Micro Frontends IN ACTION

Michael Geers

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Micro Frontends in Action
Version 4

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP for *Micro Frontends in Action*. To get the most benefit from this book, you'll want to have some established skills in programming, with experience in HTML5 and CSS, basic knowledge in HTTP and networking and a good understanding of modern JavaScript.

I first started experimenting with micro frontends in 2014 while working on a customer project that used this architecture. Micro frontends is an architectural approach for scaling development in larger projects. Micro frontends, like backend microservices, divides a bigger software system into multiple smaller independent systems. In the frontend, the customer interacts with a composition of these various frontends.

In 2017 I created micro-frontends.org, a globally recognized site where I share my knowledge and experiences on this topic. I have collected documents with different techniques, strategies, and recipes for building modern web apps within multiple teams using various JavaScript frameworks.

In *Micro Frontends in Action*, you'll follow along with a hypothetical e-commerce startup "The Tractor Store" as it adopts the micro frontends architecture. Its development teams test out different frontend integration techniques and solve organizational issues that need to be addressed in a micro frontends architecture. You will learn the benefits and drawbacks of different methods based on practical examples.

By the end of the book, you should be able to:

- Create a web application comprised of different smaller fragments by using integration strategies like AJAX, server-side includes, and web components.
- Decipher how and when micro frontends is a good fit for your organization's technical challenges.
- Construct an individual design system to ensure that the end user gets a consistent look and feel throughout the whole application.
- Determine the best techniques and tools like performance budgets, monitoring, and asset delivery strategies that help you deal with these challenges in a structured manner.

I hope you that you enjoy and find *Micro Frontends in Action* beneficial and that it will occupy an important place on your digital (and physical!) bookshelf.

Please be sure to post any questions, comments, or suggestions you have about the book in the liveBook discussion forum. Your feedback is essential in developing the best book possible.

Thanks again for your interest and for purchasing the MEAP!

—Michael Geers

brief contents

PART 1: GETTING STARTED WITH MICRO FRONTENDS

1 *What Are Micro Frontends?*

2 *My First Micro Frontends Project*

PART 2: ROUTING, COMPOSITION, AND COMMUNICATION

3 *Composition with AJAX & Server-side Routing*

4 *Server-side Composition*

5 *Client-side Composition*

6 *Communication Patterns*

7 *Client-side Routing & The Application Shell*

8 *Composition & Universal Rendering*

9 *Which Architecture Fits My Project?*

PART 3: HOW TO BE FAST, CONSISTENT, AND EFFECTIVE

10 *Asset Loading*

11 *Performance is Key*

12 *User Interface & Design System*

13 *Teams & Boundaries*

14 *Migration, Local Development & Testing*

1 What Are Micro Frontends?

This chapter covers

- Discovering what micro frontends are
- Comparing the micro frontends approach to other architectures
- Pointing out the importance of scaling frontend development
- Recognizing the challenges that this architecture introduces

I've been working as a software developer on many projects over the last 15 years. In this time, I had multiple chances to observe a pattern that repeats itself throughout our industry: Working with a handful of people on a new project feels fantastic. Every developer has an overview of all functionality. Features get built quickly. Discussing topics with your coworkers is straightforward. This changes when the project's scope and the team size increases. Suddenly one developer can't know every edge of the system anymore. Knowledge silos emerge inside your team. Complexity rises: making a change on one part of the system may have unexpected effects on other parts. Discussions inside the team are more cumbersome. Before, team members made decisions at the coffee machine. Now you need formal meetings to get everyone on the same page. Frederick Brooks described this in the book *The Mythical Man-Month* back in 1975. At some point adding new developers to a team does not increase productivity.

Projects often get divided into multiple pieces to mitigate this effect. It became fashionable to divide the software, and thereby also the team structure by technology. We introduced horizontal layers with a frontend team and one or more backend teams. Micro frontends describes an alternative approach. It divides the application into vertical slices. Each slice is built from the database to the user interface and run by a dedicated team. The different team frontend integrate to create a page in the customer's browser. This approach is related to the microservices

architecture. But the main difference is that service also includes its user interface. This expansion of the service removes the need for a central frontend team. Here are the three main goals why companies adopt a micro frontends architecture:

- **Optimize for feature development:** A team includes all skills to develop a feature. No coordination between separate frontend- and backend teams is required.
- **Make frontend upgrades easier:** Each team owns its complete stack from frontend to database. Teams can decide to update or switch their frontend technology independently.
- **Increase customer focus:** Every team ships their features directly to the customer. No pure API teams or operation teams exist.

In this chapter, you'll learn what problems micro frontends solve and when it makes sense to use them.

1.1 The Big Picture

Figure 1.1 is an overview of all the parts that are important when implementing micro frontends. Micro frontends is not a concrete technology. It's an alternative organizational and architectural approach. That's why we see a lot of different elements in this chart - like team structure, integration techniques, and other related topics. Notice the magical *Frontend Integration* box at the top. You can see its contents zoomed in at the bottom of the diagram. We'll go through the complete figure step by step. We start above the dashed line and work our way up.

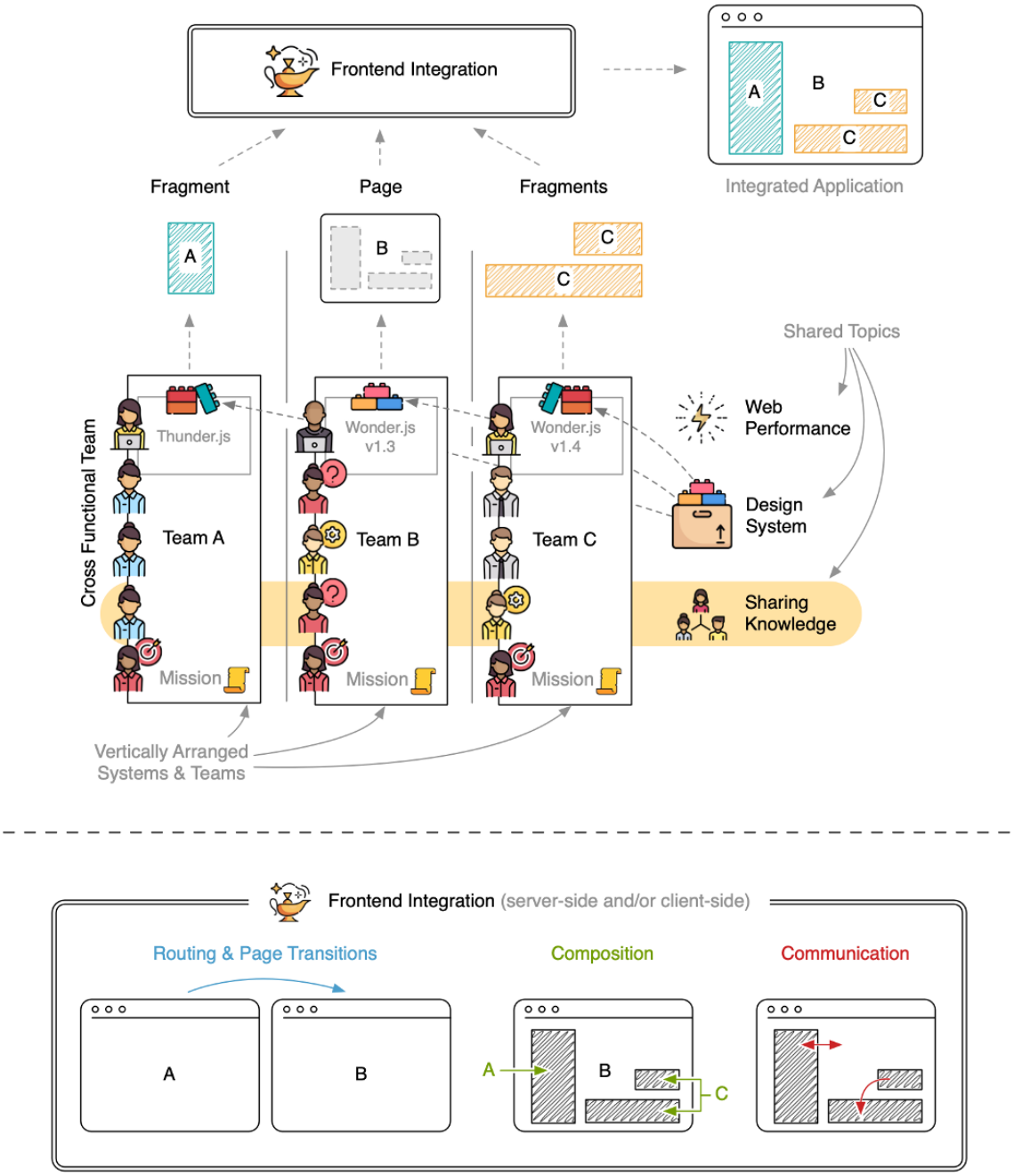


Figure 1.1 Here is the big picture overview of the micro frontends approach. The vertically arranged teams at the bottom are the core of this architecture. They each produce features in the form of pages or fragments. You can use techniques like SSI or Web Components to integrate then into an assembled page that reaches the customer.

1.1.1 Systems & Teams

The three boxes with Teams A, B and C demonstrate the vertically arranged software systems. They form the core of this architecture. Each system is autonomous, which means it can function even when the neighbor systems are down. Every system has its own data-store to achieve this. Additionally, it doesn't rely on synchronous calls to other systems to answer a request.

One system is owned by one team. This team works on the complete stack of the software from top to bottom. In this book, we will not cover the backend aspects like data replication between these systems. Here established solutions from the microservices world apply. We'll focus on organizational challenges and frontend integration.

TEAM MISSIONS

Each team has its area of expertise in which it provides value for the customer. In figure 1.2 you see an example for an e-commerce project with three teams.

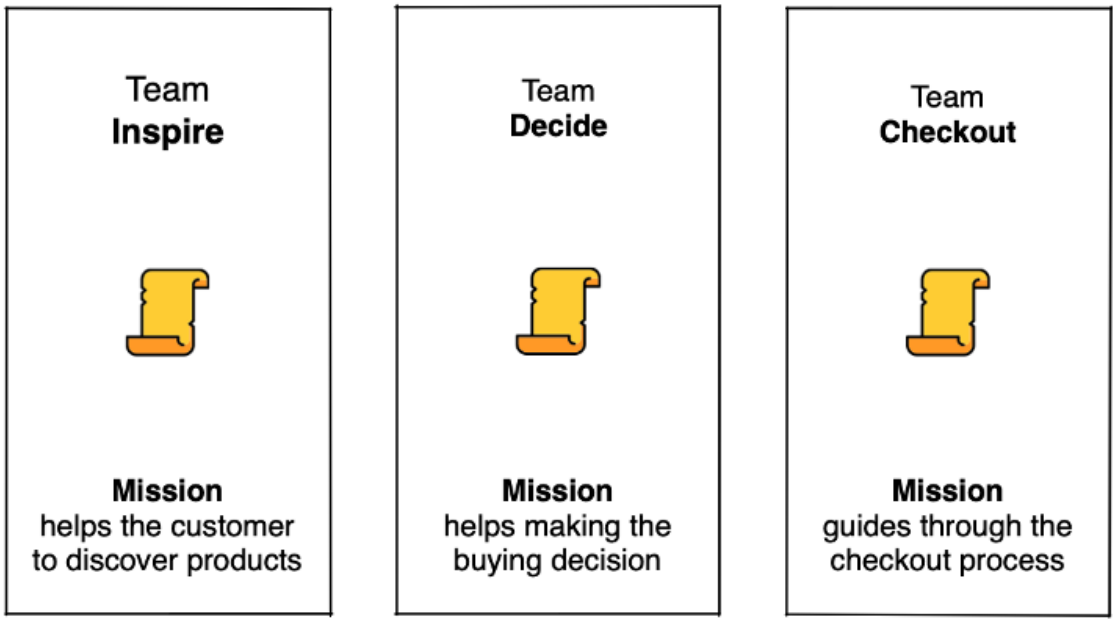


Figure 1.2 You seen an e-commerce example with three teams. Each team works on a different part of the e-commerce shop and has its mission statement that clarifies their responsibility.

Every team should have a descriptive name and a clear user-focused mission. In our projects we align the teams along the customer journey - the stages a customer goes through when buying something.

Team Inspire's mission is, as the name implies, to inspire the browsing customer and to present products that might be of interest.

Team Decide helps in making an informed buying decision by providing excellent product images, a list of relevant specs, comparison tools, and customer reviews.

Team Checkout takes over when the customer has decided on an item and guides her through the checkout process.

A clear mission is vital for the team. It provides focus and is the basis for dividing the software

system.

CROSS FUNCTIONAL TEAMS

The most significant difference, which micro frontends introduce compared to other architectures, is team structure. On the left side of figure 1.3 you see **Specialist Teams**. People are grouped by different skills or technologies. Frontend developers are working on the frontend, experts in handling payment working on a payment service. Business and operations experts also form their own teams. This structure is typical when using a microservices approach.

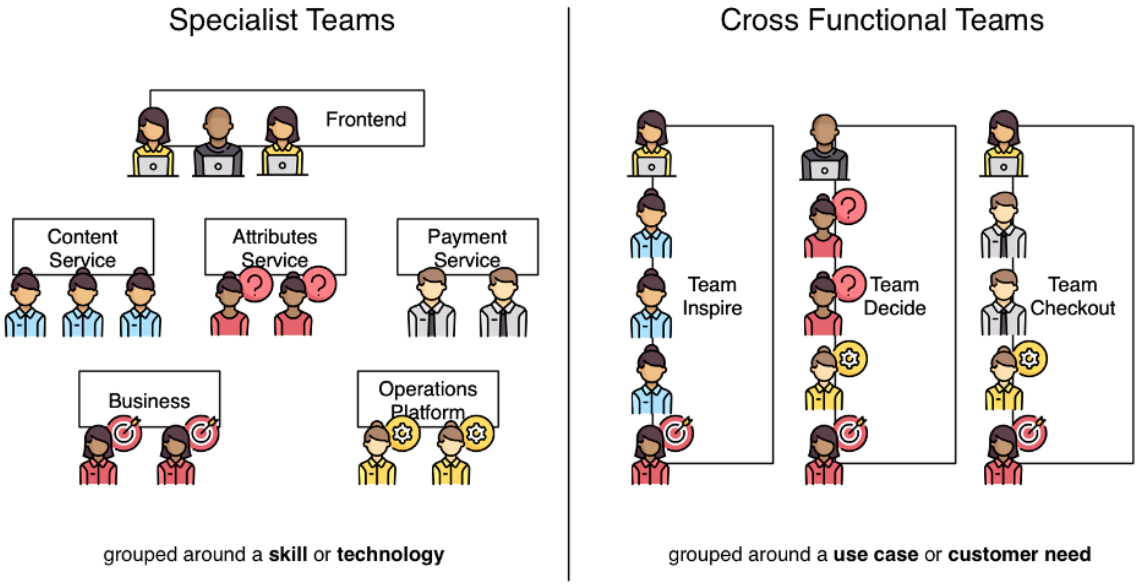


Figure 1.3 Team structure of a microservice style architecture on the left compared with micro frontends teams on the right. Here the teams are formed around a customer need and not based on technologies like frontend and backend.

It feels natural at first sight, right? Frontend developers like to work with other frontend developers. They can discuss the bugs they are trying to fix or come up with ideas on how to improve a specific part of the code. The same is true for the other teams which specialize in a specific skill. Professionals strive for perfection and have an urge to come up with the best solution in their field. When each team does a great job, the product as a whole will also be great, right?

This assumption is not necessarily valid. It's getting more and more popular to build interdisciplinary teams. You have a team where frontend and backend engineers but also operations and business people work together. Due to their different perspectives, they come up with more creative and effective solutions for the task at hand. These teams might not build the best in class operations platform or frontend layer, but they specialize in the team's mission. For

example, they are working on becoming experts in *presenting relevant product suggestions* or *building a seamless checkout experience*. Instead of mastering a specific technology, they all focus on providing the best user experience for the area they work on.

Cross-functional teams come with the added benefit that all members are directly involved in feature development. In the microservice model, the services or operation teams are not involved directly. They receive their requirements from the layer above and don't always have the full picture to know why these are important. The cross-functional team approach makes it easier for all people to get involved, contribute, and, most importantly, **self identify with the product**.

Now that we've discussed teams and their individual systems. Let's move to the next step.

1.1.2 The Frontend

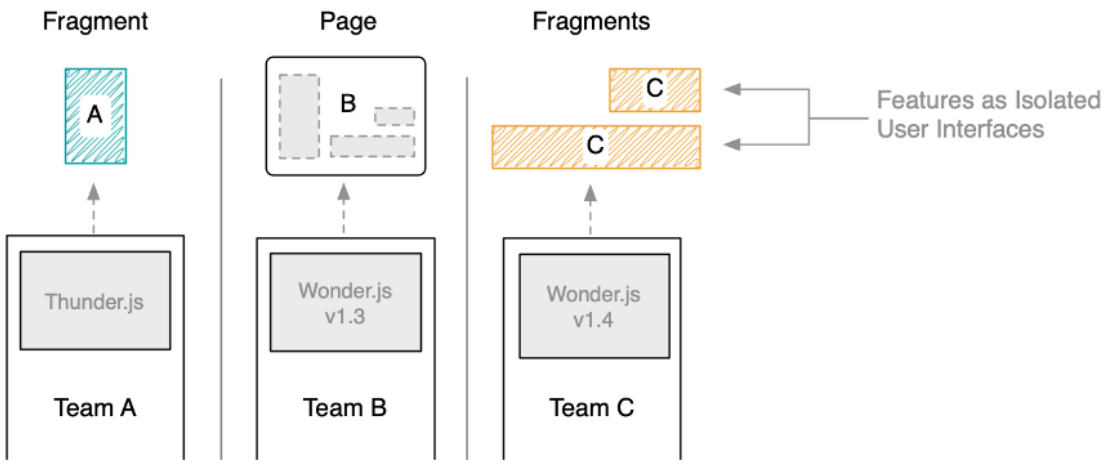


Figure 1.4 This is the middle portion of the big picture as detailed in it's entirety in figure . Each team builds it's own user interface as a page or a fragment.

This is where the actual frontend work gets done. Each team delivers its **own frontend**. A team generates the HTML, CSS, and JavaScript necessary for a given feature. To make life easier, they might use a JavaScript library or framework to do that. Teams don't share library and framework code. Each team is free to choose the tool that fits best for their use case. The imaginary frameworks **Thunder.js** and **Wonder.js** illustrate that.[1] Teams can upgrade their dependencies on their own. *Team B* uses Wonder.js v1.3, whereas *Team C* already switched to Version 1.4.

