O'REILLY®

Building Micro-Frontends

Scaling Teams and Projects
Empowering Developers

Early Release
RAW & UNEDITED

Luca Mezzalira

Building Micro-Frontends

Scaling Frontend Projects and Teams

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

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

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*.

Acquisitions Editor: Jennifer Pollock

Development Editor: Angela Rufino

Production Editor: Christopher Faucher

Interior Designer: David Futato

Cover Designer: Karen Montgomery

October 2021: First Edition

Revision History for the Early Release

- 2020-02-20: First Release
- 2020-06-10: Second Release
- 2020-08-14: Third Release
- 2021-01-25: Fourth Release
- 2021-05-11: Fifth Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492082996 for release details.

978-1-492-08292-7

Preface

At the beginning of December 2016, I was taking my first trip to Tokyo. It should have lasted just a week, but as I would discover, I would need to travel to the Japanese capital many more times in the following weeks.

I clearly remember walking to the airplane and mentally preparing my to-do list for the impending 12-hour flight. By now I'd been travelling for a couple of weeks, on the opposite side of the world: attending a couple of conferences in the Bay area and then another event in Las Vegas.

The project I was working at that time was an over-the-top platform similar to Netflix, but dedicated to sports, with daily live and on-demand content available in multiple countries and on more than 30 different devices (web, mobile, consoles, set-top-boxes and smartTVs).

It was near the end of the year, and as a Software Architect I had to make a proposal for a new architecture that would allow the company to scale to hundreds of developers distributed in different locations, without reducing the current throughput and enhancing it as much as I could.

When I was setted in my seat, I became relatively calm. I was still tired from the Vegas trip, and a bit annoyed at the 12 hours I would have to spend on the airplane, but excited to see a mystical country like Japan for the first time.
A few minutes later I had my first glass of Champagne. For the first time in my life, I was in business class, with a very comfortable seat and a lots of space for working.
Definitely the best place to start such a long journey.

At the end of the first hour, it was time to get my laptop out of my backpack and start working on "the big plan", I still had more than 10 hours of flight time during which I could start working on this huge project that would serve millions of customers around the world. I didn't know at that time that

the following hours would deeply change the way I would architect cross-platform applications for frontend.

In this book I want to share my journey into the *Micro-Frontends* world, all the lessons and tips for getting a solid *Micro-Frontends* architecture up and running, finally the *PROs and CONs* of this approach.
These lessons will allow you to evaluate if this architecture would fit your current or next project.
So now it's time for your journey to begin.

The Frontend Landscape

I remember a time when web applications were called RIAs or Rich Internet Applications, to differentiate them from the traditional, and more static, corporate websites. Nowadays we can find many "RIAs", or simply web applications, across the World Wide Web.

Now, many SaaS (Software as a Service) with more or less complex UIs allow us to print our business cards on demand, watch our favorite movies or live events, order a few pepperoni pizzas for us and our guests, manage our bank accounts from our comfortable sofas, and do many, many other things that make our life easier on a daily basis.

As CTOs, architects, tech leads, or simply developers, when we start a greenfield project, we can create a single-page application or an isomorphic one (also called a *universal application*), or instead work with a bunch of static pages that run in our cloud or on-premise infrastructure.

Yet despite this broad range of opportunities in front of us, not all of them are fitting for any project.
Instead, we need to understand first what challenges we are going to face before taking a direction.

But before jumping straight to the topic of this book, let's analyze the current architectures available to us when we work on a frontend application.

Single-Page Applications

Single-page applications (or SPAs) are probably the most used implementations by many companies. They consist of a single or a small number of JavaScript files that encapsulate the entire frontend application usually downloaded up front.

When the web servers or the Content Delivery Network (CDN) serve the HTML index page, the SPA loads the JavaScript, CSS, and any additional files needed for displaying any part of our application.

There are many benefits of using SPAs; for instance, the client downloads the application code once at the beginning of its life cycle and the entire logic is available up front.
Also SPAs usually communicate with APIs for exchanging data with the persistent layer of the backend.

An SPA avoids multiple round trips to the server for loading additional application logic and renders all the views instantaneously during the application life cycle.
Both features enhance the user experience and simulate what we usually have when we interact with a native application for mobile devices or desktop, where we can jump from one part of our application to another without waiting too long.

In addition, an SPA fully manages the routing mechanism on the client side. Usually every time the application changes a view, it rewrites the URL in a meaningful way to allow users to share the page link or bookmark the URL for starting the navigation from a specific page. SPAs also allow us to decide how we are going to split the application logic between server and client. We can have a “fat client” and a “thin server” where the logic is mainly stored on the client side and the server side is used as persistence layer, or a “thin client” and a “fat server” where the logic is mainly delegated to the backend, and the client doesn’t perform any smart logic but just reacts to the state provided by the APIs.

Over the past of several decades, different school of thoughts have prevailed on whatever far or thin clients are better solutions.
Despite these arguments altough, both approaches have their own PROs and CONs. It always depends on the type of application we are creating.
I personally found it very valuable to have a thin client and a fat server when I was targeting cross-platform applications, so I was able to design a feature once and have all clients deployed on multiple targets react to the application state stored on the server.
At the same time, I often used a fat client and a thin server when I had to create desktop applications where the offline persistence layer was an essential feature; therefore, rather than writing the state logic twice, I managed it only in one place and used the server for data synchronization.

However, SPAs have some disadvantages for a certain type of applications. Technically speaking, the first time to load usually takes longer than for other architectures, because we are downloading the entire application instead of only what the user needs to see
If not well designed, this could become a killer for our applications, especially, when they are loaded with an unstable or unreliable connection on mobile devices like smartphones and tablets.

Nowadays there are several ways to mitigate this problem caching the content directly on the client. In particular Progressive Web Apps are providing a set of new techniques, based on service workers, a script that your browser runs in the background, separate from a web page, for enhancing the user experience of an SPA serves on mobile devices with flaky or totally absent connections. Another potential issue caused by SPAs is that are not SEO-friendly; in fact, when a crawler is trying to understand how to navigate the application or website, it won't have an easy job indexing all the contents served by an SPA unless we prepare alternative ways for fetching it.

Usually when we want to provide better indexing for an SPA, we tend to create a custom experience strictly for the crawler.
For instance, Netflix lowers its geo-fencing mechanism when the user-agent requesting their web application is identified as a crawler, instead serving

content similar to what a user would watch based on the country specified in the URL. This is a very handy mechanism considering that the crawlers engine is often based in a single location from which it indexes a website all over the world.
As mentioned earlier, SPAs download all the application logic in one go, but this also can lead to potential memory leaks when the user is jumping from one view to another if the code is not well implemented and is not correctly disposing of the unused objects.

In large applications this could be a real problem that might lead to several days or weeks of application hardening in order to make the SPA code functional.
It could be even worse if the device that loads the SPA doesn't have a great hardware, like a smart TV or a set-top-box.
Too often I have seen applications running smoothly on a MacBook Pro quad-core and then failing miserably when run on a low end device.

The last disadvantage to mention when we work with SPAs is on the organizational side.
When an SPA is a large application managed by distributed or colocated teams working on the same codebase, different areas of the same application could end up with a mix of approaches and decisions that could confuse team members.
Also the communication overhead teams will use to coordinate between themselves is often a hidden cost of the application.

Many times we completely forget about calculating the inefficiency of our teams, not because they are not capable of developing an application, but because the company structure or architecture doesn't enable them to express in the best way possible, slowing down the operations and the implementation of any new feature.

Isomorphic Applications

Isomorphic applications, or universal applications, are web applications where the code between server and client is shared and can run in both

contexts.

This technique brings some benefits when used in the right way. It is in particular is convenient when the time to interaction, A/B testing, and SEO are essential characteristics for the application.

Isomorphic applications can be designed in different ways.
Because these web applications share code between server and client, the server, for instance, can do the rendering part for the page requested by the browser, retrieve the data to display from the database or from one or multiple APIs, aggregate it together, and then pre-render it with the template system used for generating the view, in order to serve to the client a page that doesn't need additional round trips for requesting data to display.

This will definitely enhance the time to interaction, considering the page requested is pre-rendered on the server and is partially or fully interpreted on the backend. This spares a lot of roundtrips on the frontend, so we won't need to load additional resources (vendors, application code, etc.) and the browser will interpret a static page with almost everything inside.

Also, as mentioned, using isomorphic applications can improve an SEO strategy, because the page is server-side rendered without the need for additional server requests when served.
The crawler is going to request the page and receives it fully prepared on the server, ready to be indexed by the search engine without any problem.
Isomorphic applications share the code between contexts, but how much code is really shared? To answer this question *depends on the context*.

For instance, we can use this technique in a hybrid approach where we server-side render part of the page in order to benefit from the time to interact and then lazy loading additional JavaScript files for having the benefits of the isomorphic application as well as the single page application where the files loaded within the HTML page served allow to add sophisticated behaviours to a static web page, transforming this page in a SPA. We can also have a more purist approach where we always render the

page and its state on the server providing only a page to display for the browser, really depends on the complexity of the project we are facing.

With this approach, we can decide how much code is shared on the backend based on the project's requirements.
For instance, we can render just the views, inlining the CSS and the bare minimum JavaScript code to have an HTML skeleton that is loaded very quicly by the browser, or completely delegate the rendering and data integration onto the server, pheraps because we have more static pages than heavy interactivity on the client side. We can also have a mixed approach where we divide the application into multiple SPAs with the first view rendered on the server side, and then some additional JavaScript to download for managing the application behaviors, the models and also the routing inside the SPA.

Routing is another interesting part of an isomorphic application because we can decide to manage it on the server-side, only serving a static page every time the user is interacting with a link on the client.
Or we can have a mixed approach if we use the benefits of server side rendering only for the first view, then load an SPA, where in this case the server will do a macro routing that serves different SPAs, each SPA has its own routing system for navigating between views. Another benefit of this approach is that we are not limited only to template libraries, but virtual DOM implementations like React, Preact, or similar, also can benefit from this approach. Many other libraries and frameworks have started to offer the server-side rendering approach out of the box like Vue with Nuxt.js, Meteor, or Angular, for instance. Also, we don't need to have a Node.js backend for working with isomorphic apps but we can use the technology we are more familiar with for serving our APIs like Go or PHP for instance. Isomorphic applications therefore won't have much of an impact on your existing backend technology stack.

The last thing to mention about isomorphic applications is that we can integrate A/B testing platforms nicely without much effort. In the past year or so, many A/B testing platforms had to catch up with the frontend technologies in not only supporting UI libraries like JQuery but also in

embracing virtual DOM libraries like React or Vue. They also have to make their platforms ready for hybrid mobile applications as well as native ones.

The strategy these companies adopted is to manage the experiments on the server side where the developers have full control of the experiments to run on the clients. This is obviously a great advantage if you are working with an isomorphic application because you can pre-render on the server the specific experiment you want to serve to a specific user. Those solutions can also communicate with the clients via APIs in the case of native mobile applications and SPAs for choosing the right experiment.

Isomorphic applications could also suffer from scalability problems if a project is really successful and visited by millions of users. Considering we are generating the HTML page pre-rendered on the server, it means we need to create the right caching strategy in order to minimize the impact on the servers.
In this case, if the responses are highly cacheable, CDNs like Akamai, Fastly, or Cloudfront could definitely improve the scalability of our isomorphic applications by avoiding all the requests hitting origin servers.

Organisation wise, an isomorphic application suffers of similar problems we can find on an SPA where the code base is unique and maintained by one or multiple teams.

There are ways to mitigate the communication overhead if a team is working on a specific area of the application without any overlap with other teams. In this case we can use architecture like Backend for Frontends (BFF) for decoupling the API implementation and allow each team to maintain their own layer of APIs specific to a target.

Static-Page Websites

Before entering the main topic of this book, it is worth mentioning the static-page website, where every time the user clicks on a link we are loading a new static page.
Fairly “old school”, I know, but still in use with some twists.

Usually this approach is useful for quick websites that are not meant to be online for a long period of time, such as ones where freelancers can promote their skills or simply for advertising a specific product or service we want to highlight without using the corporate website.

In the last few years this type of websites has mutated into a single page that expands vertically instead of loading different pages. Another trend in this last case is to lazy-load the content of the website, waiting till the user scrolls to a specific position of the website and then loading the content.

The same technique is used with hyperlinks, where all the links are anchors inside the same page and the user is browsing quickly between bits of information available on the website. These kinds of projects are usually created by small teams or individual contributors; the investment for the company is fairly low on the technical side, and it's a good playground for developers to experiment with new technologies and new practices or to consolidate existing ones.

Micro-Frontends

Micro-Frontends are an emerging architecture, clearly inspired by microservices architecture.
The main idea behind them is to break down *"the monolith"* into smaller parts, allowing an organisation to spread the amount of work with autonomous teams, colocated or distributed, without the need to slowing down their delivery throughput.

As we know, encapsulating an API inside a microservice is actually the easiest part.
When we realize there is way more to take care of, we will understand the complexity of the microservices architecture that not only adds high flexibility and good encapsulation between domains, but also an overall complexity around the observability, automation, and discoverability of a system.

Micro-Frontends probably are not suitable for all projects; nevertheless, they can alert us to a new way to structure our frontend applications, solving some of the key challenges we have encountered in the past not only from a technical perspective but also from an organizational one.

Too often I have seen great architectures on paper that didn't translate well in the real world because the creator didn't take into account the environment (company's structure, culture, developers skills, timeline, etc.) where the project should have been built.
As Conway's law, mentioned in many books, claims:

> "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."

The Conway's law could be mitigated with the "Inverse Conway Maneuver," which recommends that teams and organizations be structured on our desired architecture and not vice versa.

I truly believe that mastering different architectures and investing time on understanding how many systems work allows us to mitigate the impact of Conway's law, because it gives us enough tools in our belt to solve different challenges, not only technical ones but organizational too.

> Micro-Frontends in combination with microservices and a strong engineering culture, where everyone is responsible for their own domain and the final result as well, could result in a real silver bullet - complex to build but foolproof when used.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

> **WARNING**
>
> This element indicates a warning or caution.

O’Reilly Online Learning

> **NOTE**
>
> For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

First of all, I'd like to thank my family, my girlfriend Maela, and my daughter Emma for everything we share and the strength I receive from them to move forward every single day. Thanks to all the people who inspire me on a daily basis in any shape or form of communication.

A huge thank to DAZN, who allowed me to apply a *Micro-Frontends* architecture and to explore the benefits of it end to end trusting my ideas and judgement.
I also thank all my incredible colleagues who challenged and helped me on delivering our platform on more than 30 different targets. Last but not least, thanks to O'Reilly for the opportunity to write about *Micro-Frontends*, in particular to Jennifer Pollock and Angela Rufino for all the support I had during this journey.

Chapter 1. The Frontend Landscape

> **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 1st chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *Building.microfrontends@gmail.com*.

I remember a time when web applications were called rich internet applications (RIAs) to differentiate them from the traditional, and more static, corporate websites. Nowadays we can find many RIAs, or web applications, across the World Wide Web.

Today, a lot of software as a service (SaaS) with more or less complex user interfaces (UIs) allows us to print our business cards on-demand, watch our favorite movies or live events, order a few pepperoni pizzas for us and our guests, manage our bank accounts from our comfortable sofas, and do many, many other things that make our lives easier.

As CTOs, architects, tech leads, or developers, when we start a greenfield project, we can create a single-page application or an isomorphic one, whose code can run both in the server and the client, or even work with a bunch of static pages that run in our cloud or on-premise infrastructure.

While we now have such a broad range of options, not all of them are fit for every job. To make the right decision for our projects, we need to understand the challenges we will face along the way.

Before we jump into the topic of this book, let's analyze the current architectures available to us when we work on a frontend application.

Micro-Frontends Applications

Micro-frontends are an emerging architecture inspired by microservices architecture.

The main idea behind them is to break down a monolithic codebase into smaller parts, allowing an organization to spread out the work among autonomous teams, whether collocated or distributed, without the need to slow down their delivery throughput.

However, thinking about parallelism in the backend world, designing an application program interface (API) and encapsulating the logic into a microservice is actually the easiest part. When we realize there is significantly more to take care of, we will understand the complexity of the microservices architecture that adds not only high flexibility and good encapsulation between domains but also an overall complexity around the observability, automation, and discoverability of a system.

For instance, after creating the business logic of a service, we need to understand how a client should access our API, if it's an internal microservice that should communicate with other microservices we need to identify a security model.

Then we need to deal with the traffic that consumes our microservice, implementing techniques for spike traffic like autoscaling or caching for instance.

We also need to understand how our microservice may fail, we may fail gracefully without affecting the consumers and just hiding the functionality

on the user interface, otherwise we need to have resilience across multiple availability zones or regions.

Working with microservices simplify the business logic to handle, however we need to handle an intrinsic complexity at different levels like networking, persistence layer, communication protocols, security and many others.

This is also true for micro-frontends, if the business logic and the code complexity are reduced drastically, the overhead on automation, governance, observability and communication have to be taken into consideration.

As with other architectures, micro-frontends are not suitable for all projects; nevertheless, they can provide a new way to structure our frontend applications, solving some key scalability challenges we have encountered in the past not only from a technical perspective but also from an organizational one.

Too often I have seen great architectures on paper that didn't translate well into the real world because the creator didn't take into account the environment (company's structure, culture, developers skills, timeline, etc.) where the project would have been built.

Melvin Conway's Law said it best:

"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."

Conway's Law could be mitigated with the Inverse Conway Maneuver, which recommends that teams and organizations be structured according to our desired architecture and not vice versa.

I truly believe that mastering different architectures and investing time on understanding how many systems work allow us to mitigate the impact of Conway's Law, because it gives us enough tools in our belt to solve both technical and organizational challenges.

Micro-frontends, combined with microservices and a strong engineering culture, where everyone is responsible for their own domain, may result in a real silver bullet—complex to build but foolproof when used.

This architecture can be used in combination with other backend architecture such as a monolith backend or service oriented architecture (SOA), although micro-frontends suit very well when we can have a microservices architecture, allowing us to define slices of an application that are evolving together.

In this book, we are going to explore the possibilities provided by this architectural style and how to design a coherent architecture for our applications.

Single-Page Applications

Single-page applications (SPAs) are probably the most used implementations. They consist of a single or a few JavaScript files that encapsulate the entire frontend application, usually downloaded upfront.

When the web servers or the content delivery network (CDN) serves the HTML index page, the SPA loads the JavaScript, CSS, and any additional files needed for displaying any part of our application.

Using SPAs has many benefits. For instance, the client downloads the application code just once, at the beginning of its lifecycle, and the entire application logic is then available upfront for the entire user's session.

SPAs usually communicate with APIs by exchanging data with the persistent layer of the backend also known as the server side. They also avoid multiple round trips to the server for loading additional application logic and render all the views instantaneously during the application life cycle.

Both features enhance the user experience and simulate what we usually have when we interact with a native application for mobile devices or

desktop, where we can jump from one part of our application to another without waiting too long.

In addition, an SPA fully manages the routing mechanism on the client side.

What this means is, every time the application changes a view, it rewrites the URL in a meaningful way to allow users to share the page link or bookmark the URL for starting the navigation from a specific page. SPAs also allow us to decide how we are going to split the application logic between server and client. We can have a “fat client” and a “thin server,” where the client side mainly stores the logic and the server side is used as persistence layer, or we can have a “thin client” and a “fat server,” where the logic is mainly delegated to the backend and the client doesn’t perform any smart logic but just reacts to the state provided by the APIs.

Over the past several decades, different schools of thought have prevailed on whether fat or thin clients are a better solution.

Despite these arguments, however, both approaches have their pros and cons. The best choice always depends on the type of application we are creating.

For example, I found it very valuable to have a thin client and a fat server when I was targeting cross-platform applications. It allowed me to design a feature once and have all the clients deployed on multiple targets react to the application state stored on the server.

When I had to create desktop applications where storing some data offline was an essential feature, I often used a fat client and a thin server instead. Rather than managing the state logic in two places, I managed it in one and used the server for data synchronization.

However, SPAs have some disadvantages for certain types of applications. The first load time is usually longer than other architectures because we are downloading the entire application instead of only what the user needs to see. If the application isn’t well designed, the download time could become a killer for our applications, especially when they are loaded with an

unstable or unreliable connection on mobile devices, like smartphones and tablets.

Nowadays we can cache the content directly on the client in several ways to mitigate the problem. A technique worth a mention is for sure the progressive web apps.

Progressive web apps provide a set of new possibilities based on service workers, a script that your browser runs in the background separate from a web page for enhancing the user experience of a client application served on desktop and mobile devices with flaky or totally absent connections.

Thanks to service workers we can now create our caching strategy for a web application, with native APIs available inside the browsers.

This pattern is called offline first, or cache first, and it's the most popular strategy for serving content to the user. If a resource is cached and available offline, return it first before trying to download it from the server. If it isn't in the cache already, download it and cache it for future usage. As simple like that but very powerful for enhancing the user experience in our web application, especially on mobile devices.

Another disadvantage relates to search engine optimization (SEO). When a crawler, a program that systematically browses the World Wide Web in order to create an index of data, is trying to understand how to navigate the application or website, it won't have an easy job indexing all the contents served by an SPA unless we prepare alternative ways for fetching it.

Usually, when we want to provide better indexing for an SPA, we tend to create a custom experience strictly for the crawler.

For instance, Netflix lowers its geofencing mechanism when the user-agent requesting its web application is identified as a crawler rather than serving content similar to what a user would watch based on the country specified in the URL. This is a very handy mechanism considering that the crawler's engine is often based in a single location from which it indexes a website all over the world.

Downloading all the application logic in one go can be a disadvantage as well because it can lead to potential memory leaks when the user is jumping from one view to another if the code is not well implemented and does not correctly dispose of the unused objects. This could be a serious problem in large applications, leading to several days or weeks of code refactoring and improvements in order to make the SPA code functional.

It could be even worse if the device that loads the SPA doesn't have great hardware, like a smart TV or a set-top box. Too often I have seen applications run smoothly on a MacBook Pro quad-core and then fail miserably when running on a low-end device.

SPAs last disadvantage is on the organizational side. When an SPA is a large application managed by distributed or colocated teams working on the same codebase, different areas of the same application could end up with a mix of approaches and decisions that could confuse team members. The communication that overhead teams use to coordinate between themselves is often a hidden cost of the application.

We often completely forget about calculating the inefficiency of our teams, not because they are not capable of developing an application but because the company structure or architecture doesn't enable them to express it in the best way possible, slowing down the operations, creating external dependencies, and overall generating friction during the development of a new feature.

Also, the developers may feel a lack of ownership considering many key decisions may not come from them considering the codebase of a large SPA may be started months if not years before they join the company.

All of these situations are not presented in form of an invoice at the end of the months but they impact on the teams throughput considering that a complex codebase may slow down drastically the teams potential of delivery.

Isomorphic Applications

Isomorphic or universal applications are web applications where the code between server and client is shared and can run in both contexts.

This technique brings some benefits when used correctly. It is particularly beneficial for the time to interaction, A/B testing, and SEO, for instance, thanks to the possibility to generate the page on the server-side, a crawler can index the final page faster which leads us to have full control on how fast Isomorphic applications can be designed in different ways.

These web applications share code between server and client, allowing the server, for instance, to render the page requested by the browser, retrieve the data to display from the database or from one or multiple APIs, aggregate it together, and then pre-render it with the template system used for generating the view in order to serve to the client a page that doesn't need additional round trips for requesting data to display.

Because the page requested is pre-rendered on the server and is partially or fully interpreted on the backend, the time to interaction is enhanced. This avoids a lot of round trips on the frontend, so we won't need to load additional resources (vendors, application code, etc.) and the browser can interpret a static page with almost everything inside.

An SEO strategy can also be improved with isomorphic applications because the page is server-side rendered without the need for additional server requests. When served, it provides the crawler an HTML page with all the information inside ready to be indexed immediately without additional round trips to the server.

Isomorphic applications share the code between contexts, but how much code is really shared? The answer depends on the context.

For instance, we can use this technique in a hybrid approach, where we render part of the page on the server side to improve the time to interact and then lazy-load additional JavaScript files for the benefits of both the isomorphic application and the SPA. The files loaded within the HTML page served will add sophisticated behaviors to a static web page, transforming this page into an SPA.

With this approach, we can decide how much code is shared on the backend based on the project's requirements.

For example, we can render just the views, inlining the CSS and the bare minimum JavaScript code to have an HTML skeleton that the browser can load very quickly, or we can completely delegate the rendering and data integration onto the server, perhaps because we have more static pages than heavy interactivity on the client side. We can also have a mixed approach, where we divide the application into multiple SPAs with the first view rendered on the server side and then some additional JavaScript downloaded for managing the application behaviors, models, and routing inside the SPA.

Routing is another interesting part of an isomorphic application because we can decide to manage it on the server side, only serving a static page any time the user interacts with a link on the client.

Or we can have a mixed approach. We can use the benefits of server-side rendering for the first view, and then load an SPA, where the server will do a macro routing that serves different SPAs, each with its own routing system for navigating between views. With this approach we aren't limited to template libraries; we can use virtual document object model (DOM) implementations like React or Preact. Many other libraries and frameworks have started to offer server-side rendering out of the box, like Vue with Nuxt.js, Meteor, and Angular.

As you can see, isomorphic applications won't have much of an impact on your existing backend technology stack.

The last thing to mention about isomorphic applications is that we can integrate A/B testing platforms nicely without much effort.

A/B testing is the act of running a simultaneous experiment between two or more variants of a page to see which one performs the best.

In the past year or so, many A/B testing platforms had to catch up with the frontend technologies in not only supporting UI libraries like JQuery but also embracing virtual DOM libraries like React or Vue. Additionally, they

have to make their platforms ready for hybrid mobile applications, as well as native ones.

The strategy these companies adopted is to manage the experiments on the server side where the developers have full control of the experiments to run on the clients. This is obviously a great advantage if you are working with an isomorphic application because you can pre-render on the server the specific experiment you want to serve to a specific user. Those solutions can also communicate with the clients via APIs with native mobile applications and SPAs for choosing the right experiment.

But isomorphic applications could suffer from scalability problems if a project is really successful and visited by millions of users. Because we are generating the HTML page pre-rendered on the server, we will need to create the right caching strategy to minimize the impact on the servers.

In this case, if the responses are highly cacheable, CDNs like Akamai, Fastly, or Cloudfront could definitely improve the scalability of our isomorphic applications by avoiding all the requests hitting origin servers.

Organization-wise, an isomorphic application suffers similar problems as an SPA whose code base is unique and maintained by one or multiple teams.

There are ways to mitigate the communication overhead if a team is working on a specific area of the application without any overlap with other teams. In this case, we can use architecture like Backends for Frontends (BFF) for decoupling the API implementation and allow each team to maintain their own layer of APIs specific to a target.

Static-Page Websites

Another option for your project is the static-page website, where every time the user clicks on a link you are loading a new static page. Fairly old school, I know, but it’s still in use—with some twists.

A static-page website is useful for quick websites that are not meant to be online for a long period, such as ones that advertise a specific product or

service we want to highlight without using the corporate website, or that are meant to be simple and easier to build and maintain by the end user.

In the last few years, this type of website has mutated into a single page that expands vertically instead of loading different pages. Some of these sites also lazy-load the content, waiting until the user scrolls to a specific position to load the content.

The same technique is used with hyperlinks, where all the links are anchored inside the same page and the user is browsing quickly between bits of information available on the website. These kinds of projects are usually created by small teams or individual contributors. The investment on the technical side is fairly low, and it's a good playground for developers to experiment with new technologies and new practices or to consolidate existing ones.

JAMStack

In recent years a new frontend architecture raised a good success called JAMStack (JavaScript, APIs, and Markup).

JAMstack is intended to be a modern architecture, to help create fast and secure sites and dynamic apps with JavaScript/APIs and pre-rendered markup, served without web servers.

In fact the final output is a static artifact composed by HTML, CSS and JavaScript, basically the holy trinity of frontend development.

The artifact can be served directly by a CDN considering the application doesn't require any server-side technology to work. One of the simplest way for serving a JAMStack application is using Github pages for hosting the final result

In this category we can find famous solutions like Gatsby.js, Next.js or Nuxt.js.

The key advantages of these architectures are better performances, cheaper infrastructure and maintenance considering they can be served directly by a

CDN, great scalability because we serve static files, higher security due to decrease of attack surface and easy integration with headless CMS.

JAMStack is a great companion for many websites we have to create especially considering the frictionless developer experience.

In fact, frontend developers can focus only on the frontend development and debugging, this usually means a more focused approach on the final result.

Summary

Over the years, the frontend ecosystem has evolved to include different architectures for solving different problems. A piece has been missing, though: a solution that would allow scale projects with tens or hundreds of developers working on the same project. Micro-frontends are that piece.

Micro-frontends will never be the only architecture available for frontend projects. Yet they provide us with multiple options for creating projects at scale.

Our journey in learning micro-frontends starts with their principles, analysing how these principles should be leveraged inside an architecture, and how much they resemble microservices.

Chapter 2. 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 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *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. 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 decade, 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.

Although cloud systems allowed us to scale our projects in an easier way than before, monoliths, unfortunately, require us to scale not just a single part of our system but the entire system, causing many headaches if the said system is not modularized or written with high standards.

Furthermore, working on a monolith codebase with distributed teams and co-located ones could be challenging, 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 advance 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 monolith applications, we have to deploy the entire codebase every single time, which comes with a higher chance of breaking the application program interfaces (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 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.

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. However, 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 to end 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 amount of architecture available in software development, we should pick microservices only when needed and not choose them recklessly just because it is the latest and greatest approach.

Micro-frontends are an emerging approach to defining software deliveries along business and responsibility boundaries in contrast to the monolithic approaches we have taken with Web development in the past. Keep in mind, however, that just like microservices aren't a universal answer to all software decomposition, neither are micro-frontends. To understand where they fit in and even what they are, let's look at some of the forces that are pushing us in this direction.

Monolith to Microservices

When we start a new project or even a new business offering a service online, the first iteration should be used for understanding if our 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 an *MVP* or *minimum viable product*.

Often we design our API layer as a unique codebase (monolith) so we need to set up one 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 for retrieving 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.
For instance if we need to create a concrete view of data to populate a user interface probably having a NoSQL database would make more sense than any other database, at the same time we can say that using a graph database for mapping relations between users like in a social network application would be a better fit for this kind of database.

Finally, we need to choose a technology for representing our data, such as within a browser or 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 2-1:

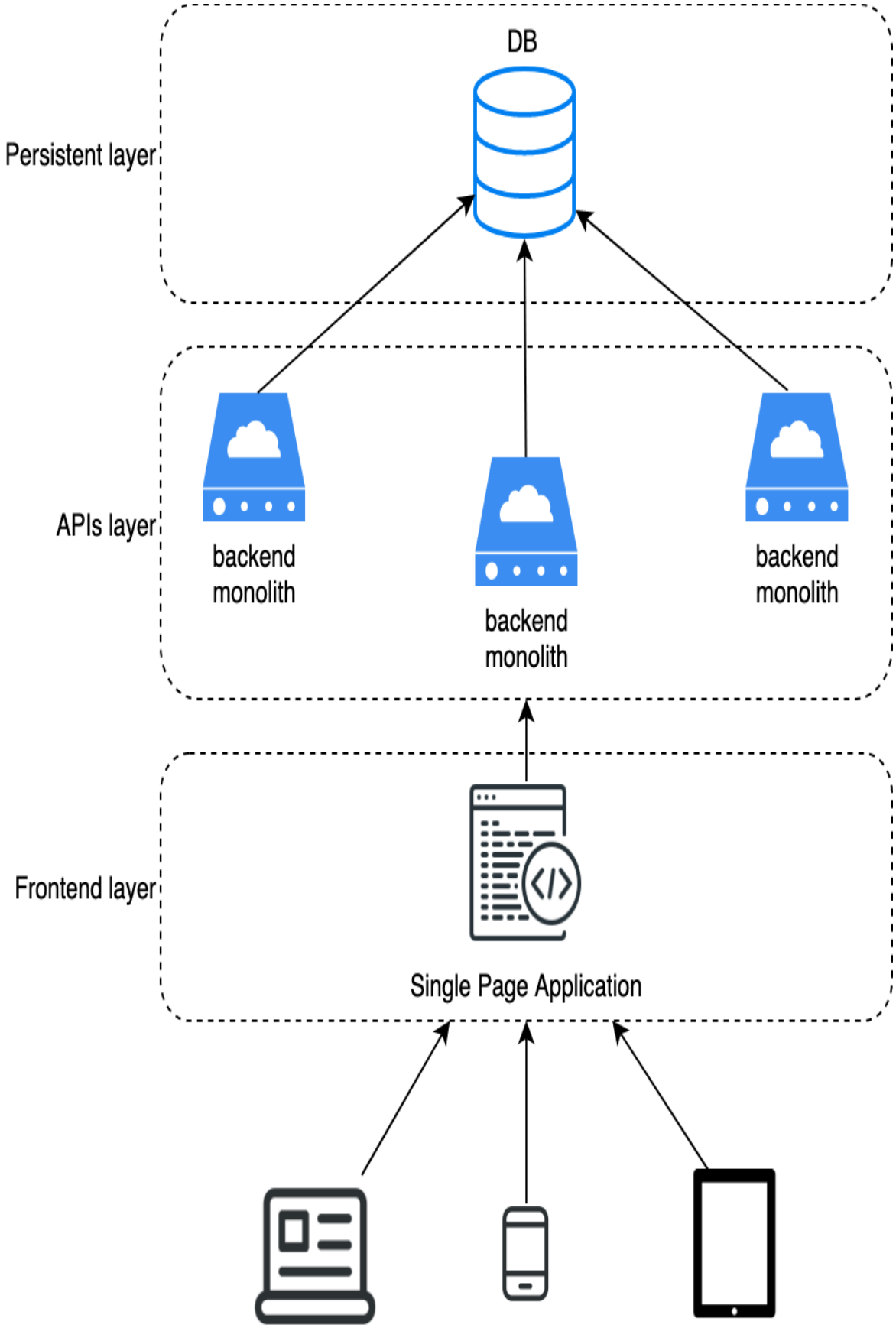

DB
Persistent layer
APIs layer
backend monolith
backend monolith
backend monolith
Frontend layer
Single Page Application

Figure 2-1. Monolith Application with Single Page Application

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 company 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 network (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 consistent 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 2-2.

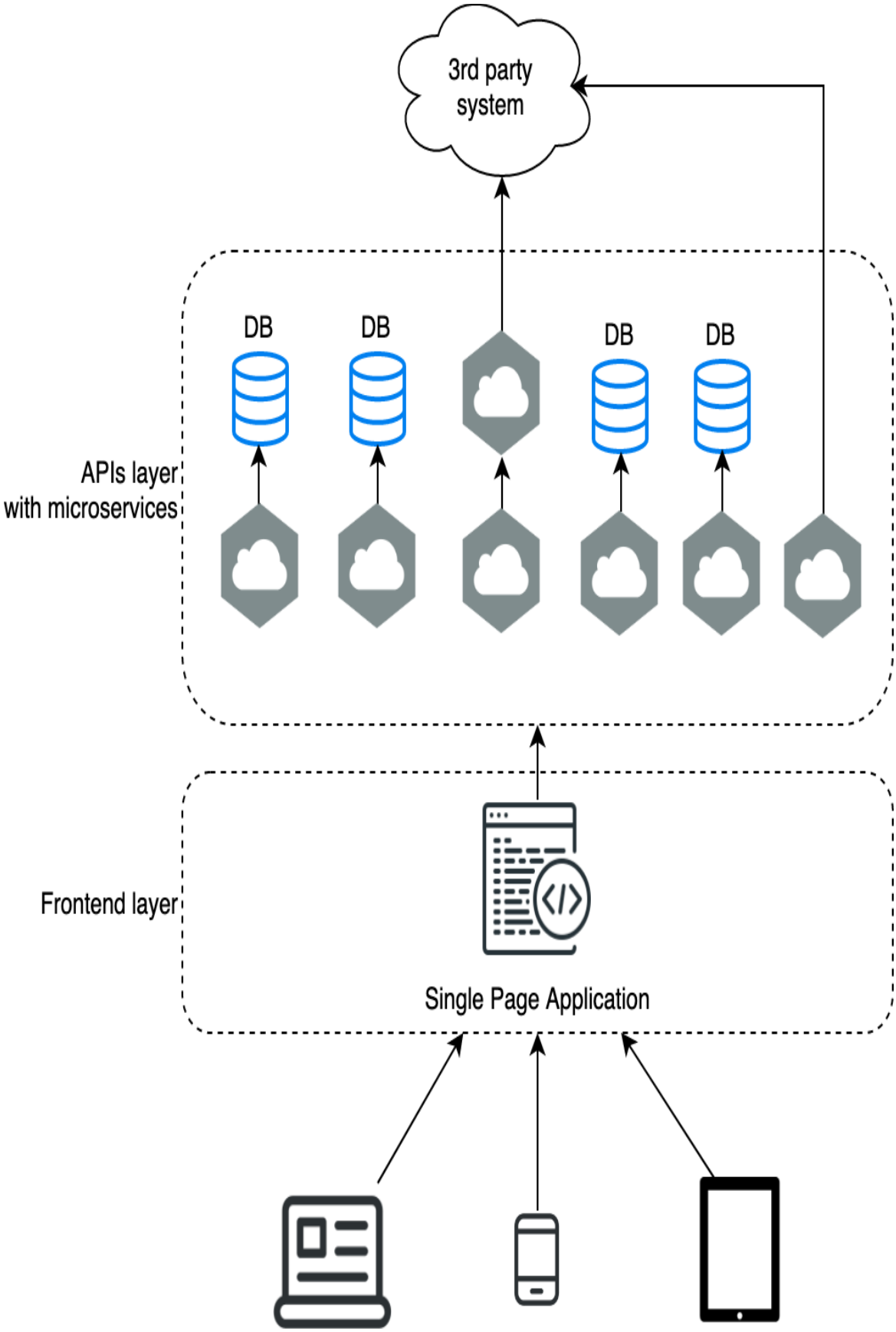

3rd party system
DB
DB
DB
DB
APIs layer with microservices
Frontend layer
Single Page Application

Figure 2-2. 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 in our web platforms, including more interactivity and better interactions.

Companies have arisen that provide 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 encounter quite a few challenges when embracing them.

For instance, when we have a medium or large team of developers, 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.

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 if not hundreds 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 2-3:

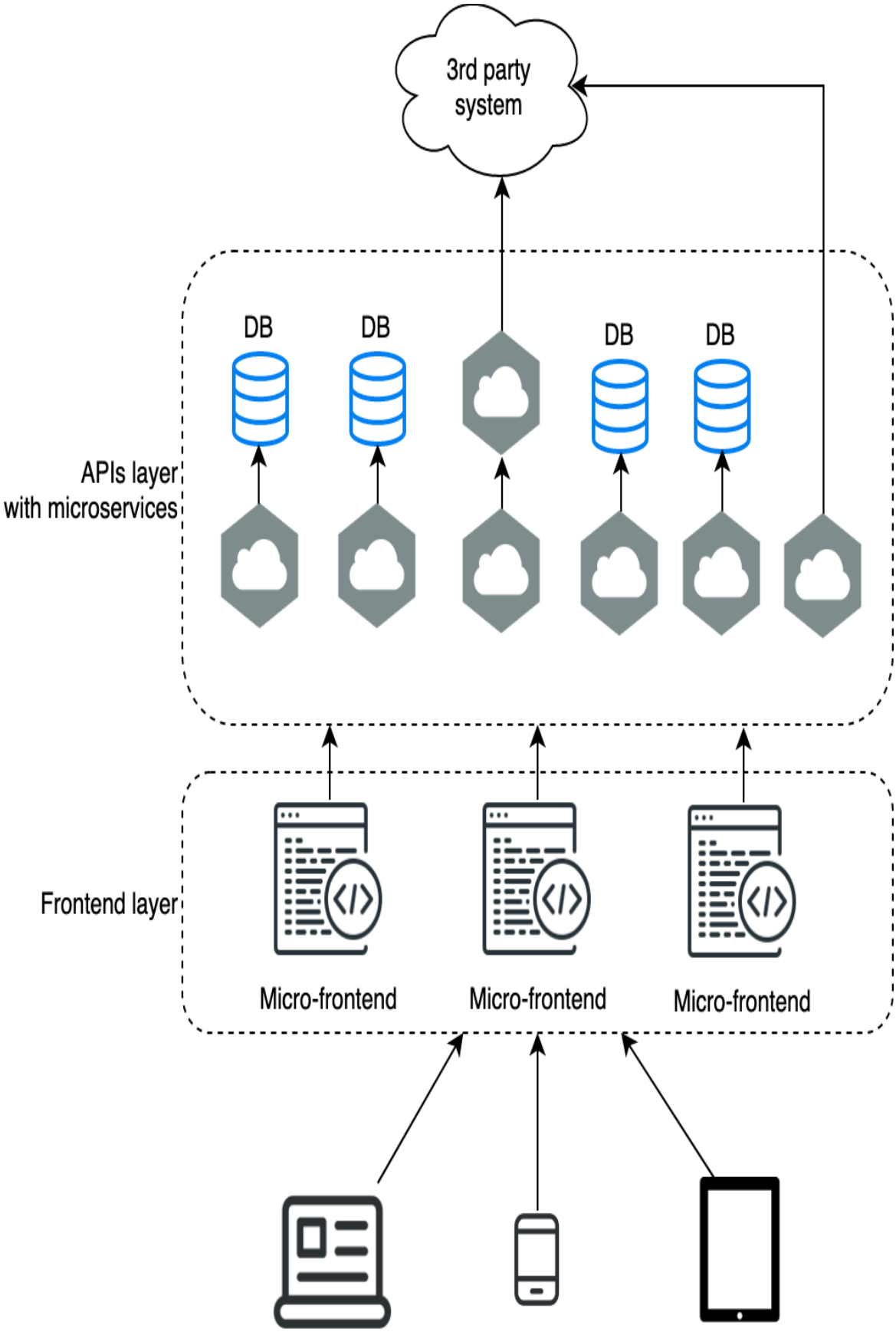

3rd party system
DB
DB
DB
DB
APIs layer with microservices
Frontend layer
Micro-frontend
Micro-frontend
Micro-frontend

Figure 2-3. 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 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 are the main principles behind micro-frontends to leverage as a guidance during the development of our projects.

Microservices Principles

At the beginning of my journey into micro-frontends, I stepped back from the technical side and looked at the principles behind other architectures for scaling a soft-

ware 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 2-4:

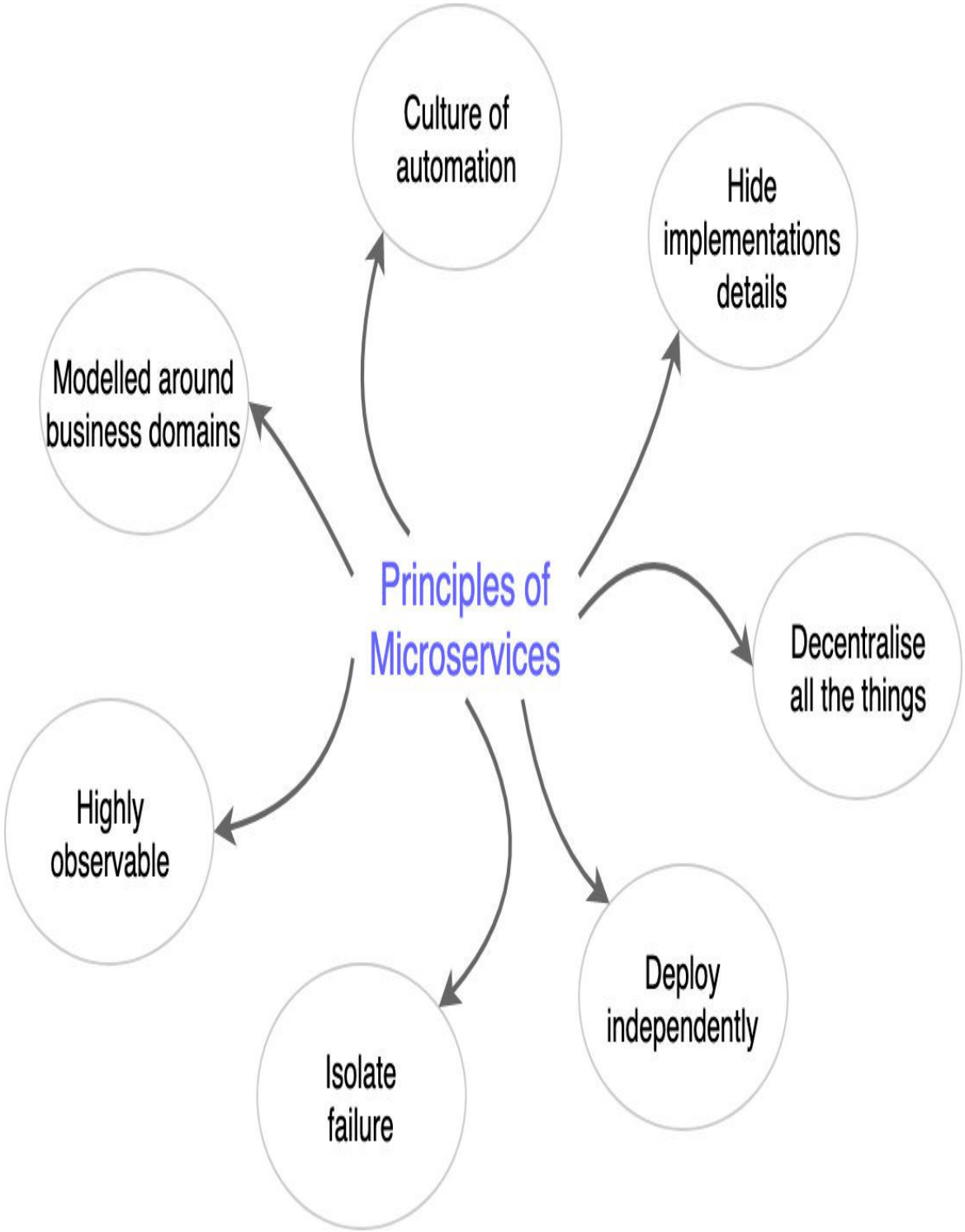


Figure 2-4. Microservices principles

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. We will discuss this topic extensively in the next chapters.

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 micro- services architecture; having a strong automation culture allows us to move faster and in a reliable way.

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.

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.

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 have 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.

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 micro-frontends 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.

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 a contract between teams and for all parties respect that during the entire development process. 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, without external dependencies, creating a more effective integration.

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) should *provide some guardrails* between which team 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.

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 isn't a huge problem due to their architecture, but it is with micro-frontends. In fact, micro-frontends may require composing a user interface at runtime, which may result in network failures or 404s for one or more files. To avoid impacting the user experience, we must provide alternative content or hide a specific part of the application.

