiting for hundreds, if not thousands, of tests to pass before moving to the next step. Yet some steps cannot be parallelized. So we need to understand how we can optimize these steps to be as fast as possible.

Working with micro-frontends, by definition, should simplify optimizing the automation strategy. Because we are dividing an entire application into smaller parts, there is less code to test and build, for instance, and every stage of a CI should be very fast.

## Iterate Often

An automation pipeline is not a piece of infrastructure that, once defined, remains as it is until the end of a life cycle project. Every automation pipeline has to be reviewed, challenged, and improved. It's essential to maintain a very quick automation pipeline to empower our developers to get fast feedback loops. In order to constantly improve, we need to visualize our pipelines. Dashboards can show how long building artifacts take, making clear to everyone on the team how healthy the pipelines are (or aren't) and immediately letting everyone know if a job failed or succeeded.

When we notice our pipelines taking more than 8 to 10 minutes, it's time to review them and see if we can optimize certain practices of an automation strategy. Review the automation strategy regularly: monthly if the pipelines are running slowly, and then every three to four months once they're healthy. Don't stop reviewing your pipelines after defining the automation pipeline. Continue to improve and pursue better performance and a quicker feedback loop; this investment in time and effort will pay off very quickly.

## **Empower Your Teams**

At several companies I worked for in the past, the automation strategy was kept out of capable developers' hands. Only a few people inside the organization were aware of how the entire automation system worked and even fewer were allowed to change the infrastructure or take steps to generate and deploy an artifact. This is the worst nightmare of any developer working in an organization with one or more teams. The developer's job shouldn't be just writing code; it should include a broad range of tasks, including how to set up and change the automation pipeline for the artifacts they are working on, whether it's a library, a micro-frontend, or the entire application.

Empowering our teams when we are working with micro-frontends is essential because we cannot always rely on all the micro-frontends having the same build pipeline due to the possibility of maintaining multiple stacks at the same time. Certainly, the deployment phase will be the same for all the micro-frontends in a project. However, the build pipeline may use different tools or different optimizations, and centralizing these decisions could result in a worse final result than one from enabling the developers to work in the automation pipeline.

Ideally, the organization should provide some guardrails for the development team. For instance, the CI/CD tool should be the platform team's responsibility, but all the scripts and steps to generate an artifact should be owned by the team because they know the best way to produce an optimized artifact with the code they have written. This doesn't mean creating silos between a team and the rest of the organization but

empowering them to make certain decisions that would result in a better outcome.

Last but not least, encourage a culture of sharing and innovation by creating moments for the teams to share their ideas, proof of concepts, and solutions. This is especially important when you work in a distributed environment. Remote meetings lack the everyday casual work conversations we have around the coffee machine. Creating a virtual moment for enjoying these conversations again may seem an overkill at the beginning, but it helps the morale and the connections between team members.

## Define Your Guardrails

An important principle for empowering teams and having a solid automation strategy is creating some guardrails for them, so we can make sure they are heading in the right direction.

Guardrails for the automation strategy are boundaries identified by tech leadership, in collaboration with architects and/or platform or cloud engineers, between which teams can operate and add value for the creation of micro-frontends.

In this situation, guardrails might include the tools used for running the automation strategy, the dashboard used for deployment in a continuous delivery strategy, or the fitness functions for enforcing architecture characteristics that we discuss extensively during this chapter.

Introducing guardrails won't mean reducing developers' freedom. Instead, it will guide them toward using the company's standards, abstracting them as much as we can from their world, and allowing the team to innovate inside these boundaries. We need to find the right balance when we define these guardrails, and we need to make sure everyone understands the *why* of them more than the *how*. Usually, creating documentation helps to scale and spread the information across teams and new employees. As with other parts of the automation strategy, guardrails shouldn't be static. They need to be revised, improved, or even removed, as the business evolves.

## Define Your Test Strategy

Investing time on a solid testing strategy is essential, specifically end-to-end testing, for instance, when we have multiple micro-frontends per view with multiple teams contributing to the final results and we want to ensure our application works end to end. In this case, we must also ensure that the transition between views is covered and works properly before deploying our artifacts in production.

While unit and integration testing are important, micro-frontends do not have particular challenges to face. Instead end-to-end testing has to be revised for applying it to this architecture. Because every team owns a part of the application, we need to make sure the critical path of our applications is extensively covered and we achieve our final desired result. End-to-end testing will help ensure those things.

## Developer Experience

A key consideration when working with micro-frontends is the developer experience (DX). While not all companies can support a DX team, even a virtual team across the organization can be helpful. Such a team is responsible for creating tools and improving the experience of working with micro-frontends to prevent friction in developing new features.

At this stage, it should be clear that every team is responsible for part of the application and not for the entire codebase. Creating a frictionless developer experience will help our developers feel comfortable building, testing, and debugging the part of the application they are responsible for. We need to guarantee a smooth experience for testing a micro-frontend in isolation, as well as inside the overall web application, because there are always touch points between micro-frontends, no matter which architecture we decide to use. A bonus would be creating an extensible experience that isn't closed to the possibility of embracing new or different tools during the project life cycle.

## WHAT DOES DEVELOPER EXPERIENCE MEAN?

DX is usually one or more teams dedicated to studying, analyzing, and improving how developers get their work done. Specifically, such teams observe which tools and processes developers use to accomplish their daily work providing support for improving the development life cycle across the entire organization. One of DX's goals is to simplify the development and process of building, testing, and deploying artifacts in different environments. But also providing tools that enhance the speed of delivery and accelerate the feedback loop of developers automating guardrails.

Many companies have created end-to-end solutions that they maintain alongside the projects they are working on, which more than fills the gaps of existing tools when needed. This seems like a great way to create the perfect developer experience for our organization, although businesses aren't static, nor are tech communities. As a result, we need to account for the cost of maintaining our custom developers' experience, as well as the cost of onboarding new employees. It may still be the right decision for your company, depending on its size or the type of the project you are working on, but I encourage you to analyze all the options before committing to building an in-house solution to make sure you maximize the investment.

## Horizontal Versus Vertical Split

The decision between a horizontal and vertical split with your new micro-frontends project will definitely impact the developer's experience. A vertical split will represent the micro-frontends as single HTML pages or SPAs owned by a single team, resulting in a developer experience very similar to the traditional development of an SPA. All the tools and workflows available for SPA will suit the developers in this case. You may want to create some additional tools specifically for testing your micro-frontend under certain conditions as well. For instance, when you have an

application shell loading a vertical micro-frontend, you may want to create a script or tool for testing the application shell version available on a specific environment to make sure your micro-frontend works with the latest or a specific version.

The testing phases are very similar to a normal SPA, where we can set unit, integration, and end-to-end testing without any particular challenges.

Therefore, every team can test its own micro-frontends, as well as the transition between micro-frontends, such as when we need to make sure the next micro-frontend is fully loaded. However, we also need to make sure all micro-frontends are reachable and loadable inside the application shell. One solution I've seen work very well is having the team that owns the application shell do the end-to-end testing for routing between micro-frontends so they can perform the tests across all the micro-frontends.

Horizontal splits come with a different set of considerations. When a team owns multiple micro-frontends that are part of one or more views, we need to provide tools for testing a micro-frontend inside the multiple views assembling the page at runtime. These tools need to allow developers to review the overall picture, potential dependencies clash, the communication with micro-frontends developed by other teams, and so on. These aren't standard tools, and many companies have had to develop custom tools to solve this challenge. Keep in mind that the right tools will vary, depending on the environment and context we operate in, so what worked in one company may not fit in another. Some solutions associated with the framework we decided to use will work, but more often than not, we will need to customize some tools to provide our developers with a frictionless experience.

Another challenge with a horizontal split is how to run a solid testing strategy. We will need to identify which team will run the end-to-end testing for every view and how specifically the integration testing will work, given that an action happening in a micro-frontend may trigger a reaction with another. We do have ways to solve these problems, but the governance behind them may be far from trivial. The developer experience with micro-frontends is not always straightforward. The horizontal split in particular is

challenging because we need to answer far more questions and make sure our tools are constantly up to date to simplify the life of our developers.

## Frictionless Micro-Frontends Blueprints

The micro-frontend developer experience isn't only about development tools; we must also consider how the new micro-frontends will be created. The more micro-frontends we have and the more we have to create, the more speeding up and automating this process will become mandatory. Creating a command-line tool for scaffolding a micro-frontend will not only cover implementation, allowing a team to have all the dependencies for starting to write code, but also take care of collecting and providing best practices and guardrails inside the company. For instance, if we are using a specific library for observability or logging, adding the library to the scaffolding can speed up creating a micro-frontend—and it guarantees that your company's standards will be in place and ready to use.

An alternative that has gained more traction in recent years is a centralized portal where developers can retrieve blueprints for micro-frontends and other resources. Open-source tools like [Spotify's Backstage](#) help centralize blueprints, documentation, and more in a single platform. This provides developers with a one-stop destination for their day-to-day needs, eliminating the need to search through scattered resources across multiple internal systems such as wikis, repositories, and other tools. I have seen tens of companies implementing this strategy successfully for their distributed systems including for micro-frontends.

Another important item to provide out of the box would be a sample of the automation strategy, with all the key steps needed for building a micro-frontend. Imagine that we have decided to run static analysis and security testing inside our automation strategy. Providing a sample of how to configure it automatically for every micro-frontend would increase developers' productivity and help get new employees up to speed faster. This scaffolding would need to be maintained in collaboration with developers learning the challenges and solutions directly from the trenches. A sample can help communicate new practices and specific changes that

arise during the development of new features or projects, further saving your team time and helping them work more efficiently.

## Environments Strategies

Another important consideration for the DX is enabling teams to work within the company's environments strategy. The most commonly used strategy across midsize to large organizations is a combination of testing, staging, and production environments. The testing environment is often the most unstable of the three because it's used for quick attempts made by the developers. As a result, staging should resemble the production environment as much as possible, the production environment should be accessible only to a subset of people, and the DX team should create strict controls to prevent manual access to this environment and provide a swift solution for promoting or deploying artifacts in production.

An interesting twist to the classic environment strategy is spinning up environments with a subset of a system for testing of any kind (end-to-end or visual regression, for instance) and then tearing them down when an operation finishes. This particular strategy of on-demand environments is a great addition for the company because it helps not only with micro-frontends but also with microservices for testing in isolation end-to-end flows. With this approach we can also think about end-to-end testing in isolation of an entire business subdomain, deploying only the microservices needed and having multiple on-demand environments, saving a considerable amount of money.

Another feature provided by on-demand environments is the possibility to offer the business or a product owner a preview of an experiment or a specific branch containing a feature. Nowadays, many cloud providers like AWS can provide great cost savings using **spot instances** for a middle-of-the-road approach, where the infrastructure is more cost effective than the normal offering because of the spare capacity borrowed by customers for a limited amount of time. Spot instances are a perfect fit for on-demand environments.

# Version Control

When we start to design an automation strategy, selecting a version control and branching strategy to adopt is a mandatory step. Although there are valid alternatives, like Mercurial, Git is the most popular version control system. I'll use Git as a reference in my examples below, but know that all the approaches are applicable to Mercurial as well. Working with version control means deciding which approach to use in terms of repositories. Usually, the debate is between monorepo and polyrepo, also called multirepo. There are benefits and pitfalls in both approaches. You can employ both, though, in your micro-frontend project, to use the right technique for your context.

## Monorepo

Monorepo (see [Figure 5-1](#)) is based on the concept that all the teams are using the same repository, so therefore all the projects are hosted together.

# My Project



Sign-in MFE



Sign-up MFE



Catalog MFE



iOS app



Android app



Auth service



Catalog service



Search service

...

*Figure 5-1. Monorepo example where all the projects live inside the same repository*

The main advantages of using monorepo are:

#### *Code reusability*

Sharing libraries becomes very natural with this approach. Because all of a project's codebase lives in the same repository, we can smoothly create a new project, abstracting some code and making it available for all the projects that can benefit from it.

#### *Easy collaboration across teams*

Because the discoverability is completely frictionless, teams can contribute across projects. Having all the projects in the same place makes it easy to review a project's codebase and understand the functionality of another project. This approach facilitates communication with the team responsible for the maintenance in order to improve or simply change the implementation pointing to a specific class or line of code without kicking off an abstract discussion.

#### *Cohesive codebase with less technical debt*

Working with monorepo encourages every team to be up to date with the latest dependency versions, specifically APIs but generally with the latest solutions developed by other teams, too. This would mean that our project may be broken and would require some refactoring when there is a breaking change, for instance. Monorepo forces us to continually refactor our codebase, improving the code quality and reducing our tech debt.

#### *Simplified dependencies management*

With monorepo, all the dependencies used by several projects are centralized, so we don't need to download them every time for all our projects. When we want to upgrade to the next major release, all the teams using a specific library will have to work together to update their codebase, reducing the technical debt that in other cases a team may

accumulate. Updating a library may cost a bit of coordination overhead, especially when you work with distributed teams or large organizations.

### *Large-scale code refactoring*

Monorepo is very useful for large-scale code refactoring. Because all the projects are in the same repository, refactoring them at the same time is trivial. Teams will need to coordinate or work with technical leaders, who have a strong high-level picture of the entire codebase and are responsible for refactoring or coordinating the refactor across multiple projects.

### *Easier onboarding for new hires*

With monorepo, a new employee can quickly find code samples from other repositories. Additionally, a developer can find inspiration from other approaches and quickly shape them inside the codebase.

Despite the undoubted benefits, embracing monorepo also brings some challenges:

### *Constant investment in automation tools*

Monorepo requires a constant, critical investment in automation tools, especially for large organizations. There are plenty of open source tools available, but not all are suitable for monorepo, particularly after some time on the same project, when the monorepo starts to grow exponentially. Many large organizations must constantly invest in improving their automation tools for monorepo to avoid their entire workforce being slowed down by intermittent commitment to improving the automation pipelines and reducing the time of this feedback loop for the developers.

### *Scaling tools when the codebase increases*

Another important challenge is that automation pipelines must scale alongside the codebase. Many whitepapers from Google, Facebook, and Twitter claim that after a certain threshold, the investment in having a

performant automation pipeline increases until the organization has several teams working exclusively on it. Unsurprisingly, every aforementioned company has built its own version of build tools and released it as open source to deal with the unique challenges they face with thousands of developers working in the same repository.

### *Projects are coupled together*

Given that all the projects are easy to access and, more often than not, they are sharing libraries and dependencies, we risk having tightly coupled projects that can exist only when they are deployed together. We may, therefore, not be able to share our micro-frontends across multiple projects for different customers where the codebase lives in a different monorepo. This is a key consideration to think about before embracing the monorepo approach with micro-frontends.

### *Trunk-based development*

**Trunk-based development** is the only option that makes sense with monorepo. This branching strategy is based on the assumption that all the developers commit to the same branch, called a trunk. Considering that all the projects live inside the same repository, the trunk main branch may have thousands of commits per day, so it's essential to commit often with small commits instead of developing an entire feature per day before merging. This technique should force developers to commit smaller chunks of code, avoiding the “merge hell” of other branching strategies. Although I am a huge fan of trunk-based development, it requires discipline and maturity across the entire organization to achieve good results.

### *Disciplined developers*

We must have disciplined developers in order to maintain the codebase in a good state. When tens, or even hundreds, of developers are working in the same repository, the Git history, along with the codebase, could become messy very quickly. Unfortunately, it's almost impossible to have senior developers inside all the teams, and that lack of knowledge

or discipline could compromise the repository quality and extend the blast radius from one project inside the monorepo to many, if not all, of them.

Using monorepo for micro-frontends is definitely an option, and tools like [Lerna](#) help with managing multiple projects inside the same repository. In fact, Lerna can install and [hoist](#), if needed, all the dependencies across packages together and publish a package when a new version is ready to be released. However, we must understand that one of the main monorepo strengths is its code-sharing capability. It requires a significant commitment to maintain the quality of the codebase, and we must be careful to avoid coupling too many of our micro-frontends because we risk losing their nature of independent deployable artifacts.

Git has started to invest in reducing the operations time when a user invokes commands like Git history or Git status in large repositories. And as monorepo has become more popular, Git has been actively working on delivering additional functionalities for filtering what a user wants to clone to their machine without needing to clone the entire Git history and all the project's folders.

Obviously, these enhancements will also be beneficial for our CI/CD, where we can overcome one of the main challenges of embracing a monorepo strategy.

## SPARSE-CHECKOUT

In Q1 2020, Git introduced the `sparse-checkout` command (for v2.25 and later) for cloning only part of a repository instead of all the files and history of a repository. Reducing the amount of data to clone for running fast automation pipelines would solve one of the main challenges of embracing the monorepo approach.

We need to remember that using the monorepo approach would mean investing in our tools, evangelizing and building discipline across our

teams, and finally accepting a constant investment in improving the codebase. If these characteristics suit your organization, monorepo would likely allow you to successfully support your projects. Many companies are using monorepo, specifically large organizations like Google and Facebook, where the investment in maintaining this paradigm is totally sustainable. One of the most famous [papers on monorepo](#) was written by Google's Rachel Potvin and Josh Levenberg. In their concluding paragraph, they write:

*Over the years, as the investment required to continue scaling the centralized repository grew, Google leadership occasionally considered whether it would make sense to move from the monolithic model. Despite the effort required, Google repeatedly chose to stick with the central repository due to its advantages.*

*The monolithic model of source code management is not for everyone. It is best suited to organizations like Google, with an open and collaborative culture. It would not work well for organizations where large parts of the codebase are private or hidden between groups.*

## Polyrepo

The opposite of a monorepo strategy is the polyrepo (see [Figure 5-2](#)), or multirepo, where every single application lives in its own repository.



Figure 5-2. Polyrepo example where we split the projects among multiple repositories

Some benefits of a polyrepo strategy are:

*Different branching strategy per project*

With monorepo, we should use trunk-based development, but with a polyrepo strategy, we can use the right branching strategy for the project we are working on. Imagine, for instance, that we have a legacy project with a different release cadence than other projects that are in continuous deployment. With a polyrepo strategy, we can use **Git flow**

in just that project, providing a branching strategy specific for that context.

#### *No risk of blocking other teams*

Another benefit of working with a polyrepo is that the blasting radius of our changes is strictly confined to our project. There isn't any possibility of breaking other teams' projects or negatively affecting them because they live in another repository.

#### *Encourages thinking about contracts*

In a polyrepo environment, the communication across projects has to be defined via APIs. This forces every team to identify the contracts between producers and consumers and to create governance for managing future releases and breaking changes.

#### *Fine-grained access control*

Large organizations are likely to work with contractors who should see only the repositories they are responsible for, or to have a security strategy in place where only certain departments can see and work on a specific area of the codebase. Polyrepo is the perfect strategy for achieving that fine-grained access control on different codebases, introducing roles, and identifying the level of access needed for every team or department.

#### *Less upfront and long-term investment in tooling*

With a polyrepo strategy, we can easily use any tool available out there. For instance, using managed solutions like CircleCI or Drone.io would be more than enough due to the contained size of a micro-frontend repository. Usually, the repositories are not expanding at the same rate as monorepo repositories because fewer developers are committing to the codebase. That means your investment up front and in maintaining the build of a polyrepo environment is far less, especially when you

automate your CI/CD pipeline using infrastructure as code or command-line scripts that can be reused across different teams.

Polyrepo also has some caveats:

#### *Difficulties with project discoverability*

By its nature, polyrepo makes it more difficult to discover other projects because every project is hosted in its own repository. Creating a solid naming convention that allows every developer to discover other projects easily can help mitigate this issue. Unquestionably, however, polyrepo can make it difficult for new employees or for developers who are comparing different approaches to find other projects suitable for their research.

#### *Code duplication*

Another disadvantage of polyrepo is code duplication. For example, a team creates a library that will be used by other teams for standardizing certain approaches, but the tech department is not aware of that library. Often, there are libraries that should be used across several micro-frontends, like logging or observability integration, but a polyrepo strategy doesn't facilitate code sharing if there isn't good governance in place. It's helpful, then, to identify the common aspects that may be beneficial for every team and coordinate the code-sharing effort across teams. Architects and tech leaders are in the perfect position to do this, since they work with multiple teams and have a high-level picture of how the system works and what it requires.

#### *Naming convention*

In polyrepo environments, I've often seen a proliferation of names without any specific convention; this quickly compounds the issue of tracking what is available and where. Regulating a naming convention for every repository is critical in a polyrepo system because working with micro-frontends, and maybe with microservices as well, could

result in a huge number of repositories inside our version control system.

### *Best practice maintenance*

In a monorepo environment, we have just one repository to maintain and control. In a polyrepo environment, it may take a while before every repository is in line with a newly defined best practice. Again, communication and process may mitigate this problem, but polyrepo requires you to think this through up front because finding out these problems during development will slow down your team's throughput.

Polyrepo is definitely a viable option for micro-frontends, though we risk having a proliferation of repositories. This complexity should be handled with clear and strong governance around naming conventions, repository discoverability, and processes. Micro-frontend projects with a vertical split have far fewer issues using polyrepo than those with a horizontal split, where our application is composed of tens, if not hundreds, of different parts. In the context of micro-frontends, polyrepo also makes it possible to use different approaches from a legacy project. In fact, we may introduce new tools or libraries just for the micro-frontend approach while keeping the same one for the legacy project without the need to pollute the best practices in place in the legacy platform. This flexibility has to be gauged against potential communication overhead and governance that has to be defined inside the organization; therefore, if you decide to use polyrepo, be aware of where your initial investment should be: communication flows across teams and governance.

## **A Possible Future for a Version Control System**

Any of the different paths we can take with a version control system won't be a perfect solution, just the solution that works better in our context. It's always a trade-off. However, we may want to try a hybrid approach (see [Figure 5-3](#)), where we can minimize the pitfalls of both approaches and leverage their benefits. Because micro-frontends and microservices should

be designed using domain-driven design, we may follow the subdomain and bounded context divisions for bundling all the projects that are included on a specific subdomain.

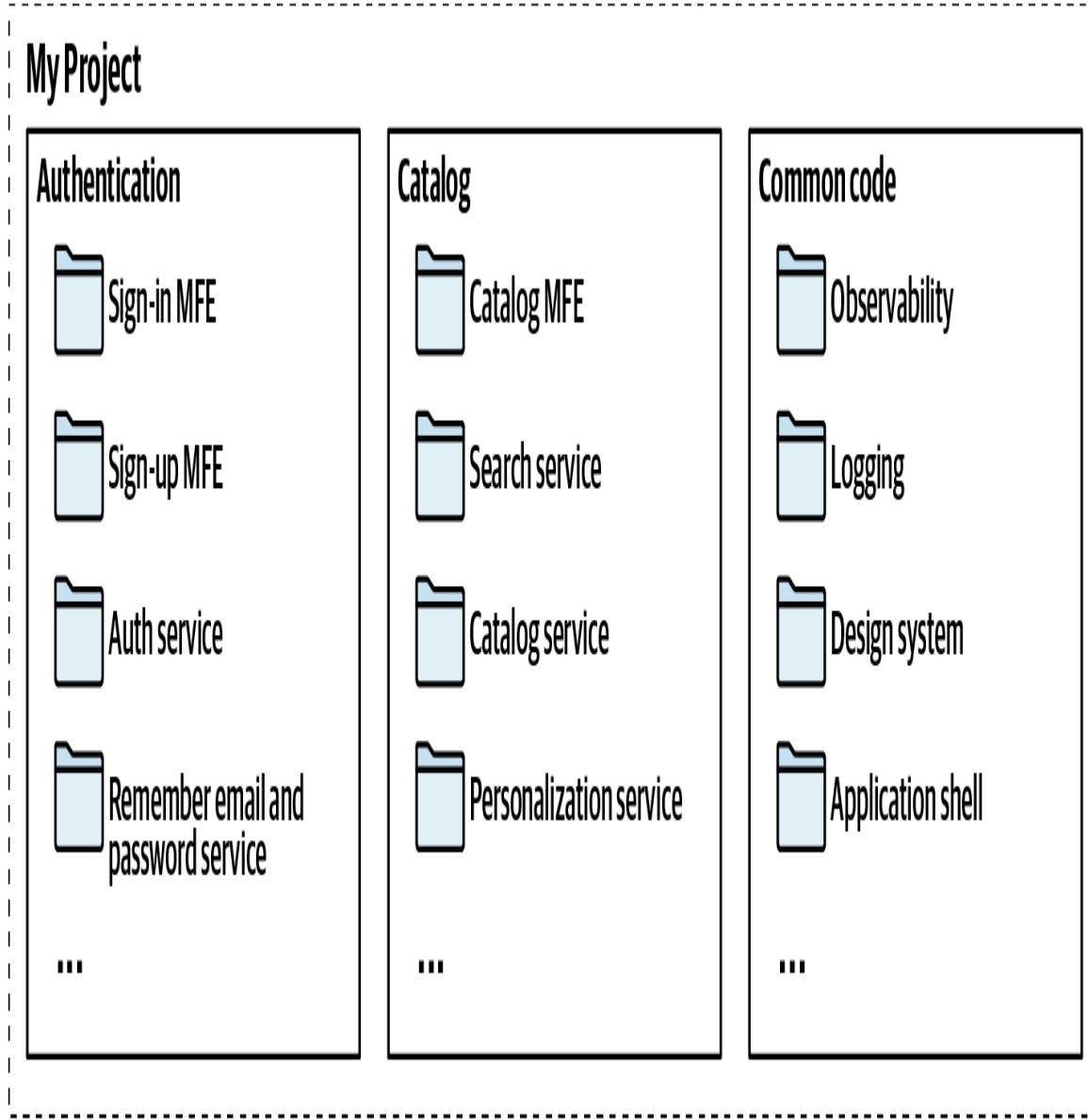


Figure 5-3. A hybrid repositories approach, where we can combine monorepo and polyrepo strengths in a unique solution

In this way, we can enforce the collaboration across teams responsible for different bounded contexts and work with contracts while benefiting from monorepo's strengths across all the teams working in the same subdomain. This approach might result in new practices, new tools to use or build, and

new challenges, but it's an interesting solution worth exploring for microarchitectures.

## Continuous Integration Strategies

After identifying the version control strategy, we have to think about the continuous integration method. Different CI implementations in different companies are the most successful and effective when owned by the developer teams rather than by an external guardian of the CI machines.

A lot of things have changed in the past few years. For one thing, developers, including frontend developers, have to become more aware of the infrastructure and tools needed for running their code because, in reality, building the application code in a reliable and quick pipeline is part of their job. In the past, I've seen many situations where the CI was delegated to other teams in the company, denying the developers a chance to change anything in the CI pipeline. As a result, the developers treated the automation pipeline as a black box—impossible to change but needed for deploying their artifacts to an environment. More recently, thanks to the DevOps culture spreading across organizations, these situations are becoming increasingly rare.

### ABOUT DEVOPS

DevOps is the combination of cultural philosophies, practices, and tools that increases an organization's ability to deliver applications and services at high velocity. Under a DevOps model, development and operations teams are no longer siloed. Sometimes, they're merged into a single team, where the engineers work across the entire application life cycle, from development and test to deployment and operations, and develop a range of skills that aren't limited to a single function.

Nowadays, many companies are giving developers ownership of automation pipelines. That doesn't mean developers should be entitled to do whatever they want in the CI, but they definitely should have some skin in

the game because how fast the feedback loop is closed depends mainly on them. The tech leadership team (architects, platform team, DX, tech leaders, engineers, managers, and so on) should provide the guidelines and the tools where the teams operate, while also providing certain flexibility inside those defined boundaries.

In a micro-frontend architecture, the CI is even more important because of the number of independent artifacts we need to build and deploy reliably. The developers, however, are responsible for running the automation strategy for their micro-frontends, using the right tool for the right job. This approach may seem like overkill considering that every micro-frontend may use a different set of tools. However, we usually end up having a couple of tools that perform similar tasks, and this approach also allows a healthy comparison of tools and approaches, helping teams to develop best practices.

More than once I would be walking the corridors and overhear conversations between engineers about how building tools like Rollup have some features or performances that the webpack tool didn't have in certain scenarios and vice versa. This, for me, is a sign of a great confrontation between tools tested in real scenarios rather than in a sandbox.

It's also important to recognize that there isn't a unique CI implementation for micro-frontends; a lot depends on the project, company standards, and the architectural approach. For instance, when implementing micro-frontends with a vertical split, all the stages of a CI pipeline would resemble normal SPA stages. End-to-end testing may be done before the deployment, if the automation strategy allows the creation of on-demand environments, and after the test is completed, the environment can be turned off. However, a horizontal split would require more thought on the right moment for performing a specific task. When performing end-to-end testing, we'd have to perform this phase in staging or production; otherwise, every single pipeline would need to be aware of the entire composition of an application, retrieving every latest version of the micro-frontends and pushing to an ephemeral environment—a solution very hard to maintain and evolve.

# Testing Micro-Frontends

Plenty of books discuss the importance of testing our code and catching bugs or defects as early as possible, and the micro-frontends approach is no different.

## TESTING STRATEGIES

I won't cover all the different possible testing strategies, such as unit testing, integration testing, or end-to-end testing. Instead, I'll cover the differences from a standard approach we are used to implementing in any frontend architecture. If you would like to become more familiar with different testing strategies, I recommend studying the materials shared by incredible authors like Kent Beck or Robert C. Martin (a.k.a. Uncle Bob), especially Beck's *Test-Driven Development: By Example* (Addison-Wesley Professional) and Martin's *Clean Code* (Pearson).

Working with micro-frontends doesn't mean changing the way we are dealing with frontend testing practices, but they do create additional complexity in the CI pipeline when we perform end-to-end testing. Since unit testing and integration testing are not changing compared to other frontend architectures, we'll focus here on end-to-end testing, as this is the biggest challenge for testing micro-frontends.

## End-to-end testing

End-to-end testing is used to test whether the flow of an application from start to finish is behaving as expected. We perform tests to identify system dependencies and ensure that data integrity is maintained between various system components and systems. End-to-end testing may be performed before deploying our artifacts in production in an on-demand environment created at runtime just before tearing the environment down. Alternatively, when we don't have this capability in-house, we should perform end-to-end tests in existing environments after the deployment or promotion of a new artifact. In this case, the recommendation would be embracing testing in

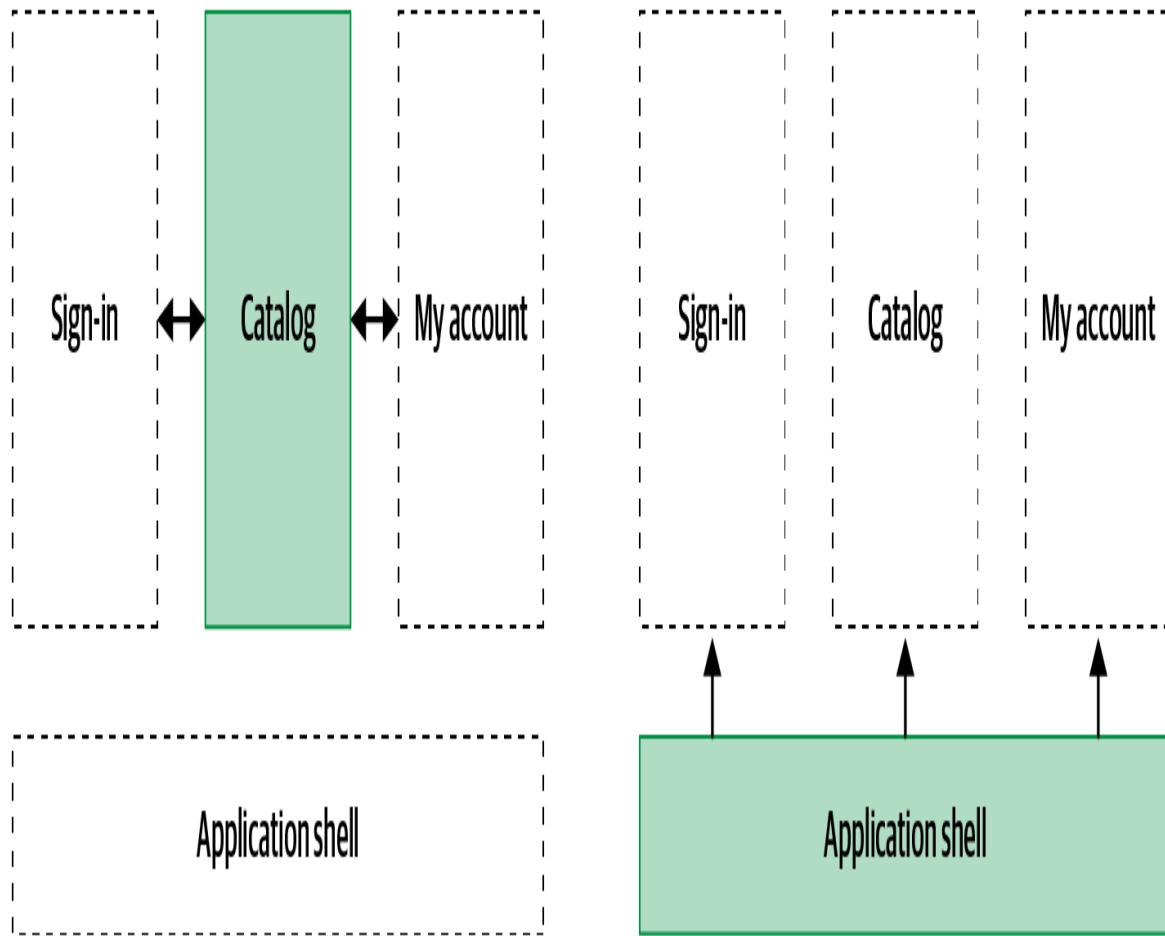
production when the application has implemented feature flags, allowing toggling of a feature on and off and granting access to test for a set of users.

Testing in production brings its own challenges, especially when a system is integrating with third-party APIs. However, it will save a lot of money on environment infrastructure, maintenance, and developers' resources because we don't have to configure and maintain multiple environments simultaneously. I'm conscious not all companies or projects are suitable for this practice; therefore, using the environments you have available is the last resort. When you start a new project or you have the possibility to change an existing one, take into consideration the possibility of introducing feature flags not only for reducing the risk of bugs in front of users but also for testing purposes.

Finally, some of the complexity brought in by micro-frontends may be mitigated with some good coordination across teams and solid governance overarching the testing process. As discussed multiple times in this book, the complexity of end-to-end testing varies depending on whether we embrace a horizontal or vertical split for our application.

## **Vertical-split end-to-end testing challenges**

When we work with a vertical split, one team is responsible for an entire business subdomain of the application. In this case, testing all the logic paths inside the subdomain is not far from what you would do in an SPA. But we have some challenges to overcome when we need test use cases outside of the team's control, such as the scenario in [Figure 5-4](#).



*Figure 5-4. An end-to-end testing example with a vertical split architecture*

The catalog team is responsible for testing all the scenarios related to the catalog. However, some scenarios involve areas not controlled by the catalog team, like when the user signs out from the application and should be redirected to the “Sign-in” micro-frontend or when a user wants to change something in their profile and should be redirected to the “My account” micro-frontend. In these scenarios, the catalog team will be responsible for writing tests that cross their domain boundary and ensuring that the specific micro-frontend the user should be redirected to loads correctly. In the same way, the teams responsible for the “Sign-in” and “My account” micro-frontends will need to test their business domain and verify that the catalog correctly loads as the user expects.

Another challenge is making sure our application behaves in cases of deep-linking requests or when we want to test different routing scenarios. It

always depends on how we have designed our routing strategy, but let's take the example of having the routing logic in the application shell, as in [Figure 5-4](#). The application shell team should be responsible for these tests, ensuring that the entire route of the application loads correctly, that the key behaviors like signing in or out work as expected, and that the application shell is capable of loading the right micro-frontend when a user requests a specific URL.

## Horizontal-split end-to-end testing challenges

Using a horizontal split architecture raises the question of who is responsible for end-to-end testing of the final solution. Technically speaking, what we have discussed for the vertical-split architecture still stands, but we have a new level of complexity to manage. For instance, if a team is responsible for a micro-frontend present in multiple views, is the team responsible for end-to-end testing all the scenarios where their micro-frontends are present? Let's try to shed some light on this with the example in [Figure 5-5](#).

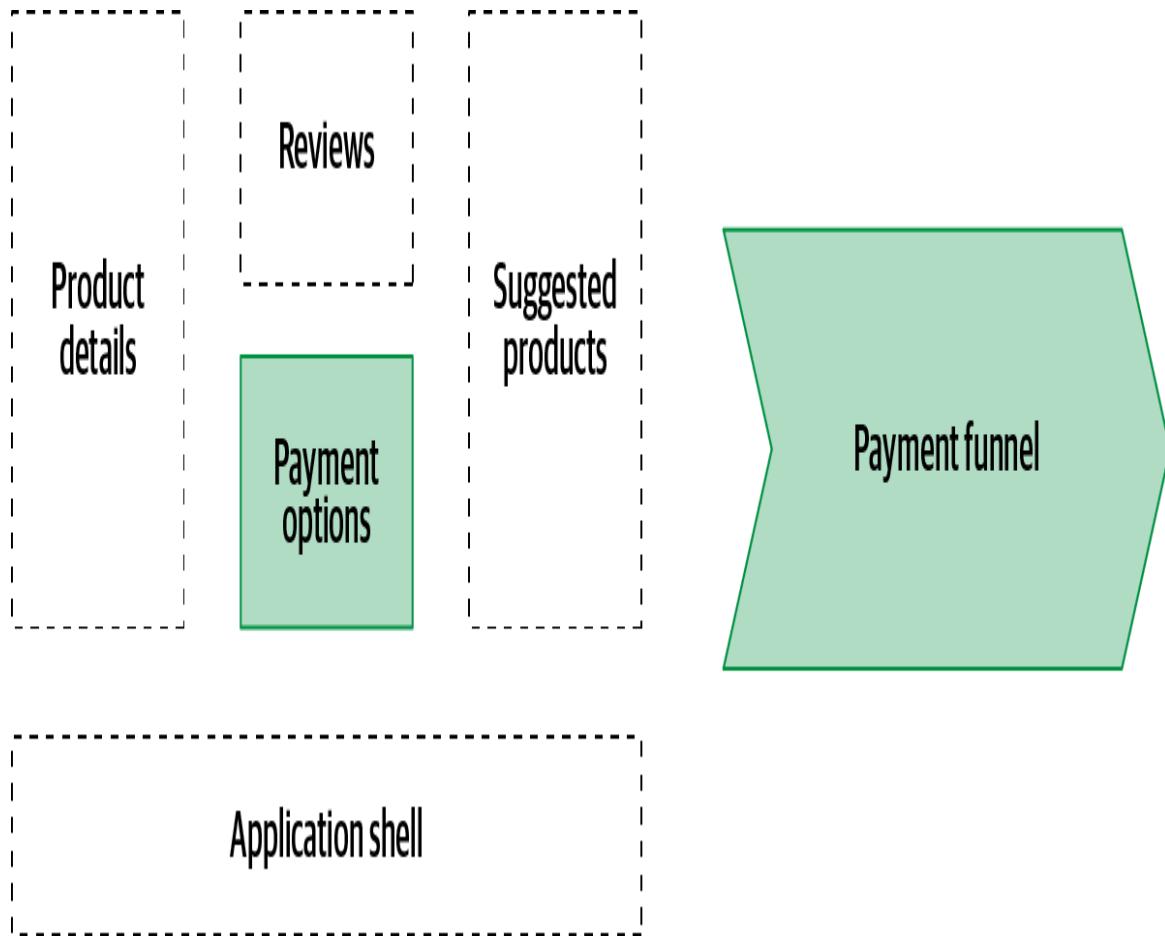


Figure 5-5. An end-to-end testing example with a horizontal-split architecture

The payments team is responsible for providing all the micro-frontends needed for performing a payment inside the project. In the horizontal-split architecture, their micro-frontends are available on multiple views. In **Figure 5-5**, we can see the payment option micro-frontend that will lead the user to choose a payment method and finalize the payment when they're ready to check out. Therefore, the payment team is responsible for making sure the user will be able to pick a payment option and finalize the checkout, showing the interface needed for performing the monetary transaction with the selected payment option in the following view. In this case, the payment team can perform the end-to-end test for this scenario, but we will need a lot of coordination and governance for analyzing all the end-to-end tests needed and then assigning them to the right teams to avoid duplication of intent, which is even more difficult to maintain in the long run.

The end-to-end tests also become more complex to maintain since different teams are contributing to the final output of a view, and some tests may become invalid or broken the moment other teams change their micro-frontends. That doesn't mean we aren't capable of doing end-to-end testing with a horizontal split, but it does require better organization and more thought before implementing.

## Testing technical recommendations

For technically implementing end-to-end tests successfully for horizontal- or vertical-split architecture, we have three main possibilities. The first one is running all the end-to-end tests in a stable environment where all the micro-frontends are present. We delay the feedback loop if our new micro-frontend works as expected, end to end.

Another option is using on-demand environments where we pull together all the resources needed for testing our scenarios. This option may become complicated in a large application, however, especially when we use a horizontal-split architecture. This option may also cost us a lot when it's not properly configured (as described earlier).

Finally, we may decide to use a proxy server that will allow us to end-to-end test the micro-frontend we are responsible for. When we need to use any other part of the application involved in a test, we'll just load the parts needed from an environment, either staging or production—in this case, the micro-frontends and the application shell not developed by our team.

In this way, we can reduce the risk of unstable versions or optimization for generating an on-demand environment. The team responsible for the end-to-end testing won't have any external dependency to manage, either, but they will be completely able to test all the scenarios needed for ensuring the quality of their micro-frontend.

## WEBPACK DEV SERVER PROXY CONFIGURATION

For completeness of information, webpack, as with many other building tools, allows us to configure a proxy server that retrieves external resources from specific URLs, like static files, or even consume APIs in a specific environment. This feature may be useful for setting up our end-to-end testing in a scenario where we want to run it during the CI pipeline. The configuration is trivial, as you can see in the following example:

```
// In webpack.config.js
{
  devServer: {
    proxy: {
      '/catalog-mfe': {
        target: 'https://other-server.example.com',
        secure: false
      }
    }
  }
}
// Multiple entry
proxy: [
  {
    context: ['/catalog-mfe/**', '/myaccount-mfe/**'],
    target: 'https://other-server.example.com',
    secure: false
  }
]
```

Additional information about the webpack proxy setup is available in the [webpack documentation](#).

When the tool used for running the automation pipeline allows it, you can also set up your CI to run multiple tests in parallel instead of in sequence. This will speed up the results of your tests specifically when you are running many of them at once; it can also be split in parts and grouped in a sensible manner. If we have a thousand unit tests to run, for example, splitting the effort into multiple machines or containers may save us time

and get us results faster. This technique may also be applied to other stages of our CI pipeline. With just a little extra configuration by the development team, you can save time testing your code and gain confidence in it sooner. Even tools that work well for us can be improved, and systems evolve over time. Be sure to analyze your tools and any potential alternatives regularly to ensure you have the best CI tools for your purposes.

## Fitness Functions

Considering the inherent complexity of distributed systems where multiple modules make up an entire platform, the architecture team should have a way to measure the impact of their architecture decisions and make sure these decisions are followed by all teams, whether they are colocated or distributed. In the book *Building Evolutionary Architecture* (O'Reilly), Neal Ford, Rebecca Parsons, and Patrick Kua discuss how to test an architecture's characteristics in CI with fitness functions. They state that a fitness function "provides an objective integrity assessment of some architectural characteristic(s)."

Many of the steps defined inside an automation pipeline are used to assess architecture characteristics such as static analyses in the shape of cyclomatic complexity or the bundle size in the micro-frontend use case. Having a fitness function that assesses the bundle size of a micro-frontend is a good idea when a key characteristic of your micro-frontend architecture is the size of the data downloaded by users. The architecture team may decide to introduce fitness functions inside the automation strategy, guaranteeing the agreed-upon outcome and trade-off that a micro-frontend application should have. Here are some key architecture characteristics to pay attention to when designing the automation pipeline for a micro-frontend project:

### *Bundle size*

Allocate a budget size per micro-frontend and analyze when this budget is exceeded and why. In the case of shared libraries, also review the size of all the libraries shared, not only the ones built with micro-frontends.

## *Performance metrics*

Tools like Lighthouse and WebPageTest allow us to validate whether a new version of our application has the same or higher standards than the current version.

## *Static analysis*

There are plenty of tools for static analysis in the JavaScript ecosystem, with SonarQube probably being the most well-known. Implemented inside an automation pipeline, this tool will provide us insights such as the cyclomatic complexity of a project (in our case, a micro-frontend). We may also want to enforce a high code-quality bar when setting a cyclomatic complexity threshold over which we don't allow the pipeline to finish until the code is refactored.

## *Code coverage*

Another example of a fitness function is making sure our codebase is tested extensively. Code coverage provides a percentage of tests run against our project, but bear in mind that this metric doesn't provide us with the quality of the test, just a snapshot of tests written for public functions.

## *Security*

Finally, we want to ensure our code won't violate any regulations or rules defined by the security or architecture teams.

These are some architecture characteristics that we may want to test in our automation strategy when we work with micro-frontends. In a distributed architecture like this one, these metrics become fundamental for architects and tech leads to understand the quality of the product developed, to understand where the tech debt lies, and to enforce key architecture characteristics without having to chase every team or be part of any feature development.

Introducing and maintaining fitness functions inside the automation strategy will provide several benefits for helping the team provide a fast feedback loop on architecture characteristics and helping the company achieve better code quality standards.

## Micro-Frontend-Specific Operations

Some automation pipelines for micro-frontends may require additional steps compared to traditional frontend automation pipelines. The first one worth mentioning would be checking that every micro-frontend is integrating specific libraries flagged as mandatory for every frontend artifact by the architecture team. Let's assume that we have developed a design system and we want to enforce that all our artifacts must contain the latest major version. In the CI pipeline, we should have a step for verifying the *package.json* file, making sure the design system library contains the right version. If it doesn't, it should notify the team or even block the build, failing the process.

The same approach may be feasible for other internal libraries we want to make sure are present in every micro-frontend, like analytics and observability. Considering the modular nature of micro-frontends, this additional step is highly recommended—no matter the architecture style we decide to embrace in this paradigm—for guaranteeing the integrity of our artifacts across the entire organization.

Another interesting approach, mainly available for vertical-split architecture, is the possibility of a server-side render at compile time instead of runtime when a user requests the page. The main reason for doing this is saving computation resources and costs, such as when we have to merge data and user interfaces that don't change very often. Another reason is to provide a highly optimized and fast-loading page with inline CSS and maybe even some JavaScript.

When our micro-frontend artifact results in an SPA with an HTML page as the entry point, we can generate a page skeleton with minimal CSS and HTML nodes inlined to suggest how a page would look, providing

immediate feedback to the user while we are loading the rest of the resources needed for interacting with micro-frontends. This isn't an extensive list of possibilities an organization may want to evaluate for micro-frontends, because every organization has its own gotchas and requirements. However, these are all valuable approaches that are worth thinking about when we are designing an automation pipeline.

## Observability

The last important part to take into consideration in a successful micro-frontend architecture is the observability of our micro-frontends. Moreover, observability closes the feedback loop when our code runs in a production environment; otherwise, we would not be able to react quickly to any incidents happening during prime time. In the last few years, many observability tools started to appear for the frontend ecosystem, such as Sentry, New Relic, or LogRocket. These tools allow us to individuate the user journey before encountering a bug that may or may not prevent the user from completing the action. Observability is a must-have feature; nowadays, it should be part of any releasing strategy and is even more important when we are implementing distributed architectures such as micro-frontends.

Every micro-frontend should report errors—custom and generic—for providing visibility when a live issue happens. In that regard, Sentry, New Relic, or LogRocket can help in this task by providing the visibility needed. In fact, these tools retrieve the user journey, collect the JavaScript stack trace of an exception, and cluster into groups. We can configure the alerting of every type of error or warning in these tools' dashboards and even plug these tools with alerting systems like PagerDuty.

It's very important to think about observability at a very early stage of the process because it plays a fundamental role in closing the feedback loop for developers, especially when we are dealing with multiple micro-frontends composing the same view. These tools will help us to debug and understand in which part of our codebase the problem is happening and quickly drive a team to the resolution, providing some user's context information like

browser used, operating system, user's country, and so on. All this information, in combination with the stack trace, provides a clear investigation path for any developer to resolve the problem without spending hours trying to reproduce a bug in the developer's machine or in a testing environment.

## Summary

We've covered a lot of ground here, so a recap is in order. First, we defined the principles we want to achieve with automation pipelines, focusing on fast feedback and constant review based on the evolution of both tech and the company. Then we talked about the developer experience. If we aren't able to provide a frictionless experience, developers may try to game the system or use it only when it's strictly necessary, reducing the benefits they can have with a well-designed CI/CD pipeline. We next discussed implementing the automation strategy, including all the best practices, such as unit, integration, and end-to-end testing; bundle-size checks; fitness functions; and many others that could be implemented in our automation strategy for guiding developers toward the right software quality.

Automation is a very interesting topic, especially when we are implementing microarchitectures. There are plenty of additional topics we could cover, but if you are capable of covering these inside your automation strategy, you will be in really great shape, and you can always extend and evolve based on the business and technical needs. The main takeaway is that automation is not a one-off action but an iterative process that has to be reviewed and improved with the life cycle of a product.

In the next chapter, we will discuss micro-frontends deployment and discoverability, another key aspect of de-risking deployments by multiple independent teams in distributed systems.

# Chapter 6. Automation Pipeline for Micro-Frontends: A Case Study

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the eighth chapter of the final book. Please note that the GitHub repo will be made active later on.

If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the author at [building.microfrontends@gmail.com](mailto:building.microfrontends@gmail.com).

Now that we’ve discussed the theory of a micro-frontend automation pipeline, let’s review a use case example, including the different steps that should be taken into consideration based on the topics we covered. Let’s keep in mind that not all the steps or the configuration described in this example have to be present in every automation strategy, because companies and projects are different.

## Setting the Scene

ACME Inc., a video-streaming service, empowers its developers and trusts them to know better than anyone else in the organization which tools they should use for building the micro-frontends needed for the project. Every team is responsible for setting up a micro-frontend build, so the developers

are encouraged to choose the tools needed based on the technical needs of micro-frontends and on some boundaries, or guardrails, defined by the company.

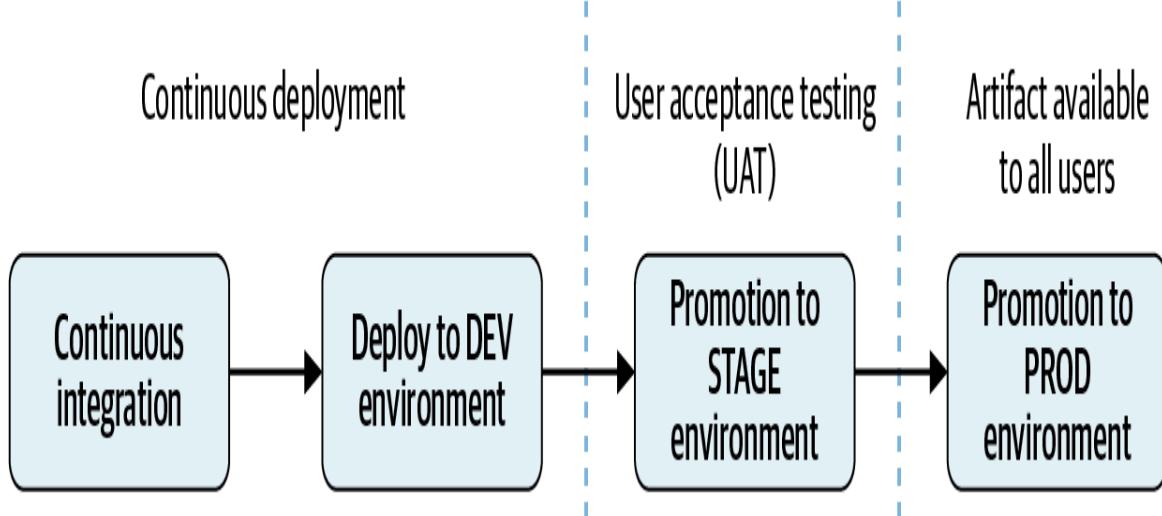
The company uses a custom cloud automation pipeline based on docker containers, and the cloud team provides the tools needed for running these pipelines. The project is structured using micro-frontends with a vertical-split architecture, where micro-frontends are technically represented by an HTML page, a JavaScript file, and a CSS file. Every development team in the organization works with unit, integration, and end-to-end testing, a decision made by the tech leaders and the head of engineering to ensure the quality and reliability of code deployed in production.

The architecture team, which is the bridge between product and engineers, requests using fitness functions within the pipeline to ensure the artifacts delivered in the production environment contain the architecture characteristics they desire. The team will be responsible for translating product people's business requirements to technical ones the techies can create.