PAGE OWNERSHIP

Let's talk about pages. In our example, we have different teams that care about different parts of the shop. If you split up an online shop by page types and try to assign each type to one of the three teams, you might end up with something like this:

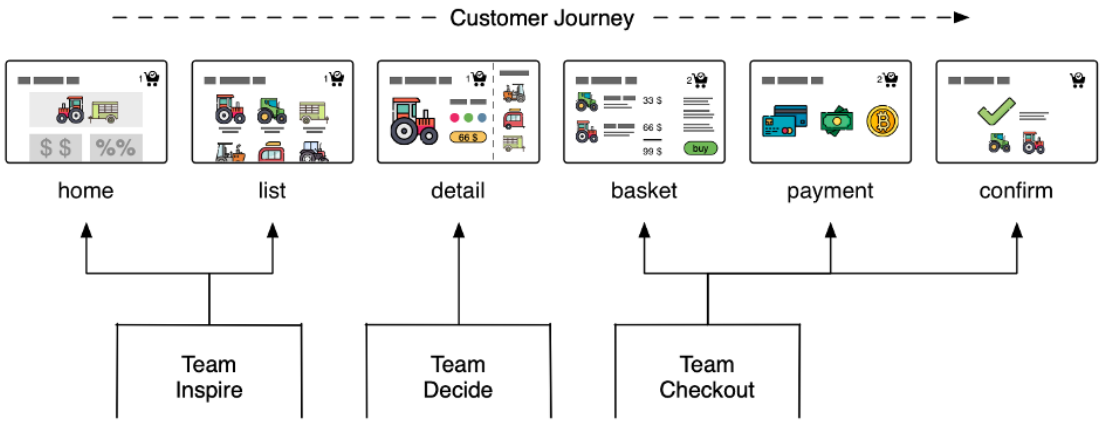


Figure 1.5 Each page is owned by one team

Because the team structure resembles the customer journey, this page type mapping works well. The focus of a homepage is indeed an inspiration, and a product detail page is a spot where the customer makes his buying decision.

How could you implement this? Each team could build their own pages, serve them from their application, and make them accessible through a public domain. You could connect these pages via links so that the end-user can navigate between them. Voila - you are good to go, right? Basically, yes. In the real world, you have requirements that make it more complicated. That's why we've written a book about this. But now you understand *the gist of the micro frontends architecture*:

- Teams can work autonomously in their field of expertise.
- They should be able to use the technology that fits best.
- They should be loosely coupled to other teams (e.g., via links).

FRAGMENTS

The concept of pages is not always sufficient. Typically you have elements that appear on multiple pages like the header or footer. You do not want every team to reimplement them. This is where *fragments* come in.

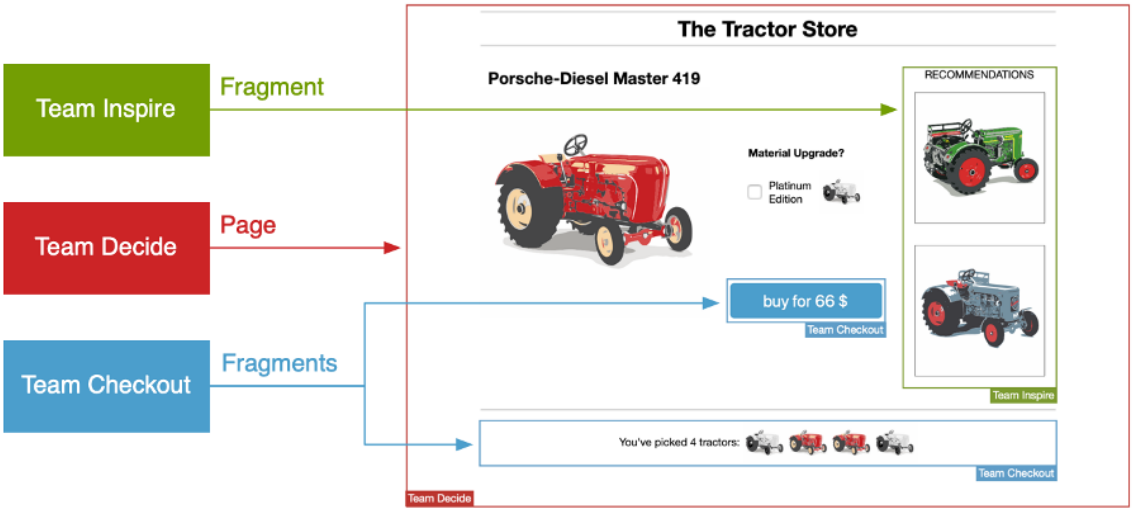


Figure 1.6 Teams are responsible for pages and fragments. You can think of fragments as embeddable mini applications that are isolated from the rest of the page.

A page also often serves more than one purpose and might show information or provide functionality that another team is responsible for. In figure 1.6, you see the product page of *The Tractor Store*. *Team Decide* owns this page. But not all of the functionality and content can be provided by them. The "Recommendations" block on the right is an inspirational element. *Team Inspire* knows how to produce those. The "Mini Basket" at the bottom shows all selected items. *Team Checkout* implements the basket and knows its current state. The customer can add a new tractor to it by clicking the "Buy Button". Since this action modifies the basket, *Team Checkout* also provides this button as a fragment.

A team can decide to include functionality from another team by adding it somewhere on the page. Some fragments might need context information like a product reference for the "Related Products" block. Other fragments like the "Mini Basket" bring their own internal state. But the team, which includes them, does not have to know about state and implementation details of the fragment.

1.1.3 Frontend Integration

Figure 1.7 shows the upper part of our big picture diagram. In this part, it all comes together.

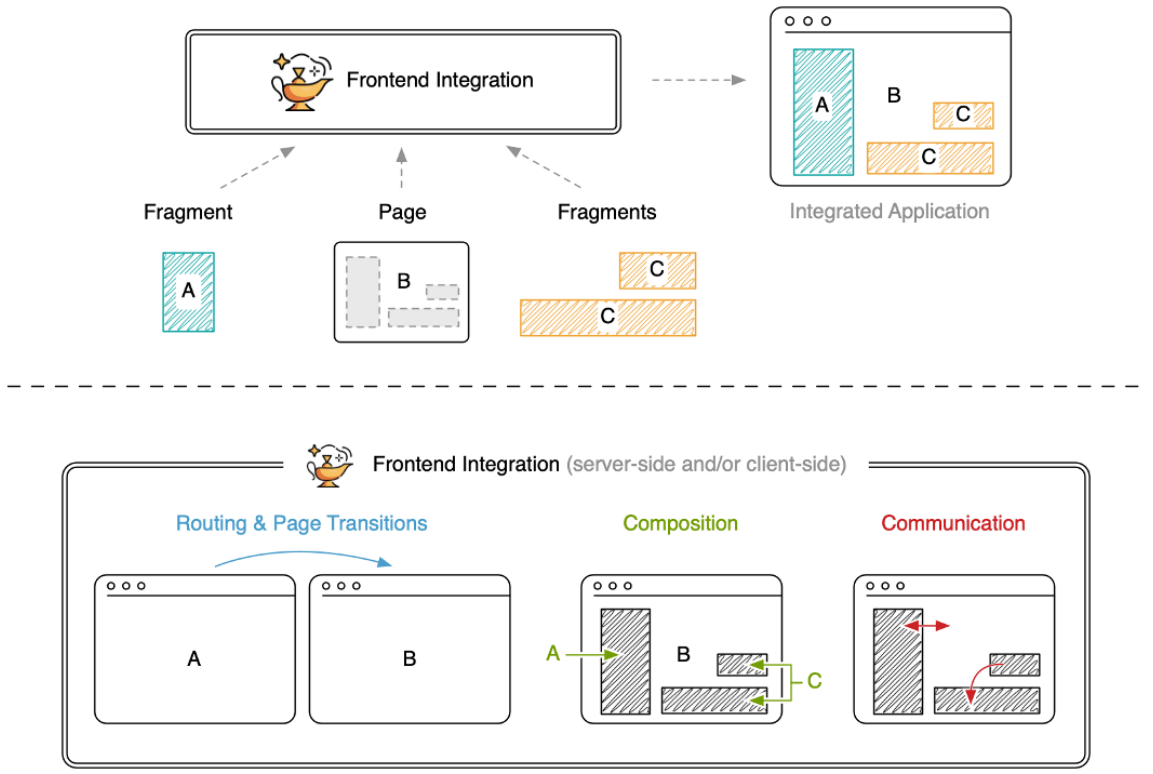


Figure 1.7 The term Frontend Integration describes a set of techniques you use to assemble the user interfaces (pages, fragments) of the teams to an integrated application. You can group these techniques into three categories: routing, composition and communication. Depending on your architectural choices you've different options to solve these categories.

Frontend Integration describes the set of tools and techniques you use to combine the Team's UIs to a coherent applications for the end-user. The zoomed-in *Frontend Integration* box at the bottom of the diagram highlights three integration aspects. Let's go through them one-by-one.

ROUTING & PAGE TRANSITIONS

Here we are talking about integration on page level. We need a system to get from a page owned by Team A to a page owned by Team B. The solutions can be straight forward. You can achieve this by merely using an **HTML link**. If you want to enable client-side navigation, so rendering the next page without having to do a reload, it gets more sophisticated. You can implement this by having a shared **Application Shell** or use a meta-framework like **single-spa**. We will look into both options in this book.

COMPOSITION

The process of getting the fragments and putting them in the right slots is performed here. The team which ships the page typically does not fetch the content of the fragment directly. It inserts a marker or placeholder at the spot in the markup where the fragment should go.

A separate composition service or technique does the final assembly. There are different ways of

achieving this. You can group the solutions into two categories:

- **Server-side composition** with, e.g., SSI, ESI, Tailor or Podium
- **Client-side composition** with e.g. iframes, Ajax or Web Components

Depending on your requirements, you might pick one or a combination of both.

COMMUNICATION

For interactive applications, you also need a model for communication. In our example, the "Mini Basket" should update after clicking the "Buy Button". The "Recommendation Strip" should update its product when the customer changes the color on the detail page. How does a page trigger the update of an included fragment? This problem is also part of frontend integration.

In part two of this book, you'll learn about different integration techniques and the benefits and drawbacks they provide. In chapter 9. Which Architecture Fits My Project? we'll round of this part with some guidance to help you make a good decision.

1.1.4 Shared Topics

Micro frontends is all about being able to work in small autonomous teams that have everything they need to create value for the customer. But some shared topics are essential to address when working like this.

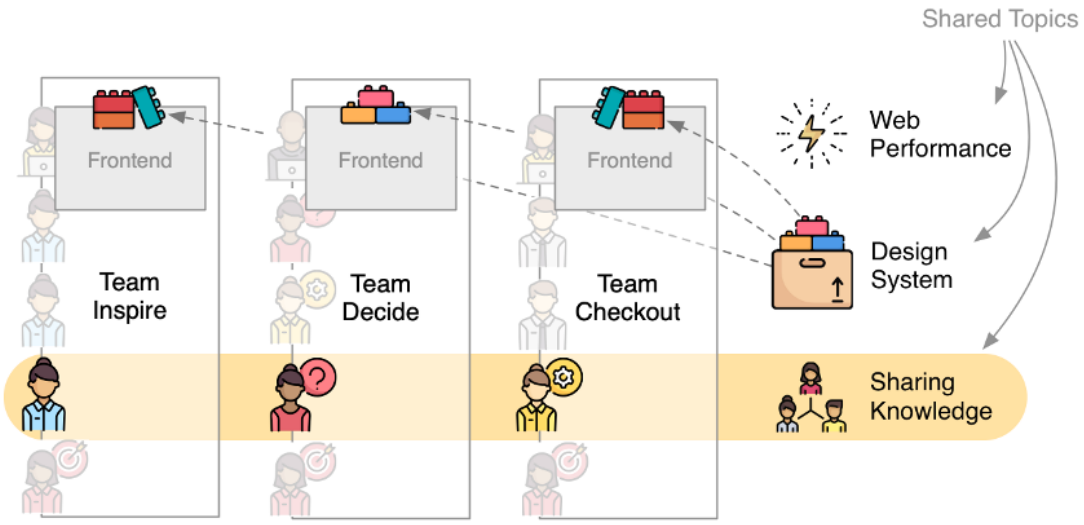


Figure 1.8 To ensure a good end-result and avoid redundant work it's important to address topics like web performance, design systems and knowledge sharing from the start.

WEB PERFORMANCE

Because we assemble a page from fragments of multiple teams, we often end up with more code that our user must download. It's crucial to have an eye on the performance of the page from the beginning. You'll learn useful metrics and techniques to optimize asset delivery. It's also possible to avoid redundant framework downloads without compromising team autonomy. In chapters 10. Asset Loading and 11. Performance is Key we dive deeper into the performance aspects.

DESIGN SYSTEMS

To ensure a consistent look and feel for the customer, it is wise to establish a **Common Design System**. You can think of the design system as a big box of branded LEGOs that every team can pick and choose from. But instead of plastic bricks, a design system for the web includes elements like buttons, input fields, typography, or icons. The fact that every team uses the same basic building blocks brings you a considerable way forward design-wise. In chapter 12. User Interface & Design System you'll learn different ways of implementing a design system.

SHARING KNOWLEDGE

Autonomy is essential, but you don't want information silos. It's not productive when every team builds an error logging infrastructure on their own. Picking a shared solution or at least adopting the work of other teams helps to stay focused on your mission. You need to create spaces and rituals that enable information exchange regularly between teams.

1.2 What problems do micro frontends solve?

Now you have an idea of what micro frontends are. Let's have a closer look at the organizational and technical benefits of this architecture. We'll also address the most prevalent challenges you have to solve to be productive with this approach.

1.2.1 Optimize for feature development

The number one reason why companies choose to go the micro frontend route is to increase development speed. In a layered architecture, multiple teams are involved in building a new feature. Here is an example. Business has the idea to create a new type of marketing banner. They talk to the content team to extend the existing data structure. The content team talks to the frontend team to discuss changes to their API. Meetings are arranged, and the specification is written. Every team plans its work and schedules it in one of the next sprints. When everything works as planned, the feature is ready when the last team finished implementing it. If not, more meetings are scheduled to discuss changes.

Reducing waiting time between teams is micro frontends' primary goal.

With the micro frontends model, all people involved in creating a feature work in the same team. The amount of work that needs to be done is the same. But communication inside a team is much faster and less formal. Iteration is quicker, no waiting for other teams, no discussion about

prioritization.

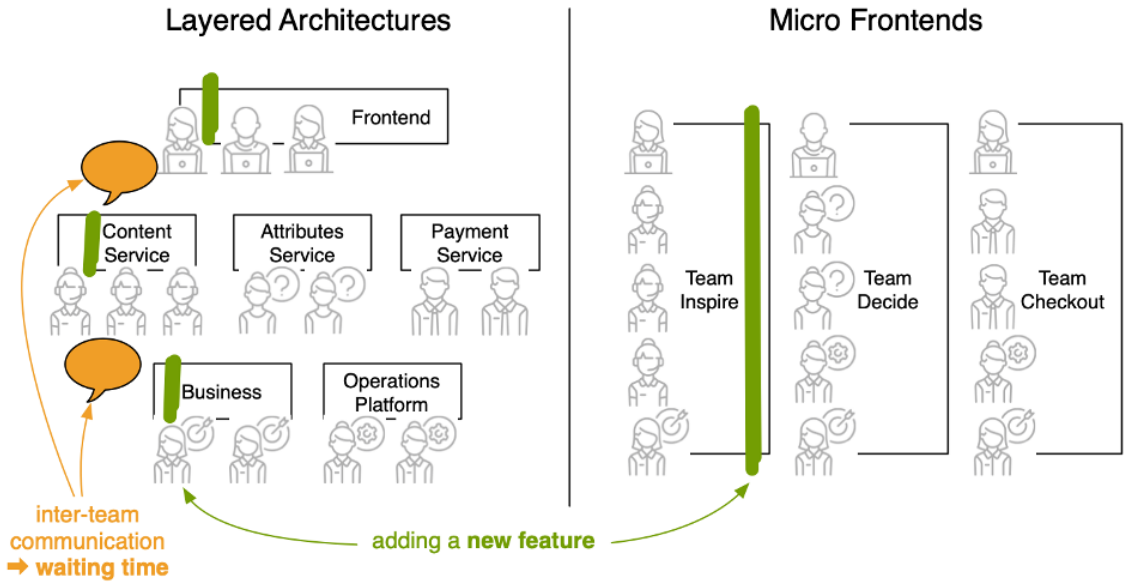


Figure 1.9 The diagram shows what it takes to build a new feature. On the left side, you see a layered architecture. Three teams are involved in building it. These teams have to coordinate and potentially wait for each other. With the micro frontends approach (right), one team can build this feature.

Figure 1.9 illustrates this difference. The micro frontend architecture optimizes for implementing features by moving all necessary people closer together.

1.2.2 No more frontend monolith

Most architectures today don't have a concept for scaling frontend development. In figure 1.10 you see three architectures: The monolith, frontend/backend-split, and microservices. They all come with a monolithic frontend. That means the frontend comes from a single codebase that only one team can work on sensibly.

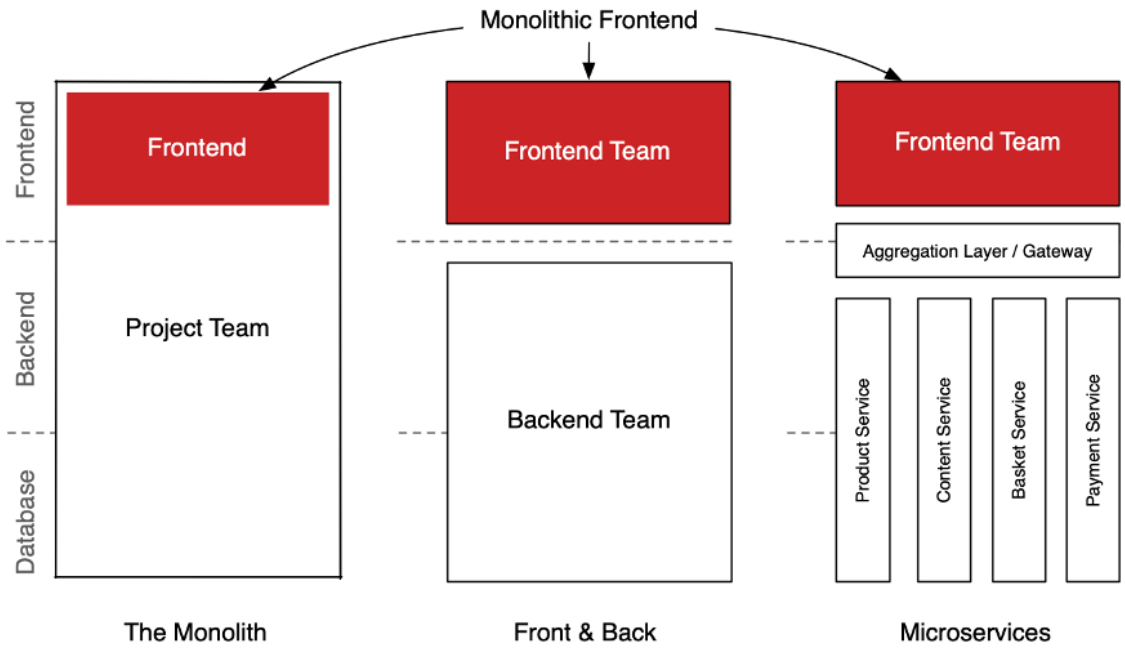


Figure 1.10 In most architectures the frontend is a monolithic system.