Highly Observable

Frontend observability is becoming more prominent every day, with tools like Sentry and LogRockets providing great visibility for every developer. Using these tools is essential to understanding where our application is failing and why. For microservices, where anything can fail at any given point, being able to resolve the issue 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 than trying to completely prevent them. As with all microservices' principles, this is applicable to the frontend, too.

The exciting part of recognizing these principles on the front- 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. 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 development team of over 30 developers or when we want to replace a legacy project in an iterative way.

However, they are not a silver bullet for all frontend applications, such as server-side rendering, SPAs, or even single HTML pages.

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. We will explore the pros and cons of this architecture later in the book.

Summary

In this chapter we introduced what micro-frontends are, what their principles are, and how those principles are linked to the well-known, established microservices architecture.

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

Chapter 3. 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 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *Building.microfrontends@gmail.com*.

A micro-frontend represents a business domain that is autonomous, independently deliverable, and owned by a team. The key takeaways in this description, which will be discussed later, are closely linked to the principles behind micro-frontends:

- Business domain representation
- Autonomous codebase
- Independent deployment
- 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 3-1).

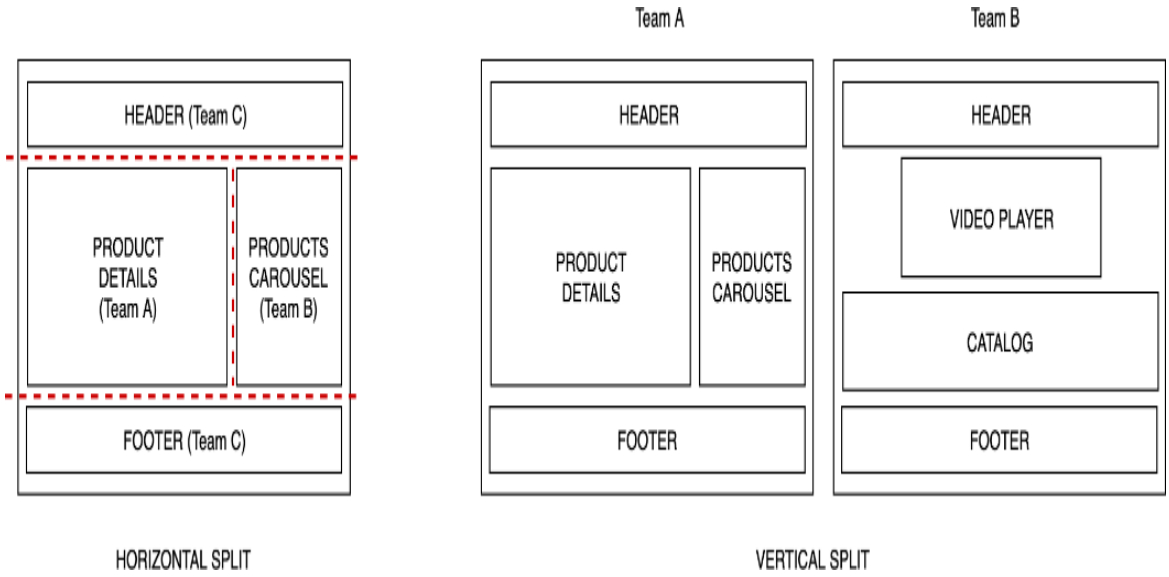


Figure 3-1. Horizontal vs. vertical split

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 a 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.

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.

DDD starts from the assumption that each software should reflect what the organization does and architectures should be designed based on domains

and subdomains leveraging ubiquitous languages shared across the business.

Applying DDD for frontend is slightly different from what we can do on the backend, certain concepts are definitely not applicable although there are others that are fundamental for designing a successful micro-frontends architecture.

For instance, Netflix's core domain is video streaming; the subdomains within that core domain are the catalogue, the sign-up functionality, and the video player.

There are three subdomain types:

- *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 for them 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 with these categories (Table 3-1).

Table 3-1. Subdomains examples

Subdomain type Example

Core subdomain Catalog

Supportive subdomain Voting system

Generic subdomain Sign in or sign up

Domain-Driven Design with Micro-Frontends

Another important term in DDD is the *bounded context*: 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, the bounded context can accommodate multiple subdomains because often the design of those systems was not thought of with DDD in mind.

The micro-frontends ecosystem offers many technical approaches. Some implementations are done with iframes, while others are done with components library or web components. Too often we spend our time identifying a technical solution without taking the business side into consideration.

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 realized that micro-frontends in the same view need to 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 make sure everything works properly.

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

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 components and 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

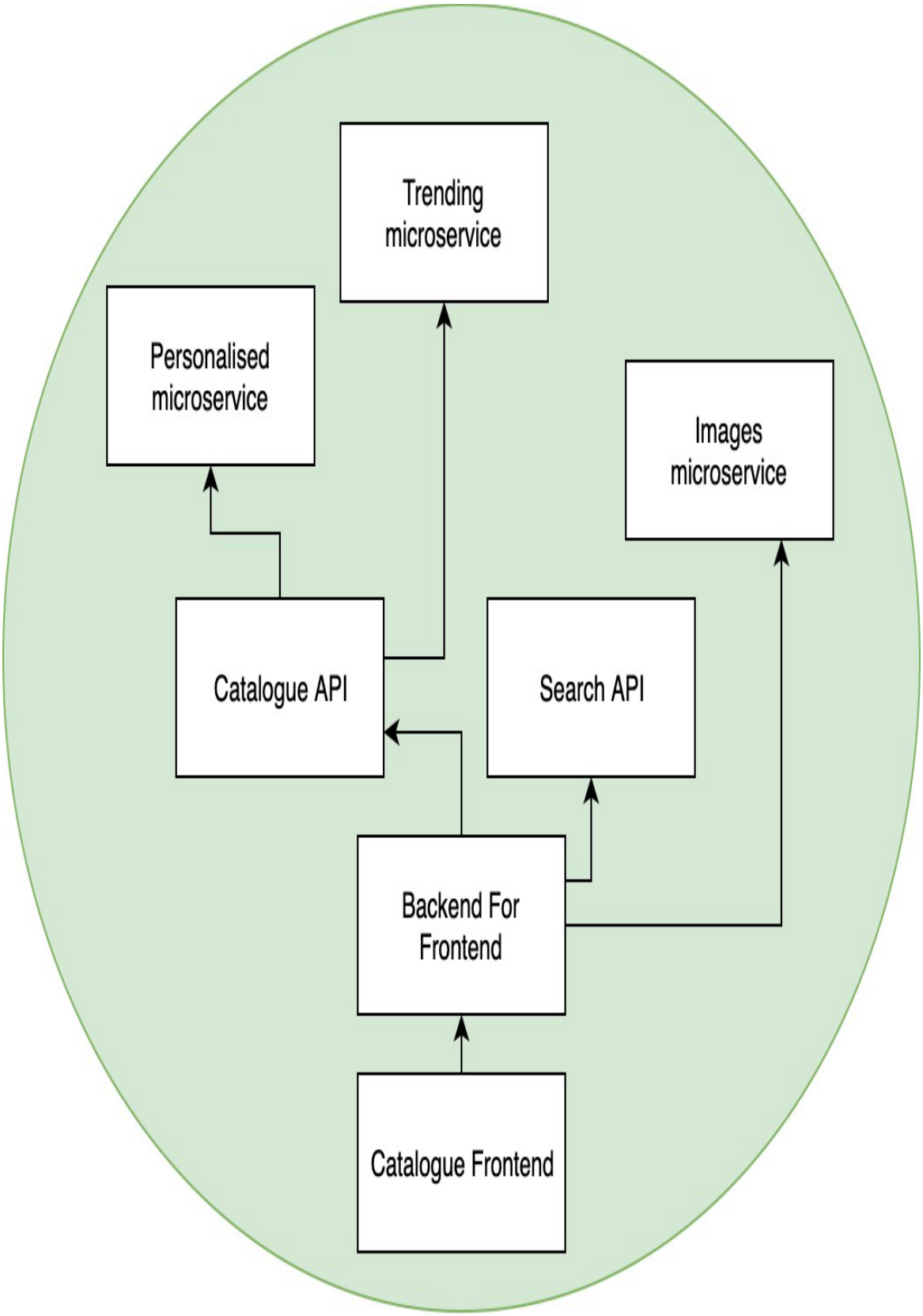
our project.

As architects or tech leads, our role is to invest enough time with the product team or the customers so we can identify the different domains and subdomains, working collaboratively with them.

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 3-2 we can see a representation of bounded context. In this example the bounded context contains the catalogue micro-frontends that consumes

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.



Catalogue Subdomain

Figure 3-2. 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.)!

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.

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 decide 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 in 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 horizontal instead of vertical depends on the type of project we have to build.

In fact, horizontal split serves better static pages like catalogs or e-commerce, instead of a more interactive project that would require a vertical split.

Another thing to be considered is the skills set of our teams, usually, a vertical split suits better for a more traditional client-side development experience, instead, the horizontal split requires an investment upfront for creating a solid and fast development experience to test their part as well trying inside the overall view.

Micro-frontends composition

There are different approaches for composing a micro-frontends application (Figure 3-3).

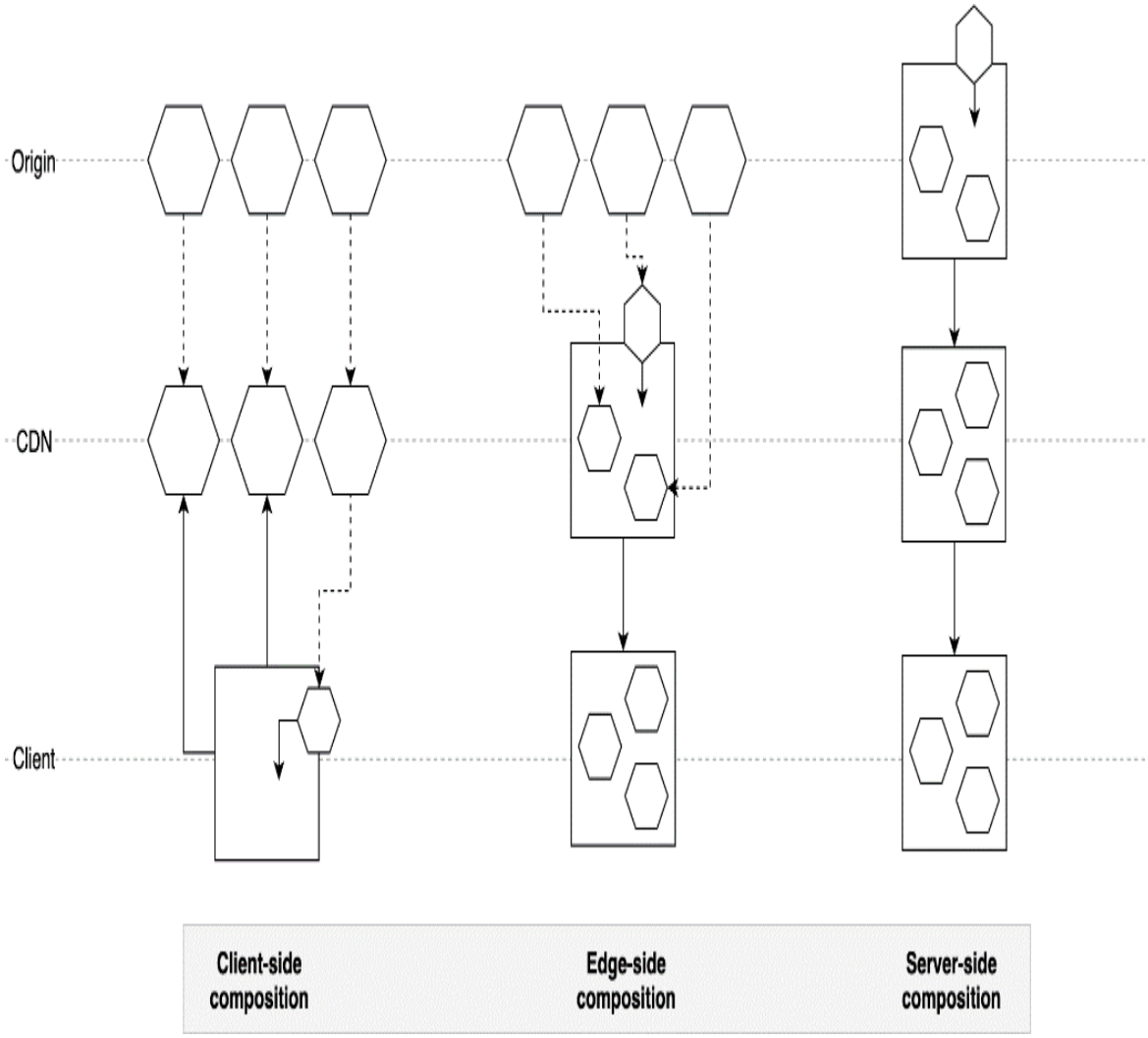


Figure 3-3. Micro-frontends composition diagram

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. 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.

Let's now see 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.

We can also use a combination of iframes to load different micro-frontends, or we could use 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. 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.

Server-Side Composition

The last possibility we have is the server-side composition, which could happen at runtime or at compile time. 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.

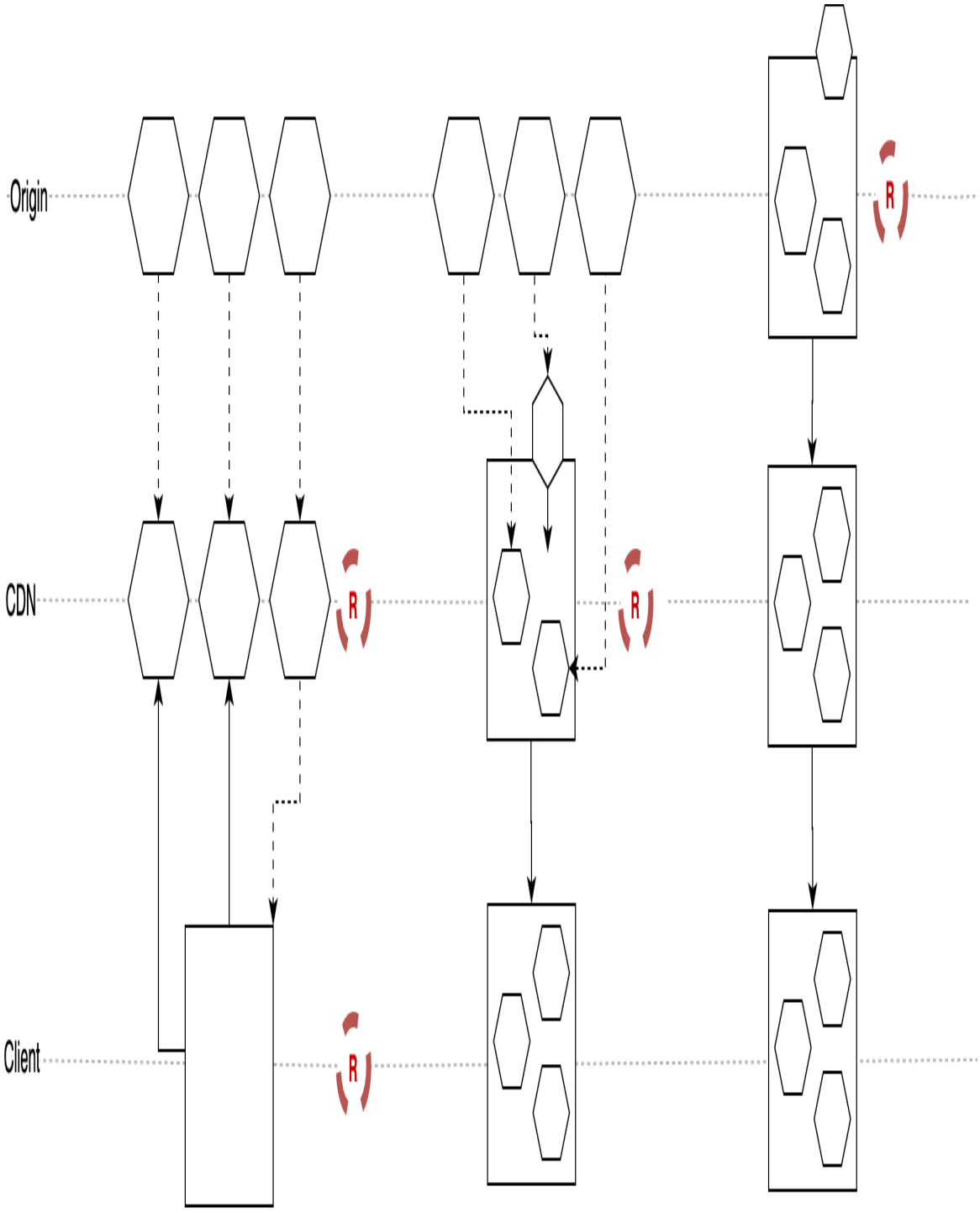
From these possibilities, we need to choose the technique that is most suitable for our project and the team structure. As we will learn later on in this journey, we also have the opportunity to deploy an architecture that exploits both client-side and edge-side composition—that's absolutely fine as 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 3-4.4).



Client-side composition

Edge-side composition

Server-side composition

Figure 3-4. 4 - Micro-frontends routing diagram

When we decide to compose micro-frontends at the origin, therefore a server-side composition on the right of Figure 3-4.4, 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 non trivial, 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 as an SPA, 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 in another page, similar to what we have when we use edge-side include approach.

We won't have any scalability issue in either case. The client-side routing is highly recommended when your teams have stronger frontend skills so that it becomes natural having a client-side routing over a backend configuration.

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

In an ideal world, micro-frontends wouldn't need to communicate with each other because all of them would be self-sufficient. In reality, it's not always possible to notify other micro-frontends about a user interaction, especially when we work with multiple micro-frontends on the same page.

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 unaware of the others on the same page, otherwise we are breaking the principle of independent deployment.

In this case, we have a few options for notifying other micro-frontends that an event occurred. 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 3-5).

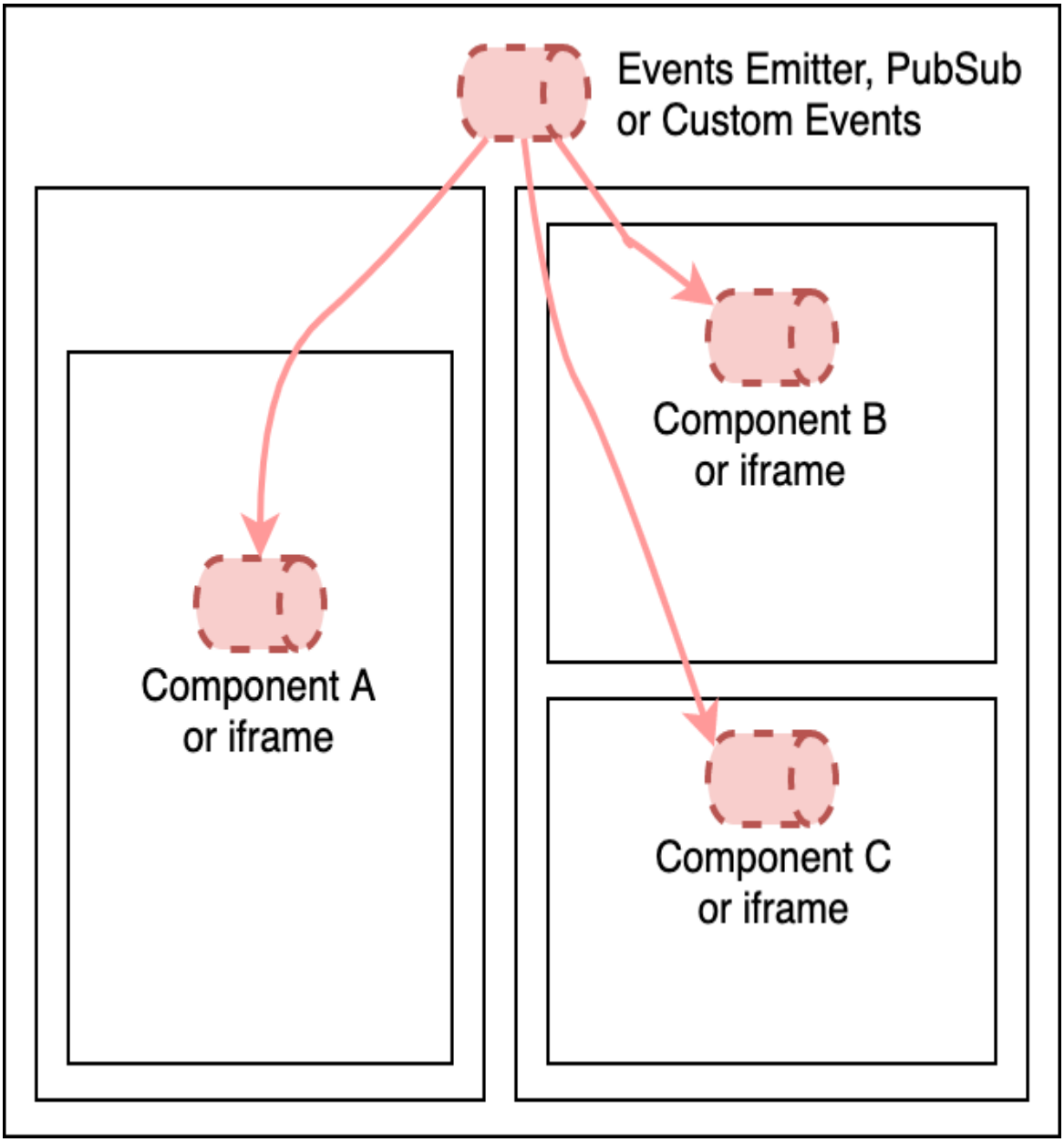


Figure 3-5. Event emitter and custom events diagram

To inject the eventbus, we need the micro-frontends container to instantiate the eventbus and inject it inside all of the page's micro-frontends.

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. Following an example

```
new CustomEvent('myCustomEvent', { someObj: "customData" });
```

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.

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 3-6). 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.

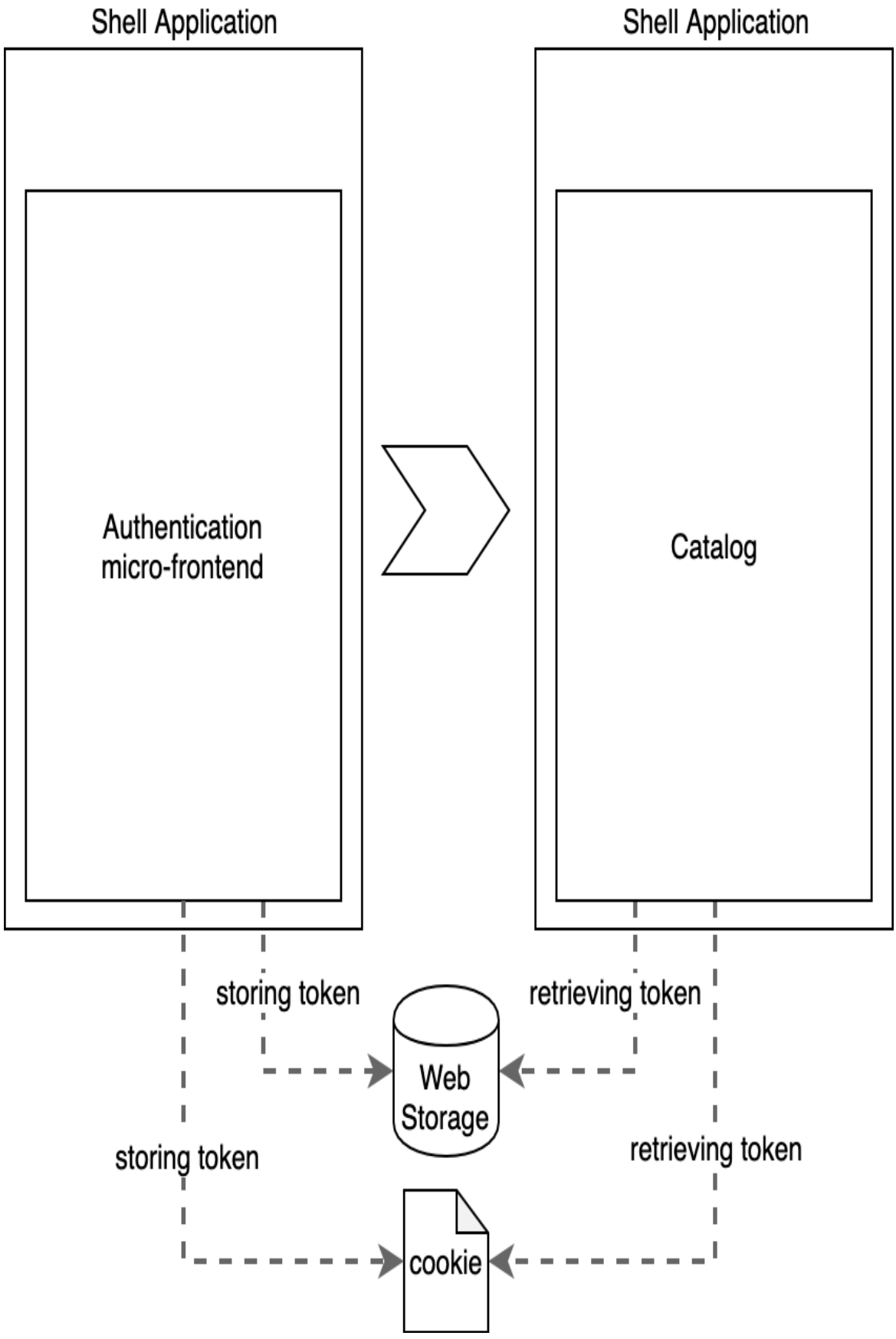

Shell Application
Shell Application
Authentication micro-frontend
Catalog
storing token
retrieving token
Web Storage
storing token
retrieving token
cookie

Figure 3-6. Sharing data between micro-frontends in different views

Another option could be to pass some data via query strings - for example, 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 3-7). Using query strings is not the most secure way to pass sensitive data, such as passwords and user IDs, however. There are better ways to retrieve that information if it's passed via the HTTPS protocol. Embrace this solution carefully.

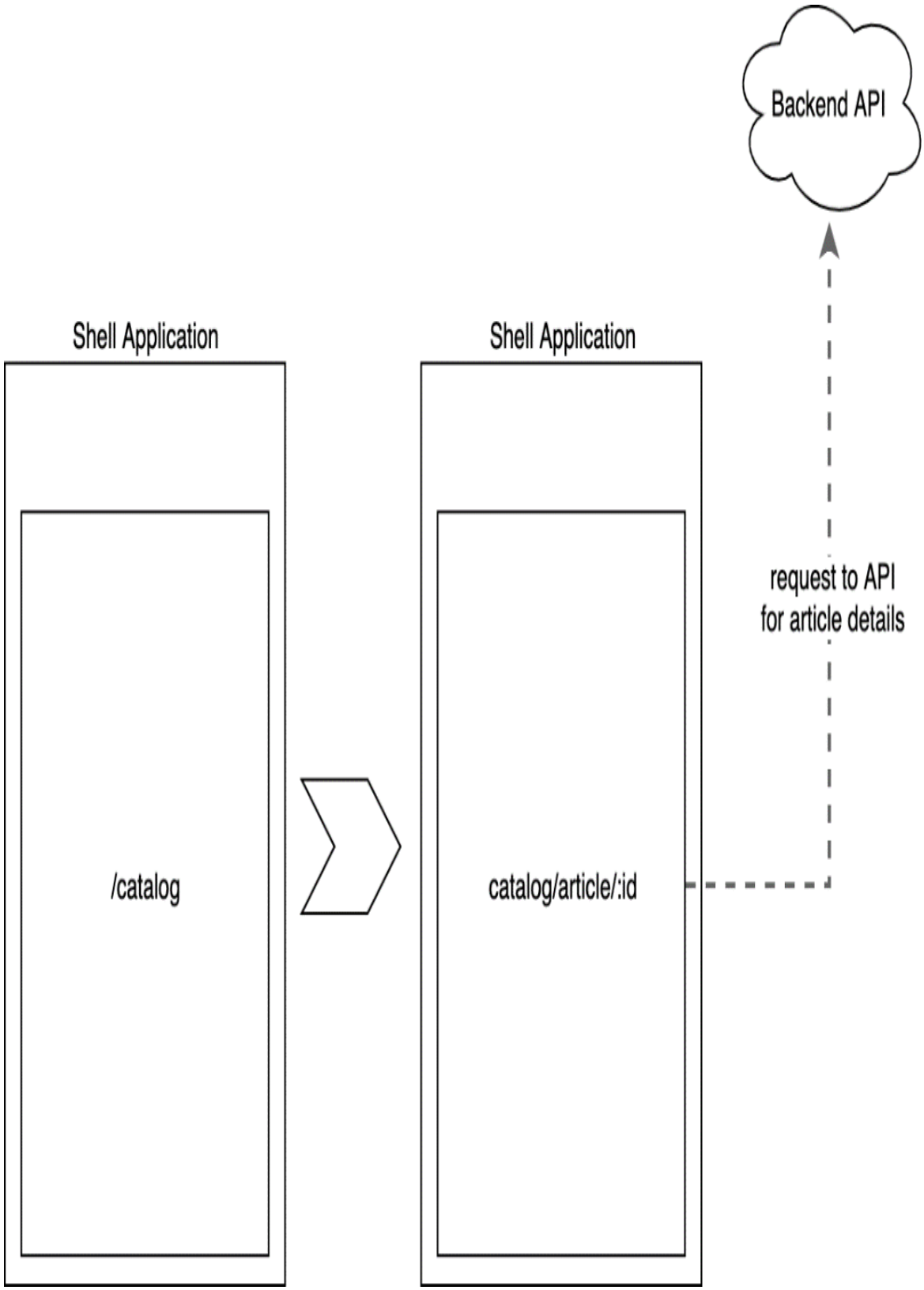


Figure 3-7. Micro-frontends communication via query strings or URL

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

In this table you can find all the combinations available based on how you identify a micro-frontend.

Table 3-2. Micro-frontends decisions framework summary

Micro-frontends definition Composition Routing Communication

Horizontal Client-side

Server-side Edge-side

Vertical Client-side

Server-side

Client-side Server-side Edge-side

Client-side Server-side Edge-side

Event emitter Custom events Web storage Query strings

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, the 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.

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

HelloFresh

HelloFresh, a digital service that provides ready-to-cook food boxes with a variety of recipes from all over the world, is another good example.

Inspired by Zalando's work, HelloFresh is now serving a multitude of SPAs orchestrated by URL.

In an interesting approach to flexibility of components, the SPAs are assembled and rendered on the servers, then cached at the CDN level, providing flexibility for generating the SPAs.

This approach also allows the development teams to be responsible for their own technology stacks; each SPA could have a different one, and each team is fully independent from the others.

AllegroTech

In 2016, Polish e-tailer and auction site AllegroTech came up with OpBox, a project that allows nontechnical people to merge UI representations (a.k.a., components) with data sources inside the same page.

At first, AllegroTech tried to work with multiple components assembled at runtime with ESI lang, but the system didn't provide the desired level of consistency. Furthermore, they had a few problems with managing specific library versions. For instance, one component could have been developed with React v13 and another one with v15, both rendered on the same page.

In the OpBox project, Allegro's teams had the opportunity to decouple the rendering part of a component (the view) from the data in order to render. As long as the contract between the component and the data source matched, they were able to assemble data and different components

together, which enhanced their ability to do A/B testing and gather data from there.

But it's the additional abstraction between how the page is composed and the components to display that really stands out in this implementation. In fact, a JSON file describes the page and the components needed, and the renderer then composes

the page as configured inside the JSON file.

Obviously two or more components on the same page could also react to a specific user interaction or to a change in a set of data, thanks to an eventbus implementation that signals the change to all the components that are listening to it.

Spotify

In this list of case histories, I can't neglect to mention Spotify.

For its desktop application, Spotify has assembled multiple components living in separate iframes that communicate via a "bridge" for the low-level implementation made with C++.

If we inspect the Spotify desktop application, we can easily find the multiple parts composing this application. Each single *.spa* file is composed by an *html* file, multiple *css* files, a *manifest.json*, **and a *JavaScript* bundle file minimized and optimized (Figure 2-8).

https://docs.google.com/drawings/u/1/d/srOqMCJWnBIaT2VKsyRwo7w/image?w=481&h=177&rev=1&ac=1&parent=1oLWU66mMvCW-van-37Sb-SEZqx9nJLgz

Figure 2-8. Spotify micro-frontend artifact

Those files will be loaded inside an iframe to compose the final application UI.

This approach was used at the beginning for the web instance of the Spotify player, but it was abandoned due to its poor performance, and Spotify has since moved back to an SPA architecture similar to what they have for the

TV application. This doesn't mean the approach can't work, but the way it was designed caused more issues for the final users than benefits.

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 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 are using 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.

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, and Microsoft.

Summary

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

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 4. Build and Deploy 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 4th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *Building.microfrontends@gmail.com*.

In this chapter, we discuss another commonality between micro-frontends and microservices: 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 amount of repetitive actions to build and deploy them.

Any company embracing the microservices architecture, therefore, has to invest a considerable amount of time and effort on their *continuous*

integration (CI) or *continuous deployment* (CD) pipelines (more on these below).

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 lifecycle.

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-frontends 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 non-trivial to manage them.

One of the key decisions listed in the micro-frontends decision framework, discussed in chapter 3, is the possibility to compose multiple micro-frontends in the same view (horizontal split) or having just one micro-frontend per 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 it's near 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 deep dive into these challenges in this chapter, 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-frontends deployment and conclude with an introduction to fitness functions

for automating and testing architecture characteristics during different stages of the automation 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.

> **NOTE**
>
> Continuous Integration vs. Continuous Delivery vs. 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 every code committed in the version of control system.
>
> If you are interested in learning more, I recommend reading Continuous Delivery, available on Safari Books Online.
>
> 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 tools 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 for 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 for creating 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 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.

However, there is a complexity factor to consider.

Due to maintaining many similar automation pipelines, we should embrace infrastructure as code (IaC) principles for spinning new pipelines without manually creating or modifying several pipelines.
In fact, IaC leverages the concept of automation for configuring and provisioning infrastructure in the same way we do for our code.
In this way, we can reliably create an infrastructure without the risk of forgetting a configuration or misconfiguring part of our infrastructure.
Everything is mapped inside configuration files, or code, providing us with a concrete way to generate our automation pipelines when they need to be replicated.
This becomes critical when you work with large teams and especially with distributed teams, because you can release modules and scripts that are reusable between teams.

Iterate Often

An automation pipeline is not a piece of infrastructure that once defined remains as it is until the end of a lifecycle 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. Screens near the developers' desks 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 take over 8-10 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 3-4 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, 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 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 because of 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 company'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 for making 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. Slack and Microsoft Teams meetings lack everyday, casual work conversations we have around the coffee machine.

Define Your Guardrails

An important principle for empowering teams and having a solid automation strategy is creating some guardrails for the teams, 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.

Guardrails for the automation strategy are usually defined by architects and/or Platform or cloud engineers in collaboration with tech leaders.

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.

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, imagine 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, with micro-frontends there aren't 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.

We will dig deeper into this topic later in this chapter, but bear in mind that automating your testing strategy guarantees your independent artifacts deployment won't result in constant rollbacks or, worse, runtime bugs experienced by your users.

Developers Experience (DX)

A key consideration when working with micro-frontends is the developers 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 frictions in developing new features.

WHAT DOES DEVELOPERS 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 lifecycle across the entire organization.

One of DX's main goals is to simplify the development and process of building, testing, and deploying artifacts in different environments.

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 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 lifecycle.

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, 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 vs. Vertical split

The decision between a horizontal and vertical split with your new micro-frontends project will definitely impact the developers' 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 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 a company may not fit in another one.

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, of course, 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 writing 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 be used.

Another important thing 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 tries 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, as well as 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 preview an experiment or a specific branch containing a feature to the business or a product owner.

Nowadays many cloud providers like AWS can provide great savings using spot instances for a middle-of-the-road approach, where the infrastructure cost is by far cheaper than the normal offering because some machines aren't available for a long time. Spot instances are a perfect fit for on-demand environments.

Version of Control

When we start to design an automation strategy, deciding a version of control and a branching strategy to adopt is a mandatory step.

Although there are valid alternatives, like Mercurial, Git is the most popular for a version of 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 a version of control means deciding which approach to use in terms of repositories.
Usually, the debate is between monorepo and polyrepo, also called multi-repo. There are benefits and pitfalls in both approaches., You can emply both, though, in your micro-frontend project, so you can use the right technique for your context.

Monorepo

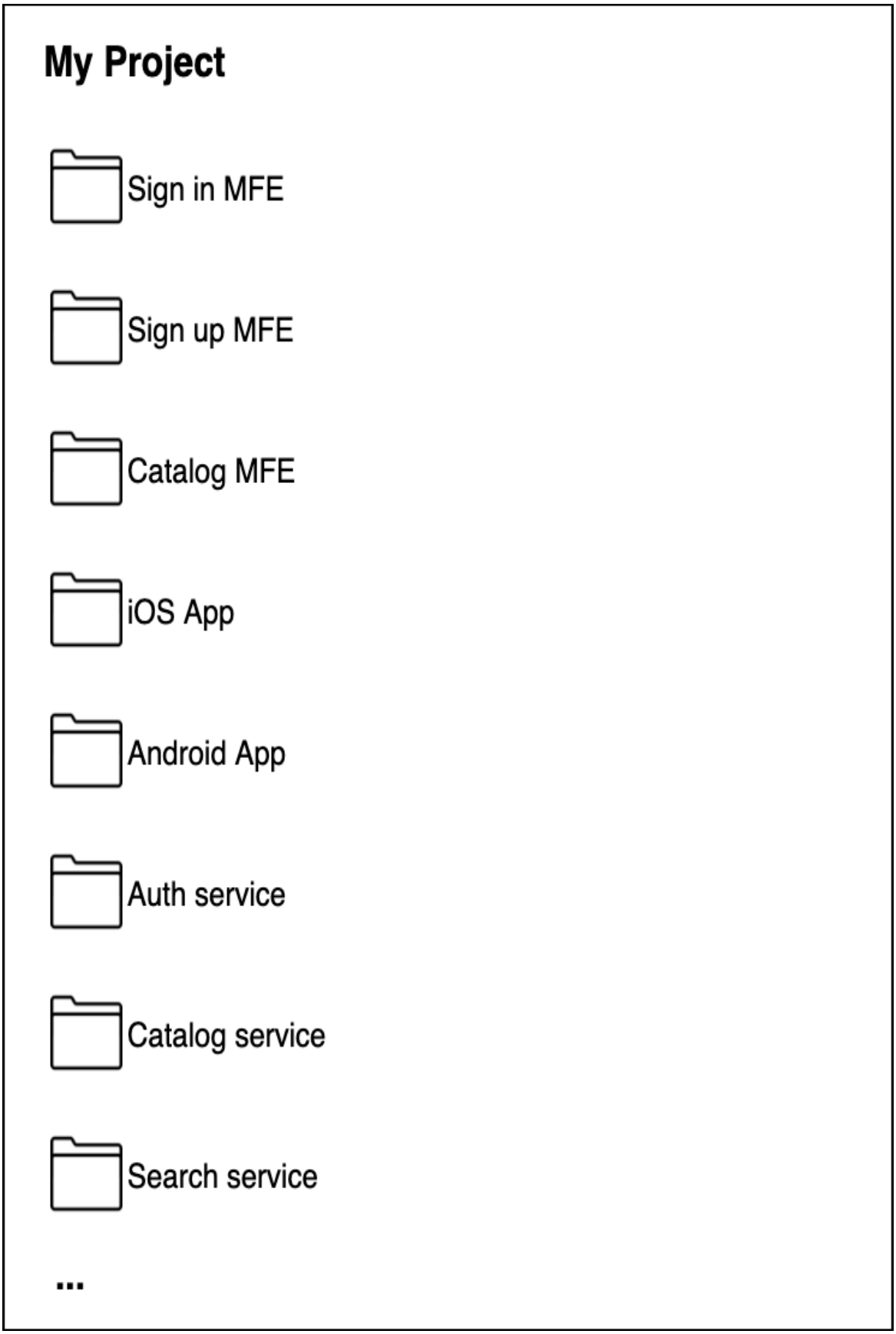

My Project
Sign in MFE
Sign up MFE
Catalog MFE
iOS App
Android App
Auth service
Catalog service
Search service
...

Figure 4-1. 1 Monorepo example where all the projects live inside the same repository

Monorepo (figure 7.1) is based on the concept that all the teams are using the same repository, therefore all the projects are hosted together.

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 on 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 dependencies 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. And 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 find code samples from other repositories very quickly.
  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 on 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 company aforementioned has built its own version of build tools and released it as open-source, considering 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 different monorepo. This is a key consideration to think 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 radio 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 into their machine without needing to clone the entire git history and all the projects folders.

Obviously, these enhancements will be beneficial for our CI/CD, as well, 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 (Git >2.25) 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 monorepo.

We need to remember that using monorepo 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

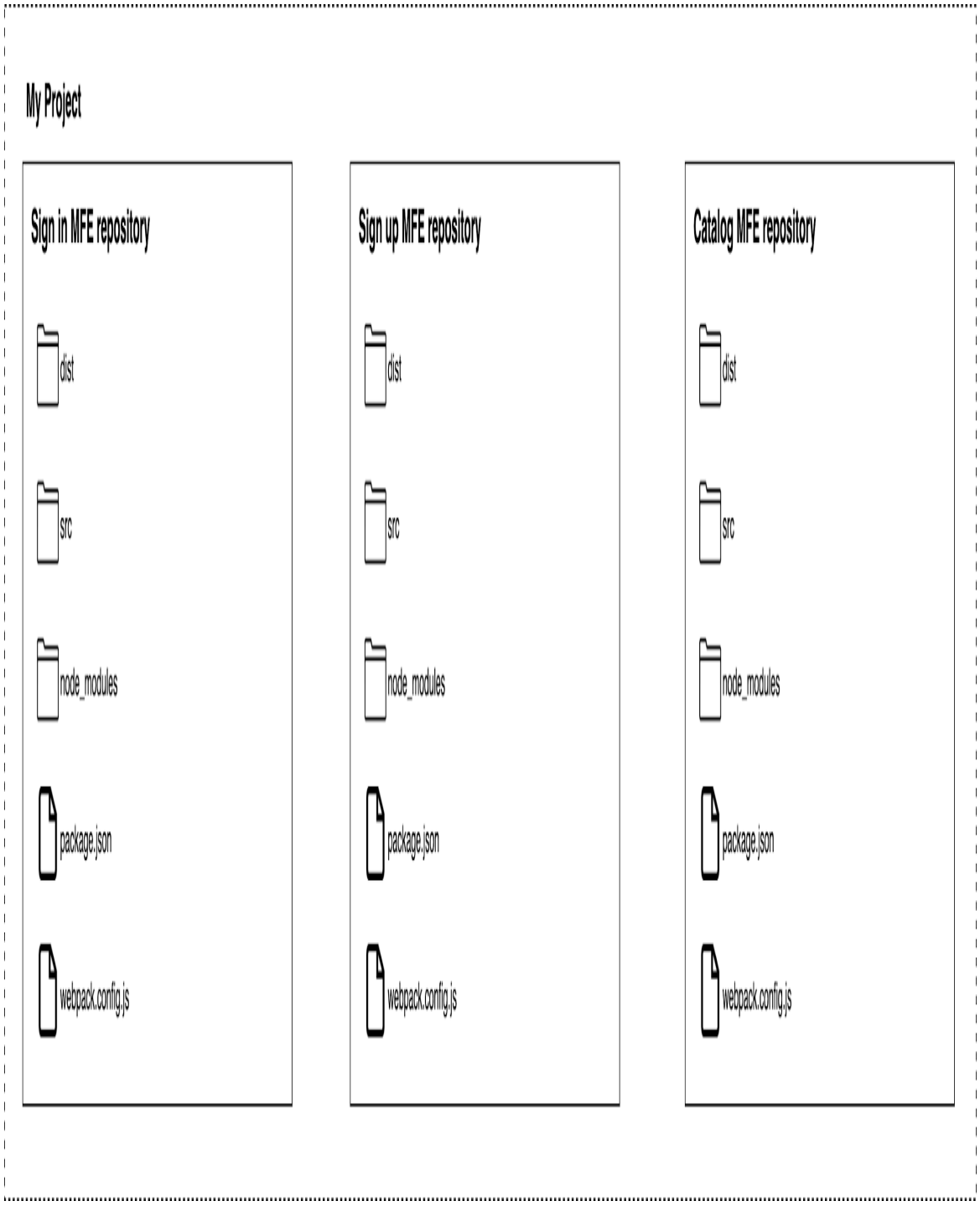


Figure 4-2. 2 Polyrepo example where we split the projects among multiple repositories

The opposite of a monorepo strategy is the polyrepo (figure 7.2), or multi-repo, where every single application lives in its own repository.

Some benefits of a polyrepo strategies 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. 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 upfront 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 researches.

- **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 codesharing 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 microservice as well, could result in a huge amount of repositories inside our version of control.

- **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 think this through upfront because finding out these problems during development will slow down your teams' 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-frontends 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-frontends approach, while keeping the same one for the legacy project without the need of polluting 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 of control systems

Any of the different paths we can take with a version of control systems won't be a perfect solution, just the solution that works better in our context. It's always a tradeoff.

However, we may want to try a hybrid approach, 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 7.3).