The development teams decide to use a monorepo strategy, so all the micro-frontends will be present in the same repository. The team will use trunk-based development for its branching strategy and release directly from the main branch instead of creating a release branch.

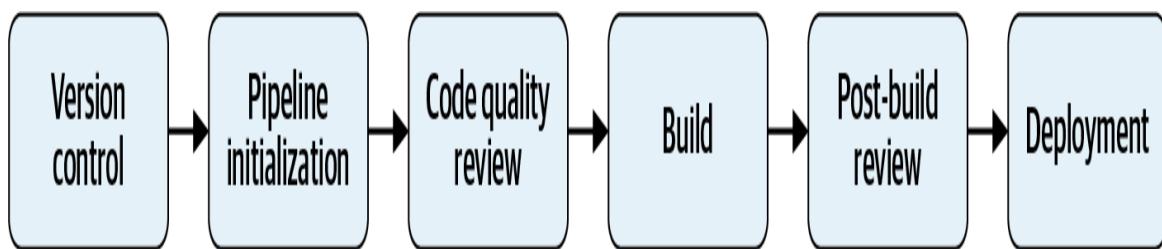
The project won't use feature flags. The team decides to defer this decision for having fewer moving parts to take care of, so manual and automated testing will be performed in existing environments already created by the DX team.

Finally, for bug fixing, the teams will use a fix-forward strategy, where they will fix bugs in the trunk branch and then deploy. The environment strategy present in the company is composed of three environments: development (DEV), staging (STAGE), and production (PROD), as we can see in [Figure 6-1](#).



*Figure 6-1. An example of an environments strategy*

The DEV environment is in continuous deployment so that the developers can see the results of their implementations as quickly as possible. When a team feels ready to move to the next step, it can promote the artifact to user acceptance testing (UAT). At this stage, the UAT team will make sure the artifact respects all the business requirements before promoting the artifact to production, where it will be consumed by the final user. Based on all this, [Figure 6-2](#) illustrates the automation strategy for our use case project up to the DEV environment. It's specifically designed for delivering the micro-frontends at the desired quality.



*Figure 6-2. High-level automation strategy design*

A dashboard built in-house will promote artifacts across environments. In this way, the developers and quality assurance have full control of the different steps for reviewing an artifact before it is presented to users. Such an automation strategy will create a constant, fast feedback loop for the developers, catching potential issues as soon as possible during the

continuous integration phase instead of further down the line, making the bug fixing as cheap as possible.

## DEFECT COSTS RISE OVER TIME

Remember, the cost of detecting and fixing defects in software increases exponentially over time in the software development workflow. That's because when a developer is working on a feature, the code developed is fresh in their mind; a code change is fairly trivial. When a developer catches bugs in production, months may have passed since the developer worked on that code. In the meantime, the developer will have worked on several other projects or features, so remembering the team's entire logic and approach will take time. Finding bugs in production costs you more than just time. It hurts the company's credibility and costs more money than just investing in a fast feedback loop at the beginning. The National Institute of Standards and Technology estimates the [cost of fixing bugs in production](#) to be 25 times more expensive than catching them during the development phase.

The automation strategy in this project is composed of six key areas, within which there are multiple steps:

1. Version control
2. Pipeline initialization
3. Code-quality review
4. Build
5. Post-build review
6. Deployment

Let's explore these areas in detail.

## Version Control

The project will use monorepo for version control, so the developers decided to use [Lerna](#), which enables them to manage all the different micro-frontend dependencies at the same time. Lerna also allows hoisting all the shared modules across projects in the same `node_modules` folder in the root directory, so that if a developer has to work on multiple projects, they can download a resource for multiple micro-frontends just once.

Dependencies will be shared, so a unique bundle can be downloaded once by a user and will have a high time-to-live time at CDN level. Considering the vendors aren't changing as often as the application's business logic, we'll avoid an increase of traffic to the origin.

ACME Inc. uses GitHub as a version control system, partially because there are always interesting automation opportunities in a cloud-based system like GitHub. In fact, [GitHub has a marketplace](#) with many scripts available to be run at different branching life cycles. For instance, we may want to apply linting rules at every commit or when someone is opening a pull request. We can also decide to run our own scripts if we have particular tasks to apply in our codebase during an opening of a pull request, like scanning the code to avoid any library secrets being presented or for other security reasons. Moreover, the core team decided to configure Dependabot for managing shared libraries across micro-frontends.

Dependabot is a critical automated dependency management tool that plays a vital role in maintaining the health and security of JavaScript applications, particularly within micro-frontend architectures. It streamlines the process of keeping dependencies up-to-date across multiple repositories or modules by regularly scanning package files to identify outdated or vulnerable dependencies. When it detects any issues, Dependabot automatically creates pull requests to update these dependencies, significantly reducing the manual effort required by developers. This automation not only enhances the security posture of the application by addressing vulnerabilities promptly but also allows development teams to focus on building features rather than getting bogged down in the tedious task of manual package updates.

In a micro-frontend architecture, Dependabot is especially valuable for managing shared dependencies. For instance, when a team responsible for a design system releases updates, Dependabot can automatically generate pull requests in all micro-frontends that utilize this shared library. This eliminates the need for manual notifications and coordination between teams, allowing for a smoother workflow. Similarly, Dependabot can apply automated updates to other shared libraries, such as analytics or observability tools, ensuring that all micro-frontends benefit from the latest versions.

## Pipeline Initialization

The pipeline initialization stage includes several common actions to perform for every micro-frontend, including:

- Cloning the micro-frontend repository inside a container
- Installing all the dependencies needed for the following steps

In [Figure 6-3](#), we can see the first part of our automation pipeline where we perform two key actions: cloning the micro-frontend repository and installing the dependencies via `yarn` or `npm` command, depending on each team's preference.

## Version control      Pipeline initialization

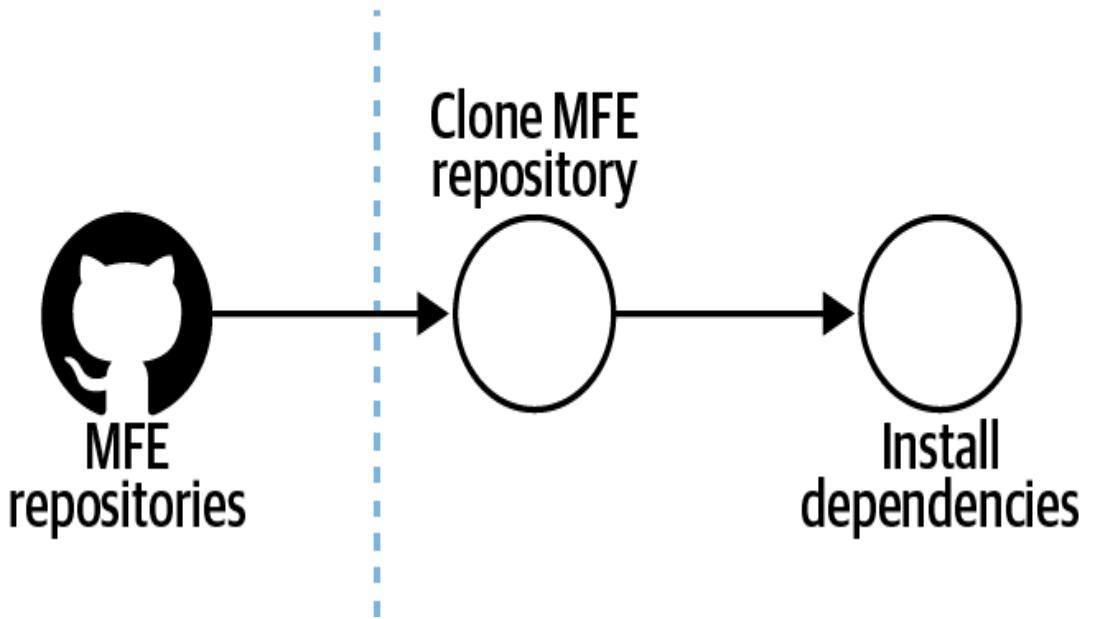


Figure 6-3. Pipeline initialization stage, showing two actions: cloning the repository and installing the dependencies

The most important thing to remember is to make the repository cloning as fast as possible. We don't need the entire repository history for a CI process, so it's a good practice to use the command depth for retrieving just the last commit, especially when we use a monorepo approach, considering the repository may grow in size very quickly. The cloning operation will speed up in particular when we are dealing with repositories with years of history tracked in the version control system:

```
git clone --depth [depth] [remote-url]
```

An example would be:

```
git clone --depth 1 https://github.com/account/repository
```

## Code-Quality Review

During this phase, we are performing all the checks to make sure the code implemented respects the company standards. [Figure 6-4](#) shows several

stages, from static analysis to visual tests. For this project, the company decided not only to cover unit and integration testing but also to ensure that the code was maintainable in the long term, the user interface integration respects the design guidelines from the UX team, and the common libraries developed are present inside the micro-frontends and respect the minimum implementations.

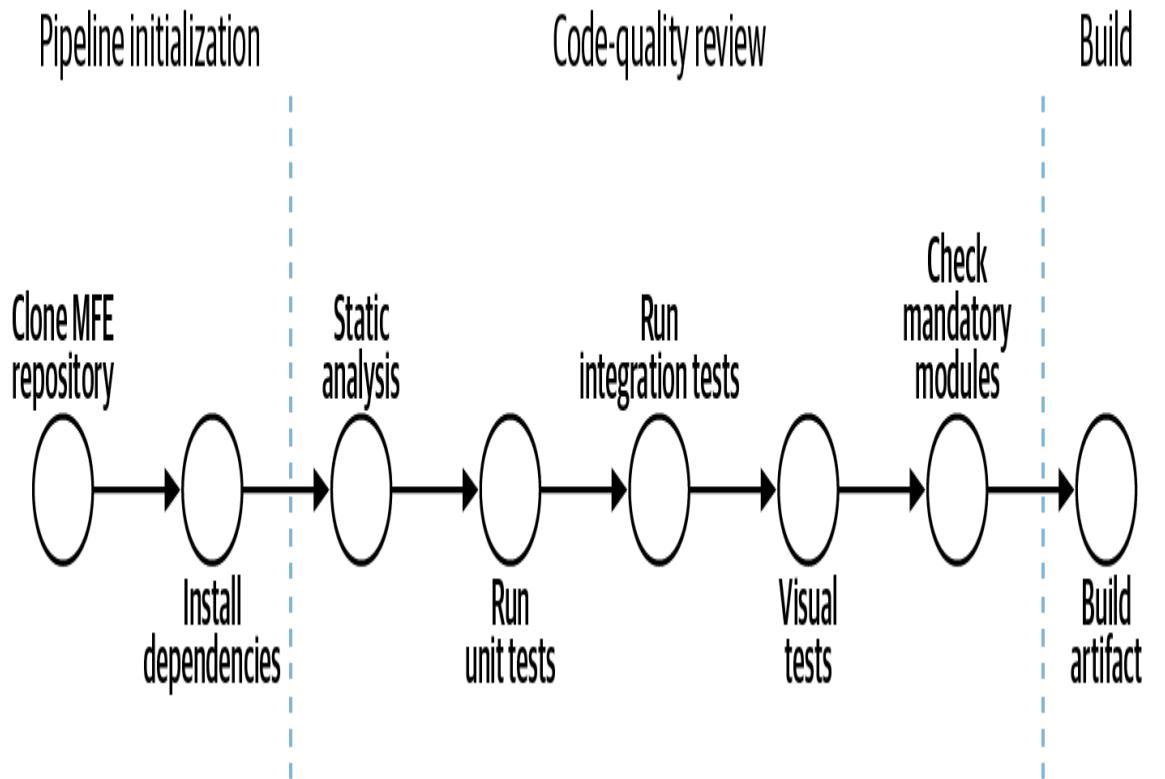


Figure 6-4. Code-quality checks like unit testing, static analysis, and visual regression tests

For static analysis, ACME Inc. uses **SonarQube** with the JavaScript plug-in. SonarQube is a tool for static analysis, and it retrieves many metrics, including cyclomatic complexity (CYC), which tech leaders and architects who aren't working every day in the codebase need in order to understand the code quality produced by a team. Often underestimated, CYC can provide a lot of useful information about how healthy your project is. It provides a score on the code complexity based on the number of branches inside every function, which is an objective way to understand if the micro-frontend is simple to read but harder to maintain in the long run.

Let's consider this example:

```
const myFunc = (someValue) =>{
    // variable definitions

    if(someValue === "1234-5678"){ //CYC: 1 - first branch
        // do something
    } else if(someValue === "9876-5432"){ //CYC: 2 - second branch
        // do something else
    } else { //CYC: 3 - third branch
        // default case
    }

    // return something
}
```

This function has a CYC score of 3, which means we will need at least three unit tests for this function. It may also indicate that the logic managed inside the function starts to become complex and harder to maintain.

By comparison, a CYC score of 10 means a function definitely requires some refactoring and simplification; we want to keep our CYC score as low as possible so that any change to the code will be easier for us but also for other developers inside or outside our team.

Unit and integration testing are becoming more important every day, and the tools for JavaScript are becoming better. Developers, as well as their companies, must recognize the importance of automated testing before deploying in production. With micro-frontends, we should invest in these practices mainly because the area to test per team is far smaller than a normal single-page application and the related complexity should be lower. Considering the size of the business logic as well, testing micro-frontends should be very quick. There aren't any excuses for avoiding this step.

ACME Inc. decided to use [Jest](#) for unit and integration testing, which is standard within the company. Since there isn't a specific tool for testing micro-frontends, the company's standard tool will be fine for unit and integration tests.

The final step is specific to a micro-frontend architecture: checking on implementing specific libraries, like logging or observability, across all the micro-frontends inside a project. When we develop a micro-frontend

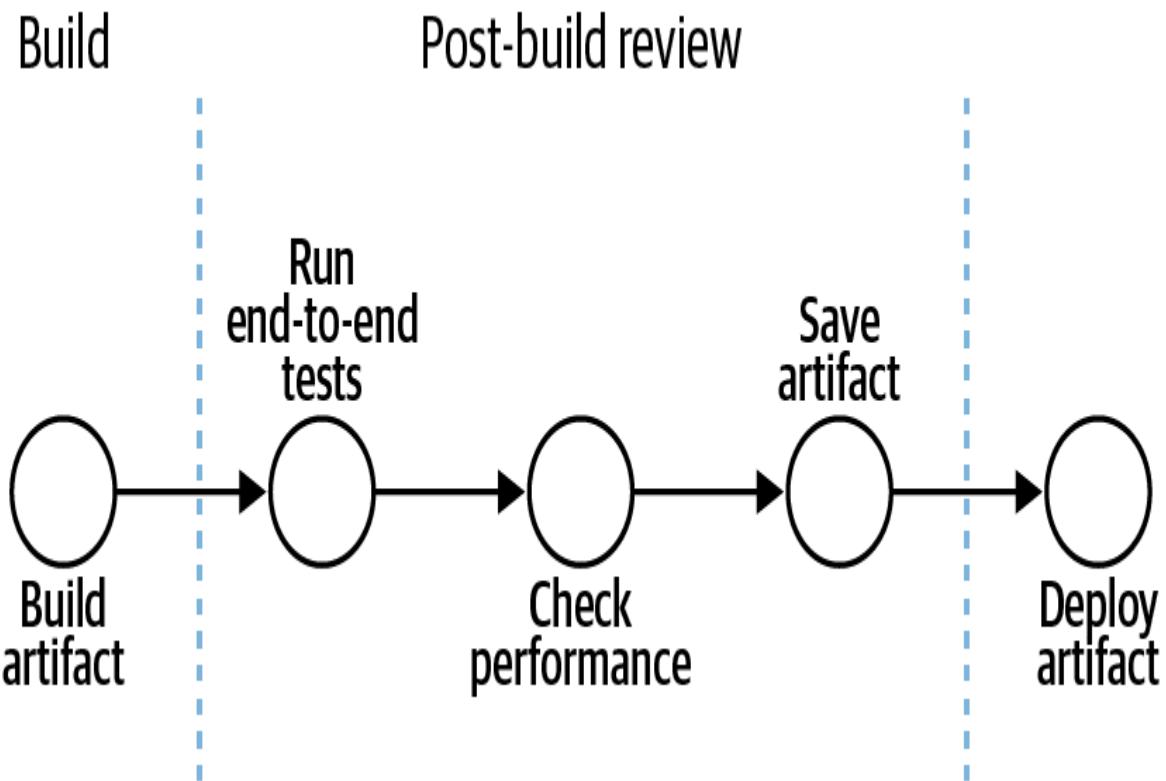
application, there are some parts we want to write once and put in all our micro-frontends. A check on the libraries present in every micro-frontend will help enforce these controls, making sure that all the micro-frontends respect the company's guidelines and we aren't reinventing the wheel. Controlling the presence inside the `package.json` file present in every JavaScript project is a simple way to do this; however, we can go a step further by implementing more complex reviews, like library versions, analysis on the implementation, and so on. It's very important to customize an automation pipeline introducing these kinds of fitness functions to ensure the architectural decisions are respected despite the nature of this architecture. Moreover, with micro-frontends where sharing code across them may result in way more coordination than a monolithic codebase, these kinds of steps are fundamental for having a positive end result.

## Build

The artifact is created during the build stage. For this project, the teams are using [webpack](#) for performing any code optimizations (like minification and dead code elimination). Micro-frontends allow us to use different tools for building our code; in fact, it may be normal to use webpack for building and optimizing certain micro-frontends and using another tool for others. The important thing to remember is to provide freedom to the teams inside certain boundaries. If you have any particular requirements that should be applied at build time, raise them with the teams and make sure when a new tool is introduced inside the build phase—and generally inside the automation pipeline—it has the capabilities required for maintaining the boundaries. Introducing a new build tool is not a problem per se, because we can experiment and compare the results from the teams. We may even discover new capabilities and techniques we wouldn't find otherwise. Yet we don't *have* to use different tools. It's perfectly fine if all the teams agree on a set of tools to use across the entire automation pipeline; however, don't block innovation. Sometimes we discover interesting results from an approach different from the one agreed to at the beginning of the project.

## Post-Build Review

The post-build stage (shown in [Figure 6-5](#)) is the last opportunity to confirm our artifact has all the performance characteristics and requirements ready to be deployed in production.



*Figure 6-5. In the post-build review, we perform additional checks before deploying an artifact to an environment*

A key step is storing the artifact in an artifacts repository, like Nexus or Artifactory. You may also decide to use a simpler storage solution, like an Amazon Web Services (AWS) S3 bucket. The important thing is to have a unique source of truth where all your artifacts are stored.

ACME Inc. decided to introduce additional checks during this stage: end-to-end testing and performance review. Whether these two checks are performed at this stage depends on the automation strategy we have in place and the capability of the system. In this example, we are assuming that the company can spin up a static environment for running end-to-end testing and performance checks and then tear it down when these tests are completed.

End-to-end testing is critical for micro-frontends. In this case where we have a vertical split and the entire user experience is inside the same artifact, testing the entire micro-frontend like we usually do for single-page applications is natural. However, if we have multiple micro-frontends in the same view with a horizontal split, we should postpone end-to-end testing to a later stage in order to test the entire view.

When we cannot afford to create and maintain on-demand environments, we might use web servers that are proxying the parts not related to a micro-frontend. For instance, webpack's dev server plug-in can be configured to fetch all the resources requested by an application during end-to-end tests locally or remotely, specifying from which environment to pull the resources when not related to the build artifact. If a micro-frontend is used in multiple views, we should check whether the code will work end to end in every view the micro-frontend is used.

Although end-to-end testing is becoming more popular in frontend development, there are several schools of thought about when to perform the test. You may decide to test in production—as long as all the features needed to sustain testing in that environment are present. Therefore, be sure to include feature flags, potential mock data, and coordination when integrating with third parties to avoid unexpected and undesirable side effects.

Performance checks have become far easier to perform within an automation pipeline, thanks to command-line interface (CLI) tools now being available to be wrapped inside a docker container and being easy to integrate into any automation pipeline. There are many alternatives, however. I recommend starting with [Lighthouse](#) CLI or [webhint](#) CLI. The former is a well-known tool created by Google and present even in Chrome browser, while the latter allows us to create additional performance tests for enhancing the list of tests already available by default.

With one of these two solutions implemented in our automation strategy, we can make sure our artifact respects key metrics, like performance, accessibility, and best practices. Ideally, we should be able to gather these

metrics for every artifact in order to compare them during the lifespan of the project. In this way, we can review the improvements and regressions of our micro-frontends and organize meetings with the tech leadership for analyzing the results and determining potential improvements, creating a continuous learning environment inside our organization.

With these steps implemented, we make sure our micro-frontends deployed in production are functioning (through end-to-end testing) and performing as expected when the architectural characteristics are identified.

## Deployment

The last step in our example is the deployment of a micro-frontend. An AWS S3 bucket will serve as the final platform to the user, and Cloudfront will be our CDN. As a result, the CDN layer will take the traffic hit, and there won't be any scalability issues to take care of in production, despite the shape of user traffic that may hit the web platform. An AWS Lambda—an event-driven serverless computing platform provided by Amazon as a part of Amazon Web Services—will be triggered to decompress the *tar.gz* file present in the artifacts repository, and then the content will be deployed inside the dev environment bucket. Remember that the company built a deployment dashboard for promoting the artifacts through different environments. In this case, for every promotion, the dashboard triggers an AWS Lambda for copying the files from one environment to another.

ACME Inc. decided to create a very simple infrastructure for hosting its micro-frontends, neatly avoiding additional investments in order to understand how to scale the additional infrastructure needed for serving micro-frontends. Obviously, this is not always the case. But I encourage you to find the cheapest, easiest way for hosting and maintaining your micro-frontends. You'll remove some complexities to be handled in production and have fewer moving parts that may fail.

To mitigate risks associated with large-scale deployments and potential bugs in new artifacts, ACME teams implemented a micro-frontends discovery pattern. This strategic move, as discussed in Chapter 9, provides a

robust mechanism for managing the rollout of new micro-frontends. After installing the **Frontend Discovery Service**, the teams began utilizing its API to create, update, and delete micro-frontend deployments. This service is designed to incrementally increase traffic to new versions based on a predefined deployment strategy, typically over a specified time interval.

The gradual rollout facilitated by the Frontend Discovery Service allows for a controlled and monitored deployment process. Developers can observe their dashboards to quickly identify any issues arising from the new version in production. If problems are detected, the system enables rapid rollback capabilities, allowing traffic to be swiftly redirected to the previous stable version. Additionally, the teams implemented automated safety measures that trigger a traffic switch if error rates exceed predefined thresholds during the initial deployment hours. This creates a safety net for developers, encouraging them to deploy more frequently in production without the fear of widespread impact in case of issues.

The positive impact of this automation has also resonated with the technical leadership at ACME. They have begun to reduce deployment gates, recognizing that this system inherently improves the quality of artifacts shipped to production. The ability to quickly identify and mitigate issues has led to a more agile and responsive development environment. By adopting this micro-frontends discovery pattern, ACME has not only enhanced its deployment safety but also fostered a culture of continuous improvement and frequent, low-risk releases. This approach aligns well with modern DevOps practices, emphasizing rapid iteration and feedback loops while maintaining high standards of reliability and performance.

## Automation Strategy Summary

Every area of this automation strategy (shown in [Figure 6-6](#)) is composed of one or more steps to provide a feedback loop to the development teams for different aspects of the development process from different testing strategies, like unit testing or end-to-end testing, visual regression, bundle-size check, and many others. All of these controls create confidence in the delivery of high-quality content. This strategy also provides developers with

a useful and constant reminder of the best practices leveraged inside the organization, guiding them to delivering what the business wants.

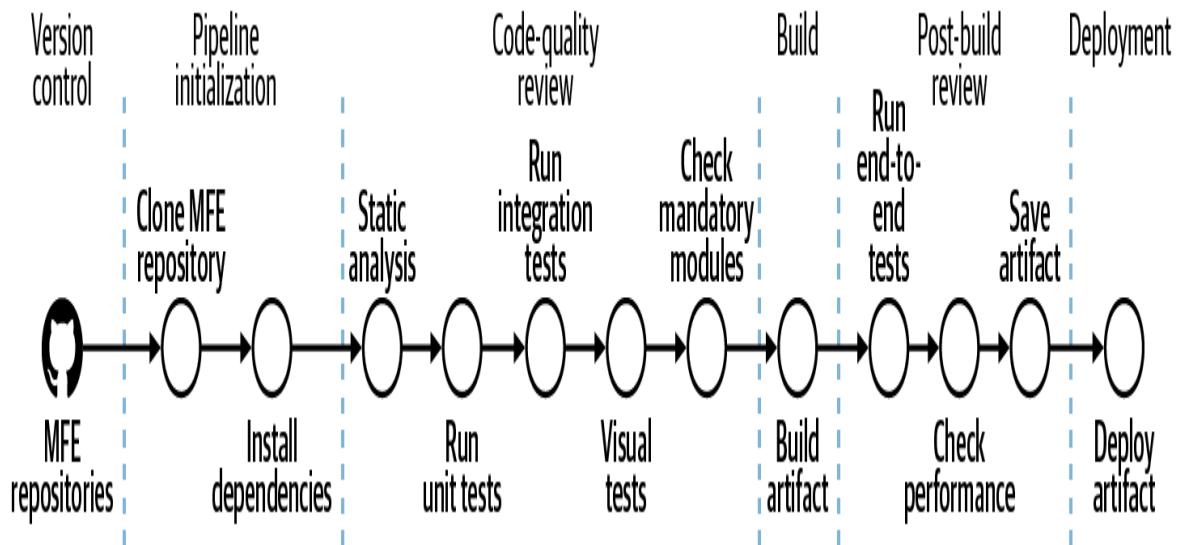


Figure 6-6. The end-to-end automation strategy diagram

The automation strategy shared in this chapter is one of many a company may decide to use. Different micro-frontend architectures will require additional or fewer steps than the ones described here. However, this automation strategy covers the main stages for ensuring a good result for a micro-frontend architecture.

Remember that the automation strategy evolves with the business and the architecture; therefore, after the first implementation, review it often with the development teams and the tech leadership. When automation serves the purpose of your micro-frontends well, implementation has a greater chance to be successful.

As we have seen, an automation strategy for micro-frontends doesn't differ too much from a traditional one used for an SPA. I recommend organizing some retrospectives every other month with architects, tech leaders, and representatives of every team to review and enhance such an essential cog in the software development process. And since every micro-frontend should have its own pipeline, the DX team is perfectly positioned to automate the infrastructure configurations as much as possible in order to have a frictionless experience when new micro-frontends arise. Using

containers allows a DX team to focus on the infrastructure, providing the boundaries needed for a team implementing its automation pipeline.

## Summary

In this chapter, we have reviewed a possible automation strategy for micro-frontends based on many concepts from the previous chapter. Your organization may benefit from some of these stages, but bear in mind that you need to constantly review the goals you want to achieve in your automation strategy. This is a fundamental step for succeeding with micro-frontends. Avoid it, and you may risk the entire project. The nature of micro-frontends requires an investment in creating a frictionless automation pipeline and enhancing it constantly. When a company starts to struggle to build and deploy regularly, that's a warning that the automation strategy probably needs to be reviewed and reassessed. Don't underestimate the importance of a good automation strategy; it may change the final outcome of your projects.

# Chapter 7. Backend Patterns for Micro-Frontends

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the ninth chapter of the final book. Please note that the GitHub repo will be made active later on.

If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the author at [\*building.microfrontends@gmail.com\*](mailto:building.microfrontends@gmail.com).

You may think that micro-frontends are a possible architecture only when you combine them with microservices because we can have end-to-end technology autonomy.

Maybe you’re thinking that your monolith architecture would never support micro-frontends, or even that having a monolith on the API layer would mean mirroring the architecture on the frontend as well.

However, that’s not the case. There are several nuances to take into consideration and micro-frontends can definitely be used in combination with microservices and monolith.

In this chapter, we review some possible integrations between the frontend and backend layers, in particular, we analyze how micro-frontends can work in combination with a monolith, with microservices, and even with the backend for frontend (BFF) pattern.

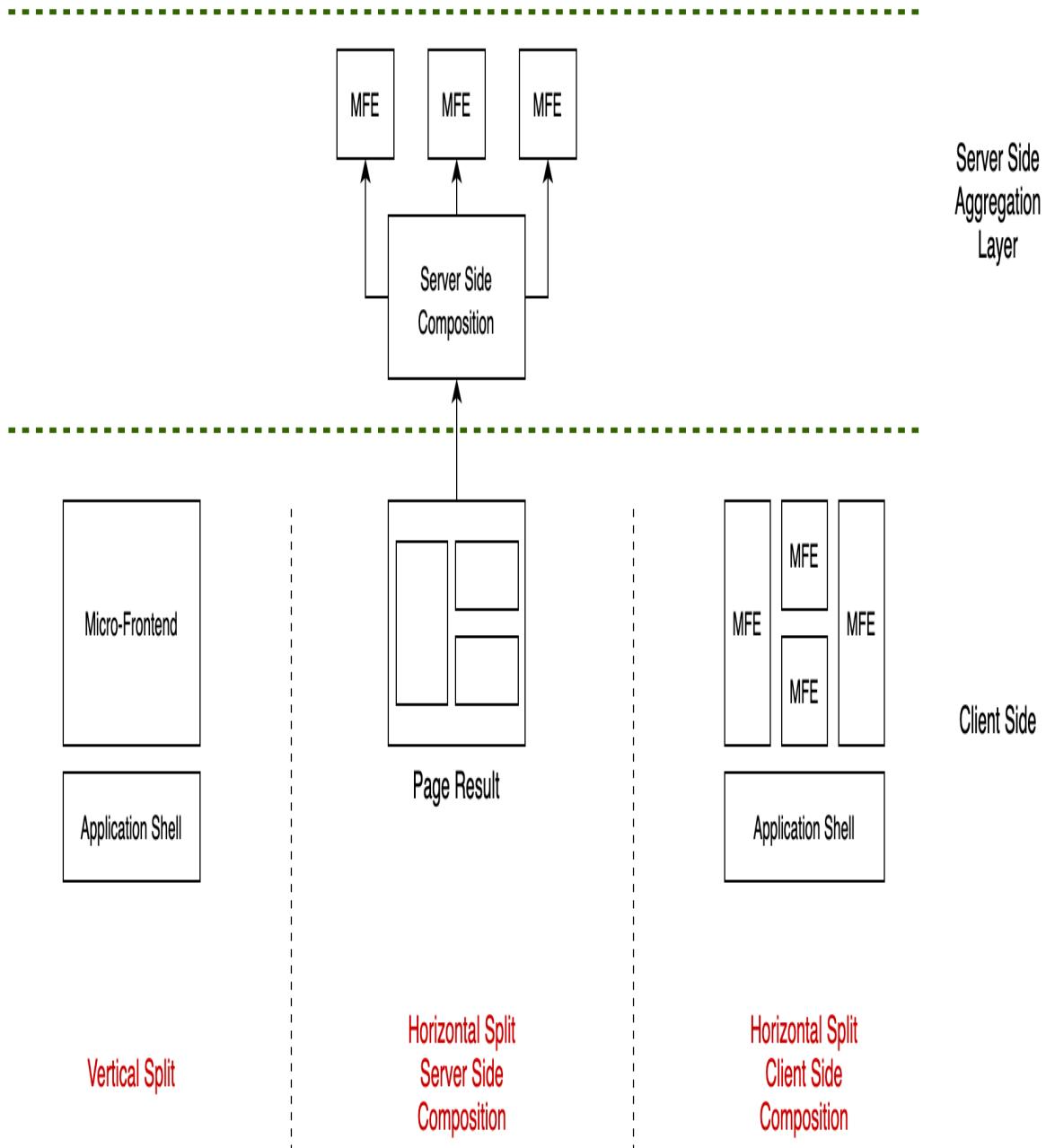
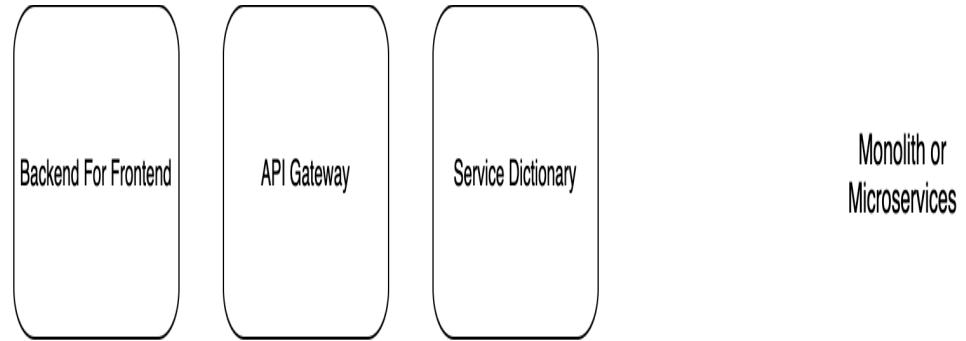
Also, we will discuss the best patterns to integrate with different micro-frontends implementations, such as the vertical split, the horizontal split with a client-side composition, and the horizontal split with server-side composition.

Finally, we will explore how GraphQL can be a valid solution for micro-frontends as a single entry point for our APIs.

## APIs integration and micro-frontends

Let's start by defining the different APIs approaches we may have in a web application. As shown in [Figure 7-1](#), we focus our journey on the most used and well-known patterns.

This doesn't mean micro-frontends work only with these implementations. You can devise the right approach for a WebSocket (a two-way computer communication protocol over a single TCP) or hypermedia (REST can be used with hypermedia links in the response contents, the client that consumes the API can dynamically navigate to the appropriate resources by traversing the hypermedia links), for instance, by learning how to deal with BFF, API gateway, or service dictionary patterns.



*Figure 7-1. Micro-frontends and API layers*

The patterns we analyze in this chapter are:

- **Service dictionary.** The service dictionary is just a list of services available for the client to consume. It's used mainly when we are developing an API layer with a monolith or modular monolith architecture; however, it can also be implemented with a microservices architecture with an API gateway, among other architectures. A service dictionary avoids the need to create shared libraries, environment variables, or configurations injected during the CI process or to have all the endpoints hardcoded inside the frontend codebase.

The dictionary is loaded for the first time when the micro-frontend loads, allowing the client to retrieve the URLs to consume directly from the service dictionary.

- **API gateway.** Well known in the microservices community, an API gateway is a single entry point for a microservices architecture. The clients can consume the APIs developed inside microservices through one gateway.

The API gateway also allows centralizing a set of capabilities, like:

- Token validation: validating the signature of a token before passing the request to a microservice
- Visibility and reporting: we have a centralized way to verify all the inbound and outbound traffic
- Rate-limiting: API Gateway rejects the request after exceeding a specific threshold, for instance we can set 100 requests per second as limit from a client when the limit is exceeded the API gateway returns errors instead calling the microservice to fulfill the request.

- **BFF.** The BFF is an extension of the API gateway pattern, creating a single entry point per client type. For instance, we may have a BFF for the web application, another for mobile, and a third for the Internet of Things (IoT) devices we are commercializing.

BFF reduces the chattiness between client and server aggregating the API responses and returning an easy data structure for the client to be parsed and render inside a user interface, allowing a great degree of freedom to shape APIs dedicated to a client and reducing the round trips between a client and the backend layer.

These patterns are not mutually exclusive, either; they can be combined to work together.

An additional possibility worth mentioning is writing an API endpoints library for the client side. However, I discourage this practice with micro-frontends because we risk embedding an older library version in some of them and, therefore, the user interface may have some issues like outdated information or even APIs errors due to dismissal of some APIs. Without strong governance and discipline around this library, we risk having certain micro-frontends using the wrong version of an API. It is way better to rely on the service discovery pattern or similar mechanisms that provide a list of endpoints at runtime.

Domain-driven design (DDD) also influences architecture and infrastructure decisions. Especially with end-to-end distributed systems, we can divide an application into multiple business domains, using the right approach for each business domain.

This level of flexibility provides architects and developers with a variety of choices not possible before. At the same time, however, we need to be careful not to fragment the client-server communication too much, instead introducing a new pattern when it provides a real benefit for our application. A beneficial approach I've observed over the years is to start by developing independent solutions for each team and then gradually consolidate them into unified entry points. As teams deploy multiple micro-frontends into production, the necessity to consolidate the API layer

becomes apparent, and governance naturally emerges from practical experience. Platform teams that attempt to design everything upfront run a higher risk of overcomplicating the entire process, leading to friction and hindering the swift flow that a distributed system requires, particularly at the onset of the journey.

## Working with a Service Dictionary

A service dictionary is nothing more than a list of endpoints available in the API layer provided to a micro-frontend. This allows the API to be consumed without the need to bake the endpoints inside the client-side code to inject them during a continuous integration pipeline or in a shared library.

Usually, a service dictionary is provided via a static JSON file or an API that should be consumed as the first request for a micro-frontend (in the case of a vertical-split architecture) or an application shell (in the case of a horizontal split).

A service dictionary may also be integrated into existing configuration files or APIs to reduce the round trips to the server and optimize the client startup.

In this case, we can have a JSON object containing a list of configurations needed for our clients, where one of the elements is the service dictionary.

An example of service dictionary structure would be:

```
{
  "my_amazing_api": {
    "v1": "https://api.acme.com/v1/my_amazing_api",
    "v2": "https://api.acme.com/v2/my_amazing_api",
    "v3": "https://api.acme.com/v3/my_amazing_api"
  },
  "my_super_awesome_api": {
    "v1": "https://api.acme.com/v1/my_super_awesome_api"
  }
}
```

As you can see, we are listing all the APIs supported by the backend. Thanks to API versioning, we can handle cross-platform applications without introducing breaking changes because each client can use the API version that suits it better.

One thing we can't control in such scenarios is the presence of a new version in every mobile device. When we release a new version of a mobile application, updating may take several days, if not weeks, and in some situations, it may take even longer.

Therefore, versioning the APIs is important to ensure we don't harm our user experience.

Reviewing the cadence of when to dismiss an API version, then, is important.

One of the main reasons is that potential attacks may harm our platform's stability.

Usually, when we upgrade an API to a new version, we are improving not only the business logic but also the security. But unless this change can be applicable to all the versions of a specific API, it would be better to assess whether the APIs are still valid for legitimate users and then decide whether to dismiss the support of an API.

To create a frictionless experience for our users, implementing a forced upgrade in every application released via an executable (think about React Native applications for instance) may be a solution, preventing the user from accessing older applications due to drastic updates in our APIs or even in our business model.

Therefore, we must think about how to mitigate these scenarios in order to create a smooth user experience for our customers.

Endpoint discoverability is another reason to use a service dictionary.

Not all companies work with cross-functional teams; many still work with components teams, with some teams fully responsible for the frontend of an application and others for the backend.

Using a service dictionary allows every frontend team to be aware of what's happening in other teams. If a new version of an API is available or a brand-new API is exposed in the service dictionary, the frontend team will be aware.

This is also a valid argument for cross-functional teams when we develop a cross-functional application.

In fact, it's very unlikely that inside a "two-pizza team" we would be able to have all the knowledge needed for developing web, backend, mobile (iOS and Android), and maybe even smart TVs and console applications considering many of these devices are supporting HTML and JavaScript.

## A TWO-PIZZA TEAM

According to Jeff Bezos, CEO of Amazon, if a team can't be fed with two pizzas, it's too big.

The introduction of the two-pizza rule in Amazon meant every team should be no larger than eight or nine people, which two pizzas would be enough to feed them!

The reasoning behind this rule isn't to save money on pizzas. It's based on the number of links between people inside a team.

There is a formula for calculating the links between members in a group:  $n(n-1)/2$  where  $n$  corresponds to the number of people.

For instance, if a team has six people, there will be 15 links between everyone. Double the team to 12 members, and there will be 66 links.

Complexity grows exponentially, not linearly, creating a higher risk of missing information across all the team's members.

Using a service dictionary allows every team to have a list of available APIs in every environment just by checking the dictionary.

We often think the problem is just a communication issue that can be resolved with better communication. However, look again at the number of

links in a 12-person team. Forgetting to update a team regarding a new API version may happen more often than not.

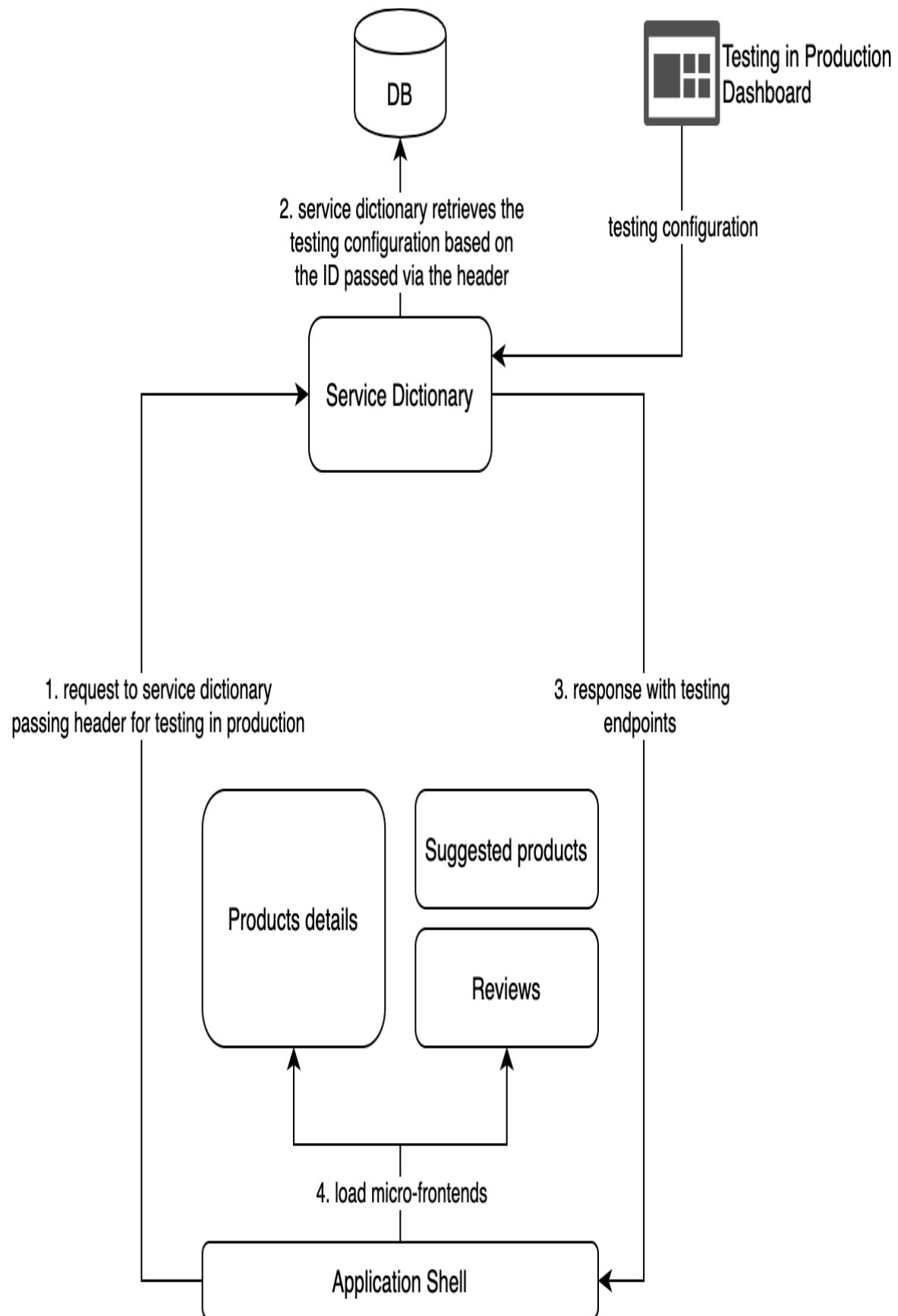
*A service dictionary aids in initiating discussions with the team responsible for the API, particularly in large organizations with distributed teams.*

Last but not least, a service dictionary is also helpful for testing micro-frontends with new endpoint versions while in production.

A company that uses a testing-in-production strategy can expand that to its micro-frontends architecture, thanks to the service dictionary, all without affecting the standard user experience.

We can test new endpoints in production by providing a specific header recognized by our service dictionary service. The service will interpret the header value and respond with a custom service dictionary used for testing new endpoints directly in production.

We would choose to use a header instead of a token or any other type of authentication, because it covers authenticated and unauthenticated use cases. Let's see a high-level design on what the implementation would look like ([Figure 7-2](#)).



*Figure 7-2. A high-level architecture on how to use a service dictionary for testing in production*

In **Figure 7-2** we can see that the application shell consumes the service dictionary API as the first step. But this time, the application shell passes a header with an ID related to the configuration that needs to be loaded.

In this example, the ID was generated at runtime by the application shell.

When the service dictionary receives the call, it will check for a header in the request. If present, it will load the associated configuration from the database

It then returns the response to the application shell with the specific service dictionary requested. The application shell is now ready to load the micro-frontends to compose the page.

Finally, the custom endpoint configuration associated with the client ID is produced via a dashboard (top right corner of the diagram) used only by the company's employees.

In this way we may even extend this mechanism for other use cases inside our backend, providing a great level of flexibility for micro-frontends and beyond.

The service dictionary can be implemented with either a monolith or a modular monolith. The important thing to remember is to allow categorization of the endpoints list based on the micro-frontend that requests the endpoints.

For instance we can group the endpoints related to a business subdomain or a bounded context. This is the strategic goal we should aim for.

A service dictionary makes more sense with micro-frontends composed on the client side rather than on the server side. BFFs and API gateways are better suited for the server-side composition, considering the coupling between a micro-frontend and its data layer.

## MODULAR MONOLITH

A modular monolith is a concept from the 1960s where the code is actually compartmentalized into separate modules. Moving to a modular monolith may be enough for some companies to continue evolving the API layer instead of doing a full migration to microservices. In his book *Monolith to Microservices*, Sam Newman provides many insights into migrating a monolithic backend to microservices and discusses the concept of the modular monolith as a potential first step for our migration journey.

Let's now explore how to implement the service dictionary in a micro-frontend architecture.

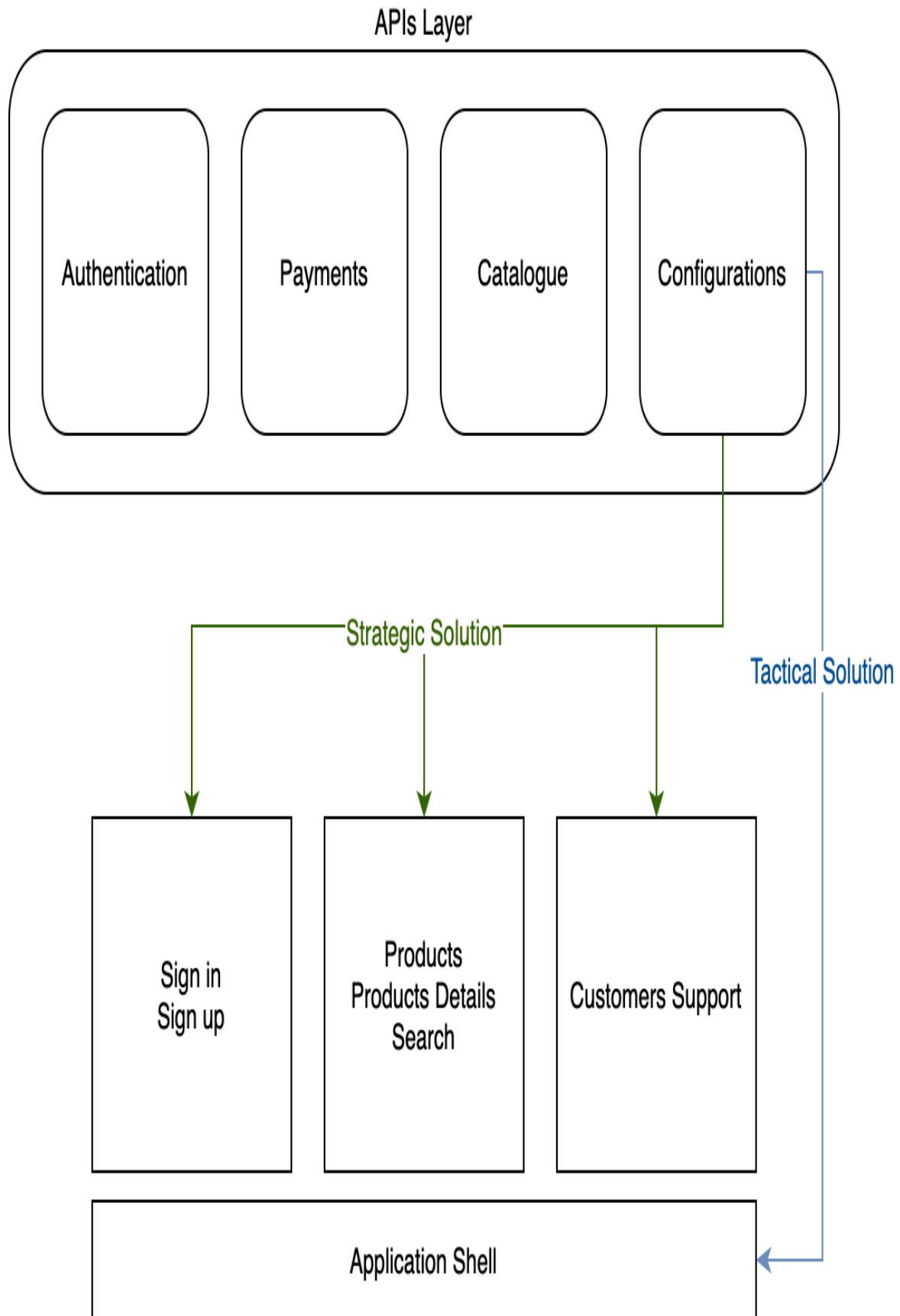
### Implementing a Service Dictionary in a Vertical-Split Architecture

The service dictionary pattern can easily be implemented in a vertical-split micro-frontends architecture, where every micro-frontend requests the dictionary related to its business domain.

However, it's not always possible to implement a service dictionary per domain, such as when we are transitioning from an existing SPA to micro-frontends, where the SPA requires the full list of endpoints because it won't reload the JavaScript logic until the next user session.

In this case, we may decide to implement a tactical solution, providing the full list of endpoints to the application shell instead of a business domain endpoints list to every single micro-frontend. With this tactical solution, we assume the application shell exposes or injects the list of endpoints for every micro-frontend.

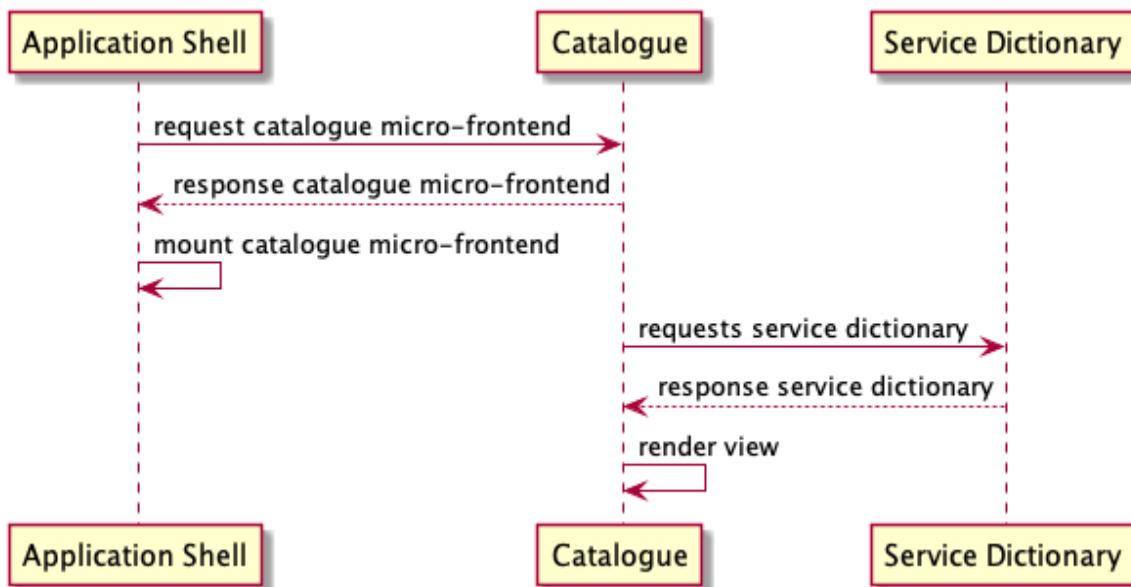
When we are in a position to divide the services list by domain, there will be a minimum effort for removing the logic from the application shell and then moving into every micro-frontend as displayed in [Figure 7-3](#).