With micro frontends, the application, including the frontend, gets split into smaller vertical systems. Each team has its own and smaller frontend. Compared to a frontend monolith building and maintaining a smaller frontend has benefits. A micro frontend …

- is independently deployable.
- isolates the risk of failure to a smaller area.
- is narrower in scope and thereby easier to understand.
- has a smaller codebase that can help when you want to refactor or replace it.
- is more predictable because it does not share state with other systems.

Let's go into detail on a few of these topics.

1.2.3 Be able to keep changing

As a software developer, constant learning and the adoption of new technologies is part of the job. But when you work in frontend development, this is especially true. Tools and frameworks are changing fast. Sophisticated frontend development started in 2005, the web 2.0 era, with Ruby on Rails, Prototype.js, and AJAX, which were essential to bringing interactivity to the before mostly static web.

But a lot has changed since then. Frontend development transformed from "making the HTML pretty with CSS" to a professional field of engineering. To deliver good work, a web developer nowadays needs to know topics like responsive design, usability, web performance, reusable components, testability, accessibility, security, and the changes in web standards and their

browser support. The evolution of frontend tools, libraries, and frameworks enabled us to build higher quality and more capable web applications to meet the rising expectations of our users. Tools like Webpack, Babel, Angular, React, Vue.js, Stencil, and Svelte play a vital role today, but, likely, we haven't reached the end of this evolution yet. Being able to adapt a new technology when it makes sense is an essential asset for your teams and your company.

LEGACY

Dealing with legacy systems is also starting to become a more prevalent topic in the frontend. A lot of developer time gets spent on refactoring legacy code and coming up with migrations strategies. Big players are investing a considerable amount of work in maintaining their large applications. Here are three examples:

- *GitHub* did a multi-year migration to remove their dependency on jQuery.[2]
- *Trivago*, a hotel search engine, made an enormous effort with Project Ironman to rework their complex CSS to a modular design system.[3]
- *Etsy* is getting rid of their JavaScript legacy baggage to reduce bundle size and increase web performance. The code has grown over the years, and one developer can't have an overview of the complete system. To identify dead code, they've built an in-browser code coverage tool that runs in the customers' browser and reports back to their servers.[4]

When you are building an application of a specific size and want to stay competitive, it's essential to be able to move to new technologies when they provide value for your team. This freedom does not mean that it's wise to rewrite your complete frontend every few years to use the currently trending framework.

LOCAL DECISION MAKING

Being able to introduce and verify a technology in an isolated part of your application without having to come up with a grand migration plan for everything is a valuable asset.

The micro frontends approach enables this on a team level. Here is an example: *Team Checkout* is experiencing a lot of JavaScript runtime errors lately, due to references to undefined variables. Since it's crucial to have a checkout process that's as bug-free as possible, the team decides to switch to Elm, which is a statically typed language that compiles to JavaScript. The language is designed to make it impossible to create runtime errors. But it also comes with drawbacks. Developers have to learn the new language and its concepts. The open-source ecosystem of available modules or components is still small. But for the use case of *Team Checkout*, the pros outweigh the cons.

With the micro frontends, approach teams are in full control of their technology stack (*Micro Architecture*). This autonomy enables them to make the decision and switch horses. They don't have to coordinate with other teams. The only thing they have to ensure is that they stay compatible with the previously agreed upon inter-team conventions (*Macro Architecture*). These

might include adhering to namespaces and supporting the chosen frontend integration technique. You'll learn more about these conventions through the course of the book.

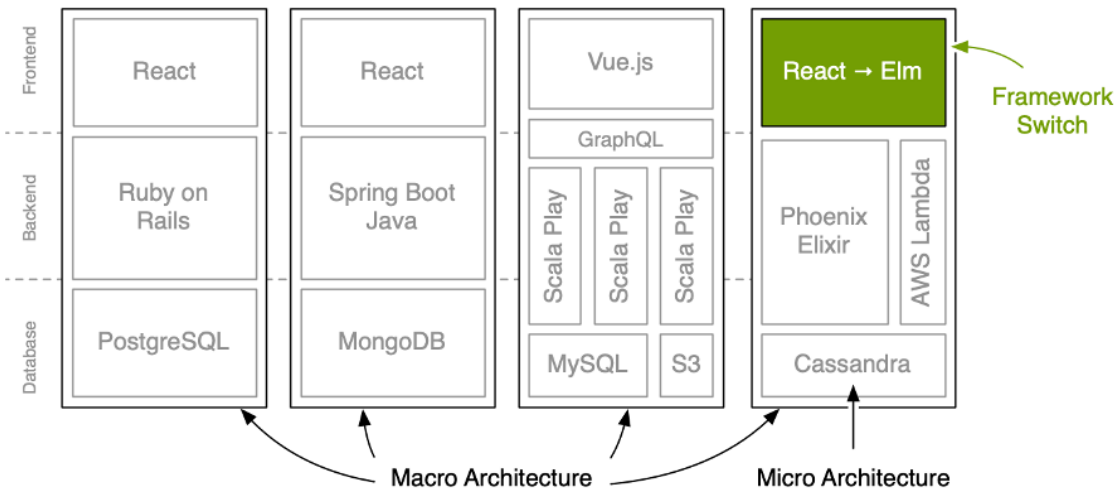


Figure 1.11 Teams can decide about their internal architecture (micro architecture) on their own as long as they stay in the boundaries of the agreed upon macro architecture.

Doing such a switch for a large application with a monolithic codebase would be a big deal with lots of meetings and opinions. The risks are much higher, and the described tradeoffs might not be the same in different parts of the application. The process of making a decision this scale is often so painful, unproductive, and tiresome that most developers shy away from bringing it up in the first place.

The micro frontends approach makes it easier to evolve your application over time in the areas where it makes sense.

1.2.4 The benefits of independence

Autonomy is one of the critical benefits of microservices and also of micro frontends. It comes in handy when teams decide to make more significant changes as described in the section before. But even when you are working in a homogenous environment where everyone is using the same tech stack, it has its advantages.

SELF-CONTAINED

Pages and fragments are self-contained. That means they bring their own markup, styles, and scripts and should not have shared runtime dependencies. This isolation makes it possible for a team to deploy a new feature in a fragment without having to consult with other teams first. An update may also come with an upgraded version of the JavaScript framework they are using. Because the fragment is isolated, this is not a big deal.

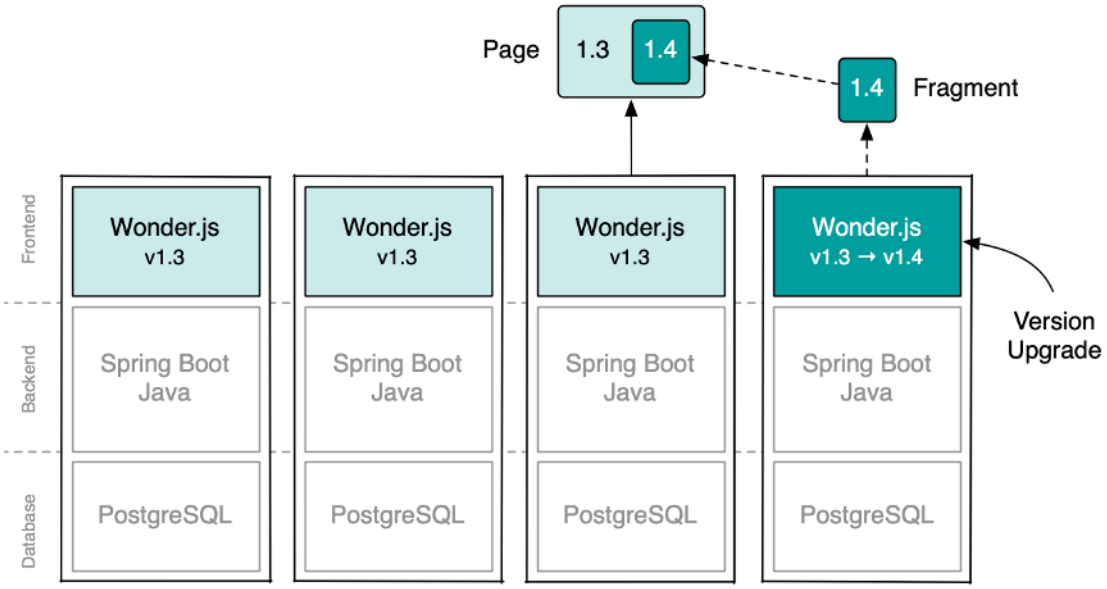


Figure 1.12 Fragments are self-contained and upgradable independently of the page they are embedded in.

At first sight, it sounds wasteful that every team brings their own assets. It's particularly true when all teams are using the same stack. But this mode of working enables teams to move much faster and deliver features more quickly.

TECHNICAL OVERHEAD

Backend microservices introduce overhead. You need more computing resources to, e.g., run different Java applications in their own virtual machine or container. But the fact that the backend-services are in itself much smaller than a monolith also comes with advantages: You can run a service on smaller and cheaper hardware. You can scale specific services by running multiple instances of it and don't have to multiply the complete monolith. You can always solve this with money and buy more or larger server instances.

This scaling does not apply to the frontend code. The bandwidth and resources of your customer's devices are limited. However, the overhead does not scale linearly with the number of teams. It heavily depends on how teams build their applications. In chapter 11. Performance is Key we will explore metrics to qualify and learn techniques to mitigate these effects. But it's safe to say that the team isolation comes with an extra cost.

So, why do we do this at all? Why don't we build a large React application where every team is responsible for different parts of it? One team only works on the components of the product page The other team builds the checkout pages. One source code repository, one React application.

SHARED NOTHING

The reasoning behind this is the realization that communication between teams is expensive - really expensive. When you want to change a piece that others rely on and be it just a utility library, you have to inform everyone, wait for their feedback, and maybe discuss other options. The more people you are, the more cumbersome this gets.

The goal is to share as little as possible to enable faster feature development. Every shared piece of code or infrastructure has the potential for creating a non-trivial amount of management overhead. This approach is also called *Shared Nothing Architecture*. The *Nothing* sounds a little bit harsh, and in reality, it's not that black and white. There are common parts like web fonts that are safe to share between teams.

1.2.5 The downsides of micro frontends

As stated earlier, the micro frontends approach is all about equipping autonomous teams with everything they need to create meaningful features for the customer. This autonomy is powerful but does not come for free.

REDUNDANCY

Everyone who studies computer science got trained to minimize redundancy in the systems they create. Be it the normalization of data in a relational database or the extraction of similar pieces of code into a shared function. The goal is to increase efficiency and consistency. Our eyes and minds have learned to find redundant code and come up with a solution to eliminate it.

Having multiple teams side-by-side that build and run their own stack introduces a lot of redundancy. Every team needs to set up and maintain its own application server, build process, continuous integration pipeline, and might ship redundant JavaScript/CSS code to the browser. Here two examples where this is an issue:

- A critical bug in a popular library can't be fixed in one central place. All teams that use it must install and deploy the fix themselves.
- When one team has put in the work to make their build process twice as fast, the other teams don't automatically benefit from this change. This team has to share this information with the others. The other teams have to implement the same optimization on their own.

The reasoning behind this shared-nothing architecture is that the costs associated with these redundancies are smaller than the negative impacts that inter-team dependencies introduce.

CONSISTENCY

This architecture requires all teams to have their own database to be fully independent. But sometimes one team needs data that another team owns. In an online store, the product is a good example of this. All teams need to know what products the shop offers. A typical solution for this is data replication using an event bus or a feed system. One team owns the product data. The other teams replicate that data regularly. When one team goes down, the other teams are not affected and still have access to their local representation of the data. But these replication mechanisms take time and introduce latency. Thereby changes in price or availability might be inconsistent for brief periods of time. A promoted product with a discount on the homepage might not have this discount in the shopping cart. When everything works as expected, we are talking about delays in the region of milliseconds or seconds, but when something goes wrong, this duration can be longer.

It's a tradeoff that favors robustness over guaranteed consistency.

HETEROGENEITY

Free technology choice is one of the most significant advantages that micro frontends introduces, but it's also one of the points that is discussed controversially. Do I want all development teams do have a completely different technology stack? It makes it harder for developers to switch from one team to another or even exchange best practices.

But just *because you can* does not mean that *you have to* pick a different stack. Even when all teams opt to use the same technologies, the core benefits of autonomous version upgrades and less communication overhead remain.

I've experienced different levels of heterogeneity in the projects I've worked on. From "Everyone uses the same tech." to "We have a list of proven technologies. Pick what fit's best and run with it.". You should discuss the level of freedom and tech-diversity that is acceptable for your project and company upfront to have everyone on the same page.

MORE FRONTEND CODE

As stated earlier, sites that are built using micro frontends typically require more JavaScript and CSS code. Building fragments that can run in isolation introduces redundancy. That said, the required code does not scale linearly with the number of teams or fragments. But it's extra essential to have an eye on web performance from the start.

1.3 When do micro frontends make sense?

As with all approaches, micro frontends are not a silver bullet and won't magically solve all your problems. It's essential to understand the benefits and also the limitations.

1.3.1 Good for medium to large projects

Micro frontends is a technique that makes scaling projects easier. When you are working on an application with a handful of people scaling is probably not your main issue. The *Two-Pizza Team Rule* Amazon CEO Jeff Bezos propagated, is an indicator for a right team size.[5] It says that a team is too big when two large pizzas can't feed it. In larger groups, communication overhead increases, and decision making gets complicated. In practice, this means that the perfect team size is between 5 to 10 people.

When the team exceeds ten people, it's worthwhile considering a team split. Doing a vertical micro frontend style split is an option you should look into. I've worked on different micro frontends projects in the e-commerce field with 2 to 6 teams and 10 to 50 people in total. For this project size, the micro frontends model works pretty well. But its not limited to that size.

Companies like Zalando, IKEA, and DAZN use this end-to-end approach to a much larger scale where every team is responsible for a more narrow set of features. In addition to the feature teams, Spotify, e.g., introduced the concept of *Infrastructure Squads*. They act as support teams that build tools like A/B-testing for the feature teams to make them more productive. In chapter 13. Teams & Boundaries, we'll dive deeper into topics like this.

1.3.2 Works best on the web

Though the ideas behind micro frontends are not limited to a specific platform, it works best on the web. Here the openness of the web plays its strength.

NATIVE MONOLITH

Native applications for controlled platforms like iOS or Android are monolithic by design. Composing and replacing functionality on the fly is not possible. For updating a native app, you have to build a single application bundle that's then submitted to Apple's or Google's review process. A way around this is to load parts of the application from the web. Embedded browsers or WebViews can help to keep the native part of the app to a minimum. But when you have to implement native UI, it's hard to have multiple end-to-end teams working on it without stepping on each other's toes.

It's, of course, always possible that every vertical team could have a web frontend and also expose their functionality through a REST API. You could build other user interfaces like native apps on top of these APIs. A native app would then reuse the existing business logic of the teams. But it would still form a horizontal monolithic layer that sits on top. So, if the web is your target platform, micro frontends might be a good fit. If you have to target native as well, you have to make some sacrifices. In this book, we will focus on web development and not cover strategies to apply micro frontends for building native applications.

MULTIPLE FRONTENDS PER TEAM

A team is also not limited to only have one frontend. In e-commerce, it's common that you have a front-office (customer-facing) and a back-office (employee-facing) side of your shop. The team that builds the checkout for the end-user will, e.g., also make the associated help desk functionality for the customer hotline. They might also build the WebView-based version of the checkout that a native app can embed.

1.3.3 Productivity vs. overhead

Dividing your application into autonomous systems brings a lot of benefits but does not come for free.

SETUP

When starting fresh, you need to find good team boundaries, set up the systems, and implement an integration strategy. You need to establish common rules that all teams agree on, like using namespaces. It's also important to provide ways for people to exchange knowledge between teams.

ORGANIZATIONAL COMPLEXITY

Having smaller vertical systems reduces the technical complexity of the individual systems. But running a distributed system adds its complexity on top.

Compared to a monolithic application, there is a new class of problems you have to think about. Which team gets paged on the weekend when it's not possible to add an item to the basket? The browser is a shared runtime environment. A change from one team might have negative performance effects on the complete page. It's not always easy to find out who's responsible.

You will probably need an extra shared service for your frontend integration. Depending on your choice, it might not come with a lot of maintenance work. But it's one more piece to think about.

When done right, the boost in productivity and motivation should be more significant than the added organizational complexity.

1.3.4 Where micro frontends are not a great fit

But of course, micro frontends are not perfect for every project. As stated earlier, they are a solution for scaling development. If you only have a handful of developers and communication is no issue, the introduction of micro frontends won't bring much value.

It's crucial to know the domain you are working in well to make good vertical cuts. Ideally, it should be obvious which team is responsible for implementing a feature. Unclear or overlapping team missions will lead to uncertainty and long discussions.

I've spoken to people working in startups that have tried this model. Everything worked fine up

until the point the company needed to pivot its business model. It's, of course, possible to reorganize the teams and the associated software, but it creates a lot of friction and extra work. Other organizational approaches are more flexible.

If you need to create a lot of different apps and native user interfaces to run on every device, it might also become tricky for one team to handle. Netflix is famous for having an app for nearly every platform that exists: TVs, set-top-boxes, gaming consoles, phones, and tablets. They have dedicated user interface teams for these platforms. That said, the web gets more and more capable and popular as an application platform, which makes it possible to target different platforms from one codebase.

1.4 Who uses micro frontends?

The described concepts and ideas are not new. Amazon does not talk a lot about its internal development structure. However, individual stories say that their e-commerce site is built like this for a long time. Amazon also uses a UI integration technique that assembles the different parts of the page before it reaches the customer.