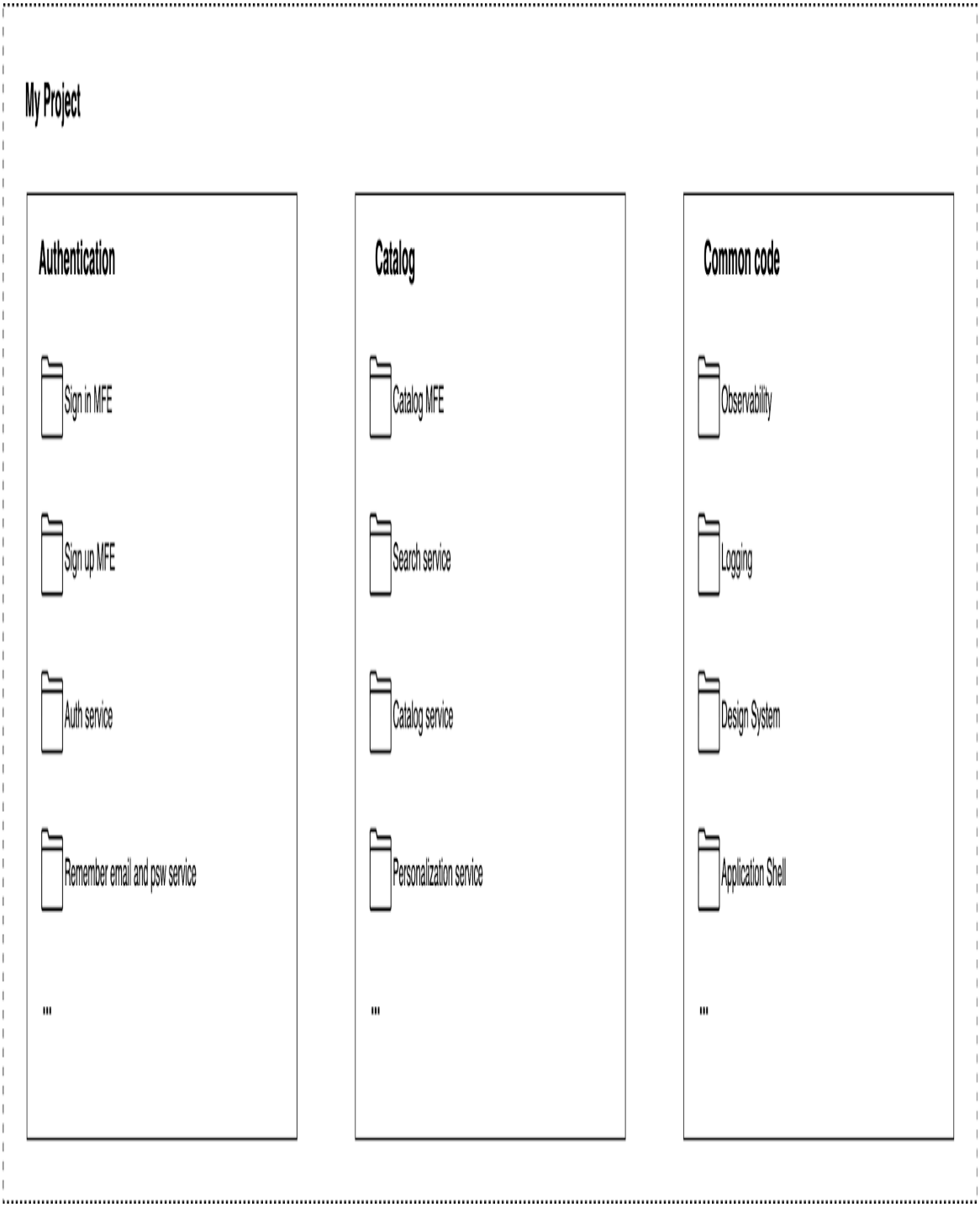


Figure 4-3. 3 A hybrid repositories approach, where we can combine monorepo and polyrepo strengths in a unique solution.

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.

Thisapproach might result in new practices, new tools to use or build, and new challenges, but it's an interesting solution worth exploring for micro-architectures.

Continuous Integration strategies

After identifying the version of the control strategy, we have to think about the continuous integration (CI) 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 lifecycle, 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-frontends 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 allows also 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 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, 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 architectures.

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 Uncle Bob, especially:

Clean Code — Robert C. Martin

Test-Driven Development: By Example — Kent Beck

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 the same in terms of the micro-frontends architecture we decide to use for our project, we'll focus here on end-to-end testing, as this is a bigger challenge in regards to 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 them 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 to turn on and off a feature and granting the access to test for a set of users for performing the tests.

Testing in production brings its own challenges, especially when a system is integrating with 3rd party APIs.
However, it will save a lot of money on environment infrastructure, maintenance and developers' resources.

I'm conscious not all the companies or the 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 in consideration the possibility to introduce features 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 7.4.

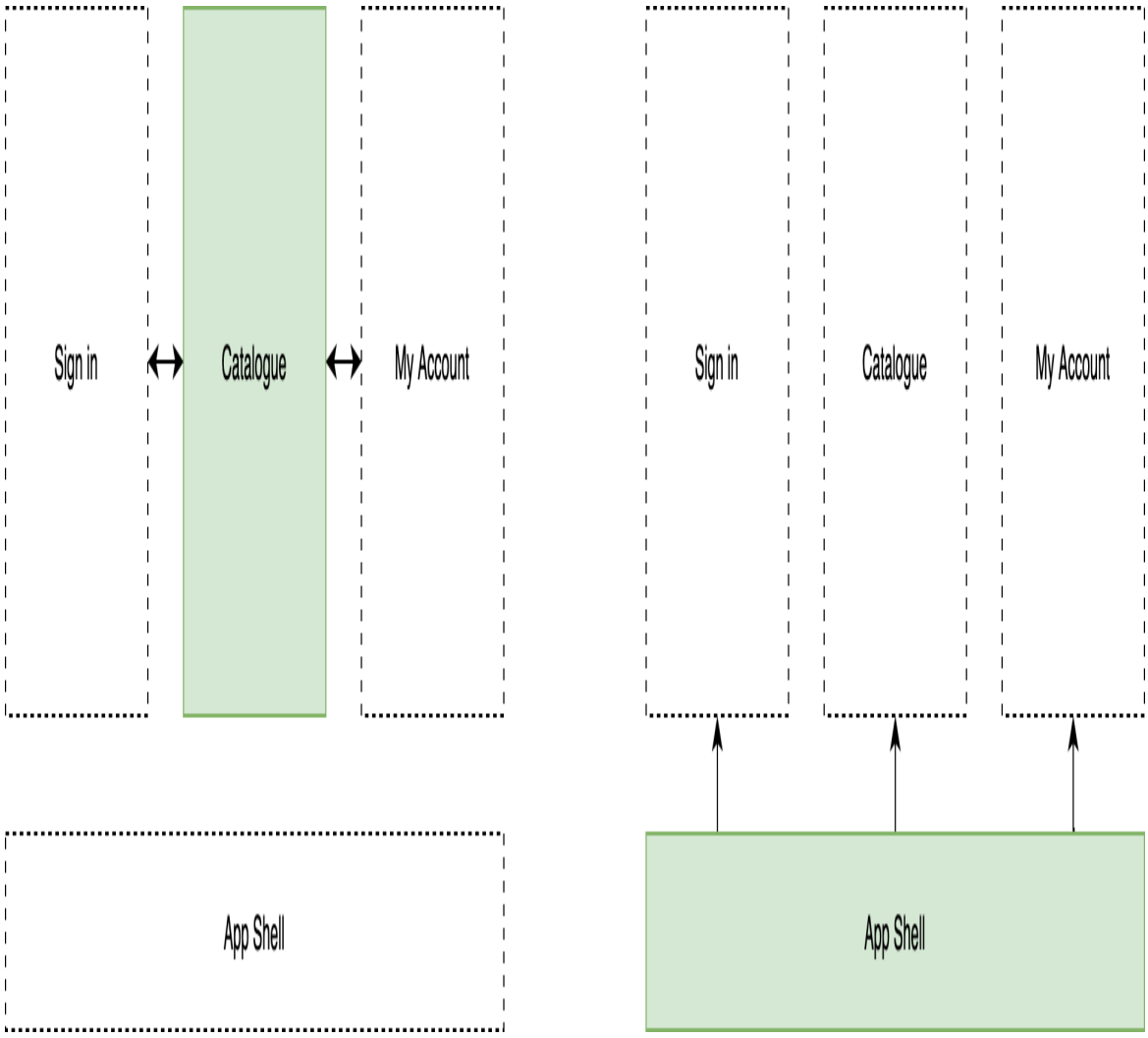


Figure 4-4. 4 - An end-to-end testing example with a vertical split architecture

The catalogue team is responsible for testing all the scenarios related to the catalogue, however, some scenarios involve areas not controlled by the catalogue 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 catalogue team will be responsible for writing tests that cross their domain boundary 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-frontend will need to test their business domain and verify that the

catalogue 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 7.4.

The application shell team should be responsible for these tests, ensuring that the entire route of the application loads correctly, that the key behaviours like signing in or out of 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 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 is present?

Let's try to shed some light on this with the example in figure 7.5.

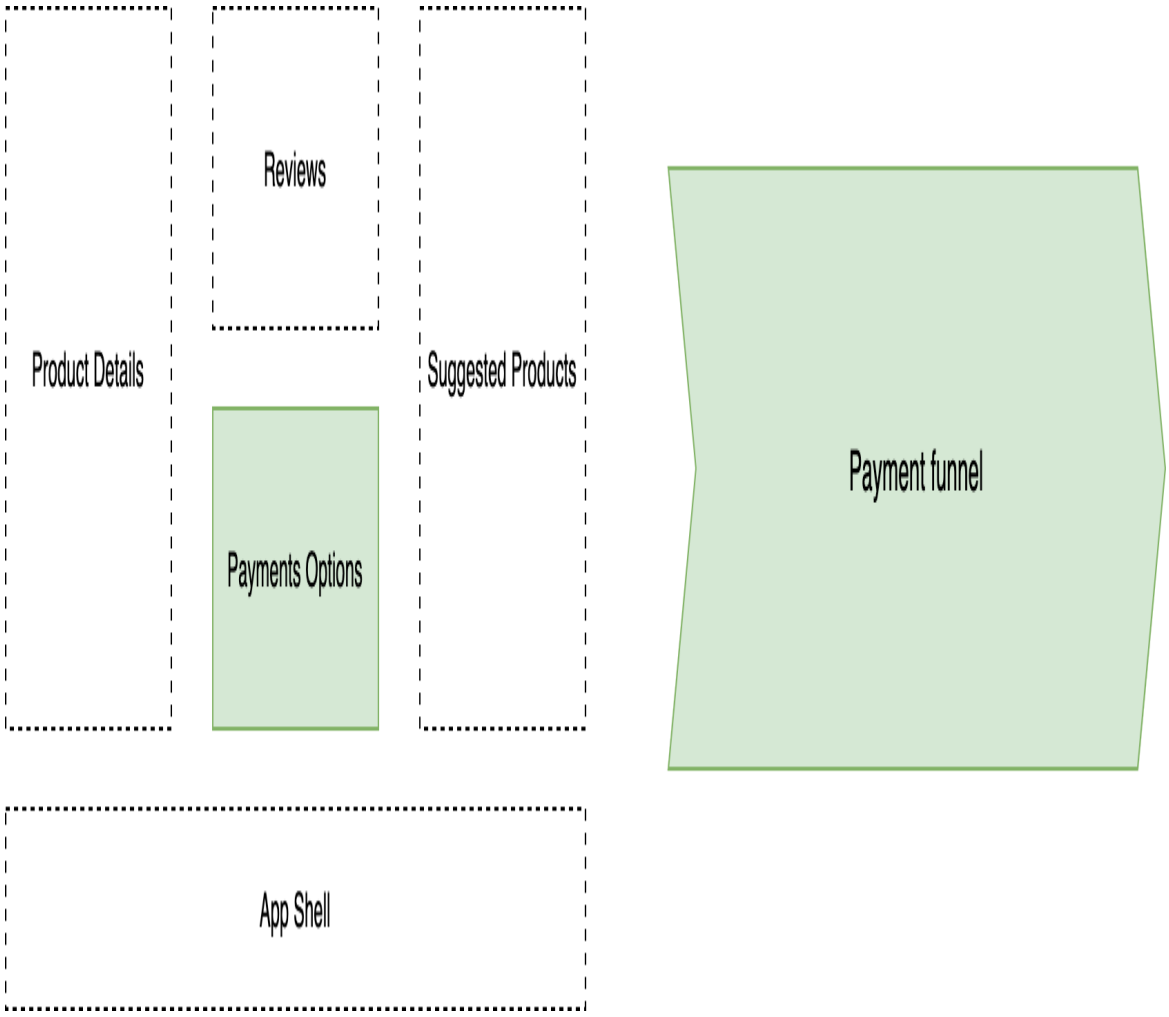


Figure 4-5. 5 - An end-to-end testing example with a horizontal split architecture.

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 7.5, 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 due to the fact that different teams are contributing to the final output of a view, and some tests may become invalid or broken the moment other teams are changing 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, particularly 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 consuming APIs in a specific environment.

This feature may come 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 very trivial, as you can see in the following example:

```
// In webpack.config.js
{
devServer: {
proxy: {
'/api’: {
target: ‘https://other-server.example.com',
secure: false
}
}
}
}
// Multiple entry
proxy: [
{
context: ['/api-v1/**', '/api-v2/**'],
target: ‘https://other-server.example.com',
```

```
secure: false
}
]
```

More 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 be applied to other stages of our CI pipeline, as well. 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

In a distributed system world 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 co-located or distributed.

In their book *Building Evolutionary Architecture*, Neal Ford, Rebecca Parson, and Patrick Kua discuss how to test an architecture's characteristics in CI with fitness functions.

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-frontends 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-frontends 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-up outcome and trade-off that a micro-frontends application should have.

Some key architecture characteristics I invite you to pay attention to when designing the automation pipeline for a micro-frontends project are:

- **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 a micro-frontend.
- **Performance metrics**: Tools like lighthouse and webperf 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 like 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 regulation 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.

While none of these metrics is likely new to you, in a distributed architecture like this one, they become fundamental for architects and tech leads to understand the quality of the product developed, to understand where the tech debt lays, 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 the company to achieve the goals agreed on the product quality.

Micro-frontends specific operations

Some automation pipelines for micro-frontends may require additional steps compared to traditional frontend automation pipelines.

The first one worth a mention 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-frontend.

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.

Deployment Strategies

The last stage of any automation strategy is the delivery of the artifacts created during the build phase.

Whether we decide to deploy our code via continuous deployment, shell script running on-prem, in a cloud provider, or via a user interface, understanding how we can deploy micro-frontends independently from each other is fundamental.

By their nature, micro-frontends should be independent. The moment we have to coordinate a deployment with multiple micro-frontends, we should question the decisions we made identifying their boundaries.

Coupling risks jeopardize the entire effort of embracing this architecture, generating more issues than value for the company due to the complicated way of deploying them.

With micro-architectures, we deploy only a small portion of code without impacting the entire codebase. As with micro-frontends and microservices, we may decide to move forward to avoid the possibility of breaking the application and, therefore, the user experience. We'll present the new version of a micro-frontend to a smaller group of users instead of doing a big-bang release to all our users.

For this scope, the microservices world uses techniques like blue-green deployment and canary releases, where part of the traffic is redirected to a new microservice. Adapting these key techniques in any micro-frontends deployment strategy is worth considering.

Blue-Green Deployment versus Canary Releases

Blue-green deployment starts with the assumption that the last stage of our tests should be done in the production environment we are running for the rest of our platform.

After deploying a new version, we can test our new code in production without redirecting users to the new version while getting all the benefits of testing in the production environment.

When all the tests pass, we are ready to redirect 100% of our traffic to the new version of our micro-frontend.

This strategy reduces the risk of deploying new micro-frontends because we can do all the testing needed without impacting our user base.

Another benefit of this approach is that we may decide to provision only two environments, testing and production; considering that all the tests are

running in production with a safe approach, we're cutting infrastructure costs not having to support the staging environment.

As you can see in figure 7.6, we have a router that should aim for shaping the traffic toward the right version.

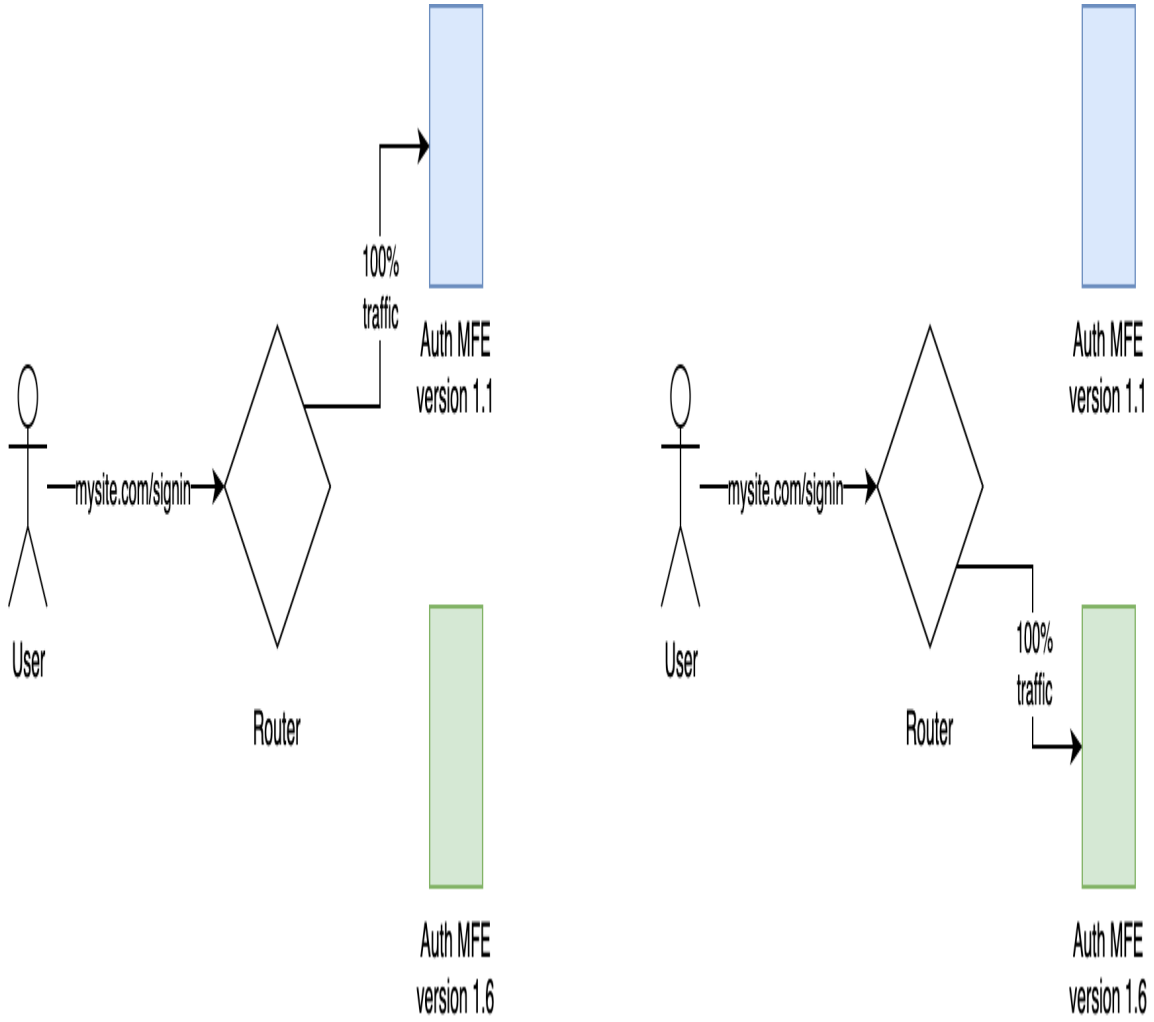


Figure 4-6. 6 Blue-green deployment

In canary releases, we don't switch all of the traffic to a new version after all tests pass. Instead, we gradually ease the traffic to a new micro-frontend version. As we monitor the metrics from the live traffic consuming our new frontend, such as increased error rates or less user engagement), we may decide to increase or decrease the traffic accordingly (figure 7.7).

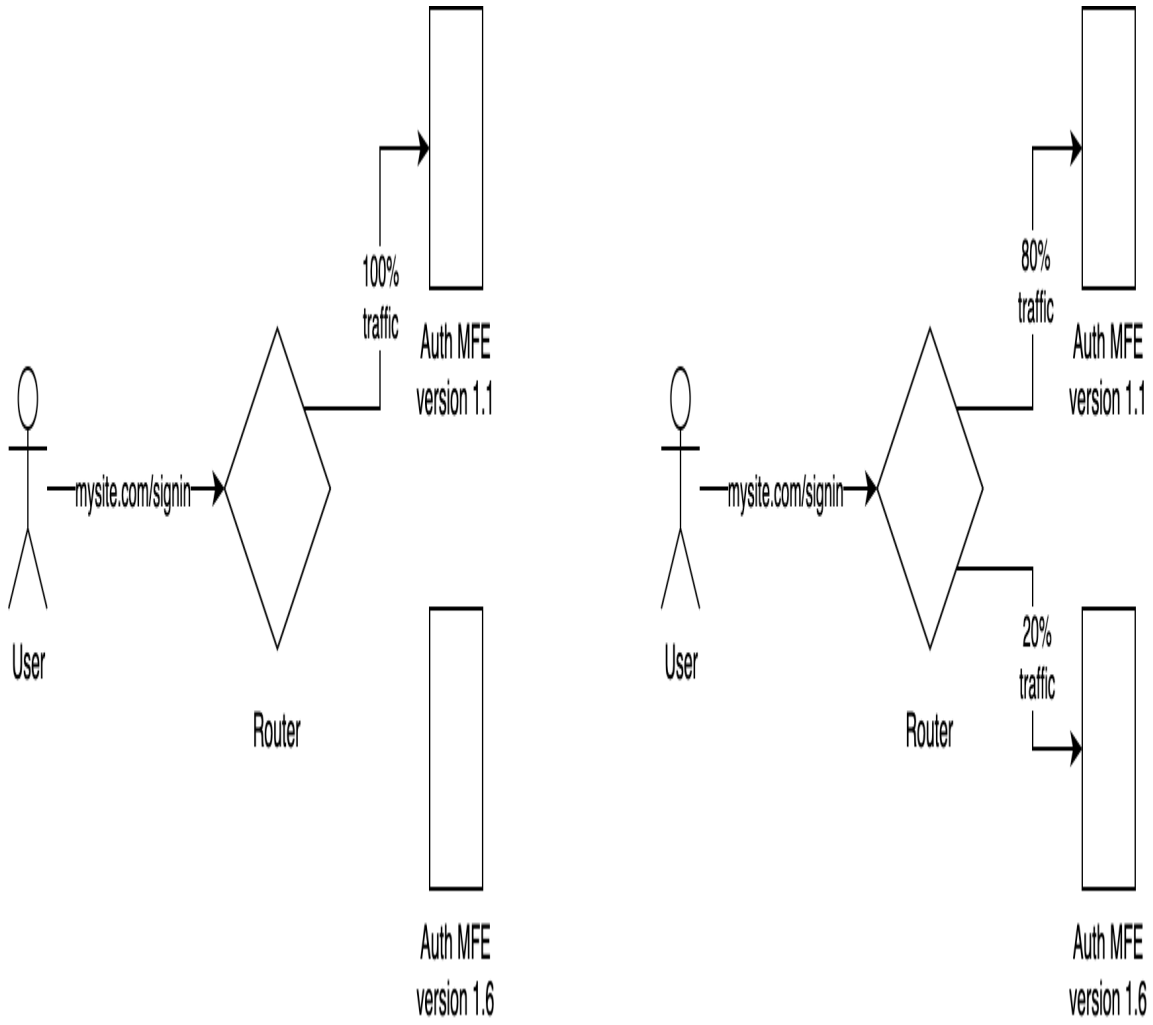


Figure 4-7. 7 Canary release

In both approaches, we need to have a router that shapes the traffic (for a canary release) or switches the traffic from one version to another (blue-green deployment).

The router could be some logic handled on the client side, server side, or edge side, depending on the architecture chosen.

We can summarize the options available in the following table (table 7.1).

	Blue-Green Deployment or Canary Release mechanism
Client-side routing	Application shell Configuration passed via static JSON or backend APIs
Edge-side routing	Logic running at the edge (e.g. AWS Lambda@Edge)
Server-side routing	Application server logic API Gateway Load Balancer

Table 7.1 This table shows the router options available for canary releases and blue-green deployments

Let's explore some scenarios for leveraging these techniques in a micro-frontends architecture.

When we compose our micro-frontends at a client-side level using an application shell, for instance, we may extend the application shell logic, loading a configuration containing the micro-frontends versions available and the percentage of traffic to be presented with a specific version.

For instance, we may want to load a configuration similar to the following example for shaping the traffic, issuing a cookie or storing in web storage the version the user was assigned to and changing it to a different version when we are sure our micro-frontend doesn't contain critical bugs.

```
{
    "homepage":{
        "v.1.1.0": {
            "traffic": 20,
            "url": "acme.com/mfes/homepage-1_1_0.html"
```

```
        },
        "v.1.2.2": {
            "traffic": 80,
            "url": "acme.com/mfes/homepage-1_2_2.html"
        }
    },
    "signin":{
        "v.4.0.0": {
            "traffic": 90,
            "url": "acme.com/mfes/signin-4_0_0.html"
        },
        "v.4.1.5": {
            "traffic": 10,
            "url": "acme.com/mfes/signin-4_1_5.html"
        }
    }
    ...
}
```

As you can see...

For another project, we may decide that introducing a canary release mechanism inside the application shell logic is not worth the effort, moving this logic to the edge using the lambda@edge, which is only available on AWS cloud.

The interesting part of moving the canary release mechanism to the edge is not only that the decision of which version to serve the user is made to the closest AWS region so latency is reduced but also, architecturally speaking, we are decoupling an infrastructure duty from the codebase of our application shell.

LAMBDA@EDGE CANARY RELEASES

During AWS:ReInvent 2019 I had the opportunity to be part of a talk about the implementation done inside DAZN for handling canary releases, strangler pattern, and dynamic rendering.
That talk is available on YouTube if you are interested in the details of how to implement a similar solution using edge computing.

With a horizontal split implementation, where we assemble at runtime different micro-frontends, introducing either blue-green or canary should be performed at the application server level when we compose the page to be served.

We may decide to do it at the client-side level as well. However, as you can imagine, the amount of micro-frontends to handle may matter and mapping all of them may result in a large configuration to be loaded client-side. So we create a system for serving just the configuration needed for a given URL to the client-side.

Other options include releasing different compositions logic and testing them using an API gateway or a load balancer for shaping the traffic toward a server cluster hosting the new implementation and the one hosting the previous version.
In this way, we rely on the infrastructure to handle the logic for canary release or blue-green deployment instead of implementing, and maintaining, logic inside the application server.

As you can see, the concept of the router present in figures 7.6 and 7.7 may be expressed in different ways based on the architecture embraced and the context you are operating in.

Moreover, the context should drive the decision; there may be strong reasons for implementing the canary releases at a different infrastructure layer based on the environment we operate in.

Strangler pattern

Blue-green deployment and canary releases help when we have a micro-frontends architecture deployed in production.
But what if we are scaling an existing web application and introducing micro-frontends?

In this scenario, we have two options: we either wait until the entire application is rewritten with micro-frontends or we can apply the

microservices ecosystem's well-known strangler pattern to our frontend application.

The strangler pattern comes from the idea of generating incremental value for the business and the user by releasing parts of the application instead of waiting for the wholly new application to be ready.

Basically, with micro-frontends, we can tackle an area of the application where we think we may generate value for the business, build with micro-frontends, and deploy them in the production environment living alongside the legacy application.

In this way, we can provide value steadily, while the frequent releases allow you to monitor progress more carefully, drifting toward the right direction for our business and our final implementation.

Using the strangler pattern is very compelling for many businesses, mainly because it allows them to experiment and gather valuable data directly from production without relying solely on projections.

The initial investment for the developers teams is pretty low and can immediately generate benefit for the final user.

Moreover, this approach becomes very useful for the developers for understanding whether the reasoning behind releasing the first micro-frontends was correct or needs to be tweaked, because it forces the team to think about a problem smaller than the entire application and try the approach out end to end, from conception to release, learning along the way which stage they should improve, if any.

As we can see in figure 7.8, when a user requests a page living in the micro-frontends implementation, a router is responsible for serving it. When an area of the application is not yet ready for micro-frontends, the router would redirect the user to the legacy platform.

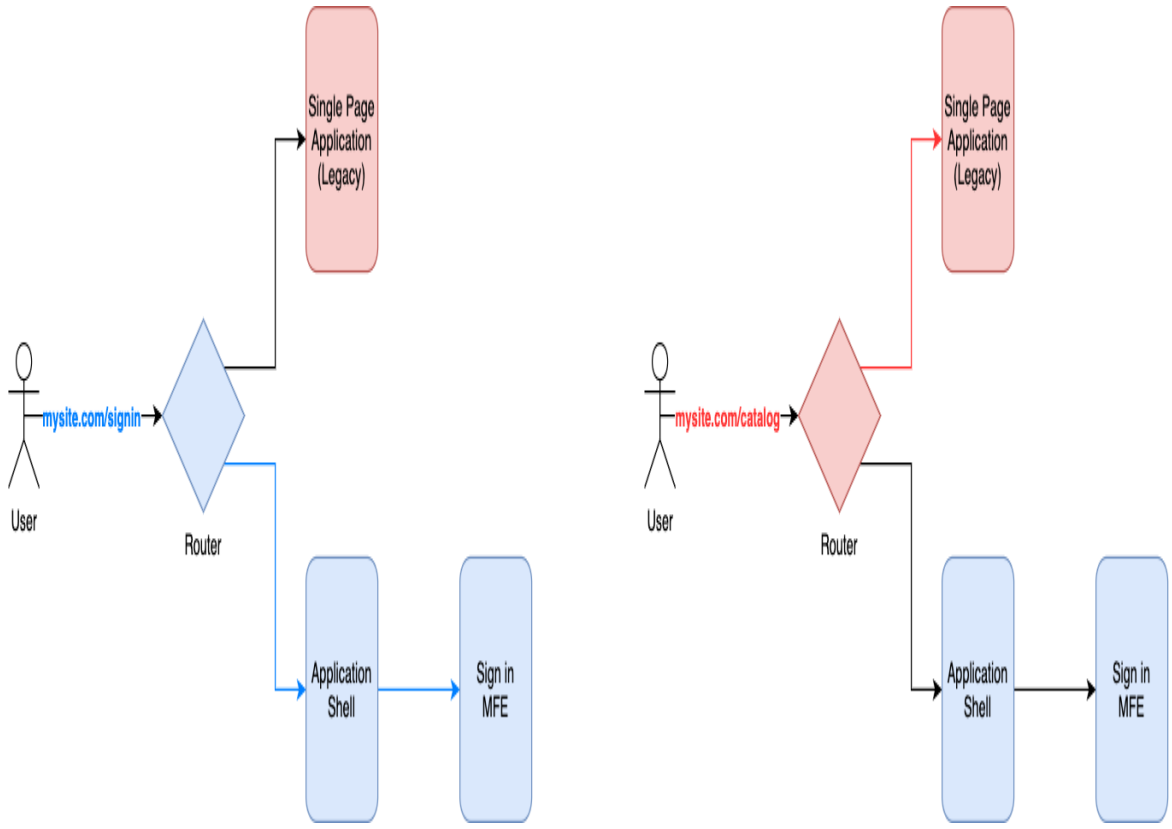


Figure 4-8. 8 A strangler pattern where the micro-frontends live alongside the legacy application so that we can create immediate value for the users and the company instead of waiting for the entire application to be developed.

Every time we develop a new part of the application, it will replace another part of the legacy application until the whole legacy application is completely replaced by the micro-frontends platform.

Implementing the strangler pattern has some challenges, of course. You'll need to make some changes in the legacy application to make this mechanism work properly, particularly when the micro-frontends application isn't living alongside the legacy application infrastructure but may live in a different subdomain.

For instance, the legacy application should be aware that the area covered by the micro-frontends implementation shouldn't be served anymore from its codebase but should redirect the user to an absolute URL so the router logic will kick in again for redirecting the user to the right part of the application.

Another challenge is finding a way to quickly redirect users from micro-frontends to another in case of errors. A technique we used for rolling out our new micro-frontends platform was to maintain three versions of our application for a period of time: the legacy, the legacy modified for co-existing with the micro-frontends (called the hybrid), and the micro-frontends platform. With this approach, we could always serve the hybrid and the micro-frontends platform, and in the extreme case of an issue we weren't able to fix quickly, we were able to redirect all the traffic to the legacy platform.

This configuration was maintained for several months until we were ready with other micro-frontends. During that time, we were able to improve the platform as expected by the business.

In some situations, this may look like an over-engineered solution, but our context didn't allow us to have downtime in production, so we had to find a strategy for providing value for our users as well for the company. The strangler pattern let us explore the risks our company was comfortable taking and then, after analyzing them, design the right implementation.

Observability

The last important part to take into consideration in a successful micro-frontends 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 beable 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 or LogRocket, these tools allow us to individuate the user journey before encountering a bug that may or may not prevent the user to complete its action.

Observability it's not a nice to have feature, nowadays it should be part of any releasing strategy, even more important when we are implementing a micro-frontends architecture.

Every micro-frontend should report errors, custom and generic, for providing visibility when a live issue happens.

In that regard, Sentry or LogRocket can help in this task providing the visibility needed, in fact, these tools are retrieving the user journey, collecting the JavaScript stack trace of an exception, and clustering into groups.

We can configure the alerting of every type of error or warning in these tools dashboard 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.

After building our artifact and performing some additional quality reviews, we are ready to deploy our micro-frontends. We talked about testing the final results in production using canary or blue-green deployment to reduce the risk of presenting bugs to the users and releasing as quickly and as often as possible without fear of breaking the entire application.

Finally, we discussed using the strangler pattern when we have an existing application and want to provide immediate value to our business and users. Such a pattern will steadily reduce the functionalities served to the user by a legacy application and increase the one in our micro-frontends platform.

Chapter 5. 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 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *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 or modular monolith backend, 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.

Let's start by defining the different APIs approaches we may have in a web application. As shown in figure 9.1, 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 or hypermedia, for instance, by learning how to deal with BFF, API gateway, or service dictionary patterns.

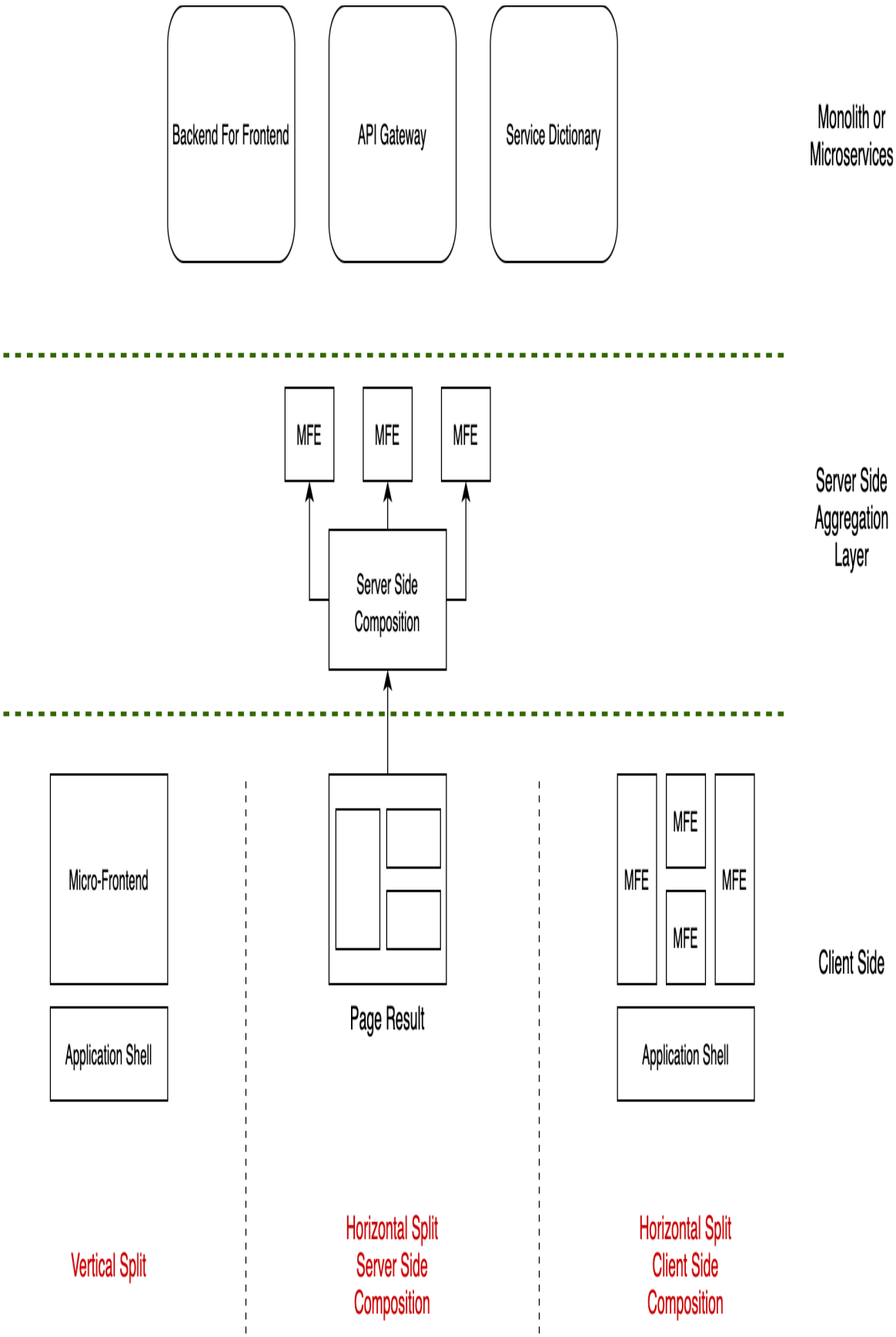

Backend For Frontend
API Gateway
Service Dictionary
Monolith or Microservices
MFE
MFE
MFE
Server Side Composition
Server Side Aggregation Layer
Micro-Frontend
Application Shell
Page Result
MFE
MFE
MFE
MFE
Application Shell
Client Side
Vertical Split
Horizontal Split Server Side Composition
Horizontal Split Client Side Composition

Figure 5-1. 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, API throttling, or rate-limiting.

- **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.

Domain-driven design (DDD) also influences architectures and infrastructure decisions. Especially with micro-architectures, we can divide an application into multiple business domains, using the right approach for each business domain.

For instance, it's not unusual to have part of the application exposing the APIs with a BBF pattern and another part exposing with a service dictionary.
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.

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-platforms 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 penetration 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 (mobile, smart TVs, or consoles)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.

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 helps introduce the discussion with the team responsible for the API, especially 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 9.2).

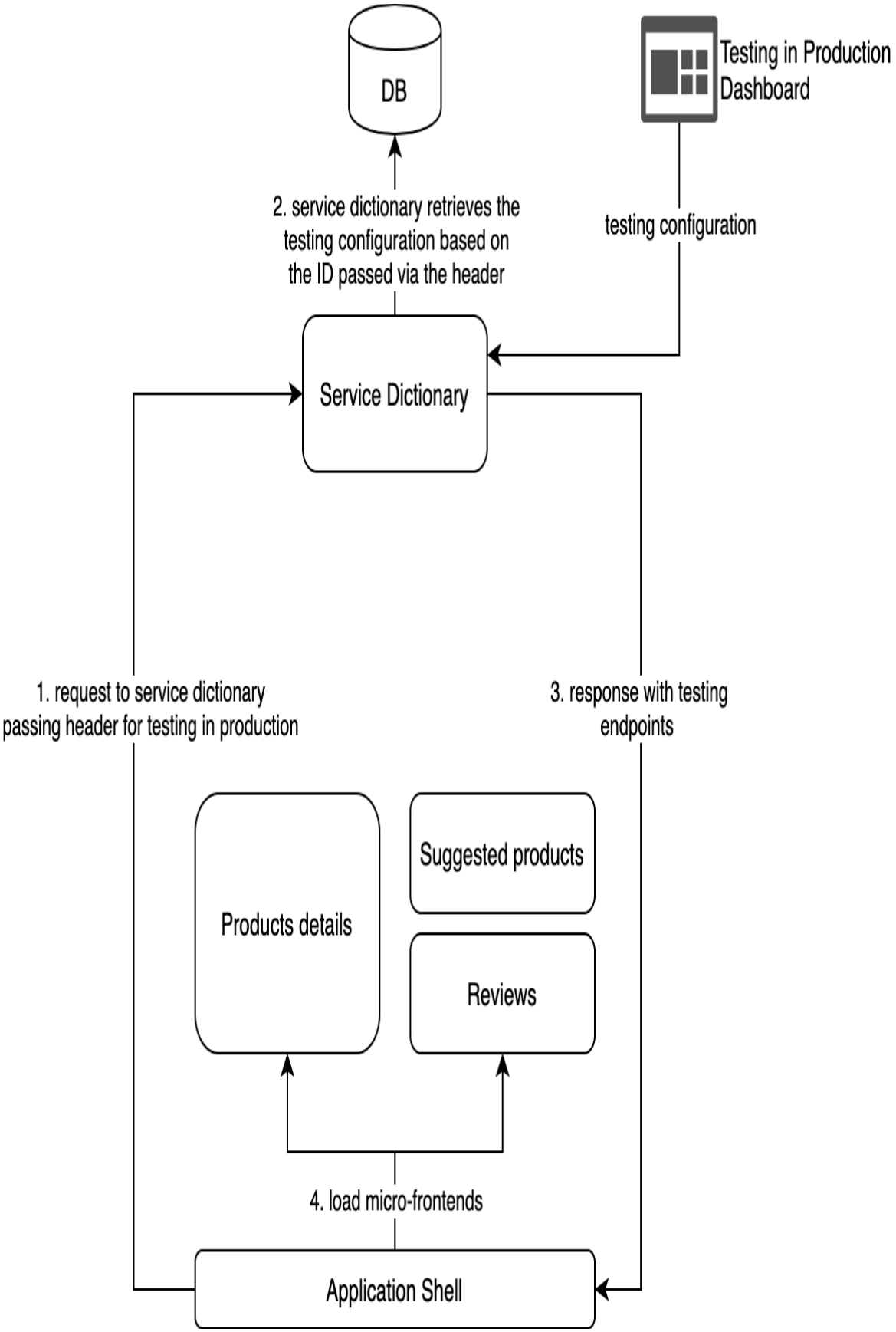

DB
Testing in Production Dashboard
2. service dictionary retrieves the testing configuration based on the ID passed via the header
testing configuration
Service Dictionary
1. request to service dictionary passing header for testing in production
3. response with testing endpoints
Products details
Suggested products
Reviews
4. load micro-frontends
Application Shell

Figure 5-2. 2 - A high-level architecture on how to use a service dictionary for testing in production

In figure 9.2 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 to load.

In this example, the ID was generated at runtime by the application shell.

When the service dictionary receives the call, it will check whether a header is present in the request and if so, it will try to load the associated configuration stored inside 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 till 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 9.3.

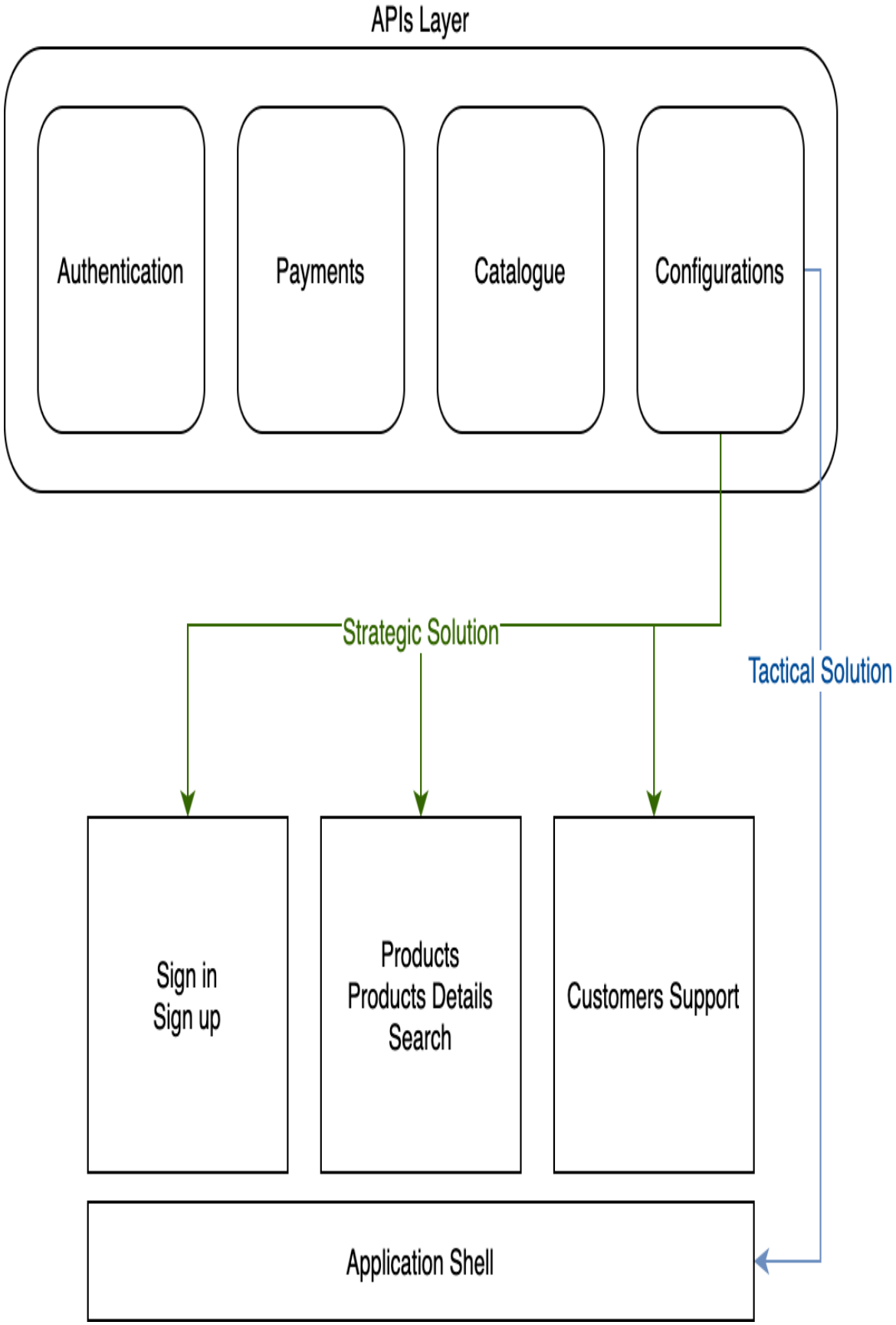

APIs Layer
Authentication
Payments
Catalogue
Configurations
Strategic Solution
Tactical Solution
Sign in
Sign up
Products
Products Details
Search
Customers Support
Application Shell

Figure 5-3. 3 With vertical-split architecture we can retrieve the service dictionary directly inside a micro-frontend, dividing the endpoints list by business domain.

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 9.3, we can derive a sample of sequence diagrams like the one in figure 9.4. 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.