*Figure 7-3. With vertical-split architecture we can retrieve the service dictionary directly inside a micro-frontend from an endpoint, in this case Configurations. Dividing the endpoints list by business domain allows us to structure our teams accordingly.*

The service dictionary approach may also be used with a monolith backend. If we determine that our API layer will never move to microservices, we can still implement a service dictionary divided by domain per every micro-frontend, especially if we implement a modular monolith.

Taking into account [Figure 7-3](#), we can derive a sample of sequence diagrams like the one in [Figure 7-4](#). Bear in mind there may be additional steps to perform either in the application shell or in the micro-frontend loaded, depending on the context we operate in. Take the following sequence diagram just as an example.



*Figure 7-4. Sequence diagram to implement a service dictionary with a vertical-split architecture*

As the first step, the application shell loads the micro-frontend requested, in this example the catalogue micro-frontend.

After mounting the micro-frontend, the catalogue initializes and consumes the service dictionary API for rendering the view. It can consume any additional APIs, as necessary.

From this moment on, the catalogue micro-frontend has access to the list of endpoints available and uses the dictionary to retrieve the endpoints to call.

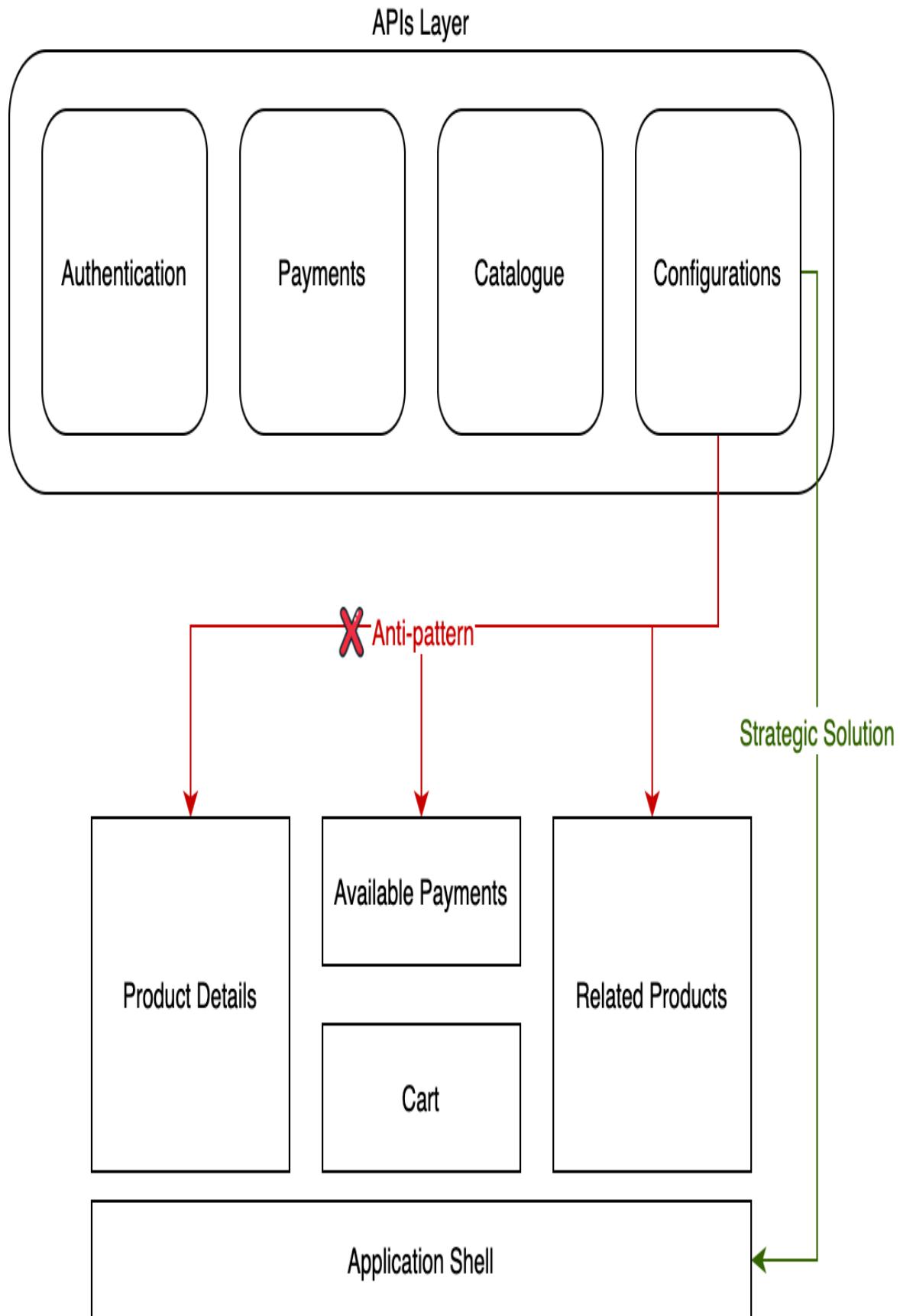
In this way we are loading only the endpoints needed for a micro-frontend, reducing the payload of our configuration and maintaining control of our business domain.

## **Implementing a Service Dictionary in a Horizontal-Split Architecture**

To implement the service dictionary pattern with a micro-frontends architecture using a horizontal split, we have to pay attention to where the service dictionary API is consumed and how to expose it for the micro-frontends inside a single view.

When the composition is managed client side, the recommended way to consume a service dictionary API is inside the application shell or host page. Because the container has visibility into every micro-frontend to load, we can perform just one round trip to the API layer to retrieve the APIs available for a given view and expose or inject the endpoints list to every loaded micro-frontend.

Consuming the service dictionary APIs from every micro-frontend would negatively impact our applications' performance, so it's strongly recommended to stick the logic in the micro-frontends container as shown in [Figure 7-5](#).



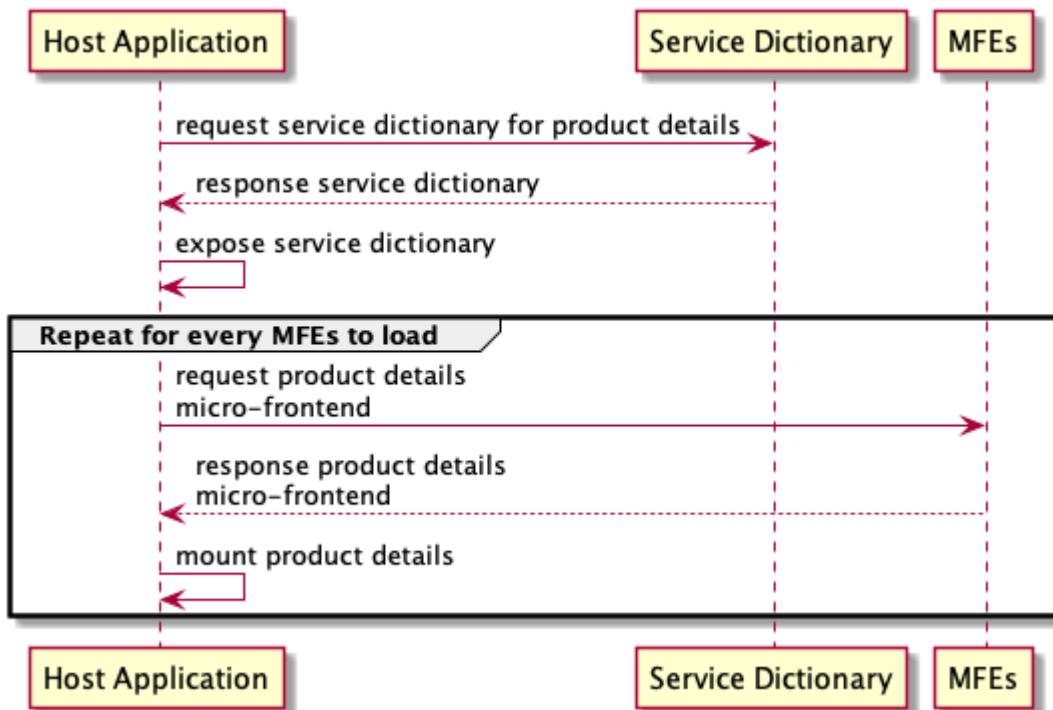
*Figure 7-5. The service dictionary should always be loaded from the micro-frontends container in a horizontal-split architecture*

The application shell should expose the endpoints list via the window object, making it accessible to all the micro-frontends when the technical implementation allows us to do it. Another option is injecting the service dictionary, alongside other configurations, after loading every micro-frontend.

For example, using module federation in a React application requires sharing the data using **React context APIs**. The context API allows you to expose a context, in our case the service dictionary, to the component tree without having to pass props down manually at every level.

The decision to inject or expose our configurations is driven by the technical implementation.

Let's see how we can express this use case with the sequence diagram in **Figure 7-6**.



*Figure 7-6. This sequence diagram shows how a horizontal-split architecture with client-side composition may consume the service dictionary API.*

In this sequence diagram, the request from the host application, or application shell, to the service dictionary is at the very top of the diagram.

The host application then exposes the endpoints list via the *window object* and starts loading the micro-frontends that compose the view.

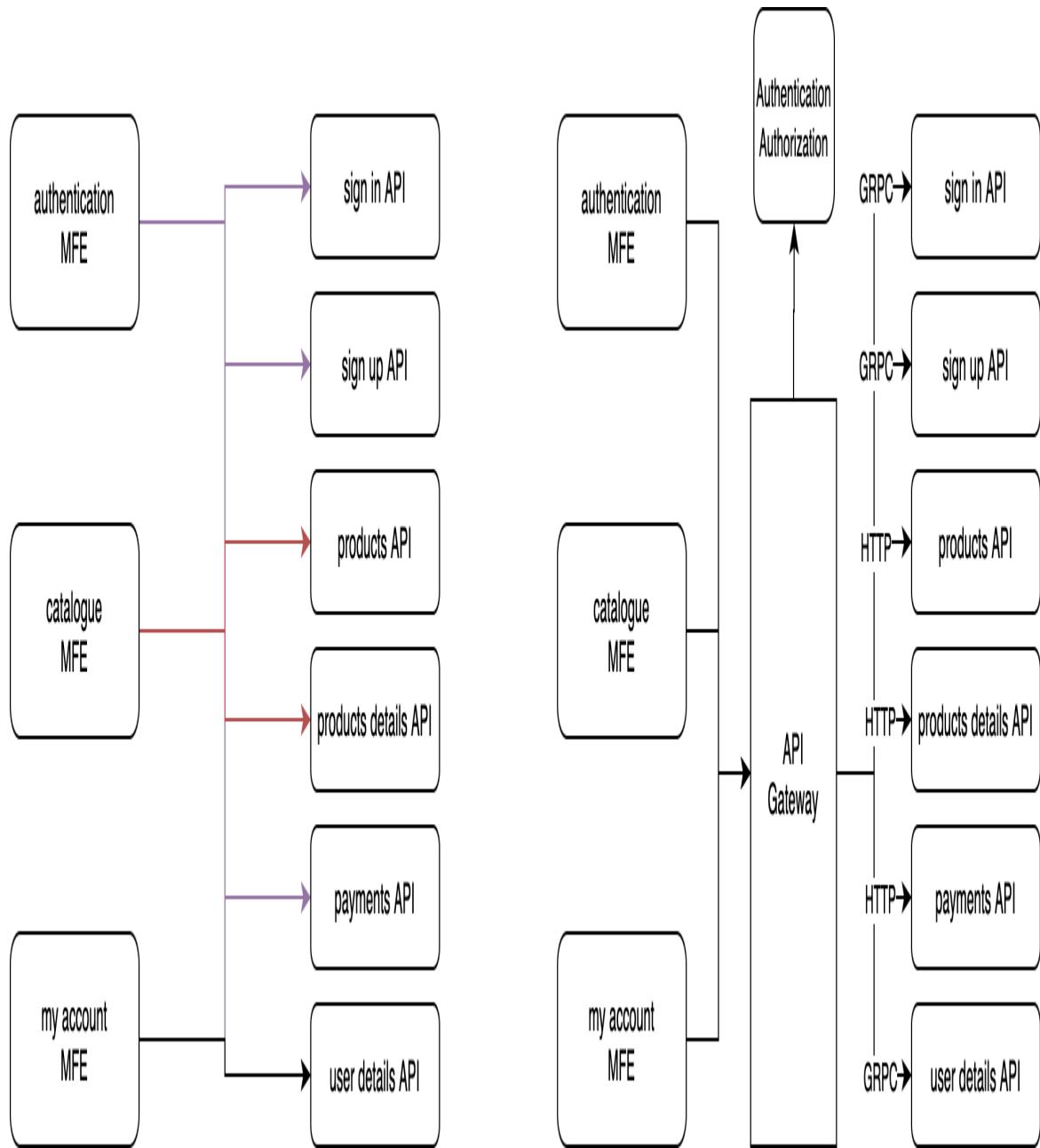
Again, in real scenarios we may have a more complex situation. Adapt the technical implementation and business logic to your project needs accordingly.

## Working with an API gateway

An API gateway pattern represents a unique entry point for the outside world to consume APIs in a microservices architecture.

Not only does an API gateway simplify access for any frontend to consume APIs by providing a unique entry point, but it's also responsible for requests routing, API composition and validation, and other edge functions, namely authorization, logging, rate limiting and any other centralized functionality we need to have before the API gateway send the request to a specific microservice.

An API gateway also allows us to keep the same communication protocol between clients and the backend, while the gateway routes a request in the background in the format requested by a microservice (see [Figure 7-7](#)).



*Figure 7-7. An API gateway pattern simplifies the communication between clients and server and centralizes functionalities like authentication and authorization via edge functions.*

Imagine a microservice architecture composed with HTTP and gRPC protocols. Without implementing an API gateway, the client won't be aware of every API or all the communication protocol details. Instead of using the API gateway pattern, we can hide the communication protocols behind the API gateway and leave the client's implementation dealing with the API contracts and implementing the business logic needed on the user interface.

Other capabilities of edge functions are rate limiting, caching, metrics collection, and log requests.

Without an API gateway, all these functionalities will need to be replicated in every microservice instead of centralized as we can do with a single entry point.

Still, the API gateway also has some downsides.

As a unique entry point, it could be a single point of failure, so we need to have a cluster of API gateways to add resilience to our application. Cloud providers typically offer services that easily address this resilience challenge, providing solutions designed to handle high traffic and well-architected for resilience.

Another challenge is more operational. In a large organization, where we have hundreds of developers working on the same project, we may have many services behind a single API gateway. We'll need to provide solid governance for adding, changing or removing APIs in the API gateway to prevent teams being frustrated with a cumbersome flow.

Finally, we'll add some latency to the system if we implement an additional layer between the client and the microservice consumed.

The process for updating the API gateway must be as lightweight as possible, making investing in the governance around this process a mandatory step. Otherwise, developers will be forced to wait in line to update the gateway with a new version of their endpoint.

The API gateway can work in combination with a service dictionary, adding the benefits of a service dictionary to those of the API gateway pattern.

Finally, with micro-architectures, we are opening a new scenario, where it may be possible and easier to manage and control because we are splitting our APIs by domain, having multiple API gateways to gather a group of APIs for instance.

## **One API entry point per business domain**

Another opportunity to consider is creating one API entry point per business domain instead of having one entry point for all the APIs, as with an API gateway.

Multiple API gateways enable you to partition your APIs and policies by solution type and business domain.

In this way, we avoid having a single point of failure in our infrastructure. Part of the application can fail without impacting the rest of the infrastructure. Another important characteristic of this approach is that we can use the best entry point strategy per bounded context based on the requirements needed, as shown in [Figure 7-8](#).

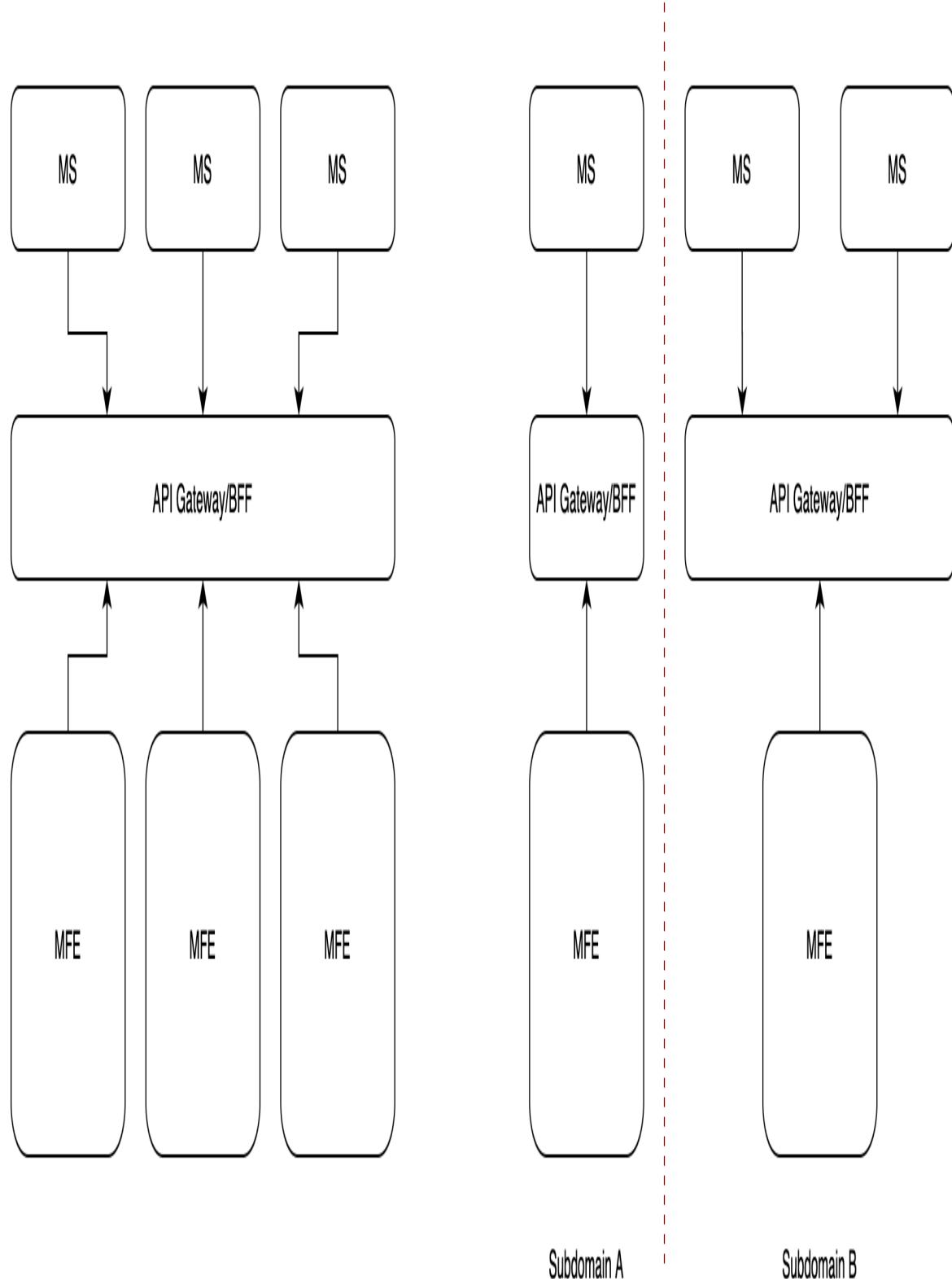


Figure 7-8. On the left is a unique entry point for the API layer; on the right are multiple entry points, one per subdomain.

So let's say we have a bounded context that needs to aggregate multiple APIs from different microservices and return a subset of the body response of every microservice. In this case, a BFF would be a better fit for being consumed by a micro-frontend rather than handing over to the client doing multiple round trips to the server and filtering the APIs body responses for displaying the final result to the user.

But in the same application, we may have a bounded context that doesn't need a BFF.

Let's go one step further and say that in this subdomain, we have to validate the user token in every call to the API layer to check whether the user is entitled to access the data.

In this case, using an API gateway pattern with validation at the API gateway level will allow you to fulfill the requirements in a simple way.

With infrastructure ownership, choosing different entry points for our API layer means every team is responsible for building and maintaining the entry point chosen, reducing potential external dependencies across teams, and allowing them to own end-to-end the subdomain they are responsible for. Therefore, potentially we can have a one-to-one relationship between subdomain and entry point.

This approach may require more work to build, but it allows a fine-grained control of identifying the right tool for the job instead of experiencing a trade-off between flexibility and functionalities. It also allows the team to really be independent end to end, allowing engineers to change the frontend, backend, and infrastructure without affecting any other business domain.

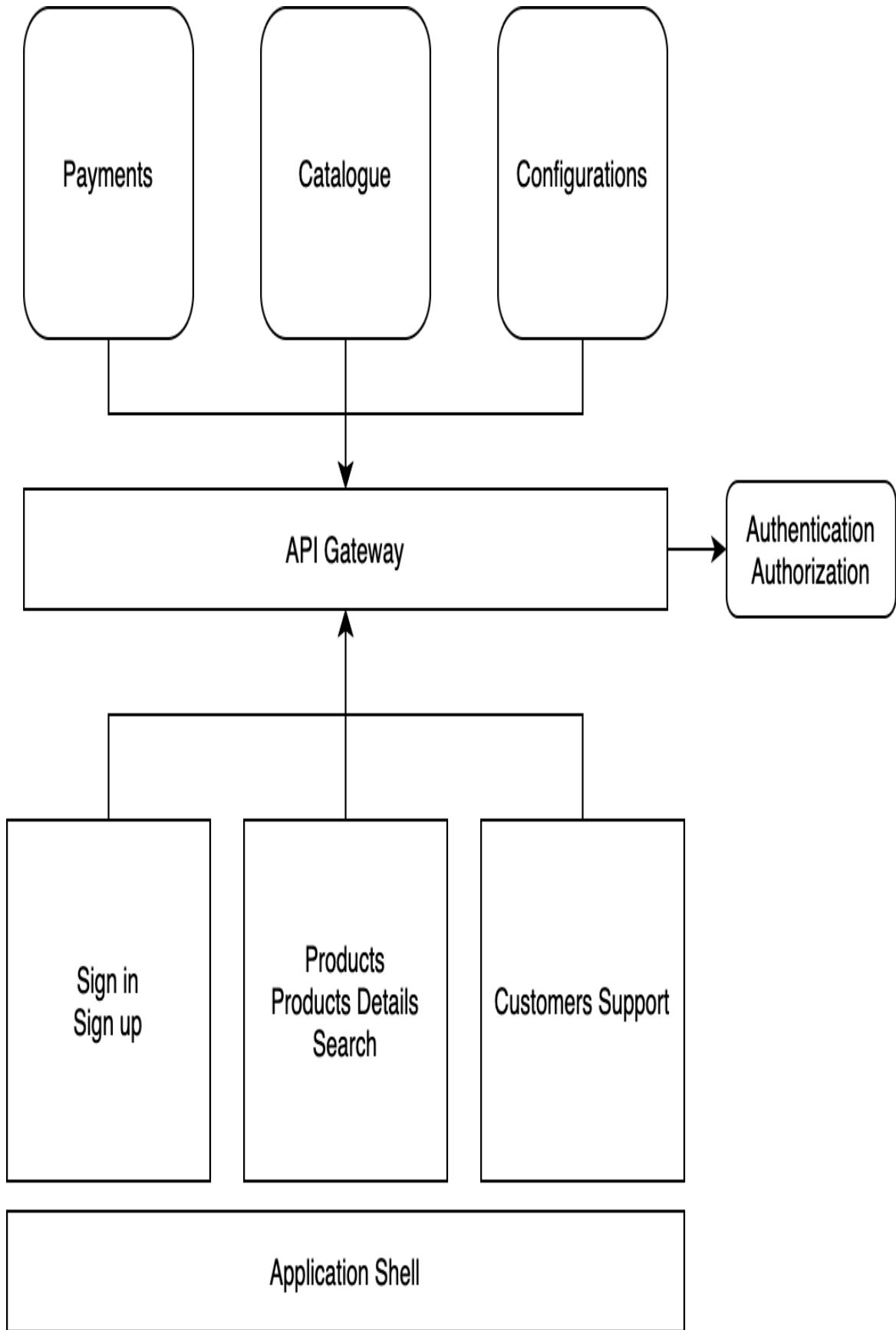
## **A client-side composition, with an API gateway and a service dictionary**

Using an API gateway with a client-side micro-frontends composition (either vertical or horizontal split) is not that different from implementing the service dictionary in a monolith backend.

In fact, we can use the service dictionary to provide our micro-frontends with the endpoints to consume, with the same suggestions we provided previously.

The main difference, in this case, will be that the endpoints list will be provided by a microservice responsible for serving the service dictionary or a more generic client-side configuration, depending on our use case.

Another interesting option is that with an API gateway, authorization may happen at the API-gateway level, removing the risk of introducing libraries at the API level, as we can see in [Figure 7-9](#).



*Figure 7-9. A vertical-split architecture with a client-side composition requesting data to a microservice architecture with an API gateway as entry point.*

Based on the concepts shared with the service dictionary, the backend infrastructure has changes but not the implementation side. As a result, the same implementations applicable to the service dictionary are also applicable in this scenario with the API gateway.

Let's look at one more interesting use case for the API gateway.

Some applications allow us to use a micro-frontends architecture to provide different flavors of the same product to multiple customers, such as customizing certain micro-frontends on a customer-by-customer basis.

In such cases, we tend to reuse the API layer for all the customers, using part or all of the microservices based on the user entitlement. But in a shared infrastructure we can risk having some customers consuming more of our backend resources than others.

In such scenarios, using API throttling at the API gateway will mitigate this problem by assigning the right limits per customer or per product.

At the micro-frontends level we won't need to do much more than handle the errors triggered by the API gateway for this use case.

## A server-side composition with an API gateway

A microservices architecture opens up the possibility of using a micro-frontends architecture with a server-side composition as explained in chapter 6.

As we can see in [Figure 7-10](#), after the browser's request to the API gateway, the gateway handles the user authentication/authorization first, then allows the client request to be processed by the UI composition service responsible for calling the microservices needed to aggregate multiple micro-frontends inside a template, with their relative content fetched from the microservices layer.

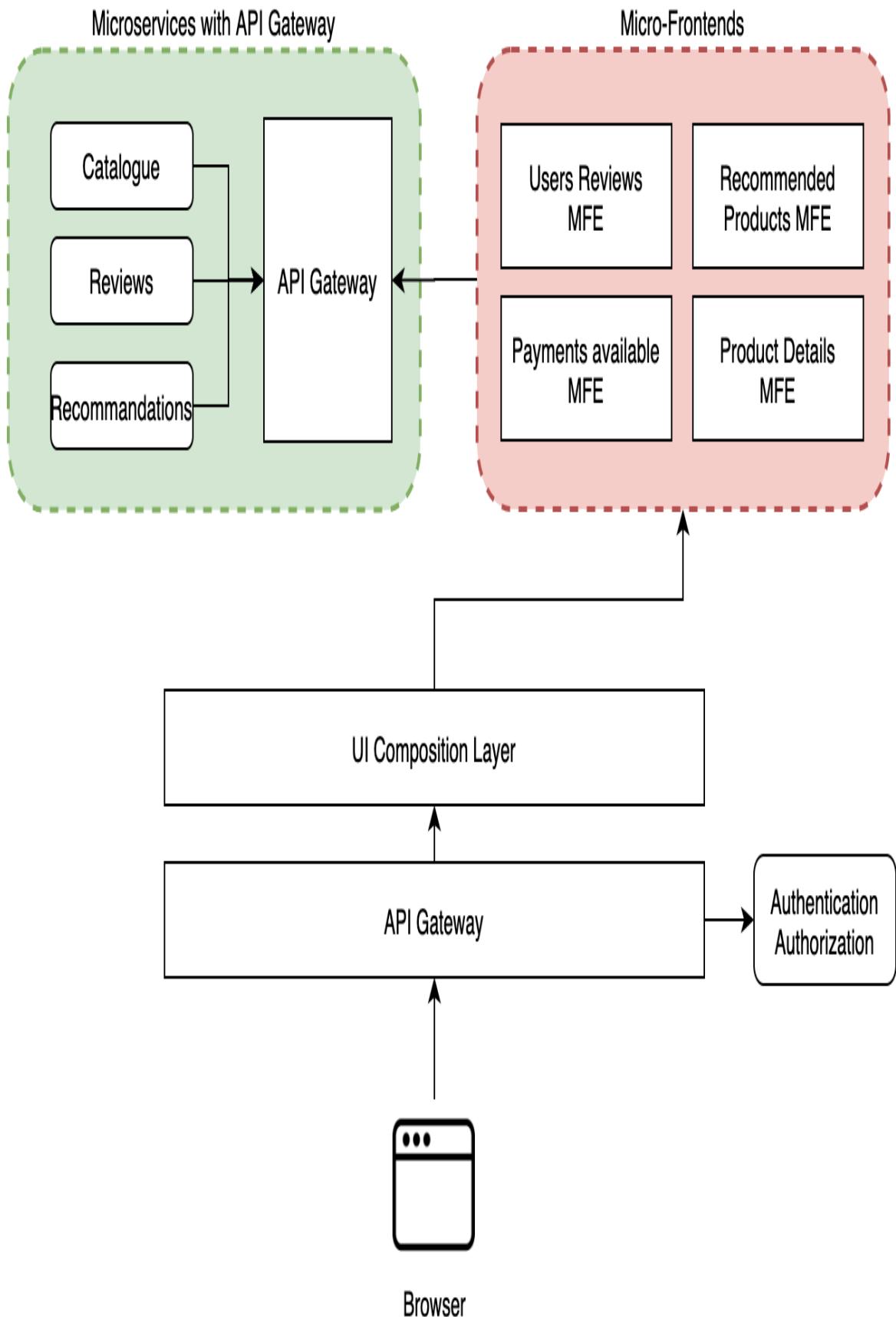


Figure 7-10. An example of a server-side composition with a microservices architecture

For the microservices layer, we use a second API gateway to expose the API for internal services, in this case, used by the micro-frontends services for fetching the related API.

Figure 7-11 illustrates a hypothetical implementation with the sequence diagram related to this scenario.

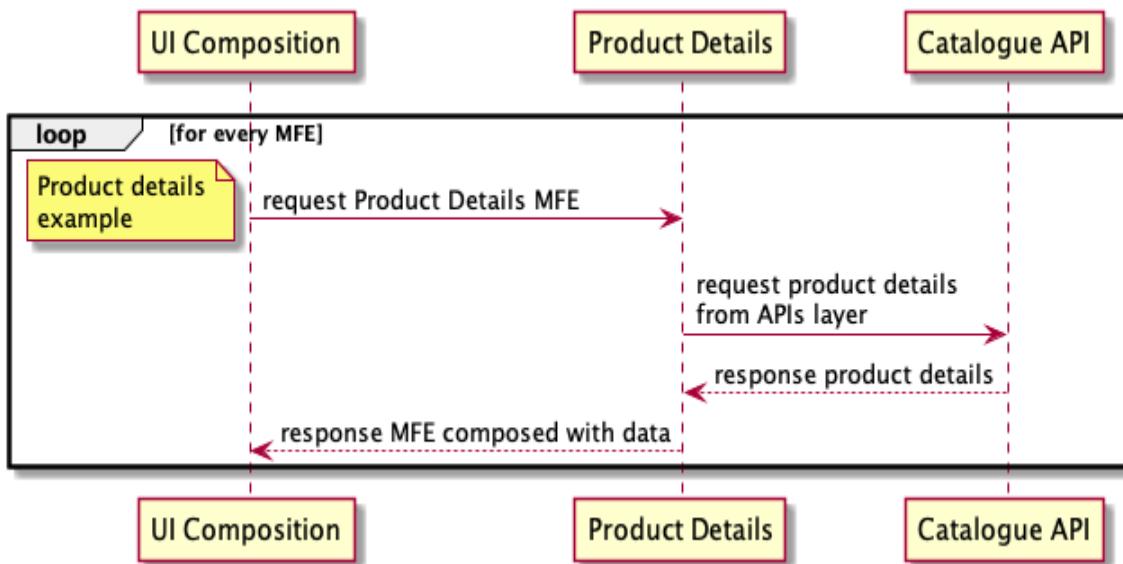


Figure 7-11. An example of server-side composition with API gateway

After the API gateway token validation, the client-side request lands at the UI composition service, which calls the micro-frontend to load. The micro-frontend service is then responsible for fetching the data from the API layer and the relative template for the UI and serving a fragment to the UI composition layer that will compose the final result for the user.

This diagram presents an example with a micro-frontend, but it's applicable for all the others that should be retrieved for composing a user interface.

Usually, the microservice used for fetching the data from the API layer should have a one-to-one relation with the API it consumes, which allows an end-to-end team's ownership of a specific micro-frontend and microservice.

## Working with the BFF pattern

Although the API gateway pattern is a very powerful solution for providing a unique entry point to our APIs, in some situations we have views that require aggregating several APIs to compose the user interface, such as a financial dashboard that may require several endpoints for gathering the data to display inside a unique view.

Sometimes, we aggregate this data on the client side, consuming multiple endpoints and interpolating data for updating our view with the diagrams, tables, and useful information that our application should display. Can we do something better than that? BFF comes to the rescue.

Another interesting scenario where an API gateway may not be suitable is in a cross-platform application where our API layer is consumed by web and mobile applications.

Moreover, the mobile platforms often require displaying the data in a completely different way from the web application, especially taking into consideration screen size.

In this case, many visual components and relative data may be hidden on mobile in favor of providing a more general high-level overview and allowing a user to drill down to a specific metric or information that interests them instead of waiting for all the data to download.

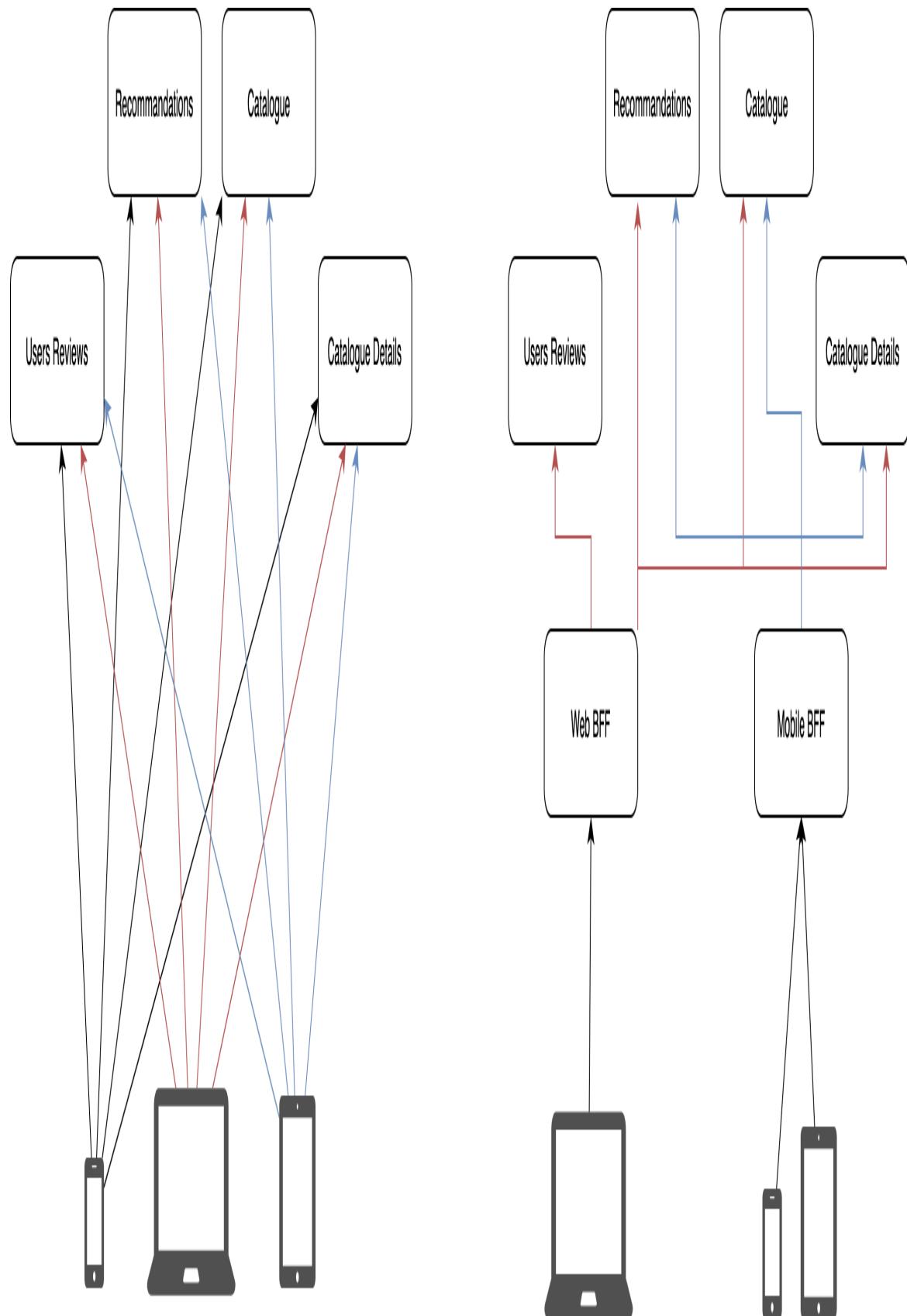
Finally, mobile applications often require a different method for aggregating data and exposing them in a meaningful way to the user. APIs on the backend are the same for all clients, so for mobile applications, we need to consume different endpoints and compute the final result on the device instead of changing the API responses based on the device that consumes the endpoint.

In all these cases, BFF, as described by [Phil Calçado](#) (former employee of SoundCloud), comes to the rescue.

The BFF pattern develops niche backends for each user experience.

This pattern will only make sense if and when you have a significant amount of data coming from different endpoints that must be aggregated for improving the client's performance or when you have a cross-platform application that requires different experiences for the user based on the device used.

This pattern can also help solve the challenge of introducing a layer between the API and the clients, as we can see in [Figure 7-12](#).



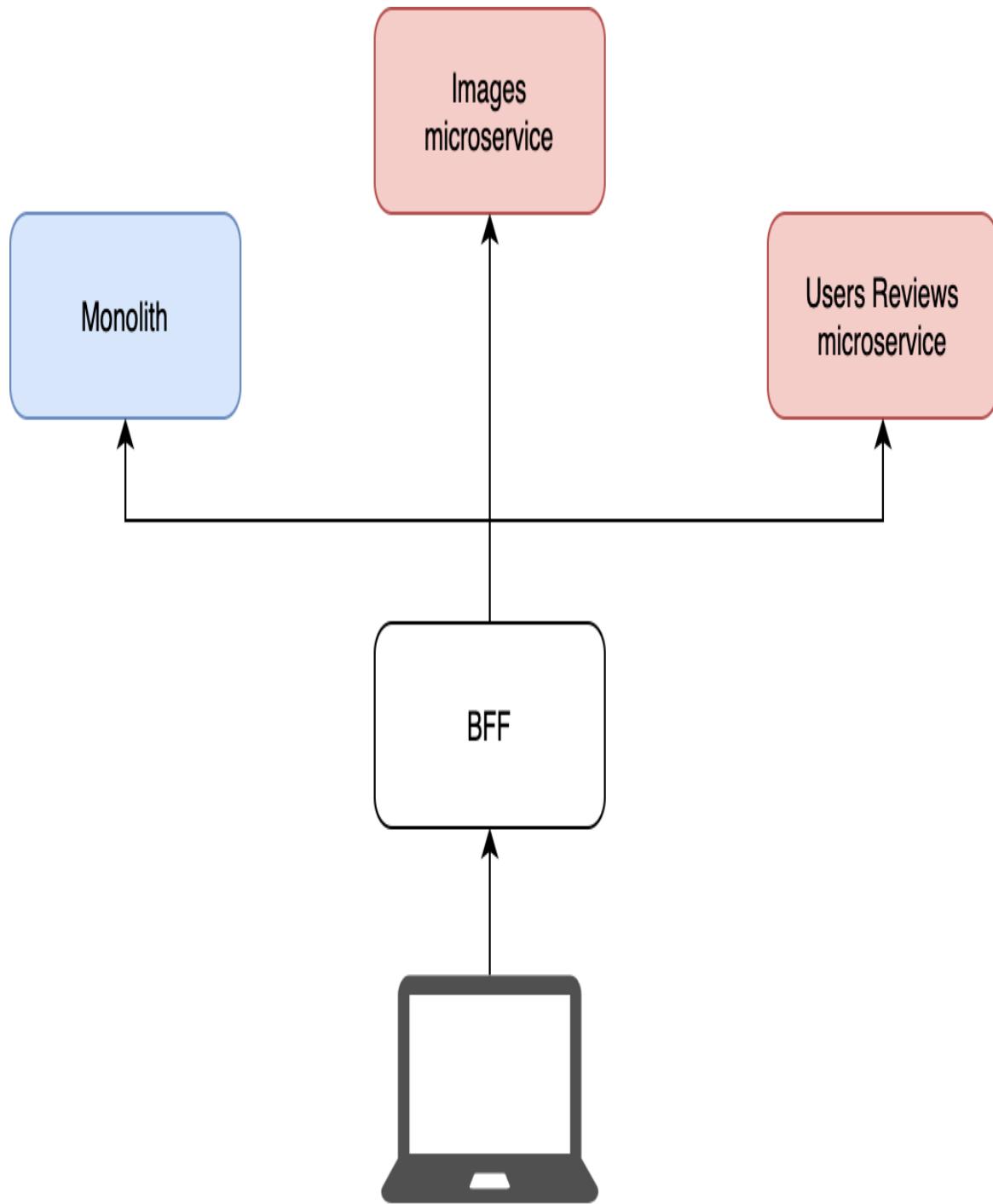
*Figure 7-12. On the left a microservices architecture consumed by different clients; on the right a BFF layer exposing only the APIs needed for a given group of devices, in this case, mobile and web BFF.*

Thanks to BFF we can create a unique entry point for a given device group, such as one for mobile and another for a web application.

However, this time we also have the option of aggregating API responses before serving them to the client and, therefore, generating less chatter between clients and the backend because the BFF aggregates the data and serves only what is needed for a client with a structure reflecting the view to populate.

Interestingly, the microservices architecture's complexity sits behind the BFF, creating a unique entry point for the client to consume the APIs without needing to understand the complexity of a microservices architecture.

BFF can also be used when we want to migrate a monolith to microservices. In fact, thanks to the separation between clients and APIs, we can use the strangler pattern for killing the monolith in an iterative way, as illustrated in [Figure 7-13](#). This technique is also applicable to the API gateway pattern.



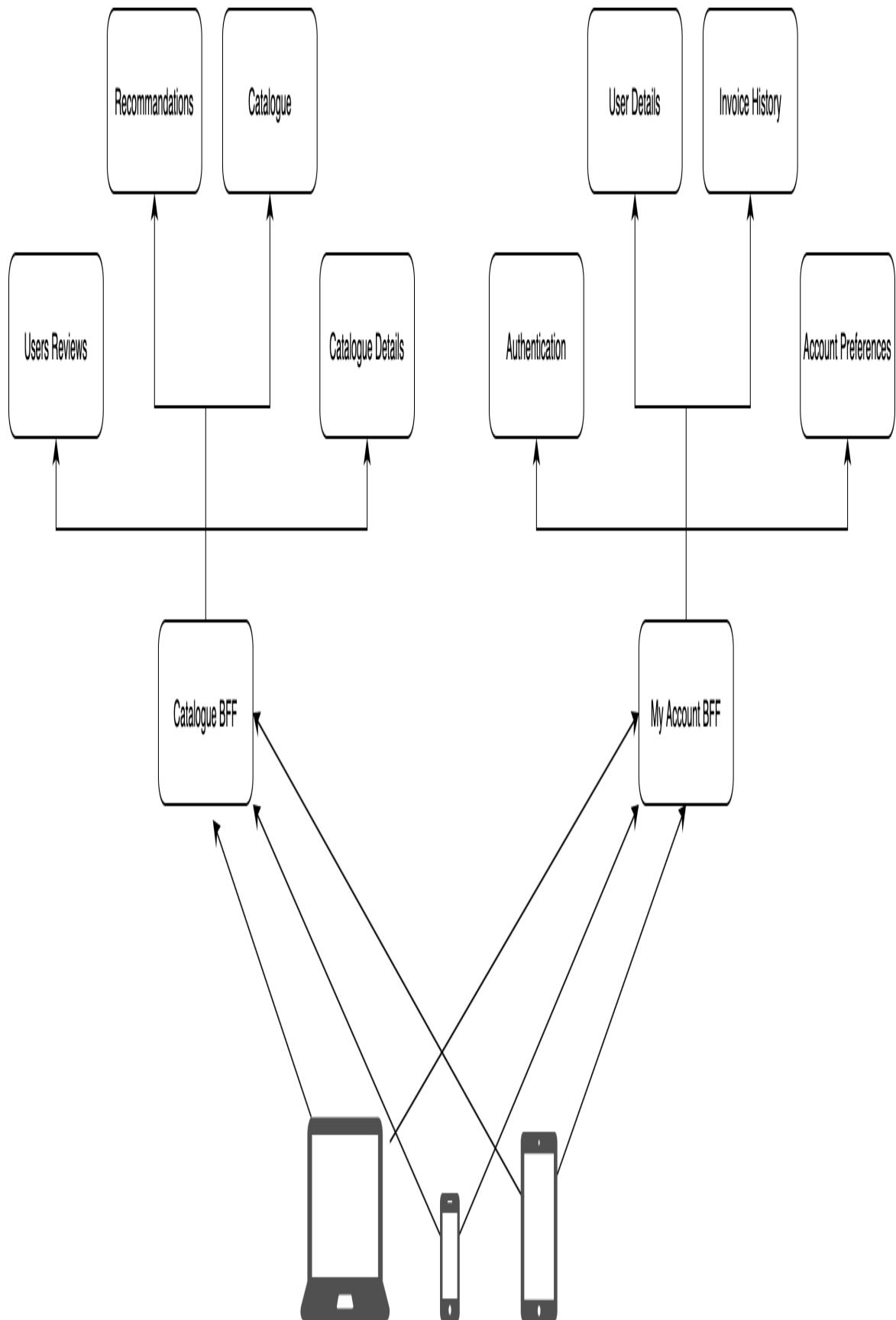
*Figure 7-13. The red boxes represent services extracted from the monolith and converted to microservices. The BFF layer allows the client to be unaware of the change happening in the backend, maintaining the same contract at the BFF level.*

Another use case that often comes to mind when we combine BFF and micro-frontends, is aggregating APIs by domain, similar to what we have seen for the API gateway.

Following our subdomain decomposition, we can identify a unique entry point for each subdomain, grouping all the microservices for a specific domain together instead of taking into consideration the type of device that should consume the APIs.

This would allow us to control the response to the clients in a more cohesive way, and allow the application to fail more gracefully than having a single layer responsible for serving all the APIs, as in the previous examples.

**Figure 7-14** illustrates how we can have two BFFs, one for the catalogue and one for the Account section, for aggregating and exposing these APIs to different clients. In this way, we can scale the BFFs based on their traffic.



*Figure 7-14. This diagram shows how to separate different domain-driven design subdomains.*

Gathering all the APIs behind a unique layer, however, may lead to an application's popular subdomains requiring a different treatment compared to less-accessed subdomains.

Dividing by subdomain, then, allows us to apply different infrastructure requirements based on the traffic and characteristics of each domain.

Sometimes BFF raises some concerns due to some inherent pitfalls such as reusability, code duplication and cross boundaries APIs.

In fact, we may need to duplicate some code for implementing similar functionalities across different BFF, especially when we create one per device family. In these cases, we need to assess whether the burden of having teams implementing similar code twice is greater than abstracting (and maintaining) the code.

It is no surprise that identifying domain boundaries for completely self-sufficient APIs is difficult.. Think about a service that is needed for multiple domains, for instance. Imagine an e-commerce where a product's service is used in multiple domains. In this case, we need to be careful to make sure that every BFF implements the latest API version of the products service. Moreover, every time a new version of the products service is released, we will need to coordinate the release of the BFF layers. Alternatively, we can support multiple versions of the products API for a period of time, allowing each BFF to update independently at its own pace.

## **A client-side composition, with a BFF and a service dictionary**

Because a BFF is an evolution of the API gateway, many of the implementation details for an API gateway are valid for a BFF layer as well, plus we can aggregate multiple endpoints, reducing client chatter with the server.

It's important to iterate this capability because it can drastically improve application performance.

Yet there are some caveats when we implement either a vertical split or a horizontal one.

For instance, in [Figure 7-15](#), we have a product details page that has to fetch the data for composing the view.

logo

menu

product details

payment options

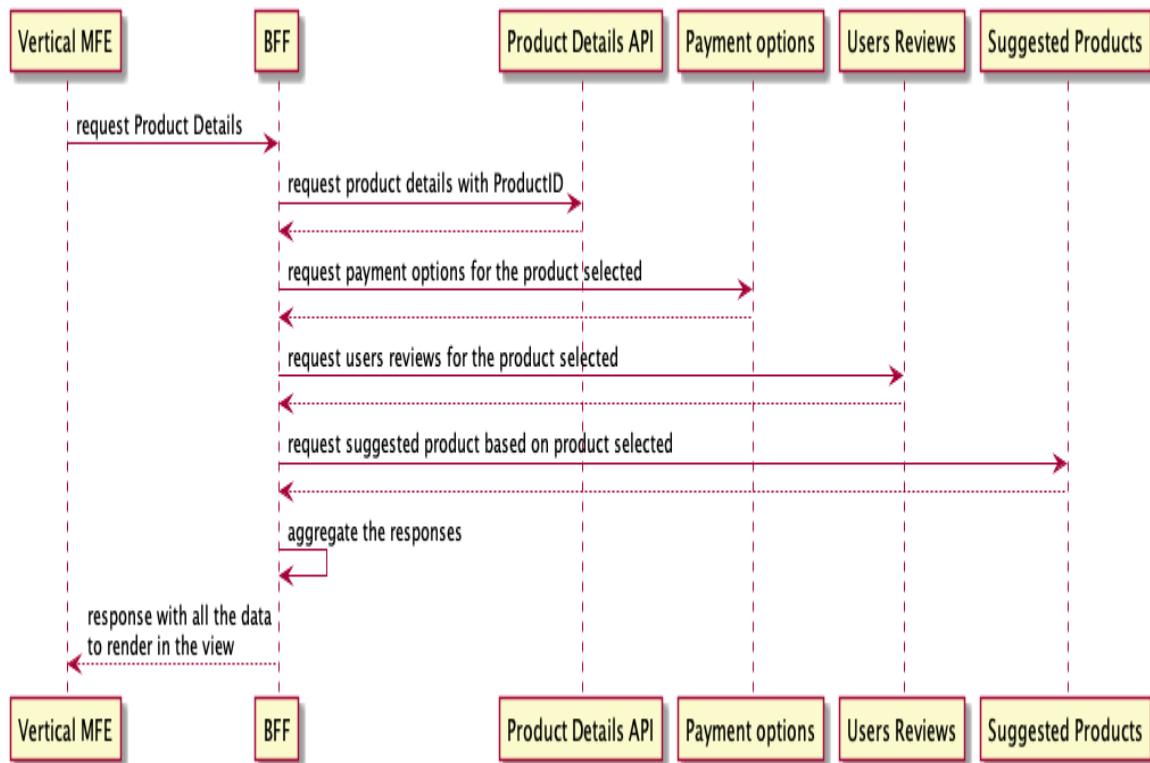
related products

users reviews

footer

*Figure 7-15. A wireframe of a product page*

When we want to implement a vertical-split architecture, we may design the BFF to fetch all the data needed for composing this view, as we can see in Figure 7-16.



*Figure 7-16. Sequence diagram showing the benefits of the BFF pattern used in combination with a vertical split composed on the client side*

In this example, we assume the micro-frontend has already retrieved the endpoint for performing the request via a service dictionary and that it consumes the endpoints, leaving the BFF layer to compose the final response.

In this use case we can also easily use a service dictionary for exposing the endpoints available in our BFF to our micro-frontends similar to the way we do it for the API gateway solution.

However, when we have a horizontal split composed on the client side, things become trickier because we need to maintain the micro-frontends'

independence, as well as having the host page domain as unaware as possible.

In this case, we need to combine the APIs in a different way, delegating each micro-frontend to consume the related API, otherwise, we will need to make the host page responsible for fetching the data for all the micro-frontends, which could create a coupling that would force us to deploy the host page with the micro-frontends, breaking the intrinsic characteristic of independence between micro-frontends.

Considering that these micro-frontends and the host page may be developed by different teams, this setup would slow down feature development rather than leveraging the benefits of this architecture.

Moreover, this might lead to creating a global state, implemented at the micro-frontends' container for simplifying the access for all the micro-frontends present in the view, creating unnecessary coupling.