Micro frontends are indeed quite popular in the e-commerce sector. In 2012 the Otto Group.[6], a Germany based mail-order company, and one of the world's largest e-commerce players started to split up its monolith. The Swedish furniture company IKEA.[7] and Zalando.[8], one of Europes biggest fashion retailers moved to this model. Thalia.[9], a German bookstore chain, rebuilt its e-reader store into vertical slices to increase development speed.

But micro frontends are also used in other industries. Spotify.[10] organizes itself in autonomous end-to-end teams they call *Squads*. SAP published a framework.[11] to integrate different applications. Sports streaming service DAZN.[12] also rebuild their monolithic frontend to a micro frontends architecture.

1.5 Summary

- Micro frontends is an architectural approach and not a specific technique
- Micro frontends removes the team barrier between frontend and backend developers by introducing cross-functional teams.
- With the micro frontends approach, the application gets divided into multiple vertical slices that span from database to user-interface.
- Each vertical system is smaller and more focused. It’s thereby easier to understand, test, and refactor than a monolith.
- Frontend technology is changing fast. Having an easy way to evolve your application is a valuable asset.
- It’s a good pattern to set the team boundaries along the user journey and customer needs.
- Team should have a clear mission like: "Help the customer to find the product she is looking for."
- A team can own a complete page or deliver a piece of functionality via a fragment.
- A fragment is a mini-application that is self-contained, which means it brings everything it needs with it.
- The micro frontends model typically comes with more code for the browser. It’s vital to address web performance from the start.
- There are multiple frontend integration techniques. They work either on the client or the browser.
- Having a shared design system helps to achieve a consistent look and feel across all team frontends.
- To make good vertical cuts it’s important to know your companies domain well. Changing responsibilities afterward works but creates friction.

2 My First Micro Frontends Project

This chapter covers

- Building the micro frontends example application for this book
- Connecting pages from two teams via links
- Integrating a fragment into a page via iframes

Being able to work on a complex application with multiple teams in parallel is the essential feature of micro frontends. But the end-user of such an application does not care about the internal team structure. That's why we need a way to integrate the user interfaces these teams are creating. As you learned in chapter 1, there are different ways of assembling separate UIs in the browser.

In this chapter, you'll learn how to integrate UIs from different teams via links and iframes. From a technology standpoint, these techniques are neither new nor exciting. But they come with the benefit that they are easy to implement and understand. The key point from a micro frontends perspective is that they introduce minimal coupling between the teams. No shared infrastructure, libraries, or code conventions are required. The loose coupling gives the teams a maximum amount of freedom to focus on their mission.

In this chapter, we'll also build the foundation of our example project *The Tractor Store*. We'll expand on this project throughout the book. You will learn different integration techniques and their benefits and drawbacks. Spoiler alert: There is no "gold standard" or "best integration technique". It's all about making the right tradeoffs for your use-case. But this book will highlight the different aspects and properties you should look for when picking a technique.

We'll start with simple scenarios in this chapter and work our way through more sophisticated ones after that.

2.1 Introducing The Tractor Store

Tractor Models Inc., an imaginary startup, manufactures high-quality tin toy models of popular tractor brands. Currently, they are in the process of building an e-commerce website: *The Tractor Store*. It allows tractor fans from all over the world to purchase their favorite models.

To cater to their audience as best as possible, they want to experiment and test different features and business models. The concepts they plan to validate are offering deep customization options, auctions for premium material models, regionally limited special editions, and booking private in-person demos in flagship stores in all major cities.

To achieve maximum flexibility in development, the company decided to build the software from scratch and not go with an off-the-shelf solution.

The company wants to evaluate their ideas and features quickly. That is why it decided to go with the micro frontends architecture. Multiple teams can work in parallel, independently build new features, and validate ideas. They are starting with two teams.

We'll set up the software project for both *Team Decide* and *Team Inspire*. *Team Decide* will create a product detail page for all tractors that displays the name and image of the model. *Team Inspire* will provide matching recommendations. In the first iteration, each team displays its' content on a separate page from its own domain. They connect the pages via links. So we have a product page and a recommendation page for every model.

2.1.1 Getting started

Now both teams start setting up their applications, deployment process, and everything that required to get their pages ready.

FREEDOM OF CHOOSING TECHNOLOGY

Team Decide chooses to go with a MongoDB database for their product data and a Node.js application, which renders HTML on the server-side. *Team Inspire* plans to use data science techniques. They'll implement machine learning to deliver personalized product recommendations. That's why they picked a Python-based stack.

Being able to choose the technology that's best for the job is one of the micro frontends benefits. It takes into account that not all tasks are the same. Building a high traffic landing page has different requirements than developing an interactive tractor configurator.

SIDEBAR **Technology Diversity & Blueprints**

Just because *you can* does not mean *you must* use different technology stacks for each team. When teams use similar stacks, it gets easier to exchange best practices, get help, or move developers between teams.

It can also save upfront costs because you could implement the basic application setup, including folder structure, error reporting, form handling, or the build process once. Every team can copy this blueprint application and build on it. This way, teams can get productive a lot quicker, and the software stacks are more similar. In chapter 13.4. Technology diversity we'll go deeper into this topic.

INDEPENDENT DEPLOYS

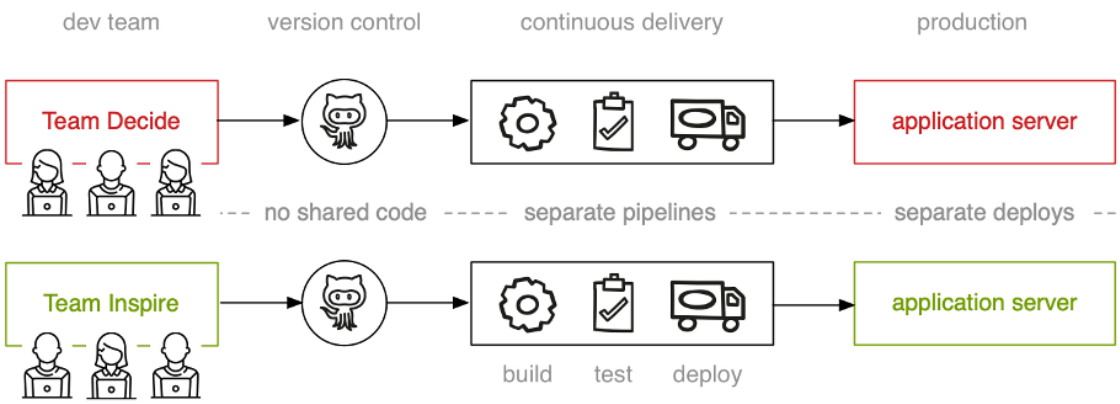


Figure 2.1 Teams work in their own source code repository, have separate integration pipelines and can deploy independently.

Both teams create their own source code repository and set up a continuous integration pipeline. This pipeline runs every time a developer pushes new code to the central version control system. It builds the software, runs all kinds of automated tests to ensure the software's correctness, and deploys the new version of the application to the team's production server. These pipelines run independently. A software change in *Team Decide* will never cause *Team Inspire's* pipeline to break.

2.1.2 Running this book's example code

For the integration techniques in the following chapters, the server-side technology stack is irrelevant. In our sample code, we'll focus on the HTML output the applications generate. We'll create a folder for every team which contains static HTML, JS, and CSS files, which we will serve through an ad-hoc HTTP server.

TIP	You can browse the source code of this book on GitHub.[13] or download a ZIP from the Manning website.[14]. If you don't want to run the code locally, you can go to the-tractor.store. There you can see and inspect all examples directly in your browser.

DIRECTORY STRUCTURE

The examples all follow the same structure. Inside of each example folder like `01_pages_links`, you find a folder for each team like `team-[name]`. Figure 2.2 shows the example.

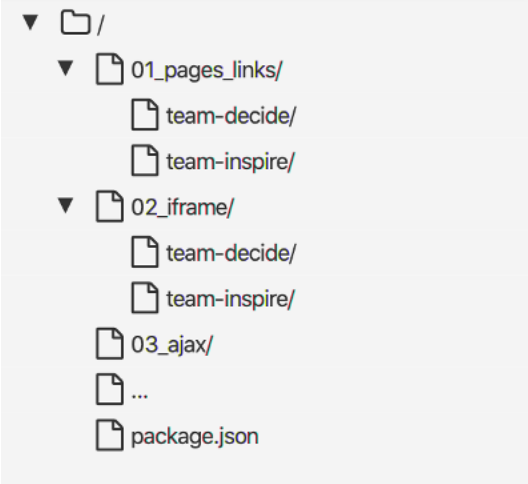


Figure 2.2 Directory structure of one code example.

A team folder represents a team's application. Code from one team folder never directly references code from another team's folder.

NODE.JS REQUIRED

Static assets like JS and CSS will go into the static folders later. You'll need to have Node.js installed to run the ad-hoc server. If you haven't, go to nodejs.org/ and follow the installation instructions. All examples run with Node.js v12. Higher versions should also work.

NOTE	We are not assuming a specific terminal or shell throughout this book. The commands work in Windows PowerShell, Command Prompt or Terminal on macOS and Linux.

INSTALL DEPENDENCIES

Navigate your terminal into the root directory for the sample code. There's a `package.json` that contains a start script for each example project. Install the required dependencies.

```
npm install
```

STARTING AN EXAMPLE

You can start each example from the root directory by running `npm run [name_of_example]`. Try this for our first example by typing this into your terminal:

```
npm run 01_pages_links
```

Each run-command performs three actions:

1. It starts a **static web-server for each team directory.** It uses ports 3000 to 3003 for this.
2. It opens the example page in your default browser.
3. It shows an aggregated network log for all applications in the terminal.

> **NOTE** Make sure ports 3000 to 3003 are not occupied by other services on your machine. If a port is blocked, the start script will not fail but start the application on another random port. Check the log if you're experiencing issues.

Running the command for our first example should have started two servers on port 3001 and 3002. Your browser should show the product page with a red tractor at localhost:3001/product/porsche.

Your terminal output should look like this:

```
$ npm run 01_pages_links

> code@1.0.0 01_pages_links [...]
> concurrently --names 'decide ,inspire' "mfserve --listen 3001 01_pages_links/team-decide"
     "mfserve --listen 3002 01_pages_links/team-inspire"
     "npm run open -- http://localhost:3001/product/porsche"

[decide ] INFO: Accepting connections at http://localhost:3001          ❶
[inspire] INFO: Accepting connections at http://localhost:3002          ❶
[2]
[2] > code@1.0.0 open [...]
[2] > sleepms 3000 && opener "http://localhost:3001/product/porsche"    ❷
[2]
[2] npm run open -- http://localhost:3001/product/porsche exited with code 0
[decide ] :3001/product/porsche                                         ❸
[decide ] :3001/static/page.css                                         ❸
[decide ] :3001/static/outlines.css                                     ❸
```

❶ Started *Team Decide's* server on port 3001 and *Team Inspire's* server on port 3002.

❷ Opening the example page in your default browser.

❸ Shows the three network calls *Team Decide's* application answered for the example page.

> **NOTE** The ad-hoc web-server uses the `@microfrontends/serve` package. It's a modified version of the great `zeit/serve` server. I've added some features like logging, custom headers, and support for delaying requests. We'll need these features in the following chapters.

You can stop the web-server by pressing `[CTRL]` + `[C]`.

With the setup and organizational stuff out of the way, we can start to focus on integration techniques.

2.2 Page transition via links

In the first iteration of their development, the teams choose to keep it as simple as possible. No fancy integration technique. Every team builds its feature as a standalone page. The team's applications serve these pages directly. Each team brings its own HTML and CSS.

2.2.1 Data ownership

They are starting with three tractor models. In table 2.1 you see the data necessary for delivering a product page: a unique identifier (SKU), name, and image path.

Table 2.1 Team Decide's product database

SKU	Name	Image
`porsche`	Porsche Diesel Master 419	mi-fr.org/img/porsche.svg
`fendt`	Fendt F20 Dieselroß	mi-fr.org/img/fendt.svg
`eicher`	Eicher Diesel 215/16	mi-fr.org/img/eicher.svg

Team Decide owns the base product data. They'll build tools that enable employees to add new products or update existing ones. *Team Decide* is also responsible for hosting the product images. They upload them to a CDN where other teams can directly reference them.

Team Inspire also needs some product data. They must know all existing SKUs and the associated image URL. That's why *Team Inspire's* backend regularly imports this data from *Team Decide's* data feed. *Team Inspire* keeps a local copy of the relevant fields in their database. In the future, they'll also consume analytics and purchasing history data to improve their recommendation quality. But for now, the product recommendations will be hard-coded. Table 2.2 shows *Team Inspire's* product relations.

Table 2.2 Team Inspire's recommendations

SKU	Recommended SKUs
`porsche`	`fendt`, `eicher`
`eicher`	`porsche`, `fendt`
`fendt`	`eicher`, `porsche`

Team Decide doesn't have to know anything about these relations. Neither do they need to know about the underlying algorithms and data sources.

2.2.2 Contract between the teams

In this integration, the URL is our **contract between the teams**. Teams that own a page publish their URL patterns. The others can use it to create a link. Here are the patterns for both teams:

- *Team Decide:* Product Page URL-pattern: `http://localhost:3001/product/<sku>` example: `http://localhost:3001/product/porsche`
- *Team Inspire:* Recommendation Page URL-pattern: `http://localhost:3002/recommendations/<sku>` example: `http://localhost:3002/recommendations/porsche`

Because we're running this locally, we use `localhost` instead of a real domain. We pick ports `3001` (Team Decide) and `3002` (Team Inspire) to differentiate the teams. In a live scenario, the teams could have picked any domain they like.

When both applications are ready, the result should look like figure 2.3. The product page shows the name and image of the tractor and links it to the corresponding recommendation page. The recommendation page shows a list of matching tractors. Each image links to the matching product page.

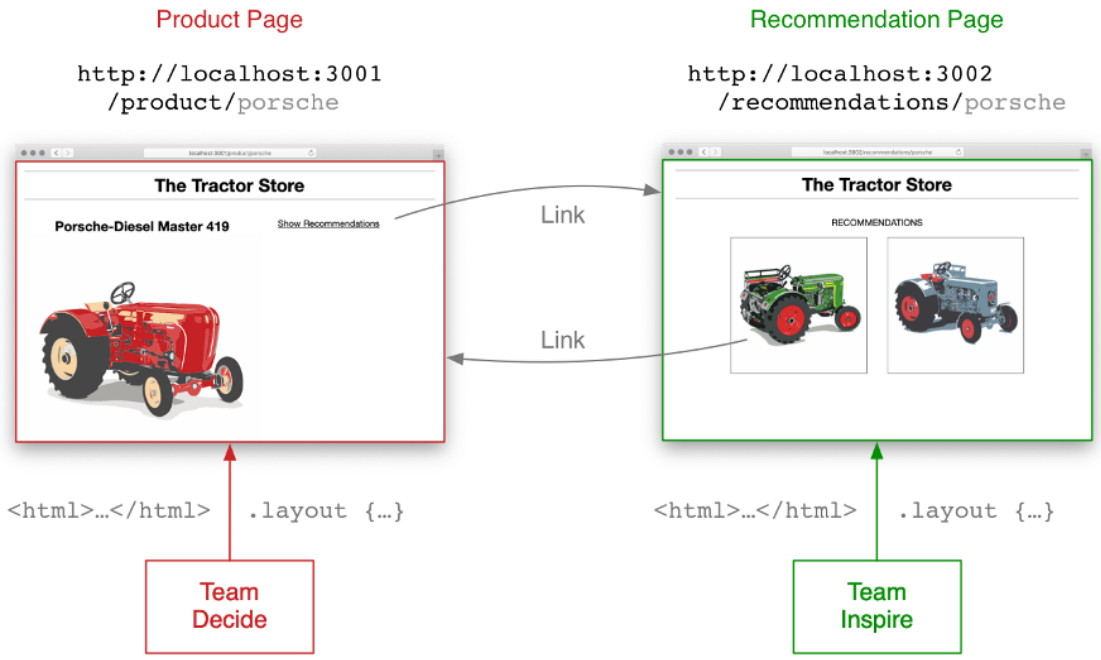


Figure 2.3 A product and recommendations page connected via links

Let's have a quick lock at the code that's involved in making this happen.

2.2.3 How to do it

You can find the code for this example in the `01_links_pages` folder. Figure 2.4 shows the directory listing.

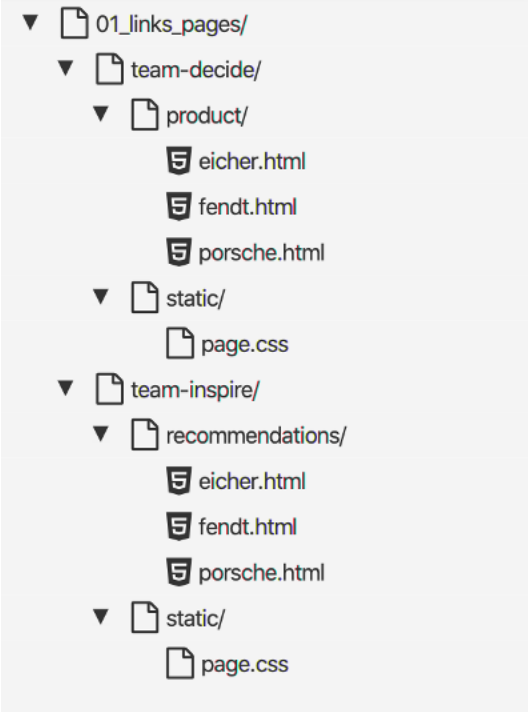


Figure 2.4 A product and recommendations page connected via links

The HTML files represent the server-generated output of the teams. Each team also brings its CSS file.

> **NOTE** The ad-hoc web-server defaults to a `.html` extension when looking up a file. Requests to `/product/porsche` will serve the `./product/eicher.html` file.

MARKUP

Have a quick look at the HTML of a product page. We'll build on this markup throughout the examples of this book.

Listing 2.1 team-decide/product/porsche.html

```
<html>
  <head>
    <title>Porsche-Diesel Master 419</title>
    <link href="/static/page.css" rel="stylesheet" />
  </head>
  <body class="layout">
    <h1 class="header">The Tractor Store</h1>
    <div class="product">
      <h2>Porsche-Diesel Master 419</h2>
      <img class="image" src="https://mi-fr.org/img/porsche.svg" />
    </div>
    <aside class="recos">
      <a href="http://localhost:3002/recommendations/porsche">   ❶
        Show Recommendations
      </a>
    </aside>
  </body>
</html>
```

❶ The link to Team Inspire's matching recommendation page.

The markup for the other product pages looks similar. The important thing here is the *Show Recommendations* link. **It's our first micro frontends integration technique.** *Team Decide* generates the link according to the URL-pattern provided by *Team Inspire*.