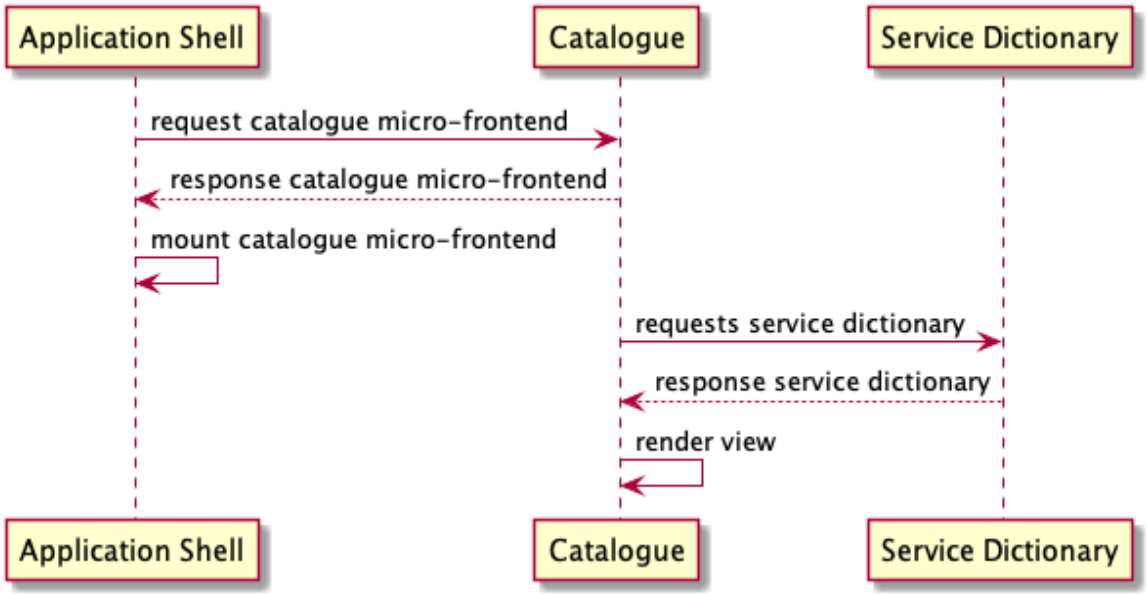


Figure 5-4. 4 Sequence diagram to implement a service dictionary with a vertical-split architecture

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 9.5.

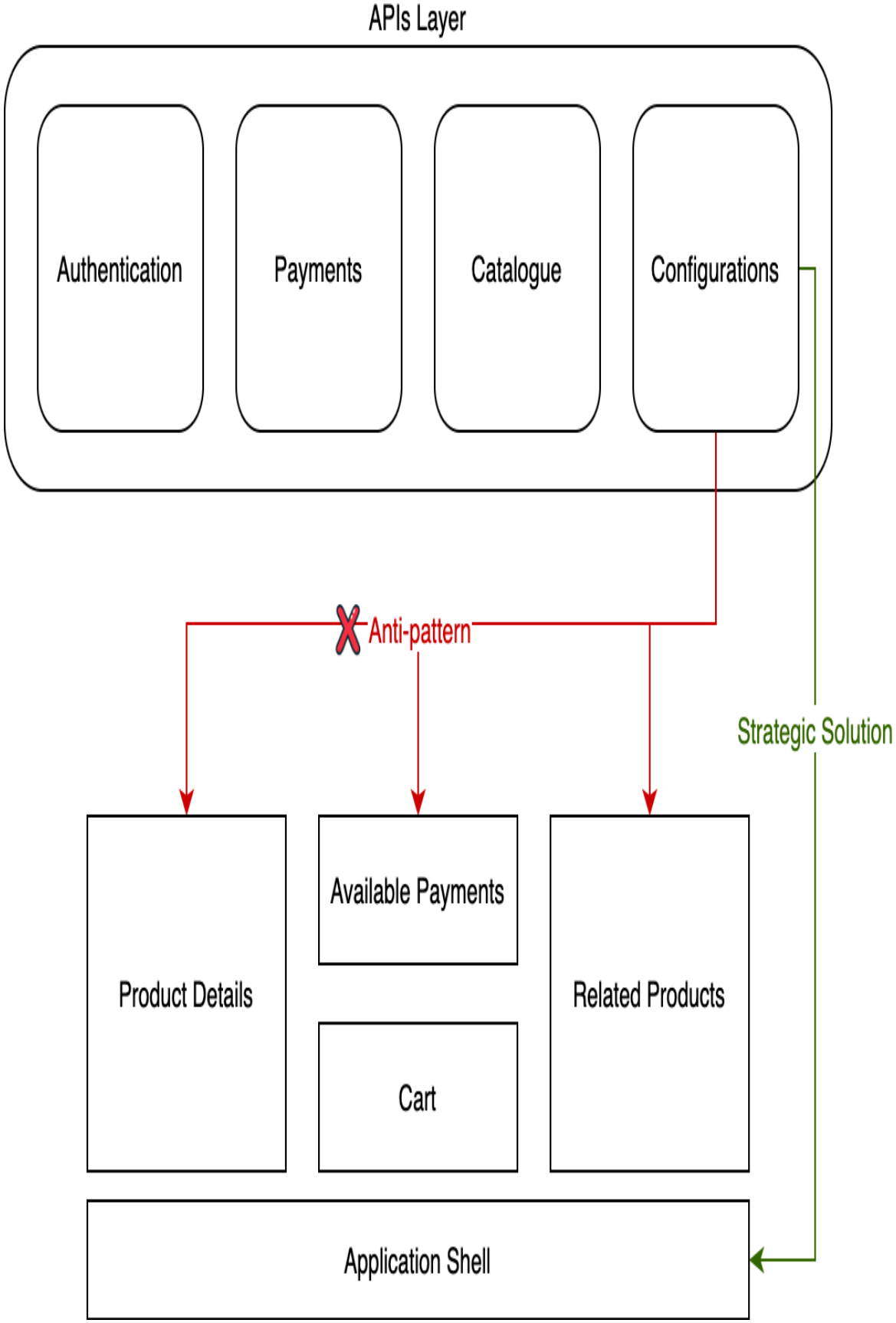

APIs Layer
Authentication
Payments
Catalogue
Configurations
Anti-pattern
Strategic Solution
Product Details
Available Payments
Cart
Related Products
Application Shell

Figure 5-5. 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 9.6.

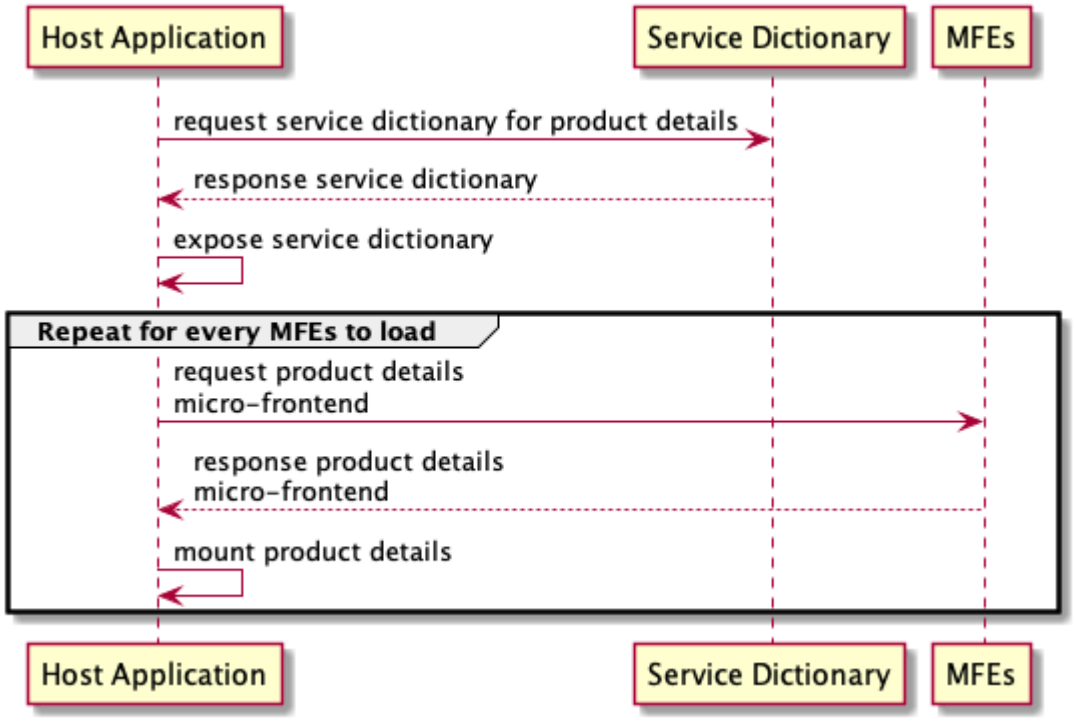


Figure 5-6. 6 - This sequence diagram shows how a horizontal-split architecture with client-side composition may consume the service dictionary API.

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, we may have a more complex situation in reality. 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, like authentication and rate limiting.

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 9.7).

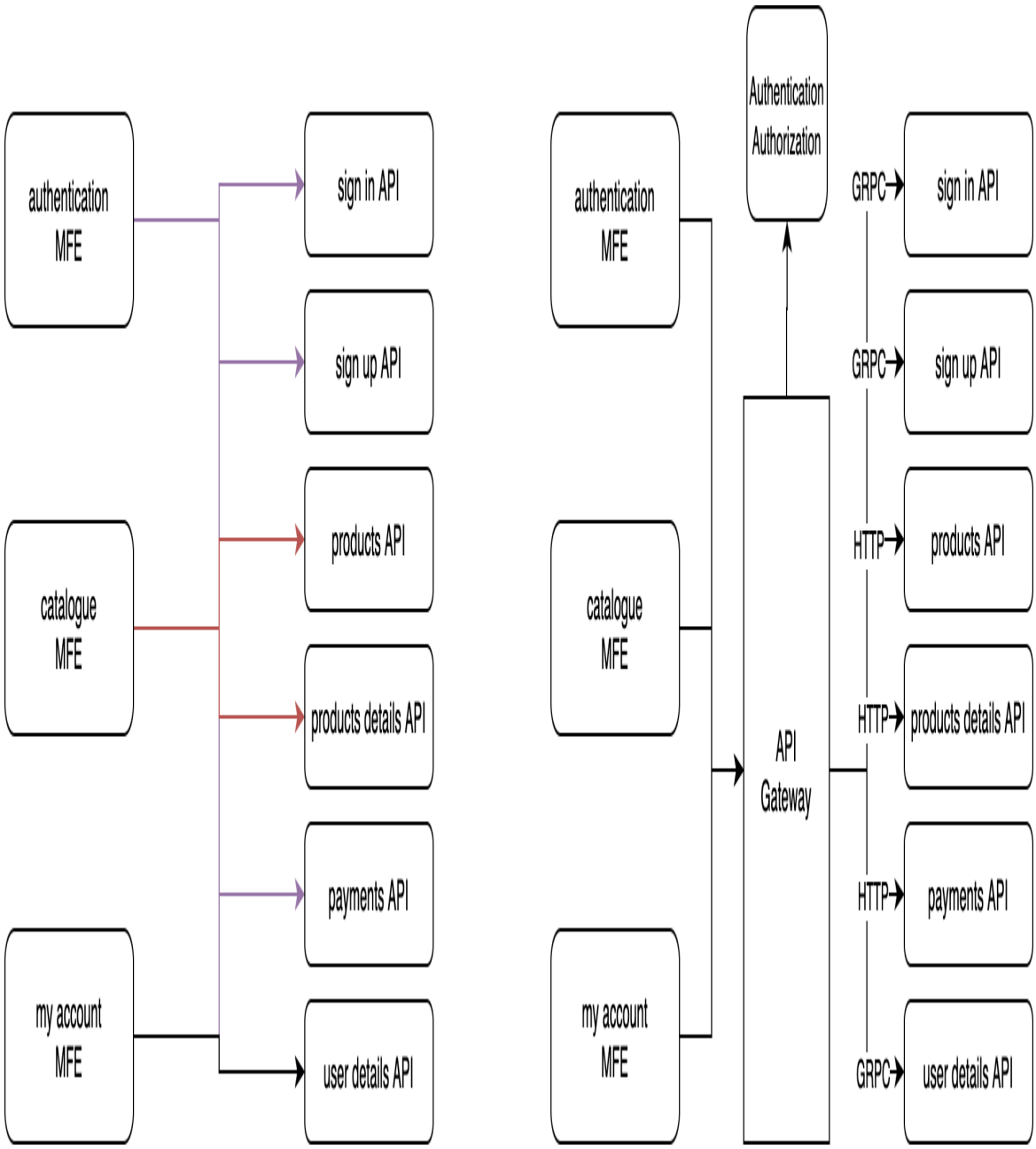


Figure 5-7. 7 - An API gateway pattern simplifies the communication between clients and server and centralizes functionalities like authentication and authorization via edge functions.

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.

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 or removing APIs in the API gateway to prevent .

Finally, if we implement an additional layer between the client and the microservice to consume, we'll add some latency to the system.

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 9.8.

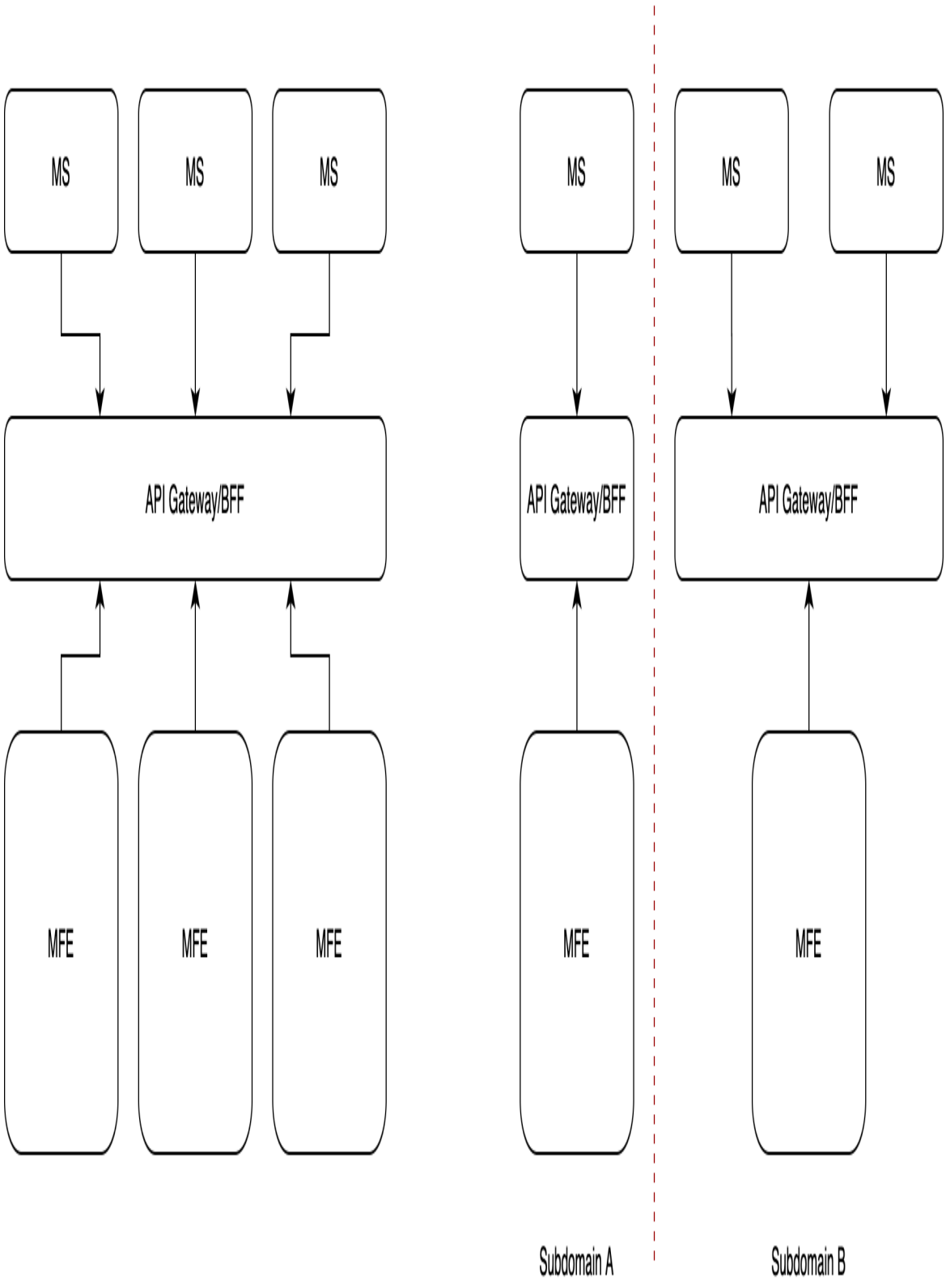


Figure 5-8. 8 - On the left is a unique entry point for the API layer; on the right are multiple entry points, one per subdomain.

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.

This approach may require more work to build, but it allows a fine-grain 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 9.9.

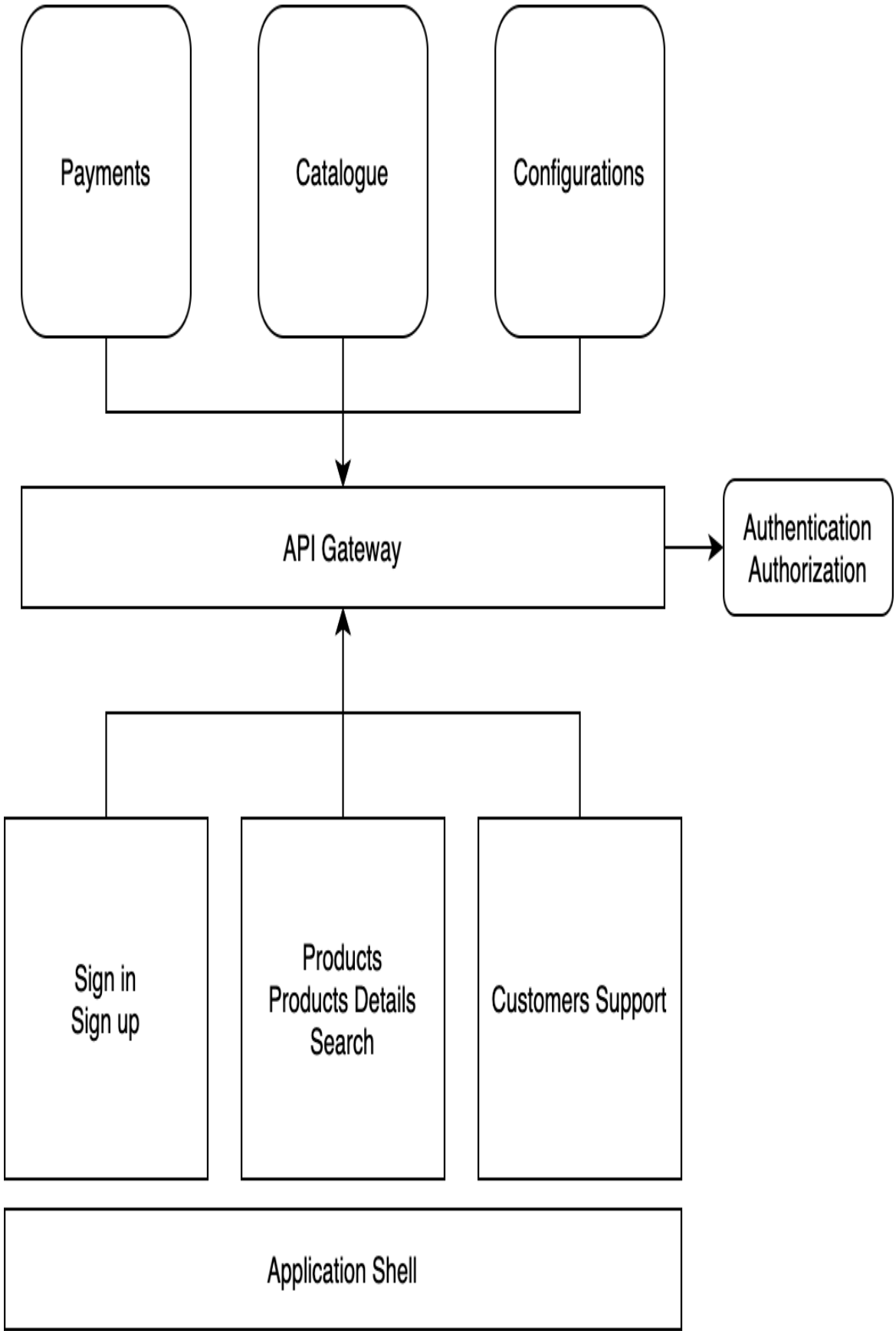

Payments
Catalogue
Configurations
API Gateway
Authentication
Authorization
Sign in
Sign up
Products
Products Details
Search
Customers Support
Application Shell

Figure 5-9. 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.

> **NOTE**
>
> Remember that with a server-side composition we identify our micro-frontends with a horizontal split, not a vertical one

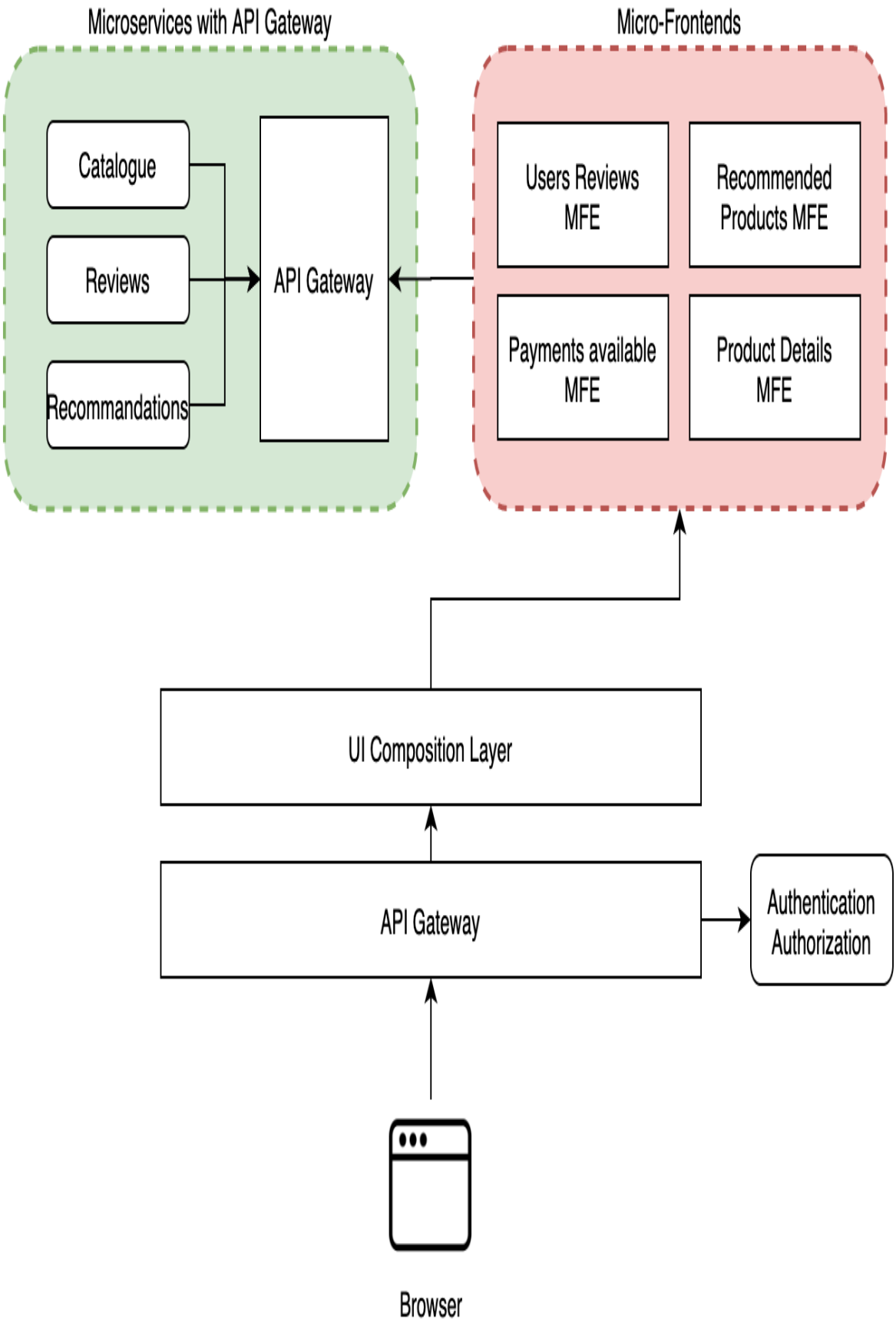

Microservices with API Gateway
Micro-Frontends
Catalogue
Reviews
Recommandations
API Gateway
Users Reviews MFE
Recommended Products MFE
Payments available MFE
Product Details MFE
UI Composition Layer
API Gateway
Authentication Authorization
Browser

Figure 5-10. 10 - An example of a server-side composition with a microservices architecture

As we can see in figure 9.10, 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, with their relative content fetched from the microservices layer.

For the microservices layer, we use a second API gateway to expose the API for internal services, in this case, used by the UI composition service.

Figure 5-20.11 illustrates a hypothetical implementation with the sequence diagram related to this scenario.

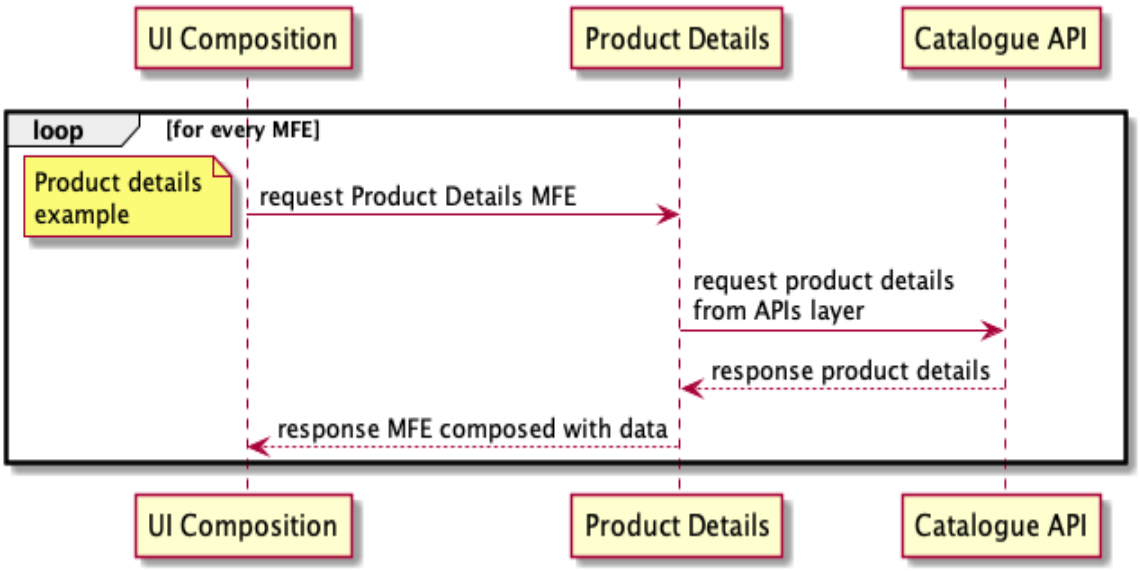


Figure 5-11. 11 - An example of server-side composition with API gateway

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 microrservice.

There are several micro-frontend frameworks with a similar implementation, such as the interface framework from Zalando, OpenComponents, Project Mosaic, and Ara Framework based on Airbnb Hypernova.

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?

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.

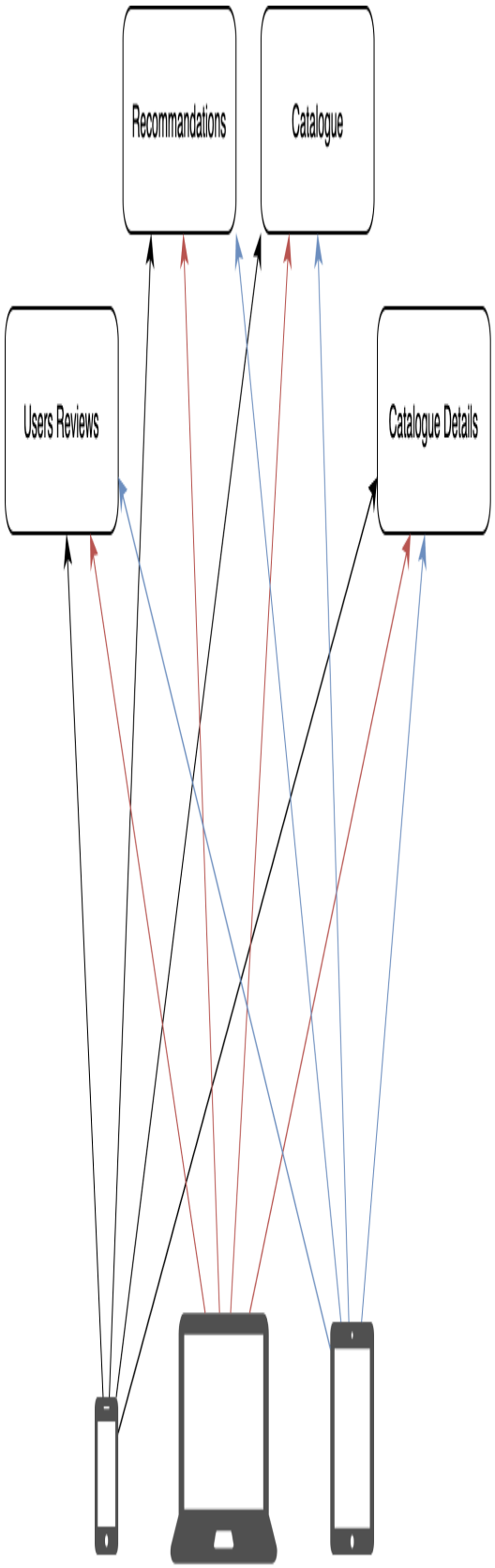
Finally, mobile applications often require a different method for aggregating data and exposing them in a meaningful way to the user. APIs on the backend are the same for all clients, so for mobile applications, we need to consume different endpoints and compute the final result on the device instead of changing the API responses based on the device that consumes the endpoint.

In all these cases, BFF, as described by Phil Calçado (formerly 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 9.12.


Recommandations
Catalogue
Users Reviews
Catalogue Details

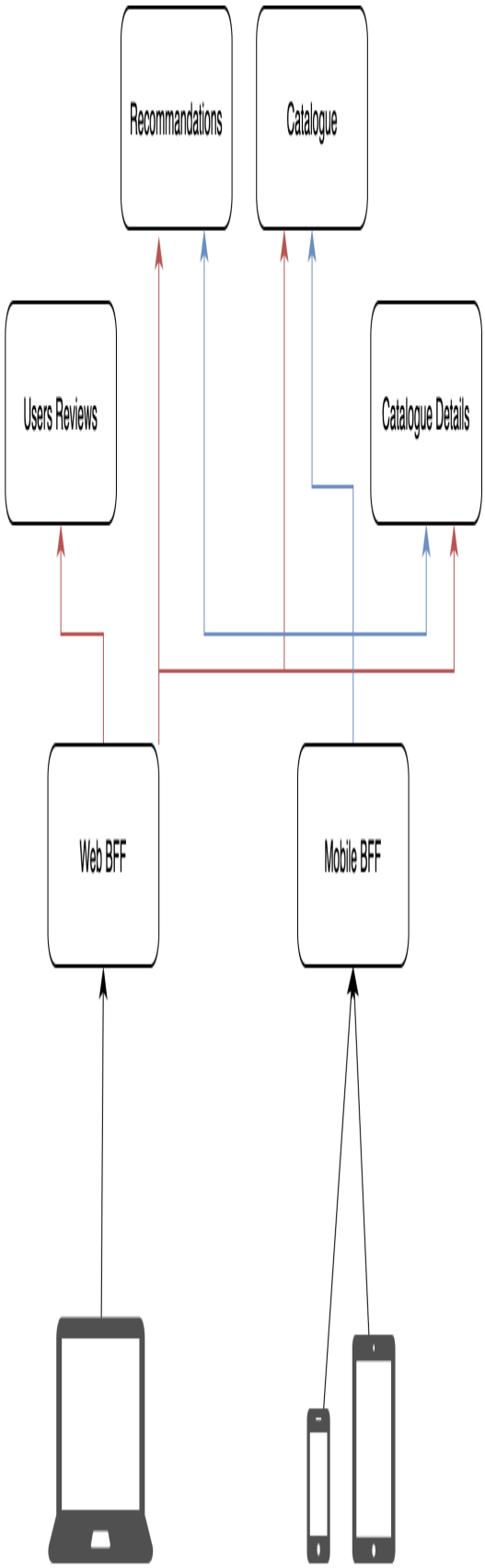

Recommandations
Catalogue
Users Reviews
Catalogue Details
Web BFF
Mobile BFF

Figure 5-12. 12 - On the left a microservices architecture consumed by different clients; on the right a BBF 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 9.13. This technique is also applicable to the API gateway pattern.

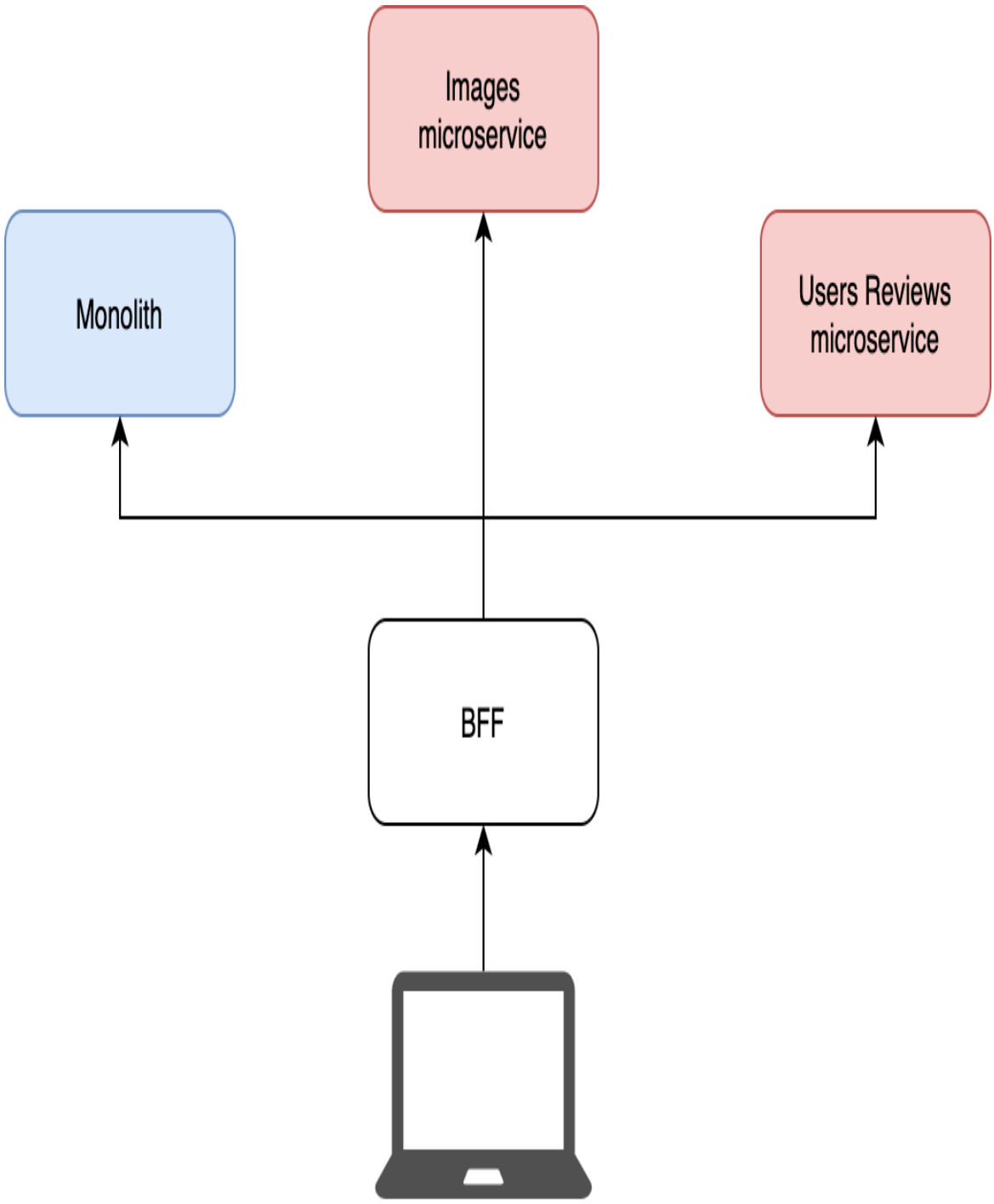


Figure 5-13. 13 - The red boxes represent services extracted from the monolith and converted to microservices. The BFF layer allows the client to be unaware of the change happening in the backend, maintaining the same contract at the BFF level.

Another interesting use case for the BFF is aggregating APIs by domain, as we have seen for the API gateway.

Creating a BFF for a group of devices could have multiple BFF calling the same microservices. When not controlled properly, this can harm platform stability. Obviously, we may decide to introduce caches in different layers for mitigating traffic spikes, but we can mitigate this problem another way.

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 have similar service-level agreements (SLAs) inside the same domain, 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 5-20.14 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.

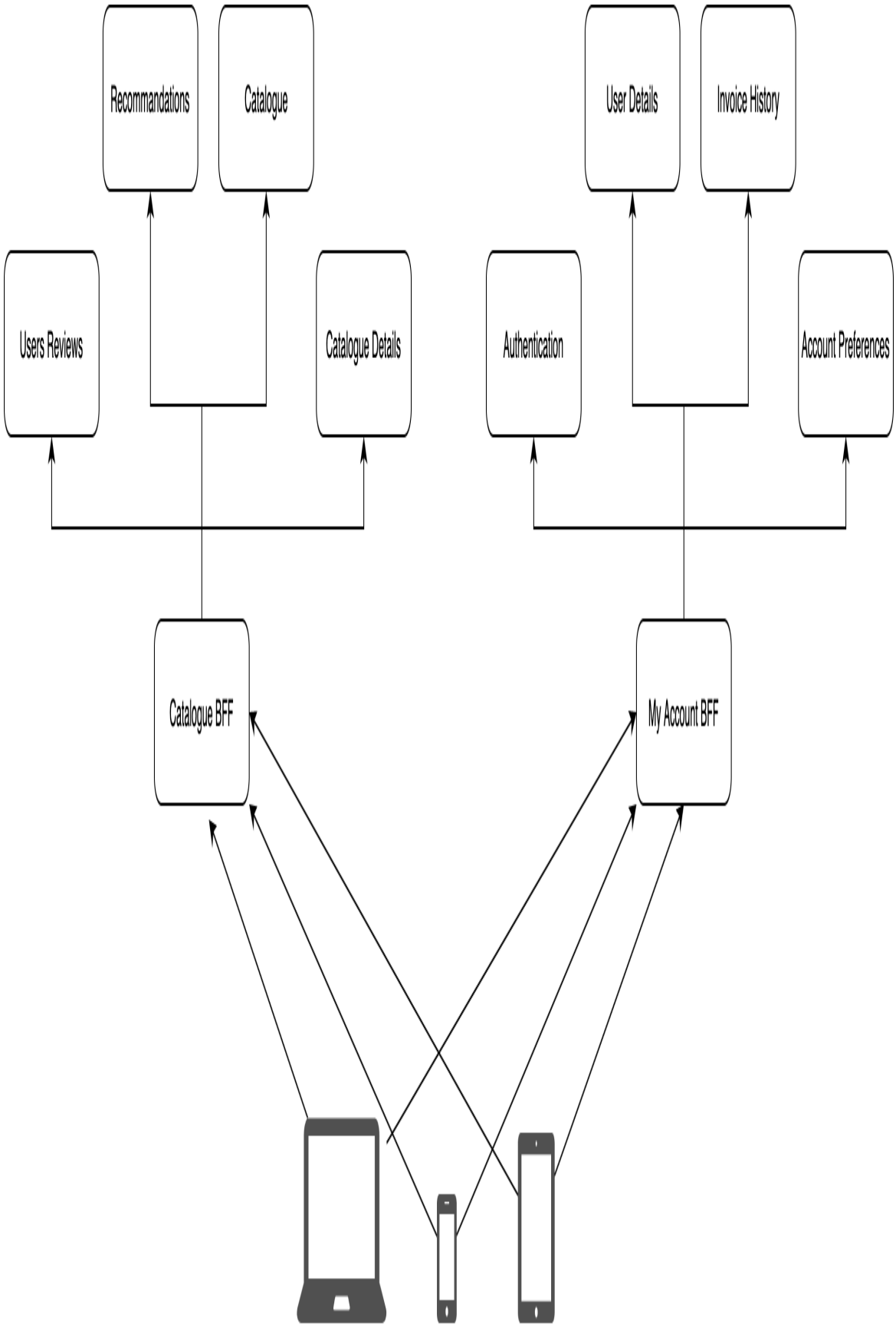

Recommandations
Catalogue
User Details
Invoice History
Users Reviews
Catalogue Details
Authentication
Account Preferences
Catalogue BFF
My Account BFF

Figure 5-14. 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, would allow us to apply the right SLA instead of generalizing one for the entire BFF layer.

Sometimes BFF raises some concerns due to some inherent pitfalls such as reusability and code duplication.

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.

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 9.15, we have a product details page that has to fetch the data for composing the view.

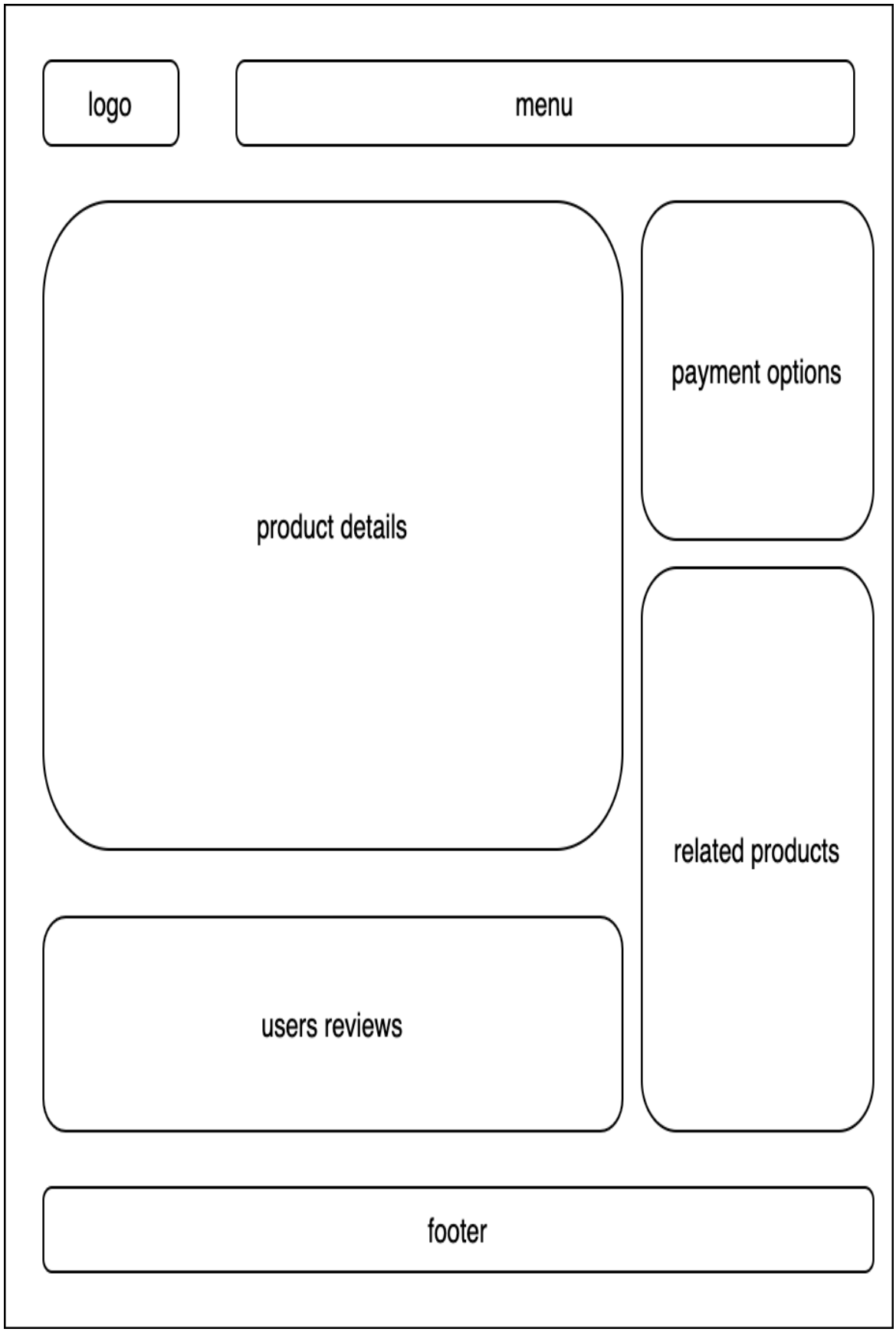

logo
menu
product details
payment options
related products
users reviews
footer

Figure 5-15. 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 9.16.

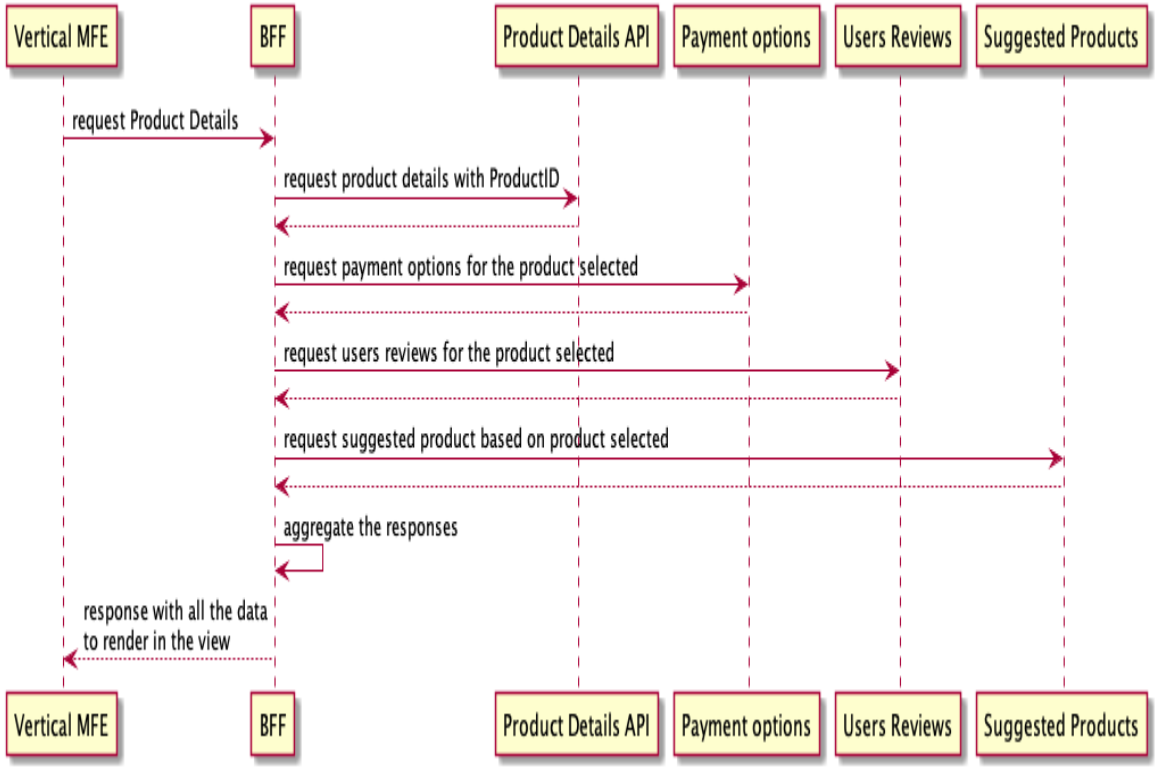


Figure 5-16. 16 - Sequence diagram showing the benefits of the BFF pattern used in combination with a vertical split composed on the client side

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.