BFF with a horizontal split composed on the client side could create more challenges than benefits in this case. It's wise to analyze whether this pattern's benefits will outweigh the challenges.

## **A server-side composition, with a BFF and service dictionary**

When we implement a horizontal-split architecture with server-side composition and we have a BFF layer, our micro-frontends implementation resembles the API gateway one.

The BFF exposes all the APIs available for every micro-frontend, so using the service dictionary pattern will allow us to retrieve the endpoints for rendering our micro-frontends ready to be composed by a UI composition layer.

# Using GraphQL with micro-frontends

In a chapter about APIs and micro-frontends, we couldn't avoid mentioning [GraphQL](#).

GraphQL is a query language for APIs and a server-side runtime for executing queries by using a type system you define for your data.

GraphQL was created by Facebook and released in 2015. Since then it has gained a lot of traction inside the developers' community.

Especially for frontend developers, GraphQL represents a great way to retrieve the data needed for rendering a view, decoupling the complexity of an API layer, rationalizing the API response in a graph, and allowing any client to reduce the number of round trips to the server for composing the UI.

The paradigm for designing an API schema with GraphQL should be based on how the view we need to render looks instead of looking at the data exposed by the API layer.

This is a very key distinction compared to how we design our database schemas or our REST APIs.

Two projects in the GraphQL community stand out as providing great support and productivity with the open source tools available, such as [Apollo](#) and [Rely](#).

Both projects leverage GraphQL, adding an opinionated view on how to implement this layer inside our application, increasing our productivity thanks to the features available in one or both, like authentication, rate limiting, caching, and schema federations.

GraphQL can be used as a proxy for microservices, orchestrating the requests to multiple endpoints and aggregating the final response for the client.

Remember that GraphQL acts as a unique entry point for your entire API layer. By design GraphQL exposes a unique endpoint where the clients can perform queries against the GraphQL server. Because of this, we tend to not

version our GraphQL entry point, although if the project requires a versioning because we don't have full control of the clients that consume our data, we can version the GraphQL endpoint. [Shopify](#) does this by adding the date in the URL and supporting all the versions up to a certain period.

It's important to highlight that GraphQL works best when it's created as a unique entry point for the client and not split by domains as seen with BFF. The graph implementation allows every client to query whatever part of the graph is exposed. Splitting it up in multiple domains would just make life harder for developers integrating with multiple graphs that now have to compose the different responses on the client-side. You might wonder how to scale the development of GraphQL across multiple teams, the answer is schema federation.

## The schema federation

Schema federation is a feature that allows multiple [GraphQL schemas](#) to be composed declaratively into a single data graph.

When we work with GraphQL in a midsize to large organization, we risk creating a bottleneck because all the teams are contributing to the same schema.

But with schema federation, we can have individual teams working on their own schemas and exposing them to the client as unique entry points, just like a traditional data graph.

Apollo Server exposes a gateway with all associated schemas from other services, allowing each team to be independent and not change the way the frontend consumes the data graph.

This technique comes in handy when we work with microservices, though it comes with a caveat.

A GraphQL schema should be designed with the UI in mind, so it's essential to avoid silos inside the organization. We must facilitate the initial

analysis engaging with multiple teams and follow all improvements in order to have the best implementation possible.

Figure 7-17 shows how a schema federation works using the gateway as an entry point for all the implementing services and providing a unique entry point and data graph to query for the clients.

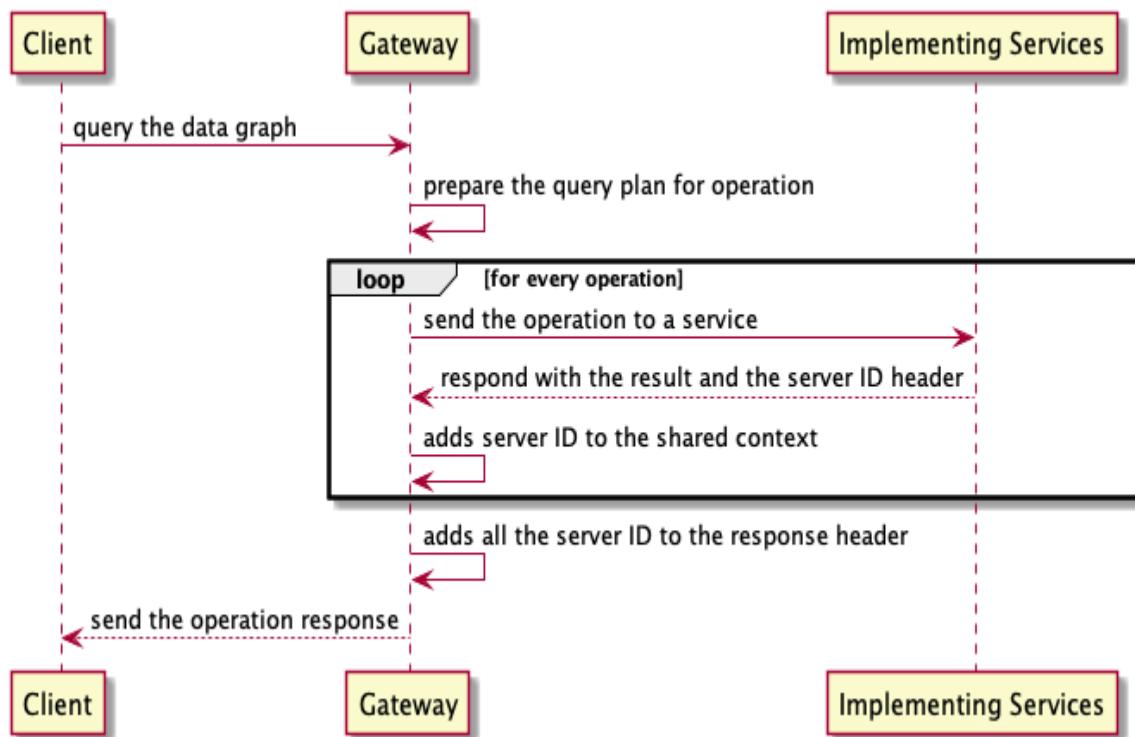


Figure 7-17. A sequence diagram showing how schema federation exposes all the schemas from multiple services

Schema federation represents the evolution of **schema stitching**, which has been used by many large organizations for similar purposes. It wasn't well designed, however, which led Apollo to deprecate schema stitching in favor of schema federation.

More information regarding the schema federation is available on [Apollo's documentation website](#).

## Using GraphQL with micro-frontends and client-side composition

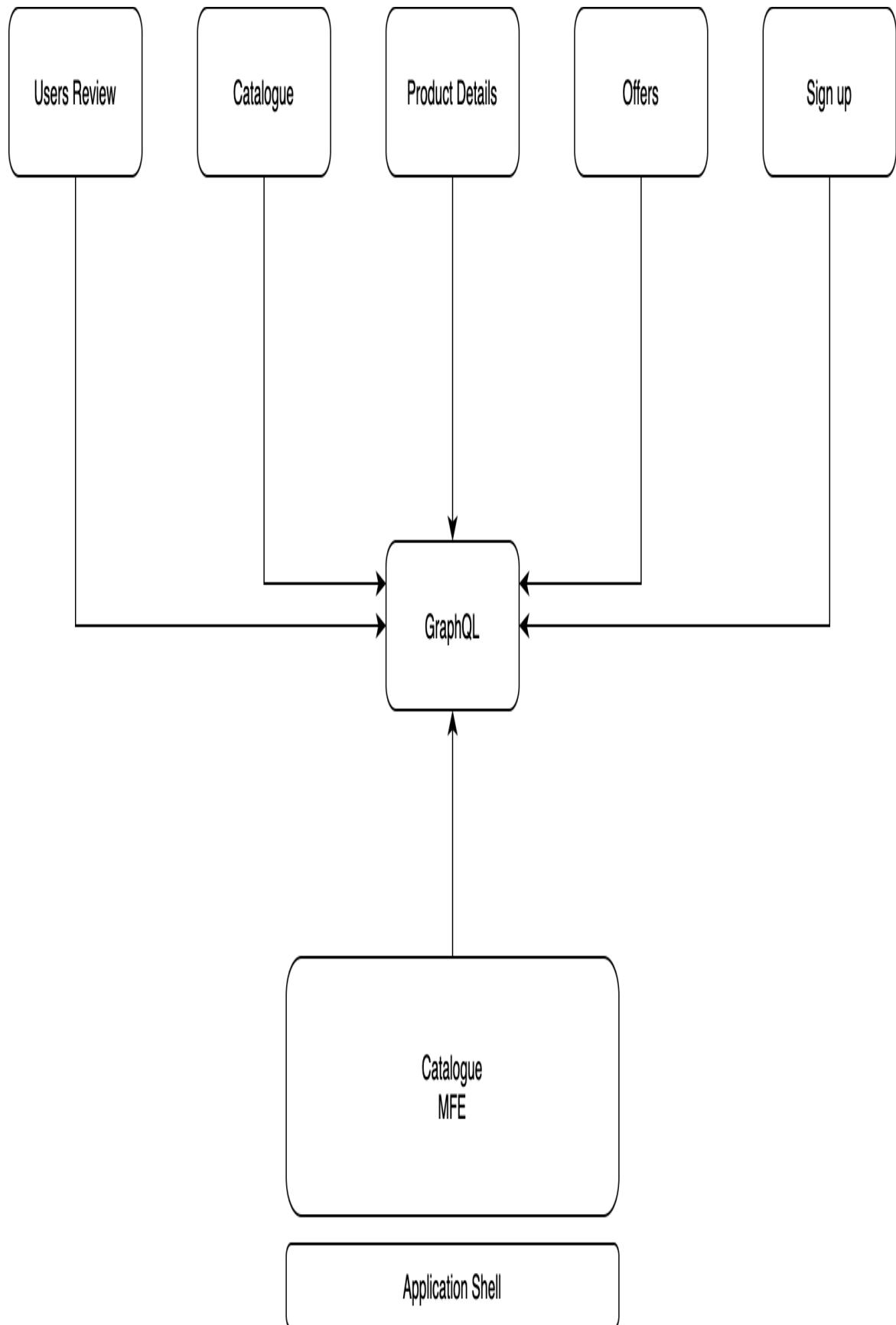
Integrating GraphQL with micro-frontends is a trivial task, especially after reviewing the implementation of the API gateway and BFF.

With schema federations, we can have the teams who are responsible for a specific domain's APIs create and maintain the schema for their domain and then merge all the schemas into a unique data graph for our client applications.

This approach allows the team to be independent, maintaining their schema and exposing what the clients would need to consume.

When we integrate GraphQL with a vertical split and a client-side composition, the integration resembles the others described above: the micro-frontend is responsible for consuming the GraphQL endpoint and rendering the content inside every component present in a view.

Applying such scenarios with microservices become easier thanks to schema federation, as shown in [Figure 7-18](#).



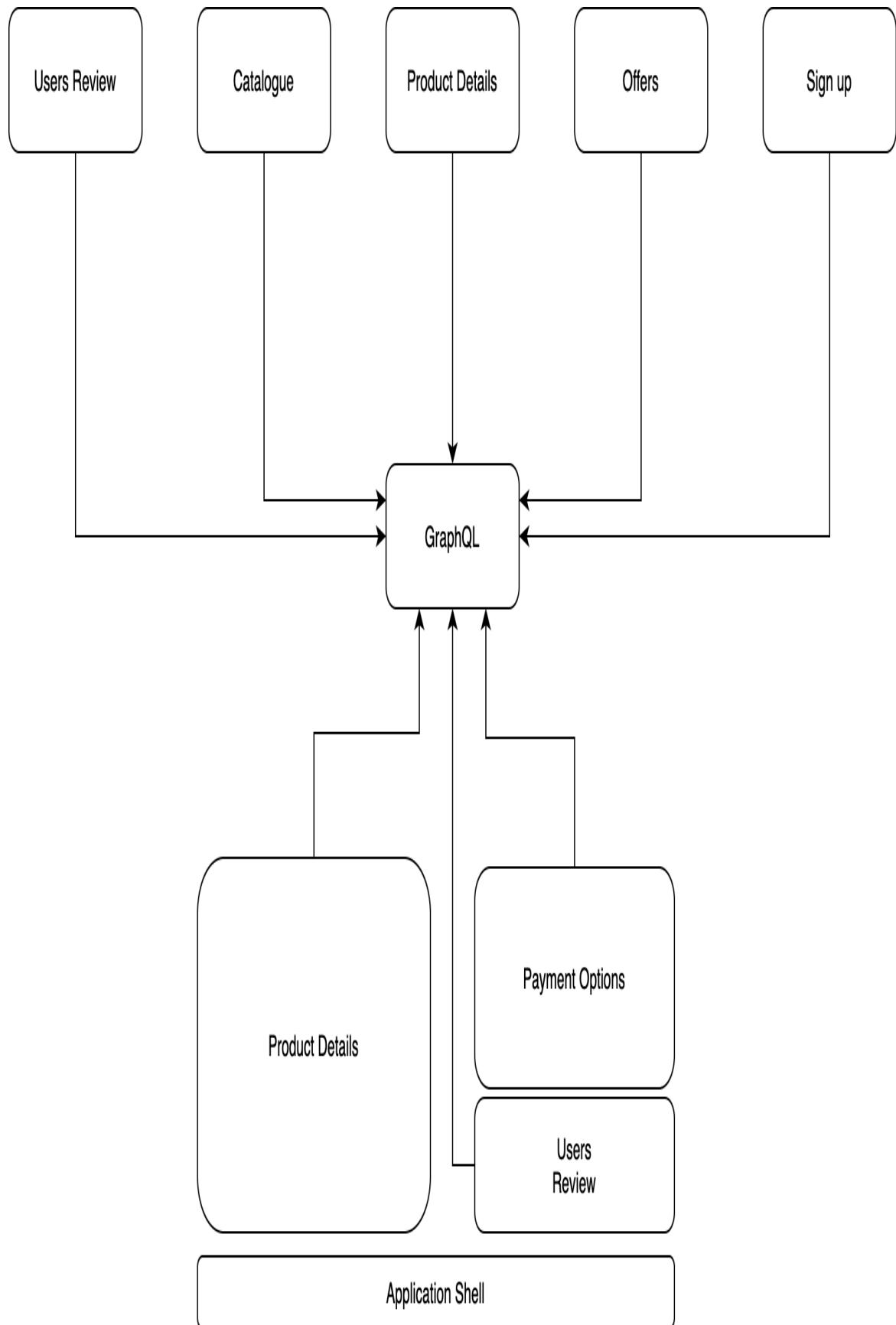
*Figure 7-18. A high-level architecture for composing a microservice backend with schema federation. The catalogue micro-frontend consumes the graph composed by all the schemas inside the GraphQL server.*

In this case, thanks to the schema federation, we can compose the graph with all the schemas needed and expose a supergraph for a micro-frontend to consume.

Interestingly, with this approach, every micro-frontend will be responsible for consuming the same endpoint. Optionally, we may want to split the BFF into different domains, creating a one-to-one relation with the micro-frontend. This would reduce the scope of work and make our application easier to manage, considering the domain scope is smaller than having a unique data graph for all the applications.

Applying a similar backend architecture to horizontal-split micro-frontends with a client-side composition isn't too different from other implementations we have discussed in this chapter.

As we see in [Figure 7-19](#), the application shell exposes or injects the GraphQL endpoint to all the micro-frontends and all the queries related to a micro-frontend will be performed by every micro-frontend.



*Figure 7-19. A high-level architecture of GraphQL with schema federation. When we implement it with a micro-frontends architecture with horizontal split and a client-side composition, all micro-frontends query the graph layer.*

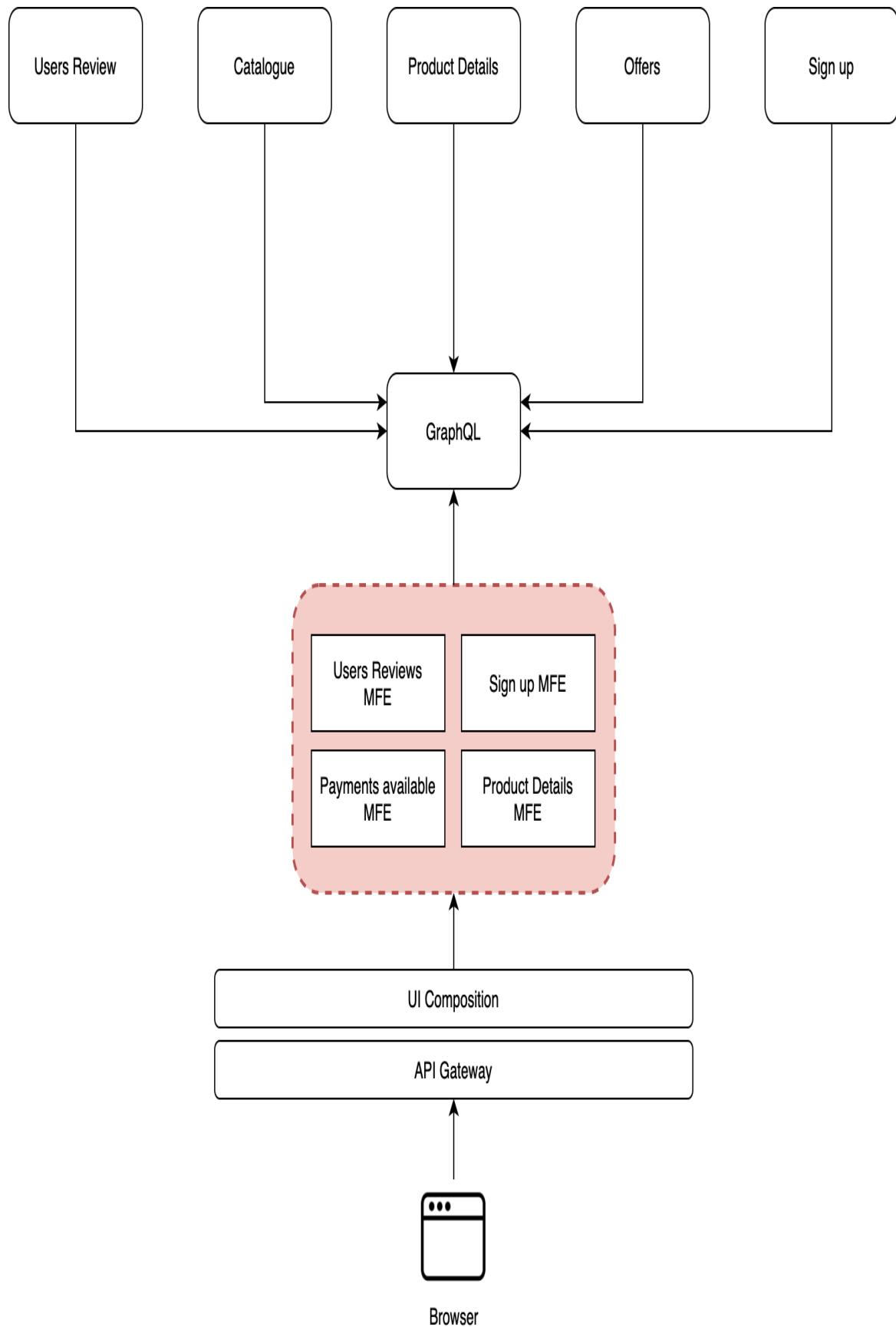
When we have multiple micro-frontends in the same or different view performing the same query, it's wise to look at the query and response cacheability at different levels, like the **CDN** used, and otherwise leverage the GraphQL server-client cache.

Caching is a very important concept that has to be leveraged properly; doing so could protect your origin from burst traffic so spend the time. Even when we have dynamic data, caching data for tens of seconds or a few minutes, helps reduce the strain on the origin and the risk of failures.

## **Using GraphQL with micro-frontends and a server-side composition**

The final approach involves using a GraphQL server with a micro-frontends architecture featuring a horizontal split and server-side composition.

When the UI composition requests multiple micro-frontends to their relative microservices, every microservice queries the graph and prepares the view for the final page composition (see [Figure 7-20](#)).



*Figure 7-20. A high-level architecture for a micro-frontends architecture with a server-side composition where every micro-frontend consumes the graph exposed by the GraphQL server*

In this scenario, every microservice that will query the GraphQL server requires having the unique entry point accessible, authenticating itself, and retrieving the data needed for rendering the micro-frontend requested by the UI composition layer.

This implementation overlaps quite nicely with the others we have seen so far on API gateway and BFF patterns.

## Best practices

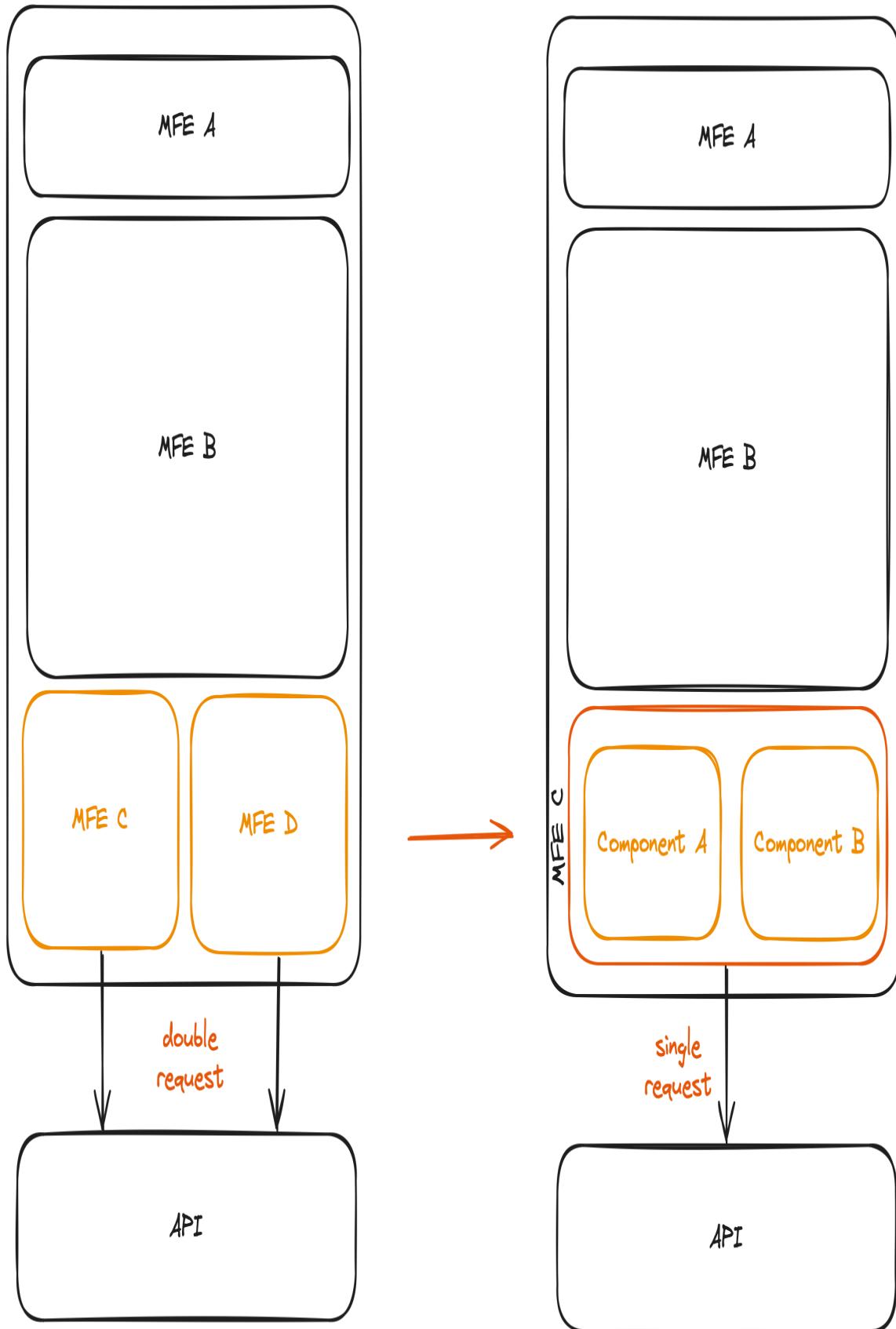
After discussing how micro-frontends can fit with multiple backend architectures, we must address some topics that are architecture-agnostic but could help with the successful integration of a micro-frontends architecture.

### Multiple micro-frontends consuming the same API

When working with a horizontal-split architecture, we might encounter situations where similar micro-frontends exist within the same view, consuming identical APIs with the same payload. This scenario could lead to an increase in backend traffic, necessitating more complex solutions for managing the traffic and its associated costs.

In such instances, it's crucial to question whether maintaining separate micro-frontends truly adds value to our system. Is grouping them into a single, unified micro-frontend a more effective approach?

A possible solution is transforming the micro-frontends into components and consolidating them within a single micro-frontend, as depicted in [Figure 7-21](#).



*Figure 7-21. When multiple micro-frontends are consuming the same API with identical payloads, a solution could involve leveraging a single micro-frontend to inject the API response into newly created components, thereby reducing the application's chattiness towards the server.*

In this scenario, the micro-frontends will execute a single request to the API, which will then inject the response into the components within the page, as we are accustomed to implementing with other architectures. These components can be easily imported by the micro-frontends as an NPM library, maintaining clear boundaries and reducing redundant API calls.

Additionally, consider reassessing team ownership. Implementing this solution may increase the team's cognitive load because of the new micro-frontends containing additional components and handling more business requirements.

Typically, such situations should prompt consideration for architectural enhancement. Do not overlook this signal; instead, reassess the decisions made at the project's outset with the available information and context, ensuring that making duplicate API requests within the same view is acceptable. If not, be prepared to reevaluate the boundaries of the micro-frontends.

## **APIs come first, then the implementation**

Independently of the architecture we will implement in our projects, we should apply API-first principles to ensure all teams are working with the same understanding of the desired result.

An API-first approach means that for any given development project, your APIs are treated as “first-class citizens.”

As discussed at the beginning of this book, we need to make sure the API identified for communicating between micro-frontends or for client-server communication are defined up front to enable our teams to work in parallel and generate more value in a shorter time.

In fact, investing time at the beginning for analyzing the API contract with different teams will reduce the risk of developing a solution not suitable for

achieving the business goals or a smooth integration within the system.

Gathering all the teams involved in the creation and consumption of new APIs can save a lot of time further down the line when the integration starts.

At the end of these meetings, producing an API spec with mock data will allow teams to work in parallel.

The team that has to develop the business logic will have clarity on what to produce and can create tests for making sure they will produce the expected result, and the teams that consume this API will be able to start the integration, evolving or developing the business logic using the mocks defined during the initial meeting.

Moreover, when we have to introduce a breaking change in an API, sharing a **request for comments** (RFC) with the teams consuming the API may help to update the contract in a collaborative way. This will provide visibility on the business requirements to everyone and allow them to share their thoughts and collaborate on the solution using a standard document for gathering comments.

RFCs are very popular in the software industry. Using them for documenting API changes will allow us to scale the knowledge and reasoning behind certain decisions, especially with distributed teams where it is not always possible to schedule a face-to-face meeting in front of a whiteboard.

RFCs are also used when we want to change part of the architecture, introduce new patterns, or change part of the infrastructure.

## API consistency

Another challenge we need to overcome when we work with multiple teams on the same project is creating consistent APIs, standardizing several aspects of an API, such as error handling.

API standardization allows developers to easily grasp the core concepts of new APIs, minimizes the learning curve, and makes the integration of APIs from other domains easier.

A clear example would be standardizing error handling so that every API returns a similar error code and description for common issues like wrong body requests, service not available, or API throttling.

This is true not only for client-server communication but for micro-frontends too. Let's think about the communication between a component and a micro-frontend or between micro-frontends in the same view.

Identifying the events schema and the possibility we grant inside our system is fundamental for the consistency of our application and for speeding up the development of new features.

There are very interesting insights available online for client-server communication, some of which may also be applicable to micro-frontends. [Google](#) and [Microsoft](#) API guidelines share a well-documented section on this topic, with many details on how to structure a consistent API inside their ecosystems.

## **Web socket and micro-frontends**

In some projects, we need to implement a WebSocket connection for notifying the frontend that something is happening, like a video chat application or an online game.

Using WebSockets with micro-frontends requires a bit of attention because we may be tempted to create multiple socket connections, one per micro-frontend. Instead, we should create a unique connection for the entire application and inject or make available the WebSocket instance to all the micro-frontends loaded during a user session.

When working with horizontal-split architectures, create the socket connection in the application shell and communicate any message or status change (error, exit, and so on) to the micro-frontends in the same view via an event emitter or custom events for managing their visual update.

In this way, the socket connection is managed once instead of multiple times during a user session. There are some challenges to take into consideration, however.

Imagine that some messages are communicated to the client while a micro-frontend is loaded inside the application shell. In this case, creating a message buffer may help to replay the last few messages and allow the micro-frontend to catch up once fully loaded.

Finally, if only one micro-frontend has to listen to a WebSocket connection, encapsulating this logic inside the micro-frontend would not cause any harm because the connection will leave naturally inside its subdomain.

For vertical-split architectures, the approach is less definitive. We may want to load inside every micro-frontend instead of at the application shell, simplifying the lifecycle management of the socket connection.

## **The right approach for the right subdomain**

Working with micro-frontends and microservices provides a level of flexibility we didn't have before.

To leverage this new quality inside our architecture we need to identify the right approach for the job.

For instance, in some parts of an application, we may want to have some micro-frontends communicating with a BFF instead of a regular service dictionary because that specific domain requires an aggregation of data retrievable by existing microservices but the data should be aggregated in a completely different way.

Using micro-architectures, these decisions are easier to embrace due to the architecture's intrinsic characteristic. To grant this flexibility, we must invest time at the beginning of the project analyzing the boundaries of every business domain and then refine them every time we see complications in API implementation.

In this way, every team will be entitled to use the right approach for the job instead of following a standard approach that may not be applicable for the solution they are developing.

This is not a one-off decision but it has to evolve and revise with a regular cadence to support the business evolution.

## Summary

We have covered how micro-frontends can be integrated with multiple API layers.

Micro-frontends are suitable for not only microservices but also monolith architecture.

There may be strong reasons why we cannot change the monolithic architecture on the backend but we want to create a new interface with multiple teams. Micro-frontends may be the solution to this challenge.

We discussed the service dictionary approach that could help with cross-platform applications and with the previous layer for reducing the need for a shared client-side library that gathers all the endpoints. We also discussed how BFF can be implemented with micro-frontends and a different twist on BFF using API gateways.

In the last part of this chapter, we reviewed how to implement GraphQL with micro-frontends, discovering that the implementation overlaps quite nicely with the one described in the API gateway and BFF patterns.

Finally, we closed the chapter with some best practices, like approaching API design with an API-first approach and leveraging DDD at the infrastructure level for using the right technical approach for a subdomain.

As we have seen, micro-frontends have different implementation models based on the backend architecture we choose.

The quickest approach for starting the integration in a new micro-frontends project is the service dictionary that can evolve overtime to more sophisticated solutions like BFF or GraphQL.

Remember that every solution shared in this chapter brings a fair amount of complexity if not analyzed and contextualized inside the organization structure and communication flow. Don't focus your attention only on the technical implementation but move a step further by looking into the governance for future APIs integration or breaking changes of an API.

# Chapter 8. Common Anti-Patterns in Micro-Frontend Implementations

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 10th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the author at [building.microfrontends@gmail.com](mailto:building.microfrontends@gmail.com).

Over the past ten years, I’ve had the privilege of guiding hundreds of teams worldwide through their micro-frontend journeys, witnessing both triumphs and pitfalls firsthand. These experiences, coupled with insights from discussions with industry experts, have shed light on crucial aspects often overlooked - organizational capabilities, platform considerations, and the socio-technical aspects of distributed systems in general.

While many teams eagerly dive into micro-frontend implementations, looking to the newest framework or the latest library to incorporate into their projects, a hard truth emerges: the technical implementation is merely the tip of the iceberg ([Figure 8-1](#)). The real challenges lie beneath the surface, in the realms of organizational dynamics, communication patterns, and architectural decision-making.

Marketing Message

New Products

Offers

Best Sellers

*Figure 8-1. The technical implementation is just one aspect to take into account when embracing micro-frontend architectures*

Let's delve into the most common micro-frontend anti-patterns, focusing on the technical pitfalls that can derail even the most promising projects. These anti-patterns are the ones I've encountered most frequently with teams embracing this approach for the first time or struggling to progress with implementation due to constant friction created by them. You may have encountered some of these anti-patterns in previous chapters, but this collection will offer a more in-depth exploration of each one.

We'll examine common mistakes that have emerged across different projects and industries. Understanding these pitfalls will help you navigate the complexities of micro-frontend architectures more effectively. While we'll focus mainly on technical aspects here, remember that organizational and strategic factors, which we'll discuss in later chapters, are just as crucial for success.

Please remember that culture, organizational structure, and software architecture are tightly coupled. When one of these dimensions shifts in a different direction, the other two will inevitably be impacted. The effects might not be immediate, but they will eventually surface. Therefore, always consider these three dimensions before making any foundational decision.

In this chapter, we'll explore specific anti-patterns and provide guidance on how to avoid them, ensuring a more successful micro-frontends implementation.

## Micro-frontend or component?

Picture this: you've decided to embrace micro-frontends for your next project, and now you're faced with the crucial task of defining the boundaries of your independent units. Where do you begin? How "micro" should a micro-frontend really be?

This is a common challenge that every organization transitioning to a micro-frontend architecture encounters. I completely understand the

complexity of this problem, but rest assured, there are effective strategies for identifying and testing appropriate boundaries for micro-frontends without confusing them with mere components.

We explored some helpful heuristics in Chapter 2, “Testing Your Micro-Frontend Boundaries,” but I’d like to provide a more concrete example here. This is because determining the right granularity for micro-frontends is, by far, the most common pitfall in their design and implementation.

Take a look at the homepage of an e-commerce in [Figure 8-2](#).

Header

Hero Product

Product A

Product B

Product C

Experience carousel

90 days return policy banner

Where to buy

Support

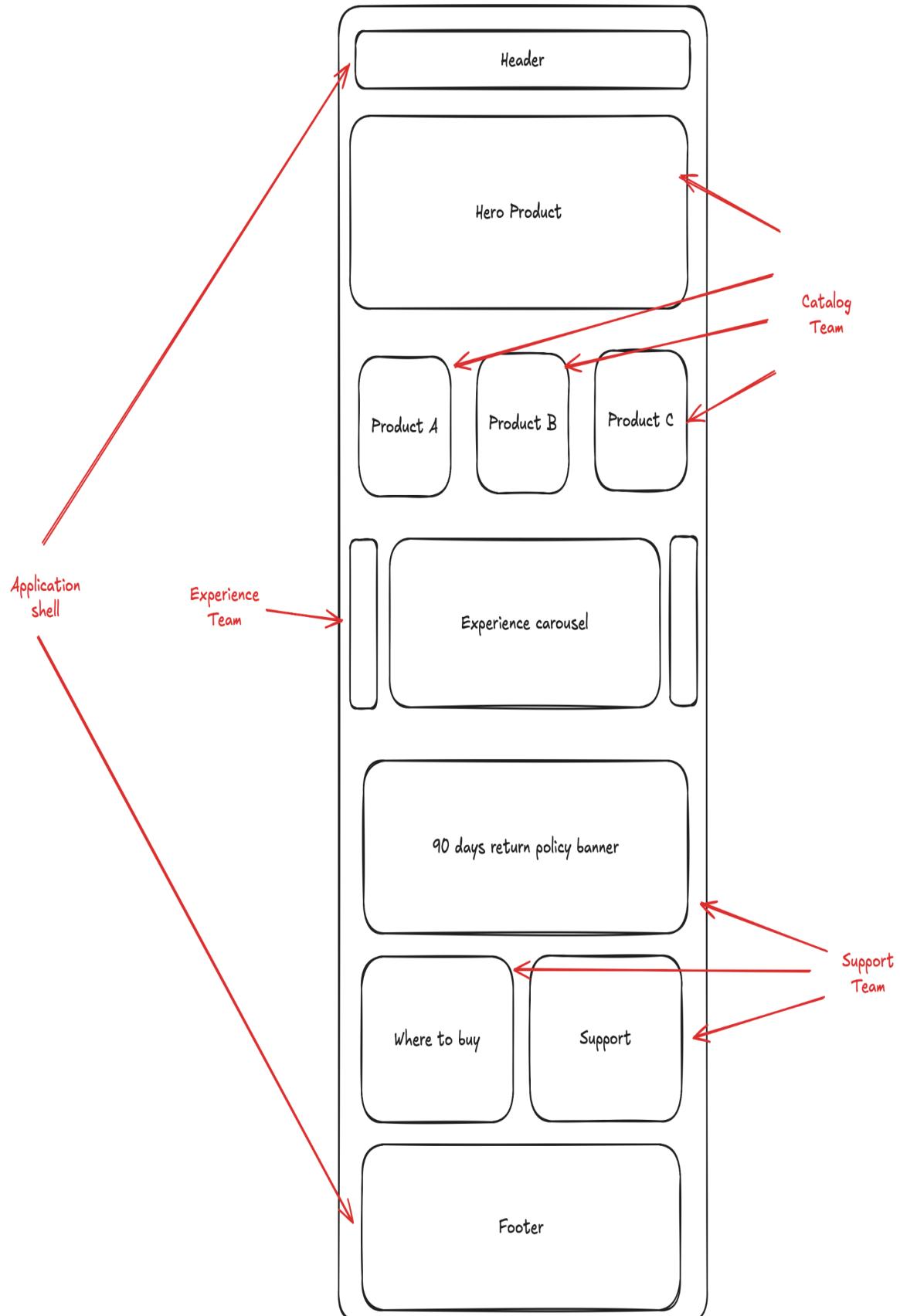
Footer

*Figure 8-2. An e-commerce homepage that wants to move from a monolithic approach to micro-frontends.*

An e-commerce homepage is a quintessential example of a user interface that typically encompasses multiple domains within an organization. The complexity of these domains can vary, but let's explore how we might divide this interface into micro-frontends.

In this scenario, the team has opted for a horizontal split approach for the homepage. This decision was driven by the need to involve multiple teams in the delivery of this page, allowing for parallel development and domain-specific expertise.

To begin, let's identify the distinct business domains that present within this view. This exercise will not only help us define our micro-frontend boundaries but also clarify which team will be responsible for each domain ([Figure 8-3](#)).



*Figure 8-3. There are 3 domains in this diagram plus the application shell for composing the final view*

The application shell contains the header and footer, elements that remain consistent across the entire website. After analyzing the GitHub history, the team discovered that these components have a low rate of change—typically once or twice a year. The changes are generally straightforward to implement, and extracting them as a shared library or loading them at runtime would introduce unnecessary complexity. Given their stability, the team decided to keep these components within the application shell, avoiding the need for additional governance and automation pipelines for rarely changing elements.

Beyond the shell, three primary domains emerge in this view: catalog, experience, and support. The catalog domain is represented by components showcasing the latest available products. While these could technically be encapsulated as individual micro-frontends, doing so might lead to over-segmentation without clear benefits. Consider the three product displays beneath the hero product component. Although they're replicas with consistent behavior and aesthetics, separating them into independent, runtime-deployed micro-frontends would likely introduce more complexity than advantages (different automation pipelines, coordination across team members, and so on).

These catalog components are best assigned to a single team responsible for catalog pages. This approach leverages domain expertise, allowing the team to align closely with product and business requirements for optimal product display.

A similar approach applies to the experience and support domains. As these components typically redirect users to their respective domain areas, it's logical to have domain experts create and maintain the associated micro-frontends that contain the components. This example illustrates how identifying domains within a view aids in understanding ownership and distinguishing between micro-frontends and components. The decision-

making process is informed by business requirements, change frequency, and domain ownership.

It should be clear that a page with multiple components often translates to fewer micro-frontends than one might initially assume. When using a horizontal split approach, having more than 5-7 micro-frontends on a single page often indicates overly fine-grained division. This can lead to domain leakage between micro-frontends, creating deployment coupling and external dependencies across teams.

Remember, these boundaries should evolve with your application. When friction arises, review whether the identified boundaries still suit your application's needs, or if aggregation or further splitting is necessary. Distributed systems are living entities that require nurturing and attention, much like children learning to manage their emotions.

## Sharing state between micro-frontends

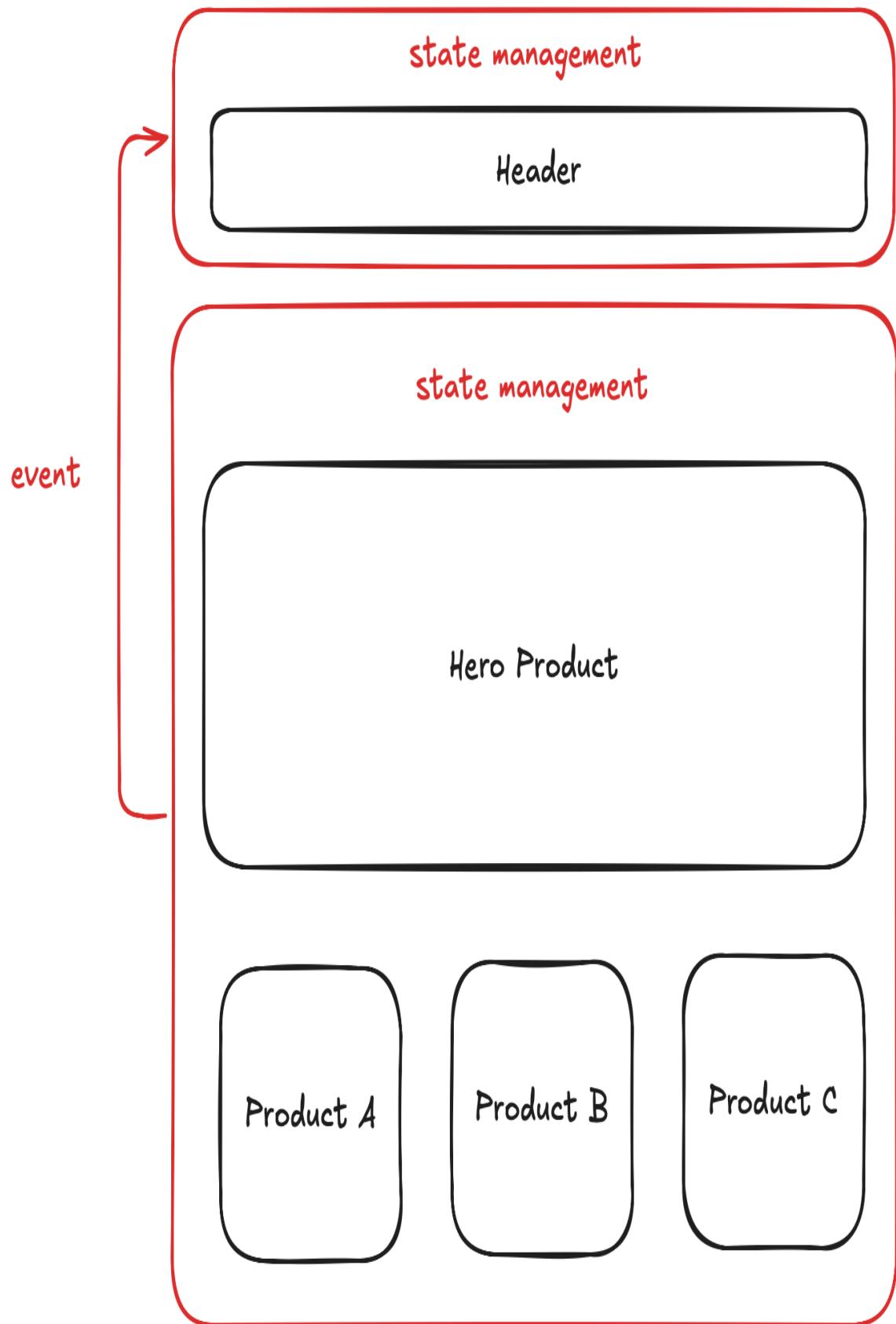
This issue frequently arises when new teams adopt micro-frontends. While we're used to designing single-page applications (SPAs) by first selecting a UI library and then choosing a suitable state management library, this approach may not be optimal for distributed systems. This anti-pattern typically emerges in horizontal splits where micro-frontends within the same view need to communicate in response to user actions or external API signals via sockets or server-sent events.

Sharing, in this context, is a form of coupling akin to design-time coupling in microservices. In microservices, design-time coupling refers to the extent to which one service must change due to alterations in another service. This coupling occurs when one service directly or indirectly depends on concepts owned by another service.

Similarly, in micro-frontends, we may encounter situations where modifying a data type in the shared state necessitates retesting multiple micro-frontends on the same page to ensure functionality. This scenario also introduces coordination challenges between teams when implementing

breaking changes in the shared state. Consider a deployment scenario where other teams rely on the data structure of the state manager. A breaking change by one team would require coordination with all affected teams to ensure compatibility with both current production versions and future iterations of their micro-frontends. If incompatibilities arise with the current production version, it may necessitate adding tasks to other teams' backlogs and waiting for them to incorporate the changes, potentially causing delays and complications.

A more efficient approach would be to implement a mechanism that keeps micro-frontends aligned, allowing them to react to user interactions or state changes in other micro-frontends ([Figure 8-4](#)). This method could potentially mitigate the challenges associated with shared state management in micro-frontend architectures.



*Figure 8-4. In this example there are two micro-frontends communicating via events and maintaining their internal state for reaching low-coupling and independence*

The publish-subscribe pattern, commonly abbreviated as pub-sub, embodies this concept. This messaging pattern is widely implemented in software architecture to facilitate asynchronous communication between various system components.

To maintain the independence of micro-frontends within the same view, it's highly recommended to encapsulate the state manager within each micro-frontend, in this way the micro-frontend can be evolved independently without the need to coordinate the changes across multiple teams.

Communication between these micro-frontends can then be established using mechanisms such as event emitters or custom events for instance.

Event emitters are generally preferable to custom events due to their DOM-agnostic nature. This characteristic offers significant advantages in micro-frontend architectures. Custom events, by design, bubble through DOM elements up to the window object. This behavior can lead to complications if event propagation is inadvertently prevented at any point in the DOM tree. In such cases, identifying and debugging the issue can become a challenging and time-consuming task.

In contrast, event emitters operate on a subscription model, where components directly subscribe to and notify subscribers. This approach is independent of the DOM structure, providing greater flexibility and reliability. As a result, you can freely move your micro-frontend within the DOM tree without risking event loss or necessitating code refactoring.

This DOM-agnostic quality of event emitters enhances the modularity and maintainability of your micro-frontend architecture. It allows for more robust communication between components, reducing the likelihood of unforeseen issues related to DOM structure changes. Ultimately, using event emitters can lead to a more resilient and easier-to-maintain system, particularly in complex micro-frontend environments where component placement and interaction are critical considerations. This approach

preserves the autonomy of individual micro-frontends while allowing for necessary interactions.

However, it's crucial to exercise caution regarding the frequency and volume of communication across micro-frontends. Excessive inter-micro-frontend communication often indicates improperly defined boundaries within the view. Striking the right balance is essential for maintaining a clean and efficient architecture. When implementing this pattern, carefully consider the appropriate level of interaction to ensure optimal system design and functionality.

## Micro-frontends anarchy

This anti-pattern typically manifests when organizations embrace the idea of independent development without establishing proper guidelines, standards, or governance structures. Sometimes organizations want to optimize their architecture for a multi-framework approach without realizing the long-term impact in this decision.

Let me ask you a question, would you use a multi-framework approach for an SPA?

Your answer is probably, but technically you could do that!

In a micro-frontends anarchy scenario, different teams within an organization start developing their parts of the application with complete autonomy. While this autonomy can foster innovation and allow teams to move quickly, it can also lead to a fragmented and inconsistent application architecture.

In the absence of clear guidelines, teams may adopt a wide array of technologies, frameworks, and libraries. While this diversity can be beneficial in some cases, it often leads to a chaotic tech stack. You might find one team using React, another using Vue.js, and yet another opting for Angular – all within the same application. This proliferation of technologies can result in an increased bundle size due to multiple framework libraries

being loaded and an inconsistent user experience across different parts of the application.

As the application grows and evolves, the maintenance challenges posed by an anarchic micro-frontends approach become increasingly apparent. The fragmented nature of the architecture introduces significant hurdles in managing and updating the system as a whole. Implementing application-wide changes or upgrades becomes a daunting task, requiring coordination across multiple teams and potentially incompatible technologies.

Debugging issues that span multiple micro-frontends grows in complexity, often requiring developers to navigate through disparate codebases and technologies to identify and resolve problems.

Maintaining consistent security practices across all micro-frontends becomes a concern, as each team may implement security measures differently, potentially creating vulnerabilities in the overall application. The integration and testing of micro-frontends developed with different technologies introduce additional complications, requiring sophisticated testing strategies and potentially custom integration solutions.

One of the often-overlooked consequences of the micro-frontends anarchy anti-pattern is its impact on knowledge sharing and developer mobility within an organization.

As teams become deeply entrenched in their chosen technologies, knowledge silos begin to form. This fragmentation leads to several significant challenges in the development process. Sharing best practices across teams becomes increasingly difficult due to the differing technological contexts, limiting the spread of valuable insights and innovations.

The ability to move developers between teams is greatly reduced, as each micro-frontend requires a unique skill set, hampering organizational flexibility and career growth opportunities. This fragmentation of knowledge not only impacts the efficiency of the development process but can also lead to a sense of isolation among teams. The resulting silos can potentially affect morale and hinder collaboration, as developers may feel

disconnected from the broader organizational goals and their peers working on other parts of the application. Finally, this knowledge fragmentation can undermine the very benefits that micro-frontends were intended to provide, such as increased agility and improved team autonomy.

However, a multi-framework approach in micro-frontends can be a valuable strategy in specific scenarios, particularly during transitional phases in an organization's technical evolution. This approach, while not ideal for long-term implementation, offers significant benefits in certain contexts. One prime scenario for leveraging a multi-framework approach is during the migration of a monolithic system to a micro-frontends architecture. In this case, organizations can gradually transition different parts of their application to micro-frontends while maintaining the existing system, as described in chapter 12. This incremental approach allows for continuous delivery of value to customers without the need for a complete system overhaul, which could potentially take months or even years.

Similarly, when an organization decides to migrate from one UI framework to another, like from a version of Angular to a new one for instance, a multi-framework approach can be instrumental. It enables teams to begin developing new features or refactoring existing ones using the new framework while keeping the rest of the application functional in the original framework. This strategy allows for a smoother transition, minimizing disruption to the user experience and maintaining business continuity.

Moreover, this approach provides a safety net during the transition period. If critical issues arise in the newly developed micro-frontends, teams can quickly revert to the original implementation. This fallback option reduces risk and provides peace of mind during the migration process, allowing teams to be more adventurous in their approach to new technologies and architectures.

However, it's crucial to view the multi-framework approach as a temporary solution rather than a long-term architectural strategy. Organizations should have a clear plan to eventually converge on a more standardized approach

to avoid the pitfalls associated with long-term maintenance of diverse technologies within the same application.

In conclusion, while a multi-framework approach in micro-frontends comes with its challenges, it can be a powerful tool for organizations undergoing significant technological transitions. By enabling gradual migration, continuous delivery, skill development, and risk mitigation, this approach can bridge the gap between legacy systems and modern architectures, ensuring that businesses can evolve their technology stacks without compromising their ability to deliver value to customers.

## **Anti-corruption layer to the rescue**

You are tasked to integrate an existing application into your brand new micro-frontends architecture. The main problem is that you have to integrate it in just a month.

The application shell is framework agnostic; however, the decision was to use an event emitter for communicating across micro-frontends, and the existing web application is not designed with micro-frontends in mind. Therefore, due to the time constraints and lack of understanding of the impact of loading the application from the application shell, the decision is to use an iframe to isolate the code and avoid potential runtime errors while using the micro-frontends system.

In this case, the challenge is deciding if the application shell should support the iframe communication via postMessage API or continue to work with the event emitter but refactor the existing application, the complexity of which the team is not aware of.

Another factor to take into account is that at some point in the future, the existing application will be refactored to micro-frontends, so the code to support the iframe in the application shell will eventually be thrown away.

This approach might open the door to multiple ways to compose micro-frontends and create complexity in the long term because the application shell, which should be the most stable part of the system, is constantly

subject to changes and new implementations, causing it to test all the integration points on every single change. Moreover, multiple teams could ask to introduce other ways to compose micro-frontends considering there isn't a single one. As you can imagine, this simple decision could open difficult discussions in the future for evolving the architecture.