Let's switch to *Team Inspire*. The markup for a recommendation page looks like this:

Listing 2.2 team-inspire/recommendations/porsche.html

```
<html>
  <head>
    <title>Recommendations</title>
    <link href="/static/page.css" rel="stylesheet" />
  </head>
  <body class="layout">
    <h1 class="header">The Tractor Store</h1>
    <h2>Recommendations</h2>
    <div class="recommendations">
      <a href="http://localhost:3001/product/fendt">       ❶
        <img src="https://mi-fr.org/img/fendt.svg" />      ❶
      </a>                                                 ❶
      <a href="http://localhost:3001/product/eicher">      ❶
        <img src="https://mi-fr.org/img/eicher.svg" />     ❶
      </a>                                                 ❶
    </div>
  </body>
</html>
```

❶ Links to Team Decide's product pages.

Again, the markup for the other tractors pages is the same but shows different recommendations.

STYLES

You've noticed that both teams bring their CSS files. When you compare these files (`team-decide/static/page.css` vs. `team-inspire/static/page.css`), you'll find redundancy. Both teams include basic layout, reset, and font styles.

We could introduce a master CSS file that all teams include. Having centralized styling might sound like a good idea. However, relying on a central CSS file introduces a considerable amount of coupling. Since micro frontends is all about decoupling and maintaining team autonomy, we have to be careful - even with styling.

In chapter 12. User Interface & Design System we'll discuss the coupling aspect in greater detail and illustrate different solutions for shipping a coherent user interface across teams. So, for the examples in the following chapters, we'll have to live with this styling redundancy.

STARTING THE APPLICATIONS

Let's run the example and look at it in the browser. Execute the following command in the sample codes root folder.

```
npm run 01_pages_links
```

It opens localhost:3001/product/porsche in your browser, and you see the red Porsche Diesel Master tractor. The result should look like the screenshot in figure 2.5.

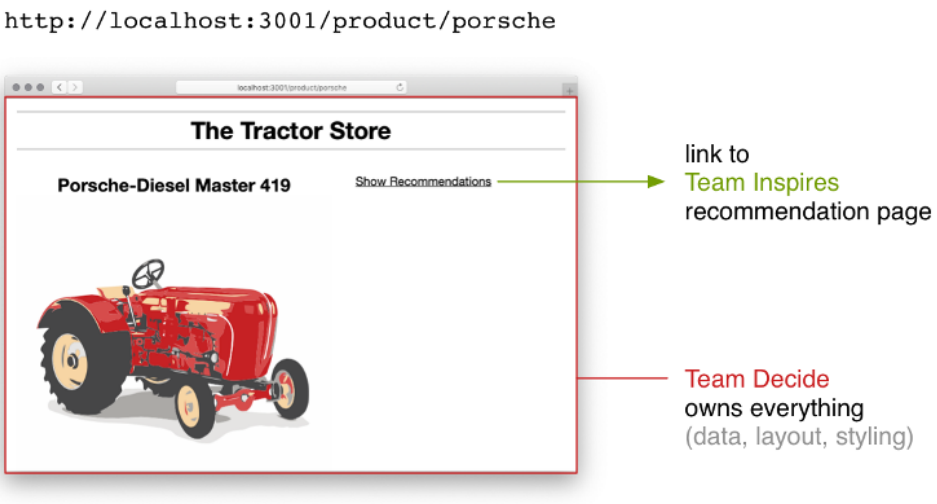


Figure 2.5 Team Decides product detail page. The team owns everything on this page.

You can click on the "Show Recommendations" link to see other matching tractors on *Team Inspire's* recommendation page. From there, you can jump back to a product page by clicking on another tractor. In the browser address bar, you see the browser jumping from `localhost:3001`

to `localhost:3002`.

Congratulations, we've created our first e-commerce project that adheres to the micro frontends principals. The following sections will build on this code so that we can focus more on the actual integration techniques and care less about the boilerplate.

2.2.4 Dealing with changing URLs

The integration works because both teams exchanged their URL-patterns beforehand. URLs are a popular and powerful concept which we will also see with other integration techniques. Sometimes URLs need to change because your application migrated to another server, a new scheme would be better for search engines, or you want language-specific URLs. You can manually notify all other teams. But when the number of teams and URLs grows, you want to automate this process.

HTTP includes primitives like redirects for this case, which are an excellent fit for a lot of use cases. A more robust mechanism that has proven valuable for the projects we've worked on is that every team provides a machine-readable directory of all their URL-patterns. A JSON file in a known location usually does the trick. This way, all applications can lookup the URL-patterns regularly and update their links if needed. Standards like URI-templates.[15], json-home.[16] or Swagger OpenAPI.[17] can help here.

2.2.5 The benefits

Though the outcome might not look impressive, the solution we just built has two properties that are important for running a micro frontends application. The coupling between the two applications is low, and the robustness is high.

LOOSE COUPLING

In this context, coupling describes how much one team needs to know about the other team's system to make the integration work. In this example, every team only needs to implement the URL-pattern of the other team to link to them. A team does not have to care about what programming language, frameworks, styling approach, deployment technique, or hosting solution the other team uses. As long as the sites are available at the previously defined URLs, everything works magically. We see the beauty of the open web in action here.

HIGH ROBUSTNESS

When the recommendation application goes down, the detail page still works. The solution is robust because the applications share nothing. They bring everything they need to deliver their content. An error in one system can not affect the other team's system.

2.2.6 The drawbacks

The fact that the teams share nothing does come with a cost. An integration via links-only is not always optimal from a user's point of view. She has to click a link to see the information that is owned by another team. In our case, she bounces between the product and the recommendation page. With this simple integration, we have no way of combining data from two different teams into one view.

This model also comes with a lot of technical redundancy and overhead. Common parts like the page header need to be created and maintained by each team.

2.2.7 When do links make sense?

When you are building a somewhat complicated site, an integration that relies on links-only is not sufficient in most cases. Often you need to embed information from another team. But you don't have to use links alone. They play well with other integration techniques.

2.3 Composition via iframe

The whole company staff is pleased about the progress both teams made in this short amount of time. But everyone agrees that we have to improve the user experience. Discovering new tractors via the "Show Recommendations" link works but is not apparent enough for the customer. First studies show that more than half of the testers did not notice the link at all. They left the site under the assumption that *The Tractor Store* only offers one product.

The plan is to integrate the recommendations into the product page itself. We'll replace the "Show Recommendations" link on the right side. The visual style of the recommendations can stay the same.

In a short technical meeting, both teams weighed possible composition solutions against each other. They quickly realized that composition via iframe would be the fastest way to get this done.

With iframes, it's possible to embed one page into another page while maintaining the same loose coupling and robustness properties the link integration provides. Iframes come with strong isolation. What happens in the iframe stays in the iframe. But they also have significant drawbacks which we'll also discuss in this chapter.

Only a few lines of code have to be changed by each team. Figure 2.6 illustrates how the recommendation look on the product page. It also shows the team responsibilities. The complete recommendation page gets included on the product page.

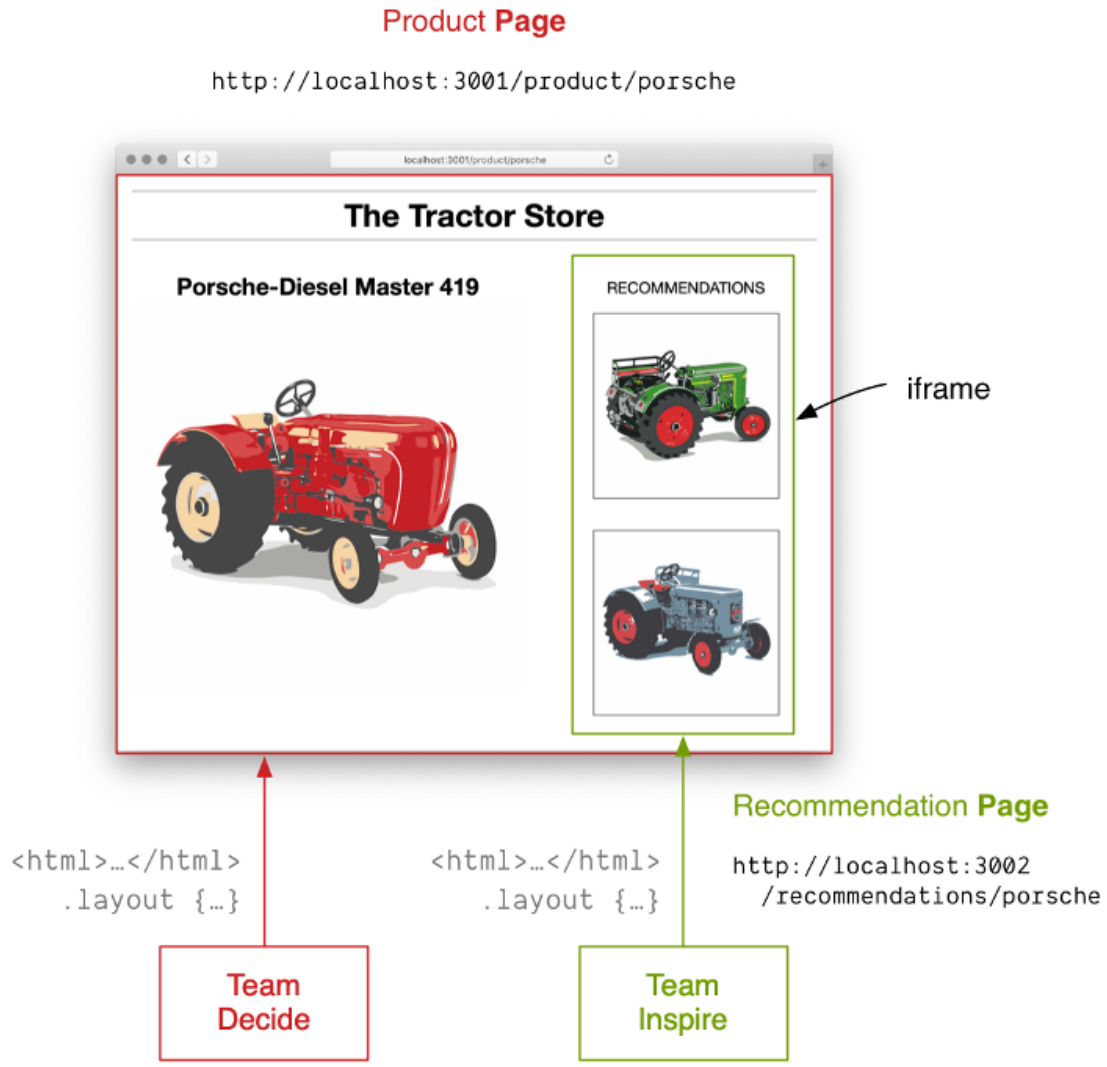


Figure 2.6 Integrating the recommendation page into the product page via iframe. These pages don't share anything. Both are standalone HTML documents with their own styling.

2.3.1 How to do it

Ok, off to work. Our first task is to replace the *Show Recommendations* link. *Team Decide* can do that in their HTML.

Listing 2.3 team-decide/product/porsche.html

```
...
<iframe src="http://localhost:3002/recommendations/porsche"></iframe>
...
```

After that, *Team Inspire's* removes the "The Tractor Store" header from the recommendation pages markup because we don't need it in the iframe.

You find the updated example code in the `02_iframe` folder. Run it via this command.

```
npm run 02_iframe
```

Your browser shows the recommendations inlined into the product page like you've seen before in figure 2.6.

There's one other code change *Team Decide* had to do to make the iframe composition work. Iframes have one major drawback when it comes to layout. The outer document needs to know the exact height of the iframe's content to avoid scrollbars or whitespace. *Team Decide* added this code to their CSS:

Listing 2.4 team-decide/static/page.css

```
...
.recos iframe {
  border: 0;       ❶
  width: 100%;     ❷
  height: 750px;   ❸
}
```

❶ remove the browser's default iframe border

❷ iframe should be as wide as its parent container

❸ fixed height to make enough space for the content

For static layouts, this might not be an issue, but if you're building a responsive site, it can become tricky. The height of the content might change depending on the size of the device.

Another issue is, that *Team Inspire* is now bound to the height *Team Decide* has defined. They can't experiment with, e.g., adding a third recommendation image without having to talk to the other team. JavaScript libraries.[18] exist to automatically update the iframe size when it's content changes.

The **contract between the teams** has become more complicated. Before, teams only needed to know the URL. Now they must **also know the height of its content**.

2.3.2 The benefits

In theory, the iframe is the optimal composition technique for micro frontends. Iframes **work in every browser**. They provide **strong technical isolation**. Scripts and styles can't leak in or out. They also bring a lot of **security features** to shield the team's frontends against each other.

2.3.3 The drawbacks

While iframes provide high isolation and are easy to implement, they also have a lot of negative properties, which has led to the iframe's lousy reputation in web development.

LAYOUT CONSTRAINTS

As already discussed, the absence of a reliable solution for automatic iframe height is one of the most significant drawbacks in day to day use.

PERFORMANCE OVERHEAD

Heavy use of iframes is terrible for performance. Adding an iframe to a page is a costly operation from a browser's perspective. Every iframe creates a new browsing context, which results in extra memory and CPU usage. If you are planning to include many iframes on a page, you should test the performance impact they introduce.

BAD FOR ACCESSIBILITY

Structuring the content of your page semantically is not only a hygienic factor. It enables assistive technologies like screen readers to analyze the content of the page and gives a visually impaired user the ability to interact with the content via voice. Iframes break the semantics of the page. We can style an iframe to blend in with the rest of the page seamlessly. But tools like screen readers have a hard time conceptualizing what's going on. They see multiple documents that all have their own title, information hierarchy, and navigation state. Be careful with iframes if you don't want to break your accessibility support.

BAD FOR SEARCH ENGINES

When it comes to search engine optimization (SEO), iframes also have a bad reputation. A crawler would index our product page as two distinct pages. The outer page and the included inner page. The search index does not represent the fact that one includes the other. Our page would not show up for the search term "tractor recommendations". The user sees both words in her browser window However, these words do not exist in the same document.

2.3.4 When do iframes make sense?

These are quite strong arguments against the use of iframes. So when does an iframe make sense at all? As always, it depends on your use-case.

Spotify, for example, has implemented a micro frontends architecture early on for their desktop application.[19]. Their integration technique relied on using iframes for the different parts of the application. Since the overall layout of their application is quite static, and search engine indexing is not an issue, this was an acceptable tradeoff for them.

You shouldn't use iframes if you are building a customer-facing site where loading performance, accessibility, and SEO matter. But for internal tools, they can be an excellent and straightforward option to get started with a micro frontends architecture.

2.4 What's next?

In this chapter, we've successfully built a micro frontends application. Two teams can develop and deploy their part of the application autonomously. Both applications are decoupled. When one application breaks, the other still works.

Have a look at the integration techniques in figure 2.7. You've already seen these three types of

integration in the big picture diagram in chapter 1.

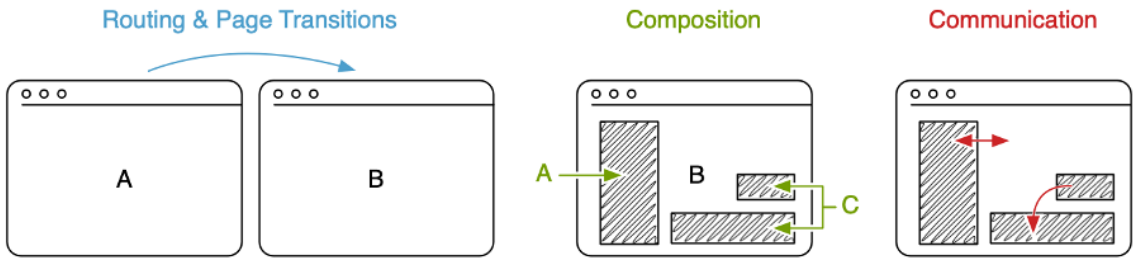


Figure 2.7 We can divide frontend integration techniques into three categories: routing, composition and communication.

We covered the first two groups. **Transitioning between different teams pages** using a link and using the **iframe as a composition technique** to include content from another team. We didn't need communication yet.

In the next chapters, we'll fill our toolbox with more server- and client-side integration techniques. We've arranged the chapters by complexity - starting with the simplest and working our way up to more sophisticated methods.

In chapter 9. Which Architecture Fits My Project? we'll zoom out a bit. We discuss different micro frontend high-level architectures like building *server rendered pages* or constructing a single page app composed out of other single-page apps (*Unified SPA*).

If you have a clear project in mind and you're limited in time there's a shortcut: Feel free to skip to chapter 9 to get an overview and decide which architecture fits best. You can then selectively jump back to the chapters that discuss its required techniques.

2.5 Summary

- Teams should be able to develop, test and deploy independently. That's why it's crucial to avoid coupling between their applications.
- Integration via links or iframes is simple. A team only needs to know the URL patterns of the other teams.
- Each team can build, test and deploy their pages with the technology they like.
- High isolation and robustness. When one system is slow or broken, the other systems are not affected.
- A page can be integrated into other pages via iframes.
- The page which integrates another page via iframe needs to know the size of its content. This knowledge introduces new coupling.
- Iframes provide strong isolation between the teams. No shared code conventions or name-spacing for CSS or JavaScript are required.
- Iframes are suboptimal for performance, accessibility, and search engine compatibility.

3 Composition with AJAX & Server-side Routing

This chapter covers

- Integrating fragments into a page via AJAX
- Applying project-wide namespaces to avoid style or script collisions.
- Utilizing the nginx web-server to serve all applications from one domain
- Implementing request routing to forward incoming requests to the right server

We covered a lot of ground in the previous chapter. The applications for two teams are ready to go. You learned how to integrate user interfaces via links and iframes. These are all valid integration methods, and they provide strong isolation. But they come with tradeoffs in the areas of usability, performance, layout flexibility, accessibility, and search engine compatibility. In this chapter, we'll look at fragment integration via AJAX to address these issues. We'll also configure a shared web-server to expose all applications through a single domain.

3.1 Composition via AJAX

Our customers love the new product page. Presenting all recommendations directly on that page has measurable positive effects. On average people spend more time on the site than before.

But Waldemar, responsible for marketing, noticed that the site does not rank very well in most search engines. He suspects that the suboptimal ranking has something to do with the use of iframes. He talks to the development teams to discuss options to improve the ranking.

The developers are aware that the iframe integration has issues, especially when it comes to semantic structure. Since good search engine ranking is essential for getting the word out and reaching new customers, they decide to address this issue in the upcoming iteration.