Taking into consideration these micro-frontends and the host page may be developed by different teams, this setup would slow down features development rather than leveraging the benefits that this architecture provides us.

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.

Because GraphQL is a client-centric API, the paradigm for designing an API schema 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 an API gateway, acting as a proxy for specific microservices, for instance, or as a BFF, 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.

GraphQL simplifies data retrieval for the clients, allows us to query only the fields needed in a view based on client type (e.g., mobile or web), and simplifies the maintenance and evolution of the GraphQL layer compared to more complicated backend ecosystems.

The data graph is reachable via a unique endpoint. When a new microservice is added to the graph, the only change for the client to make would be at the query level, also minimizing maintenance.

The schema federation

Schema federation is a set of tools to compose multiple GraphQL schemas 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 a 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 5-20.17 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.

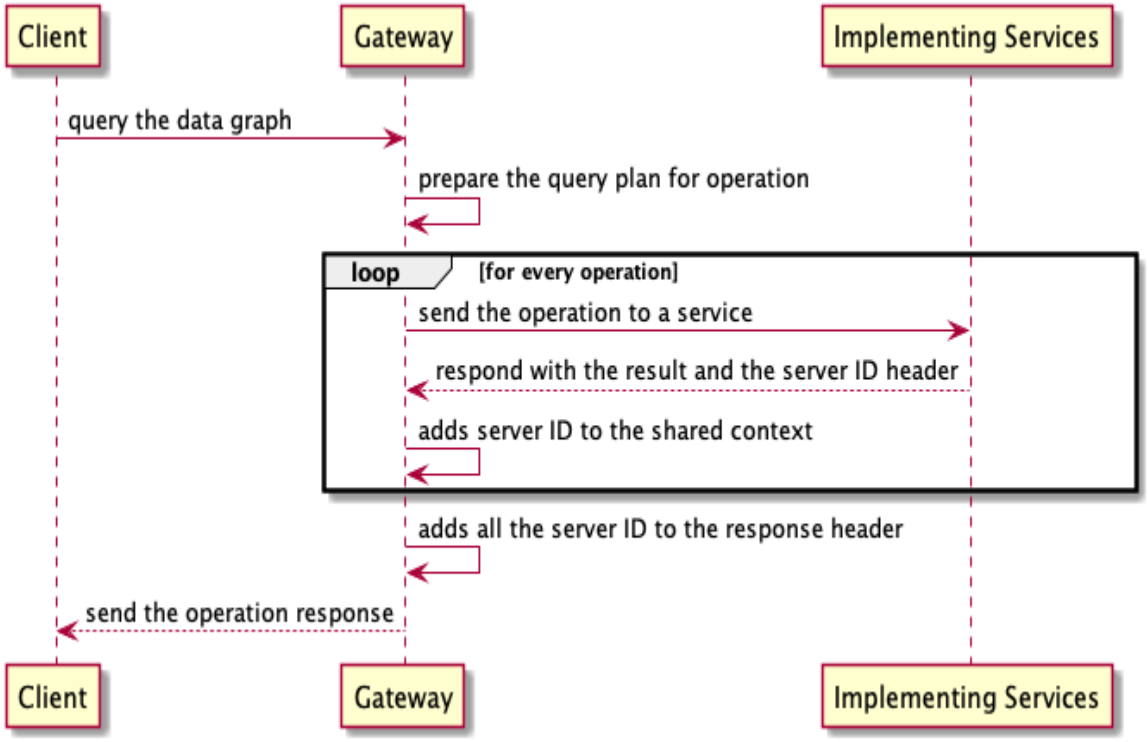


Figure 5-17. 17 - A sequence diagram showing how schema federation exposes all the schemas from multiple services

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 9.18.

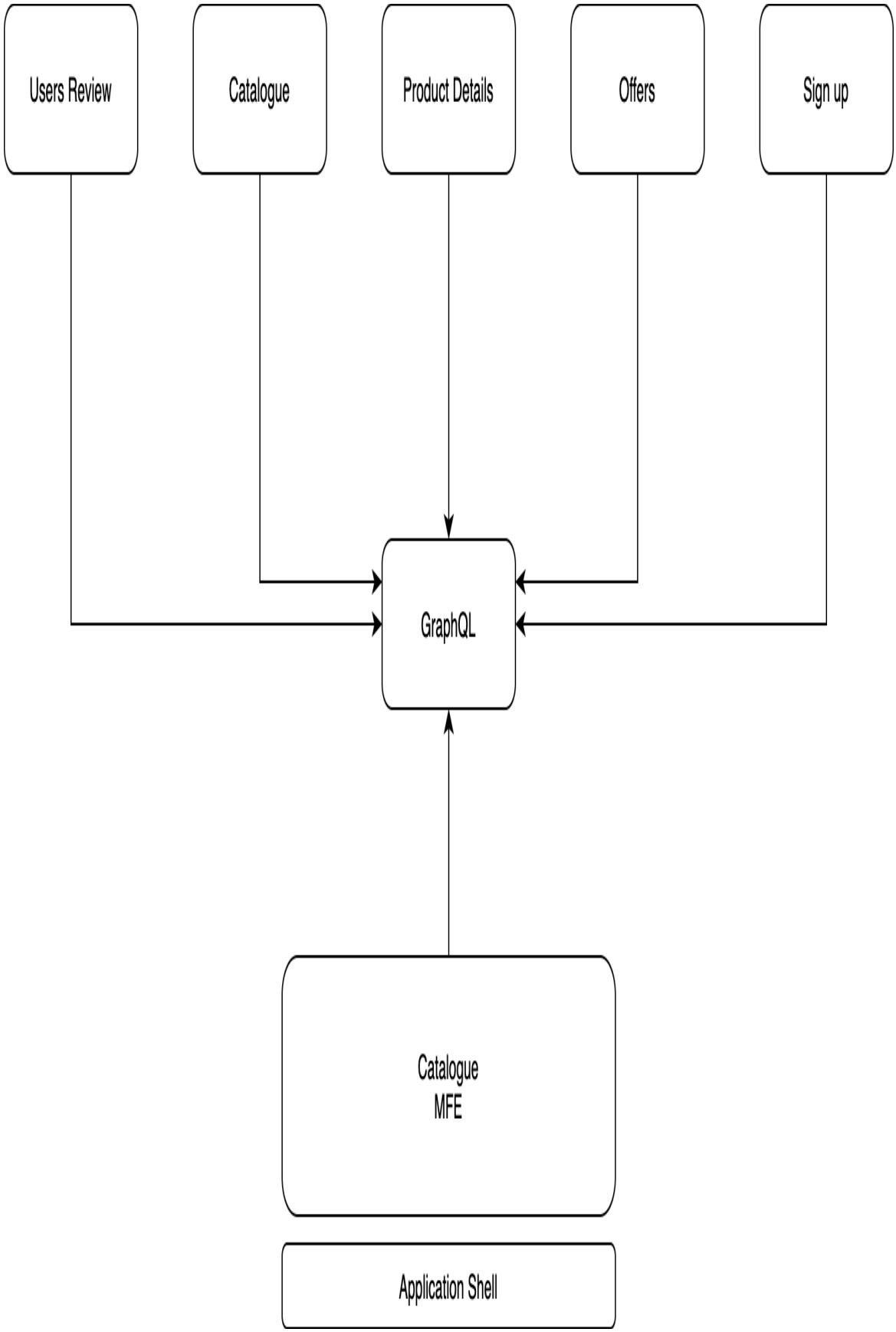

Users Review
Catalogue
Product Details
Offers
Sign up
GraphQL
Catalogue
MFE
Application Shell

Figure 5-18. 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 unique data graph 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 9.19, 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.

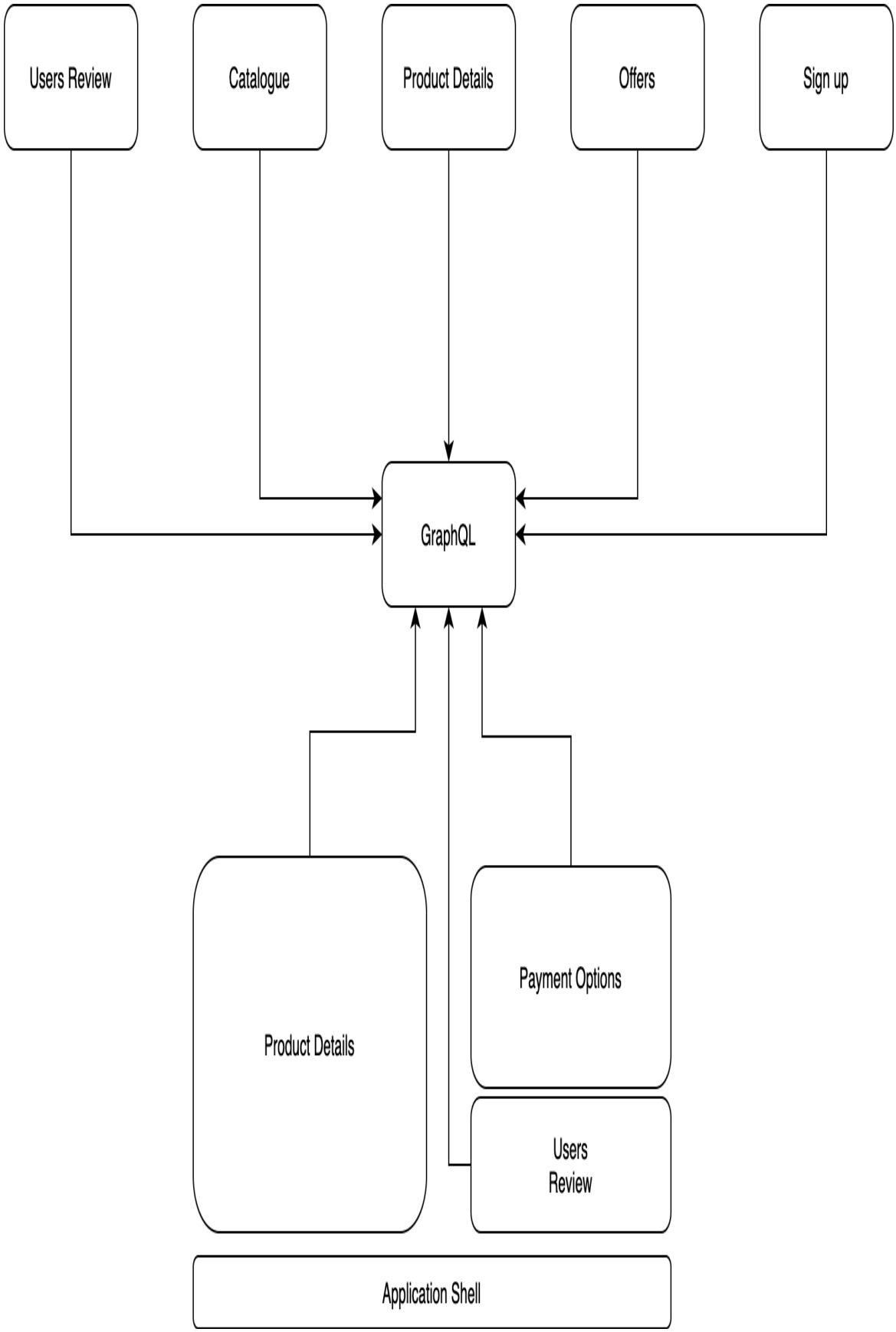

Users Review
Catalogue
Product Details
Offers
Sign up
GraphQL
Product Details
Payment Options
Users
Review
Application Shell

Figure 5-19. 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.

Using GraphQL with micro-frontends and a server-side composition

The last approach is using a GraphQL server with a micro-frontends architecture with horizontal split and a 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 9.20).

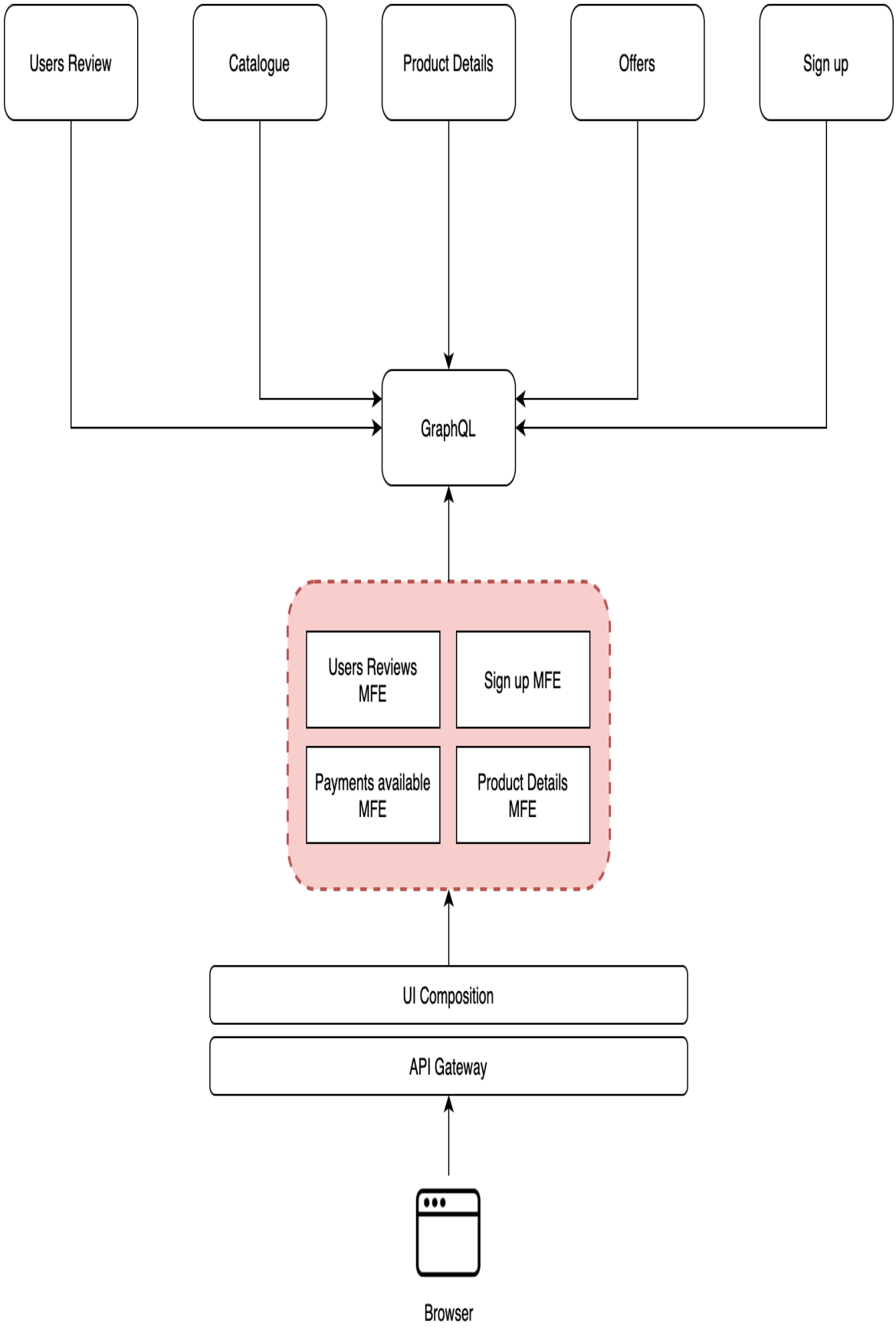

Users Review
Catalogue
Product Details
Offers
Sign up
GraphQL
Users Reviews MFE
Sign up MFE
Payments available MFE
Product Details MFE
UI Composition
API Gateway
Browser

Figure 5-20. 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 we work with a horizontal-split architecture, we may end up having similar micro-frontends in the same view consuming the same APIs.

In this case, we should challenge ourselves to determine whether maintaining separate micro-frontends brings any value to our system. Would grouping them in a unique micro-frontend be better?

Usually, such scenarios should indicate a potential architectural improvement. Don’t ignore that signal; instead, try to revisit the decision made at the beginning of the project with the information and the context available, making sure performing the same API request twice inside the same view is acceptable. If not, be prepared to review the micro-frontends boundaries.

APIs come first, then the implementation

Independent 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-fronted 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 N 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.

Designing APIs for cross-platform applications

Nowadays we are developing cross-platform applications more often than not.
Mobile devices are part of our routine. They help us accomplish our daily tasks and a tablet may have already replaced our laptop for working.

When we approach a cross-platform application and we aren't using a BFF layer to aggregate the data model for every device we target, we need to remember a simple rule: move the configurations as much as you can on the API layer.

With this approach, we will be able to abstract and control certain behaviors without the need to build a new release of our mobile application and wait for the penetration in the market.

For example, let's say you need to create a polling strategy for consuming an API and react to the response every few minutes. Usually, we would just define the interval in the client application. However, in some use cases, this implementation may become risky, such as when you have very bursty traffic and you want to create a mechanism to back off your requests to the server instead of throttling or slowing down the communication between server and client.
In this case, moving the interval value to the body response of the API to

pull would allow you to manage situations like that without distributing a new version of the mobile application.

This also applies to micro-frontends, where we may have multiple micro-frontends that should implement similar logic. Instead of implementing inside the client-side code, consider moving some configurations on the server and implementing the logic for reacting to the server response.

In this way, we will be able to solve many headaches that may happen in production and that affects our users with a simple and strategic decision.

Summary

We have covered how micro-frontends can be integrated with multiple API layers.
Micro-frontends are suitable with 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 BBF 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, leveraging DDD at the infrastructure level for using the right technical approach for a subdomain, and designing APIs for cross-platform applications by moving some logic to the backend instead of replicating into multiple frontend applications.

Chapter 6. Automation Pipeline for Micro-Frontends: A Use Case

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 7th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *Building.microfrontends@gmail.com*.

Now that we've discussed the theory of a micro-frontends automation pipeline, let's review a use case, including the different steps that should be taken into consideration based on the topics we covered. 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. 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, 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 people and techies, requested 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 decided 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, it was decided to defer this decision for having less moving parts to take care of, so manual and automating 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 environments strategy present in the company is composed of three environments: development (DEV), staging (STAG), and production

(PROD), as we can see in figure 8.1.

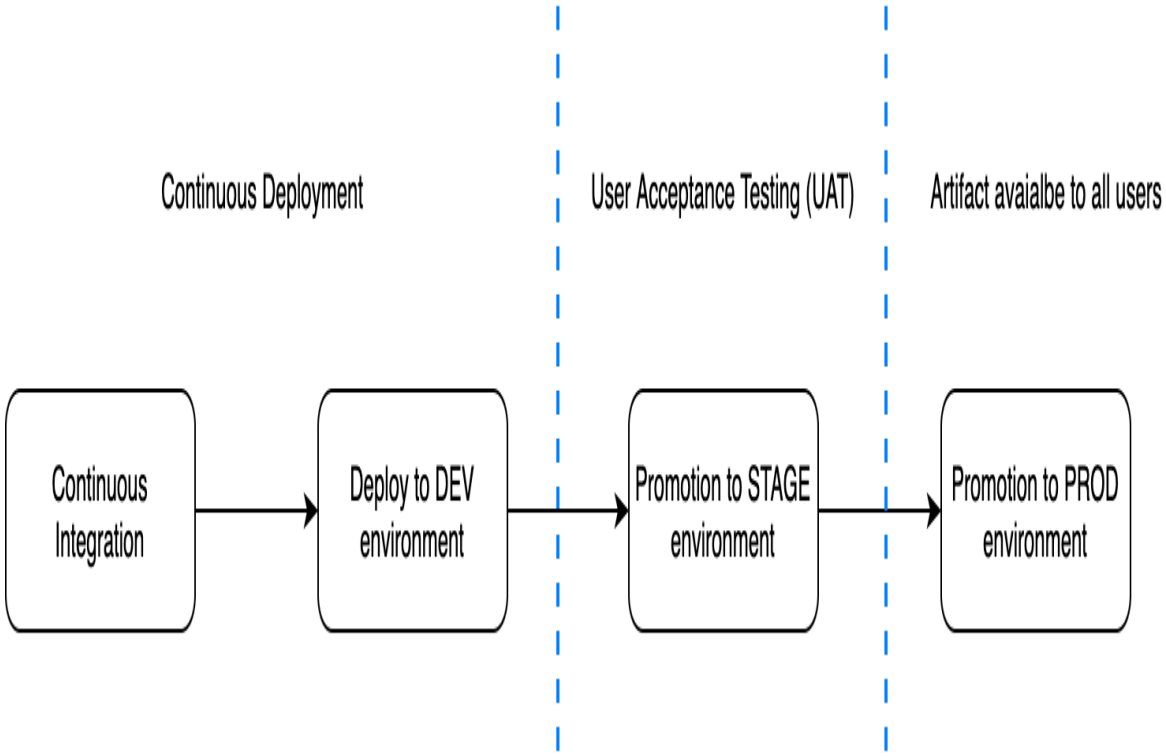


Figure 6-1. 1 - An example of an environments strategy

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 8.2 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.

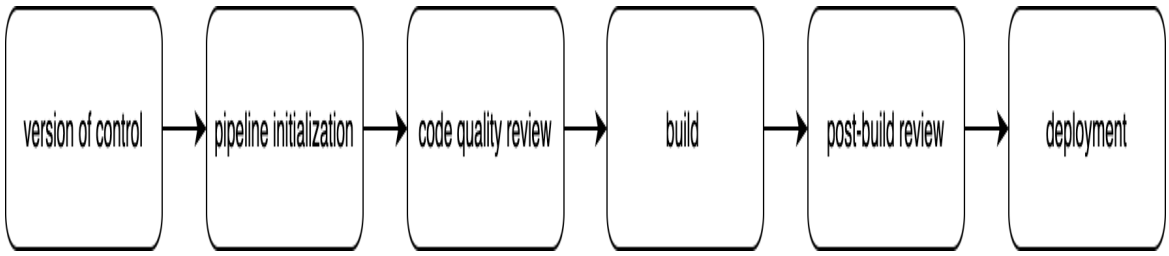


Figure 6-2. 2 - High-level automation strategy design

A dashboard built in house will promote artifacts across environments. In this way, the developers and QAs 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 brain; 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 entire logic and approach their team took 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 of control
2. Pipeline initialization
3. Code-quality review
4. Build
5. Post-build review
6. Deployment

Let's explore these areas in detail.

Version of Control

The project will use monorepo for version of 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 of control, partially because there are always interesting automation opportunities in a cloud-based version of control like GitHub.
In fact, GitHub has a marketplace with many scripts available to be run at different branching lifecycles. 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.

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 8.3 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.

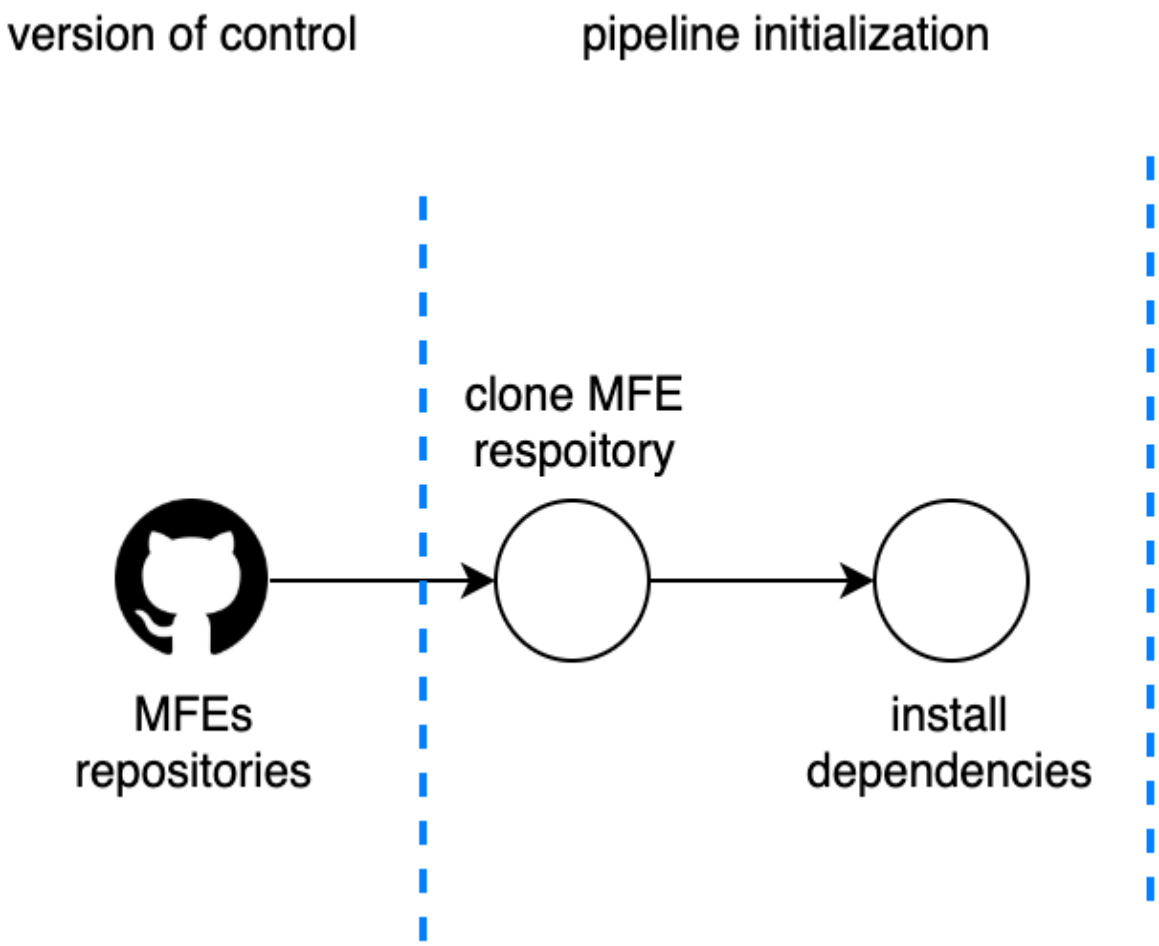


Figure 6-3. 3 - Pipeline initialization stage, showing two actions: cloning the repository and installing the dependencies

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. The cloning operation will speed up in particular when we are dealing with repositories with years of history tracked in the version of control.

```
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 6-6.4 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.

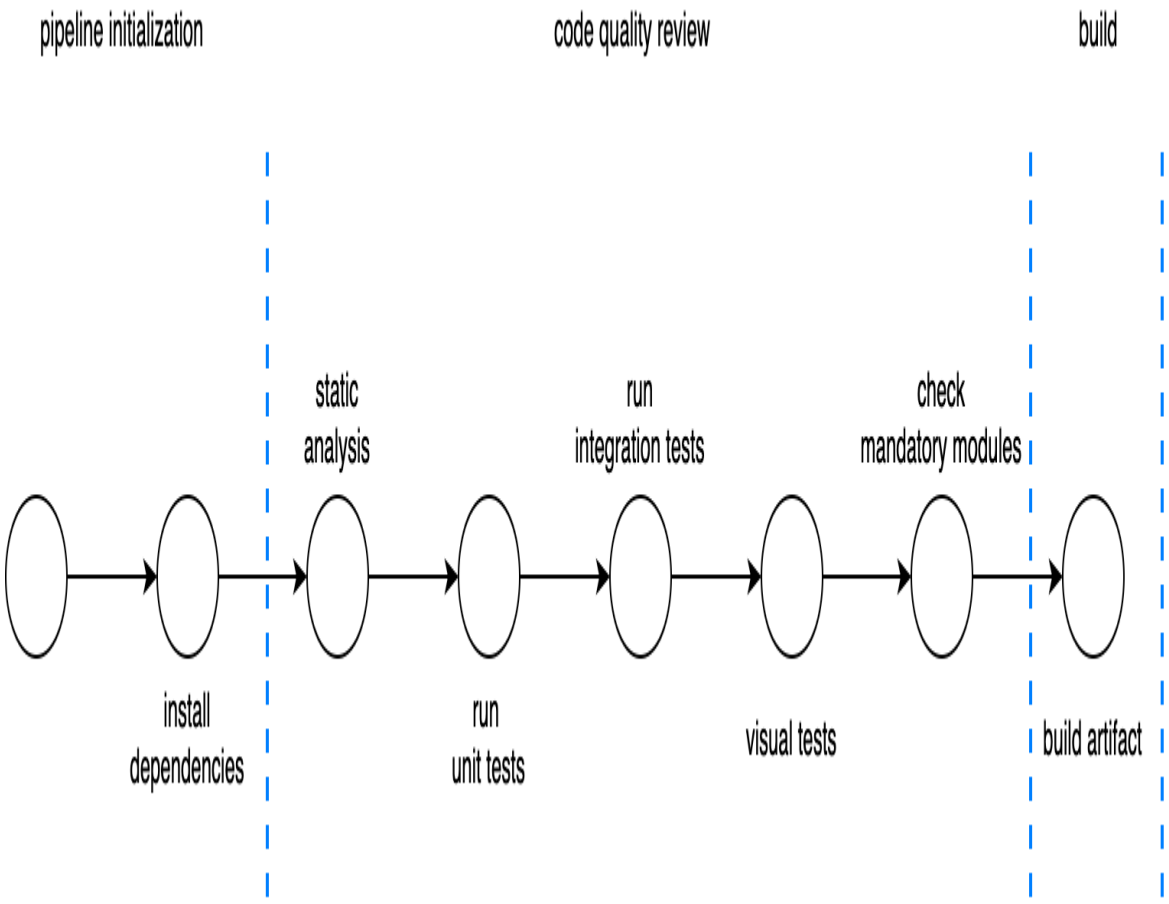


Figure 6-4. 4 - Code-quality checks like unit testing, static analysis, and visual regression tests

For static analysis, ACME Inc. uses SonarQube with the JavaScript plugin. 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.

Take this code example for instance:

```
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-frontends architecture: checking on implementing specific libraries, like logging or observability, across all the micro-frontends inside a project.

When we develop a micro-frontends application, there are some parts we want to write once and put them in all our micro-frontends.
A check on the libraries present in every micro-frontend will help enforce these controls, making sure 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 libraries 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 minifying, magnifying, and, for certain cases, even obfuscating the code.

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 Rollup 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

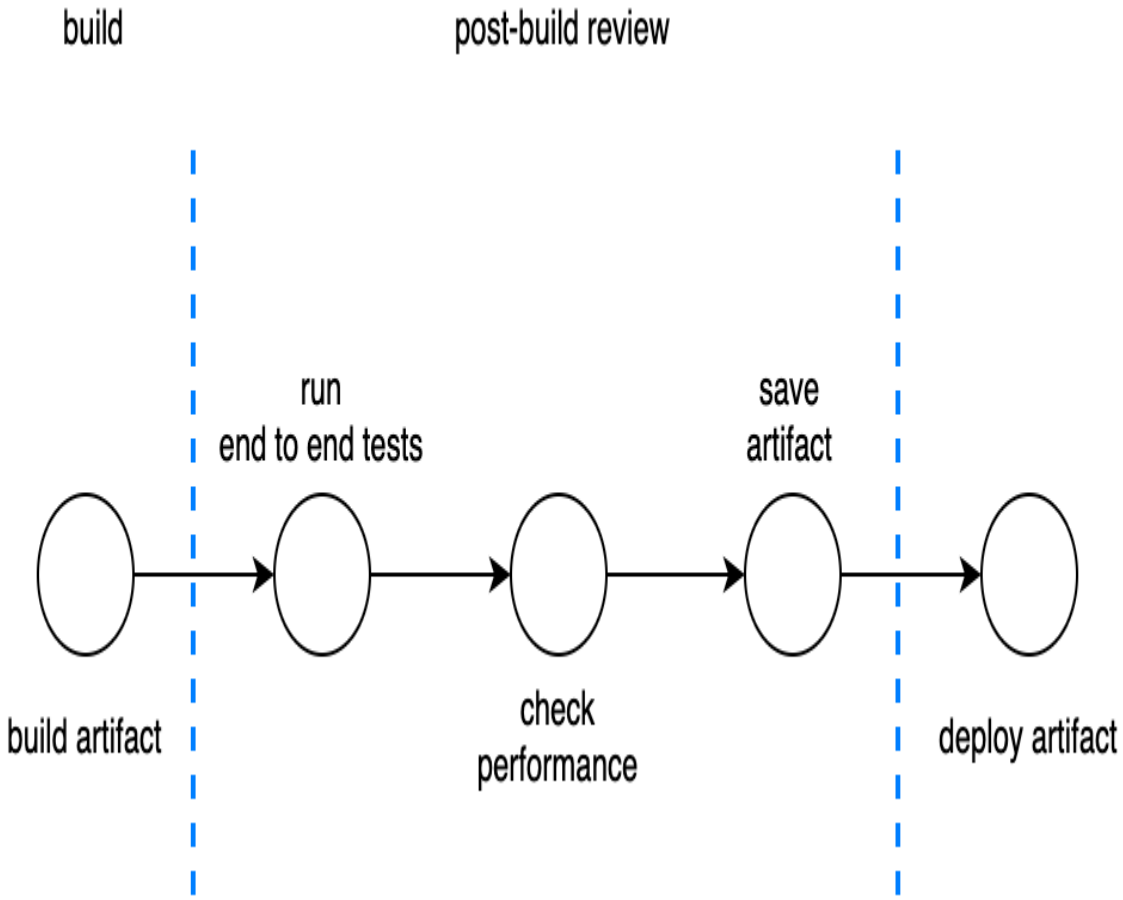


Figure 6-5. 5 - In the post-build review, we perform additional checks before deploying an artifact to an environment.

The post-build stage (figure 8.5) is the last opportunity to confirm our artifact has all the performance characteristics and requirements ready to be deployed in production.

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 plugin 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 in.

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 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 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 available inside any recent version of Chrome, 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 organizing 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 were 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 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.

Automation Strategy Summary

Every area of this automation strategy (figure 8.6) 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 on the best practices leveraged inside the organization, guiding them to delivering what the business wants.

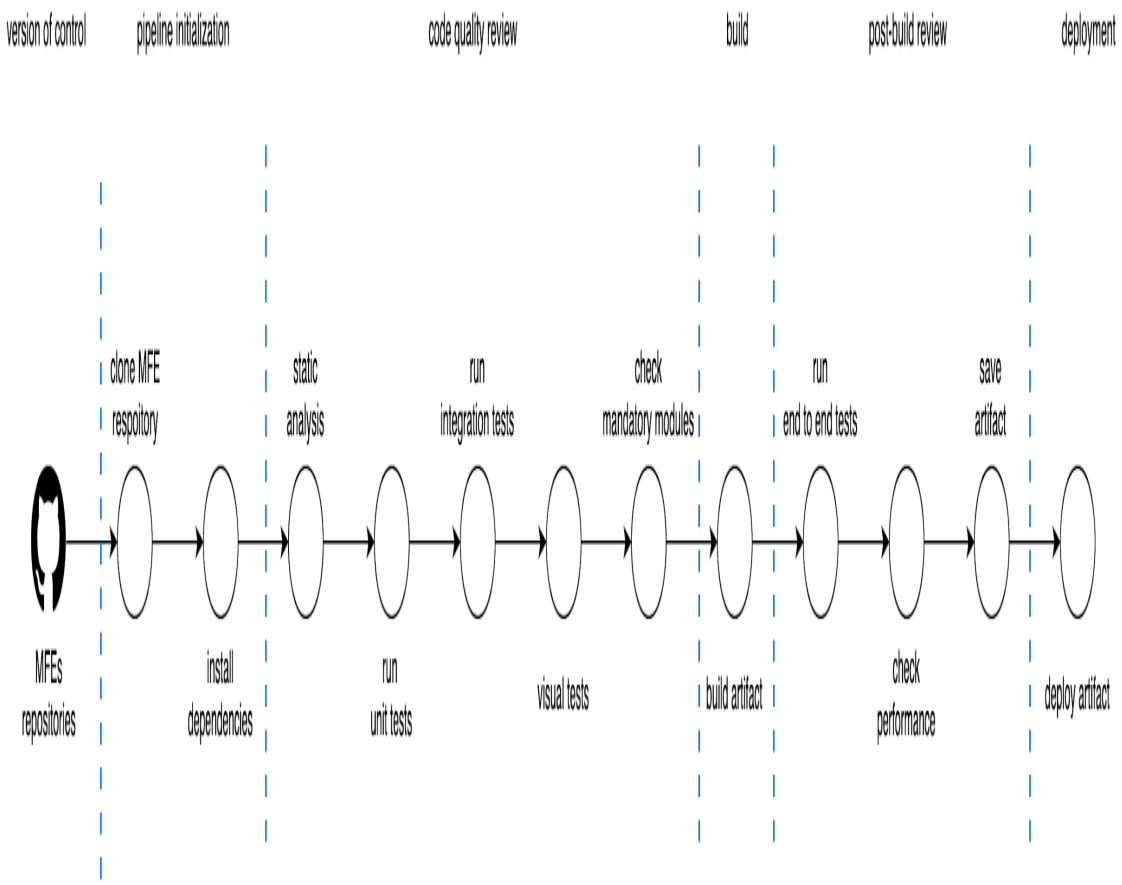


Figure 6-6. 6 - The end-to-end automation strategy diagram

The automation strategy shared in this chapter is one of many a company may decide to use. Different micro-frontends 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-frontends 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 the 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 which

discussed 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.

Micro-frontends' nature 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. Discovering Micro-Frontends Architectures

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 8th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *Building.microfrontends@gmail.com*.

Micro-frontends can be architected in different ways. In the previous chapter, we have learnt about the *decisions framework,* the foundation for any micro-frontends architecture.

In this chapter, we apply what we have learnt so far reviewing possible micro-frontends architectures and analyzing the different approaches and when to use them in real case scenarios.

Micro-Frontends Decisions Framework Applied

The decisions framework helps you to filter the architecture or framework to use for a micro-frontends project based on the characteristics of your project (figure 4.1).

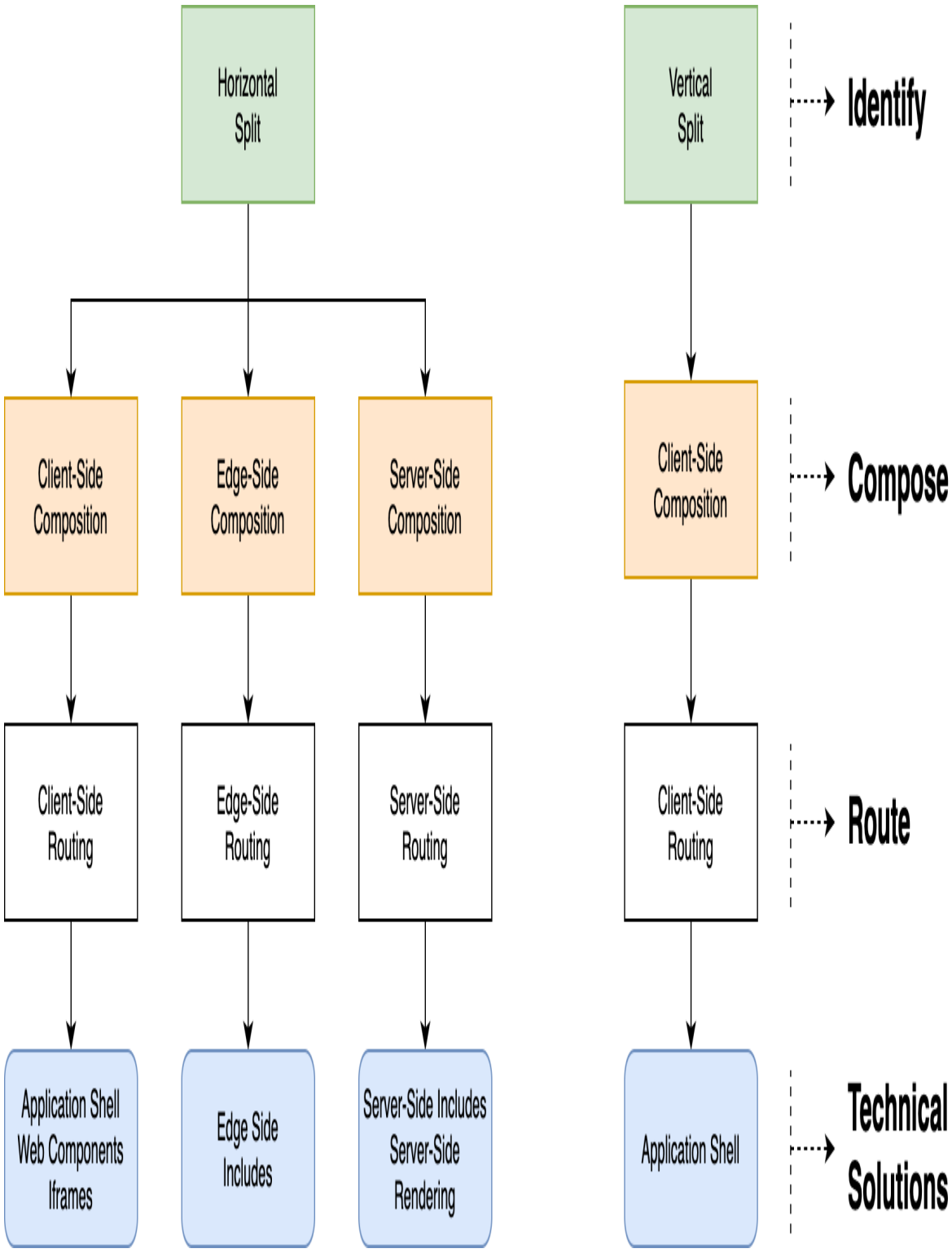


Figure 7-1. The Micro-Frontends decisions framework addresses the challenges you are facing and provides a clear path for choosing a technical solution for a project.

Horizontal Split

Your project may have a single domain (payment methods for instance) that should be presented across several views or you need to optimize your project for search engines using Server-Side Rendering (SSR) to optimize the final output, these are some of the cases where using a horizontal split can help you to achieve the goals of the project.

Choosing to implement multiple micro-frontends in the same view (horizontal split) allows you to embrace different composition techniques such as a client-side, edge-side or server-side composition.

Client-side composition would be a wise choice when your teams are more familiar with the frontend ecosystem or when your project is subject to high traffic with significant spikes so you won't need to deal with scalability challenges on the frontend layer. Edge-Side composition, instead, can be used for a project with static content but high traffic in order to delegate the scalability challenge to the Content Delivery Network (CDN) provider instead of dealing internally. As we have discussed in the previous chapter there are some challenges in embracing this architecture style (not all the CDN supports it and a rough developer experience) but projects like online catalogues with no personalized content may be a good candidate for this approach.

Another approach is the Server-Side composition where we have the most control of our output. This approach is great for highly indexed websites such as news websites or when you have a website with a fixed layout similar to PayPal or American Express websites where both are using this composition approach.

There are obviously some underlying challenges of every composition approach that we are going to investigate further in this chapter.

For every composition pattern there is a routing strategy associated with it, obviously you can technically apply any routing to any composition, however the most common in every architecture I've seen so far is the one associated with the composition layer. Therefore in the case of a client-side composition, the vast majority of the times, the routing happens at the client-side level. Sometimes its capabilities can be augmented using

computation logic at the edge (using Lambda@Edge in case of AWS or Workers in CloudFlare) but it's not a common practice for the client-side routing.

When we decide to use an edge-side composition we associate a HTML template per view so every time a user loads a new page, a new template will be composed in the CDN retrieving multiple micro-frontends that compose the final view.

Similar logic for the server-side routing where the application server knows which template is associated with a specific route and the routing and composition happens in the server-side layer.

These choices are filtering the possibilities you have for building a micro-frontends project.

In fact, when you decide to take the client-side composition and routing you can implement it with an application shell loading multiple micro-frontends in the same view, iframes or web components. Every approach has its own characteristics and we should be careful to select the right one.

In case you select the edge-side composition and routing the only solution available is using edge-sides include. The edge solution may be changed in the future when cloud providers extend their edge services and provide more computational and storage resources like they are starting in these years. However, before having a solid and battle tested solution available worldwide it may take quite a few years, therefore let's keep an eye on the evolution of these technologies but don't bet on them right now (2021) unless you have strong requirements that are pushing towards this approach.

Finally, when you decide to go through the server-side composition and routing you can technically use server-side includes or server-side rendering for your applications. Using this last approach allows you great flexibility, therefore technically you can do what you want, however the most used implementations are the ones mentioned above.

As you can see I didn't mention the fourth decisions of the framework, how the micro-frontends communicate.

Mainly because when we select a horizontal split you have to remember to avoid sharing any state across micro-frontends but used the techniques mentioned in the previous chapter, therefore event emitter, custom events, reactive streams or any other technique using the observer pattern for decoupling the micro-frontends and maintain their independent nature.

Instead, when we have to communicate between different views, using querystring parameter, for sharing information such as product identifiers, or web storage/cookes, for more persistent information such as users tokens or local users settings, are the default option without the need of consuming a backend API and avoiding additional round trips to the server.

OBSERVER PATTERN

The Observer Pattern (also known as Publish-Subscribe Pattern) is a behavioral design pattern which defines a one-to-many relationship between objects such that, when one object changes its state, all dependent objects are notified and updated automatically. An object with a one-to-many relationship with other objects who are interested in its state is called the subject or publisher. Its dependent objects are called observers or subscribers. The observers are notified whenever the state of the subject changes and can act accordingly. The subject can have any number of dependent observers which it notifies, and any number of observers can subscribe to the subject to receive such notifications.

Vertical Split

The vertical split path is more limited and probably well known by frontend developers that are used to write Single Page Application (SPA).

This type of split is helpful when your project requires a consistent evolution of the user interface and a fluid graphics across multiple views. Using this approach, provides the closest developer experience to a SPA and therefore tools, best practices and patterns can be used inside a micro-

frontend, learning the external communication with other parts of the system mainly.