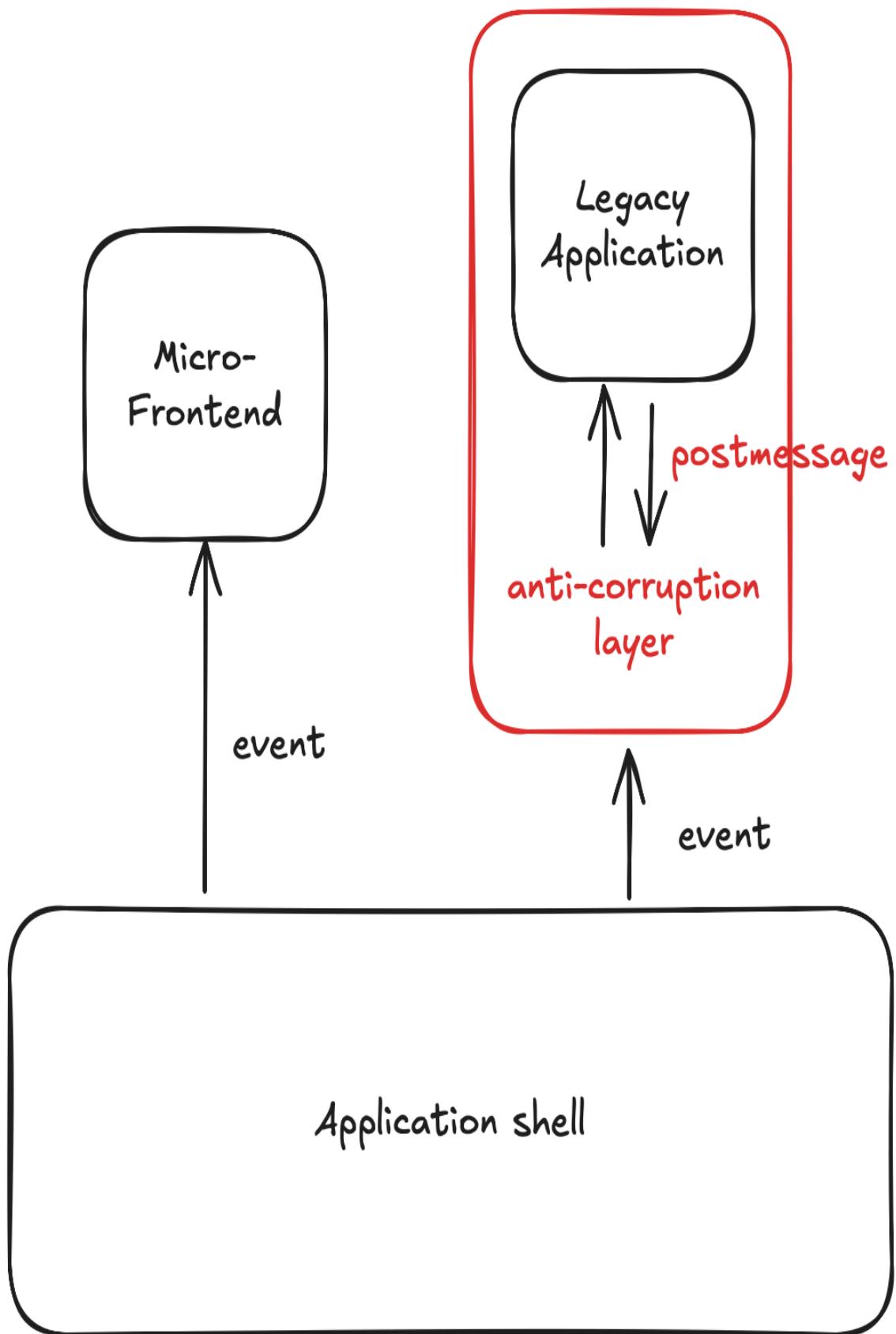
This problem is more common than you think. It happens when a company acquires another one or when the C-suite wants to create a more consistent experience for their customers or for internal systems. teams usually jump straight away to the solution, without evaluating the trade-offs they will need to live with in the long run.

The question is, do we have any alternative?

Luckily, the answer is yes, we do. The alternative is called the anti-corruption layer.

The Anti-Corruption Layer (ACL) pattern is a fantastic architectural strategy that helps keep your systems clean and manageable, especially when integrating different technologies or legacy systems. Traditionally associated with backend architectures, the ACL acts as a translator and mediator, ensuring that the complexities and potential inconsistencies of one system do not “corrupt” the integrity of another. This pattern is especially valuable when dealing with legacy systems that cannot be easily modified or replaced, allowing new systems to evolve independently while maintaining necessary interactions.

In micro-frontends, we can use the ACL by creating a wrapper around the iframe that contains the existing application and leverage the postMessage communication between the anti-corruption layer and the iframe, while using the event emitter for the communication between micro-frontends and the shell ([Figure 8-5](#)).



*Figure 8-5. The application shell codebase remains the same across all the micro-frontends and the legacy system is well isolated in an iframe that communicates with its wrapper via the postmessage API.*

In this way, the application shell codebase will remain the same for all micro-frontends. The wrapper around the legacy system will sanitize the communication between the application shell, other micro-frontends, and the iframe without leaking implementation details. Finally, when the legacy application is refactored, the only change needed will be to load a new micro-frontend endpoint instead of the anti-corruption layer. This is a very neat approach that is evolutionary by design.

One of the biggest perks of adopting an ACL in your frontend architecture is the consistency it brings to your application. By establishing a clear interface for all external interactions, you create a stable foundation that makes it easier to maintain and evolve your code over time. Plus, if you ever need to update or replace an external service, you can do so without disrupting the rest of your application. This flexibility is invaluable in today's fast-paced development environment, where change is the only constant.

## Unidirectional sharing

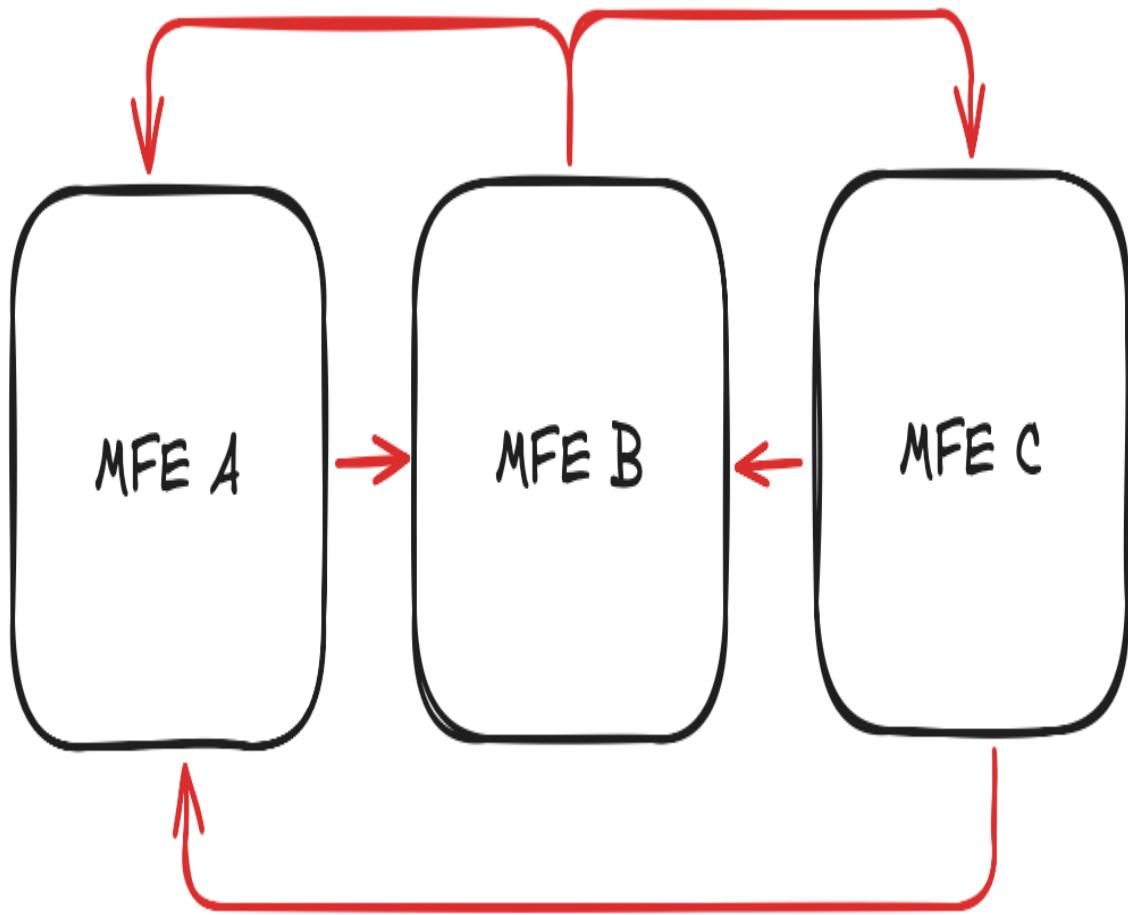
The next anti-pattern is common when we treat micro-frontends and properties like components, ready to be shared everywhere. Even when you want to reuse components in real-time and share across other micro-frontends, we always have to bear in mind how we are sharing our data and resources.

Bi-directional or omni-directional sharing across micro-frontends can quickly become a double-edged sword. While it might seem convenient at first, it often leads to a tangled web of dependencies that can stifle development speed and introduce unexpected bugs. The core issue lies in the increased communication and coordination burden placed on development teams. When multiple micro-frontends share data and functionality in multiple directions, changes in one component can have far-

reaching and often unforeseen consequences on others. This scenario necessitates constant communication between teams, potentially leading to bottlenecks in the development process.

The challenges of bi-directional sharing extend beyond mere inconvenience. They can fundamentally undermine the very benefits that micro-frontend architecture aims to provide. One of the primary goals of adopting micro-frontends is to enable teams to work independently, allowing for faster development cycles and easier maintenance. However, when components are tightly coupled through bi-directional dependencies, this independence is compromised.

Moreover, consider the impact on testing and deployment. In a system with extensive bi-directional sharing, a change in one micro-frontend might require comprehensive testing across all interconnected components. This not only slows down the development process but also increases the risk of introducing bugs that are difficult to isolate and fix. Furthermore, it complicates the deployment process, as teams must coordinate their release schedules to ensure compatibility across all shared interfaces ([Figure 8-6](#)).



*Figure 8-6. Omnidirectional sharing will harm your deployment and rollback strategies due to shared dependencies.*

Another significant drawback is the potential for circular dependencies. When micro-frontends share data and functionality in both directions, it's easy to inadvertently create loops where component A depends on B, which in turn depends on C, which then depends back on A. Such circular dependencies are notoriously difficult to manage and can lead to runtime errors, performance issues, and debugging nightmares.

In contrast, a unidirectional flow of data from a parent component to its children would localize the impact of every change. The application shell will pass, through dependency injection, only the necessary data to each micro-frontends.

Moreover, unidirectional data flow promotes a clearer mental model of the application's structure. Developers can more easily reason about data changes and their effects when they know that information flows in only one direction. This clarity not only aids in development but also simplifies debugging and maintenance tasks.

Bear in mind that events with a pub-sub approach, like an event emitter, won't follow this rule because they are hierarchy unaware. The pattern removes the idea of relations and lets micro-frontends subscribe and be notified to events.

It's worth noting that adopting a unidirectional sharing approach doesn't mean completely isolating micro-frontends from each other. Instead, it encourages thoughtful design of component interfaces and promotes the use of well-defined APIs for necessary inter-component communication. This strategy strikes a balance between component independence and system cohesion, allowing teams to work autonomously while still creating a unified user experience.

By embracing unidirectional sharing from parent, the application shell, to child components, every micro-frontend, development teams can maintain the independence and flexibility that make micro-frontends so powerful, while still creating cohesive and feature-rich applications.

## Premature abstraction

Sharing or not sharing, this is the dilemma. When developing micro-frontends, it's crucial to approach abstraction with caution, particularly when considering whether to encapsulate specific functionality or utilities in a shared library. While abstraction can promote code reuse and maintainability, premature or restless abstraction can lead to unforeseen challenges. The real test isn't in the initial implementation, but in the long-term maintenance and evolution of these shared dependencies.

Kent C. Dodds, full time educator very well known for his contribution on React and Remix, captures this concept with his AHA programming

principle, which stands for “Avoid Hasty Abstractions”. This approach encourages developers to resist the urge to abstract code too quickly, instead allowing patterns to emerge naturally over time. By doing so, teams can create more robust and flexible abstractions that truly serve their needs.

In the context of micro-frontends, hasty abstractions in shared libraries can create a ripple effect of compatibility issues across multiple teams and components. For instance, consider a scenario where a team creates a shared date formatting utility. Initially, it might seem like a straightforward abstraction that could benefit multiple micro-frontends. However, as different teams begin to rely on this utility, they may discover edge cases or require slight variations in formatting. Suddenly, what started as a simple utility becomes a complex, over-engineered solution trying to accommodate every possible use case. When this library is used in multiple micro-frontends, you risk forcing other teams to update it quickly due to critical bugs or enhancements, without knowing how busy their backlogs are or if they can accommodate the update.

By embracing the AHA principle, teams can avoid these pitfalls. Instead of rushing to create shared libraries, they might start by duplicating code across micro-frontends where necessary. This approach allows each team to tailor the functionality to their specific needs while providing the opportunity to observe common patterns over time. Once clear, repeated use cases emerge across multiple micro-frontends, teams can then create thoughtful, well-designed abstractions that truly serve the project’s needs.

Remember, these decisions must be taken into consideration the trade-offs and the intention you want to express sharing the library. The goal is to strike a balance between code reuse and flexibility. By being mindful of when and how to abstract shared functionality in micro-frontends, teams can create more maintainable, evolvable systems that stand the test of time and changing requirements.

## Summary

When embarking on a micro-frontend journey, it's essential to ensure that your architectural choices align seamlessly with your business objectives and system requirements.

This alignment involves a thoughtful evaluation of several critical factors. You'll want to consider your application's scalability needs, assessing whether the projected growth justifies the added complexity that micro-frontends introduce. Equally important is your organization's ability to support autonomous teams for each micro-frontend, as this structure is fundamental to realizing the full benefits of this architecture.

By carefully weighing these aspects and understanding the associated trade-offs, you'll be well-equipped to make an informed decision about implementing micro-frontends in your system. Remember, there's no universal solution in architecture – the key lies in tailoring your approach to your specific business needs and technical constraints.

# Chapter 9. Migrating to Micro-Frontends

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 11th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the author at [building.microfrontends@gmail.com](mailto:building.microfrontends@gmail.com).

Let’s be honest: the idea of migrating a large, established application to micro-frontends can feel overwhelming. Maybe you’ve heard stories of migrations stretching on for months (or even years), sapping team energy and putting business goals on hold. It’s no wonder so many teams hesitate to get started.

But it doesn’t have to be that way!

One of the best things about distributed systems and micro-frontends in particular is that you don’t have to do everything at once. In fact, the most successful migrations are usually *iterative*. You can start small, deliver value quickly, and keep your users happy along the way. With the right approach, you can see real business impact in just a few weeks.

For this chapter, I want to try something a little different. I’ve collected the most common questions I’ve heard from teams over the past five years. Questions about what to migrate first, how to avoid breaking things, and

how to keep everyone on the same page. If you’re wondering about it, chances are someone else has asked it before.

Most people don’t read a technical book cover-to-cover (despite the fact that you should with this book). You might be flipping straight to the section you need right now. Hence why I’ve organized this chapter around real questions, with clear answers and practical tips you can use right away.

## Why Go Iterative?

One of the strengths of micro-frontends is making change easier. By breaking your app into smaller, independent pieces, you give your teams more freedom to experiment, adopt new tech, and scale up smoothly. And best of all, you can migrate one piece at a time. No need for a risky, all-at-once “big bang.”

Here’s why an iterative migration is usually the way to go:

- **Faster wins:** You can start delivering improvements to your users and stakeholders right away while creating momentum and a winning mindset within your teams.
- **Lower risk:** Small, focused changes are easier to test and roll back if something goes wrong.
- **Continuous learning:** Each step is a chance to learn what works (and what doesn’t) before moving on. This will help to build or configure the different tools and practices we discussed in this book. Not everything has to be there on day one.
- **Maintain business continuity:** You don’t have to freeze all new development while you migrate. You can keep building features and fixing bugs as you go.

To make this chapter as practical and actionable as possible, I’ve structured each question using a consistent format. Here’s what you can expect for every topic:

- **Best practices and patterns:** Actionable advice and proven strategies you can use right away, including real-world techniques that have worked for other teams.
- **Trade-offs and pitfalls:** An honest look at what could go wrong, common mistakes to watch out for, and the trade-offs you'll need to consider as you make decisions.
- **Checklist:** A quick set of questions or steps to help you apply the guidance to your own situation, so you can move from theory to action.
- **A quick example:** A scenario or mini case study to ground the advice in a real-world context and help you visualize how it might work in practice.
- **Personal experience:** Where it's relevant, I'll share stories from my own journey. What worked, what didn't, and what changed for me or my teams along the way.

This structure is designed to help you quickly find the information you need, understand the reasoning behind each recommendation, and feel confident applying these lessons to your own migration.

Following is a list of questions we are going to cover in this chapter:

1. Which problem(s) are you trying to solve with micro-frontends?
2. How can I get the management on board with this migration?
3. How should I actually plan the migration?
4. How do you decide which modules or features to migrate first?
5. How do you maintain user experience and consistency during migration?
6. How do you handle shared dependencies and version management between legacy and micro-frontend code?

7. How do you manage cross-cutting concerns like auth, routing, or state during migration?
8. Do I need microservices to migrate to micro-frontends?

## Which problem(s) are you trying to solve with micro-frontends?

Migrating is a significant investment, so make sure you're solving real, pressing problems, not just following a trend. The main drivers often include team autonomy, faster release cycles, scalability, tech stack flexibility, and improved maintainability. If your current monolithic frontend is slowing down releases, making it hard for teams to work independently, or struggling to scale with your business, these are strong signals that micro-frontends could help in your context.

## Best Practices and Patterns

- **Team autonomy and parallel development:** Micro-frontends let multiple teams own and release parts of your system independently, removing bottlenecks where one team's delay holds up everyone else.
- **Faster, safer releases:** Independent deployment means you can release updates to one part of your app without risking the whole system. This isolation reduces the blast radius of bugs and makes rollbacks easier.
- **Scalability:** As your organization grows, micro-frontends scale with you. Teams can be added or reorganized around business domains without stepping on each other's toes.
- **Improved maintainability:** Smaller, focused codebases are easier to maintain, test, and refactor. Teams can specialize and take full ownership of their domains.

- **Faster onboarding:** Because the system is split, the cognitive load for new joiners is lower compared to a monolithic application. I've seen new teams contribute production-ready features within weeks instead of months.

## Trade-offs and Pitfalls

- **Increased complexity:** Micro-frontends introduce new challenges in dependency management, debugging, and environment configuration. Without strong governance, you risk creating a “Frankenstein” user experience that doesn't benefit your users.
- **Overhead for small teams/apps:** If your team or app is small, the overhead of micro-frontends may outweigh the benefits. Sometimes, simply splitting the app by routes or page groups can offer most of the advantages without the added complexity.
- **Shared code and dependencies:** Managing shared libraries across micro-frontends can lead to larger bundle sizes or version conflicts. Solutions like Module Federation or Import Maps can definitely help because they employ mechanisms to simplify the dependencies sharing with minimal to no configuration. Remember, with distributed systems we are aiming for reduced external dependencies and a fast flow. Starting with some duplication where it makes sense allows you to make more informed decisions about what to abstract later. Don't rush into premature abstractions. Gather data before introducing shared dependencies.
- **Organizational readiness:** Micro-frontends work best when your organization is ready for distributed ownership and has mature coordination and communication processes.

## Checklist

- Is your frontend codebase large and growing?

- Do multiple teams need to work independently on different features?
- Are release cycles slowed down by dependencies between teams?
- Do you want to adopt new technologies without a full rewrite?
- Is scaling your engineering organization a priority?
- Are you struggling with performance due to large bundle sizes or slow builds?
- Is your business domain complex enough to justify splitting into separate modules?

## A Quick Example

Let's say you're working at a fast-growing ecommerce platform. Every new feature requires coordination across several teams, causing delays and frustration. Releases are infrequent and risky. The codebase is increasingly hard to maintain. Sound familiar?

By moving to micro-frontends, each product area, like the product catalog, checkout, and user profile, becomes a separate module owned by an autonomous team. Releases speed up, teams can experiment with new technologies, and the platform scales more effectively as the business grows.

## Personal Experience

I've personally experienced a 10x increase in deployments after moving to micro-frontends. Implementing tools like canary releases, blue/green deployments, and feature flags enabled developers to deploy safely to production, leading to greater confidence and more testing with real users. We moved from a few releases per month to tens per micro-frontend per month-sometimes even per day!

Migrating to micro-frontends is best suited for organizations struggling with team autonomy, release speed, scalability, or tech stack flexibility. While the benefits can be significant, it's important to weigh them against the added complexity and ensure your organization is ready for the transition.

## How can I get the management onboard with this migration?

From experience, selling an iterative migration to the business is far easier than pitching a risky “big bang” rewrite. Why? Because business stakeholders care about outcomes-faster time-to-market, lower risk, and delivering value to customers, more than the technical elegance of a fresh start.

When you tie your migration strategy directly to business goals, you build trust and buy-in, which is absolutely critical for success.

## Best Practices and Patterns

- **Speak the business language:** When pitching this approach, focus on outcomes that matter to stakeholders, like faster releases mean quicker response to market changes and customer needs, or better ROI by spreading investment and value over time instead of waiting months or years for a payoff.
- **Start with a Proof of Concept (PoC):** Start with a small, high-impact feature or module as a pilot. This demonstrates value early, builds confidence, and gives you a template for the rest of the migration.

## Trade-offs and Pitfalls

Remember that with an iterative migration you will gain immediate value, safer releases, short feedback cycles, and easier alignment with business milestones. However, you will face more complex integration (the legacy

and new micro-frontends coexist for a while), which requires careful planning and communication, and you need to avoid “split brain” scenarios where two systems do the same thing for too long.

Remember to avoid focusing only on technical wins. Always connect migration steps to business goals, set clear milestones, celebrate progress to keep up the morale internally, and finally, make sure teams are aligned and understand the shared vision.

## Checklist

- Have you mapped business goals to migration milestones?
- Can you identify a pilot area that will deliver visible value quickly?
- Do you have stakeholder buy-in for an incremental approach?
- Are you set up to measure and communicate progress regularly?
- Is there a clear plan for routing, integration, and retiring legacy code?

## A Quick Example

Let’s say you’re working at a fast-growing streaming platform. Every new feature requires coordination across several teams, causing delays and frustration. Releases are infrequent and risky. The codebase is increasingly hard to maintain.

Instead of pitching a full rewrite, you propose migrating incrementally. You start with the video catalogue. With just three developers and three weeks, you build a micro-frontend version of the catalogue that runs on web, PlayStation, and Tizen. Not only does it match the old functionality, but it also performs better than the monolith. This quick win demonstrates value, builds trust, and sets the stage for further migration.

## Personal Experience

Back in 2015, when I pitched this architecture to the CTO of DAZN, I asked for three developers and three weeks. In that time, we recreated the video catalogue as a micro-frontend, running on web, PlayStation, and Tizen.

The result? We showed a clear improvement in performance compared to the monolith, and, crucially, we provided tangible business value in less than a month. This early success made it much easier to get the business on board for the rest of the migration.

Iterative migration is almost always the best path, both technically and from a business perspective. It's easier to sell, delivers value sooner, and reduces risk. The key is to speak the language of your stakeholders, tie migration steps to business outcomes, and start with a pilot that proves the value of your approach.

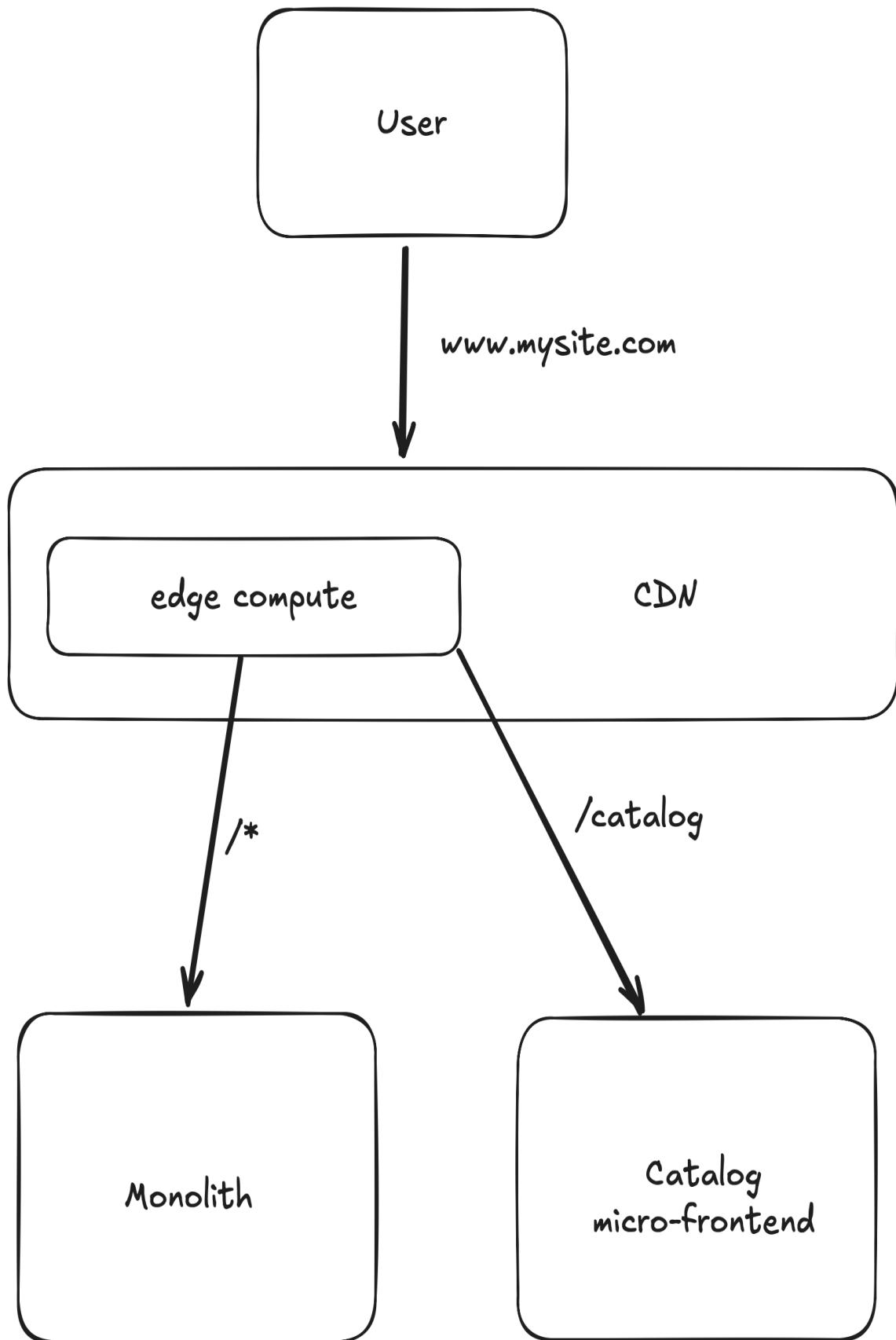
## How should I actually plan the migration?

Once you have agreement to move forward, planning the migration is where your choices will have the biggest impact on both the speed of delivery and the safety of your rollout. Business stakeholders want to see progress and value quickly, without risking the stability of your current system.

## Best Practices and Patterns

- **Favor entire pages or groups of pages:** Whether you choose a horizontal or vertical split, the most effective approach is to migrate and deploy entire pages or logical groups of pages at a time. This allows you to intercept user requests and route them to the new micro-frontend, minimizing integration complexity and making it easier to roll back if needed.

- **Single source of truth for routing:** By centralizing routing at the edge, you avoid having to keep routing logic in sync across multiple codebases. Any changes to routing can be made in one place, reducing the risk of inconsistencies and simplifying maintenance ([Figure 9-1](#)).
- **Use the Strangler Fig Pattern:** Leveraging a router at the edge helps you to implement a strangler fig pattern, a proven method for incremental migration. The strangler fig pattern introduces a façade or proxy that routes requests to either the legacy system or the new micro-frontends. Over time, more functionality is handled by the new system, and the old monolith is gradually phased out. This approach minimizes risk, allows for continuous delivery, and keeps the user experience seamless.
- **Progressive rollouts and release strategies:** Take advantage of advanced release strategies like canary releases, feature flags, or targeting specific user segments (such as a country or a percentage of users) to further de-risk your rollout. This way, you can validate new micro-frontends in production with real users, catch issues early, and build business confidence.
- **Faster rollbacks and safer releases:** If you need to roll back a migration, you can simply update the routing rule at the edge to point requests back to the monolith, with no redeployments or code changes required



*Figure 9-1. Configure routing at the edge instead of inside the code, so you won't have to maintain throwaway code in 2 codebases.*

## Trade-offs and Pitfalls

At all costs, avoid rendering micro-frontends and legacy code together on the same page or UI view. This increases integration complexity, can lead to inconsistent user experiences, and often slows down the migration process.

There is a clear possibility of having dependency clashes or unexpected UI behaviors that may occur at runtime. Despite it being tempting, it usually requires way more work to design, build, and maintain.

## Checklist

- Have you mapped your application into logical pages or groups of pages that can be migrated independently?
- Do you have the ability to route user requests at the CDN, edge, or server level?
- Do you have a rollback plan that can be executed quickly (ideally by just changing a routing rule)?

## A Quick Example

Suppose you're migrating a large e-commerce platform. Instead of rewriting the entire frontend or mixing micro-frontends into existing pages, you start with the checkout flow. You configure your edge compute service to route all `/checkout` requests to the new micro-frontend, while the rest of the site still runs on the monolith. If you encounter issues, rolling back is as simple as changing the routing rule to point back to the old system. You can even release the new checkout to just 5% of users or only to users in a specific country, giving you time to observe real-world performance and user feedback before a full rollout.

## Personal Experience

I have implemented several ways to apply a strangler pattern on frontend. So far, the easiest to build and maintain were on the edge, where you can store a configuration on an edge storage service provided by your CDN provider and apply the logic to the edge compute service for routing your system correctly. Many organizations went down this safe route, creating a lot of confidence thanks to the possibility to roll back quickly without the need for a new redeployment. You can see a [basic implementation in this video](#).

## How do you decide which modules or features to migrate first?

Prioritizing the right areas can make your migration smoother, deliver value faster, and build confidence with both your team and the business.

## Best Practices and Patterns

- **Start with new features or major refactors:** If your product team is planning new functionality or a significant redesign, that's often the best place to introduce micro-frontends. You're already investing effort, so you can modernize without duplicating work. This also avoids the challenge of justifying the migration of a module that will "look the same" after the work is done.
- **Pick less risky, self-contained modules:** For your first migration, consider starting with a module that's relatively isolated and low risk-something that won't break the whole application if issues arise. This allows you to test out the migration process and integration patterns before tackling more complex areas. Moreover, all the learnings during the first development will be helpful for the next ones.

- **Go end-to-end with your first micro-frontend:** Choose a module you can take through the entire lifecycle: development, testing, observability, and deployment. This approach will enable you to learn a ton about how micro-frontends work in your environment. Don't be afraid to challenge your initial findings and adapt as you discover more, embrace a lean, learning-focused mindset.
- **Proof of concept and pilot:** develop a low-level design and a proof of concept for your first micro-frontend. Use this as a “point of no return” milestone: if it works and integrates well, you can confidently proceed; if not, you can adjust your approach before scaling up.

## Trade-offs and Pitfalls

- **Migrating high-traffic modules first can be risky:** While you'll see business impact quickly, any issues will be highly visible. Make sure you have strong monitoring and rollback strategies in place.
- **Migrating low-traffic or less critical modules first is safer but slower to show value:** This approach is great for learning and de-risking, but may not excite stakeholders or justify the migration investment early on.
- **Mixing too many approaches can create confusion:** For example, mixing micro-frontends and monolith code on the same page can add complexity and slow down progress. Aim for clear, page-level or route-level boundaries when possible.
- **Technical debt and duplicated effort:** If you migrate modules that are about to be retired or redesigned, you risk wasting effort. Always align migration priorities with the product roadmap.

## Checklist

- Are there any new features or major refactorings planned?
- Which modules are most critical for business value or user experience?
- Which modules are self-contained and low risk for a first migration?
- Do you have a clear design and proof of concept for your first micro-frontend?
- Have you aligned your migration plan with the product and business roadmap?
- Is your team ready to monitor, measure, and roll back if needed?

## A Quick Example

Suppose your product team is about to launch a new user dashboard. Instead of building it in the monolith, you develop it as a micro-frontend. This lets you test your integration approach, validate performance, and show business value early. Once that's live, you can tackle the next module—maybe the account settings or checkout flow—using the lessons you've learned.

## Personal Experience

While I was consulting a financial unicorn startup based in Singapore, we chose the main dashboard as a good starting point. It was a high-visibility area with some interesting challenges to solve, but also had clear boundaries, making it a good candidate for migration. Going end-to-end with this module taught them a huge amount about development, deployment, and monitoring in the new architecture. The early success there made it much easier to get buy-in for the rest of the journey. In only a month, they deployed the first version in production with good success.

Deciding what to migrate first is about balancing risk, value, and learning. Start where you can deliver impact and gain confidence, and use those wins to fuel the rest of your migration. Embrace a lean mindset: treat every migration as a learning opportunity, and don't be afraid to adapt as you go.

## How do you maintain user experience and consistency during migration?

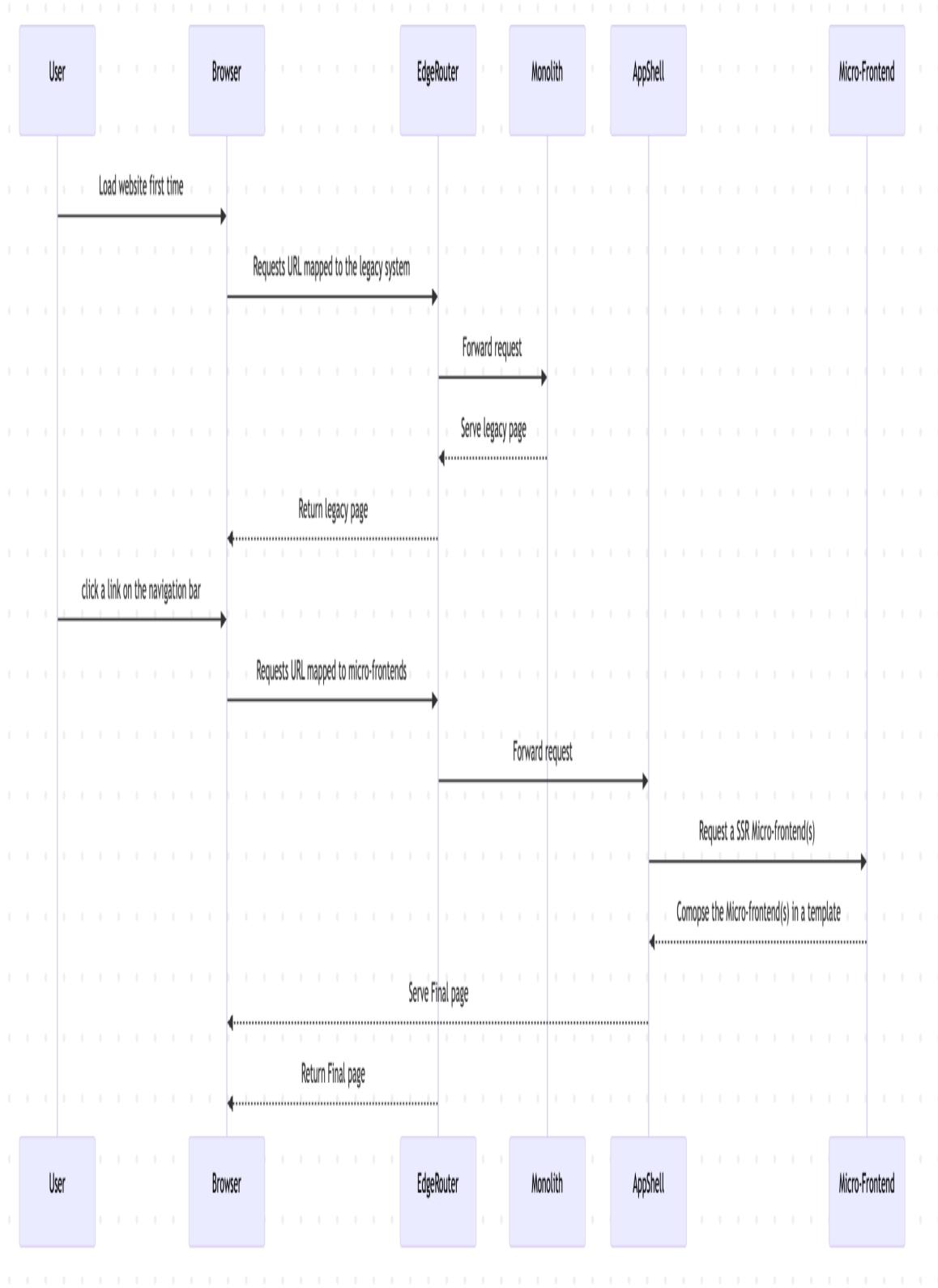
Before we get into the details, let's be honest: some discrepancies in look and feel are likely to happen as you migrate to micro-frontends, especially if different teams are moving at different speeds or using different technologies. But here's the good news: this isn't necessarily a bad thing if you're delivering real value to your users, such as new features or a better experience. Most customers will be forgiving during the transition.

### Best Practices and Patterns

- **Invest in a design system:** A robust design system is an essential tool for maintaining consistency across micro-frontends. It provides shared styles, components, and guidelines so that every team can build features that look and feel like they belong together, even if they're using different frameworks or tech stacks.
  - If possible, use web components for your design system. Web components are framework-agnostic and encapsulated, so they can be reused across all your micro-frontends, no matter what technology each team picks.
  - If you don't have a design system yet, start with design tokens (colors, spacing, typography) and build up a new system using web components as you go.
- **Avoid reusing monolith component libraries:** It may be tempting to pull components from your old monolith, but this can create unwanted dependencies and slow you down. Instead, focus on

building only the components you need for the first micro-frontends, and expand your library as new modules are built.

- **Build only what you need, when you need it:** Don't feel pressured to recreate every single component up front. Start with the essentials for your first micro-frontend, and let your design system grow organically as you migrate more features.
- **Easy routing between the old and new world:** If you have centralized the routing on the edge as explained earlier, the way you will handle the hard navigation between your previous system and the new one is via absolute URL. A simple change will trigger the router on the edge and will load the next system without creating too much complexity ([Figure 9-2](#)).



*Figure 9-2. The sequence diagrams that show the hard navigation between micro-frontends and the legacy system leveraging absolute URLs*

## Trade-offs and Pitfalls

- **Some inconsistency is inevitable:** Early in the migration, you may see minor visual or behavioral differences. As long as you’re delivering value, most users will accept these temporary quirks.
- **Over-engineering the design system:** Trying to build a “perfect” design system before you start can slow you down. Focus on the basics, and let your system evolve as you learn what works.
- **Tight coupling to legacy code:** Reusing monolith components can create dependencies that are hard to untangle later. Build new, modular components when possible.
- **Lack of communication between teams:** Without regular check-ins and shared guidelines, teams may drift apart in their implementation. Keep everyone aligned with clear documentation and open channels.

## Checklist

- Do you have a design system (or at least design tokens) that all teams can use?
- Are your shared components framework-agnostic (e.g., web components)?
- Are you building only the components needed for the current migration phase?
- Is navigation seamless between the old and new world?
- Are you running regular design reviews and automated UI tests?
- Are teams communicating and sharing feedback regularly?

## A Quick Example

Imagine you're migrating the account dashboard to a micro-frontend. Instead of reusing the old monolith's component library, you build a few core web components-like buttons and form fields-based on your new design tokens. The result: the new dashboard looks and feels consistent with the rest of the app, even though it's built with different technology. As you migrate more features, you add new components to your shared library, keeping the experience cohesive for users.

## Personal Experience

In my experience across many teams, I've found that starting with a small, focused set of web components and a clear design system made a huge difference. We built only what we needed for the first micro-frontend, then expanded as we went. This approach lets us deliver value quickly, avoid unnecessary dependencies, and keep the user experience consistent, even as we move fast and learn along the way.

Consistency is important, but don't let the pursuit of perfection slow you down. Focus on delivering value, use a design system and web components to align teams, and let your user experience improve as your migration progresses

## How do you handle shared dependencies and version management between legacy and micro-frontend code?

When migrating to micro-frontends, one of the most common concerns is how to manage shared dependencies and versions between your legacy monolith and the new micro-frontends. It's easy to fall into the trap of trying to avoid all duplication at any cost, but in distributed systems, the priorities are different: we optimize for speed and autonomy, not just for reusability.

## Best Practices and Patterns

- **Don't fear duplication:** In distributed systems, some duplication is not only acceptable, but often desirable. It allows teams to move faster, make independent decisions, and avoid unnecessary coupling. Use the migration as an opportunity to rebuild or rethink parts of your system that have become bloated or outdated.
- **Balance reuse and independence:** If you have a well-designed, framework-agnostic design system or utility library that genuinely fits both worlds, feel free to reuse it. But don't obsess over sharing everything. The goal is to enable fast flow and independent deployment, not to create a tangled web of shared dependencies.
- **Create clear rules to follow:** Define straightforward guidelines for every engineer who encounters roadblocks during the migration. Don't stop or overthink every obstacle, there will be many, and moving fast is important. Remember, many of the decisions you make can be changed easily later. Don't worry if a choice leads to some friction; you'll learn and adapt as you go.
- **Decouple version management:** Set clear rules so that versioning for micro-frontends is completely independent from the legacy monolith. This avoids “dependency hell” where changes in one world break the other. For example, if you need to update a library, do it in the micro-frontend first and only touch the monolith if absolutely necessary.
- **Domain-driven splits help:** If you've identified your domains well and are splitting your app by first-level URL (e.g., /catalogue, /checkout), it's much easier to keep dependencies and versions isolated. Each domain can evolve at its own pace.

## Trade-offs and Pitfalls

- **Too much sharing creates coupling:** Over-sharing libraries or components between legacy and micro-frontends can slow everyone down and make upgrades risky. It also makes it harder to refactor or modernize your new codebase.
- **Unmanaged duplication can lead to bloat:** If you duplicate everything without discipline, you may end up with larger bundle sizes or inconsistent user experiences. Be intentional about what you duplicate and why.
- **Premature abstraction:** Don't rush to abstract or share code before you have enough duplication to justify it. Let patterns emerge naturally, then extract shared libraries when there's a proven need.

## Checklist

- Are you comfortable with some duplication if it improves team autonomy and speed?
- Did you define some common rules to follow to avoid slowing down the migration?
- Are versioning and release processes for micro-frontends completely independent from the monolith?
- Have you split your domains clearly, preferably by first-level URL?
- Do you regularly review duplicated code to identify when (and if) it should be abstracted?

## A Quick Example

Suppose your legacy app and your new micro-frontend both need a date picker. Instead of forcing both to use the same shared component, you build

a new, lightweight date picker for the micro-frontend that fits your new design system. If, over time, you find that several micro-frontends need the same version, you can extract it into a shared package for the new world only, leaving the monolith untouched.

## Personal Experience

When we started the migration in 2016, the only thing we reused between the old system and the new one was the design tokens. We didn't have a design system, so we took this opportunity to revisit our framework, library choices and plan on a design system that would serve well the micro-frontends architecture. We also involved a large portion of the developers in the initial phase to collectively determine how to approach specific problems that were affecting multiple teams.

It's important to create a collaborative environment where people feel included and empowered. This kind of culture encourages everyone to push the boundaries and continuously improve the system-whether you're in the room or not.

## How do you manage cross-cutting concerns like auth, routing, or state during migration?

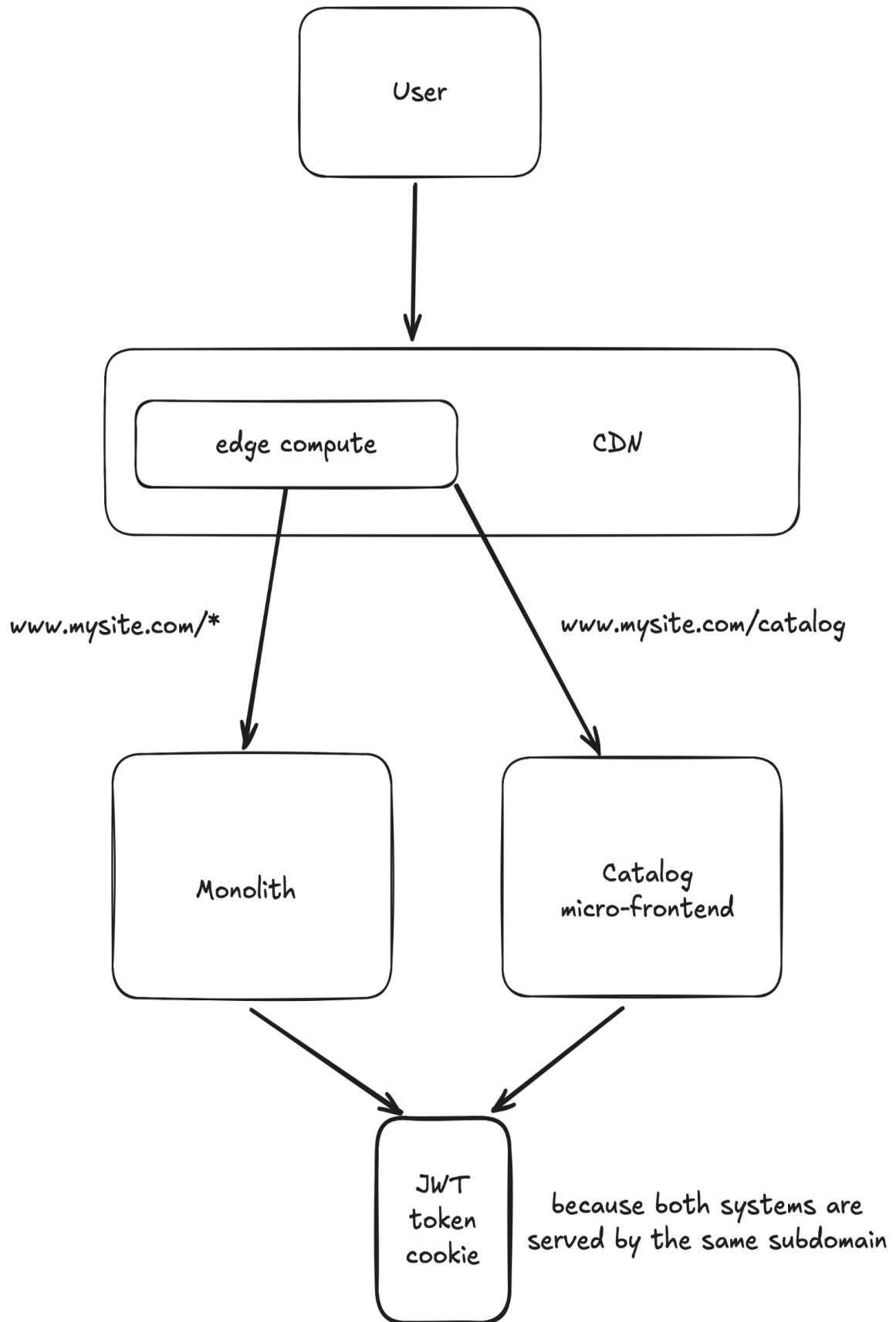
Managing cross-cutting concerns-such as authentication, routing, and state-is one of the trickiest parts of migrating to micro-frontends. These concerns span multiple modules and can easily become sources of bugs or user frustration if not handled thoughtfully. The good news is that with the right approach, you can keep things simple and robust while maintaining a smooth user experience.

## Best Practices and Patterns

- **Authentication:** As long as your micro-frontends are served from the same subdomain (www. or similar) as your legacy application,

they'll have access to the same cookies and session/local storage. This makes it straightforward for each micro-frontend to access session information. Just remember: you'll need to implement refresh token logic in both the legacy system and the new micro-frontends to keep users logged in seamlessly ([Figure 9-3](#)).

- **Routing:** Where possible, centralize routing at the edge (CDN or edge compute) and use absolute URLs for navigation between the legacy app and micro-frontends. This approach drastically simplifies navigation logic, avoids coupling between systems, and provides a fast, reliable rollback strategy if something goes wrong.
- **State management:** If you need to share state between micro-frontends and the legacy app (or vice versa), use query strings for ephemeral data that only needs to persist for a single navigation event. For more persistent or configuration data, rely on backend APIs to pass information between systems. Avoid using local or session storage for critical data, as this can lead to awkward edge cases (like browser refreshes) and complicate your migration in the long run.



*Figure 9-3. Cookies, session and local storage are accessible by the legacy system and micro-frontends as long served by the same subdomain in case of local and session storage or with the right configuration in case of the cookies*

## Trade-offs and Pitfalls

- **Auth drift:** If your authentication logic diverges between the legacy system and micro-frontends, you risk inconsistent user experiences or security issues. Keep the core logic aligned and test thoroughly across both systems. Handle the identity provider change later on, after the migration is done or create a proxy on the backend that takes care of routing the request to the right identity provider.
- **Overcomplicating routing:** Trying to handle routing logic inside each app can lead to confusion and bugs. Centralizing at the edge keeps things simple and makes rollbacks much easier.
- **State synchronization headaches:** Relying on browser storage for key data can create hard-to-debug issues, especially with refreshes or multiple tabs. Stick to query parameters for transient data and APIs for anything more substantial.

## Checklist

- Are all micro-frontends and the legacy app served from the same subdomain for seamless cookie and storage access?
- Is refresh token logic implemented in both systems?
- Are you using edge routing and absolute URLs for navigation between systems?
- Do you use query strings for passing ephemeral state between apps?
- Are APIs in place for sharing configuration or persistent data?

- Have you avoided relying on local/session storage for critical state?
- Are you regularly testing authentication, navigation, and state flows across both systems?

## A Quick Example

Suppose you're migrating the user profile section to a micro-frontend. The user logs in through the legacy app, and the session cookie is available to the new micro-frontend because both are on the same subdomain. When navigating from the legacy dashboard to the new profile page, you use an absolute URL, which triggers edge routing and loads the correct system. If you need to pass a temporary filter or view state, add it to the query string. For persistent preferences or user data, both systems read from a shared backend API.

## Personal Experience

In my own migrations, keeping authentication and session management consistent across both worlds was crucial. At DAZN, we made sure all micro-frontends were served from the same subdomain, so cookies and session data were always accessible. We also centralized routing at the edge, which made navigation seamless and rollbacks trivial. This approach kept the migration smooth and minimized user disruption—even as we moved quickly.

Cross-cutting concerns can make or break your migration. Keep things simple: centralize where you can, avoid unnecessary coupling, and let real needs drive your abstractions. This will help you maintain a consistent, reliable experience for your users throughout the transition.

## Do I need microservices to migrate to micro-

# frontends?

This is a very common question, and the answer is simple: no, you do not need microservices to migrate to micro-frontends. Any frontend—including micro-frontends—just needs a clear contract to interact with APIs. Don't overcomplicate things by thinking you need to modernize your backend first or in parallel.

## Best Practices and Patterns

- **Frontend and backend are independent:** Micro-frontends are about how you structure and deliver your frontend code. As long as your frontend can call APIs (REST, GraphQL, etc.), it doesn't matter if those APIs are powered by a monolith or microservices. Remember that every software is a living system, hence there will be changes, optimizations and throwaway code every single sprint. Just accept the nature of software and you will be fine
- **Start from the frontend:** You can begin your modernization journey by breaking up your frontend into micro-frontends and connecting them to your existing backend. There's no technical requirement to migrate your backend to microservices first. This approach is usually faster due to the stateless nature of frontends.
- **Data has gravity:** In my experience, the frontend is stateless and much quicker to modernize. The backend—especially when it involves data replication or redesigning database schemas—takes much longer. Focus on delivering value quickly through frontend improvements while planning for backend changes at your own pace.
- **Iterative backend modernization:** Once your frontend is modular and delivering value, you can gradually modernize your backend if and when it makes sense for your business.

## Trade-offs and Pitfalls

- **Don't block on backend modernization:** Waiting for a full backend migration before starting on the frontend can delay value for users and the business. Frontend and backend can evolve independently.
- **API contracts matter:** As long as your APIs are well-defined, your micro-frontends can work with any backend architecture. Just ensure the contracts are stable and documented.
- **Backend migration is harder and slower:** Migrating backend systems, especially with complex data, is often much more time-consuming and risky than frontend work. Don't underestimate this difference.
- **Partial modernization still delivers value:** Even if some backend APIs remain in the monolith, you can still achieve significant improvements in user experience, team autonomy, and release speed through micro-frontends.