The plan is to ditch the document-in-document approach of the iframe and choose a deeper

integration using AJAX. With this model, *Team Inspire* will deliver the recommendations as a fragment - a snippet of HTML. This snippet is loaded by *Team Decide* and integrated into the product page's DOM. Figure 3.1 illustrates this. They'll also have to find a good way to ship the styling that's necessary for the fragment.

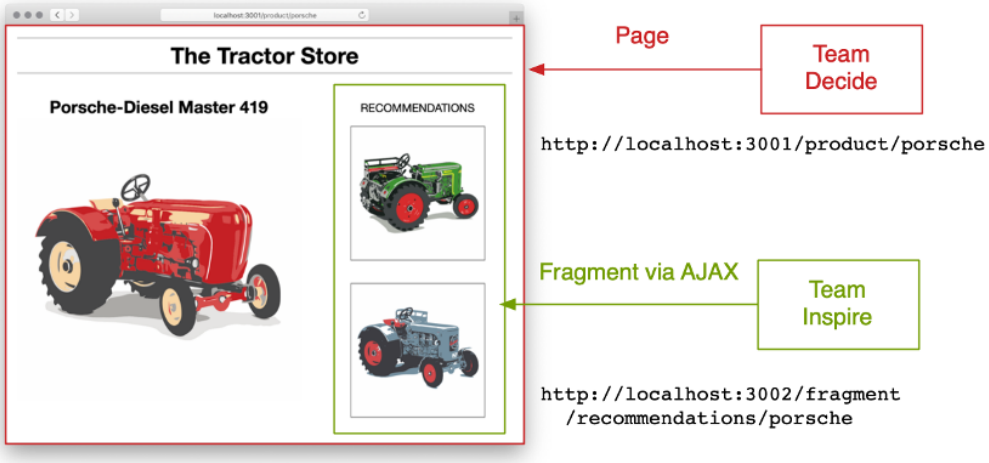


Figure 3.1 Integrating the recommendations into the product page's DOM via AJAX.

We have to complete two tasks to make the AJAX integration work:

1. *Team Inspire* exposes the recommendations as a fragment
2. *Team Decide* loads the fragment and inserts it into their DOM

Before getting to work, *Team Inspire* and *Team Decide* must talk about the URL for the fragment. They choose to create a new endpoint for the fragments markup and expose it under `http://localhost:3002/fragment/recommendations/<sku>`. The existing standalone recommendation page stays the same. Now both teams can go ahead and implement in parallel.

3.1.1 How to do it

Creating the fragment endpoint is straightforward for *Team Inspire*. All data and styles are already there from the iframe integration. Diagram 3.2 shows the updated folder structure:

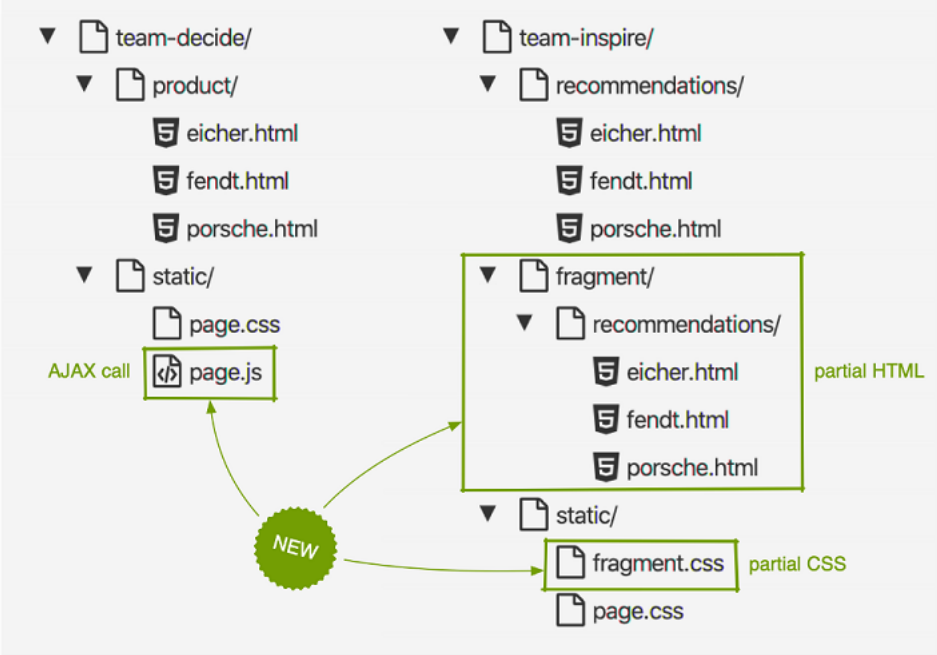


Figure 3.2 Folder structure of the AJAX example code 03_ajax.

Team Inspire adds an HTML file for each fragment, which is a stripped-down version of the recommendation page. They also introduce dedicated fragment styles (`fragment.css`). *Team Decide* introduces a `page.js`, which will trigger the AJAX call.

MARKUP

The fragment markup looks like this:

Listing 3.1 team-decide/fragment/recommendations/porsche.html

```
<link href="http://localhost:3002/static/fragment.css" rel="stylesheet" />    ❶
<h2>Recommendations</h2>
<div class="recommendations">
  ...
</div>
```

❶ Reference to the recommendation styles

Note that the fragment references it's own CSS file from the markup. The URL has to be absolute (`http://localhost:3002/…`) because *Team Decide* will insert this markup into its DOM, which they serve from port `3001`.

Shipping a link-Tag together with the actual content, is not always optimal. If a page included multiple recommendation strips, it would end up with multiple redundant link tags. In 10. Asset Loading we'll explore some more advanced techniques for referencing associated CSS and JavaScript resources.

AJAX REQUEST

The fragment is ready to use. Let's switch hats and slip into *Team Decides* shoes.

Loading a piece of HTML via AJAX and appending it to the DOM is not that complicated. Let's introduce our first client-side JavaScript.

Listing 3.2 team-decide/static/page.js

```
const element = document.querySelector(".decide_recos");    ❶
const url = element.getAttribute("data-fragment");   ❷

window
  .fetch(url)                                        ❸
  .then(res => res.text())
  .then(html => {
    element.innerHTML = html;                        ❹
  });
```

❶ Finding the element to insert the fragment in.

❷ Retrieving the fragment URL from an attribute.

❸ Fetching the fragment HTML via the native window.fetch API.

❹ Inserting the loaded markup to the product pages DOM.

Now we have to include this script into our page and add the `data-fragment` attribute to our `.decide_recos` element. The product page markup now looks like this:

Listing 3.3 team-decide/view.js

```
...
  <aside
    class="decide_recos"
    data-fragment="http://localhost:3002/fragment/recommendations/porsche"    ❶
  >
    <a href="http://localhost:3002/recommendations/porsche">    ❶
      Show Recommendations                                      ❶
    </a>                                                        ❶
  </aside>
  <script src="/static/page.js" async></script>                 ❶
</body>
...
```

❶ *Team Inspire's* recommendation fragment URL

❷ Link to the recommendation page. In case the AJAX call failed or hasn't finished yet, the customer can use this link as a fallback: Progressive Enhancement.

❸ Referencing the JavaScript file, which will make the AJAX request.

Let's try the example by running the following code:

```
npm run 03_ajax
```

The result looks the same as with the iframe. But now the product page and recommendation strip live in the same document.

3.1.2 Namespacing styles and scripts

Running inside the same document introduces some challenges. Now both teams have to build their application in a way that doesn't conflict with the others. When two teams style the same CSS class or try to write to the same global JavaScript variable weird side-effects can happen that are hard to debug.

ISOLATING STYLES

Let's look at CSS first. Sadly browsers don't offer much help here. The deprecated *Scoped CSS* specification would have been an excellent fit for our use-case. It allowed you to mark a `style`- or `link`-tag with the attribute `scoped`. The effect was that these styles are only active in the DOM-subtree they're defined in.. Styles from higher up in the tree will still propagate down, but styles from within a `scoped` block would never leak out. Sadly this specification did not last very long, and browsers which already supported it pulled their implementation.[20] Some frameworks like Vue.js still use the `scoped` syntax to achieve this isolation. But they use automatic selector prefixing under the hood to make this work in the browser.

> **NOTE** In modern browsers.[21] it's possible to get strong style scoping today via JavaScript and the ShadowDOM API which is part of the Web Components specification. We'll talk about this in chapter 5.2. Style isolation using Shadow DOM.

Since CSS rules are global by nature, the most practical solution is to **namespace all CSS selectors**. Many CSS methodologies like BEM.[22] use strict naming rules to avoid unwanted style leaking between components. But two different teams might come up with the same component name independently like the headline component in our example. That's why it's a good idea to introduce an extra team level prefix. Table 3.1 shows how this namespacing could look like.

Table 3.1 Namespacing all CSS selectors with a team prefix

Team Name	Team Prefix	Example Selectors
Decide	`decide`	`.decide_headline .decide_recos`
Inspire	`inspire`	`.inspire_headline .inspire_recommendation__item`
Checkout	`checkout`	`.checkout_minicart .checkout_minicart--empty`

> **NOTE** To keep the CSS and HTML size small, we like to use two-letter prefixes like `de`, `in` and `ch`. But for easier understanding, I opted for using longer and more descriptive prefixes in this book.

When every team follows these naming conventions and only uses class-name based selectors, the issue of overwriting styles should be solved.

Prefixing does not have to be done manually. Some tools can help here. CSS-Modules, PostCSS, or SASS are a good start. You can configure most CSS-in-JS solutions to add a prefix to each class-name.

It does not matter which tool a team chooses as long as all selectors are prefixed.

ISOLATING JAVASCRIPT

The fragment, in our example, does not come with any client-side JavaScript. But you also need inter-team conventions to avoid collisions in the browser. Luckily JavaScript makes it easy to write your code in a non-global way.

A popular way is to wrap your script in an IIFE (immediately invoked function expression).[23]. This way, the declared variables and functions of your application are not added to the global `window` object. Instead, we limit the scope to the anonymous function. Most build tools already do this automatically. For the `static/page.js` of *Team Decide* it would look like this:

Listing 3.4 team-decide/static/page.js

```
(function () {                ❶
  const element = ...;        ❷
  ...
})();                         ❸
```

❶ Immediately invoked function expression

❷ Variable is not added to the global scope

But sometimes you need a global variable. A typical example is that you want to ship structured data in the form of a JavaScript object alongside your server-generated markup. This object must be accessible by the client-side JavaScript. A good alternative is to write your data to your markup in a declarative way.

Instead of writing this:

```
<script>
const MY_STATE = {name: "Porsche"};
</script>
```

You could express it declaratively and avoid creating a global variable.

```
<script data-inspire-state type="application/json">
{"name":"Porsche"}
</script>
```

Accessing the data can be done by looking up the script-tag in your part of the DOM tree and

parsing it.

```
(function () {
  const stateContainer = fragment.querySelector("[data-state]");
  const MY_STATE = JSON.parse(stateContainer.innerHTML);
})();
```

But there are a few places where it's not possible to create real scopes, and you have to fall back to namespaces and conventions. Cookies, storage, events, or unavoidable global variables should be namespaced. You can use the same prefixing rules we've introduced for CSS class-names for this. Table 3.2 shows a few examples.

Table 3.2 Some JavaScript functionalities also need namespacing

Function	Example
Cookies	`document.cookie = "decide_optout=true";`
Local Storage	`localStorage["decide:last_seen"] = "a,b";`
Session Storage	`sessionStorage["inspire:last_seen"] = "c,d";`
Custom Events	`new CustomEvent("checkout:item_added"); window.addEventListener("checkout:item_added", …);`
Unavoidable Globals	`window.checkout.myGlobal = "needed this!"`
Meta-Tags	`<meta name="inspire:feature_a" content="off" />`

Namespacing does not only help with avoiding conflicts. Another valuable factor in day-to-day work is that they also **indicate ownership**. When an enormous cookie value leads to an error, you just have to look at the cookie-name to know which team can fix that.

The described methods for avoiding code interference are not only helpful for the AJAX integration. They also apply for nearly all other integration techniques. **I highly recommend setting up global namespacing rules like this when you're setting up a micro frontends project.** It will save everyone a lot of time and headaches.

3.1.3 Declarative loading with h-include

Let's look at a way to make composition via AJAX even easier. In our example, *Team Decide* loads the fragment's content imperatively by looking up a DOM element, running `fetch()`, and inserting the resulting HTML into the DOM.

The JavaScript library *h-include* provides a declarative approach for fragment loading.[24]. Including a fragment feels like including an iframe in the markup. You don't have to care about finding the DOM element and making the actual HTTP request. The library introduces a new HTML-element called `h-include`, which handles everything for you. The code for the recommendations would look like this:

Listing 3.5 team-decide/product/porsche.html

```
...
<aside class="decide_recos">
  <h-include
    src="http://localhost:3002/fragment/recommendations/porsche">   ❶
  </h-include>
</aside>
...
```

❶ `h-include` fetches the HTML from the `src` and inserts it into the element itself.

The library also comes with extra features like defining timeouts, reducing reflows by bundling the insertion of multiple fragments together, and lazy loading.

3.1.4 The benefits

AJAX integration is a technique that is easy to implement and understand. Compared to the iframe approach, it has a lot of advantages.

NATURAL DOCUMENT FLOW

In contrast to the iframe, we integrate all content into one DOM. Fragments are now part of the pages document flow. Being part of this flow means that a fragment takes precisely the space it needs. The team that includes the fragment does not have to know the height of the fragment in advance. When *Team Inspire* would display one or three recommendation images, the product page adapts in height automatically.

SEARCH ENGINES AND ACCESSIBILITY

Even though integration happens in the browser and the fragment is not present in the page's initial markup yet this model works well for search engines. Their bots execute JavaScript and index the assembled page.[25] Assistive technologies like screen readers also support this. It's essential, though, that the combined markup semantically makes sense as a whole. So make sure that your content hierarchy is marked up correctly.

PROGRESSIVE ENHANCEMENT

An AJAX-based solution typically plays well with the principals of progressive enhancement.[26]. Delivering server-rendered content as a fragment or as a standalone page doesn't introduce a lot of extra code.

You can provide a reliable fallback in case JavaScript failed or hasn't executed yet. On our product page, users with broken JavaScript will see the "Show Recommendations" link, which will bring them to the standalone recommendations place. Architecting for failure is a valuable technique that will increase the robustness of your application. I recommend checking out Jeremy Keith's publications.[27] for more details on progressive enhancement.

FLEXIBLE ERROR HANDLING

You also get a lot more options for dealing with errors. When the `fetch()` call fails or takes to long, you can decide what you want to do. Show the progressive enhancement fallback, remove the fragment from the layout altogether, or display a static alternative content you've prepared for this case.

3.1.5 The drawbacks

The AJAX model also has some drawbacks. The most obvious one is already present in its name: asynchronous.

ASYNCHRONOUS LOADING

You might have noticed that the site jumps or wiggles a little bit when it's loading. The asynchronous loading via JavaScript causes this delay. We could implement the fragment loading so that it would block the complete page rendering and only show the page when the fragments are successfully loaded. But this would make the overall experience worse.

Loading content asynchronously always comes with the tradeoff that the content pops in with a delay. For fragments that are further down the page and outside the viewport, this is not an issue. But for content inside of the viewport, this flickering is not nice. In the next chapter, you'll learn how to solve this with server-side composition.

MISSING ISOLATION

The AJAX model does not come with any isolation. To avoid conflicts, teams have to agree on inter-team conventions for namespacing. Conventions are fine when everyone plays by the book. But you have no technical guarantees. When something slips through, it can affect all teams.

SERVER REQUEST REQUIRED

Updating or refreshing an AJAX fragment is as easy as loading it initially. But when you implement a solution that relies purely on AJAX, this means that every user interaction triggers a new call to the server to generate the updated markup. A server-roundtrip is acceptable for many applications, but sometimes you need to respond to user input quicker. Especially when network conditions are not optimal, the server roundtrip can get quite noticeable.

NO LIFECYCLE FOR SCRIPTS

Typically a fragment also needs client-side JavaScript. If you want to make something like a tooltip work, an event handler needs to be attached to the markup that triggers it. When the outer page updates a fragment by replacing it with new markup fetched from the server, this event handler needs to be removed first and re-added to the new markup.

The team which owns the fragment must know when their code should run. There are multiple ways to implement this. MutationObserver.[28], annotation via `data-*` attributes, custom

elements, or custom events can help here. But you have to implement these mechanisms manually. In chapter 5. Client-side Composition we'll explore how Web Components can help here.

3.1.6 When does an AJAX integration make sense?

Integration via AJAX is straight forward. It's **robust and easy** to implement. It also introduces with **little performance overhead**, especially compared to the iframe solution, where every fragment creates a new browsing context.

If you are **generating your markup on the server-side**, this solution makes sense. It also plays well together with the server-side-includes concept we'll learn in the next chapter.

For fragments that contain a lot of interactivity and have local state, it might become tricky. Loading and reloading the markup from the server on every interaction might feel sluggish due to network latency. The use of Web Components and client-side rendering we'll discuss later in the book can be an alternative.

3.1.7 Summary

Let's revisit the three integration techniques we've touched so far. Figure 3.3 shows how the links, iframe, and AJAX approach compare to each other from a developer's and user's perspective.

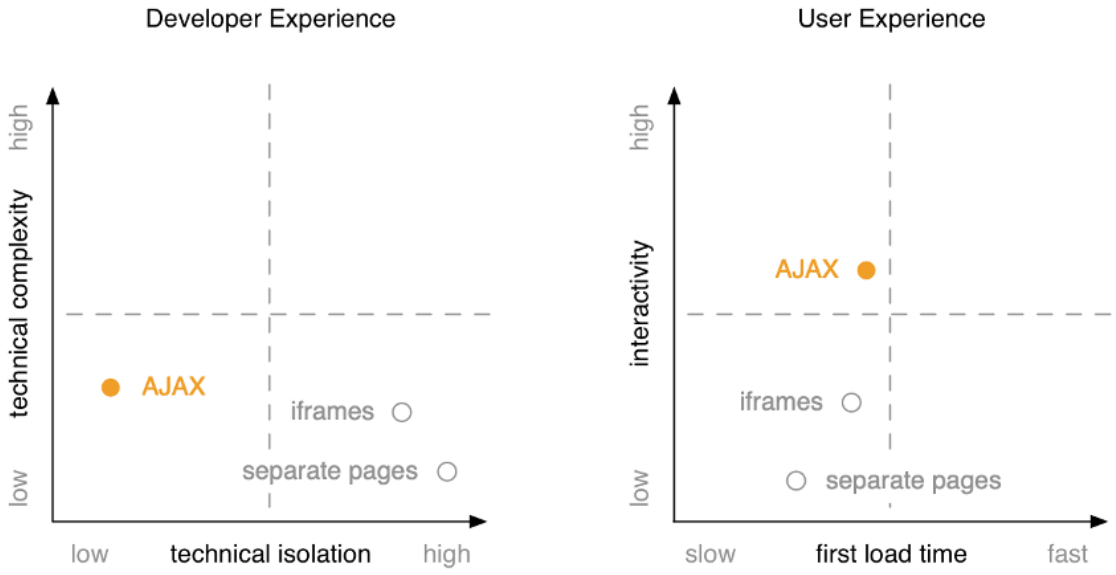


Figure 3.3 Comparison of different integration techniques. Compared to the iframes or links approach, its possible to build more performant and usable solutions with AJAX. But you lose technical isolation and need to rely on inter-team conventions like using CSS-prefixes.