Despite technically you can serve vertical split micro-frontends with server-side composition, so far all the implementations I have explored have a client-side composition where an application shell is responsible for mounting and unmounting micro-frontends. The relation between a micro-frontend and the application shell is always 1 to 1, therefore the application shell loads only one micro-frontend per time.

The routing is usually split in two parts, a global routing used for loading different micro-frontends is handled by the application shell (figure 4.2).

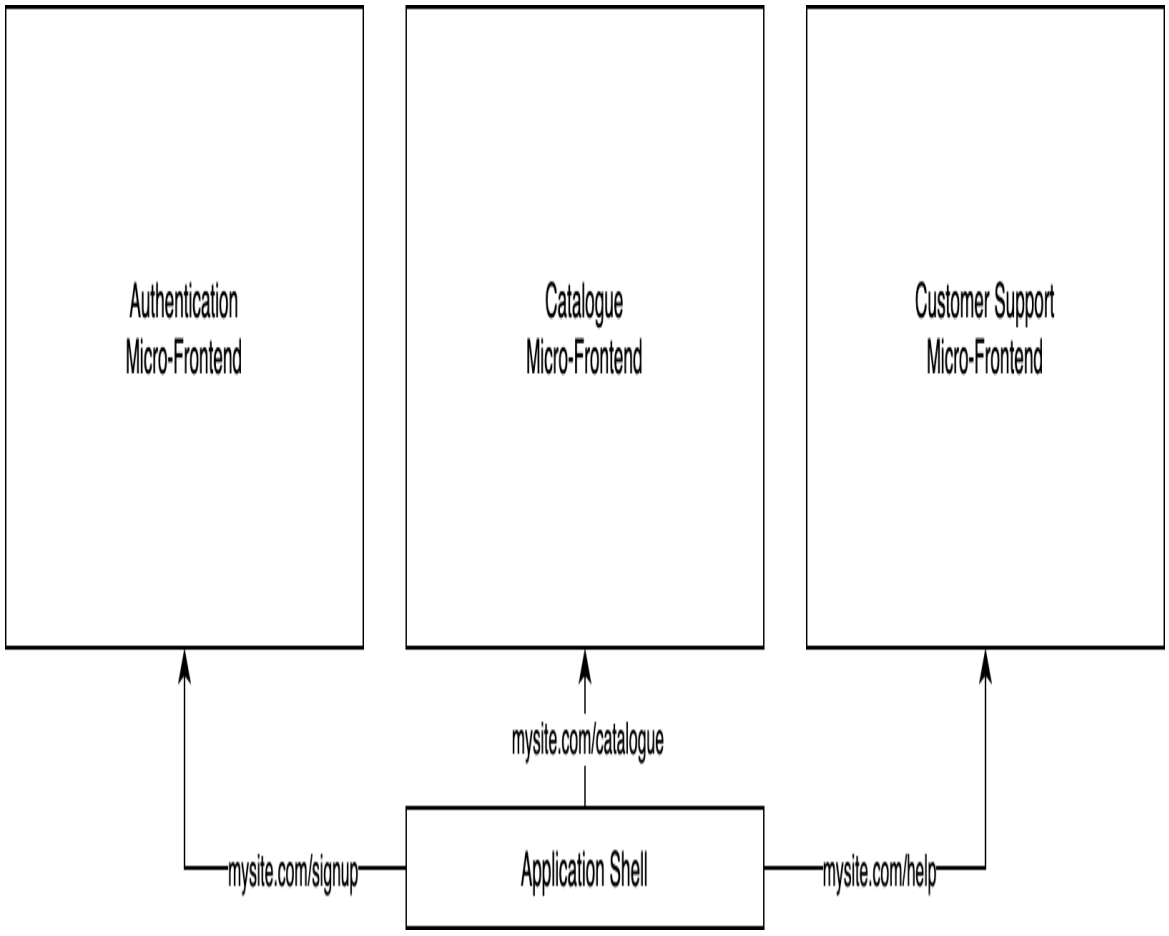


Figure 7-2. The application shell is responsible for the global routing between micro-frontends

Moreover the local routing between views of the same micro-frontend is managed by the micro-frontend itself, having full control of the

implementation and evolution of the views present inside the micro-frontend (figure 4.3).

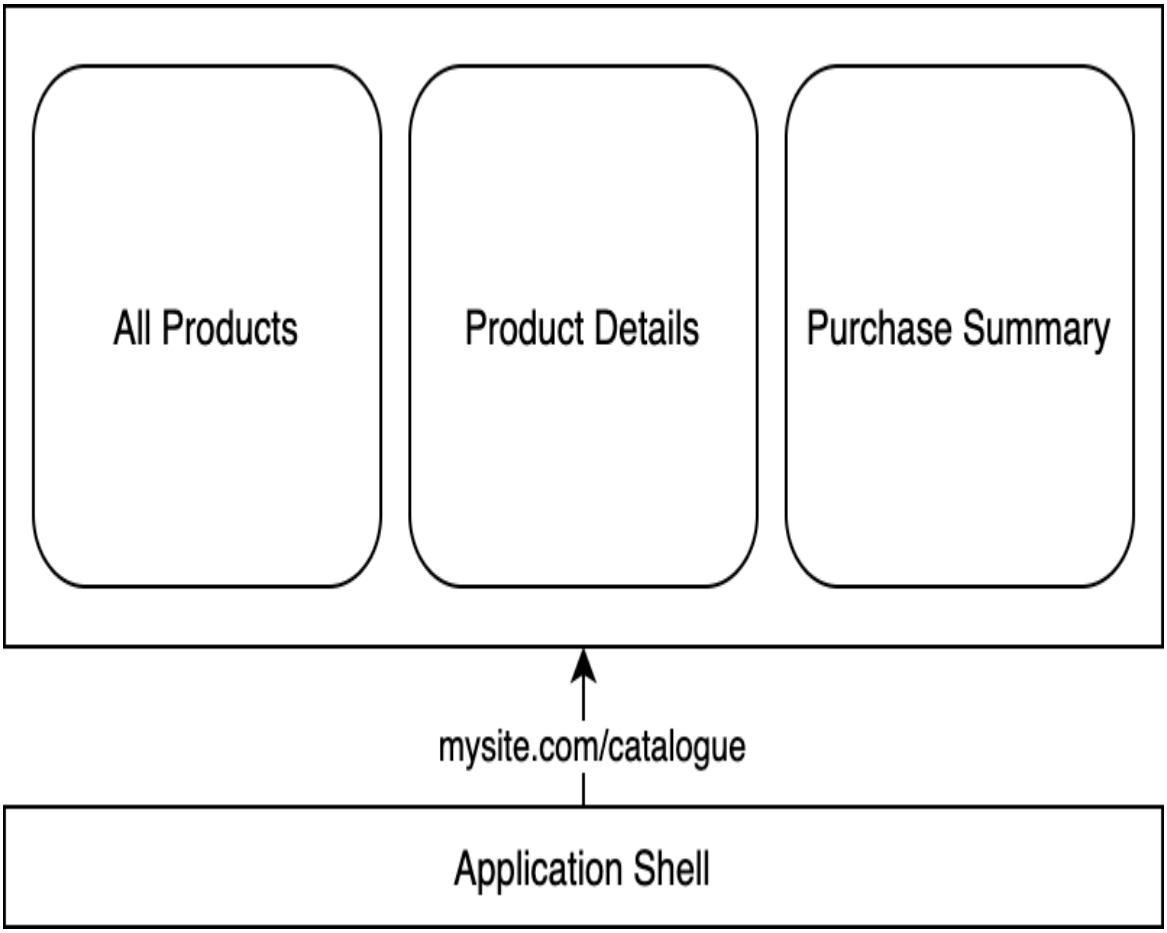


Figure 7-3. A micro-frontend is responsible for the routing between views available inside the micro-frontend itself

For implementing an architecture with a vertical split micro-frontend, the application shell loads an HTML or JavaScript as entry point; the application shell shouldn't share any business domain logic with the other micro-frontends and it should be technology agnostic, therefore don't use any specific framework for building an application shell but try to use Vanilla JavaScript in case you built your own implementation. The application shell is always present during the users sessions because it's responsible for orchestrating the web application as well as exposing some lifecycle APIs for micro-frontends in order to react when they are fully mounted or unmounted.

When vertical split micro-frontends have to share information such as tokens or user preferences, with other micro-frontends, they can use querystring or web storages like the horizontal split ones are doing when a view changes.

Architecture Analysis

After this brief overview of the different architectures associated with the decisions framework pillars, it's time to analyze the technical implementations and understand the implementation challenges and benefits.

First, we review the different implementations available more in detail and then, for every architecture, we assess different architecture characteristics so you will be able to select the right approach for any project based on the requirements and predominant characteristics you want to emphasize within the architecture.

The architecture characteristics we analyze for every implementation are:

Deployability

> how reliably and easily a micro-frontend can be deployed into an environment

Modularity

> degree to which a system's components may be separated and recombined, often with the benefit of flexibility and variety in use.

Simplicity

> the quality or condition of being easy to understand or do. If a piece of software is considered "simple", then chances are it has been found to be easy to understand and easy to reason about

Testability

degree to which a software artifact supports testing in a given test context. If the testability of the software artifact is high, then finding faults in the system by means of testing is easier.

Performance

indicator of how well a software system or component meets its requirements for timeliness. Timeliness is measured in terms of response time or throughput.

Developer Experience

it describes the experience developers have when they use your product, be it client libraries, SDKs, frameworks, open source code, tools, API, technology or service.

Scalability

attribute that describes the ability of a process, network, software or organization to grow and manage increased demand

Coordination

it is the unification, integration, synchronization of the efforts of group members so as to provide unity of action in the pursuit of common goals.

A one-point rating in the characteristics ratings table means the specific architecture characteristic isn't well supported in the architecture, whereas a five-point rating means the architecture characteristic is one of the strongest features in the architecture style.

Architecture and Trade-offs

As you will read in many parts of this book, I'm a big believer that the perfect architecture doesn't exist, it's always a trade-off. The trade-offs are

not merely technical, but also based on business requirements and organization structure.

Modern architecture doesn't take into consideration only the technical aspects but also other forces that contribute to the final outcome. It's important to recognize the social-technical aspects and optimize for the context we operate in instead of researching for the "perfect architecture".

In Fundamentals of Software Architecture, Neal Ford and Mark Richards highlighted very well the new modern architecture practices and invite the readers to optimize for the less worse architecture.

> From chapter 4 of Fundamentals of Software Architecture: ***"Never shoot for the best architecture, but rather the least worst architecture."***

My advice before nailing any architecture is investing time to understand the context you operate in, the teams structure and the communication flows between teams. Don't underestimate these aspects because you may risk to create a great technical proposition but not suitable for the company you work for. The same when you see case studies from other companies embracing specific architectures. They may operate in a different way compared to your working place and often the case studies focus on how these companies solved a specific problem that may or may not overlap with your challenges.

Read, research, try, engage with different people in the community for understanding the forces behind certain decisions, these steps will avoid making wrong assumptions and you will become more aware of the environment you are working in.

This is also the reason why you will see multiple different approaches for micro-frontends, every architecture is optimized for solving specific technical and organizational challenges. **There isn't right or wrong in architecture but just the best trade-off for your own context.**

Vertical Split Architectures

The main implementation of vertical split micro-frontends is with a client-side composition and routing using an application shell, a light and generic layer that should avoid holding any business domain logic or implementing the bare minimum.

Technically speaking we can use a vertical split micro-frontends architecture also with other composition types, however all the projects that identify their micro-frontends with the vertical split, are composing and routing on the client-side.

This approach is fantastic for teams who want to approach micro-frontends for the first time and they have a solid background on building SPA because they are going to have a familiar development experience with not many twists.

Application Shell

The application shell is a persistent part of a micro-frontends application that shepherds a user session from the beginning to the end.

In fact, the application shell is the first thing downloaded when an application is requested and it's responsible for loading and unloading micro-frontends based on the endpoint requested by a user.

Moreover, the application shell usually consumes a configuration, in the form of a static JSON file or provided by a backend service, containing the information of which endpoint is associated to a specific micro-frontend. You can also embed the routes inside your application shell code, however this would mean deploying a new application shell version everytime a route change occurs.

The application shell loads one micro-frontend per time in this architecture, so there is no need of creating a mechanism for encapsulating conflicting dependencies between micro-frontends because there won't be any clash between libraries or CSS styles (figure 4.4).

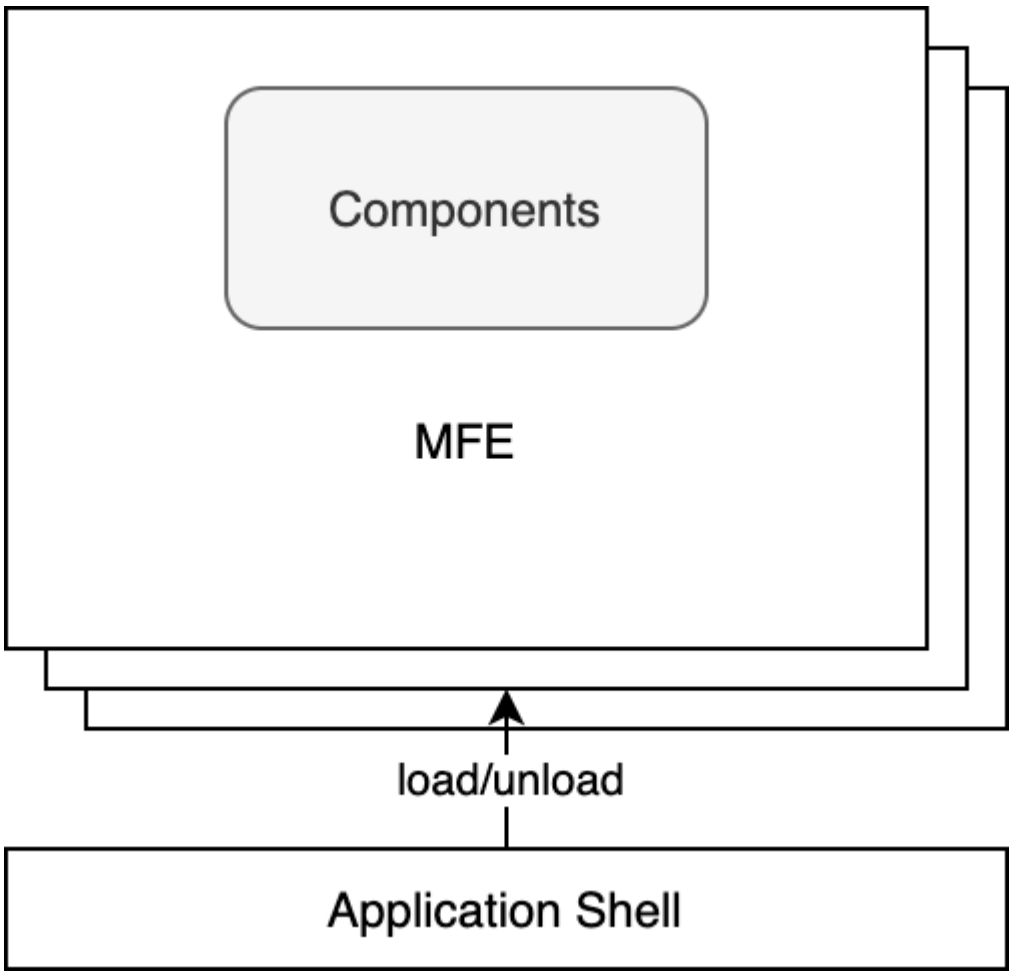


Figure 7-4. Vertical split architecture with client-side composition and routing using the application shell

The application shell is nothing more than a simple HTML page with logic wrapped in a JavaScript file and it may or not include some CSS styles for the initial load, for instance for showing a spinner or any other loading animation.

Every entry point of a micro-frontend is represented by a single HTML page containing the logic and style of a single view or a small SPA containing a bunch of routes that includes all the logic needed for allowing a user to consume an entire subdomain of our application without the need to load a new micro-frontend.

This approach is favorable when we want to create a consistent experience for our users and we want to provide the full control to a single team. A clear sign that this may be the right approach for your application is when

we analyze the different user interface of our website or web application and there aren't many repetitions of elements across multiple views but every part of the application may be represented by an application itself.

It's also a simple approach for frontend developers used to develop SPA considering the tools and practices are similar for building micro-frontends in this way.

Identifying micro-frontends are quite trivial when we have a clear understanding how the users are interacting with the application, maybe using tools like Google Analytics, and when we can group services and micro-frontends in the same bounded context, developed by one or a few teams.

If we don't have this information, the first investment to make is understanding how the users are interacting with our platform and then reviewing how to structure the architecture and accordingly the organization's structure.

When we approach this architecture, there isn't a high reusability of micro-frontends, therefore it's unlikely that a vertical split micro-frontend will be reused in the same application in another context.

However, inside every micro-frontend we can decide to reuse components that are complex or used in several places on our platform (think about a design system), generating a modularity that may help avoiding duplication.

It's definitely more likely that micro-frontends may be reused in different applications maintained by the same company. For instance, imagine you have multiple software as a service to develop and you want to have a similar user interface with some changes for every product you sell. This is another scenario where you will be able to reuse vertical split micro-frontends, reducing the code fragmentation and evolving the system independently based on the business requirements.

Challenges

In this architecture pattern there are many challenges we face during the implementation phase.

Independently from the domain specific ones, there are common challenges we have to overcome, some of them have an immediate answer, some others are more dependent by the context.

Sharing State

The first challenge we face when we work with micro-frontends in general is how to share states between micro-frontends. With a vertical split architecture, the need for sharing information is by far less than when we have multiple micro-frontends in the same view, anyway the needs still stand.

There are certain types of information that we may want to share across multiple micro-frontends in particular when we have multiple vertical split micro-frontends inside an authenticated area of our platform.

A classic example is sharing volatile data that are fine stored in the browser, such as the audio volume level for the media contents played by a user or the recent fonts used for editing a document. Usually these data are shared via the web storage, either the local or session storage, depends for how long you need to store them.

When we talk about more sensitive information such as personal user data or authentication token, we need a way to retrieve this information from a public API and then share across all the micro-frontends interested in this information.

In this case, a best practice is at the beginning of a user's session, the first micro-frontend loaded to the user would retrieve the data, stored in a web storage with a timestamp for when the data was retrieved.

Then, every micro-frontend that requires these data can retrieve directly from the web storage, however if the timestamp is older than a certain amount of time, the micro-frontend can request the data again.

In this way, we are sure to retrieve the data needed for a user, share with other micro-frontends and in case it is needed, refreshing them from any other micro-frontend.

Finally, it's important to highlight that every micro-frontend will have access to the selected web storage, considering the application loads one micro-frontend per time, there is no strong requirement to pass through the application shell for storing data in the web storage. However, when your application relies heavily on the web storage and you decide to implement some security checks for validating the space available or type of message stored, you may want to re-evaluate the decision and possibly create an abstraction via the application shell that will expose an API for storing and retrieving data and it will centralize where the data validation happens providing meaningful errors to every micro-frontend in case a validation fails.

Multi-Frameworks approach

This is one of the controversial points for using micro-frontends, many people think that embracing this architecture would force them to use multiple UI frameworks like React, Angular or Svelte.

However what is true for frontend applications written in a monolithic way, it's also true for micro-frontends.

Despite being technically doable implementing multiple UI frameworks in a SPA, we won't do that due to performance issues and potential dependencies clashes.

This applies to micro-frontends as well, therefore in general it's not recommended using a multi-framework implementation for this architecture style either.

Instead, follow the best practices such as reduce the external dependencies as much as you can, import only what you use and not entire packages that may increase the final JavaScript bundle and make sure you are using libraries without security vulnerabilities.

This is a good rule of thumb, however there are some use cases where having a muti-framework approach with micro-frontends becomes less important than the benefits of developing and deploying iteratively your application.

Imagine you start a porting of a frontend application from a SPA to micro-frontends.

One approach that helps you to provide value for your business and users would be working on a micro-frontend and deploying alongside the SPA codebase.

This approach helps in multiple ways, first of all we will have a team finding best practices for approaching the porting such as libraries to re-use, how to setup the automation pipeline, how to share code between the micro-frontends they are responsible for and many others.

Secondly, after creating the minimum viable product (MVP), the micro-frontend can be shipped to the final user retrieving metrics, and comparing with the older version.

In a situation like this, asking a user to download multiple UI frameworks is less problematic than developing for several months the new architecture without understanding if the direction taken is leading to a better result.

Validating your assumptions is crucial for generating the best practices shared by different teams inside your organization. Increasing the feedback loop, bringing code to production as fast as possible is demonstrating the best approach for overcoming future challenges with micro-architectures in general. The same reasoning is applicable to other libraries used in the same application but in different versions

Multi-Frameworks approach

This is

- Composition -> how
- Team composition and communication
- Evolving architecture (splitting the micro-frontends)
- Design system / code sharing -> libraries
- Developers experience

- Testing
- SEO
- PWA
- Performance (check MFE in action)

Frameworks available

There are some frameworks available for embracing this architecture, however building an application shell by your own it won't require too much effort, either for building or maintaining as long you keep the application shell decoupled from every micro-frontend. Injecting domain code in the application shell it's not only a bad practice but also it may invalidate the all effort and investment of using micro-frontends in the long run.

Two frameworks that are fully embracing this architecture are Single SPA and Qiankun. Qiankung is built on top of Single SPA, adding some functionality that in the latest releases of Single SPA are available also in this most popular option.

Considering they are almost identical, I'm going to focus on Single SPA, being more well-known than the other, describing how it works and what are its key characteristics.

Finally, we may have another option, despite it not coming to your mind immediately. Module Federation may be a good alternative for implementing a vertical split architecture considering all the mounting and unmounting mechanism, the dependencies management, the orchestration between micro-frontends and many other features are already available to use. Also it's built on top of WebPack, and other famous bundlers like Rollup, therefore if your projects are already using WebPack it may be a good alternative to look at without the need of learning new frameworks for composing and orchestrating all the micro-frontends of your project.

Use Cases

This architecture style is a good solution when you have frontend developers implementing it with experience on SPAs development. It may scale up to certain extent when we look to the organizational approach, for instance, if you have hundreds of frontend developers working on the same frontend application, probably an horizontal split may suit the organization scalability considering you can modularize even further your application.

It's a great architecture when you aim for user interface and user experience consistency, considering every team is responsible for a specific business domain, often seen as user experience, and they can develop this experience end to end without the need to coordinate with other teams.

Another key indicator for leaning towards this architecture style is the level of reusability you want to have across multiple micro-frontends. For instance, if you reuse mainly components of your design system and some libraries like logging or payments, this may be a great architecture fit for your project. However, if part of your application is replicated in multiple domains, probably an horizontal split may be a better solution to implement. Again, understand your context to make the best trade-off.

Also, this architecture is my first recommendation when you start embracing micro-frontends because it doesn't introduce too much complexity, it has a smooth learning curve for frontend developers, it distributes the business domains to tens of frontend developers without any problem and it doesn't require huge upfront investment in tools but more in general in the developer experience.

Architecture Characteristics

Deployability (5/5): considering every micro-frontend is a single HTML page or a SPA, we can easily deploy our artifacts on a cloud storage or an application server, sticking a CDN in front of it and we don't have to think about it anymore. Well-known approach, used for several years by many frontend developers for delivering their web applications.

Modularity (2/5): this architecture is not the most modular one, we have a certain degree of modularization and reusability but more at the code level,

components or libraries, then at micro-frontend level despite few exceptions. Also, there is a clear approach when we need to divide a vertical split micro-frontend in smaller units, however it requires a good effort for decoupling all the shared dependencies implemented when it was an unique logical unit.

Simplicity (4/5): considering the main aim of this approach is reducing the teams cognitive load, creating domain experts using well-known approaches with some twists, the simplicity for a frontend developer is intrinsic considering there aren't too many shifts of mindset and new techniques to learn for embracing this architecture.

Testability (3/5): compared to SPAs, this approach shows some weakness in the application shell end to end testing, however apart from that edge case, the different type of testing per micro-frontend is bread and butter for any frontend developer.

Performance (3/5): there is the possibility to share the common libraries for a vertical split architecture, it requires a minimum of coordination across teams but taking into account it's very unlikely having hundreds of micro-frontends with this approach, it becomes easy creating a deployment strategy decoupling the common libraries from the micro-frontend business logic and maintain the commonalities in sync across multiple micro-frontends.

Compared to other approaches such as server-side rendering, there is a delay on downloading the code of a micro-frontend because the application shell should initialize the application with some logic that may impact the load of a micro-frontend when it's too complex or makes many roundtrips to the server.

Developer Experience (4/5): when a team is familiar with SPA tools, they won't suffer this shift of mindset embracing the vertical split. There may find some challenges during end to end testing, but all the other engineering practices as well as tools may remain the same.

The teams may feel the need to build additional tools (CLI, desktop apps, dashboards…) for filling some gaps for their specific context. However the

out-of-the-box tools available should be enough for starting the development and they can defer the decisions to build new tools if and when a strong case arises.

Scalability (5/5): The scalability aspect of this architecture is so great that we can even forget about it when we serve our static content via a CDN and we configure the time-to-live accordingly the assets we are serving, higher time for assets that doesn't change often like fonts or vendor libraries and lower time for assets that change often like the business logic of our micro-frontends. This architecture can scale almost "indefinitely" based on CDN capacity that usually is great enough for serving billions of users simultaneously. In certain cases, when it is an absolute must avoiding single point of failure, you can even create a multiple CDN strategy where your micro-frontends are served by multiple CDN providers. Despite being more complicated it solves the problem elegantly without investing too much time creating custom solutions.

Coordination (4/5): This architecture, compared to others, enables a strong decentralization of decision making as well as autonomy of each team.

Usually the touching points between micro-frontends are minimum when the domain boundaries are well defined. Therefore there isn't too much coordination needed apart from an initial investment for defining the application shell APIs and keeping them as domain unaware as possible.

In table 4.1 we can find a synthetic view of every architecture characteristics and their associated score for this micro-frontends architecture:

Table 7-1. Architecture characteristics

i
c
s
s
u
m
m
a
r
y
f
o
r
d
e
v
e
l
o
p
i
n
g
a
m
i
c
r
o
-
f
r
o
n
t
e

n
d
s
a
r
c
h
i
t
e
c
t
u
r
e
u
s
i
n
g
v
e
r
t
i
c
a
l
s
p
l
i
t
a
n
d

applicationshellascompositionandorch

e
s
t
r
a
t
o
r
.

Architecture Characteristics	Score (1 - lowest, 5 - highest)
Deployability	5/5
Modularity	2/5
Simplicity	4/5
Testability	3/5
Performance	3/5
Developer Experience	4/5
Scalability	5/5

Horizontal Split Architectures

- Client-side
    - Application Shell
    - Iframes
    - Web components
- Server-side
- Edge-Side

Summary

In this chapter we have applied the micro-frontends decisions framework to multiple architectures. Defining the four pillars offered by this mental model helps us to filter our choices and select the right architecture for a project.

We have analysed different micro-frontends architectures, highlighting their challenges and scoring the architecture characteristics so we can easily select the right architecture based on what we have to optimize for.

Finally we understood that the perfect architecture doesn't exist, we have to find the *less worse architecture* based on the context we operate in. In the next chapter we will take a step further and we analyse a technical implementation and focus our attention to the main challenges we may encounter in a micro-frontends implementation.

Chapter 8. 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 10th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *Building.microfrontends@gmail.com*.

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.

In the meanwhile, you also have a platform to maintain in production, which the product team would like to evolve because the re-platforming 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.

In this chapter we will see what's next. 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-frontends one, by far the most common use case of this architecture. I believe any project should start simple. Then over the course of the months or years, when the business and the organization are growing, the architecture should evolve accordingly to support the evolving business needs. 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 them 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 with you a story 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 in the world. The company is growing fast. In the last couple of years it has

moved from hundreds of employees to thousands, all across the globe, and the tech department is no exception.

The streaming platform is currently available on desktop and mobile browsers, as native application, and on some living-room devices, like 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.

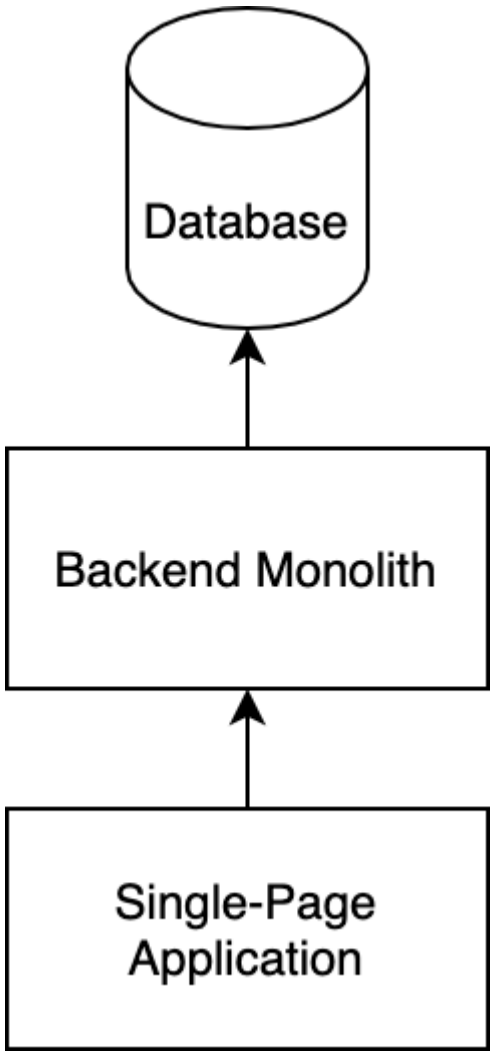


Figure 8-1. The ACME platform is a three-tier web application

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, choosing the right solution for each domain, and scaling the platform according to the traffic on a service-by-service basis, 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 9.

Platform and Main User Flows

The frontend is composed of the following views:

- Landing Page
- Sign In
- Sign Up
- Payment
- Remember Email
- Remember Password
- Redeem Gift Code
- Catalogue (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 (figure 10.2):

1. The user arrives on the landing page, which explains the value proposition.
2. They then move 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.

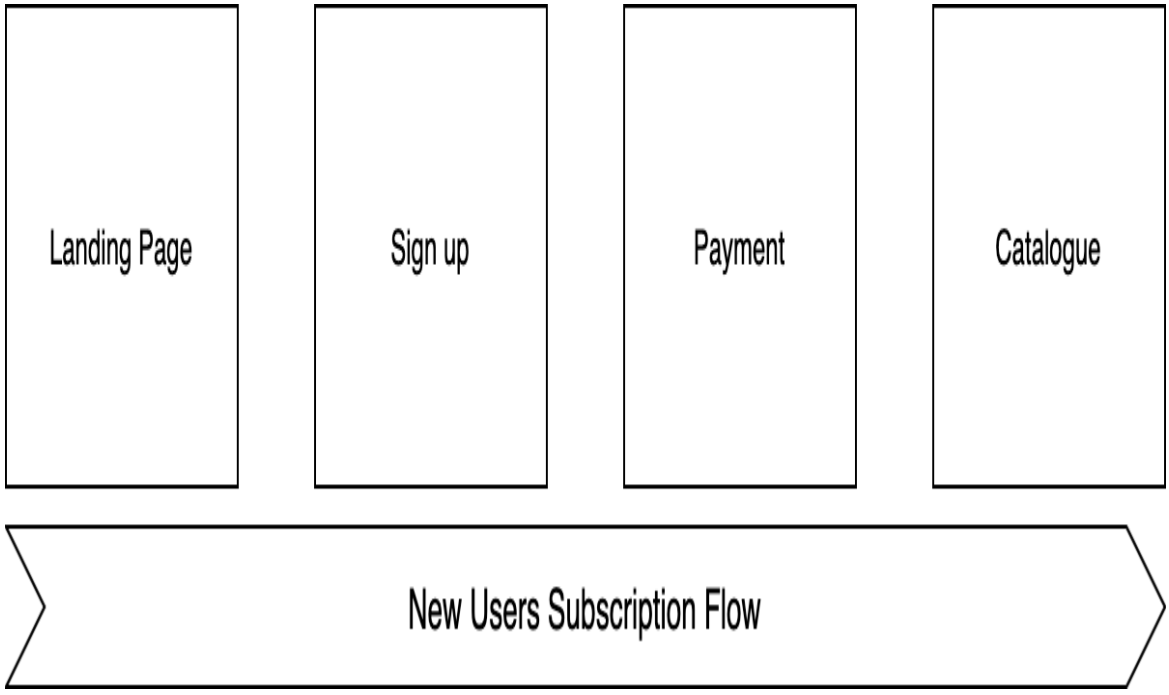


Figure 8-2. New user subscription flow

When an existing user wants to sign in on a new platform (browser or mobile device, for instance) to watch some content, they will (figure 10.3):

1. Access the platform in the landing page view
2. Select the signin button, which redirects them to the signin view
3. Insert their credentials
4. Access the authenticated area and explore the catalogue

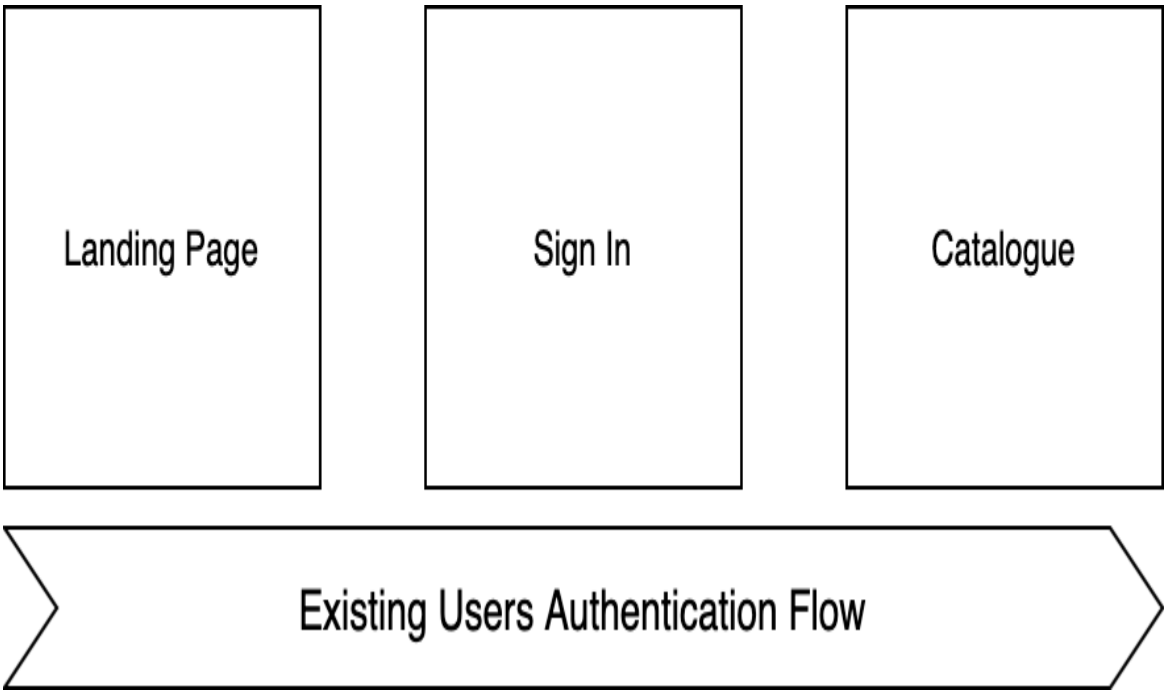


Figure 8-3. Existing users authenticating in a new platform (browser or mobile devices for instance)

Once a user is authenticated, they can watch video content and explore the catalogue following these steps (figure 10.4):

1. They start at the catalogue 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 catalogue view, they can choose to use the search functionality.

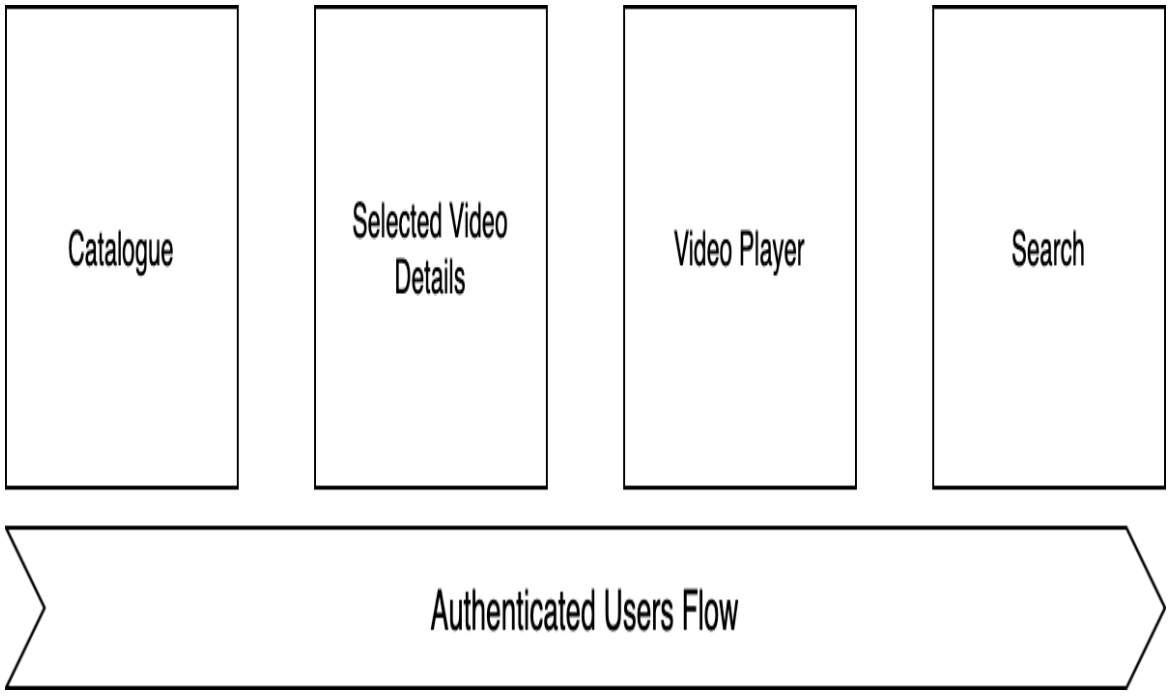


Figure 8-4. Existing users can navigate content via the catalogue or search functionality and then play any content after discovering the details of what they are about to watch.

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 these 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, 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, Cascading Style Sheet (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.

Moreover, tech leadership wants to reduce the risk of introducing bugs or defects in production, easing the traffic toward new micro-frontends 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-frontends 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 teams started to work on a plan for migrating the entire platform to micro-frontends.

The first step was embracing the micro-frontends decisions framework, outlined in chapter 2. 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-frontends decisions framework gives us a skeleton to architect a micro-frontends project on to. All the other decisions will build on top of this frame, creating a reliable structure.

Micro-Frontends 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 (figure 10.5).

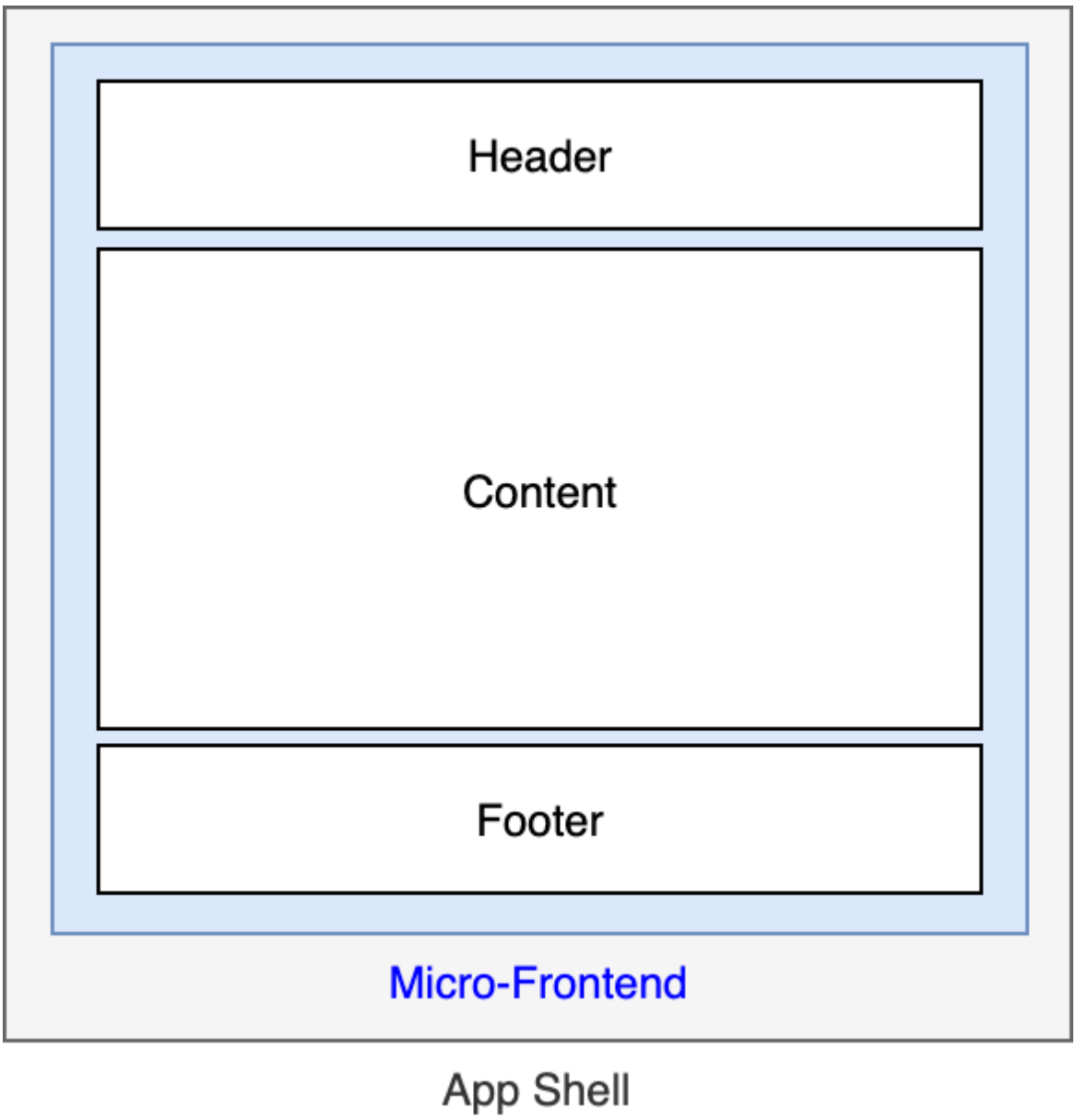


Figure 8-5. A vertical split micro-frontend, where the application shell loads only one micro-frontends at a time

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 a SPA, a vertical split allows developers to work similarly to how they have worked 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, it's not needed. A light design system will ensure consistency across micro-frontends, and it doesn't create a huge overhead of dealing with it.

Better integration with current automation strategy

The vertical split fits very well with the current automation strategy, considering right now ACME is building a 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 we always load 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 touching points of their business domain and recognize where a bug may appear despite not inside their domain.

Faster on boarding 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 (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.

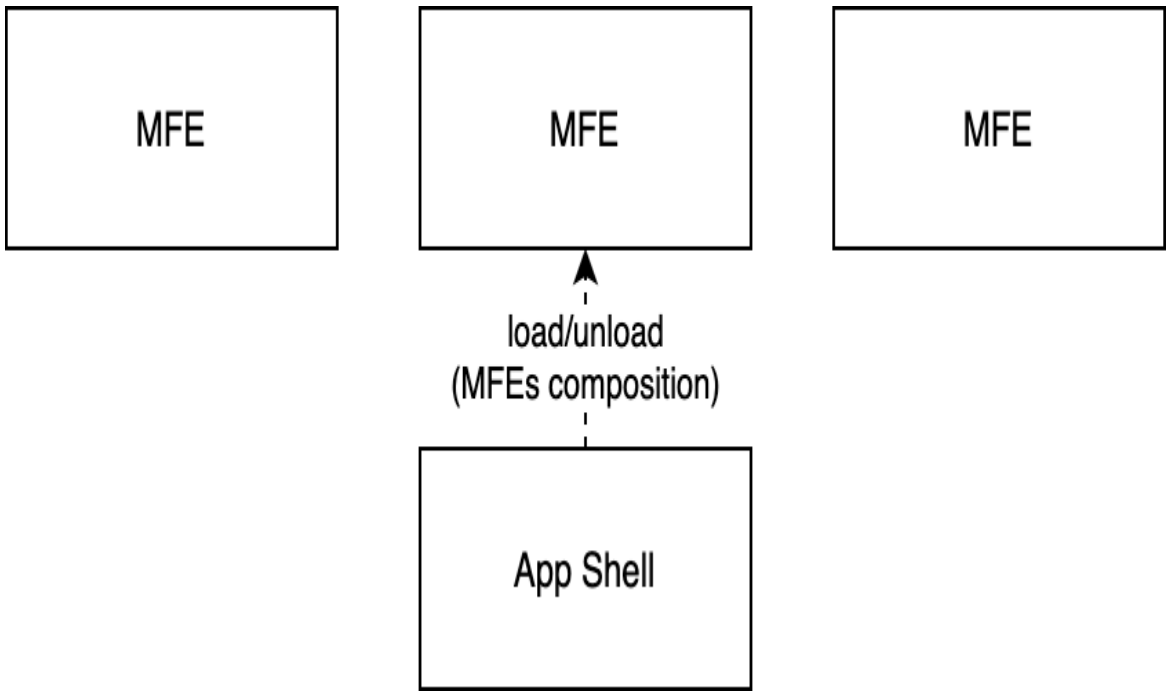


Figure 8-6. Client-side composition, where the application shell is responsible for loading and unloading one micro-frontend at a time.

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 composing 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 (figure 10.7).