## Checklist

- Can your frontend teams deliver value independently of backend modernization?
- Are you able to iterate on the frontend without waiting for backend changes?
- Have you planned for backend modernization as a separate, future project if needed?

## A Quick Example

Let's say you have a monolithic backend serving all your APIs. You start migrating your frontend to micro-frontends, each one calling the same set of APIs as before. Users see new features and improved performance

quickly, while the backend remains untouched for now. Later, you can refactor backend services one by one, but you're already delivering value.

## Personal Experience

In dozens of modernization projects, I've seen that micro-frontends can be delivered in as little as 14 months, while backend migrations often take much longer due to data complexity and schema changes. The frontend is stateless and much easier to move quickly. Even when backend APIs are still running on the old monolith, the value delivered by a modern, modular frontend is huge. Don't let backend modernization block your progress, start where you can make the biggest impact fastest.

You do not need microservices to migrate to micro-frontends. Focus on decoupling and improving your frontend first; the backend can follow when you're ready.

## Summary

Migrating to micro-frontends is a meaningful journey, challenging at times, but far from impossible. The most important lesson? Start small, move iteratively, and always tie your work to real business value. You don't need to wait for a perfect backend or a flawless design system to begin.

Remember:

- **Iterative beats big bang:** Break your migration into manageable pieces. Use edge routing and patterns like the strangler fig to keep changes safe and reversible.
- **Prioritize for impact and learning:** Start with modules that let you deliver value and learn quickly. Don't be afraid of a little duplication or some temporary inconsistency.
- **Keep things simple:** Centralize cross-cutting concerns, avoid over-engineering, and let your design system and shared code evolve as you go.

- **Microservices aren't required:** Micro-frontends can thrive with any backend, as long as you have clear API contracts.

With clear goals, teamwork, and a willingness to adapt, you'll unlock new speed and flexibility for your teams and your business.

And if you're curious about how all these ideas come together in practice, stay tuned: towards the end of the book, I'll share a real migration story, including the reasoning and steps that took us all the way to production with a micro-frontends system.

# Chapter 10. From Monolith to Micro-Frontends: A Case Study

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 12th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the author at [building.microfrontends@gmail.com](mailto:building.microfrontends@gmail.com).

Let’s say that in the last few weeks, you’ve researched and reviewed articles, books, and case studies and completed several proofs of concept. You’ve spoken with your managers to find the best people for the project, and you’ve even prepared a presentation for the CTO explaining the benefits you can get from introducing micro-frontends in your platform. At last, you’ve received confirmation that you have been granted the resources to prepare a plan and start migrating your legacy platform to micro-frontends. Great job! It’s been a long few weeks, and you’ve done an amazing job, but this is only the start of a large project.

Next, you will need to prepare an overall strategy, one that’s not too detailed but not too loose. Too detailed, and you’ll spend months just trying to nail everything down. Too loose, and you won’t have enough guidance. You need enough of a strategy to get started and a North Star to follow during the journey whenever you discover—and, trust me, you will—new

challenges and details you didn't think about until that point. Meanwhile, you also have a platform to maintain in production, which the product team would like to evolve because the replatforming to micro-frontends shouldn't block the business. The situation is not the simplest ever, but you can mitigate these challenges and find the right trade-off to make everyone happy and the business successful while the tech teams are migrating to the new architecture.

We have learned a lot about how to design and implement micro-frontends, but I feel this book would not be complete without looking at migration from a monolithic application to a micro-frontend one—by far, the most common use case of this architecture. I believe any project should start simple. Then, over the course of months or years, when the business and the organization are growing, the architecture should evolve accordingly to support the evolving needs of the business. There may be some scenarios where starting a new application with micro-frontends may help the business move in the right direction, such as when you have an application that is composed of several modules that you can ship all together along with some customization for every customer. But the classic use case of micro-frontends is the migration from a legacy frontend application to this new approach. In this chapter, I will share a case study example that stitches together all the information we have discussed in this book.

## The Context

ACME Inc. is a fairly new organization that, in only a few years, has gained popularity for its video-streaming service across several countries around the world. The company is growing fast. In the last couple of years, it has moved from hundreds of employees to thousands, located all across the globe, and the tech department is no exception.

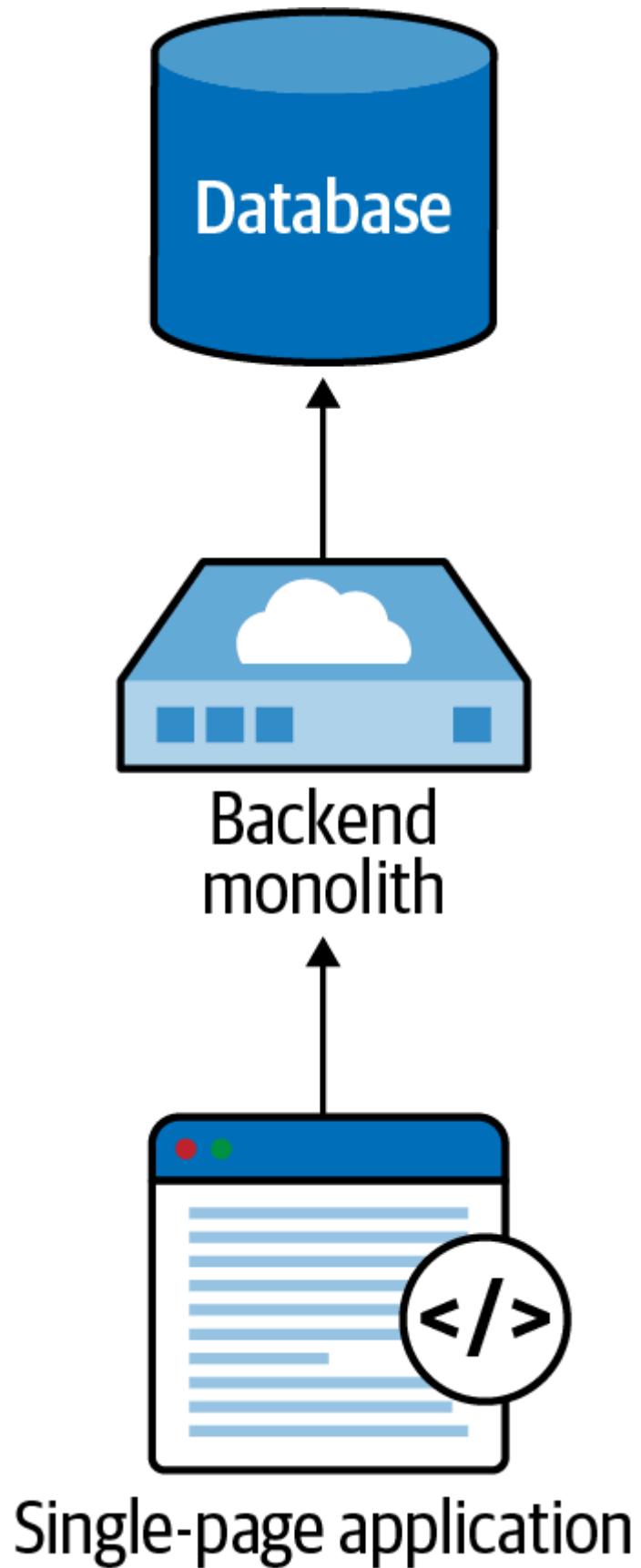
The streaming platform is currently available on desktop and mobile browsers, and on some living-room devices such as smart TVs and consoles. Currently the company is onboarding many developers in different locations across Europe. Having all the developers in Europe was

a strategic decision to avoid slowing down the development across distributed teams while having some hours of overlap for meetings and coordination.

Due to the tech department's incredible growth from tens to hundreds of people, tech leadership reviewed and analyzed the work done so far, finally embracing a plan to adapt their architecture to the new phase of the business. Leadership acknowledged that maintaining the current architecture would slow down the entire department and wouldn't allow the agility required for the current expansion the business is going through.

## **Technology Stack**

The current platform uses a three-tier application deployed in the cloud, composed of a single database with read replicas (they have more reads than writes in their platform), a monolithic API layer with auto-scaling for the backend that scales horizontally when traffic increases, and a single-page application (SPA) for the frontend, as shown in [Figure 10-1](#).



*Figure 10-1. The ACME platform is a three-tier web application*

A three-tier application allows the layers to be independently scaled and developed. However, ACME is scaling and increasing in complexity, as well as increasing the teams working on the same project. This architecture is now impacting the day-to-day throughput and generating communications overhead across teams that may lead to more complexity and coordination despite not being necessary in other solutions.

As the tech leadership team rightly points out, in this new phase of the business, the tech department needs to scale with more developers and with more features than before. A task force with different skill sets reviews how the architectures—frontend and backend—should evolve in order to unblock the teams and allow the company to scale in relation to business needs.

After several weeks, the task force proposed migrating the backend layer to microservices and the frontend to micro-frontends. This decision was based on the capabilities and principles of these architectures. They will allow teams to be independent, moving at their own speed, scaling the organization as requested by the business, drifting towards the direction the business needs, choosing the right solution for each domain, and scaling the platform according to the traffic on a service-by-service basis by leveraging the power of cloud vendors.

From here, we'll focus our discussion on the frontend part. There will be some references on how the frontend layer is decoupled from the backend using the service dictionary approach discussed in Chapter 11.

## Platform and Main User Flows

The frontend is composed of the following views:

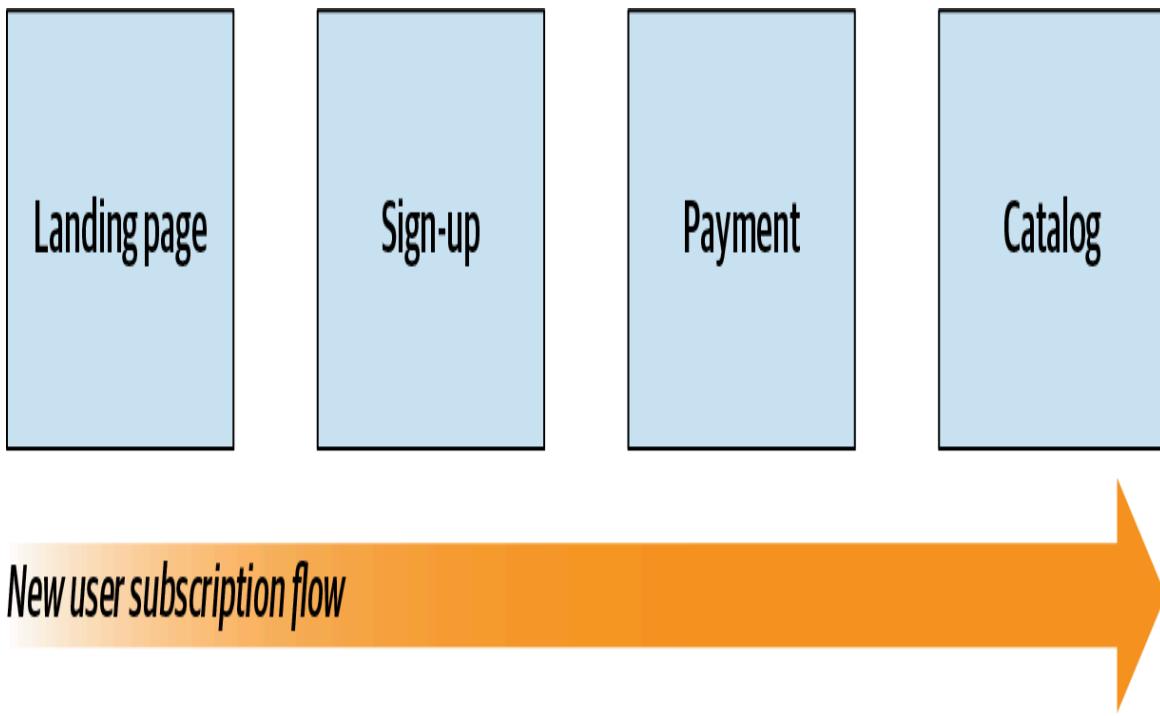
- Landing page
- Sign-in
- Sign-up

- Payment
- Forgot email
- Forgot password
- Redeem gift code
- Catalog (with video player)
- Schedule
- Search
- Help
- My account

To provide us enough information to understand how the migration to micro-frontends will work, we will analyze the authentication flow for existing customers, the creation of a subscription flow for new customers, and the experience inside the platform for authenticated customers. Many of these suggestions can be replicated for other areas of the application or applied with small tweaks.

When a new user wants to subscribe to the video-streaming platform, they follow these steps (see also [Figure 10-2](#)):

1. The user arrives on the landing page, which explains the value proposition.
2. The user then moves to the sign-up page, where they create an account.
3. On the next page, the user adds their payment information.
4. The user can then access the video platform.



*Figure 10-2. New user subscription flow*

When an existing user wants to sign in on a new platform (browser or mobile device, for instance) to watch some content, they will do the following (see also [Figure 10-3](#)):

1. Access the platform in the landing page view
2. Select the sign-in button, which redirects them to the sign-in view
3. Insert their credentials
4. Access the authenticated area and explore the catalog

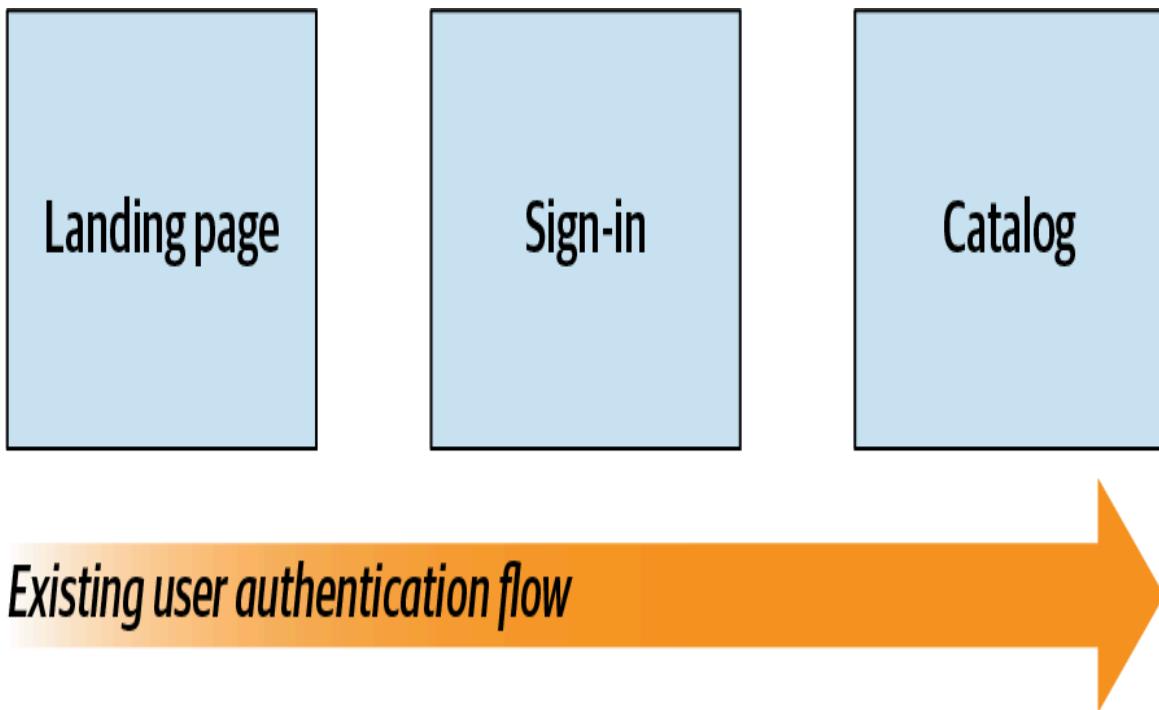
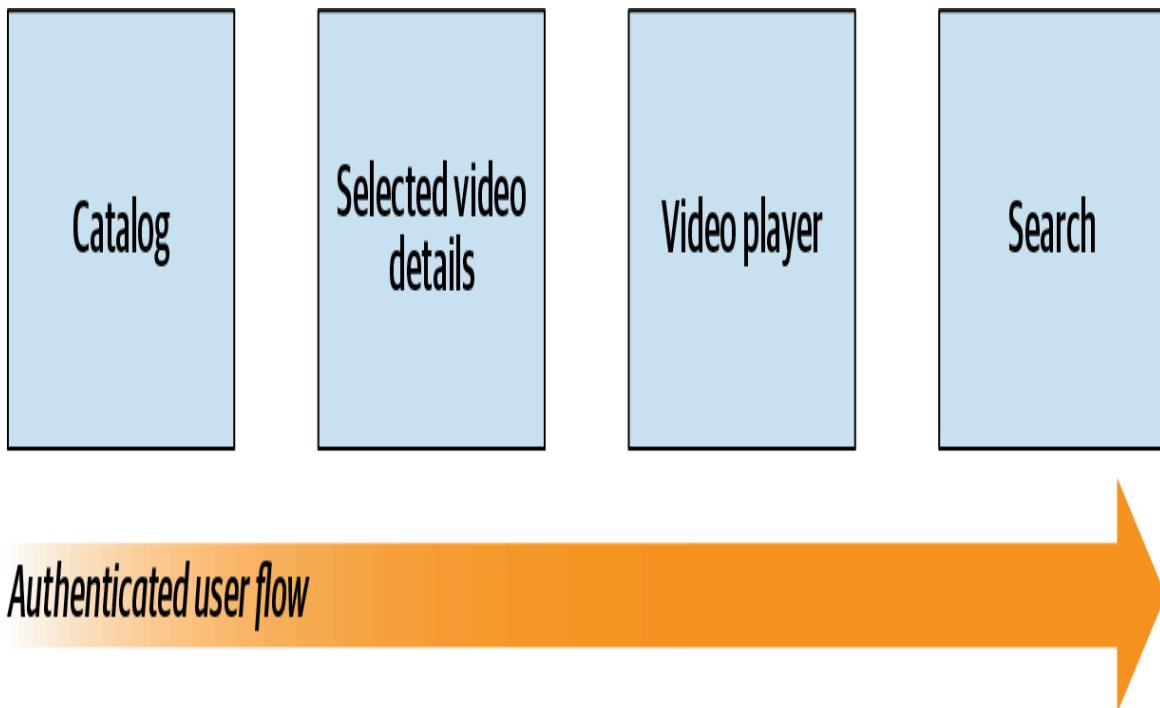


Figure 10-3. Existing user authenticating on a new platform (browser or mobile device, for instance)

Once a user is authenticated, they can watch video content and explore the catalog following these steps (see also [Figure 10-4](#)):

1. They start at the catalog to choose the content to view.
2. When content is selected, the user sees more details related to the content and the possibility to search for similar content or just play the content.
3. When the user chooses to play the content, they are redirected to a view with only the video player.
4. When the user wants to search for specific content not available in the catalog view, they can choose to use the search functionality.



*Figure 10-4. Existing users can navigate content via the catalog or search functionality and then play any content after discovering the details of what they are about to watch*

These are the main flows, which should be enough to explore how to migrate to micro-frontends. Obviously, there are always more edge cases to cover, especially when we implement errors management, but we won't cover those in this chapter.

The application is written with Angular, with a continuous integration pipeline and a deployment that happens twice a month because it is strictly coupled with the backend layer. In fact, the static files are served by the application servers where the APIs live, so therefore, every time there is a new frontend version, the teams have to wait for the release of a new application server version. The release doesn't happen very often due to the organization's slow release cycle process.

The final artifact produced by the automation pipeline is a series of JavaScript and CSS files with an HTML entry point. In the continuous integration process, the application has some unit testing, but the code coverage is fairly low (roughly 30%), and the automation process takes about 15 minutes to execute end to end to create an artifact ready to be deployed in production.

The organization is using a three-environment strategy: testing, staging, and production. As a result, the final manual testing happens in the staging environment before being pushed into production, another reason why deployments can't happen too often. The user acceptance testing department (UAT) does not have enough resources, compared to the developers who handle platform enhancement. Due to the simple automation process put in place, some developers on different teams are responsible for maintaining the automation pipelines; however, it's more of an additional task to shoehorn into their busy schedules than an official role assigned to them. This sometimes causes problems because resolving issues or adding new functionalities in the continuous integration process may require weeks instead of days or hours.

Finally, the platform was developed with observability in mind, not only on the backend but also on the frontend. In fact, both the product team and the developers have access to different metrics to understand how users interact with the platform so they can make better decisions for enhancing the platform's capabilities. They are also using an observability tool for tracking JavaScript runtime errors inside their frontend stack.

## Technical Goals

After deciding to move their frontend platform to micro-frontends, the tech leadership identified the goals they should aim for with this investment.

The first goal is maintaining a seamless experience for developers despite the architectural changes. Degrading a frictionless developer experience, available with the SPA, could lead to a slower feedback loop and decrease the software quality. Moreover, the leadership decided that it doesn't want to reinvent the wheel either, so it will be acceptable to create some tools for filling certain gaps but not a complete custom developer experience that may prevent new tools from being embraced in the future. It's important to fix the automation strategy for reducing the feedback loop that now takes too long.

Another key project goal is to decouple the micro-frontends and allow independent evolution and deployment. Micro-frontends that are tightly coupled together must be released all together. Every micro-frontend should be an independent artifact deployable in any environment. It needs to optimize for fast flow, reducing the external dependencies for each team.

Moreover, tech leadership wants to reduce the risk of introducing bugs or defects in production, easing the traffic toward new micro-frontend versions. This way, developers can test with real data in production but not affect the entire user base.

An additional goal is to generate value as soon as possible to demonstrate to the business the return of value of their investment. Therefore, a strategy for transitioning the SPA to micro-frontends has to be defined in a way that when a micro-frontend is available, it will initially work alongside the monolith.

The tech leadership has also requested tracking the onboarding time for new joiners in order to understand whether this approach extends developer onboarding time. The team will need to figure out a way to reduce this period, perhaps by creating more documentation or using different approaches.

The last goal for this project is finding the right organization setup for reducing external dependencies between teams and reducing the communication overhead that could increase due to the company's massive growth.

## Migration Strategy

Based on tech leadership's requirements and goals, the ACME teams started to work on a plan for migrating the entire platform to micro-frontends. The first step is embracing the micro-frontends decisions framework as a guideline to define the foundation of the new architecture. The first four decisions—defining what a micro-frontend is in your architecture, composing micro-frontends, routing micro-frontends, and communicating

between micro-frontends—will lead the entire migration toward the right architecture for the context.

As discussed in several chapters of this book, the micro-frontend decisions framework gives us a skeleton to architect a micro-frontend project onto. All the other decisions will build on top of this frame, creating a reliable structure.

## **Micro-Frontend Decisions Framework Applied**

The first decision of the framework is how a micro-frontend will look. The ACME teams decided on a *vertical split*, where micro-frontends represent a subdomain of the entire application (see [Figure 10-5](#)).

Header

Content

Footer

Micro-frontend

Application shell

*Figure 10-5. A vertical-split micro-frontend, where the application shell loads only one micro-frontend at a time*

A vertical-split is the approach chosen by several other organizations worldwide for kicking off their new micro-frontends architecture. It has fewer sharp edges to consider and can always be combined with a horizontal split when needed.

The teams took into account the following characteristics for their context before deciding to use a vertical split:

#### *Similar developer experience*

Because the current platform is an SPA, a vertical split allows developers to work like they have so far but with a smaller context and less code to be responsible for.

#### *Low component reusability*

The teams have identified that not many components are similar across the different subdomains. This clearly indicates that the reusability of micro-frontends, a plus of a horizontal-split approach, is not needed. A light design system will ensure consistency across micro-frontends and reduce overhead.

#### *Better integration with current automation strategy*

The vertical split fits very well with the current automation strategy, considering right now ACME is building an SPA. The teams have enhanced their automation pipelines for building multiple SPAs without the need to create custom tools for embracing this architecture style. They will need to use infrastructure as code for automating the process of building their pipelines and replicating them without human intervention.

#### *No risk of dependency clashes*

In a vertical split, the application shell always loads one micro-frontend at a time, due to its nature. As a result, the teams won't have to deal

with dependency clashes, like different versions of the same library, because there will be dependencies of just one micro-frontend, reducing the possibility of runtime errors and bugs in production. There also won't be any CSS style clash because only one stylesheet per micro-frontend will load.

### *A consistent user experience*

Creating a consistent user experience is easier with a vertical split because the same team is working on one or multiple views inside the same SPA. Obviously, a level of coordination is required for maintaining consistency across micro-frontends, but it's definitely less prone to errors than having multiple micro-frontends in the same view developed by multiple teams.

### *Reduction of cognitive load*

For ACME, a vertical split will decrease its developers' cognitive load, because they'll only have to master and maintain a part of the platform. This choice also won't dilute the decisions made by developers inside their business subdomain. However, every developer should have an overall understanding of the platform architecture so that when they're on call, they can understand the touch points of their business domain and recognize where a bug may appear despite not being inside their domain.

### *Faster onboarding process*

As the tech leadership requested, using this approach will lead to a faster onboarding process because the teams can use well-known, standard tools and won't need to create their own to build, test, and deploy micro-frontends. Also, because teams will be responsible for only a part of the platform, less coordination with other teams will be required. New joiners can hit the ground faster, with less information needed to start. Finally, every team will be encouraged to create a starter kit and induction for every new joiner to speed up the learning process

and make a person capable of contributing to the base code in the fastest way possible.

The second decision of the framework is related to the composition of the micro-frontends. In this case, the best approach is composing them on the client side considering they are using a vertical-split approach. This means that the teams will have to create an *application shell* that is responsible for mounting and unmounting micro-frontends, exposing some APIs to allow communication between micro-frontends and ensuring it will always be available during the user session (see [Figure 10-6](#)). A server-side composition was rejected immediately due to the traffic spikes, which required more effort to support and maintain than the simple infrastructure they would like to use for this project. Moreover, since the majority of the application is behind authentication, it can't benefit from the organic SEO offered by the server-side rendering architecture.

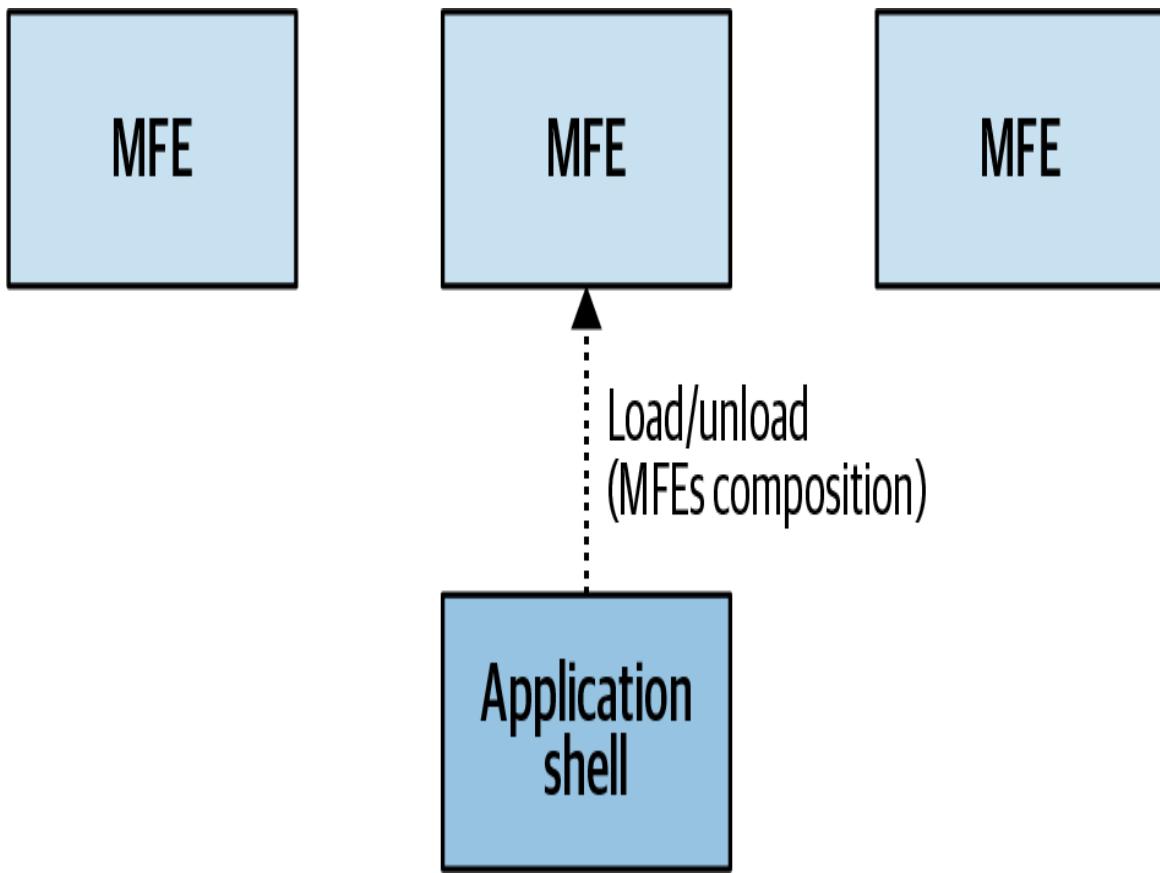
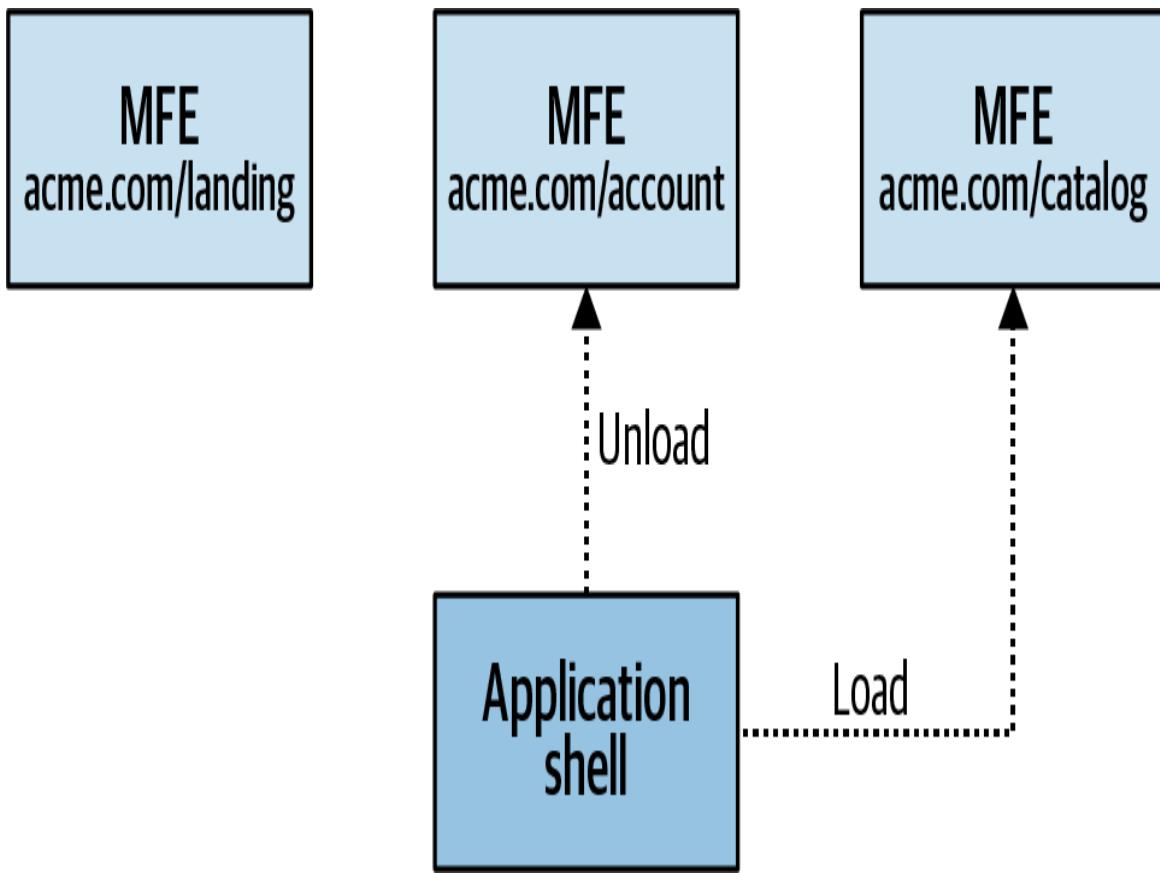


Figure 10-6. Client-side composition, where the application shell is responsible for loading and unloading one micro-frontend at a time

The third decision is the routing of micro-frontends, that is, how to map the different application paths to micro-frontends. Because ACME will use a vertical split and is composed on the client side, the routing must happen on the client side, where the application shell knows which micro-frontend to load based on the path selected by the user. This mechanism also has to handle the deep-linking functionality; if a user shares a movie's URL with someone else, the application shell should load the application in exactly that state (see [Figure 10-7](#)).



*Figure 10-7. When the user signs in from the /account path, they are redirected to the authenticated area (/catalog). The application shell owns the logic for unloading the current micro-frontend and loading the next one based on the URL.*

When an unauthorized user tries to access an authenticated part of the system via deep linking, the application shell should validate *only* if the user has a valid token. If the user doesn't have a valid token, it should load the landing page so the user can decide to sign in or subscribe to the service.

Last but not least, ACME teams have to decide how micro-frontends communicate with each other. With a vertical split, communication can happen only via a query string or using web storage, eliminating the need for other techniques like event emitters or custom events which are required for a horizontal split. ACME decided to mainly leverage the web storage and use the application shell as a proxy for storing the data. In this way, the application shell can verify the space available and make sure data won't be overridden by other micro-frontends (see [Figure 10-8](#)).

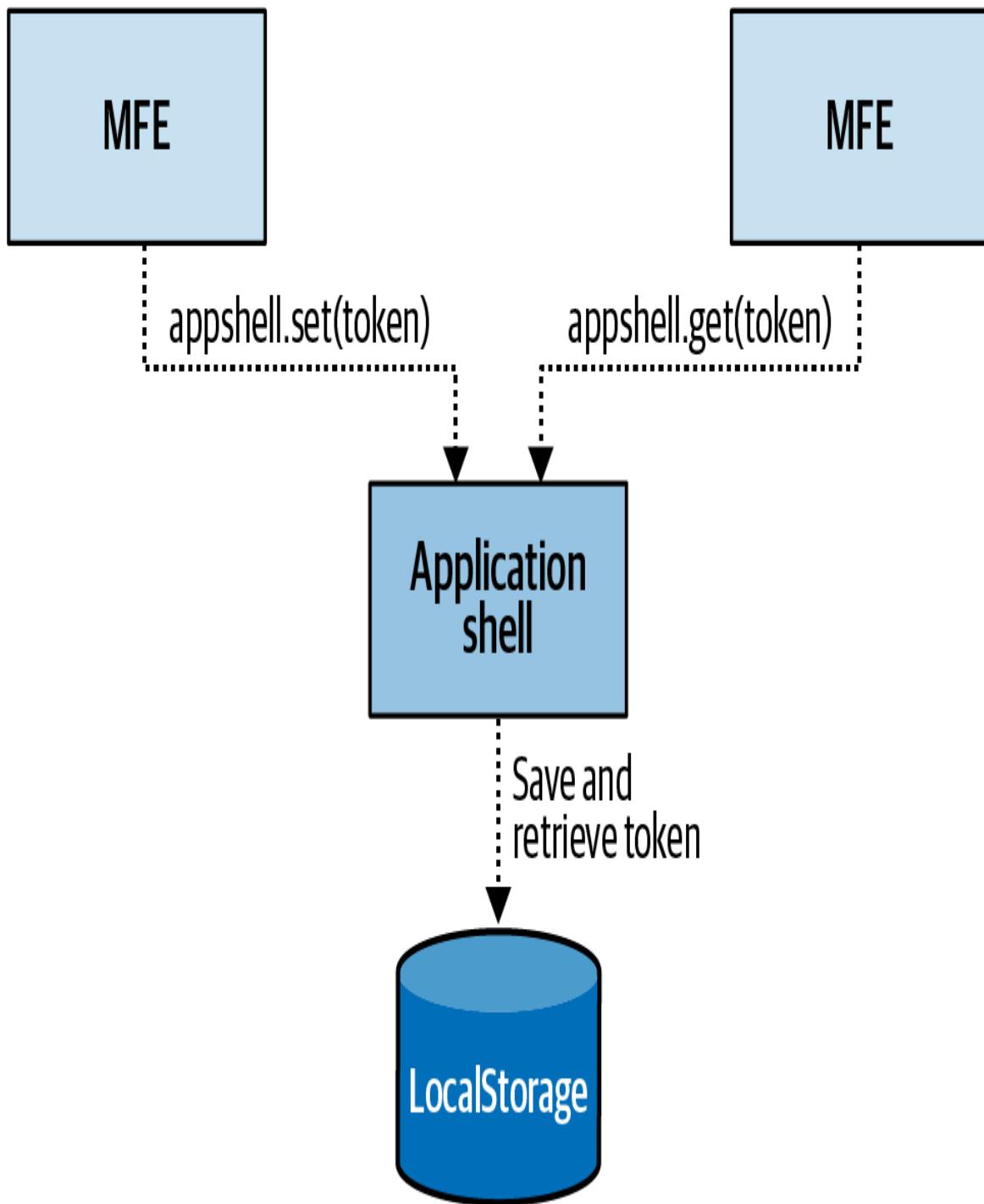


Figure 10-8. The application shell is responsible for storing data in the local storage and exposing several APIs to the micro-frontends for storing and retrieving data

Let's summarize the decisions made by the teams in [Table 10-1](#).



*T*

*a*

*bl*

*e*

*l*

*0-*

*l.*

*S*

*u*

*m*

*m*

*a*

*ry*

*of*

*A*

*C*

*M*

*E*

*a*

*rc*

*hi*

*te*

*ct*

*u*

*r*

*al*

*d*

*e*

*ci*

*si*

*o*

*n*

*s*

---

## Micro-frontend decisions framework

---

Vertical split

### Defining micro-frontends

<b>Composing micro-frontends</b>	Client side via application shell
<b>Routing micro-frontends</b>	Client side via application shell
<b>Communication between micro-frontends</b>	Using web storage via application shell

## Splitting the SPA in Multiple Subdomains

After creating their micro-frontend framework, the ACME tech teams analyzed the current application's user data to understand how the users were interacting with the platform. This is another fundamental step that provides a reality check to the teams. Often what tech and product people envision for platform usage is very different from what users actually do.

The SPA was released with a Google Analytics integration, and the teams were able to gather several custom data points on user behavior for developing or tweaking features inside the platform. These data are

extremely valuable in the context ACME operates because they help identify how to slice the monolith into micro-frontends.

Looking at user behaviors, the teams discover the following:

#### *New users*

Users who are discovering the platform for the first time follow the sign-up journey as expected. However, there are significant drops in visualization from one view to the next. As we can see in [Table 10-2](#), all the new users access the landing page, but only 70% of that traffic moves to the next step, where the account is created. At the third step (payment), there is a drop of an additional 10%. At the last step, only 30% of the initial traffic has converted to customers.



$T$

$a$

$bl$

$e$

$I$

$0\text{-}$

$2.$

$N$

$e$

$w$

$u$

$se$

$r$

$tr$

$af$

$fi$

$c$

$p$

$er$

$vi$

$e$

$w$

$o$

$n$

$A$

*C*

*M*

*E*

*pl*

*at*

*fo*

*r*

*m*

---

## **View**

## **Traffic**

---

Landing page      100%

Sign-up              70%

Payment              60%

Catalog              30%

### *Unauthenticated existing users*

Existing users who want to authenticate on a new browser or another platform, such as a mobile device, usually skip the landing page, going

straight to the sign-in URL. After signing in, they have full access to the video catalog, as seen in [Table 10-3](#).



$T$

$a$

$bl$

$e$

$I$

$0\text{-}$

$\beta.$

$U$

$n$

$a$

$ut$

$h$

$e$

$nt$

$ic$

$at$

$e$

$d$

$e$

$xi$

$st$

$in$

$g$

$u$

$se$

*r*

*tr*

*af*

*fi*

*c*

*p*

*er*

*vi*

*e*

*w*

*fo*

*r*

*a*

*c*

*c*

*es*

*si*

*n*

*g*

*A*

*C*

*M*

*E*

*pl*

*at*

*fo*

*r*

*m*

---

<b>View</b>	<b>Traffic</b>
Landing page (as entry point)	25%
Sign-in (as entry point)	70%

---

### *Authenticated existing users*

Probably the most interesting result is that authenticated users are not signing out. As a result, they won't see the landing page or sign-in/sign-up flows anymore. They occasionally explore their account page or the help page. But a vast majority of the time, authenticated users are staying in the authenticated area and not navigating outside of it (see **Table 10-4**).



*T*

*a*

*bl*

*e*

*I*

*0-*

*4.*

*A*

*ut*

*h*

*e*

*nt*

*ic*

*at*

*e*

*d*

*e*

*xi*

*st*

*in*

*g*

*u*

*se*

*r*

*tr*

*af*

*fi*

*c*

*p*

*er*

*vi*

*e*

*w*

*fo*

*r*

*a*

*c*

*c*

*es*

*si*

*n*

*g*

*A*

*C*

*M*

*E*

*pl*

*at*

*fo*

*r*

*m*

---

<b>View</b>	<b>Traffic</b>
Landing page	0%
Sign-in	1%
Sign-up	0%
Catalog	92%
My account	4%
Help	2%

---

Landing page 0%

Sign-in 1%

Sign-up 0%

Catalog 92%

My account 4%

Help 2%

This is extremely valuable information for identifying micro-frontends. In fact, ACME developers can assert the following:

- The landing page should immediately load for new users, giving them the opportunity to understand the value proposition.

- Landing page, sign-in, and sign-up flows should be decoupled from the catalog since authenticated users only occasionally navigate to other parts of the application.
- “My account” and “Help” don’t receive much traffic.
- There is a considerable drop of new users between landing page and sign-up flows, and we can expect the product team would like to make multiple changes to reduce this drop.

Another important aspect is understanding how the current architecture can be split into multiple subdomains following domain-driven design practices. Taking into consideration the whole platform—not only the client-side part—the teams identified some subdomains and relative bounded context.

For the frontend part, the subdomains that the teams took into consideration for their final decisions are:

### *Value proposition*

A subdomain for sharing all the information needed to make a decision for subscribing to the platform.

### *Onboarding*

A subdomain focused on subscribing new users and granting access to the platform for existing users. In the future, should complexities arise, this may be split into smaller subdomains, such as payment methods, user creation, and user authentication, but for now they will be one subdomain.

### *Catalog*

A core subdomain where ACME gathers the essential part of its business proposition, such as the catalog, video player, and all the controls for allowing users to consume content respecting the rights holders’ agreements.

### *User management*

A subdomain where the user can change account preferences, payment methods, and other personal information.

### *Customer support*

A subdomain for helping new and existing users to solve their problems in any part of the platform.

With this information in mind and the decisions made for approaching this project using the micro-frontend decisions framework, the teams identified the migration path with the following micro-frontends (see also [Figure 10-9](#)):

### *Landing page*

Considering that the landing page is viewed by all new users, the teams want to have a super-fast experience where the page is rendered in a blink of an eye. It needs its own micro-frontend so all the technical best practices for a highly cacheable micro-frontend with a small size to download can be applied.

### *Authentication*

This micro-frontend is composed of all the actions an unauthenticated user should perform before accessing the catalog, such as moving from sign-in to sign-up view, retrieving their credentials, and so on.

### *Catalog*

This is an authenticated area frequently viewed by authenticated users. The teams want to expedite the experience for these users when they return to the platform, so they encapsulate it in a single micro-frontend.

### *My account*

This micro-frontend is a combination of information available in different domains of the backend, allowing users to manage their account preferences. It is available only for authenticated users. Because

of the small traffic and the cross-cutting nature of this domain, ACME decided to encapsulate it in a micro-frontend.

### *Help*

Like the “My account” micro-frontends, Help has low traffic, a different use case from other micro-frontends, and highly cacheable content (because Help pages are not updated very often). Encapsulating this subdomain in a micro-frontend allows ACME to use the right infrastructure for optimizing this part of the platform.

### *Application shell*

This is the micro-frontend orchestrator. Because ACME decided to use a vertical split with a client composition, this element is mandatory to build. The main caveat is trying to keep it light and as decoupled as possible from the rest of the application so that all the other micro-frontends can be independent and evolve without any dependency on the application shell.

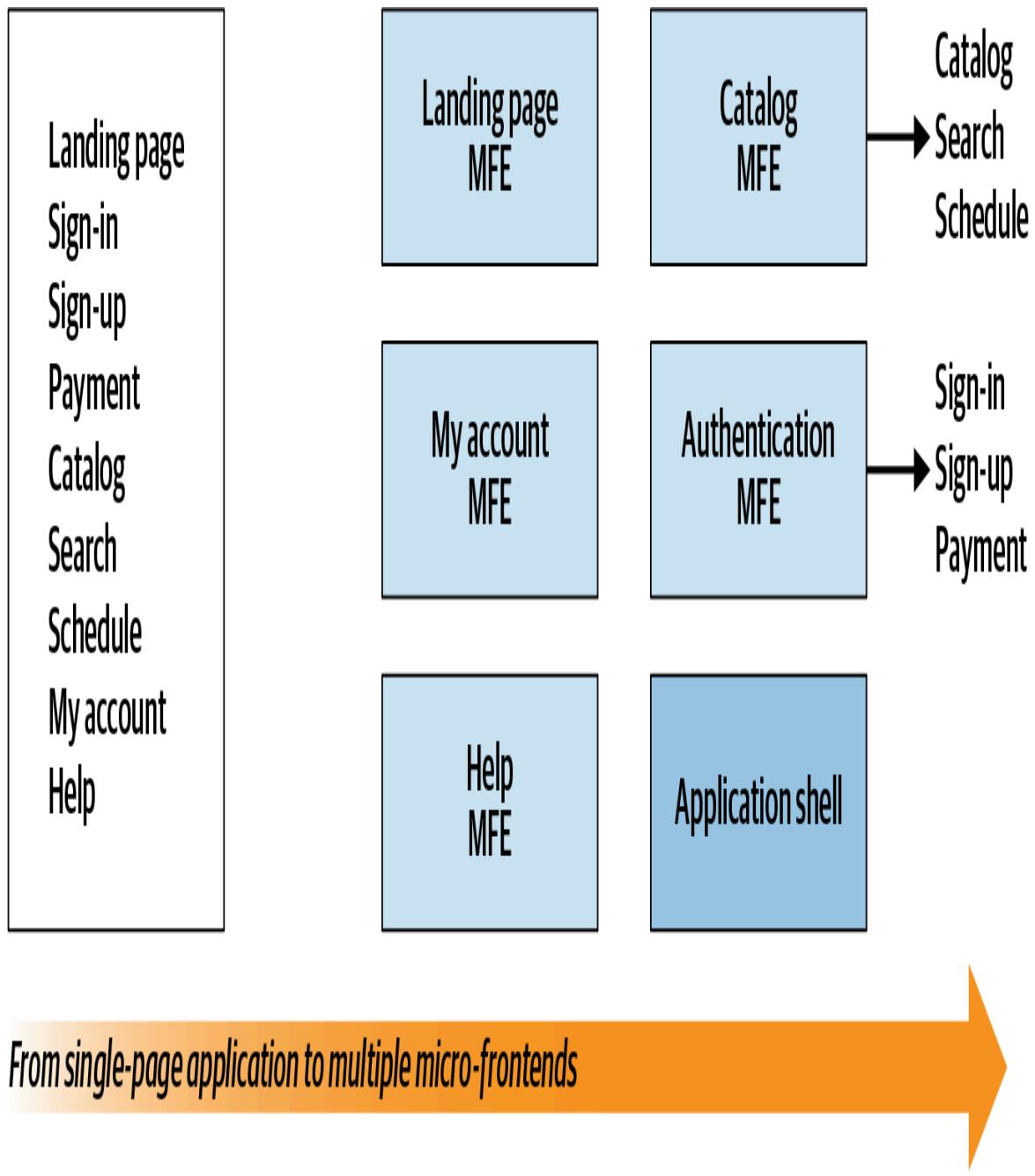


Figure 10-9. Migration path: from SPA to micro-frontends

## Technology Choice

Because the Angular SPA was developed some years ago with patterns and assumptions that were best practices at that time, ACME tech teams investigated their relevance, as well as new practices that might make developers' lives easier and more productive. The teams agreed to use

React, and they have discovered in the reactive programming paradigm a development boost during their proof of concepts.

Although Redux allows them to embrace this paradigm using libraries such as [redux-observable](#), they found in [MobX-State-Tree](#) an opinionated and well-documented reactive state management that works perfectly with React and allows state composition so they can reuse states across multiple views of the same micro-frontend. This will enhance the reusability of their code inside the same bounded context.

Thanks to the nature of the vertical-split micro-frontends, which loads only one micro-frontend at a time, there is no need to coordinate naming conventions or similar agreements across multiple teams. The teams will mainly share best development practices and approaches to make the micro-frontends similar and allow team members to understand the codebase of other micro-frontends or even join a different team.

The micro-frontends will be static artifacts and therefore highly cacheable through a content delivery network (CDN), so there's no need for runtime composition on the server side. The delivery strategy will need to change, however, because of this aspect. Currently, ACME is serving all the static assets directly from the application server layer. Because the API integrations are happening on the client side, there will be no need to continue maintaining the application servers for serving static contents but only for exposing the backend API.

ACME decided to use object storage like AWS S3, storing all the artifacts to serve in production in a regional bucket and enhancing the distribution across all the countries they need to serve using a CDN such as Amazon CloudFront. This will simplify the infrastructure layer, reducing the possible issues happening in production due to misconfiguration or scalability. Additionally, the frontend has a different infrastructure than the API layer, allowing the frontend developers to evolve their infrastructure as needed. This new infrastructure allows every team to independently deploy their micro-frontend artifacts (HTML, CSS, JS files) in a S3 bucket and have them automatically available for the application shell to load them.

Another goal for this migration is to reduce the risk of bugs in production when a new micro-frontend version is deployed while immediately creating value for the users and the company without waiting for the entire application to be rewritten with the new architecture. Considering the simple frontend infrastructure adopted for the project, the ACME teams decided to leverage Lambda@Edge, a serverless computation layer offered by AWS (see Chapters 10 and 11), for analyzing the incoming traffic and serving a specific artifact to the application shell, implementing a de facto frontend canary release mechanism at the infrastructure level that won't impact the application code but will run in the cloud (see [Figure 10-10](#)).