I decided to compare them along four properties:

- **Technical complexity** describes how easy or complicated it is to set up and work in a model like this.
- **Technical isolation** indicates how much native isolation you get out of the box.
- **Interactivity** says how well this method is suited for building applications that feel snappy and respond to user input quickly.
- **First load time** describes the performance characteristics. How fast does the user get to the content she wants to see.

Note that this comparison should only give you an impression of how these techniques relate to each other in the defined categories. It's by no means representative, and you can always find counterexamples.

Next, we'll look at how to integrate our sample applications further. The goal is to make the applications of all teams available under one single domain.

3.2 Server-side routing via Nginx

The switch from iframe to AJAX had measurable positive effects. Search engine ranking improved, and we received emails from visually impaired users who wrote in to say that our site is much more screen reader-friendly now. But we also got some negative feedback. Some customers complained that the URLs for the shop are quite long and hard to remember. *Team Decide* picked Heroku as a hosting platform and published their site under `team-decide-tractors.herokuapp.com/`. *Team Inspire* chose Google Firebase for hosting. They've released their application at `tractor-inspirations.firebaseapp.com /`. This distributed setup worked flawlessly, but switching domains on every click is not optimal.

Ferdinand, the CEO of Tractor Models Inc., took this request seriously. He decided that all of the company's web properties should be accessible from one domain. After lengthy negotiations, he was able to acquire the domain `the-tractor.store`.

The next task for the teams is to make their applications accessible through the-tractor.store. Before going to work, they need to make a plan. A shared web-server is needed. It will be the central point where all requests to `https://the-tractor.store` will arrive initially. Figure 3.4 illustrates this concept.

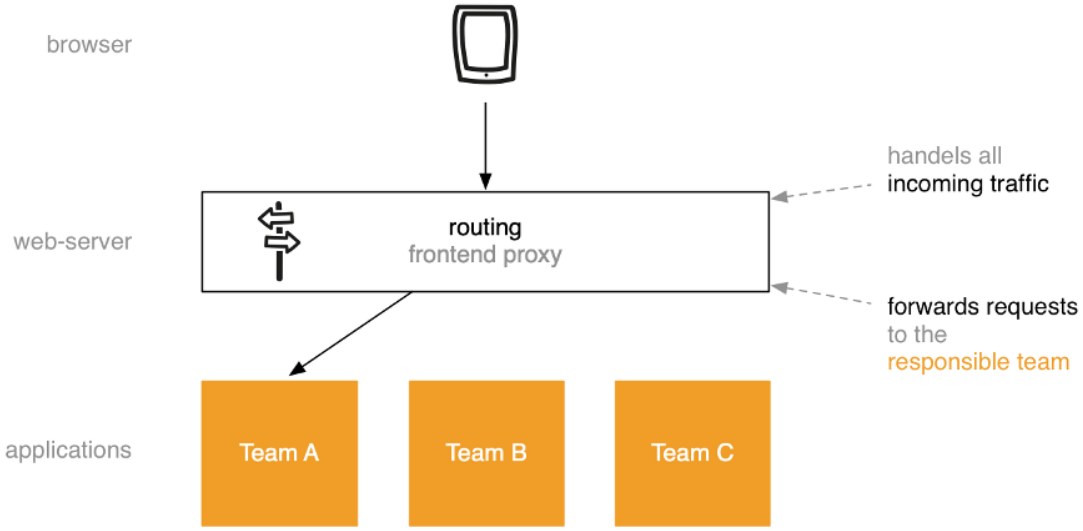


Figure 3.4 The shared web-server is inserted between the browser and the team applications. It acts as a proxy and forwards the requests to the responsible teams.

The server routes all requests to the responsible application. It does not contain any business logic besides this. This routing web-server is often called **frontend proxy**. Each team should receive its own path prefix for every team. The frontend proxy should route all requests starting with `/decide/` to *Team Decide's* server. They also require additional routing rules. The frontend proxy passes all requests starting with `/product/` go to *Team Decide*, the ones with `/recommendations/` go to *Team Inspire*.

In our development environment, we again use different port numbers instead of configuring actual domain names. The frontend proxy we will setup listens on port `3000`. Table 3.3 shows the routing rules our frontend proxy should implement.

Table 3.3 Frontend proxy routes incoming requests to the teams applications

Rule No	Path prefix	Team	Application
per team prefixes *(default)*			
#1	`/decide/`	Decide	`localhost:3001`
#2	`/inspire/`	Inspire	`localhost:3002`
per page prefixes *(additional)*			
#3	`/product/`	Decide	`localhost:3001`
#4	`/recommendations/`	Inspire	`localhost:3002`

Figure << web-server>> illustrates how an incoming network request is processed. Let's follow the numbered steps:

1. The customer opens the URL `/product/porsche`. The request reaches the frontend proxy
2. Frontend proxy matches the path `/product/porsche` against his routing table. Rule #3

2. `/product/` is a match.
3. Frontend proxy passes the request to *Team Decides* application.
4. The application generates a response and gives it back to the frontend proxy.
5. The frontend proxy passes the answer to the client.

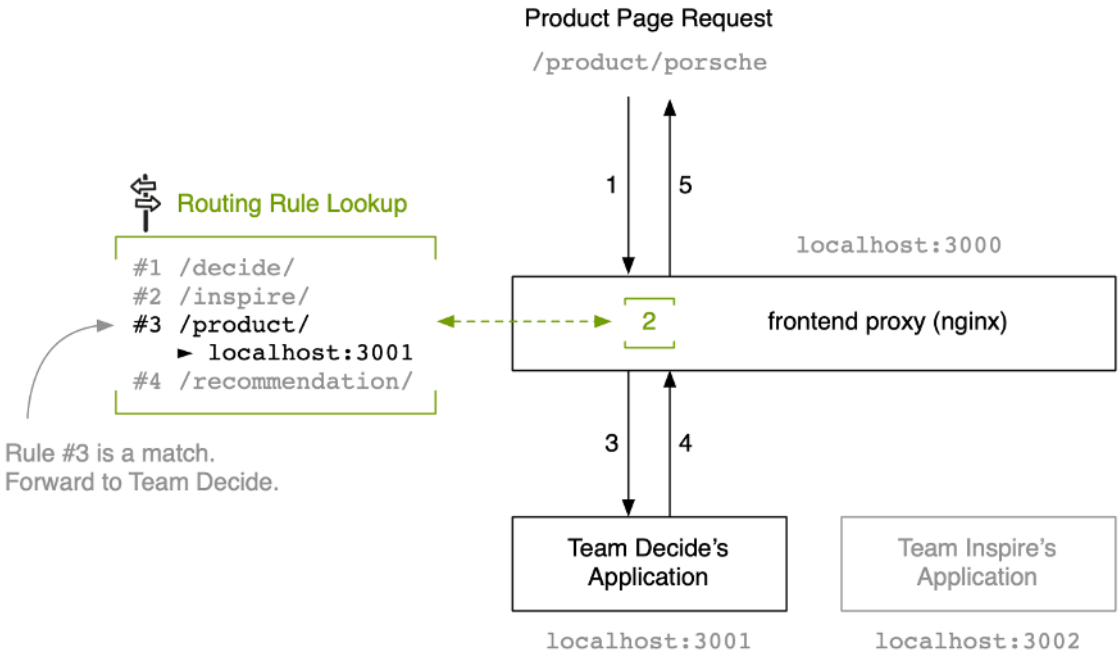


Figure 3.5 Flow of a request. The frontend proxy decides which application should handle an incoming request. It decides based on the URL-path and the configured routing rules.

Let's have a look at how to build a frontend proxy like this.

3.2.1 How to do it

The teams picked *Nginx* for this task. It's a popular, easy-to-use and pretty fast web-server. Don't worry if you haven't worked with Nginx before. We'll explain the fundamental concepts necessary to make our routing work.

If you want to run the example code locally, you need Nginx on your machine. For Windows users, it should work out of the box because I've included the Nginx binaries in the sample code directory. If you're running macOS.[29] or Linux.[30] and haven't installed Nginx yet you can get it via your package manager of choice. Start all three services by running this:

```
npm run 04_routing
```

The familiar Porsche Diesel Master should appear in your browser. Check your terminal to find a logging output that looks like this.

```
[decide ] :3001/product/porsche
[nginx  ] :3000/product/porsche 200
```

```
[decide ] :3001/decide/static/page.css
[decide ] :3001/decide/static/page.js
[nginx  ] :3000/decide/static/page.css 200
[nginx  ] :3000/decide/static/page.js 200
[inspire] :3002/inspire/fragment/recommendations/porsche
[nginx  ] :3000/inspire/fragment/recommendations/porsche 200
[inspire] :3002/inspire/static/fragment.css
[nginx  ] :3000/inspire/static/fragment.css 200
```

In this log message, we see two entries for each request. One from the team (`[decide]` or `[inspire]`) and one from the frontend proxy `[nginx]`. You see that all requests pass through the Nginx. The services create the log entry when they've produced a response. That explains why we always see the team application first and then the message from the Nginx.

NOTE On Windows, the `nginx` log messages don't appear because `nginx.exe` doesn't offer an easy way to log to `stdout`. If you're running Windows, you have to believe it's working as described (or reconfigure the `access_log` in the `nginx.conf` to write them to a local file of your choice).

Let's look into the frontend proxy configuration. You'll need to understand two Nginx concepts for this:

- forwarding a request to another server (`proxy_pass`/`upstream`)
- differentiating incoming requests (`location`)

Nginx's *upstream* concept allows you to create a list of servers that Nginx can forward requests to. The upstream configuration for *Team Decide* looks like this:

```
upstream team_decide {
  server localhost:3001;
}
```

You can differentiate incoming requests using *location*-blocks. A location-block has a matching rule which gets compared against every incoming request. Here's a location block that matches all requests starting with `/product/`:

```
location /product/ {
  proxy_pass  http://team_decide;
}
```

See the `proxy_pass` directive in the location block above? It advises Nginx to forward all matched requests to the `team_decide` upstream. Now you know everything we need to understand the `nginx.config` for our example.

Listing 3.6 webserver/nginx.conf

```
upstream team_decide {                    ❶
  server localhost:3001;                  ❶
}                                         ❶
upstream team_inspire {
  server localhost:3002;
}
http {
  ...
  server {
    listen 3000;
    ...
    location /product/ {                  ❷
      proxy_pass  http://team_decide;     ❷
    }                                     ❷
    location /decide/ {
      proxy_pass  http://team_decide;
    }
    location /recommendations {
      proxy_pass  http://team_inspire;
    }
    location /inspire/ {
      proxy_pass  http://team_inspire;
    }
}
```

❶ Registers *Team Decide's* application as an upstream called *team_decide*

❷ Handles all request starting with `/product/` and forwards them to the *team_decide* upstream.

NOTE In our example, we use a local setup. The upstream points to `localhost:3001`. But you can put in every address you want here. *Team Decides* upstream might be `team-decide-tractors.herokuapp.com`. Keep in mind that the web-server introduces an extra network hop. To reduce latency, you might want your web- and application servers to be located in the same data center.

3.2.2 Namespacing resources

Now that both applications run under the same domain, their URL structure mustn't overlap. For our example, the routes for their pages (`/product/` and `/recommendations`) stay the same. All other assets and resources are *moved* into a `decide/` or `inspire/` folder.

We need to adjust the internal references to the CSS and JS files. But also the URL-patterns both teams agreed upon (*the contract between the teams*) need to be updated. With the central frontend proxy in place, a team does not have to know the domain of the other team's application anymore. It's sufficient to use the path of the resource. Now Nginx's upstream configuration encapsulates the domain information. Since all requests should go through the frontend proxy, the domain we can remove the domain from the pattern.

- **product page** old: `http://localhost:3001/product/<sku>` new: `/product/<sku>`
- **recommendation page** old: `http://localhost:3002/recommendations/<sku>` new: `/recommendations/<sku>`
- **recommendation fragment** old: `http://localhost:3002/fragment/recommendations/<sku>` new: `/inspire/fragment/recommendations/<sku>`

> **NOTE** Notice that the path of the recommendation fragment URL received a team prefix (`/inspire`).

Introducing URL-namespaces is a crucial step when working with multiple teams on the same site. It makes the route configuration in the web-server easy to understand. Everything that starts with `/<teamname>/` goes to upstream `<teamname>`. *Team prefixes help with debugging because they make attribution easier.* Looking at the path of a CSS file that's causing an issue reveals which team owns it.

3.2.3 Route configuration methods

When your project grows, the number of entries in the routing configuration also grows. It can get complicated quickly. There are different ways to deal with this complexity. We can identify two different kinds of routes in our example application:

1. Page-specific routes (like `/product/`)
2. Team specific routes (like `/decide/`)

STRATEGY 1: TEAM ROUTES ONLY

The easiest way to simplify your routes is to apply a team prefix to **every** URL. This way, your central routes configuration only changes when we introduce a new team to the project. The configuration looks like this.

```
/decide/   -> Team Decide
/inspire/  -> Team Inspire
/checkout/ -> Team Checkout
```

The prefixing is not an issue for internal URLs. URLs the customer does not see like APIs, assets, or fragments. But for URLs that show up in the browser address bar, search results, or printed marketing material, this may be an issue. You are exposing your internal team structure through the URLs. You also introduce words (like decide, inspire) which a search engine bot would read and add to their index.

Choosing a shorter one- or two-letter-prefixes can moderate this effect. This way your URLs might look like this:

```
/d/product/porsche  -> Team Decide
/i/recommendations  -> Team Inspire
```

```
/c/payment          -> Team Checkout
```

STRATEGY 2: DYNAMIC ROUTE CONFIGURATION

If prefixing everything is not an option, it's unavoidable to put the information on which team owns which page into your frontend proxy's routing table.

```
/product/*        -> Team Decide
/wishlist         -> Team Decide
/recommendations  -> Team Inspire
/summer-trends    -> Team Inspire
/cart             -> Team Checkout
/payment          -> Team Checkout
/confirmation     -> Team Checkout
```

When you start small, this is usually not a big issue, but the list can quickly grow. And when your routes are not only prefix-based but include regular expressions, it can get hard to maintain.

Since routing is a central piece in a micro frontend architecture, it's wise to invest in quality assurance and testing. You don't want a new route entry to bring down other pieces of software.

There are multiple technical solutions for handling your routing. Nginx is only one option. Zalando open-sourced its routing solution called Skipper.[31]. They've built it to handle more than 800.000 route definitions.

3.2.4 Infrastructure Ownership

The key factors when setting up a micro frontends style architecture are team autonomy and end-to-end responsibility. Consider these aspects of every decision you make. Teams should have all the power and tools they need to accomplish their job as best as possible. In a micro frontends architecture, we accept redundancy in favor of decoupling.

Introducing a central web-server does not fit this model. To serve everything from the same domain, it's technically necessary to have one single service that acts as a common endpoint, but it also introduces a single point of failure. When the web-server is down, the customer sees nothing, even if the applications behind it are still running. Therefore, you should keep central components like this to a minimum. Only introduce them when there is no reasonable alternative.

Clear ownership is vital to ensure that these central components run stable and get the attention they need. In classical software projects, a dedicated platform team would run it. The goal of this team is to provide and maintain these shared services. But in practice, these horizontal teams create a lot of friction.

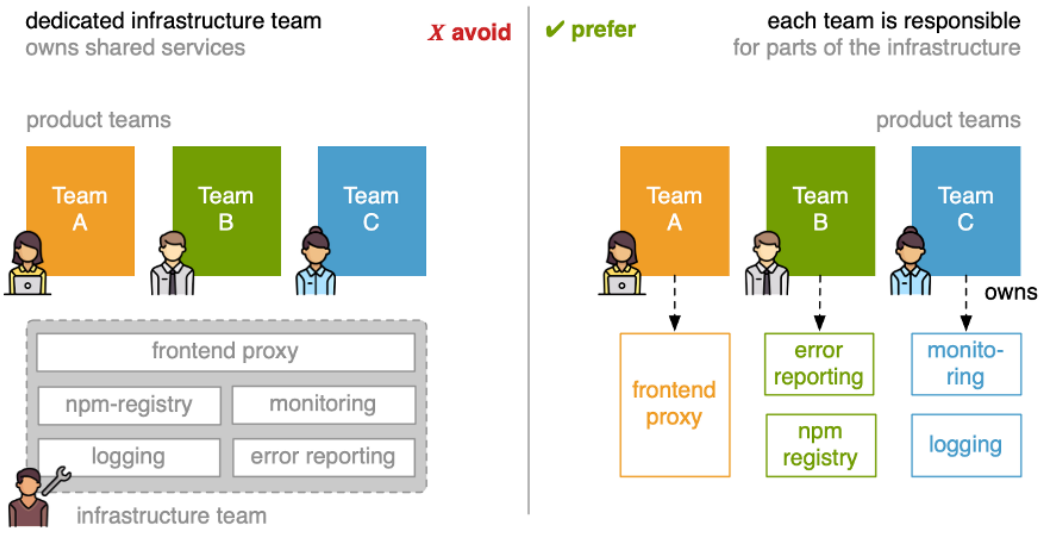


Figure 3.6 Avoid introducing pure infrastructure teams. Distributing responsibility for shared services to the product teams can be a good alternative model.

Distributing infrastructure responsibility across the product teams can help to keep the focus on customer value. In our example, *Team Decide* could take responsibility for running and maintaining the Nginx. They, and the other teams, have a natural interest that this service is well maintained and runs stable. However, they have no motivation to make it fancier as it needs to be. In chapters 12. User Interface & Design System and 13. Teams & Boundaries, we'll go deeper into the centralized vs. decentralized discussion.

3.2.5 When does it make sense?

Delivering the contents of multiple teams through a single domain is pretty standard. Customers expect that the domain in their browser address-bar does not change on every click.