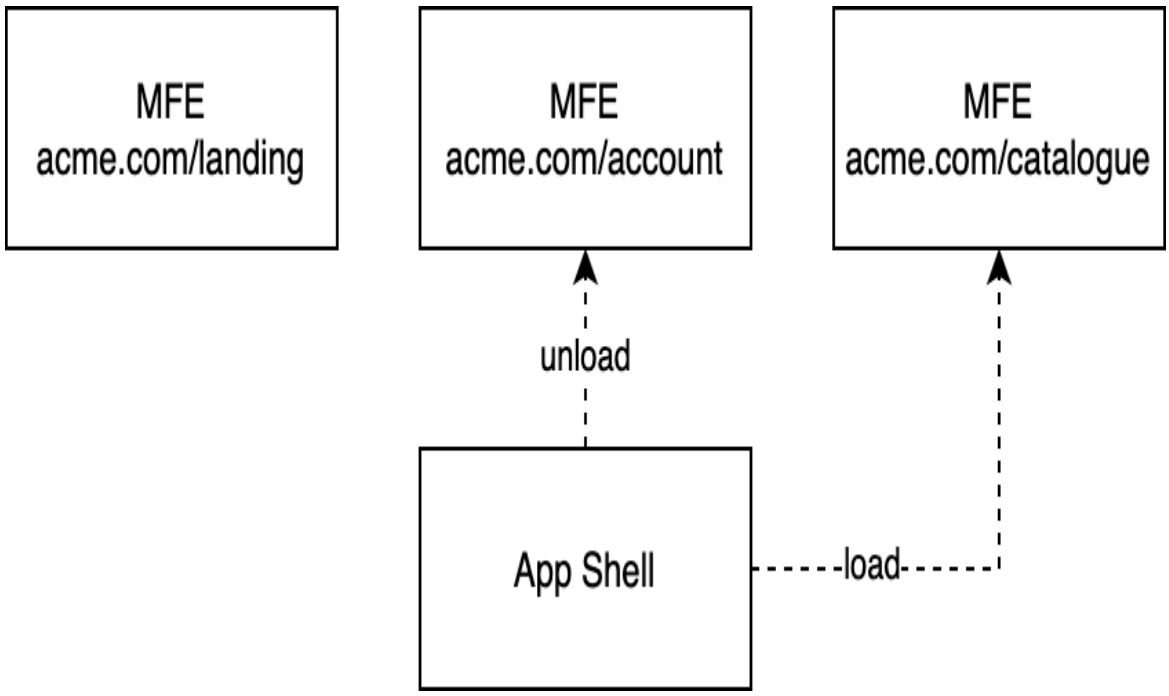


Figure 8-7. When the user signs in from the /account path, they are redirected to the authenticated area (/catalogue). The application shell owns the logic for unloading the current micro-frontend and loading the next one based on the URL.

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 query string or using web storage. 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 (figure 10.8).

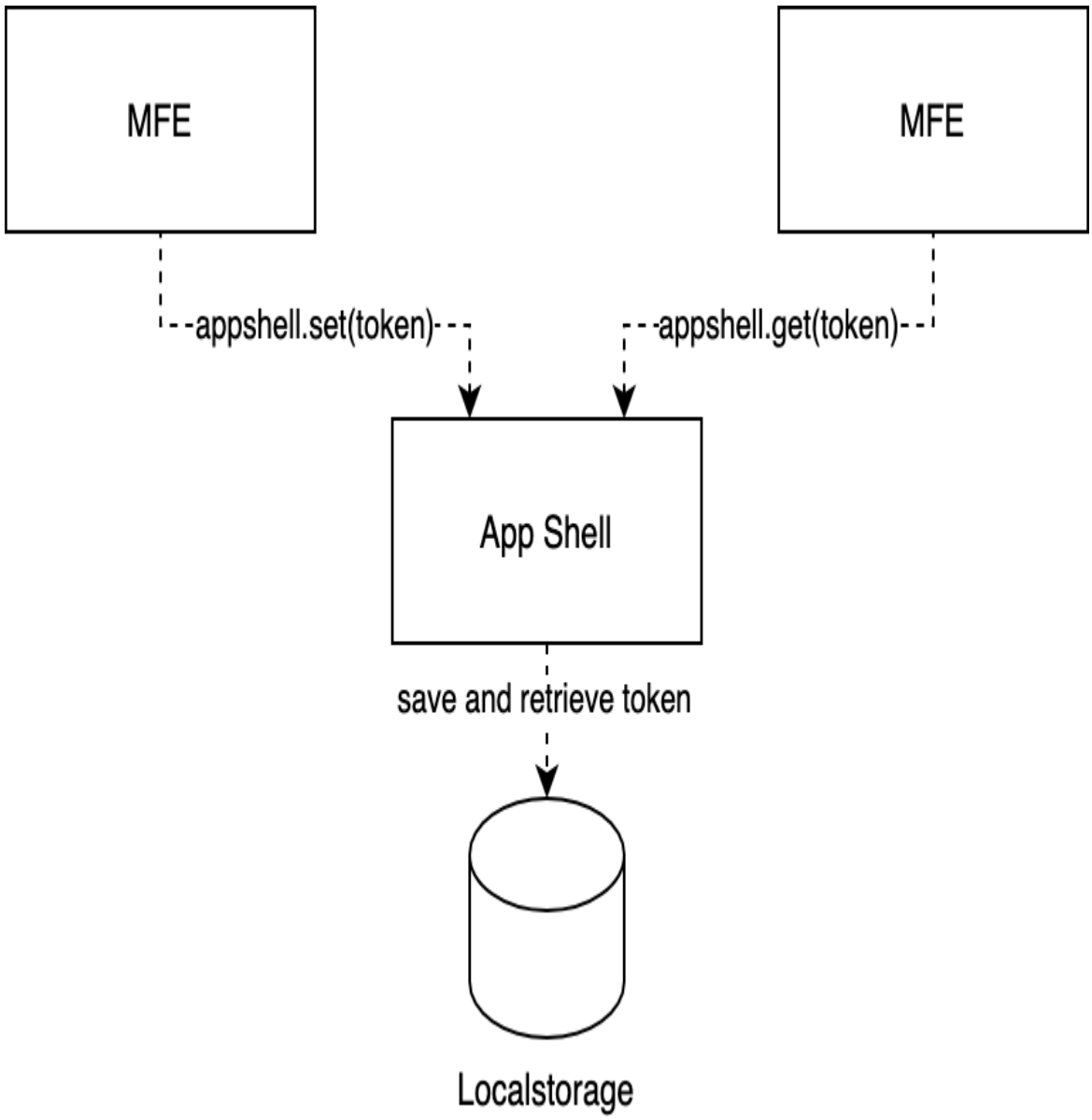


Figure 8-8. The application shell is responsible for storing data in the local storage and exposing several APIs to the micro-frontends for storing and retrieving data.

Let’s summarize the decisions made by the teams in the following table 10.1:

Table 8-1. Summary of ACME architecture

a
l
d
e
c
i
s
i
o
n
s

Micro-Frontends Decisions Framework
Define micro-frontends Vertical split
Compose micro-frontends Client side via application shell
Routing micro-frontends Client side via application shell
Communication between micro-frontends Using web storage via application shell

Splitting the SPA in Multiple Subdomains

After creating their micro-frontends 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 discovers 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 customer.

Table 8-2. New user trait affi cper view in

A
C
M
E

p
l
a
tf
o
r
m

View	Traffic
Landing Page	100%
Sign Up	70%
Payment	60%
Catalogue	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 catalogue, as seen in table 10.3:

Table 8-3. Unauthenticated dexisting use

r
t
r
a
ff
i
c
p
e
r
v
i
e
w

f
o
r
a
c
c
e
s
s
i
n
g
A
C
M
E

p
l
a
tf

o
r
m

View	Traffic
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).

Table 8-4. Authenticated existing users

traffic perview for accessing ACME platfo

r
m

View	Traffic
Landing Page	0%
Sign In	1%
Sign Up	0%
Catalogue	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 catalogue since authenticated users only occasionally navigated to other parts of the application.

- My Account and Help don’t receive much traffic.
- Since there is a considerable drop of new users between landing page and sign-up flows, it’s very likely the business would like to iterate often to bridge the chasm and move more users through the sales funnel.

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. This may be split into smaller subdomains, such as payment methods, user creation, and user authentication, in the future should complexities arise, but for now they will be one subdomain.

Catalogue

a core subdomain where ACME gathers the essential part of its business proposition, such as the catalogue, 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-frontends decisions framework, the teams identified the migration path with the following micro-frontends (figure 10.9).

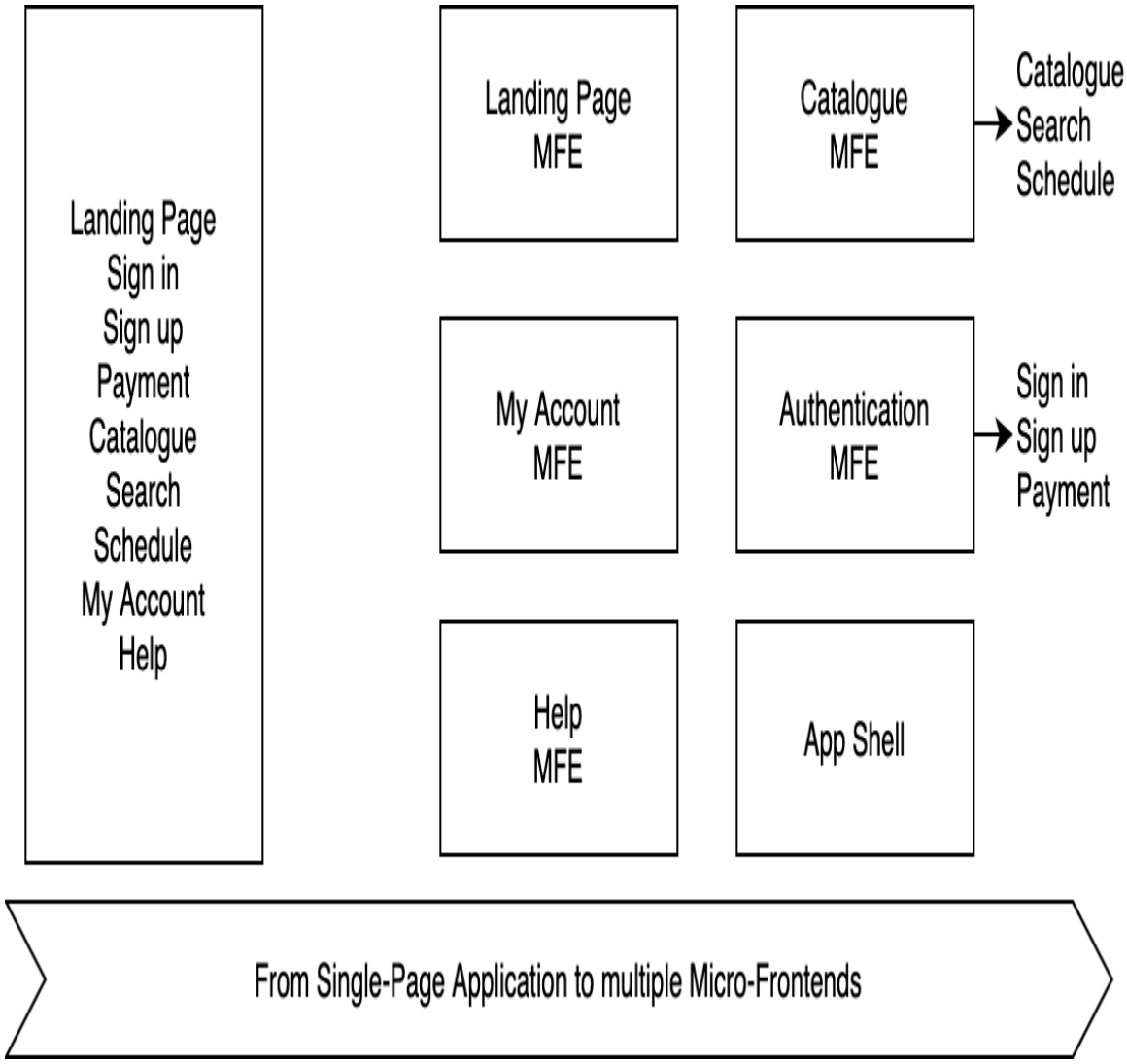


Figure 8-9. Migration path: from SPA to micro-frontends

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 the 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 catalogue, such as moving from sign in to sign up view, retrieving their credentials, and so on.

Catalogue

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's available only for authenticated users. Considering 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 changed 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-frontends 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.

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' life 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 code base of other micro-frontends or even join a different team.

The micro-frontends will be static artifacts, therefore highly cacheable through a content delivery network (CDN), so there's no need of 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 an 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 AWS 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 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 reducing the risk of bugs in production when a new micro-frontends 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 7 and 8), for analyzing the incoming traffic and serving a specific artifact to the application shell, implementing de facto a frontend canary release mechanism at the infrastructure level that won't impact the application code but will run in the cloud (figure 10.10).

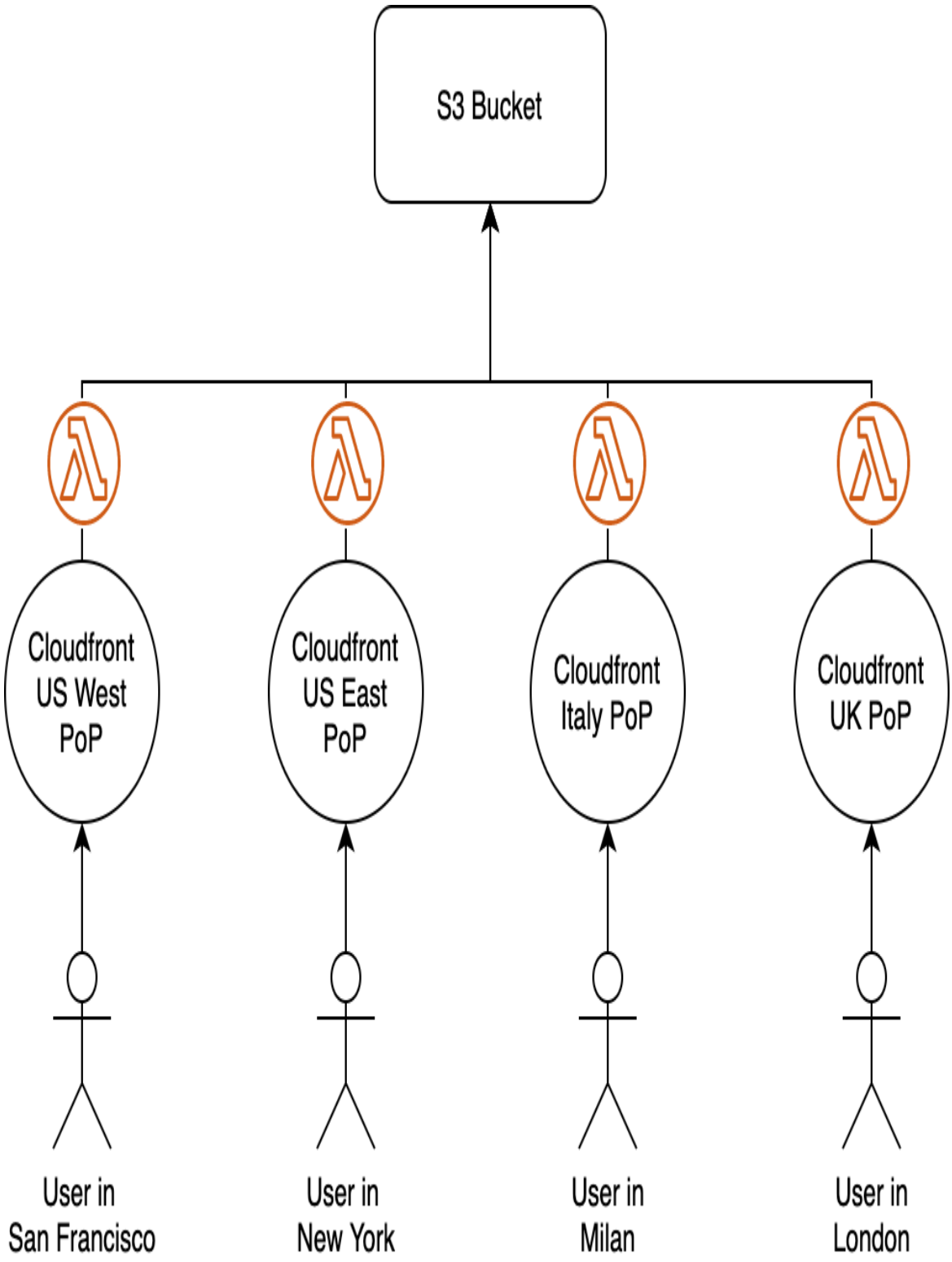


Figure 8-10. Simple infrastructure based on S3 bucket, Lambda@Edge, and a CDN like Cloudfront distribution with point of presence (PoP) available all over the world

Thanks to this implementation, ACME can also apply the strangler pattern (see chapter 7) for gradually moving to micro-frontends while maintaining the legacy application. In fact they can use the application URL for triggering the Lambda@Edge that will serve the legacy or application shell to the user (figure 10.11).

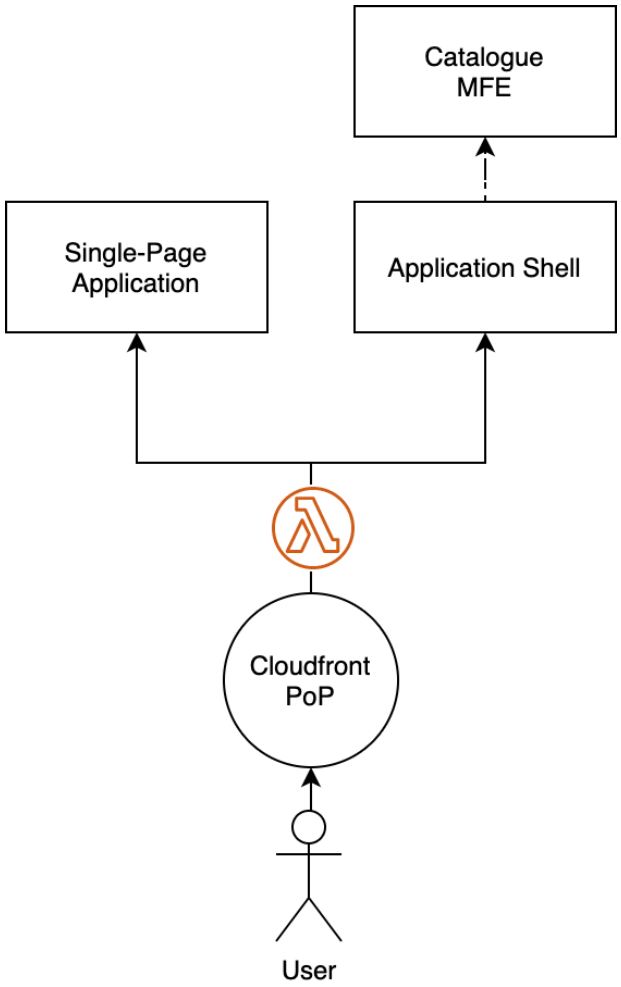


Figure 8-11. Strangler pattern applied at the infrastructure level using Lambda@Edge for funneling the traffic toward either the legacy or micro-frontend application

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-frontends application. Let's clarify this with an example.

Imagine that the catalogue micro-frontend is released first, because at this stage you want all or part of the traffic going toward the micro-frontend

branch (see the 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 catalogue (e.g., www.acme.com/catalogue). This request will be picked up by 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 decide to create a new team to take care of the catalogue 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 catalogue 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 catalogue 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-frontends lifecycle, 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 are:

- Application shell responsibilities
- Integration with the APIs that takes into account the migration to microservices that is 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 a 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 lifecycle 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 returning features 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. Expose the relevant part of the configuration to any micro-frontends appending it to a window object so that every mounted micro-frontend will have access to it.
2. Check business logic: If there is a token in web storage, validate it with the API layer. Route the user to the authenticated area if they're entitled or to the landing page if they're not.
3. Mount 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 lifecycle

methods.

There are three important goals addressed by the application shell exposing these APIs:

1. Exposing the lifecycle 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.
2. 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.
3. 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 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 a 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.

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 bootstrap (global routing), while the sign-up 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-frontends 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 implementing micro-frontends 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 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 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 catalogue 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 catalogue micro-frontend, the micro-frontend will then request the token through the application shell and validate it against the backend (see figure 10.12).

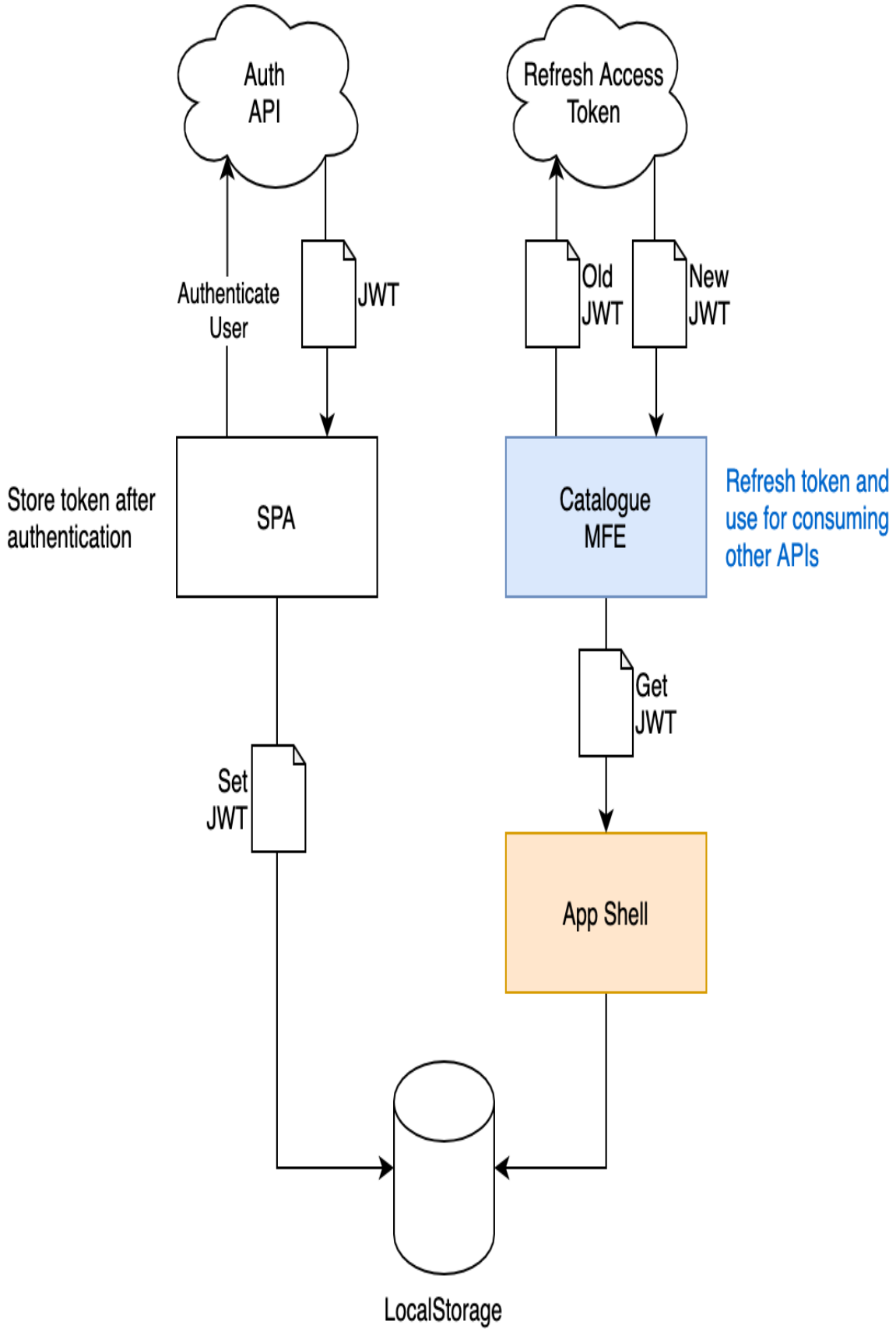

Auth
API
Refresh Access
Token
Authenticate
User
JWT
Old
JWT
New
JWT
Store token after
authentication
SPA
Catalogue
MFE
Refresh token and
use for consuming
other APIs
Get
JWT
Set
JWT
App Shell
LocalStorage

Figure 8-12. 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 (figure 10.13).

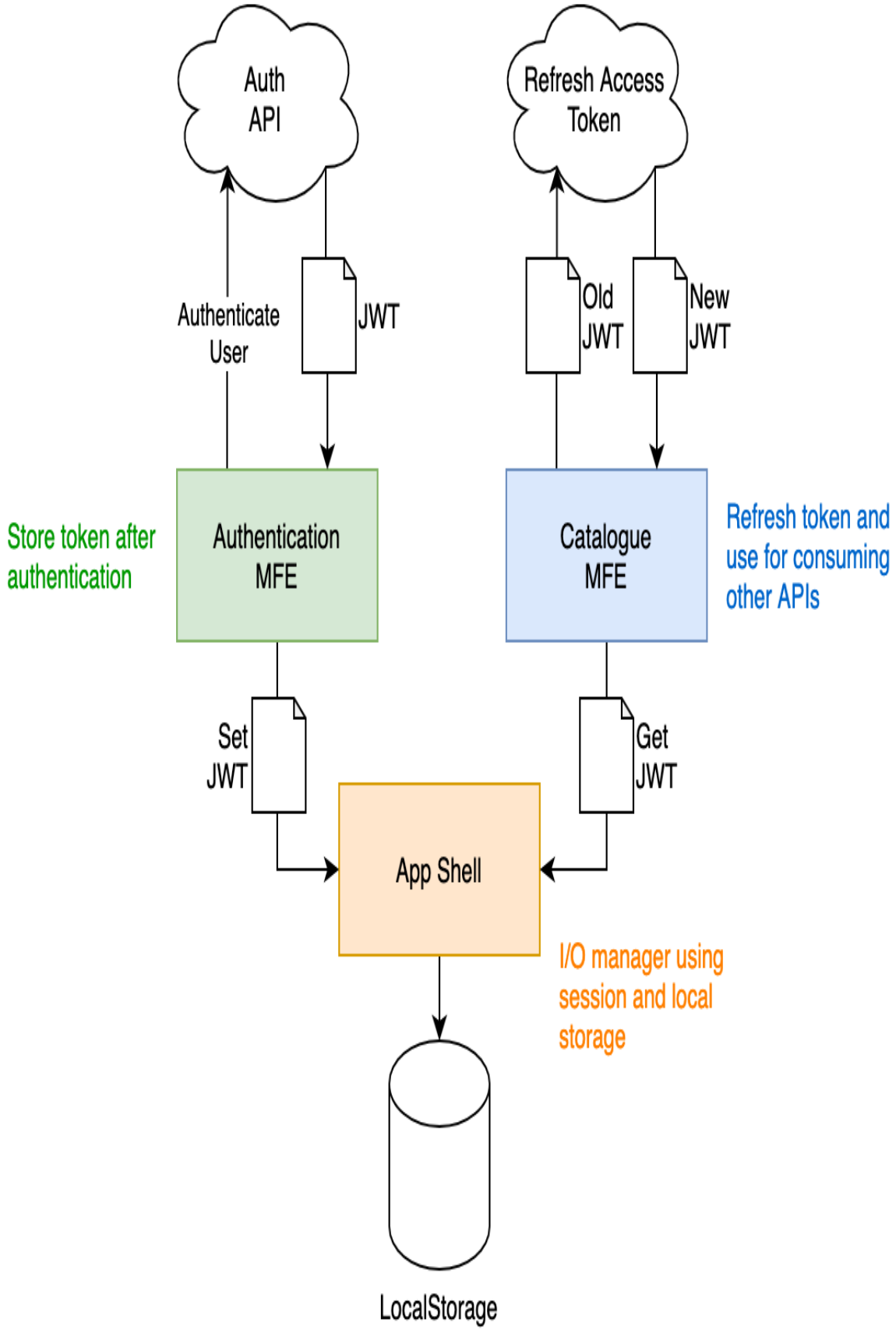

Auth
API
Refresh Access
Token
Authenticate
User
JWT
Old
JWT
New
JWT
Store token after
authentication
Authentication
MFE
Catalogue
MFE
Refresh token and
use for consuming
other APIs
Set
JWT
Get
JWT
App Shell
I/O manager using
session and local
storage
LocalStorage

Figure 8-13. 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 successful and more hackers would be interested in attacking the platform.

CROSS-SITE SCRIPTING (XSS)

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 (figure 10.14).

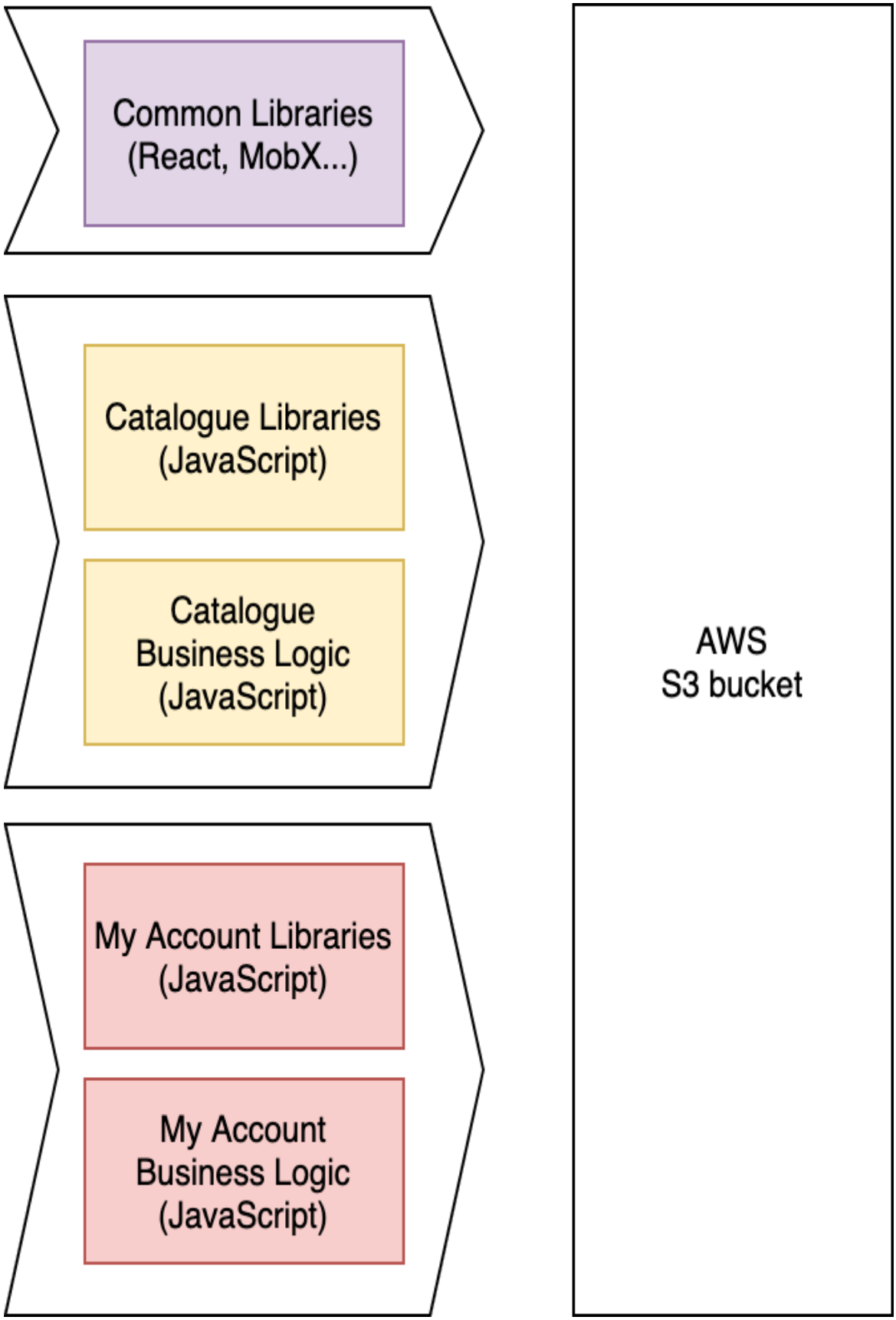

Common Libraries
(React, MobX...)
Catalogue Libraries
(JavaScript)
Catalogue
Business Logic
(JavaScript)
My Account Libraries
(JavaScript)
My Account
Business Logic
(JavaScript)
AWS
S3 bucket

Figure 8-14. 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 libraries 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 user interface (UI) consistency across all micro-frontends, the tech teams and the user experience (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 one. 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 a validation 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, like React or MobX (Figure 10.15).

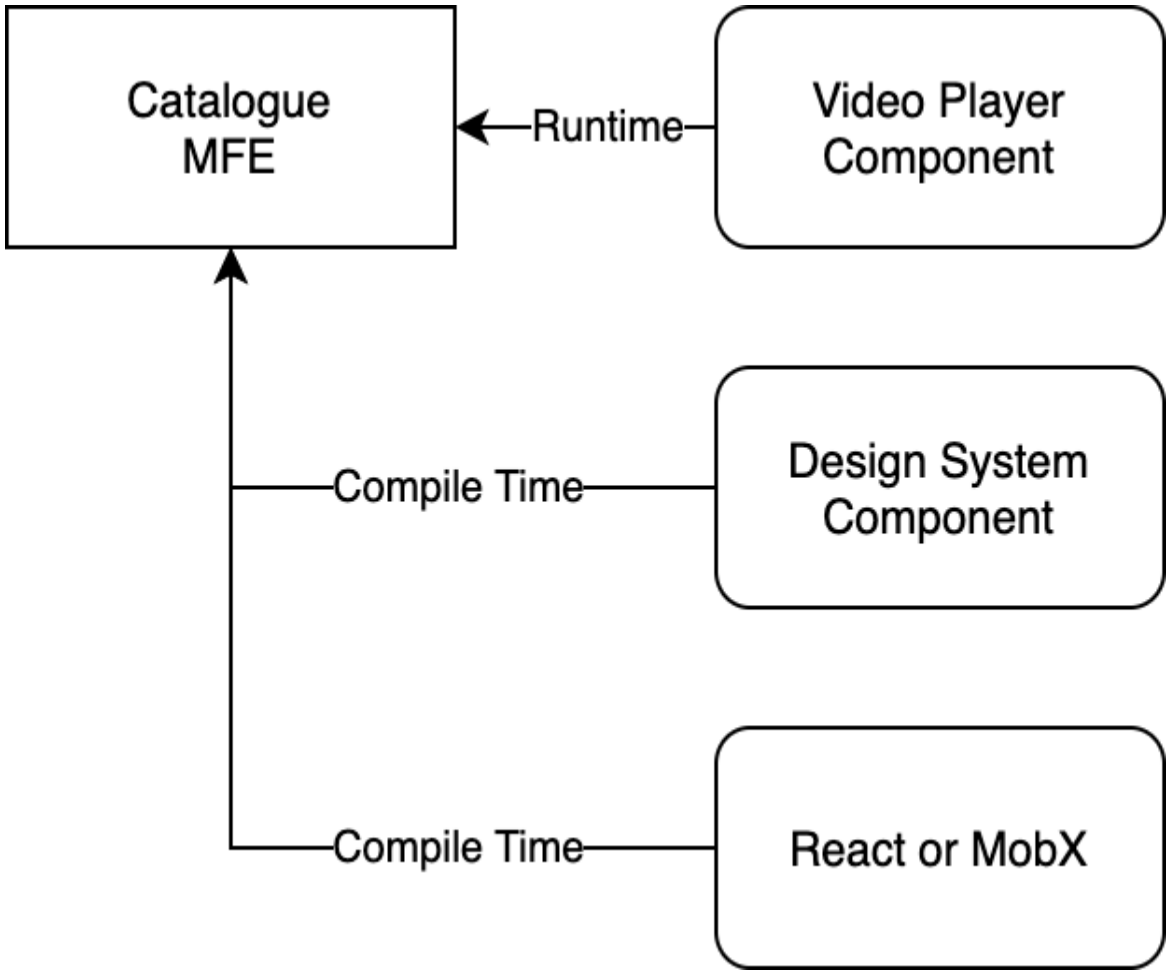


Figure 8-15. In a micro-frontend, complex components owned by a single team are loaded at runtime, while all the others are embedded at compile time. The only exception is the design system due to its modular nature.

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 roadmap.

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 the 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, increment the:

MAJOR version when you make incompatible API changes,

MINOR version when you add functionality in a backwards compatible manner, and

PATCH version when you make backwards compatible bug fixes.

Additional labels for pre-release 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 backwards compatibility. They also only need to upgrade the major version when a breaking change updates the configuration file served by the APIs.

When the artifact request hits 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.

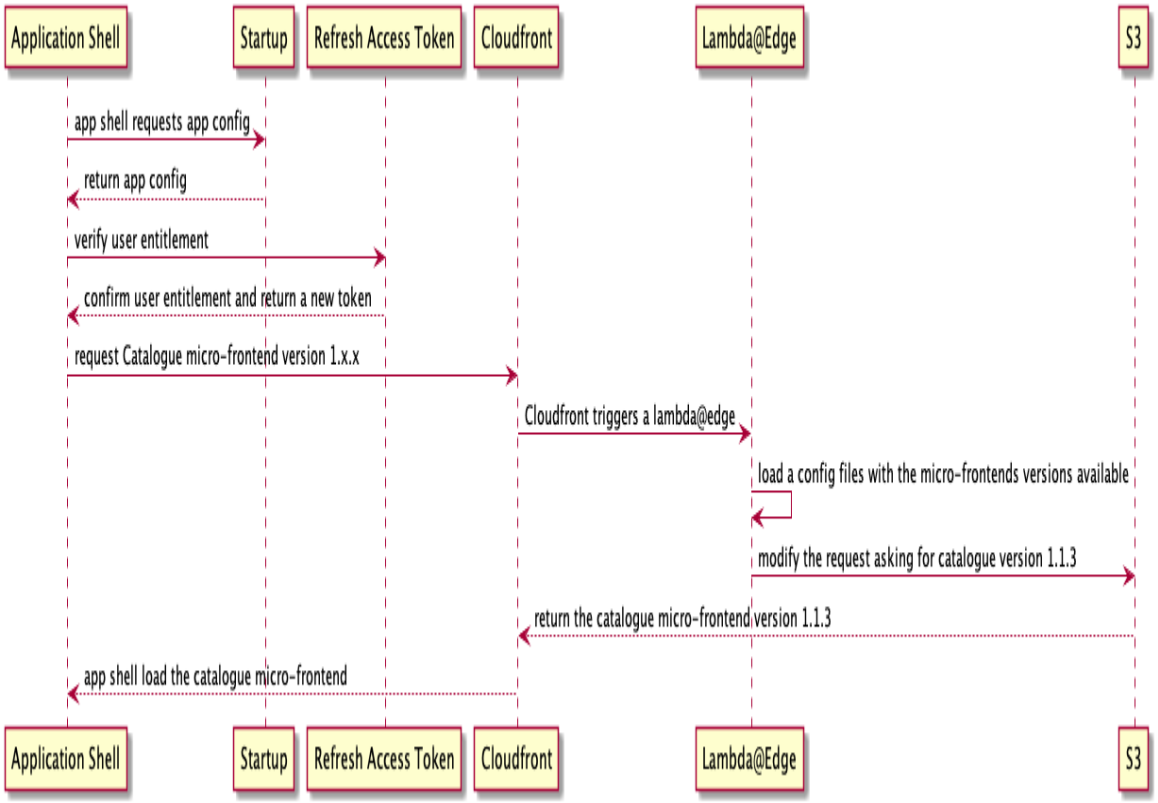


Figure 8-16. Sequence diagram describing how ACME implements canary releases for micro-frontends

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.

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 persist for authenticated users inside their profiles 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 (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.

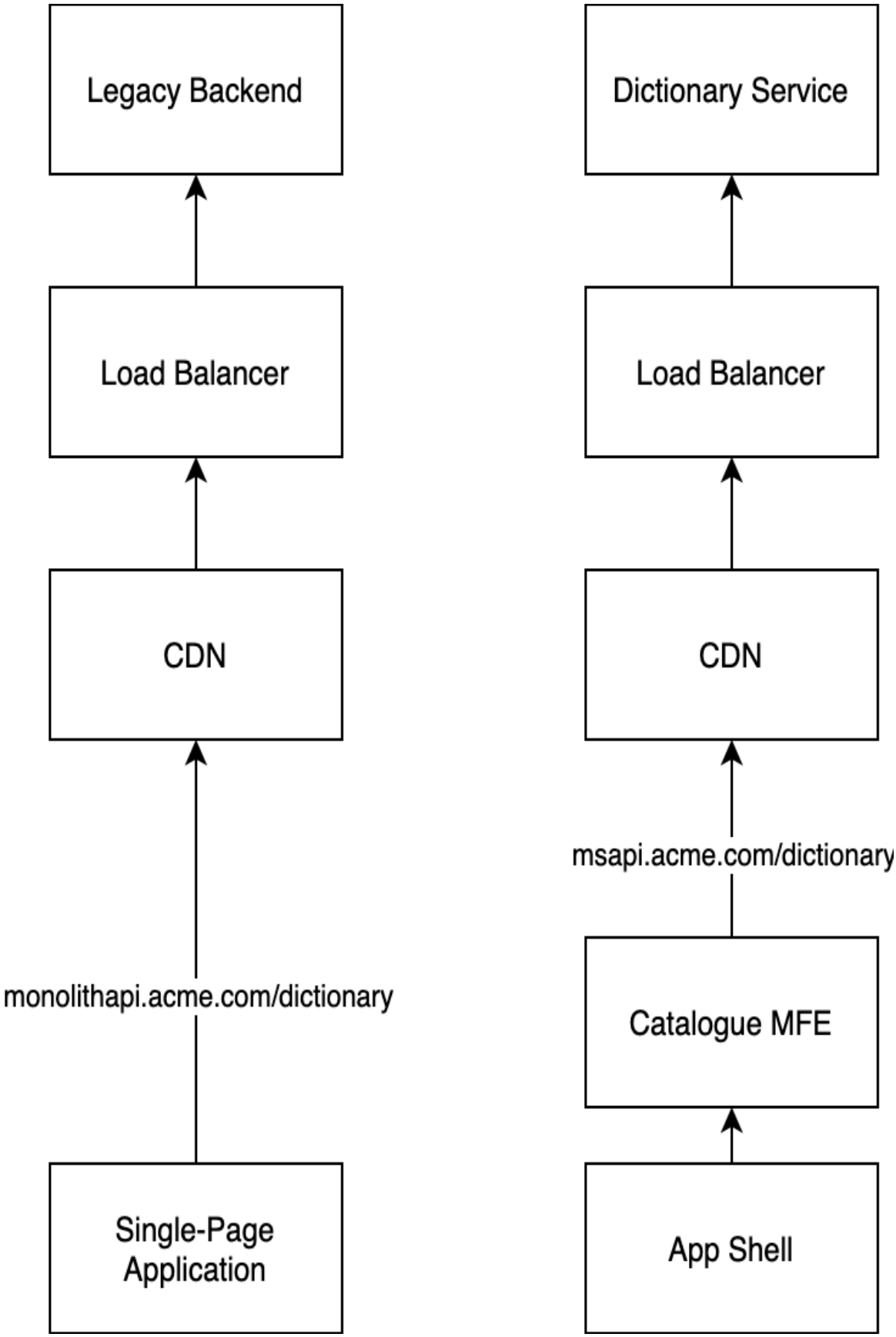

Legacy Backend
Load Balancer
CDN
monolithapi.acme.com/dictionary
Single-Page Application
Dictionary Service
Load Balancer
CDN
msapi.acme.com/dictionary
Catalogue MFE
App Shell

Figure 8-17. 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.

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.

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 too 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 a 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 on 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.

We still have a lot to explore with micro-frontends, including finding the right balance of both technology and people. I find this aspect both extremely challenging and fascinating. Nowadays we have the opportunity to shape this ecosystem with tools, practices, and patterns. The only limit is our imagination.

Appendix A. What does the community think about micro-frontends?

Throughout this book, we have discussed what micro-frontends are, how we can create a micro-frontends architecture, and what the socio-technical impact of this paradigm is.

> **NOTE**
>
> As explained in Chapter 1, many companies implemented micro-frontends in production in many different ways.

I interviewed several industry professionals who are embracing micro-frontends in their organizations to share the breadth of uses.

Following are interviews with tech leads, architects, and developers sharing their development challenges with micro-frontends projects.

Enjoy the interviews!

Please introduce yourself.

I'm Felipe Guizar, a senior software engineer at Wizeline. I have over five years of experience working mainly with content publishing platforms for big media companies.

What is your experience with micro-frontends?

I haven't had the opportunity to implement micro-frontends in a greenfield project yet. However, I've worked on several projects that migrated websites to new technologies.

On those projects, we agreed that migrating from scratch was not a good business approach because we can't deliver new features until we finish the migration. Instead,we decided to look for an approach to gradually migrate the websites and set the basis for integrating new features easily, introducing micro-frontends to our solutions.

What benefits and pitfalls did you encounter in your journey with micro-frontends?

Context:

Frontend technologies (FE) evolve very fast. For instance, I've seen companies that started developing their applications using old frameworks/libraries (AngularJS, BackboneJS) years ago. At some point, new frameworks were introduced in the FE world (React, VueJs), and then developers started learning and choosing those as their main tech stack.

For those companies, it's difficult to keep developers engaged and hire developers who are glad to maintain applications using old technologies.

Benefits:

Micro-frontends help to evolve applications along with the technologies. However, it's not only about bringing the application up to date in terms of technology; it's also about aligning the company tech stack with the developer's expertise.

Pitfalls:

This architecture adds more complexity for managing different pipelines, versioning and integrating the micro-frontends in the host application. Additionally, managing authentication is one of the trickier parts.

Did you contribute to any OSS project related to micro-frontends? If so, which one?

Yes, I'm the creator of Ara Framework.

Ara Framework makes developing micro-frontends easier by integrating any view library (React, VueJS, Svelte) on any web technology/framework, such as NuxtJS, GatsbyJS, WordPress, Flask, and more.

When would you suggest using micro-frontends, and when we should avoid them?

I believe having good knowledge of the business domains is necessary to identify each micro-frontend's boundaries; otherwise, it leads to bad abstractions that can introduce more complexity for integrating and communicating them.

At the end of your last micro-frontends project, what worked and what didn't?

Server-side includes using a proxy server works well for integrating non-authenticated micro-frontends in non-JavaScript frameworks. This approach helps us to migrate partial views from WordPress to React along with the content necessary to render those views.

In the early days of the project, we tried to use authenticated micro-frontends using server-side rendering through the SSL proxy server (Nova Proxy). We faced issues forwarding the authentication header, but we realized we could just render those views from the client side using a kind of client-side includes (Nova Bridge).

https://ara-framework.github.io/website/docs/nova-architecture

https://ara-framework.github.io/website/docs/nova-bridge

Also, centralizing the SSL proxy server created the main point of failure. We tackled it by using the server as a sidecar proxy.

https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar

What are the must-have tools for developers to have an efficient experience with micro-frontends?

A command-line interface (CLI) for scaffolding, running services locally, and easily deploying micro-frontends.

For example, Ara Framework has a CLI to create new micro-frontends for different view libraries, and commands to run Nova Proxy and Nova Cluster locally.

What would you suggest for a person who wants to embrace this architecture?

Evaluate the problems you're trying to solve against the challenges you'll face implementing this architecture.