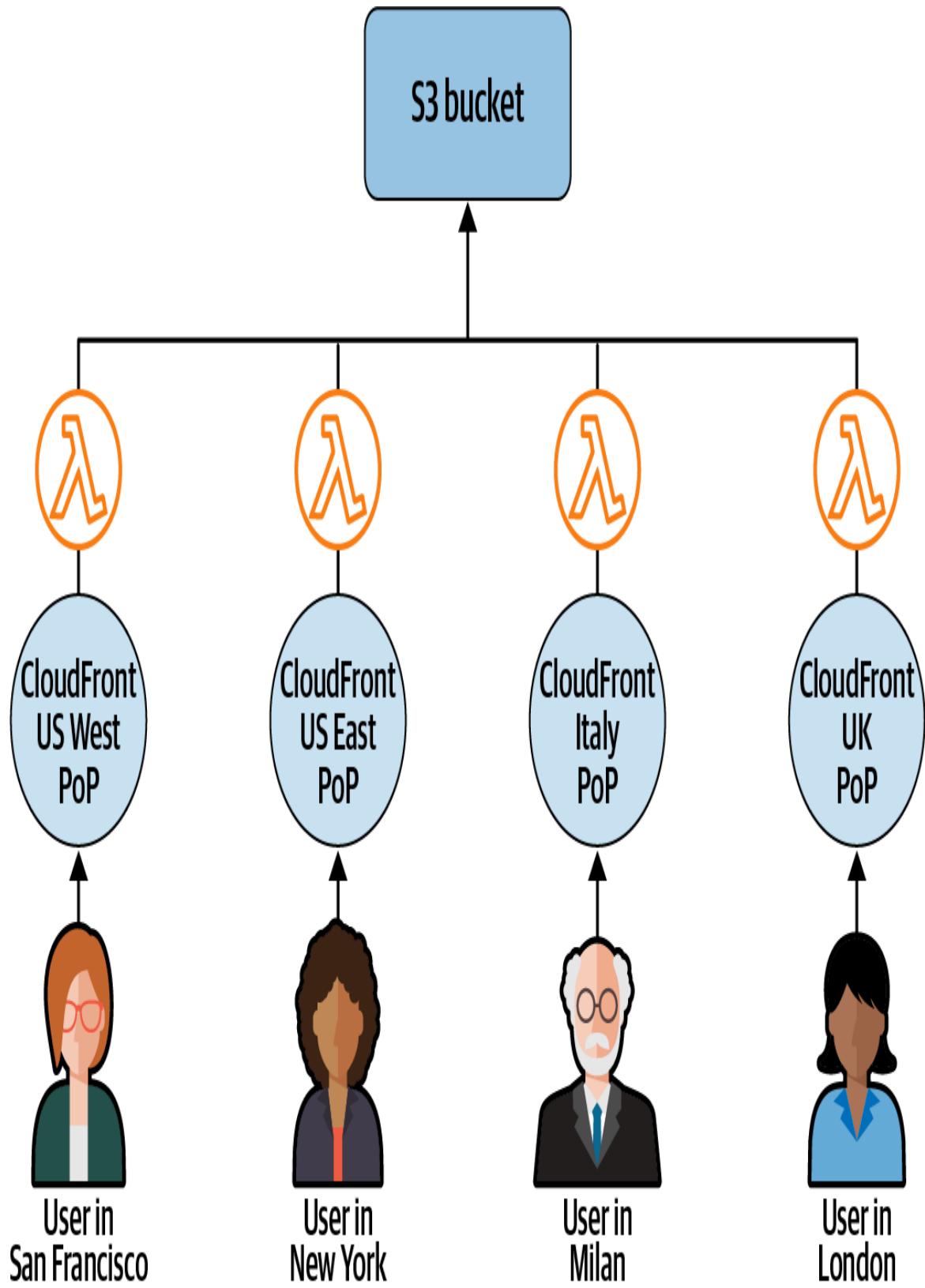
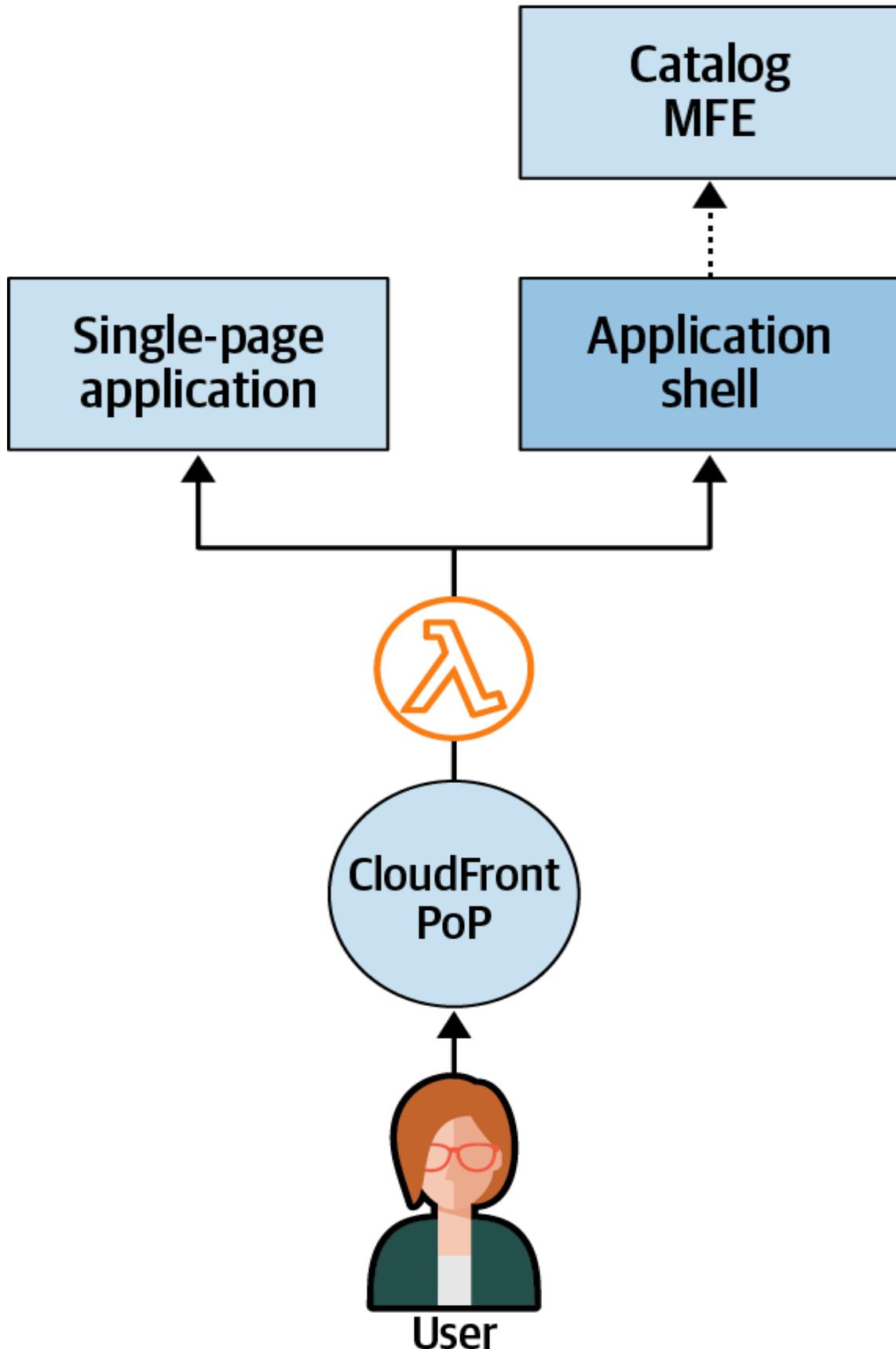


Figure 10-10. Simple infrastructure based on S3 bucket, Lambda@Edge, and a CDN like Amazon CloudFront distribution, with point of presence (PoP) available all over the world

Thanks to this implementation, ACME can also apply the strangler pattern (see Chapter 10) for gradually moving to micro-frontends while maintaining the legacy application. In fact, they can use the application URL to trigger the Lambda@Edge that will serve the legacy or application shell to the user (see [Figure 10-11](#)).



*Figure 10-11. Strangler pattern applied at the infrastructure level using Lambda@Edge for funneling the traffic toward either the legacy or micro-frontend application*

In the configuration file loaded by the Lambda@Edge at the initialization phase, developers mapped the URLs belonging to the legacy application and the ones to the micro-frontend application. Let's clarify this with an example. Imagine that the catalog micro-frontend is released first, because at this stage you want all or part of the traffic going toward the micro-frontend branch (see [Figure 10-11](#)). The authentication remains inside the legacy application, so after the user signs in or signs up, the SPA will load the absolute URL for the catalog (e.g., [www.acme.com/catalog](http://www.acme.com/catalog)). This request will be picked up by Amazon CloudFront, which will trigger the Lambda@Edge and serve the application shell instead of the SPA artifact.

This plan acknowledges that during the transition phase, a user will download more library code than before because they're downloading two applications at the end. However, this won't happen for existing users; they will always download the micro-frontend implementation, not the legacy one.

As you can see, there is always a trade-off to make. Because ACME's goal was finding a way to mitigate bugs in production and generate value immediately, these were the points they have to optimize for, especially if this is just a temporary phase until the entire application is switched to the new architecture. At this stage, ACME teams have made enough decisions to start the project. They decided to create a new team to take care of the catalog micro-frontend, which will be the first to be deployed into production when ready.

The teams know that the first micro-frontend will take longer to be ready because on top of migrating the business logic toward this new architecture style, the new team has to define the best practices for developing a micro-frontend that other teams will follow. For this reason, the catalog team starts with some proofs of concept to nail down a few details, such as how to share the authentication token between the SPA and the micro-frontend initially and then between micro-frontends when the application is fully

ported to this pattern, or how to integrate with the backend APIs with consideration for the migration on that side as well as the potential impact to API endpoints, contracts, and so on.

Initially, the team splits the work in two parts. Half of the team works on the automation pipeline for the application shell and the catalog micro-frontends. The other half focuses on building the application shell. The shell should be a simple layer that initializes the application retrieving the configuration for a specific country and orchestrates the micro-frontend life cycle, such as loading and unloading micro-frontends or exposing some functions for notifying when a micro-frontend is fully loaded or about to be unloaded.

The first iteration of this process will be reviewed and optimized when more teams join the project. The automation and application shell will be enhanced as new requirements arise or new ideas to improve the application are applied.

## Implementation Details

After identifying the next steps for the architecture migration, ACME has to solve a few additional challenges along the migration journey. These challenges, such as authentication and dependencies management, are common in any frontend project. Implementing the following features in a micro-frontend architecture may have some caveats that are not similar in other architectures. The topics we'll dive deeply into include:

- Application shell responsibilities
- Integration with the APIs that takes into account the migration to microservices happening in parallel
- Implementation of an authentication mechanism
- Dependencies management between micro-frontends
- Components sharing across multiple micro-frontends

- Introduction of design consistency in the user experience
- Canary releases for frontend
- Localization

In this way, we cover the most critical aspects of a migration from SPA to micro-frontends. This doesn't mean there aren't other important considerations, but these topics are usually the most common ones for a frontend application, and applying them at scale is not always as easy as we think.

## **Application Shell Responsibilities**

The application shell is a key part of this architecture. It's responsible for mounting and unmounting micro-frontends, initializing the application. It's also responsible for sharing an API layer for the micro-frontends to store and retrieve data from the web storage and triggering life cycle methods. Finally, the application shell knows how to route between micro-frontends based on a given URL.

## **Application Initialization**

The first thing the application shell does is consume an API for retrieving the platform configuration stored in the cloud. It consumes an API and returns feature flags, a services dictionary with a few endpoints used for validating tokens before granting the access to an authenticated area, and a list of micro-frontends available to mount.

After consuming the configuration from the backend, the application shell performs several actions:

1. Exposes the relevant part of the configuration to any micro-frontends and appends it to a window object so that every mounted micro-frontend will have access to it.
2. Checks business logic: if there is a token in web storage, validates it with the API layer. Routes the user to the authenticated area if

they're entitled or to the landing page if they're not.

3. Mounts the right micro-frontend based on the user's state (whether they're authenticated).

## Communication Bridge

The application shell offers a tiny set of APIs that every micro-frontend will find useful for storing or retrieving data or for dealing with life cycle methods. There are three important goals addressed by the application shell exposing these APIs:

- Exposing the life cycle methods for micro-frontends frees up memory before it is unmounted or removes listeners and starts the micro-frontend initialization when all the resources are loaded.
- Being the gatekeeper for managing access for the web storage in this way, the underlying storage for a micro-frontend won't matter. The application shell will decide the best way to store data based on the device or browser it is running on. Remember that this application runs on web, mobile, and living room devices, so there is a huge fragmentation of storage to take care of. It can also perform checks on memory availability and return consistent messages to the user in case all the permissions aren't available in a browser.
- Allow micro-frontends to share tokens or other data using in-memory or web storage APIs.

All the APIs exposed to micro-frontends will be available at the window object in conjunction with the configurations retrieved consuming the related API.

## Mounting and unmounting micro-frontends

Since ACME's micro-frontends will have HTML files as an entry point, the application shell needs to parse the HTML file and append inside itself the

related tags. For instance, any tag available in the body element of the HTML file will be appended inside the application shell body. In this way, the moment an external file tag is appended inside the application shell Document Object Model (DOM), such as JavaScript or a CSS file, the browser fetches it in the background. There is no need to create custom code for handling something that is already available at the browser level.

To facilitate this mechanism, the teams decided to add an attribute in the HTML elements of every micro-frontend for signaling which tags should be appended and which should be ignored by the application shell.

## Sharing state

A key decision made by ACME was that the sharing state between micro-frontends has to be as lightweight as possible. Thus, no domain logic should be shared with the application shell that should be only used for storing and retrieving data from web storage. Because the vertical split architecture means only one micro-frontend can load at a time in the application shell, the state is very well encapsulated inside the micro-frontend. Only a few things are shared with other micro-frontends, such as access tokens and temporary settings that should expire after a user ends the session. Some components will be shared across multiple micro-frontends, but in this case there won't be any shared states, just well-defined APIs for the integration and a strong encapsulation for hiding the implementation details behind the contract.

## Global and local routing

Last but not least, the application shell knows which micro-frontend to load based on the configuration loaded at runtime, where a list of micro-frontends and their associated paths is available. In this configuration, every micro-frontend has a global path that should be linked to it. For example, the authentication micro-frontend is associated with *acme.com/account*, which will load when a user types the exact URL or selects a link to that URL.

When a micro-frontend is an SPA, it can manage a local route for navigating through different views. For instance, in the authentication micro-frontend, the user can retrieve a password or sign up to the service. These actions have different URLs available, so that the logic will be handled at the micro-frontend level. The application shell is completely unaware, then, of how many URLs are handled inside the micro-frontend logic. The only important URL part handled by the application shell is the first level URL (e.g.: *acme.com/first-level-url*).

In fact, the micro-frontend appends a parameter belonging to a view to the path. The sign-up view, for instance, will have the following URL: *acme.com/account/signup*. The first part of the URL is owned by the application shell (global routing), while the *signup* part is owned by the micro-frontend. In this way, the application shell will have a high-level logic for handling a global routing for the application, and the micro-frontend will be responsible for knowing how to manage the local routing and evolving, avoiding the need to change anything in the application shell codebase.

## Migration strategy

During the migration period, the application shell will live alongside the SPA. In this way, ACME can deliver incremental value to their user, testing that everything works as expected and redirecting traffic to the SPA if it finds some bugs or unexpected behavior in the new codebase. The trade-off will be in the platform performances because the user will download more code than formerly. However, this method will enable one of the key business requirements: reduce the risk of introducing the micro-frontend architecture. In combination with the canary release, this will make the migration bulletproof to massive issues, thanks to several levers the teams can pull if any inconveniences are found during the migration journey.

## Backend Integration

Because ACME is migrating the backend layer from a monolith to microservices, the first step will be a lift and shift, in which they will

migrate endpoint after endpoint from the monolith to microservices. Using a [strangler fig pattern](#), they will redirect traffic to either the monolith or a new microservice. This means the API contract between frontend and backend will remain the same in the first release. There may be some changes, but they will be the exception rather than the rule.

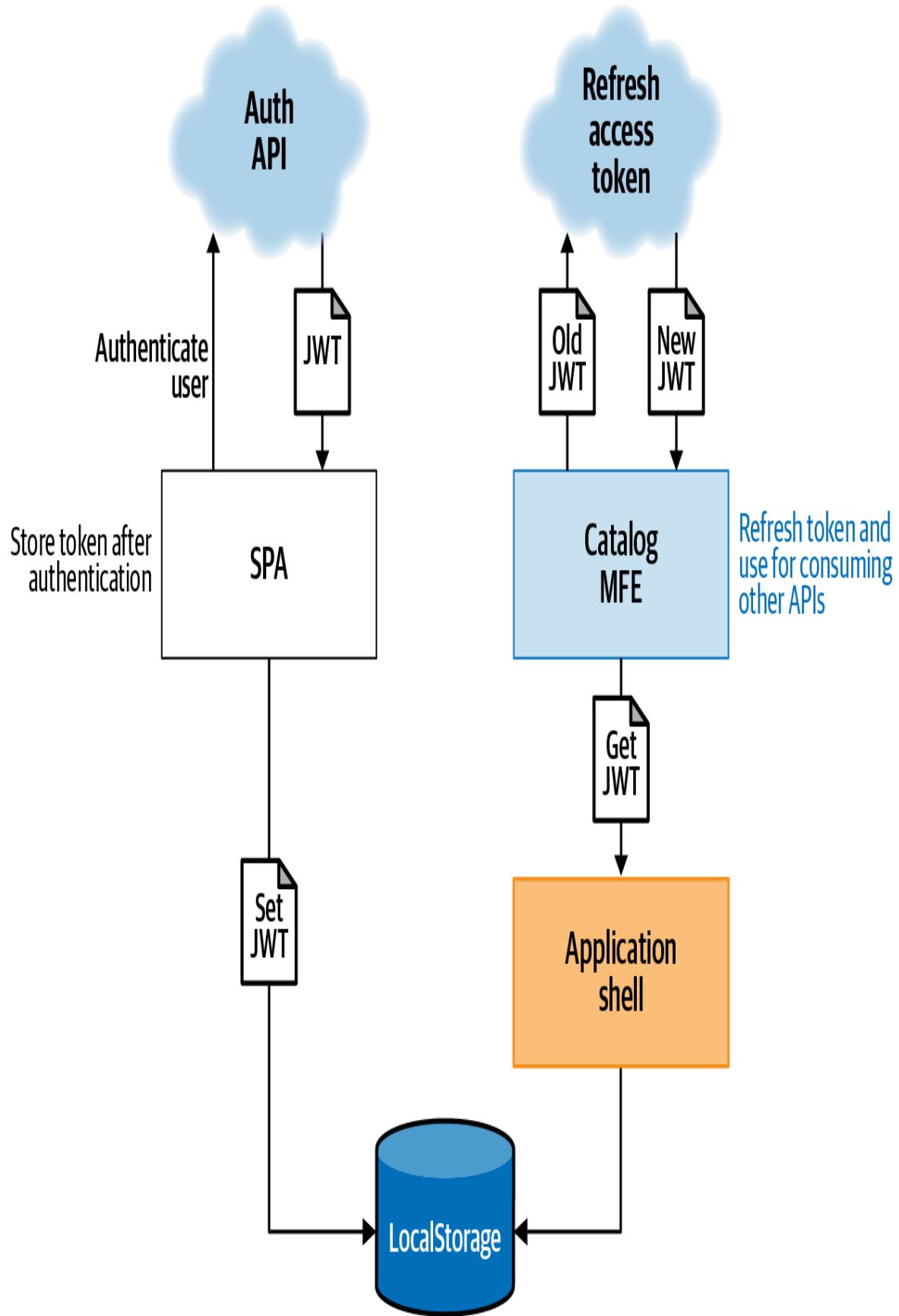
This approach allows ACME to work in parallel at different speeds between the two layers. However, it may also create a suboptimal solution for data modeling. The drastic changes required to accommodate microservices' distributed nature means some services may not be as well designed as they can be. For the ACME teams, though, this is an acceptable trade-off, considering there are a lot of moving parts to define and learn on this journey. The tech teams agreed to revisit their decisions and design after the first releases to improve the data modeling and APIs' contracts.

Based on this context, the development and platform teams agreed to use load balancers to funnel the traffic to the monolithic or microservices layer, so that the client won't need to change much. Every change will remain at the infrastructure level. Deciding the best way to roll out a new API version can be done without making the client aware of all these changes. The client will fetch the list of endpoints at runtime via the configuration retrieved initially by the application shell, eliminating the need to hardcode the endpoints in the JavaScript codebase.

## Integrating Authentication in Micro-Frontends

One of the main challenges of implementing micro-frontend architecture is dealing with authentication, especially with a shared state across multiple micro-frontends. The ACME teams decided to ensure that the application shell is not involved in the domain logic, keeping every micro-frontend as independent as possible. Thanks to the vertical-split approach, sharing data between micro-frontends is a trivial action because they can use web storage or query string for passing persistent data (e.g., simple user preferences) or volatile data (e.g., product ID).

ACME uses the local storage in its monolithic SPA for storing basic user preferences that don't require synchronization across devices, such as the video player's volume level and the JSON web token (JWT) used for authenticating the user. Because the developers want to generate immediate value for their users and company, they decided to stick with this model and deliver the authenticated area of the catalog alongside the SPA. When a user signs up or signs in within the SPA, the JWT will be placed in the local storage. When the application shell loads the catalog micro-frontend, the micro-frontend will then request the token through the application shell and validate it against the backend (see [Figure 10-12](#)).



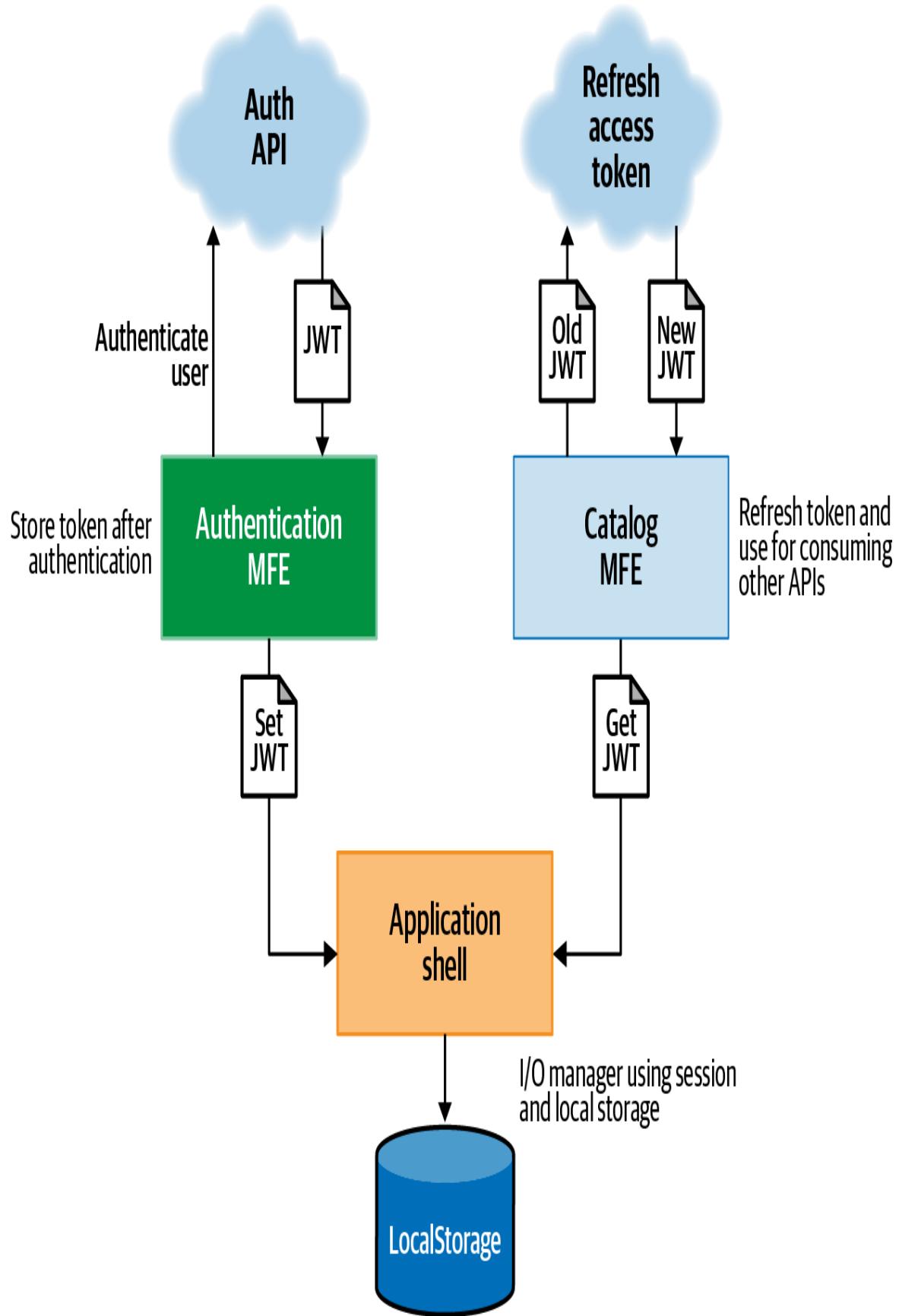
*Figure 10-12. During the migration, the SPA authenticates a user and stores the token in the local storage, which the authenticated micro-frontend retrieves once loaded*

Due to the local storage security model, the SPA, the application shell, and all the micro-frontends have to live in the same subdomain because the local storage is accessible only from the same subdomain. Therefore, the SPA will have to be moved from being served by an application server to the S3 bucket, where the new architecture will be served from.

## **LOCAL STORAGE SECURITY MODEL**

The data processed using the local storage object persists through browser shutdowns, while data created using the session storage object will be cleared after the current browsing session. It's important to note that this storage is origin specific. This means that a site from a different origin cannot access the data stored in an application's local database. For instance, if we store some website data in the local storage on the main domain `www.mysite.com`, the data stored won't be accessible by any other subdomain of `mysite.com` (e.g., `auth.mysite.com`).

Thanks to this approach, ACME can treat the SPA as another micro-frontend with some caveats. When it finally replaces the authentication part and finishes porting to this new architecture, every micro-frontend will have its own responsibility to store or fetch from the local storage via the application shell (see [Figure 10-13](#)).



*Figure 10-13. When the frontend platform is fully migrated to micro-frontends, every micro-frontend will be responsible for managing part of the users' authentication*

After the architecture migration, ACME will revisit where to store the JWT. The usage of local storage exposes the application to cross-site scripting (XSS) attacks, which may become a risk in the long run when the business becomes more popular and more hackers would be interested in attacking the platform.

## CROSS-SITE SCRIPTING

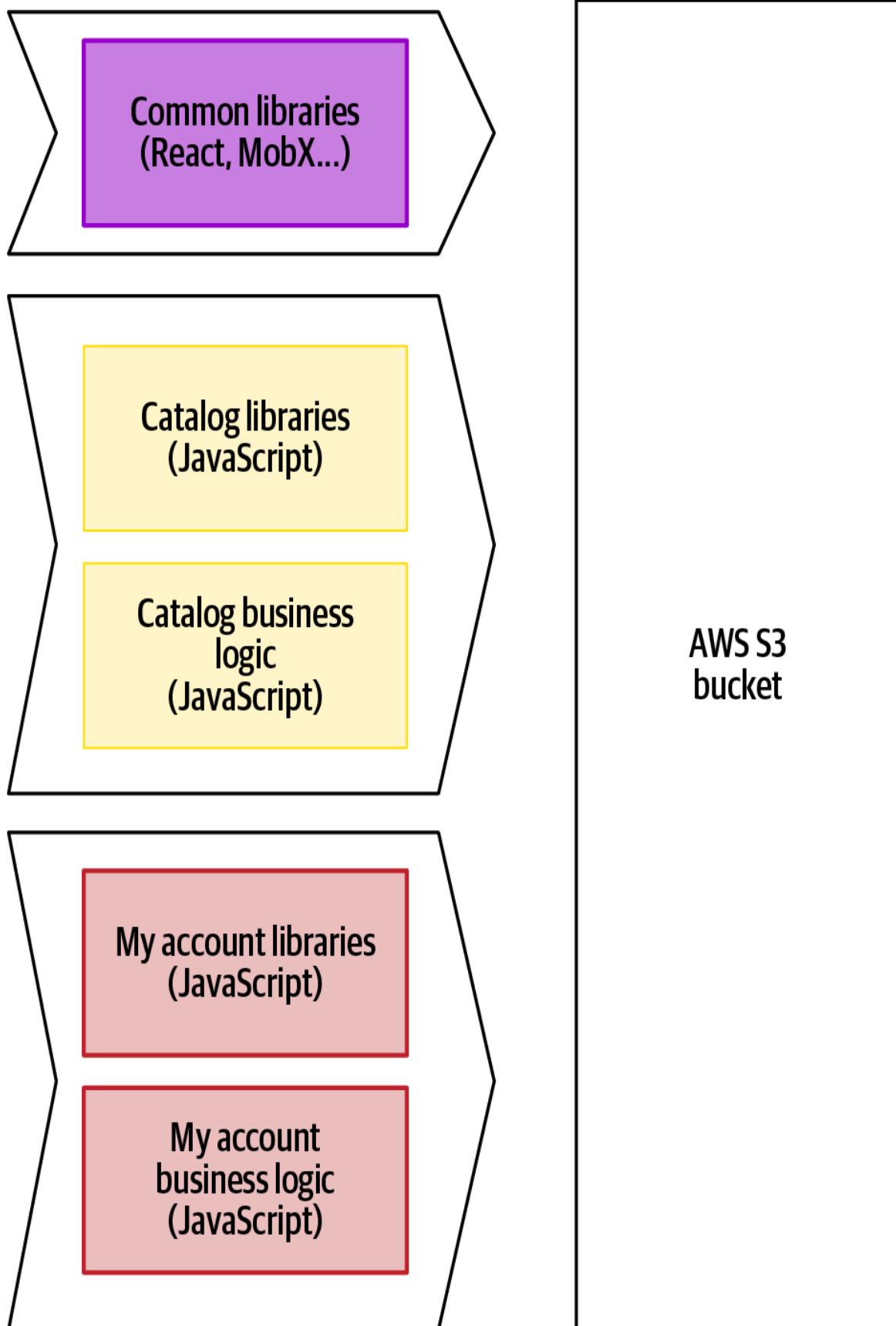
Cross-site scripting (XSS) attacks are a type of injection in which malicious scripts are injected into otherwise benign and trusted websites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser-side script, to a different end user. Flaws that allow these attacks to succeed are widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.

An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way of knowing that the script should not be trusted and will execute it. Because the browser thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information the browser retains and uses with that site. These scripts can even rewrite the content of the HTML page.

## Dependencies Management

ACME decided to share the same versions of React and MobX with all the micro-frontends, reducing the code the user has to download. However, the teams want to be able to test new versions on limited areas of the application so they can test new functionalities before applying them to the entire project. They decided to bundle the common libraries and deploy to

the S3 bucket used for all the artifacts. This bundle doesn't change often and therefore is highly cacheable at the CDN level (see [Figure 10-14](#)).



*Figure 10-14. Every micro-frontend builds and deploys its own JavaScript dependencies, apart from the common libraries, which have a separate automation strategy*

Other teams that want to experiment with new common library versions can easily deploy a custom bundle for their micro-frontend alongside the other final artifact files and use that version instead of the common one.

In the future, ACME's teams are planning to enforce bundle size budgets in the automation pipelines for every micro-frontend to ensure there won't be an exploit of libraries bundled together, which increases the bundle size and the time to render the whole application. This way, ACME aims to keep the application size under control while keeping an eye on the platform evolution, allowing the tech teams to innovate in a frictionless manner.

## Integrating a Design System

To maintain UI consistency across all micro-frontends, the tech teams and the UX department decided to revamp the design system available for the SPA using web components instead of Angular. Migrating to web components allows ACME to use the design system during the transition from monolith to distributed architecture, maintaining the same look and feel for the users. The first iteration would just migrate the components from Angular to web components maintaining the same UI. Once the transition is completed, there will be a second iteration where the web components will evolve with the new guidelines chosen by the UX department.

The initial design system was extremely modular, so developers can pick basic components to create more complex ones. The modularity also means the design system library will not be a huge effort to migrate and the implementation will be as quick as it was before.

Due to the distributed nature of the new architecture, ACME decided to enforce at the automation pipeline level using a fitness function for checking that every micro-frontend should use the latest version of the design system library. In this way, they will avoid potential discrepancies across micro-frontends and force all the teams to be up to date with the

latest version of the design system. The fitness function will control the existence of the design system in every micro-frontend's `package.json` and then validate the version against the most up-to-date version in case the design system version is older than the current one. The build automation will be blocked and return a message in the logs, so the team responsible for the micro-frontend will know the reason why their artifact wasn't created.

## Sharing Components

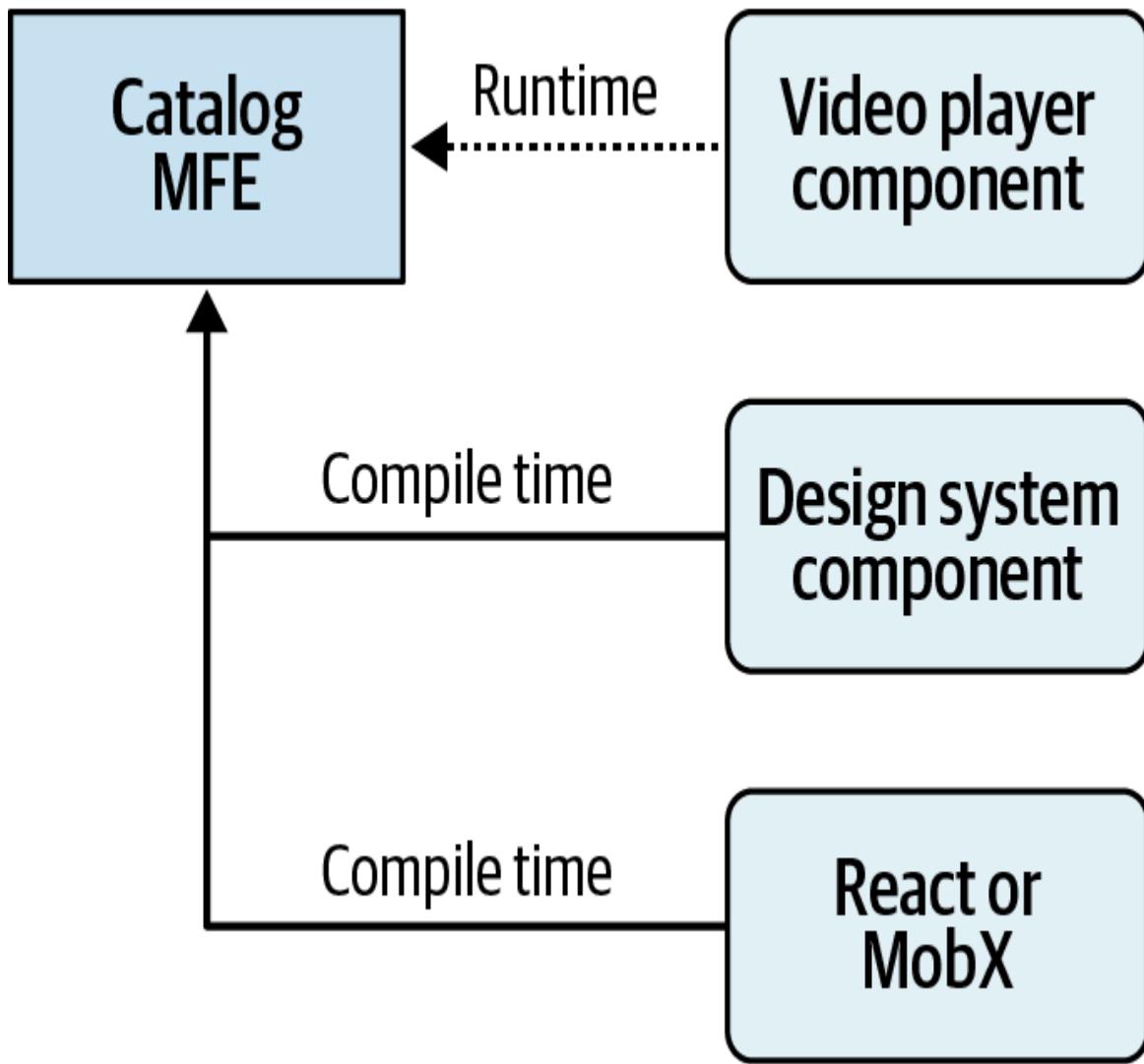
ACME wants a fast turnaround on new features and technical improvements to reduce external dependencies between teams. At the same time, it wants to maintain design consistency and application performance, so it will share some components across micro-frontends. The guidelines for deciding whether a component may be shared is based on complexity and the evolution, or enhancement, of a component.

For example, the footer and header formerly changed once a year. Now, however, these components will change based on user status and the area a user is navigating. The solution applied for the header and footer will be created with the different modular elements exposed by the design system library. These two elements won't be abstracted inside a component, since the effort to maintain this duplication is negligible and there are only a few micro-frontends to deal with. These decisions may be reverted quickly, however, if the context changes and there are strong reasons for abstracting duplicated parts into a components library.

To avoid external dependencies for releasing a new version or bug fix inside a component, the teams decided to load components owned by a single specialized team, like the video player components, at runtime. A key component of this platform, the video player evolves and improves constantly, so it's assigned to a single team that specializes in video players for different platforms. The team optimizes the end-to-end solution, from encoders and packagers to the playback experience. Because the header and footer will load at runtime, they won't need to wait until every micro-frontend updates the video player library. The video player team will be

responsible for avoiding contract-breaking changes without the need to notify all the teams consuming the component.

ACME will make an exception for the design system. Although it's built by a team focused only on the consistency of the user experience, the design system will be integrated at development time to allow developers to control the use of different basic components and to create something more sophisticated inside their micro-frontends. All the other components will be embedded inside a micro-frontend at development time, like any other library such as React or MobX (see [Figure 10-15](#)).



*Figure 10-15. In a micro-frontend, complex components owned by a single team are loaded at runtime, while all the others are embedded at compile time. The only exception is the design system due to its modular nature.*

None of the components created inside each team will be shared among multiple micro-frontends. If there are components that might simplify multiple teams' work if shared, a committee of senior developers, tech leads, and architects will review the request and challenge the proposal according to the principle defined at the beginning of the project. These principles will be reviewed every quarter to make sure they are still aligned with the platform evolution and business road map.

## Implementing Canary Releases

Another goal of this project is being able to release often in production and gather real data directly from the users. It's a great target to aim for, but it's not as easy to reach as we may think.

Based on its infrastructure for serving frontend artifacts, ACME decided to implement a canary release mechanism at the edge, so that it can extend the logic of its Lambda@Edge once the migration is completed, adding logic to manage the micro-frontend releases.

ACME will also need to modify the application shell to request specific micro-frontend versions and delegate retrieving the exact artifact version to Lambda@Edge. The tech teams decided to identify every micro-frontend release using **semantic versioning (semver)**. This allows them to create unique artifacts, appending the semver in the filename and easily avoid caching problems when they release new versions.

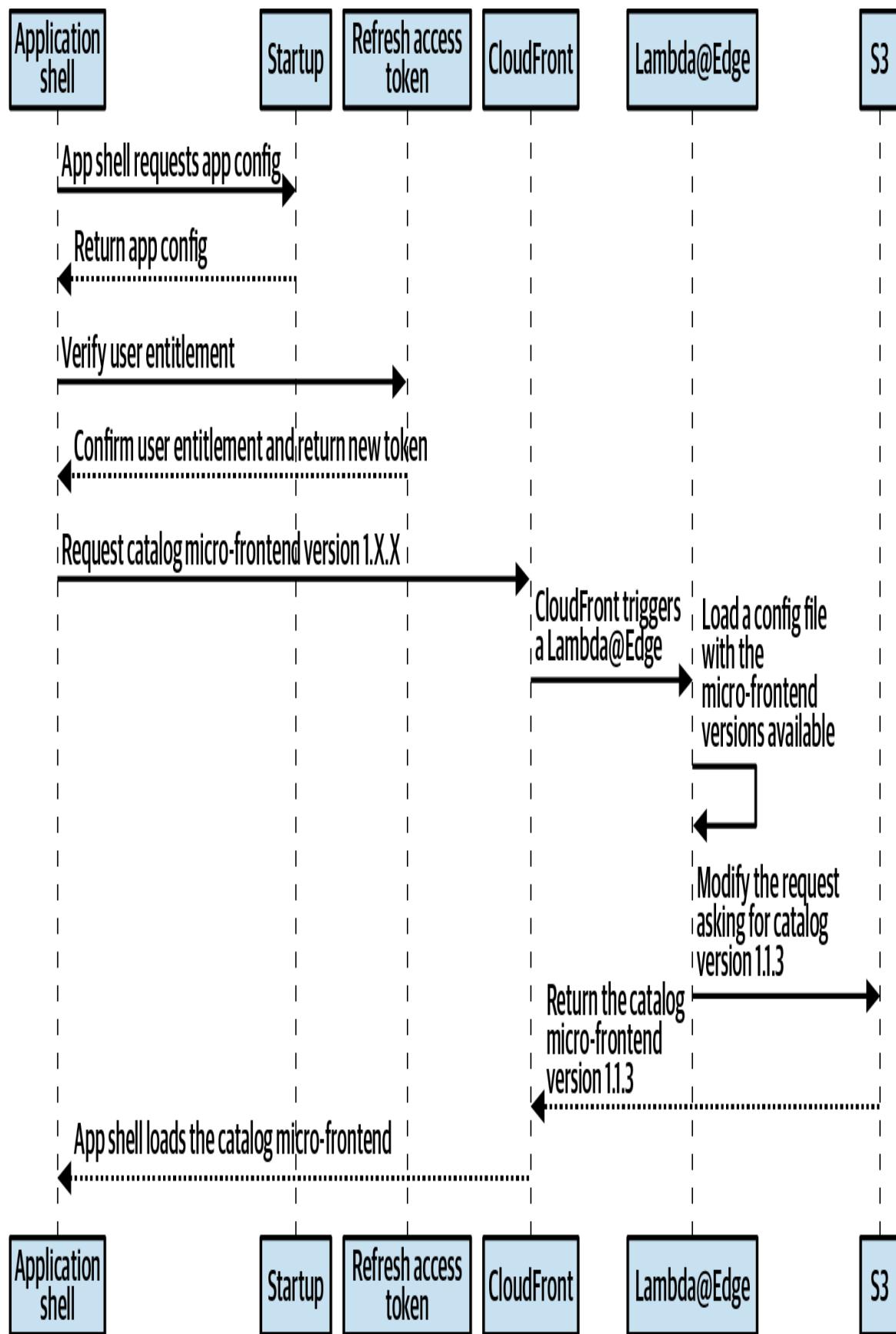
## SEMANTIC VERSIONING

Given a version number MAJOR.MINOR.PATCH like 1.1.0, increment the:

- MAJOR version when you make incompatible API changes
- MINOR version when you add functionality in a backward-compatible manner
- PATCH version when you make backward-compatible bug fixes

Additional labels for prerelease and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

As we can see in [Figure 10-16](#), first the application shell retrieves a configuration from the APIs. The configuration contains a map of available micro-frontends versions where only the major version is specified (e.g., 1.x.x). This allows the teams to upgrade the application while maintaining backward compatibility. They also only need to upgrade the major version when an API breaking change updates the configuration file served by the backend.



*Figure 10-16. Sequence diagram describing how ACME implements canary releases for micro-frontends*

When the artifact request hits Amazon CloudFront, a Lambda@Edge that retrieves a list of versions available for the micro-frontends is triggered; the traffic should then be redirected to a specific version. The logic inside the Lambda will associate a random number—from 1 to 100—to every user. If a user is associated with 20% and 30% of the traffic should be redirected to a new version of the requested micro-frontend, that user will see the new version. All the users with a value higher than 30 will see the previous version.

The Lambda returns the selected artifact and generates a cookie where the random value associated with the user is stored. If the user comes back to the platform, the logic running in the Lambda will validate just the rule applied to the micro-frontend requested and evaluate whether the user should be served the same version or a different one based on the traffic patterns defined in the configuration. As a result, both authenticated and unauthenticated traffic will have a seamless experience during the canary exploration of an artifact.

Using this mechanism, ACME can reduce the risk of new releases without compromising fast deployment because they can easily move users from newer versions to an older one simply by modifying the configuration retrieved by the Lambda@Edge.

The team plans to introduce a **frontend discovery service** to facilitate canary releases once the system has fully migrated to micro-frontends (as discussed in Chapter 9). This approach will allow seamless integration with the initial call made by the application shell, ensuring that the number of API requests made at the start of the application remains consistent, while also supporting the canary release process.

## Localization

The ACME application has to render in different languages based on the user's country. By default, the application will render in English, but the

product team wants the user to be able to change the language in the application and have the choice to persist for authenticated users inside their profile settings, creating a seamless experience for the user across all their devices.

In this new architecture, ACME tech teams have to consider two forces:

- Every micro-frontend has a set of labels to display in the UI, some of which may overlap with other micro-frontends, such as common error messages.
- Every micro-frontend represents a business subdomain, so the service has to return just enough labels to display for that specific subdomain and not much more; otherwise, resources will be wasted.

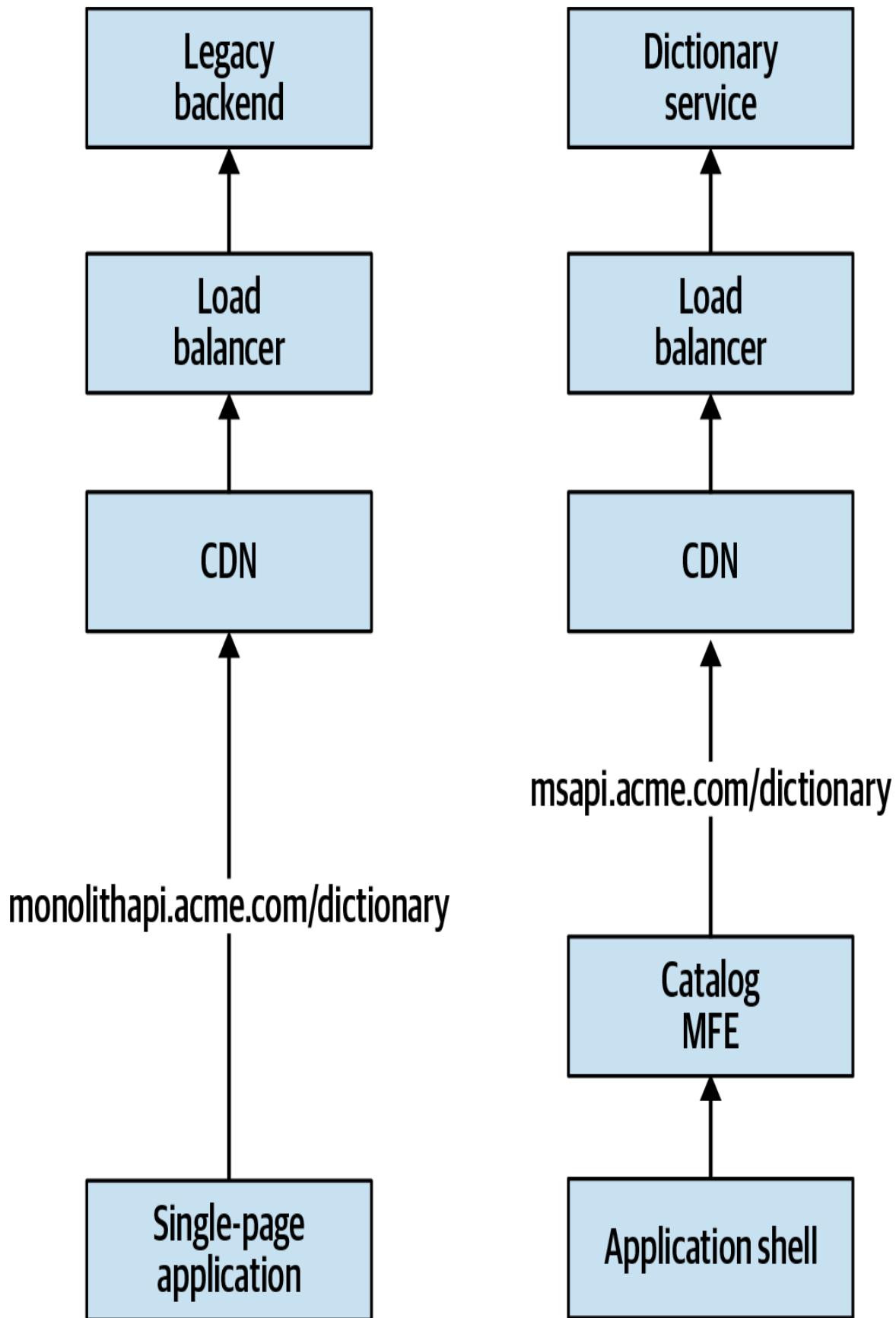
ACME tech teams decided to modify the dictionary API available in the monolith to return only the labels needed inside a micro-frontend. In this way, the SPA can still receive all the labels available for a given language, and the micro-frontend will only receive the label needed for its subdomain during the transition (see [Figure 10-17](#)). At migration completion, all the micro-frontends will consume the microservices API instead of the legacy backend, and there won't be a way to retrieve all the labels available in the application through the legacy backend.

When a micro-frontend consumes the dictionary API, it has to pass the subdomain as well as the language and country related to the labels in the request body in order to display them in the user interface. When it receives the request, the microservice will fetch the labels from a database based on the user's country, favorite language, and the micro-frontend subdomain.

Because micro-frontends are not infinite and the platform supports less than a dozen languages, having a CDN distribution in front of the microservice will allow it to cache the response and absorb the requests coming from the same geographical area.

Being able to rely on the monolith via a different endpoint during micro-frontend development creates a potential fallback, if needed. It allows older

versions of native applications on mobile devices to continue working without any hiccups.



*Figure 10-17. The micro-frontend consumes a new API for fetching the labels to display in the interface through a new microservice. The SPA will consume the API from the legacy backend.*

## Summary

In this chapter, we have gathered all the insights and suggestions shared across the book and demonstrated how they play out in a real-world example. Sharing the reasoning behind certain decisions—the *why*—is fundamental for finding the right trade-off in architecture and, really, in any software project. When you don’t know the reasons for certain decisions inside your organization, I encourage you to find someone who can explain them to you. You will be amazed to discover how much effort is spent before finding the right trade-offs between architecture, business outcomes, and timing.

You will see in your career that what works in one context won’t work in another because there are so many factors stitching the success of the project together, such as people skills, environment, and culture. Common obstacles include the seniority of the engineers, company culture, communication flows not mapping team interactions, dysfunctional teams, and many more.

When we develop any software project at scale, there are several aspects we need to take into consideration as architects and tech leaders. With this chapter, I wanted to highlight the thought process that moved ACME from an SPA to micro-frontends because these are decisions and challenges you may face in the real world. Some of the reasoning shared in these pages may help you to take the right direction to project success.

One thing that I deeply like about micro-frontends is that we finally have a strong say about how to architect our frontend applications. With SPAs, we followed well-known frameworks that provided us speed of development and delivery because they solved many architectural decisions for us. Now we can leverage these frameworks and contextualize them using their strengths in relevant parts of our projects.

Now's our chance to shape this space with new tools, practices, and patterns. Imagine a world where micro-frontends empower teams to work independently, yet seamlessly integrate their components into a cohesive whole. Consider the agility and speed we can achieve by breaking down monolithic structures and embracing a modular approach. With micro-frontends, we're not just building applications; we're crafting experiences that can evolve and scale with ease.

The only thing holding us back is our imagination. So let's dive in, experiment, and push the boundaries of what's possible.

# Chapter 11. Introducing Micro-Frontends in Your Organization

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 13th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the author at [building.microfrontends@gmail.com](mailto:building.microfrontends@gmail.com).

You’ve arrived at the last chapter of this book and have learned a lot about how to create micro-frontends, the best architectural approach for your project, and all the best practices to follow to make your project successful. It’s time to start your project and write a few lines of code, right?

Not quite. There are still some key topics related to the human aspect that we must take into consideration when we introduce this architecture, just as we do whenever we revisit our architecture or introduce a new one. When making significant changes to architecture, we need to think about how to organize the communication flows, how to avoid siloed groups, and how to empower the developers to make the right decisions inside a business domain. These are just some of many important considerations related to the human side of the project we need to think about at the beginning and during the entire life cycle. Micro-frontends may help you mitigate some of these considerations, but they can make others more complex if not

approached properly. Therefore, it's crucial for you to invest the time needed to analyze your current organization structure and see how it would fit inside your new architecture.

## Why Should We Use Micro-Frontends?

Tech leaders and CTOs often ask this question when someone introduces the idea of micro-frontends inside an organization. It's a valid question, and the best way to answer it is to use a common language to evaluate the benefits of this architecture paradigm. Micro-frontends bring several benefits to the table, such as team independence, riskless deployment, reduction of cognitive load for developers, fast iterations, and innovation. Despite all of that, they also bring challenges, such as a risk of creating silos inside the organization, higher investment in automation pipelines, and risk of user interface discrepancies. When you introduce the idea of micro-frontends to your organization, focus your presentation on not only the technical benefits but also the organizational benefits. Let me provide some food for thought to help you prepare an impactful presentation for your stakeholders:

- Point out that micro-frontends allow faster feature iterations and reduce the risk of introducing bugs into the entire application.
- Research and describe the context you operate in daily and why micro-frontends may help you to achieve business goals.
- List the problems you are trying to solve with this paradigm.
- Ponder the best way to implement micro-frontends in your context.
- Analyze the impact this architecture may have on team communication.
- Identify the ideal governance for managing such an architecture.
- Retrieve metrics from your automation pipeline, like time to deployment and testing, and think how you would be able to

improve them.

These are just some topics that are relevant to your organization's tech leaders or clients.

Remember to present not just a technical solution, which can leave many organizational challenges to overcome. Instead, think about an end-to-end transformation that brings value to the company as well as to your customers. To discover the best technical solution for your context, I strongly encourage you to first run a proof of concept (PoC) to understand the challenges and benefits of this approach better; what works for one team or organization doesn't always work for another. Be mindful and share the insights that will work best for your organization with your peers and tech leaders. Try to involve the right people up front, because understanding the context in which you operate may result in a nontrivial activity, especially in midsize to large organizations, where you may be dealing with distributed teams whose culture and context change office by office. In the following sections, you'll discover some insights on how to manage the governance, documentation, organization setup, and communication flows for a micro-frontend project.