What was the impact of introducing micro-frontends to developers who didn't know about them? What challenges have you faced?

It encouraged developers to be more involved in the business side, promoting a common language that improves communication. It also encouraged developers to be more involved in making architectural decisions.

However, when micro-frontends were announced in 2019 in the ThoughtWorks Radar as an "adopt technique" in the social networks, there were some misunderstandings about the main goal of micro-frontends. It's still challenging to introduce the architecture to developers biased by those comments.

What was the developer experience like in your last project?

As I mentioned before a CLI tool significantly improves the developer experiences. However, we needed to extend the Ara CLI to automate deployments and provisioning infrastructure necessary to run the application.

Many developers are concerned about performance and design consistency with micro-frontends. What are your suggestions for overcoming these challenges?

I believe performance issues related to loading several micro-frontends views on the same page is a sign that micro-frontends boundaries are not well defined. A user flow can involve several subdomains but the user only interacts with one at a time. For example, when a user navigates to the product listing page (product subdomain) and chooses a product, they finally go to the payment page (payment subdomain).

I recommend lazy-loading each micro-frontend entry point based on routing (routes usually represent a subdomain involved in a user flow).

Obviously, there are cases when we have several subdomains on the same page. For example, in an article page, we have the content itself (content subdomain) and the comments and rating section (content feedback subdomain). Users mainly view an article to read the content. Micro-frontends give us the flexibility to server-side render the content and make it available as soon the user opens the page, and we can let the browser client-side render the other sections on demand (lazy-loading based on scrolling).

Regarding design consistency, I suggest using design systems with reusable components that are domain agnostic. Atomic design is a good methodology for implementing design systems in micro-frontends.

What are the first steps for working with micro-frontends?

Define the micro-frontends' boundaries based on the business subdomains, and identify subdomains that interact together in the user flows.

Can you share the main thing to avoid when working with micro-frontends?

Avoid thinking of micro-frontends as components we can deploy and integrate independently (we can use Module Federation instead). Micro-frontends are views that represent a business subdomain.

What are the main challenges in embracing this architecture from your perspective?

- - Identifying the business subdomains and defining the micro-frontend boundaries
- - Looking for an approach to aggregate them in the host application
- - Handling authentication and authorization

Would you like to share some useful resources about micro-frontends?

- Luca's resources: https://medium.com/@lucamezzalira/micro-frontends-resources-53b1ec7d512a,
- Ara's articles https://ara-framework.github.io/website/blog/
- My articles: https://medium.com/js-dojo/micro-frontends-using-vue-js-react-js-and-hypernova-af606a774602; https://medium.com/js-dojo/serverless-micro-frontends-using-vue-js-aws-lambda-and-hypernova-835d6f2b3bc9; and https://itnext.io/strangling-a-monolith-to-micro-frontends-decoupling-presentation-layer-18a33ddf591b
- This curated list of resources: https://github.com/rajasegar/awesome-micro-frontends

Micro-frontends in three words…

Evolutionary, resilient, agile

==========

Please introduce yourself.

I'm Anthony Frehner. I currently work as a frontend architect, which means I have the privilege of working on just about anything related to the frontend :). I've spoken at React Rally, I'm helping drive the W3C CSS proposal for "vhc" units (name subject to change), and I'm a core-team member of single-spa. You can find me on GitHub at github.com/frehner and Twitter at twitter.com/aahfrena.

What is your experience with micro-frontends?

I was introduced to micro-frontends (MFEs) when I joined CanopyTax, which is the company where Joel Denning and Bret Little (the creators of single-spa and related tools) worked. It was an amazing transition to be in a place where anywhere from six to twelve— depending on the situation— squads were able to work autonomously. (Squads is a term that was taken from the Spotify model, but essentially they're full teams that are centered around a feature instead of being organized by role.)

I left Canopy because another company was interested in setting single-spa up for two reasons: 1) to enable future growth and scale, and 2) to help gradually sunset their legacy application and write new code, while running both side-by-side. I like to think we were successful at both, and the company was later acquired.

I think that was one of the first implementations of single-spa which used SystemJS 3.0 with import maps, so it was fun to see how these new standards greatly simplified the infrastructure required for single-spa. That infrastructure would also later help form the foundation for single-spa's "recommended setup."

My current company is currently investigating using single-spa because it acquired a company that built a tool in a different frontend framework than the one we use, and we would like to potentially integrate the two without having to do a major rewrite of either.

What benefits and pitfalls did you encounter in your journey with micro-frontends?

You've heard this a thousand times, but MFEs aren't a silver bullet. That being said, they have some great pros and a couple of cons to be aware of:

Pros:

- * Team/squad independence. No release trains, code freezes, merge conflicts, long-lived feature branches, QA frustration with testing environments changing/not changing, etc.
- * A shared infrastructure that means that everyone is always on the latest version of a library. For example, designers love it because changes to your style guide go out instantly to the whole app.
- * A (variable) amount of freedom to experiment and let the best technologies rise out of those experiments, instead of being stuck on the legacy technology because of either the fear of trying something and it failing or not wanting to rework your whole app.

- * Lazy loading built in, which means a better and faster user experience.
- * Bundle sizes and speeds comparable to a monolithic app. There's a misconception that MFEs are significantly bigger and slower than a monolith, and that's simply not true.
- * Ability to seamlessly combine legacy software with new software and to integrate software that may have come through an acquisition with in-house software.
- * An amazing developer experience. You only have to run a single MFE locally at a time while all the others can just run their production code. Nearly every single developer that I've worked with has mentioned how much better the DX is for them over a monolithic setup.
- * For single-spa at least, it's based on web standards: ES modules, import maps, etc.
- * Teams that are focused on vertical slices of the app translates into teams that are specialized and know their feature set really well.

However, there are some cons as well:

- * On rare occasions, you will need to do an update on all your MFEs, which is a monotonous task.
- * The shared infrastructure can be a double-edge sword. For example, it's great to have your styleguide update for everyone at the same time, until you accidentally break something and now it's broken everywhere.
- * This is minor, but you need to ensure that your CSS is scoped so that one MFE's CSS doesn't conflict with another's.
- * The infrastructure takes a bit more work than just building a monolith. However, to put it into perspective, in my experience the

vast majority (~90%) of DevOp's time was spent working on the backend's microservices infrastructure

Did you contribute to any OSS projects related to micro-frontends? If so, which one?

I'm a core-team member of single-spa, so I've worked on the library itself as well as the documentation, browser plugin, example repos, and so on, and I am active in the Slack channel. I've been a participant in other projects related to MFEs, namely SystemJS, the Import Map specification, and webpack's Module Federation plugin.

When would you suggest using micro-frontends, and when we should avoid them?

I generally don't recommend them for small teams of one to five developers. That being said, there are still situations where that can make sense. Beyond that, it really comes down to how comfortable (or willing to learn) you are with frontend architecture, such as CI/CD pipelines and webpack/rollup configurations.

At the end of your last micro-frontends project, what worked and what didn't?

Almost everything went well, except for one thing: the decision to support multiple frameworks by using web components. In practice that turned into everyone still only using one framework, but at the cost of additional time and overhead to support web components instead of using framework-specific components. My recommendation is to stick to just one framework when at all possible.

What are the must-have tools for developers to have an efficient experience with micro-frontends?

With single-spa, it's really just about getting the infrastructure up and running; after that, the tools used on a day-to-day basis are exactly the same as a monolith.

For the infrastructure, you need to understand import maps and webpack/rollup configuration and have a way to scope your CSS.

It's also important to understand coupling and cohesion and how they relate to microservices. You want MFEs that are highly cohesive and have low coupling in order to succeed.

What would you suggest for a person who wants to embrace this architecture?

Ask yourself for what purpose do you want this infrastructure: If you're doing it because you're a small team but you want independent deploys, it may not be worth the effort.

What was the impact of introducing micro-frontends to developers who didn't know about them? What challenges have you faced?
When I've had genuine conversations with developers about it, they generally are open to the idea but don't know how to set it all up in practice. After helping them out, they're almost always excited about it all and love it.

I've had conversations with people that are very hesitant about certain aspects (e.g. "the infrastructure is difficult" or "what about consistent styles?") and the conversations generally go well, even if they still decide to not implement them. That's ok! Not everyone is up to the task of spending a couple days to work on infrastructure, but at least some misconceptions were dispelled.

And then there's been conversations with people who are completely unwilling to listen and just want to make memes about MFEs. In those situations, there hasn't been much I could say that would help them understand.

What was the developer experience on your last project?

Excellent. There are open source tools for doing MFE overrides with single-spa, so that makes working on a MFE easy. Additionally, running only a portion of your whole app locally means that when you hit the save

button, webpack takes only a fraction of a second to update instead of multiple seconds to recompile.

Many developers are concerned about performance and design consistency with micro-frontends**, what are your suggestions for overcoming these challenges?**

Regarding performance, you mainly just need to put constraints on your team\, such as saying “We’ll only support Vue, not any other framework.” Just doing that will take care of 90% of your performance issues; the other 10% is no different than taking care of performance for a monolith.

As far as design consistency goes, that’s actually a strawman put up by people who haven’t used modern MFEs. You’ll find that your designs will actually be *more* consistent when you only have to deploy your styleguide once and it’s updated for all apps everywhere, instead of needing to npm-install your style guide in each app.

What are the first steps for working with micro-frontends?

Go to one of the example websites that exist for single-spa, and set up an override for one MFE. Try it out, and see what you like and don’t like. Then find us on GitHub, Twitter, or on Slack and ask questions.

Can you share the main thing to avoid when working with micro-frontends?

Just because you can have multiple frameworks, doesn’t mean you should. It’s still recommended to try and stick to one framework if at all possible.

What are the main challenges in embracing this architecture from your perspective?

Your willingness to be open to new ideas, and to work on infrastructure such as CI/CD pipelines and webpack/rollup. You also need to understand when you should create a new MFE versus adding to an existing one.

Would you like to share some useful resources about micro-frontends?

We try to keep the single-spa website up to date. Also, the single-spa Slack channel is open to anyone, and we frequently talk about tech-related things

besides single-spa in the #randomstuff channel. The book *Building Microservices* is a good reference, even though it's very backend focused.

Micro-frontends in three words…

Enables team independence

Please introduce yourself.

Joel Denning, frontend software dev and independent consultant. I've authored single-spa and maintain a lot of other open source, too. I made more than 4,000 GitHub contributions in the last year.

What is your experience with micro-frontends?

I've implemented micro-frontends at five companies, and consulted with several dozen more. I created several example repositories (github.com/vue-microfrontendss, github.com/react-microfrontends, github.com/polyglot-microfrontends). I talk to people every day in the single-spa Slack workspace and GitHub issue queues about implementing micro-frontends.

Which benefits and pitfalls did you encounter in your journey with micro-frontends?

Pros:

1. 1. Independently deployed micro-frontends are a huge organizational win. This is the primary benefit in my opinion.
2. 2. Incremental migration between frameworks, with "strangler pattern." If you can convert your existing app into a Micro-Frontend, you can start adding new micro-frontends without rewriting the old one. This lets you introduce the new framework without the cost of rewriting everything.
3. 3. Ability to hire developers with a larger range of talents, since they can work in more technologies rather than living with the one set chosen for a monolith.

4. 4. Ability to use the “best tool for the job.” Does that React library solve everything? Use it. Does that Vue library solve everything? Use it.

Cons:

1. 1. Conceptual complexity. It takes a while to explain what’s going on to everyone.
2. 2. Technical complexity. Separate CI processes, in-browser module loader, Module Federation, and so on.
3. 3. Possibility for duplicated dependencies between micro-frontends, which is worse for performance than a single monolithic build. There are solutions to this, but they are often not implemented perfectly.
4. 4. Deployment dependencies between micro-frontends are a new thing to consider that don’t exist if you have a single deployable.

Did you contribute to any OSS project related to micro-frontends? If so, which one?

Yes, single-spa and all its helper projects (single-spa-react, single-spa-vue, single-spa-angular, import-map-deployer, import-map-overrides, systemjs, etc.).

When would you suggest using micro-frontends and when we should avoid them?

Use micro-frontends when:

- - You’re trying to migrate away from an old framework or monolith.
- - You want independent deploys for separate dev teams.
- - You want some level of independent technical decision-making for separate dev teams (which date formatting lib to install, which react css lib, or perhaps even which UI framework).

- - You want to split your UI into highly cohesive, loosely coupled sections. This comes from the Building Microservices O'Reilly book.

Avoid micro-frontends when:

- - You don't want to do micro-frontends.
- - Your monolith is working well for you.
- - There is only one dev on the project.
- - Separate deployments cause more pain than benefit, due to deployment dependencies. This occurs especially when you have very few developers.
- - Your micro-frontends regularly engage in heavy, chatty communication. If the micro-frontends are talking to each other all the time, perhaps you should not be using micro-frontends.
- - Your dev team does not have the technical expertise, time, or desire to manage a more complex system.

At the end of your last micro-frontends project, what worked and what didn't?

What worked:

- - We created a separately deployable project with its own package.json, build, and CI process. This was a huge win over our PHP/Laravel monolith that built react with an old version of node and gulp.
- - We were able to free ourselves from many of the technical decisions of the past.
- - We were able to create a style guide/component library that lets us collaborate with UX a lot easier.

What didn't work:

- - DevOps was very resistant and took a lot of convincing. The new CI pipelines and infrastructure took a long time to build.
- - Some devs confuse micro-frontends with react. They were new to and couldn't tell what things came from what.

What are the must-have tools for developers to have an efficient experience with micro-frontends?

- - import-map-overrides: It allows you to develop one micro-frontend at a time, instead of running all of them locally.
- - single-spa: The most popular open source framework for micro-frontends that I'm aware of.
- - import-map-deployer: A clear way to achieve independent deployments for your separate projects.
- - import maps: A separate name of the micro-frontend from its url. This is important for independent deploys.
- - systemjs: For in-browser module and import maps polyfill support.

What would you suggest for a person who wants to embrace this architecture?

Look at your backend code's architecture and evaluate whether it is working for you. It often makes sense for your backend and frontend architecture to match. If one is a monolith, the other should be. If one is microservices, the other should also be microservices.

Look at how your organization gets things done. If your organization's culture is geared toward product ownership, team autonomy, and distributed decision-making, then micro-frontends might make sense. If it's more of a centralized decision-making process, then it might not make sense.

What was the impact of introducing micro-frontends to developers who didn't know about them? What challenges have you faced?

Where are the lines between micro-frontends and a UI framework? What is doing what? How do the pieces fit together? What does this repo (or that repo) do? What's the mental model for the whole system?

What was the developer experience on your last project?

See https://github.com/joeldenning/import-map-overrides. You do `npm install` and `npm start`. Then you go to a deployed environment and set up an override so that it uses your local version of the micro-frontend instead of the deployed version.

Many developers are concerned about performance and design consistency with micro-frontends. What are your suggestions for overcoming these challenges?

For shared dependencies, see https://single-spa.js.org/docs/recommended-setup#shared-dependencies

For design consistency, create a shared style guide module and/or choose a design system such as bootstrap. See https://single-spa.js.org/docs/microfrontends-concept#types-of-microfrontends and https://github.com/react-microfrontends/shared-dependencies.

What are the first steps for working with micro-frontends?

1. 1. Create a POC to help you decide whether you want to do it.
2. 2. Convert your existing app to be a single micro-frontend. Release that to production.
3. 3. Pull out shared navigation into its own micro-frontend. Release that to production.
4. 4. Implement your next new feature as its own micro-frontend. Release that to production.
5. 5. Pull out a small part of your monolithic app into its own micro-frontend. Release that to production.

Can you share the main thing to avoid when working with micro-frontends?

- - Splitting them up too much or incorrectly, such that they're all highly coupled
- - Shared, single deployment for all your micro-frontends

What are the main challenges in embracing this architecture from your perspective?

1. 1. Converting your existing code into a Micro-Frontend, so that future code can be split into separate micro-frontends.
2. 2. Setting up CI/CD
3. 3. Organizational buy-in, trust, and training

Would you like to share some useful resources about micro-frontends?

This YouTube playlist is great: https://www.youtube.com/playlist?list=PLLUD8RtHvsAOhtHnyGx57EYXoaNsxGrTU.

Micro-frontends in three words…

Microservices for frontends

Please introduce yourself.

Zack Jackson, principal engineer of lululemon.

I focus on distributed JavaScript application architecture and how to scale a company's codebase, teams, and platforms. I'm passionately involved in open source! I created the first code-split SSR system for React, and I'm the creator of webpack 5's flagship feature, Module Federation.

What is your experience with micro-frontends?

I have exclusively built micro-frontend stacks for companies since 2015. The largest stack I've built consisted of 150 separate micro-frontends. It consisted of a shared component library; feature-based components used the component library but most features were deployed independently as a

micro-frontend. The range of what the micro-frontends were made of was pretty wide. Some were a single component, some were full features, and others were whole pages or user flows.

I designed the Starbucks inventory management platform, used by all its stores. This stack consisted of six separate micro-frontend applications with helper services for authentication.

At lululemon, I am building a powerful stack that leverages an AppShell and Module Federation and that enables drag-and-drop refactors as features or code that can be moved between servers with no need for regression or extra engineering time. I've extended Module Federation beyond managing seamless micro-frontend experiences, into analytics, A/B testing, and configuration management—all while remaining standalone and independently deployable at any time providing evergreen code to consumers.

What benefits and pitfalls did you encounter in your journey with micro-frontends?

One pitfall was poor code sharing. Sharing vendor code is manual and primitive, causing centralized dependency on an external set of vendors, and upgrading package versions is complex as breaking changes would require all micro-frontends to be prepared for the upgrade of a shared vendor.

Another was poor UX. When moving between micro-frontends, a page will reload. There are very few solutions to sharing global state or making micro-frontends work as well as a monolithic SPA. Huge amounts of time can go into improving the UX.

The benefits of micro-frontends outweigh the pitfalls at scale. Code can be deployed independently, builds are faster, regressions are easier to run, and the blast radius of a critical failure is well contained. It saves engineering time and company money, as features can be delivered at a fast rate, unlike in a monolith, where the rate of delivery slowly degrades as the codebase increases in size and complexity. Micro-frontends remove the harsh requirement of communication and coordination overhead between teams.

They are also cheaper to run and scale because you can use cheaper, less-powerful servers. Unlike a monolith, you can scale per page or per component on cheaper hardware instead of scaling expensive and powerful hardware to meet the base demands and memory consumption of the entire monolith.

Micro-frontends are far more agile and they safeguard companies against site-wide critical failures. Redundancy layers can be built easier, and teams can model a platform to fit their needs instead of using a one-fits-all model that monolithic platforms enforce.

Did you contribute to any OSS project related to micro-frontends? If so, which one?

Next.js, webpack 5 Core, single-spa, React Static, and Module Federation extensions and enhancements.

When would you suggest using micro-frontends, and when we should avoid them?

Small companies will likely not benefit from the engineering overhead. Larger companies with challenges at scale or companies with multiple teams who rapidly deploy are likely best suited to benefit from distributed JavaScript applications.

Regardless of use case and scale, it's very important to design your platform from the ground up to handle scale. If you foresee rapid scale in the future, designing a system that can be migrated into a distributed application model is key. You'll save time and money by avoiding a full-scale rewrite.

At the end of your last micro-frontends project, what worked and what didn't?

Automatic vendor sharing was challenging, routing between the separate apps and making that route transition seamless were a major challenge, and maintaining authentication sessions and sharing state were very challenging as well. My current micro-frontend project has been designed to avoid these issues by rethinking how applications interface with each other and are designed in general.

What are the must-have tools for developers to have an efficient experience with micro-frontends?

Webpack 5 Module Federation is a massive unlock, single-spa provides a strong orchestration layer, Next.js with a custom AppShell, and yarn workspaces that serve as sub-apps is a robust design pattern, which can enable scale and can integrate with Module Federation if or when needed. Micro is another fantastic tool for creating an ingress to route a user to the correct micro-frontend as well. Leveraging monorepos keeps code organized but will still have the pitfall of having only one master branch, bottlenecking deployments. Semantic-release is vital for micro-frontend architecture, where semver plays an important role in the scalability and reliable code distribution.

What would you suggest for a person who wants to embrace this architecture?

Give Module Federation a try. Most importantly, design a system that supports scale. Think about monorepos, feature binding, how bound a page or feature is to a specific server, and how hard it would be to split some of the app into another micro-frontend at a later point in time. Avoid hard binding to a server; build software that can be easily migrated to a new stack. Globals like shared state should encapsulate a page or feature, keeping it independent and unbound to the server. Moving a page or feature to another server instance should be built in a way that will provide any globals needed out the box, not involve multiple copy-pastes of various parts of the application. Gitlab CI is powerful and a strong contender for sophisticated infrastructure requirements.

What was the impact of introducing micro-frontends to developers who didn’t know about them? What challenges have you faced?

Development time and efficiency usually quadruples. Introducing the pattern has given development teams a better experience and the ability to move more code through the pipeline at a faster rate. Challenges revolved around performance concerns from SRE and getting developers used to working in more than one repo at the same time. In highly granular MFE

stacks, it can be a learning curve to run multiple MFEs locally and to get used to having several IDEs open, depending on what feature is being developed.

What was the developer experience on your last project?

Kubernetes, custom router and auth layers, shared global packages, special script to boot and run all MFE's in one place for full workflow use.

Many developers are concerned about performance and design consistency with micro-frontends. What are your suggestions for overcoming these challenges?

Webpack 5 Module Federation is the best solution to this problem. There are no performance concerns or design consistency issues. Code is shared at runtime; it's evergreen.

A non-webpack-5 solution would be to use *renovate bot* and depend on abstraction for distribution or having the micro-frontends supply a render API to allow other applications to retrieve HTML and other resources over a network call. Ultimately, these are better ways to share feature and vendor code along with automation around upgrading dependencies.

What are the first steps for working with micro-frontends?

Figure out how it's going to scale. Centralize shared code, utilities, and data calls. Make sure the platform layer does not become fragmented when managing several independent servers. Think about routing and how you will map various routes to their MFEs.

Can you share the main thing to avoid when working with micro-frontends?

Not abstracting core code to npm. Copy pasting server infrastructure, which leads to maintenance challenges and fragmentation at the platform level. Either an application shell or server shell should exist, which holds high-level aspects, like auth, user state, tooling, and translation configurations.

What are the main challenges in embracing this architecture from your perspective?

Avoiding UX degradation in favor of DX improvement, neither should be compromised. Thinking out how your MFE stack is going to look and work in one year's time with over 10 stacks: does it architecturally scale?

Would you like to share some useful resources about micro-frontends?

https://medium.com/@ScriptedAlchemy/webpack-5-module-federation-a-game-changer-to-javascript-architecture-bcdd30e02669?source=friends_link&sk=c779636999c8644a7fc0df3aaa96223e

Micro-frontends in three words…

Cheap, flexible, scalable

Please introduce yourself.

My name is Erik Grijzen. I'm from The Netherlands, but currently living in Barcelona, Spain where I work for New Relic, a company that has an observability platform that gives its customers insights on how their systems and software are running in real time.

About three years ago, shortly after I joined the company, I've been working on a new project to build a completely new platform UI, using a micro-frontend architecture. As of today, I'm still working and evolving this project as a lead software engineer on the team and the technical product manager for the UI of the platform.

I've worked as a software engineer for many years in a variety of different companies throughout my career. My focus has always been more on the frontend side of things, which is what I'm most passionate about.

What is your experience with micro-frontends?

My experience with micro-frontends has been solely working on this new platform within New Relic over the last three years. Let me give you some context on how we ended up deciding to use this architecture because I think this is quite interesting and I believe many other companies find themselves in a similar position.

New Relic is a fast-growing company and in the last 10 years, it has built many different products. All these products were built separately as

monolithic single-page applications that were linked together through one common navigation. This approach was very successful until more or less three years ago, when we discovered that we had several problems related to the consistency of our user experience, extensibility of our UI, and the way we did UI development within our company. Let me go over each one of these problems to explain them a bit more in depth.

Many of our customers were starting to have more and more complex systems because they were adapting to a microservices architecture and moving towards the cloud. As a result of this, our users were forced to switch constantly between our separate applications to troubleshoot problems in their systems. Despite all the best efforts of our UI engineers, our products were all working and looking slightly different (in some cases very different), which led to undesirable user experiences. On top of that, switching between applications also caused our users to lose the context of the issues that they were troubleshooting, which was an even bigger problem.

The landscape of technology (software, tools, programming languages, integrations, open-source, etc.) that our customers wanted to instrument and observe was (and still is) exploding. It's almost impossible to keep up, so it was clear that we wanted a new approach where we could provide new functionality for our users in an easy and fast manner. Besides that, we also noticed that some customers had very specific use cases that only applied to them. To also cover these cases, we wanted to provide a way to make our user interfaces programmable so that they could extend the platform for their specific needs.

We also saw that inside our company, many UI engineers were doing a lot of duplicate efforts regarding project setup, tooling, configuration, etc. We wanted to reduce the toil and boilerplate that each team has to go through to build new features so that they could spend more time on building innovative and creative product solutions for our users.

It turned out that a micro-frontends architecture was a perfect fit for us to tackle these problems. Users are demanding more unified product

experiences, but it's not easy to build a product with hundreds of UI engineers at the same time, especially when they are located in different offices around the world and in different time zones. And in our case, allowing customers to build on top of the platform as well made the problem even more difficult. In the end, we decided that a micro-frontend architecture was how we wanted to scale our UI development within our organization.

What benefits and pitfalls did you encounter in your journey with micro-frontends?

In our experience, this architecture has very similar benefits and downsides that you can expect from using a microservices architecture on the backend. However, there are a few exceptions to this due to the nature of how browsers work.

Let's start with the benefits. I believe the main topics that we should cover where we noticed the biggest differences are the following:

- * Team autonomy
- * Small and decoupled codebases
- * Modeled around a business domain
- * Automation and standardization

To scale the UI development inside our organization, the most important thing we wanted to achieve was for teams to work autonomously. They should be able to deploy new code whenever and as many times as they want without depending on any other team. Each team should be cross-functional, meaning they have every role (designer, frontend engineer, backend engineer, QA engineer, etc.) on the team to build the functionality they want, so that they can do a complete end-to-end implementation. This means they have full ownership and can take all the responsibility for one or more related micro-frontends that are part of a specific business domain. This is important because this allows parallel development without slowing down when more and more teams are working on our platform.

To achieve this autonomy from a technical point of view, the micro-frontends must be loosely coupled, so that they don't depend on each other and whenever they interact they should have clear contracts. Each micro-frontend is also small in size so that they are easier to reason about (you don't need to know the whole system), easier to test and you can easily add, change or remove them over time.

Our micro-frontends are modeled around a business domain or subdomain, because this aligns better with the structure of our business. It creates fewer team dependencies, gives teams more autonomy, and improves the communication to make quick decisions so that teams can iterate over features faster. To give a more concrete example, one team could be organized around the domain of NodeJS application monitoring. They are highly specialized and are subject matter experts on that topic, which typically results in higher-quality code and better solutions for our end users.

With our micro-frontend architecture setup, we moved from several monolithic single-page applications to many small micro-frontends that are composed at runtime into one unified platform. This resulted in an explosion of a lot of small codebases, which are each owned by separate teams. It was very important to be prepared for this because this architecture introduces a lot of repetitive work and duplication. That's why it's very important to have the proper infrastructure, tools, and standardization in place. We automated every step in the development process, from project creation and pipeline build to continuous integration and continuous deployment, basically providing everything that the team needs, so they can focus on building functionality that our users love.

Let's move now to some of the pitfalls we encountered using micro-frontends. In our journey the two hardest parts were:

- * UX consistency
- * Performance

Every architecture comes with tradeoffs. I don't think we have a perfect solution to make the UI always consistent and performant, but we try to mitigate the downsides as much as we can. In most cases, this means putting certain constraints in place and reducing the autonomy of teams. For example, we have a constraint in place that every team should use a specific version of the ReactJS library to build their user interfaces. This obviously limits the teams from using any technology they want, but we think it's worth it because this constraint reduces the performance costs a lot for our users. We don't want to limit innovation in the organization, but when we try out new technologies and we carefully evaluate the impact it has on the system, we then update the organization standards and move this innovation to the platform level, so that everyone can benefit from it.

When you have many teams working on the same platform, the consistency of the UI is at risk. To reduce this downside as much as possible, we think it's critical to at least have a design system in place. This won't magically make everything consistent, but I think without it, you will definitely be in a world of trouble. In our experience, this design system is best owned and maintained by one team. This doesn't mean others cannot contribute, but there's one team making sure it aligns with the bigger picture. We've tried an open source model where everyone could contribute to the design system, but this didn't work out for us, because when everyone is responsible for it, nobody is responsible for it. Maintenance work, bug fixes, and keeping everything aligned with the bigger picture are especially hard to do in this setup. To further improve consistency, we also implemented an SDK that all micro-frontends have access to. This SDK has all the UI components from our design system and provides several APIs to standardize certain patterns such as navigating around the platform.

To make sure our platform stays as performant as possible, we have several things in place. First of all, the platform is built with the application shell model which makes it very minimal and fast loading to achieve a good initial perceived performance for the users. When the platform is loaded, we lazy-load the micro-frontends based on the client-side routing, which allows us to only load the minimal JavaScript and CSS necessary to render the

screen. To reduce the payload size of the assets we load and memory consumption of the application, we think it's necessary to deduplicate the frontend dependencies as much as possible. On the platform level, we provide some dependencies (i.e., ReactJS) that we are sure every micro-frontend is going to need. We do that by defining those dependencies as webpack externals so that these don't get bundled up for each micro-frontend. This alone reduces the bundle sizes by an incredible amount. For each repository that contains micro-frontends, we are code-splitting the bundles so that we can lazy-load them incrementally at runtime inside the platform. The last thing we do is provide the before mentioned SDK on the platform level by injecting it in each micro-frontend. This reduces the need to use other NPM dependencies, which should decrease bundle sizes even more.

Did you contribute to any OSS project related to micro-frontends? If so, which one?

No, I'm not contributing to any open source projects at the moment. I looked into several of the bigger micro-frontend frameworks and libraries a while back, but none of them really matched with how we wanted this architecture to look like for our platform.

I've been asked several times if we will open source what we've built at New Relic. Unfortunately, we don't have any plans to do so. I also think that what we have right now is too tailored for our needs and we would have to change it quite a bit before it would make sense to release it to the public.

When would you suggest using micro-frontends, and when we should avoid them?

Using an architecture like this comes with a lot of tradeoffs that have to be evaluated carefully and need to make sense for your project and company. Typically the benefits outweigh the downsides when you need to scale your UI development to a lot of teams, which normally only happens for mid-sized to large companies. So my recommendation is to not use this architecture for small projects or companies. If you're not sure you need

micro-frontends, you most likely don't need them. Just start simple; you can always slowly migrate to micro-frontends over time when there's a need for them.

At the end of your last micro-frontends project, what worked and what didn't?

I never officially completed a project using micro-frontends. We are continuing to evolve the architecture of the platform as the product and organization change over time. I consider it nearly impossible to get the architecture right from the start, so you are guaranteed to encounter things that are not working as you might expect. Our current architecture is very different than the one we started with three years ago.

What really worked well for us, was to set up some ways to regularly communicate and get feedback from other teams. This was very crucial in adjusting over time and refining the balance on several architectural topics. I think we have been too restrictive in some areas and we had to put some more restrictions in other places to get the results that we wanted. We try to keep a close eye on what's not working for us and adjust over time to improve the situation.

What are the must-have tools for developers to have an efficient experience with micro-frontends?

This depends on a lot of things because there are many ways you could implement micro-frontends based on the requirements of the project, its business requirements, and how your company is organized. But generally speaking, you want to make it as easy as possible for teams to be successful, whatever that implies in your context.

At New Relic, we've built a command-line interface that uses our internal infrastructure and tools to give teams everything they need to develop features fast. Almost everything you can think of is automated, from project creation to the final deployment to production. We are in a very competitive market, so for us, it's vital to have a fast time to market, be able to quickly iterate on features based on the feedback from the user and spend as little

time as possible on technical configuration, setup, and other repetitive boilerplate.

What would you suggest for a person who wants to embrace this architecture?

Micro-frontends are not a silver bullet. Just like with any architecture, there are many tradeoffs to be made. You have to find the right balance between those tradeoffs that works best for your project, company culture, and organization. What typically happens is that you go from one or several big codebases to many small codebases. That's why I think it's important as a company to make sure you first have the necessary infrastructure, tooling, and standardization in place to support this architecture before you make the change.

What was the impact of introducing micro-frontends to developers who didn't know about them? What challenges have you faced?

I think people don't talk about this a lot, but introducing micro-frontends can be quite a cultural and organizational change that requires kind of a shift in the way you work. This might come as a surprise sometimes to developers who never worked with such an architecture. For us, this was a change that happened slowly over time, so as more and more teams onboarded to work on top of the new platform, the company and the people slowly transitioned to this new way of working.

Especially for the teams that were building micro-frontends on top of the platform in the early stages, it was not always easy. They didn't always have everything they needed and there were still dependencies and things that were blocking them from completing their work. This has improved a lot over time, with better communication, documentation, resources, and better tooling in place to support developers from day one.

What was the developer experience on your last project?

To build micro-frontends on top of our platform, you have to use a command-line interface. Both our customers and internal teams can use this to extend the platform with all different kinds of micro-frontends. This CLI

automatically takes care of the project setup, pipeline build, continuous integration, and continuous delivery. This allows for very rapid feature development; you can go from idea to production within hours.

When you create a new project it automatically scaffolds the repository with all the required dependencies, structure, configuration, version control, and integration with internal tools. By default there are many NPM scripts configured to take care of typical developer experience, such as local development, linting, prettifying, bundling, testing, and other automated tasks to make sure that what you build will work within the platform. When you do pull requests we automatically generate URLs to test the changes against the platform, which makes code review much smoother. Finally, when a pull request gets merged we automatically create a release for our continuous delivery system, which makes it easy to deploy a specific version to any environment, do rollbacks, and so forth.

Many developers are concerned about performance and design consistency with micro-frontends. What are your suggestions for overcoming these challenges?

I've already explained how we are trying to tackle these two challenges. Like I mentioned before, we don't have a perfect solution, but we try to reduce the downsides as much as we possibly can in the context of how we implement micro-frontends in New Relic.

My suggestion would be to carefully evaluate the importance of these topics; based on that you can define the appropriate constraints on the autonomy of the teams that are building micro-frontends. The more constraints you put in place, the more you can reduce the downsides. You will probably end up with a middle-ground solution that is best suited for your project and organization.

What are the first steps for working with micro-frontends?

It's hard to give specific recommendations on how to start working with micro-frontends because this depends so much on the type of project, culture, organization, and the size of the company. The way we have

organized, set up, and architectured our UI development with micro-frontends might be a complete disaster for another company.

Some general recommendations when you get started:

- * Make sure you find the right balance of trade offs that work for you.
- * Make sure to communicate and get feedback, so you can adjust architectural decisions to overcome challenges.
- * Make sure you have enough infrastructure, tooling, and standardization in place that supports this architecture.

Can you share the main thing to avoid when working with micro-frontends?

The main goal of micro-frontends is to scale the UI development within the organization. So the main thing to avoid is creating dependencies between teams or blocking the development of teams in any way.

What are the main challenges in embracing this architecture from your perspective?

I think the main challenge is to tackle some difficult tradeoff decisions, where you are forced to choose between the autonomy of teams and the user experience of the end users. This is not always easy, but I think with time you will always find a good middle ground for each of these tradeoffs.

Would you like to share some useful resources about micro-frontends?

If you are reading this, you already have the best resource at hand. I would also recommend the talks that you can find on YouTube from Luca, especially for people that are new to micro-frontends.

Micro-frontends in three words...

Scaling UI development

Please introduce yourself.

David Leitner is co-founder of SQUER Solutions, a Viennese software company, and describes himself as an enthusiastic software professional who works on various projects using a bunch of different stacks and environments. He spends much of his time on the frontlines tackling the challenges of scaling software and complex domains. A software engineer with more than 10 years' experience, David prefers his code simple and small instead of clever and edgy. David enjoys sharing his knowledge as speaker at conferences, as a podcast co-host, and as a lecturer for his post-diploma courses at the University of Applied Sciences Technikum Vienna.

In 2016 David was one of the first who dealt with the topic of micro-frontends intensively and proposed his ideas to international conferences.

What is your experience with micro-frontends?

When consulting with our customers, we always stress that microservices are about end-to-end verticals that enable independent deployments of autonomous parts of an application at a high pace. Following this idea, it was clear from the very beginning that we had to somehow make this possibility on the frontend parts of these architectures, as well. We experimented in a dozen of projects with different approaches, including simple ones, like linked applications, but also more sophisticated ideas, like the integration on the client side with web components.

What benefits and pitfalls did you encounter in your journey with micro-frontends?

One big lesson was the impact on the look and feel of huge frontend applications. Thus, a Micro-Frontend architecture must go hand-in-hand with a strong understanding and maturity in design systems. In addition, you also have to deal with the classical issues of distributed systems, like performance and latency. For example, over the years we discovered that a shared caching layer on the frontend is a good idea; it was a game changer for how we designed our micro-frontend architectures. And last but not least, it's nearly always a wise decision to start with a monolith-first approach.

Did you contribute to any OSS project related to micro-frontends? If so, which one?

Unfortunately, I have not actively contributed to one of them so far, mainly because we almost never use off-the-shelf frameworks for our micro-frontend architectures. We try to keep the dependencies small and stuck to basic web standards, like web components, to build micro-frontends.

When would you suggest using micro-frontends, and when we should avoid them?

It's really hard to answer this question without any further context, but, I think it is important that as with every new architecture or technology, micro-frontends should not be an end in itself. Still, in our experience at SQUER Solutions, it shows its benefits mainly when the team size gets too big to work on one codebase in the frontend, when resilience issues start to erase, or when the time to market is below expectations.

At the end of your last micro-frontends project, what worked and what didn't?

I think we started to have the maturity that allowed us to spot the right points where the frontend should be split up. But especially for client-side integration, it's a daily challenge to let the integration module of your micro-frontend architecture not sprawl too big and become a bottleneck.

What are the must-have tools for developers to have an efficient experience with micro-frontends?

In most cases the concept of monorepo makes a lot of sense: a single repo for all the micro-frontend projects, especially to share and ensure consistency for commonly used UI components. Besides this, each module that is used inside the micro-frontend architecture should, of course, strictly follow semantic versioning, and the team should have a common understanding about breaking changes.

What would you suggest for a person who wants to embrace this architecture?

As mentioned before, start with a monolith-first approach. Only start to use micro-frontends once you understand the domain well enough to split them up reasonably.

What was the impact of introducing micro-frontends to developers who didn't know about them? Which challenges have you faced?

I have seen similarities to the introduction of micro services a few years ago, which makes total sense, as in both cases distributed architectures are introduced. The new challenges are therefore to design micro-frontends to be backward compatible and to enforce asynchronous communication over synchronous one.

What was the developer experience on your last project?

I think a solid CI/CD pipeline is essential. In a micro-fronted architecture, most of the complexity moves from a developer's machine to the build-and-deployment process. Thus, a feature in such an architecture is delivered once it's deployed to production, not once it's committed to the version-control system. All the tooling should support and align with this thinking.

Many developers are concerned about performance and design consistency with micro-frontends. What are your suggestions for overcoming these challenges?

Basically, it can be said that these concerns are absolutely justified. The question should be whether other advantages of micro-frontends outweigh these problems for a specific use case. My rule of thumb for performance is the more we strive for performance in a micro-frontend architecture, the more we must move the integration to the client. For design consistency, a mature design system can usually overcome most of the challenges.

What are the first steps for working with micro-frontends?

Don't be too opinionated, search for diverse resources that will help you make decisions. In addition, conference talks are a good source for practitioners' reports and learning from the mistakes others have already made.

Would you like to share some useful resources about micro-frontends?

As mentioned before, I usually like to listen and learn from the experiences of others; conference videos on YouTube are a good way to get those insights. In addition, the micro-frontend introduction on Martin Fowler's blog is a good jump start to this topic. Well, and of course the book you are holding in your hands!

Micro-frontends in three words…

Go for it!

Please introduce yourself.

My name is Philipp Pracht. I work as an architect and product owner at SAP, located in Munich, and I'm a proud father and husband.

What is your experience with micro-frontends?

I'm currently working on project Luigi (luigi-project.io), a technology-agnostic micro-frontend framework for admin and business UIs. Before that, I was working on the user interface part of YaaS (a microservice-centric platform and commerce-as-a-service solution), where we successfully established a micro-frontend architecture back in 2014, before the term "micro-frontends" even existed.

What benefits and pitfalls did you encounter in your journey with micro-frontends?

The main benefit is the efficiency boost for any large-scale UI development landscape with multiple teams. In my previous project, I was part of the team responsible for the UI. There were more than 20 teams developing microservices. After we established a micro-frontend architecture (again, it wasn't called that back then) and asked the service teams to take ownership of their UI parts, I was extremely impressed and surprised by how well it worked out. After a short technical introduction, all teams were able to develop and release completely independently, and we never heard back from most of them. It just worked.

The main pitfalls were not on a technical level but rather with some people who did not really stick to the philosophy of micro-frontends, or even of independent, self-empowered teams in general. Sometimes this led to long-lasting and unproductive discussions.

Did you contribute to any OSS project related to micro-frontends? If so, which one?

Apart from Luigi, which is open source, I contributed to Kyma (kyma-project.io), a platform based on Kubernetes for extending applications with serverless functions and microservices. Luigi started out as a Kyma side project, with the goal of extracting the micro-frontend architecture of Kyma's admin console into a framework so that it can be reused in other applications. Of course, Kyma is now using Luigi.

When would you suggest using micro-frontends, and when we should avoid them?

In general, there is a correlation between the (predicted) functional scope and the benefits of using a micro-frontend architecture. If you are certain your UI will have a fixed set of UI components and will be developed by only one team, then you should go with a conventional approach. For all other scenarios, you should check if there is a micro-frontend framework out there that can help you. Even small projects can benefit from something like Luigi, as it offers some extra features that go beyond a pure micro-frontend framework.

At the end of your last micro-frontends project, what worked and what didn't?

On a technical level, everything worked and there were no major issues.

What are the must-have tools for developers to have an efficient experience with micro-frontends?

Tools for reliable development and testing environments are key. Developers should feel confident at all times that what they are currently implementing will work in the bigger context. For example, in my previous

project we offered a CLI tool with which developers could run an emulated main application with their micro-frontend included.

An excellent IDE is also a must.

What would you suggest for a person who wants to embrace this architecture?

As with any hype, read about the topic first, then lean back and think about it from different angles. Think about the possible impact for people in different roles and try to come to a good understanding of where micro-frontends would be a good fit—not only from a technical point of view but also when it comes to organization structures and people.

What was the impact of introducing micro-frontends to developers who didn't know about them? What challenges have you faced?

From my experience with introducing Luigi, developers were happy with it in general. probably because Luigi helps where help is needed but doesn't impose anything—developers still had full freedom within their boundaries. In my previous project, though, there were situations where developers had problems focusing only on their part.

What was the developer experience on your last project?

In project Luigi, we created a tool called Luigi Fiddle (fiddle.luigi-project.io), a playground where you could try out most of our core functionality. It turned out to be pretty helpful, especially for onboarding new developers.

Many developers are concerned about performance and design consistency with micro-frontends. What are your suggestions for overcoming these challenges?

This depends heavily on the approach you choose, especially the performance topic. For example, Luigi has various mechanisms (like the "viewgroups" concept, caching, preloading) to mitigate performance issues for the end user. With concerns about design consistency, I was a bit surprised when I found out people consider this an issue. I usually ask them

how they ensure design consistency in an app that doesn't use micro-frontends, and the answer is something like "This is not a problem, because we have the same CSS." Eventually most of them realize that you can also share CSSs across different applications. You could also develop Angular components that are not consistent with the rest of the app, so it is the developer who ensures consistency, because he wants his piece of UI looking good in its context.

What are the first steps for working with micro-frontends?

Think about how you want to subdivide your UI, then have a look at Luca Mezzalira's micro-frontends decision framework.

Also, check if there is already a framework out there that fits your requirements.

Can you share the main thing to avoid when working with micro-frontends?

Avoid introducing a monolithic layer somewhere else in your stack, because you lose most of the benefits from micro-frontends. Micro-frontends work best with micro services.

From your perspective, what are the main challenges in embracing this architecture?

The main challenge would be if your organization structure isn't a good fit. Your management has to set up a structure where dedicated units (a single dev team in the easiest case) can independently deliver end-to-end features.

Would you like to share some useful resources about micro-frontends?

All publications from Luca Mezzalira, of course.

There is also a good explanation of micro-frontends on martinfowler.com. And of course you can look at luigi-project.io and our YouTube channel, where you can find luigi-specific content, as well as some general information about micro-frontends.

Micro-frontends in three words…

Divide and conquer!

About the Author

Luca Mezzalira is the VP of Architecture at DAZN with more than 15 years of experience, a Google Developer Expert on Web Technologies and the London JavaScript community Manager. He has the chance to work on cutting-edge projects for mobile, desktop, web, TVs, set-top boxes and embedded devices. Luca is currently designing the DAZN platform with his team, a sports video platform based on the cloud that enables millions of users to watch live and on-demand content. He has long experience working with Micro-Frontends, having introduced the architecture at DAZN for the web platform as well as for living room devices, which has provided a high level of flexibility, delivery speed, and independence for DAZN's distributed teams. Luca is also the author of Front-End Reactive Architectures published by APress

In his spare time, he writes for national and international technical magazines and editors. He is also a technical reviewer for APress, Packt Publishing, Manning Publications, Pragmatic Bookshelf and O'Reilly Media.

Luca was speaker and/or keynote speaker at: O'Reilly media webinars, O'Reilly Software Architecture (New York, San Francisco and London), O'Reilly Fluent (San Jose), O'Reilly Oscon (London), AWS Re:Invent (Las Vegas), QCon (London), SDD (London), Google Developers Summit (Krakow), Google DevFest (London), UXDXConf (Dublin), Frontend Devs Love (Amsterdam), Voxxed Days (Belgrad), JeffConf (Milan), International Javascript Conference (Munich and London), JS Poland (Warsaw), Codecamp (Cluj), Code Europe (Wroclaw), JSDay (Verona), CybercomDev (Łódź), Jazoon Conference (Bern), JDays (Göteborg), Codemotion (Milan), FullStack Conference (London), Bitshift (Bergen), React London UserGroup (London), Scrum Gathering (Prague), Agile Cymru (Cardiff), Scotch on the rocks (Edinburgh and London), 360Max (San Francisco), PyCon (Florence), Lean Kanban Conference (London), Adobe Creative Suite CS 5.5 - Launch event (Milan), Mobile World Congress (Barcelona) and many others