## Data to the rescue

In recent years, I've spoken with numerous companies in the industry, from tech leaders to VPs worldwide. A common strategy for raising awareness about the necessity of a distributed system in frontend development involves establishing a baseline and setting goals to facilitate significant improvements.

Let me explain with an anecdote. In 2024, I started a video podcast where I interviewed engineers, architects, and micro-frontends practitioners. During an interview with Warren Fitzpatrick, Principal Engineer at Dunelm, he shared how he convinced management to embrace this model. I had a similar experience at my previous company, using the same approach to gain development approval. In both cases, the strategy involved identifying

existing problems that hindered innovation and new feature development, establishing a baseline, and presenting potential improvements with data relevant to the business.

Warren shared that their key problem was the slow deployment of features to production due to convoluted automation pipelines and a high number of tests. Interestingly, Warren didn't emphasize technical improvements to his stakeholders. Instead, he focused on the slow pace of deploying new features and how a more modular strategy, like micro-frontends, could increase deployment frequency. Although their first attempt didn't achieve the desired results, they adjusted practices and revisited decisions. Now, they are fully committed to a server-side rendering micro-frontends architecture powered by serverless services in the cloud.

The main takeaway from this story is that identifying a problem slowing down the business can be an effective strategy for encouraging investment in a new approach. This method is not limited to micro-frontends but applies to any organizational improvement. Gathering data and proving with a proof of concept (PoC) that the company could improve its current situation is a technique I've used extensively in the past and continue to use. It might not always work, but it will certainly spark discussions within your tech leadership, guaranteed!

#### NOTE

If you are interested in learning more about [Dunelm's journey](#) and the insights from other companies who embraced micro-frontends, I highly encourage you to listen to the “micro-frontends in the trenches” show on [Spotify](#), [Apple Podcast](#) or [YouTube](#).

## Create a trade-off analysis

A methodical way to work with data when making architectural and design decisions is to conduct a trade-off analysis. This analysis typically involves considering three key dimensions:

- Business requirements
- Architecture characteristics
- Organizational capabilities

### NOTE

I strongly recommend documenting these findings to allow for revalidation of your decisions at any time. This practice will help your team make informed decisions and provide future team members with a clear understanding of the rationale behind the chosen architectural approach.

## Business requirements

Always start with what the business aims to achieve. Engage with key stakeholders to understand why a particular characteristic is important, what they want to attain, and why it matters. It is even better if you can gather some baseline metrics from which to work backwards.

For example, if the business wants to achieve a faster time to market for new features, investigate the current time frame from ideation to successful deployment of your workload and identify the bottlenecks. Visualizing these findings will help communicate the potential solutions more effectively and improve the current process.

## Architecture characteristics

The next step is to gather the architectural characteristics defined in collaboration with your tech leadership for implementation within the system. For instance, if you need to build a latency-sensitive solution in the cloud, a multi-region server-side approach could help reduce response times and improve the core web vitals of your application.

If you need to integrate a design system into a micro-frontends implementation, consider automating this process to accelerate the feedback loop for developers, such as when they open a pull request.

## Organization capabilities

Finally, you need to understand your specific context. This is crucial for selecting the right approach. Too often, companies adopt a mechanism or practice simply because a large company did, but this does not necessarily lead to success for your organization.

I recommend identifying the bottlenecks your developers face, reviewing the current feedback loop, and planning improvements through automation. Additionally, assess the skills you have in-house and consider providing training to enhance your team's capabilities. Every context is different, and it is essential to take this into account.

This is a valuable exercise to invest time in before starting your journey with micro-frontends. It will provide clarity of intent, set expectations, and ensure alignment across teams.

## The Link Between Organizations and Software Architecture

What sort of software architecture should you be implementing? You'd be forgiven for wanting to copy others' success. But there is no such thing like the "best architecture". Your job is finding the best trade-off for your context. Perfection is unavailable, unfortunately. What we need to create is an architecture that fits our organization's needs and, especially, the context we operate in.

We often hear conference talks that explain a specific use case. The ideas and solutions the speaker brings up feel like a perfect fit for what we are trying to solve in our organization. Unfortunately, it's not always the case. In any talk or book, the solution to a problem is given from the perspective of just one person, who may represent only part of the organization. Often the speaker or writer focuses more on the how and less on the context where their specific solution was successful. There's little on *why* this solution worked for this context. But that context provides us with the information we need to make the right trade-offs for our architectures.

What software architecture should you be implementing? Which business and architectural characteristics should you take into account? How would you express them in your design decisions? These are the initial questions everyone should start with. There is no single architecture that works well in all cases and at all times. You have to customize your architecture for your needs, applying the patterns that solve your problems and fit your situation best. Bear in mind that the business evolves over time and, therefore, a good trade-off today may not be a good one tomorrow.

Modularity is the key for moving in the direction the business wants to go, allowing it to arrive faster and with minimal complexity. At the same time, modularity is far from a trivial task. It requires discipline, analysis, and a lot of work from everyone involved in the project. Micro-frontends are no exception to this. They're not suitable for all projects. But they may be useful when you work in a mid-sized to large environment, with three or more teams working exclusively on the frontend side of a project. If you have cross-functional teams working already with microservices, it's very likely micro-frontends are a suitable approach for your client-side application. Even with mobile applications, many organizations started to use micro-frontends in conjunction with React Native, for instance. They may also be really helpful when scaling the organization is a requirement, when an application's success depends on time to market, when we are transitioning from a legacy application to a new one and want to generate immediate value for our users instead of waiting several months before the application is finished, and in many other scenarios.

You may want to embrace a micro-frontends implementation different from the approaches described in this book or even try new approaches for solving specific problems inside your organization. This is absolutely fine, as long as there are strong reasons for doing so and the new trade-off will benefit a team, the entire organization, or a process. Remember, we are writing code for our customers or users, and this should be at the forefronttop of mind for every developer's mind when developer working on a software project.

## How Do Committees Invent?

In the first chapter, I briefly introduced Conway's law: "Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure." This law is from the 1968 paper "[How Do Committees Invent?](#)" by Melvin Conway. In his paper, Conway explains how software architecture is usually designed alongside the company structure. But this is not always the case. Sometimes we want to focus on a high-level architecture that is the best trade-off for designing a platform and then restructure our teams around that architecture. In these situations, we are applying the opposite technique, called the "inverse Conway maneuver." In the "Stages of Design" section of the paper where he describes the steps for designing software architectures, Conway recommends the following:

- Understanding of the boundaries, both on the design activity and on the system to be designed, placed by the sponsor and by the world's realities
- Achievement of a preliminary notion of the system's organization so that design task groups can be meaningfully assigned

Though these principles are half a century old, they feel more relevant than ever. In the book *Accelerate*, authors Nicole Forsgren, Jez Humble, and Gene Kim share incredible research on the best practices of high-performance organizations from across multiple industries. The inverse Conway maneuver plays an important part in the social-technical aspect of every high-performance organization the authors study.

Architecture and team communication are strongly linked. It's crucial to understand and internalize this, because it will greatly influence which micro-frontend architecture we decide to use. Ideally, we would design the best architecture possible for a given context and then assign teams to fulfill the design, but that's not always possible. In fact, in my experience, it's rarely possible, but it could happen sometimes. In cases where we need to

respect the current organization structure, we need to take into consideration teams' current communication flows, daily interactions, and the organization structure during our architecture design process in order to design an architecture suitable for our teams. Realizing that communication flow and architecture are linked together allows you to aim for the best architecture for the context you operate within.

When considering communication flows, we need to distinguish between collocated and distributed teams. Sam Newman shared a very valid point in an [article about Conway's law](#): “The communication pathways that Conway refers to are in contrast to the code itself, *where a single codebase requires fine-grained communication, but a distributed team is only capable of coarse-grained communication*. Where such tensions emerge, looking for opportunities to split monolithic systems up around organizational boundaries will often yield significant advantages” (my emphasis).

The communication type, coarse or fine, is another essential consideration when we design our architecture. In a distributed company, the best way to achieve fine-grained communication across teams is to have a fully remote organization so that there isn't any difference between teams. However, the moment an organization has multiple developer centers in multiple locations, the communication flow changes again, and having multiple teams working on the same area of the codebase in different offices may be more of an issue than a benefit. A good way to mitigate this problem is by assigning all the subdomains that intersect and share similarities to a colocated team instead of distributing them. For example, imagine a video-streaming platform composed of the following areas:

- Landing page
- Movies catalog
- Playback
- Search
- Personalization

- Sign-in
- Sign-up
- Payment
- Remember email
- Remember password
- Help
- My account

When we group subdomains that intersect and share similarities, we can group subdomains related to new-user onboarding:

- Landing page
- Sign-up
- Payment

Then we can group subdomains related to existing users who may or may not already have authenticated inside our platform:

- Sign-in
- Remember email
- Remember password
- My account

Finally, we can group subdomains related to existing users who have authenticated:

- Movies catalog
- Playback
- Search

- Personalization

What we've done is group subdomains by user journey, that is, subdomains that intersect. The playback experience, for instance, certainly has more in common with the movies catalog than with the Help pages. Did you notice that the Help domain wasn't in any of the previous groups? That's because the Help section may be useful for authenticated and nonauthenticated users alike.

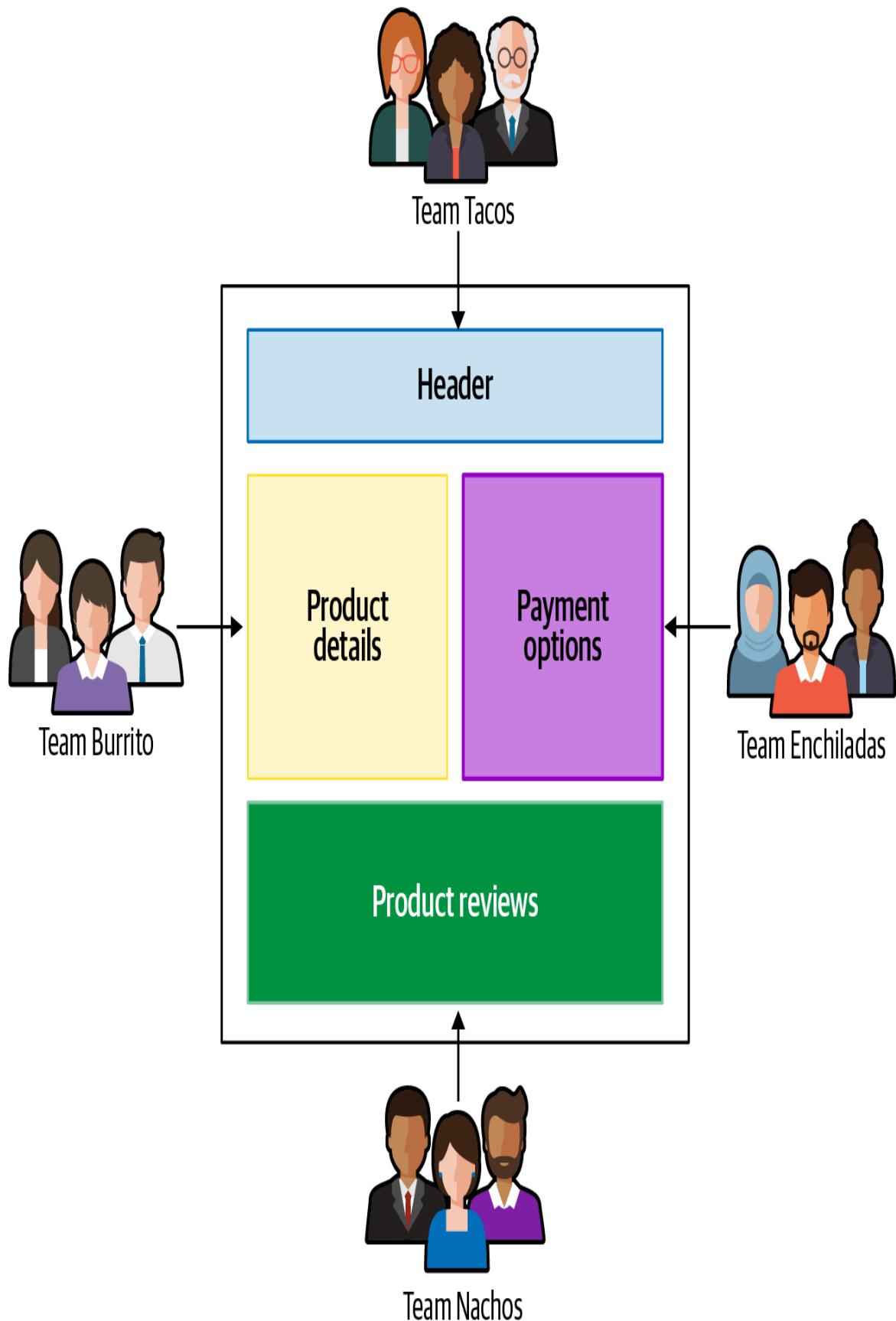
It's very hard to have a perfect split between the user journeys. This is also true when we identify the different subdomains available in these user journeys. In this example, we ended up with several buckets of user journeys, with one or more subdomains within each bucket. However, the moment we are able to identify these subdomains, we can determine how to map the development of our micro-frontends inside a company. This exercise may force us to swap some domains from one office to another one, although it provides great long-term benefits reducing external dependencies across offices and keeping them in the same one, maintaining a fine-grained communication where subdomains work together and a coarse-grained communication across offices where the need of synchronization happens less often and with fewer touching points. Yet, as stated before, it's very unlikely to have this perfect split, so don't be surprised if you end up with some subdomains developed by different offices. Just be sure to constantly review the performance and bottlenecks created inside the organization and adjust your decisions accordingly. When done right, microarchitectures are great because they can follow a business's evolution and, thanks to their modular nature, provide the tech department a great degree of flexibility.

## Features Versus Components Teams

Nowadays, many companies are debating which team structure they should use to enable developers to work on their tasks without impediments or external dependencies. Usually agile methodologies suggest one of two structures: features teams and components teams. Features teams, also

known as cross-functional teams, are organized with all the skills needed for delivering a specific feature. When we are developing a web application, for instance, a cross-functional team is organized to deliver user value around a specific feature. Let's imagine that we have a cross-functional team that will create the credit card payment feature inside an ecommerce store. The team will have both frontend and backend (a.k.a. full-stack) developers who will develop, test, and deploy the feature end to end. **Figure 11-1** depicts this example with Team Burrito, which is responsible for delivering the product details micro-frontend.

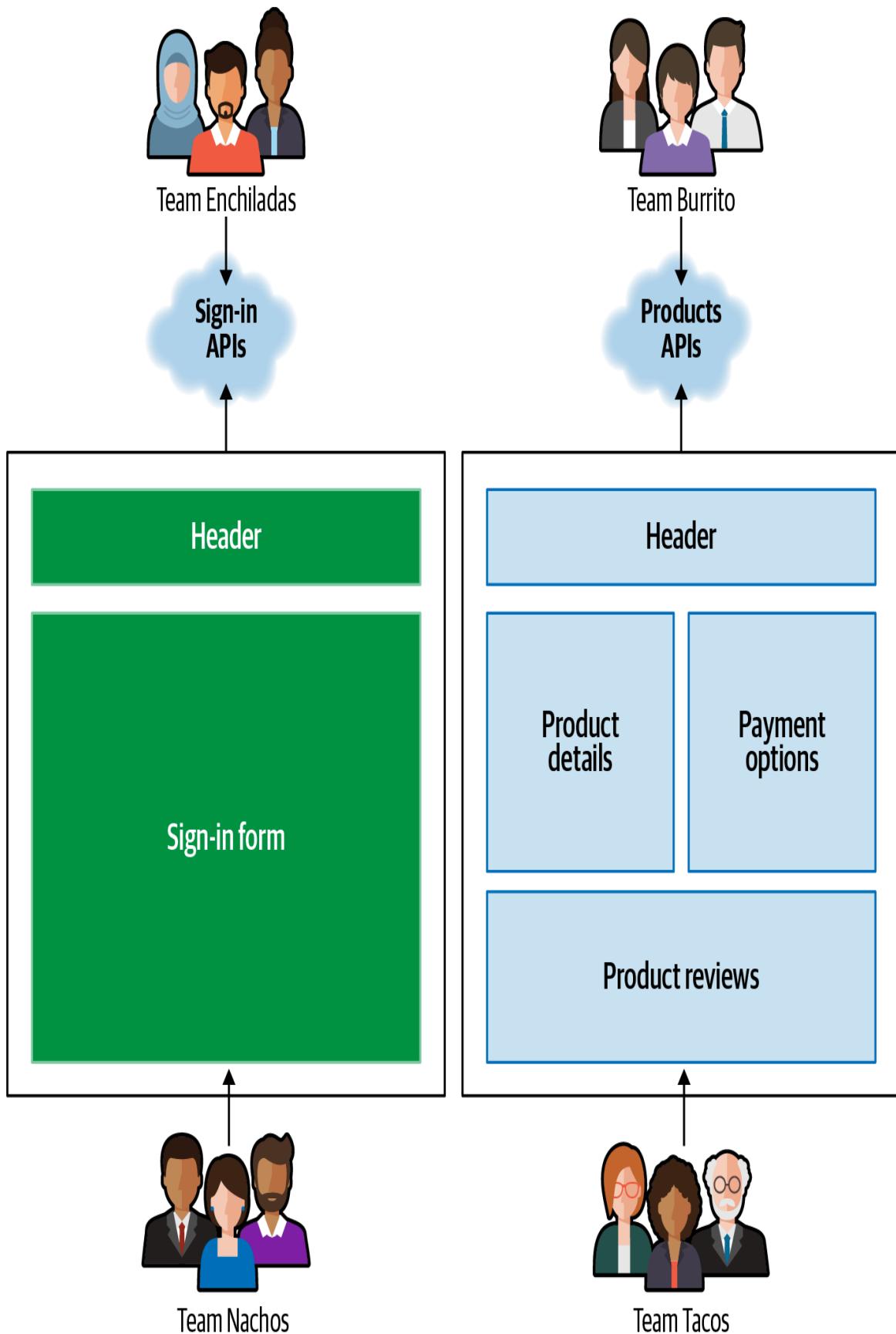
In this case, Team Burrito will be composed of full-stack developers working on the APIs, as well as the frontend that will consume these APIs.



*Figure 11-1. This diagram depicts the responsibilities of the features teams working on a horizontal-split architecture, where every team is responsible end to end for a micro-frontend and the APIs that it consumes*

Features teams are recommended when you can organize the architecture using a horizontal split, and every team is responsible for one or more micro-frontends. With this approach, the teams can focus solely on their features with an end-to-end approach, taking care of the entire feature life cycle. The cognitive load of a features team is more manageable than any other team's structure because every person responsible for generating value for the user is part of the same team. Usually, features teams are highly focused on the user, iterating constantly to enhance the user experience and the value created by their development effort. This approach allows feature teams to become domain experts in specific parts of the system, contributing not only to the technical aspects but also to product decisions. Providing valuable trade-offs accelerates the deployment of micro-frontends and reduces complexity upfront.

One challenge with this approach is that we will need to assign page composition to an external team or, more likely, to one of the teams developing a feature. The team responsible for composing the page must ensure the final result for the user is the one expected, without any logical or cosmetic bugs. We can mitigate this challenge by standardizing the page composition with templates and conventions. In this case, it would be easier to manage but will offer a lot less flexibility across page layouts. Another option would be working with component teams, where every team is responsible for a specific component of a platform (a vertical split), as we can see in [Figure 11-2](#).



*Figure 11-2. In this example, components teams are each responsible for a specific part of a platform*

With a vertical-split architecture, Teams Burrito and Enchiladas are responsible for the backend services, and Teams Nachos and Tacos are responsible for the frontend. This way of organizing teams works better when we are dealing with cross-platform applications in midsize to large organizations, where we usually develop at least a web application in conjunction with a native mobile application each for Android and iOS. In this instance, when the APIs are consumed by several client applications (mobile, web, and maybe other devices), a backend team responsible for creating an API takes into consideration all the needs of the consumer (frontend teams) instead of features teams optimizing the APIs for just one platform and treating the other clients' teams as second-class citizens. When we are working with cross-platform applications and hybrid technologies like Flutter, Ionic, or React Native, cross-platform teams are a more viable option than component teams. The codebase between frontend projects may be shared across targets, so organizing teams around features becomes the better choice. If you want to pursue the native option, however, think twice about how to organize your teams because switching from one native language to another and finally to web and backend languages is challenging and increases the cognitive load.

It's important to recognize that different stages of the business life cycle require different team structures. In the case of growth, you will encounter a moment where the organization requires some structural changes. These changes will lead you to reassess the teams around the architecture in order to reduce the friction for delivering a feature or any other stream of work. Invest time analyzing the communication flows and potential patterns established across teams, like constant external dependencies or slow stories throughput due to distributed teams in multiple time zones.

Domain-driven design (DDD) is helpful when we are organizing our team structure, because it helps to consider the direct connection between architecture and team structure. Not only does DDD help identify the boundaries between subdomains, but we can follow these boundaries for structuring our teams as well. For instance, with a small company, it's very

likely that a team would be responsible for a specific business subdomain end to end; however, within a larger organization, a multitude of teams create a subdomain, working together due to the work's inherent complexity and scope. It's not always possible to create the perfect structure, and often we need to make some trade-offs to create a model that almost fits everywhere. This is not necessarily a roadblock for your strategy, but do understand that trade-offs could happen. When trade-offs become a constant across the entire organization, we need to step back and review our team structure and architecture.

The structure of our teams—whether organized around features or components—and how we structure our micro-frontend applications will impact the communication flow inside the organization. There are certain practices that may help us achieve an efficient spread of information across teams, enhancing the governance for developing new features and capabilities inside our platform. Let's analyze some of them in the next section.

## **Implementing Governance for Easing the Communication Flows**

Working for midsize to large organizations means defining communication flows that work; otherwise, we risk slowing down development or creating too many external dependencies across teams. An investment worth making is governance. This is not just an upfront investment; it has to be a constant review and optimization of the practices and documents needed for scaling an organization. There are some simple wins for allowing our developers to scale their communication, especially for the future. We should remember that there will always be new employees at our company; the best way for them to fill their knowledge gaps would be to understand the context in which certain decisions were made. Without that context, they won't have enough information to fully understand the situation. Architecturally speaking, there are two practices that can spread the information and track why certain decisions were or were not made: the request for comments

(RFC) and the architecture decision record (ADR). Remember that asynchronous communication can greatly benefit introverts or those who don't perform well in meetings, allowing their voices and ideas to be heard. Don't underestimate the power of this approach—you might be surprised by the valuable insights from less vocal members of your organization.

## Requests for Comments

RFCs are an established way to gauge the interest in a change in a technical approach or a new technology or practice inside an organization. Usually RFCs are kicked off by developers or tech leads who see some gaps or potential improvement in the organization and want to understand if there is room for a change. RFCs are often available in the version control system (GitHub, for example), so every technical person has access to them. RFCs are timeboxed and are a short markdown document composed of the following sections:<sup>1</sup>

### *Feature name*

The name of the feature or practice to introduce or change

### *Summary*

One-paragraph explanation of the feature or practice

### *Motivation*

Usually the *why* of making a change

### *Description of the change*

Detailed analysis of the change or new feature

### *Drawbacks*

All the potential issues identified by the proposal submitter

### *Alternatives*

Potential alternatives to achieve the goals with pros and cons

*Unresolved questions*

Any blind spots in the proposal

*Additional resources*

A list of resources related to the RFC

After filing an RFC, the submitter shares the link with the interested parties inside the organization. Then the dialogue starts to flow, with people sharing ideas, asking questions, and trying to understand whether the proposal has any potential for being introduced inside the company. Despite its simplicity, this document is important in the present because it may improve current software development or practices. Moreover, the RFC has a fundamental benefit for the future as well. In fact, this document tracks the discussion happening among all the developers, architects, and tech leads, recording the full discussion and approaches. This history will give new employees a clear context that describes the reasons behind certain decisions. RFCs are great for proposing not only new features but also changes. For instance, when we need to update an API contract, using an RFC allows us to gather our consumers' thoughts, ideas, and concerns, allowing us to shape the best way to achieve the goal. This scenario often happens when we work with component teams, and the backend team comprises multiple teams that consume their APIs.

With an RFC, the team that owns the API contract can propose changes and collect the feedback from the other teams, gathering the evolution of the API and the reasons behind them. This practice becomes even more important when we work with distributed teams across multiple time zones, because we can share all the information needed without remote meetings, closing the feedback loop in a reasonable time.

## Architectural Decision Records

Another useful document for sharing decision context for current and future developers is the ADR, in which architects or tech leads gather the decisions behind a specific architecture implementation. ADRs are focused solely on architecture, but they are still useful for providing for future readers a context and a snapshot in time of your organization. In fact, an ADR specifically describes the company context when the ADR is first written. ADRs also differ from RFCs in that they provide the context of why an architecture change is needed and explain why a specific decision was made. Architecture is always about finding the balance between long-term and short-term wins. These trade-offs are defined in the company context we operate within. With ADR, we want to create a snapshot of the company context and provide a description about why we pick one direction over another. An ADR structure is composed of the following sections:

### *Status*

The status of an ADR (e.g., draft, agreed)

### *Stakeholders*

People behind the ADR, usually architects and tech leads

### *Outcome*

The final decision made

### *Due date*

The date for when the decision has to be made

### *Owners*

Document's owners

### *Introduction*

A paragraph describing the company context and the problem the ADR is trying to solve

### *Forces*

The parallel or overlapping streams of work that are pushing toward an architecture change

### *Options*

List of potential solutions with business and technical details and the pros and cons for every proposal

### *Final decision and rationale*

A summary of the final decision explaining the reasons behind choosing a proposal listed in the options paragraph

### *Appendix*

Additional resources needed for providing more context to the readers

As with RFCs, not all these parts are mandatory, but they are highly recommended. Remember, ADRs have to provide the context for everyone interested in why an architectural decision was made, so the reader needs to have clarity on the technical and business context when the ADR was created. When we design our micro-frontends, we may change the framework or design patterns implemented inside the architecture. By using ADRs, we can provide the context that existed before the architecture decision and why we now want to change it. This way, everyone will be on the same page, despite not being physically present in the meetings.

# **Techniques for Enhancing the Communication Flow**

When first approaching micro-frontends, many people think this architectural pattern may result in organizational silos due to its intrinsic characteristics, such as independency and decentralization. Although micro-frontends enable teams to work in parallel and release artifacts independently, there is no excuse for not creating a collaborative environment inside the tech department. We cannot embrace distributed systems without establishing mechanisms for teams to come together on a regular basis for sharing knowledge, solutions, and challenges. It's essential to curate the technical as well as the social aspect for guaranteeing the right flow of information inside an organization. Everything should start from the feature's specifications.

## **Working Backward**

Famous for its customer-centric approach, Amazon often works backward when considering product ideas. Simply put, they start with the customer and work backward to the product rather than starting with a product idea and bolting customers onto it. It's a method for creating a customer-focused vision of your product. A working-backward document, called a PR/FAQ, is up to six pages long: a one-page press release (PR) and up to five pages of frequently asked questions (FAQs). An appendix section is also included. While working backward can be applied to any specific product decision, using this approach is especially important when developing new products or features.

Because it starts with where you want to be 12 months in the future, the working-backward method forces you to think big, focusing on big goals and the changes you need to achieve. A well-crafted press release is a great use of storytelling. It gets the team excited and focused before any lines of code are written.

The FAQ section is composed of two subsections. The first one is based on the public questions a customer might have about the product or feature, written as if it is public product documentation that is released at the same time as the press release. The second subsection consists of questions internal stakeholders might have asked during the product development process.

The PR/FAQs focus effort on how a specific feature benefits the customer and why the company should invest in a product or feature like that in its system. After a PR/FAQs is written, a meeting is scheduled with the main stakeholders, including developers, QAs, tech leads, architects, and other product people. In general, though, any stakeholder who may help improve the decision process is invited. This may seem like overkill, but one hour of socializing requirements can allow techies to raise questions and become familiar with the feature. It's a first step for having multiple teams understand the initial requirements of a new feature and aligning it with the business goals to reach. When we work with micro-frontends, a PR/FAQs can bridge the teams that will collaborate in the implementation phase.

Two extremely valuable benefits of the PR/FAQs process are the resulting concise documentation and initial collaboration phase before the implementation phase. Usually a PR/FAQs document is a good starting point for architects to think about the high-level design, including the challenges and the architecture characteristics needed for implementing a feature.

This is also true for micro-frontends when a new requirement arises and it has to be implemented across multiple domains. Having this kind of document can facilitate the discussion between teams via the requirements socialization between engineers and product teams.

If you are interested in knowing more, I recommend reading Chris Vander Mey's **Shipping Greatness** (O'Reilly), a book that provides more information on how to write PR/FAQs, following Amazon learnings and suggestions. If you prefer a short document about PR/FAQs instead, check

out “[PR FAQs for Product Documents](#)”, a blog post by Robert (Munro) Monarch that offers a great summary of this topic.

## Community of Practice and Town Halls

The community of practice and town halls are two more important practices for facilitating the communication flows across the organization. With both cross-functional and components teams, there is a need to spread knowledge among developers of the same discipline (frontend developers in the micro-frontend world). Usually communities of practice are biweekly or monthly meetings scheduled by engineer managers or tech leads to facilitate discussions across team members responsible for the same discipline. In these meetings, the developers share best practices, how they have solved specific problems, new findings, or topics they’ve recently been exposed to inside their domain. Communities of practice are useful for introducing new practices across the organization, discussing automation pipelines improvements, or even hosting [mob programming](#) events, which has engineers collaboratively implementing a new feature or discussing a specific programming approach all together. While usually restricted to a team, I’ve experienced some mob programming sessions during a community of practice that have worked well.

### MOB PROGRAMMING

Mob programming is a software development approach in which a whole team works on the same project as a group, working on the same computer at the same time. This is similar to pair programming, in which two people work on the same code together at one computer. Mob programming just extends the collaboration to everyone on the team. This technique is typically used when a team is implementing an important but complex feature or during a community of practice where a vertical inside an organization wants to introduce new practices across the tech department.

Town halls are events organized across the tech department that provide a general knowledge of what's happening across teams, such as a team's recent achievements or new practices to introduce inside the organization. Town halls work especially well when an organization works with distributed teams and developers cannot engage with all the teams on a daily basis. During these events, the tech leadership facilitates the knowledge to be shared through short presentations covering the key initiatives brought up by different teams. Considering the large audience attending these events, any questions or deep dives should be taken up at a separate time by the interested people and the team or person involved in a given initiative.

Town halls are very useful when a team would like to share a new library they have developed that may be used by different teams, new practices introduced by a team and the results after embracing it, or more general topics like new joiners or shared goals across the department.

Depending on the company's size, town halls may not be the right choice for some companies. A good alternative for spreading these initiatives and communications could be an internal newsletter for the tech department. We may decide to split the newsletter by topics and allow developers to pick their favorite information, but this will depend on the organization's size and structure.

## Managing External Dependencies

Sometimes during a **sprint**, external dependencies may impede the delivery of a task or story. While this is not usually a problem, when distributed teams work on microarchitectures, it can slow down feature delivery, creating frustration and frictions across teams. When a team is hampered by too many external dependencies, it may be time to revisit our decisions and review the boundaries of a micro-frontend. Frequently occurring external dependencies is one of the strongest signs that something is not working as expected. But fear not: it's a fixable problem! As long as the information bubbles up from the teams to the tech leadership, leadership may decide to

rearrange the organization, reducing communication friction and improving the throughput.

With micro-frontends, this situation can occur when we share libraries across micro-frontends or when we compose multiple micro-frontends in the same view, if we don't pay enough attention to how to decouple them.

### NOTE

"Reusability represents a form of coupling!", as Neal Ford, Director of Architecture at ThoughtWorks

With a horizontal-split architecture, we need to invest time reviewing the communication flows, especially at the beginning of the project. A classic example is when we are porting a frontend project from a monolith, single-page application or from a server-side-rendering one to micro-frontends. When we embrace the horizontal-split architecture, we need to assign ownership of the view composition process. In fact, despite having multiple teams contributing with their micro-frontends to the final result, we need to identify the owner of the composition stage (either client or server side). This team should be responsible for not only composing the view but also understanding potential scenarios where a micro-frontend could cause other micro-frontends problems due to CSS style issues or events dispatched but not properly handled by other micro-frontends.

It's true that this architecture style provides more flexibility on reusing micro-frontends across a project, but at the same time, we need to make sure the final result is what the user expects to have. Another challenge in the communication flow for the horizontal split architecture may happen when a micro-frontend is reused in different views of one or more applications. In this case, the team responsible for the micro-frontend should create strong relationships with all the other teams that may asynchronously interact with the micro-frontend and have regular catchups to make sure the touching points within a view are respected.

The problem of too many external dependencies may happen in very limited cases with a vertical-split architecture, especially if we are using an application shell that orchestrates the micro-frontend life cycle. For example, let's say we want to add a new route inside our application. The application shell will need to be aware that a new micro-frontend needs to be loaded and that it will have to manage the new route. When well designed, the application shell loads an external configuration retrieved from a static file served by a content delivery network (CDN) or as a response of an endpoint. In this case, the effort to coordinate with the team that owns the application shell would be minimal because all that's required is changing the configuration, adding new automation tests, and following the testing life cycle implemented inside the organization.

Another potential activity slowdown can occur when you need to make changes across multiple micro-frontends. This usually happens once or twice a year; if it occurs more often, that's another sign we should review the division of our micro-frontends. However, with a vertical split, teams have more autonomy. If we are able to review the communication flows iteratively, we shouldn't have many surprises or external issues.

For all these situations, reviewing the communication flows with the right cadence (every quarter, for instance), and making sure the assumptions made during the architecture design process are still valid, are good practices. Remember that friction between teams should be seen as a signal of a problem within the organization, not necessarily as an architectural issue. Often, the root cause lies elsewhere, such as in organizational structure, micro-frontends boundaries, or a lack of practices to encourage collaboration within the developer community.

Another way to ensure the communication is flowing across teams is by having ad hoc meetings for the teams that have to work together for the final-view result, especially for a horizontal-split architecture. Using agile ceremonies like Scrum of Scrums or less informal catch-ups on a regular cadence result in a better understanding of the overall system, as well as better bounding between team members.

Finally, agile practices provide some tools that may be used either ad hoc or on a case-by-case basis to solve specific challenges we face with our teams. One tool that I have personally experienced in multiple companies I've worked for is big room planning, where we gather an entire department for one or two days inside a room and we map the activities for the next few months. In this way, we can immediately spot external dependencies and potential bottlenecks due to a wrong sequence of deliverables. There are many techniques available for solving specific challenges. I recommend first gaining an understanding of the problem you need to solve and then finding the right approach for that problem.

## A Decentralized Organization

A key advantage of working with micro-frontends specifically, and with microarchitectures in general, is the possibility of empowering the teams to own a business domain end to end. As we have seen throughout this book, micro-frontends are not for all organizations. They work well for midsize to large ones, where insight on the intrinsic complexity the company is working on is needed. Complexity is not necessarily a negative attribute; it can allow us to move from a centralized approach to a decentralized one.

Every company moves through different phases. In the startup phase, a company usually has a small tech team capable of working on a project end to end. The communication flow in the startup phase is straightforward because all the developers are aware of the goals to achieve, and the number of connections needed for a correct communication flow is manageable. When the startup grows larger, it's usually structured around business function hierarchies, introducing roles like head of engineering, engineer manager, tech leader, and many other well-known titles in the tech ecosystem. Usually these organization layers provide directions to coordinate teams, defining the communication flows. In an ideal scenario, the teams may have the level of autonomy needed to do their jobs and to experiment with new practices and technologies at the same time, but this

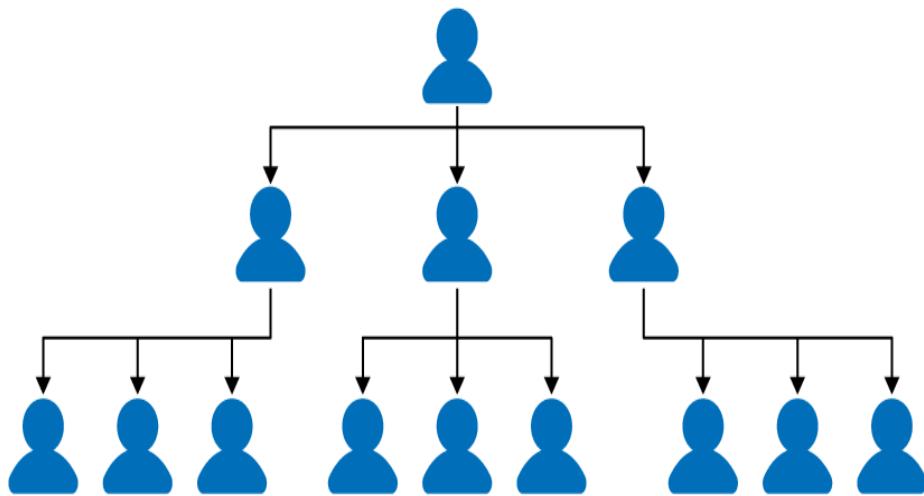
doesn't always happen. The reality will depend on the company culture, as well the leadership style of every individual.

When the organization moves from hundreds to thousands of employees, we need to look again at how to organize our teams. A natural evolution from the hierarchical structure would be aligning teams around value streams instead of following a centralizing hierarchy. Decentralizing the decision-making and allowing an independent path for the teams inside a value stream will empower the technical teams to express themselves in the best way in the large, complex context where they operate. These three types of structure are visually represented in [Figure 11-3](#).

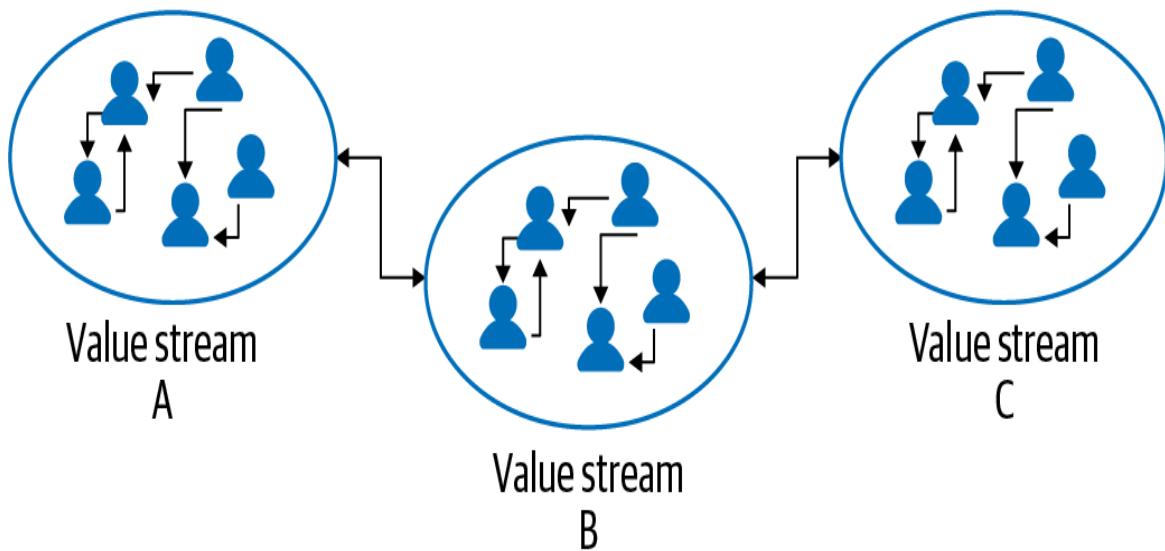
## Startup structure



## Hierarchical structure



## Decentralized structure



*Figure 11-3. Different organizational structures are usually implemented in different stages of a company life cycle*

An interesting point highlighted in [Figure 11-3](#) is that, with the decentralized structure, teams must coordinate among themselves when needed. Empowering these teams means giving not only technical freedom but also organizational duties. Technical leaders then become a support function that should facilitate the streams of work inside the teams, providing context or technical direction when a team requests it and driving and aligning the technical boundaries with the business results so that the team knows how to achieve these goals.

Another great achievement of decentralization is error mitigation. In this structure, it's unlikely that only one or a few people are capable of making every decision for every team, especially because the leadership is not always involved in the day-to-day conversations. Therefore, the role of an architect or a tech lead would include posing the right questions and becoming a servant-leader for the team, creating solutions hand in hand with the team rather than in isolation.

## ORGANIZE FOR COMPLEXITY

Many of the concepts of decentralization that I've described are part of a great book called *Organize for Complexity* by Niels Pflaeging (BetaCodex Publishing). This short, straightforward book provides many insights on how to decentralize an organization handling complexity. I was lucky enough to meet Niels during an agile retreat and received a complimentary copy of his book. It changed the way I thought about tech organizations, opening several doors in my mind. The book doesn't focus on tech organizations but more generally on any organization. That's the reason I shared these insights, contextualizing the core concepts in the tech context. I found these concepts extremely valid for microarchitectures.

Since 2019, the DDD community has increasingly emphasized the social-technical aspect of software architecture, drawing numerous insights from the remarkable book *Team Topologies* by Manuel Pais and Matthew Skelton. This book is eye-opening and a must-read for any tech leader seeking to organize their organization for fast flow and distributed systems.

## Decentralization Implications with Micro-Frontends

The first step in decentralizing decision making and empowering the teams that are closer to the business domain is identifying the subdomains available in an application. As we have seen so far, DDD helps us identify the business subdomains where we create a common language (ubiquitous language), we introduce patterns for communicating across subdomains for decoupling them, and we allow them to evolve at their own pace. Another important aspect to take into consideration is user behavior, especially when we are porting an existing application to micro-frontends. These two metrics allow us to identify the different pieces of a complex puzzle and assign every piece to a specific team.

I highly recommend basing the micro-frontends split on data, because this can really save you a lot of aggravation in the long run. Starting with incorrect assumptions creates friction across teams and release cycles. Data can help prevent those incorrect assumptions.

Another important thing to consider is balancing complexity when we assign a team to a subdomain. When a team is assigned to multiple complex subdomains, there is a high risk of resource burnout and intrinsic maintenance complexity. There are several situations we need to be aware of:

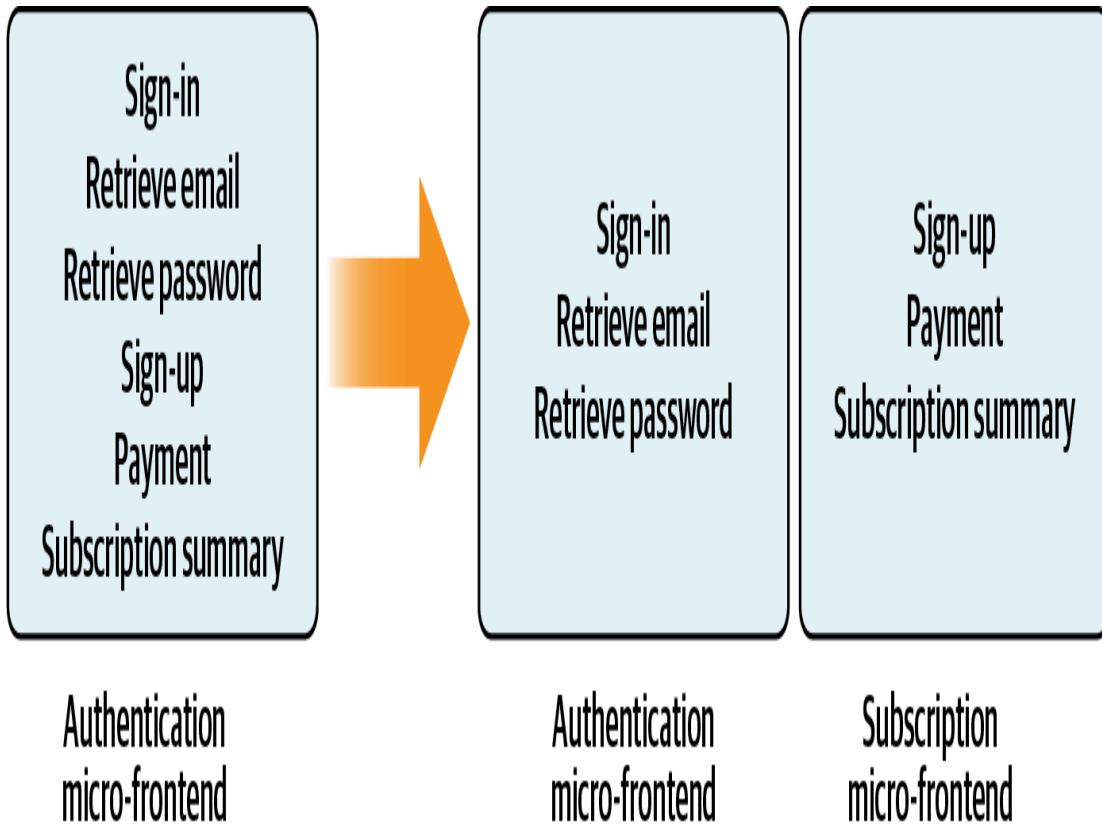
#### *High-complexity subdomain*

This type of subdomain usually doesn't manifest at the beginning of the project. They're created when we underestimate a subdomain's complexity in terms of the business logic and permutations that a micro-frontend needs. Usually a micro-frontend becomes complex over time because new features are added to it. A good practice, therefore, is to understand the cognitive load of every team member working on that subdomain and determine whether they can handle the situation properly. Are there enough team members equipped to handle the demands of the subdomain? Do they possess the requisite knowledge to implement all requested features effectively? Should we further decompose or consolidate micro-frontends? Are there any sources of friction that need addressing? These are excellent starting questions.

The main struggle is often in maintenance and support, especially when the team deals with projects that have to be live 24/7 and where bugs have to be fixed as quickly as possible. High complexity is difficult to handle in general, and it's even more difficult when a developer is under pressure because a live bug is found in the middle of the night and requires a quick fix.

In these cases, remember to review the boundaries of your subdomain and see if it can be split in a more sensible manner, especially when you work with a vertical-split architecture. Consider, for instance, when you have an authentication micro-frontend containing the sign-in and sign-

up flows in a vertical-split architecture. If you are working on a global platform, you may have to support multiple payment methods, which adds a lot of information to remember and own. Splitting the authentication micro-frontends into sign-in and sign-up micro-frontends would maintain a frictionless user experience while reducing the cognitive load on the team, as described in [Figure 11-4](#).



*Figure 11-4. Another option is to split the micro-frontend to reduce the team's cognitive load without impacting the user's experience*

Now we have a team dedicated to new users who want to sign up and another dedicated to existing users who have to sign in or retrieve their email or password. In this way, we simplify the logic and code and can have quicker fixes when bugs are discovered.

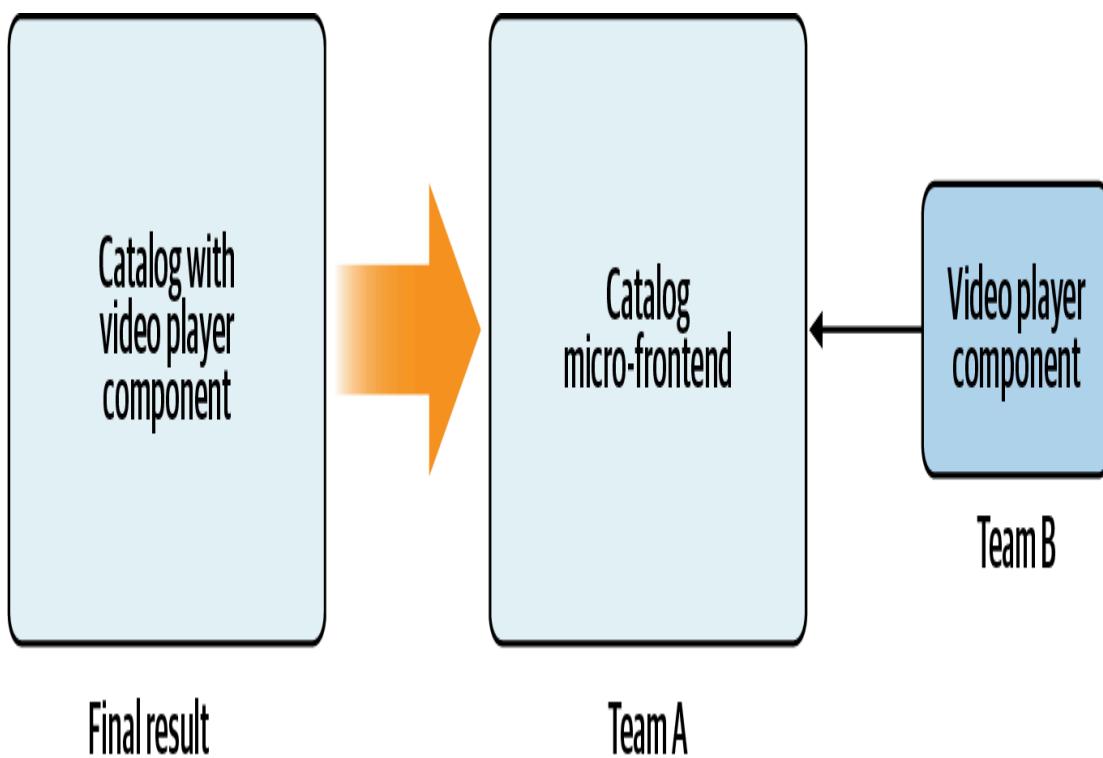
### *High initial-effort subdomain*

Sometimes we may have micro-frontends that require a high effort to create them, but then they don't evolve very often in the application life

cycle. In this case, we can afford to have a team with multiple micro-frontends, bearing in mind we need to balance the micro-frontends' complexity to avoid drowning the team in work.

#### *Normal-complexity subdomain*

When considering a single team, these are the subdomains we should aim for. Sometimes when we have high-complexity subdomains, we may decide that splitting the micro-frontend representing that subdomain would not help much. However, we can componentize a specific complex part of the micro-frontend and assign the component to another team, as we can see in [Figure 11-5](#).



*Figure 11-5. This example shows a project with high complexity but the possibility of extracting a component to another team to spread the cognitive load*

In this case, we have a vertical-split architecture with a micro-frontend with multiple views and a complex business logic to implement for the video player that should play video, advertising, and so on. The video player can be peeled off from the micro-frontend. This self-contained component may be reused in other micro-frontends and has an intrinsic

complexity, making it a good choice to be handled by a different team, reducing the cognitive load of both the micro-frontend and the video player. The teams can collaborate when new versions of the video players version are available. In this case, working with an API contract and scheduling regular touch points between teams is sufficient to coordinate the integration, new releases, and breaking changes.

### *Low-complexity subdomain*

Finally, some micro-frontends are easy to build and maintain so one team can own a multitude of them. As with the previous cases, make sure to regularly rebalance the complexity assigned to these teams because, while the complexity may be low in this case, having dozens of low-complexity micro-frontends may cause high-context switching, reducing the team's productivity.

Decentralizing decision-making and empowering teams doesn't mean we need to create chaos inside the organization. In fact, some decisions should remain centralized and made by the teams across all of an organization's domains, such as a platform or developer experience team, or by tech leadership, like head or vice president of engineering, providing a framework for the teams to operate with. We mentioned such decisions in previous chapters, like the platform to run the automation strategy; programming languages or frameworks available for the teams to use guidelines on when to abstract and when to duplicate code; and architecture characteristics, such as performance metrics, code coverage and complexity, setting up the observability of the entire platform (frontend and backend), and support governance when the application fails in production. All these decisions provide a concrete framework for the teams to operate on. They don't affect the teams' freedom, and they align the company behind some guidelines that should allow your technical teams to achieve business goals.

## Summary

In this chapter, we learned that we cannot design a software architecture without taking into consideration the human factor. Architecture, company culture, and organizational structure are interdependent factors crucial for the success of any project. We need to be aware that these two forces are part of a project's success and they cannot be decoupled. They must be looked at together and revised often. Any business can evolve over time, and the same is true for software architecture and communication flows inside a company. The communication flow can be enhanced by spreading information across the teams, but it has to be thought through and designed carefully, and we need to iterate to find the right balance. What works in one company will not necessarily work in others, so carefully analyze your context and apply the best practices for your organization.

Finally, we looked at how decentralization helps the implementation of any microarchitecture, whether microservices or micro-frontends. It's important to highlight that the micro-frontends architecture we chose should influence the way we structure our teams. It's very unlikely that when we move from a monolith architecture to a microarchitecture the organization can remain the same. The communication flow changes with the new architecture, and we need to at least review the flow so that we don't create bottlenecks inside the teams due to the wrong setup.

---

<sup>1</sup> This list is just a suggestion. Not all of the items may be present in your RFC template, but it's a good starting point.