It also has technical benefits:

- Avoids browser security issues (CORS)
- Enables sharing data like login-state through cookies
- Better performance (only one DNS lookup, SSL handshake, …)

If you are building a customer-facing site that should be indexed by search engines, you most definitely want to implement a shared web-server. For an internal application, it might also be ok to skip the extra infrastructure and just go with a subdomain-per-team approach.

Now we've discussed routing on the server-side. Nginx is only one way to do it, other tools like Traefik.[32] or Varnish.[33] offer similar functionality. In 7. Client-side Routing & The Application

Shell you'll learn how to move these routing rules to the browser. Client-side routing enables us to build a Unified Single Page App. But before we get there, we'll stay on the server and look at more sophisticated composition techniques.

3.3 Summary

- You can integrate the contents of multiple pages into a single document by loading them via AJAX.
- Compared to the iframe approach, a deeper AJAX integration is better for accessibility, search engine compatibility, and performance.
- Since the AJAX integration puts fragments into the same document, its possible to have style collisions.
- You can avoid CSS collisions by introducing team namespaces for CSS-classes.
- You can route the content of multiple applications through one frontend proxy, which serves all content through a unified domain.
- Using team prefixes in the URL path is an excellent way to make debugging and routing easier.
- Every piece of software should have clear ownership. When possible, avoid creating horizontal teams like a platform team.

4 Server-side Composition

This chapter covers

- Examining server-side composition using Nginx and SSI.
- Investigating how timeouts and fallbacks can help when dealing with broken or slow fragments.
- Comparing the performance characteristics of different composition techniques.
- Exploring alternative solutions like Tailor, Podium, and ESI.

In the previous chapters, you learned how to build a micro frontends style site using client-side integration techniques like links, iframes, and AJAX. You've also learned how to run a shared web-server that routes incoming requests to the responsible team for a specific part of the application. In this chapter, we will build upon these and look at server-side integrations. Assembling the markup of different fragments on the server is a widespread and popular solution. Many e-commerce companies like Amazon, IKEA, or Zalando have chosen this way.

Server-side composition is typically done by a service that sits between the browser and the actual application servers, as illustrated in figure 4.1.

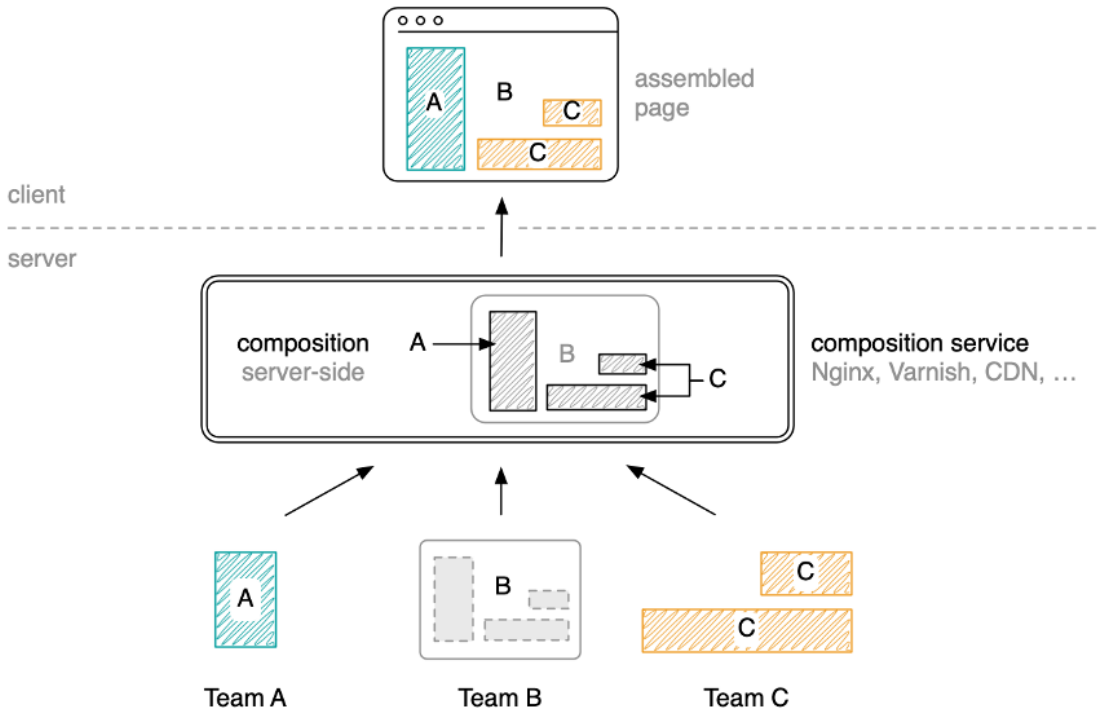


Figure 4.1 Composition of fragments happens on the server. The client receives an already assembled page.

The most significant benefit if server-side integration is that the page is already fully assembled when it reaches the customer's browser. You can achieve incredibly good first-page load speeds that are hard to match using pure client-side integration techniques.

Another essential factor is robustness. Composing the application server-side provides the foundation for adopting the principals of progressive enhancement. Teams can decide to add client-side JavaScript to the fragments where it improves the user experience.

4.1 Composition via Nginx & Server-Side Includes (SSI)

In the last iteration, the teams switched their integration from iframes to an AJAX-based solution. This switch improved their search engine ranking noticeably. To validate their work *Tractor Models Inc.* conducts surveys regularly. Tina, responsible for customer service, speaks more than ten languages. She talks to enthusiasts from around the world to get their opinion and feedback.

The overall reaction to their work is stellar. The fans can't wait to get their hands on the real tractor models. But a topic that came up multiple times during these conversations was the loading speed of the site. Customers reported that using the-tractor.store does not feel as snappy as their competitor's online shop. Elements like the recommendation strip appear with a noticeable delay.

Tina organized an in-person meeting with the development teams to share the insights from her calls. The developers were surprised by the poor performance reports. On their machines, all pages load pretty fast. The couldn't even see the effects that the customers described on their machines. But this might be because their customers don't own $3,000 notebooks, aren't on a fiber connection, and don't live in the same country where the datacenter is located. Most of them don't even live on the same continent.

To test their site under suboptimal network conditions, one developer opens his browsers developer tools and loads the page with the network throttled to 3G speeds. He was quite surprised to see it took 10 seconds to load.

The developers are confident that there is room for improvement. They plan to move to a server-side integration technique. This way, the first HTML response would already include the references for all assets the site needs. The browser has a complete picture of the page much sooner. It can load the needed resources earlier and in parallel.

Since they already have an Nginx in place, the teams chose to use it's Server-Side Includes (SSI) feature to do the integration.

4.1.1 How to do it

SIDEBAR **SSI history**

Server-Side Includes is an old technique. It dates back to the 1990s. Back in the days, people used it to embed the current date into an otherwise static page. In this book, we will focus on SSI's `include` directive in the Nginx server.

The specification is stable. It has not evolved over the last years. The implementations in popular web-servers are rock solid and come with little management overhead.

Let's get to work. This time *Team Inspire* can lean back. They can reuse the recommendation fragment endpoint from the last chapters (AJAX).

Team Decide needs to make two changes:

1. Activate Nginx' SSI support in the web-servers configuration
2. Add an SSI comment which is pointing to the recommendation fragment location to their view.

HOW SSI WORKS

Let's have an overview look at how SSI processing works. A SSI include directive looks like this:

```
<!--#include virtual="/url/to/include" -->
```

The web-server replaces this directive with the contents of the referenced URL before it passes the markup to the client.

Figure 4.2 shows how our systems generate and compose the HTML for the product page using server-side includes. Let's follow the arrows from the initial request to the final response from top to bottom. All the steps are happening in sequential order.

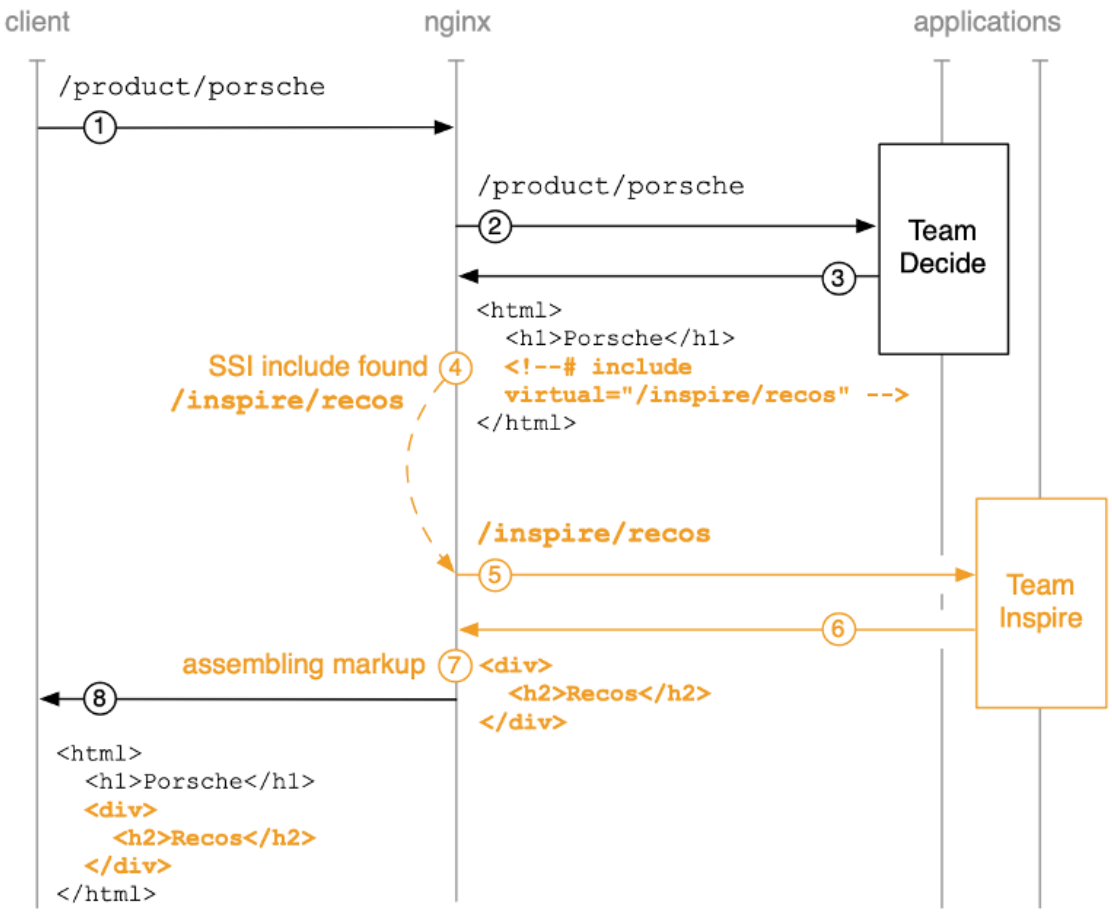


Figure 4.2 SSI processing inside Nginx

1. **client** requests `/product/porsche`
2. **Nginx** forwards the request to *Team Decide* because it starts with `/product/`
3. **Team Decide** generates the markup for the product page, including an SSI directive where the recommendations should be placed and send it to Nginx
4. **Nginx** parses the response body, finds the SSI include and extracts the URL (`virtual`)
5. **Nginx** requests its content from *Team Inspire* because the URL starts with `/inspire/`
6. **Team Inspire** produces the markup for the fragment and returns it
7. **Nginx** replaces the SSI comment on the product pages markup with the fragments markup

8. **Nginx** sends the combined markup to the browser

The Nginx serves two roles: **request forwarding** based on the URL path and **fetching and integrating fragments**.

INTEGRATING A FRAGMENT USING SSI

Let's go-ahead to try this in our example application. Nginx's SSI support is disabled by default. You can activate it by putting `ssi on;` into the `server {…}` block of your `nginx.conf`.

Listing 4.1 webserver/nginx.conf

```
...
server {
  listen 3000;
  ssi on;          ❶
  ...
}
```

❶ activates Nginx's server-side include feature

Now we must add the SSI include directive to the product pages markup. It follows a simple structure: `<!--#include virtual="[/url-to-include]" -`. We can use the same URL for the fragment as we did with the AJAX example before.

Listing 4.2 team-decide/product/porsche.html

```
...
<aside class="decide_recos">
  <!--#include virtual="/inspire/fragment/recommendations/porsche" -->   ❶
</aside>
...
```

❶ Nginx will replace this SSI directive by the contents of the URL.

Start the example by running the following command:

```
npm run 05_ssi
```

Your browser now shows the tractor page as we know it. However, we don't need client-side JavaScript for the integration anymore. The markup is already integrated when it reaches your customer's device. Check this by opening "view source" in your browser.

4.1.2 Better load times

Let's have a look at page-load speed. Open the network tab in your browser's developer tools. We activate network throttling to 3G speeds. Figure 4.3 shows the result.

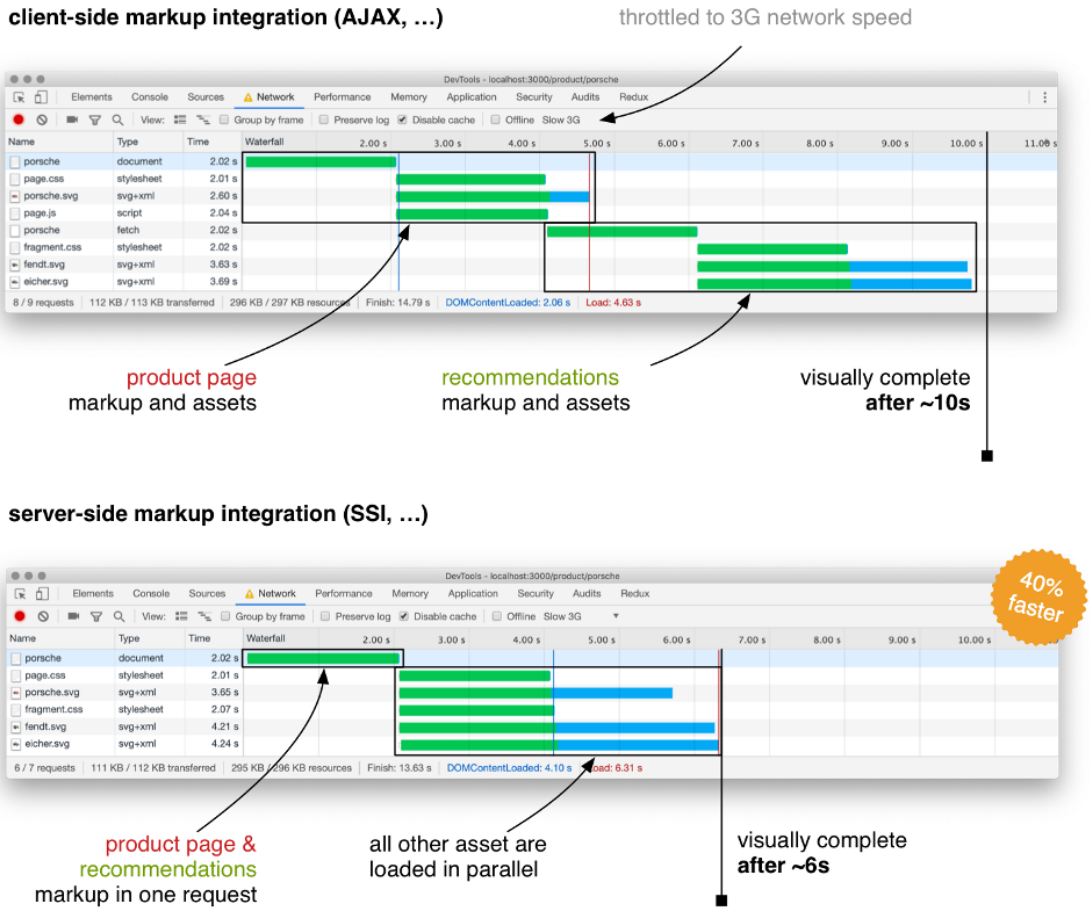


Figure 4.3 Page-load speed for the product page with client-side and server-side composition. Server-side integration optimizes the critical paths.

Loading the AJAX integrated version takes around 10 seconds compared to only 6 seconds for the SSI solution. The page indeed loads 40% faster. But where did we save so much time? Let's have a more in-depth look. The 3G throttling mode limits the available bandwidth but also delays all requests by around two seconds. We removed the need for the separate fragment AJAX call. The recommendations are already bundled into the initial markup. This bundling saves us two seconds. The other factor is that JavaScript triggered the loading of the AJAX call. The browser had to wait for the JavaScript file to finish before it was able to load the fragment. This *waiting for JavaScript* accounted for another two seconds.

Granted, delaying all requests by two seconds seems harsh and might not accurately represent the average connectivity of your customer. But it highlights the dependencies of your resources, also called the critical path. It's essential to give the browser the information about all crucial parts of the page, like images and styles, as early as possible. Server-side integration is essential in making this happen.

The critical difference is that latency inside one data center is magnitudes smaller and more predictable. You are talking about single-digit milliseconds for service to service

communications. Whereas the back and forth over the internet, between data-center and end-user, is much more unreliable. Latency ranges from < 50ms on good connections and multiple seconds for bad ones.

4.2 Dealing with unreliable fragments

The developers of *Team Decide* generated a comparison video showing the realtime page load before and after the server-side integration .[34] and posted it to the companies Slack channel. As expected, the responses were extremely positive.

But what happens when one of the applications is slow or has a technical problem? In this section, we'll dig a little bit deeper into server-side integration and explore how timeouts and fallbacks can help.

4.2.1 The flaky fragment

While *Team Decide* worked on the server-side integration, *Team Inspire* was also busy. They were able to build the prototype of a new feature called "Near You". It informs the tractor fan when a real-world version of one of his favorite models is working on a field nearby. Making this work wasn't easy: Talking to farmer associations, distributing GPS kits to the farmers, and making the realtime data-collection happen.

When a user visits the site, and the system detects that there is indeed a real version of the tractor in a 100km radius near him, we show a little information box on the product page. The first version of this feature will be limited to Europe and Russia and locates the user by her IP-address. The location is not always accurate, and they plan to leverage the browser's native geolocation and notifications APIs in the future.

Both teams sit together and talk about how to integrate this feature. *Team Inspire* just needs a second space in the product pages layout. *Team Decide* agrees to provide a slot for the "Near You" fragment as a long banner underneath the header on the product page. When *Team Inspire's* system can't find a nearby tractor, it will show no banner. But *Team Decide* does not have to know and care about the business logic and concrete implementation of the fragment. Topics like localization, finding a match, rollout plans, etc. are handled by *Team Inspire*. When they are unable to find a match, they'll return an empty fragment. Figure 4.4 shows how the banner will look like.

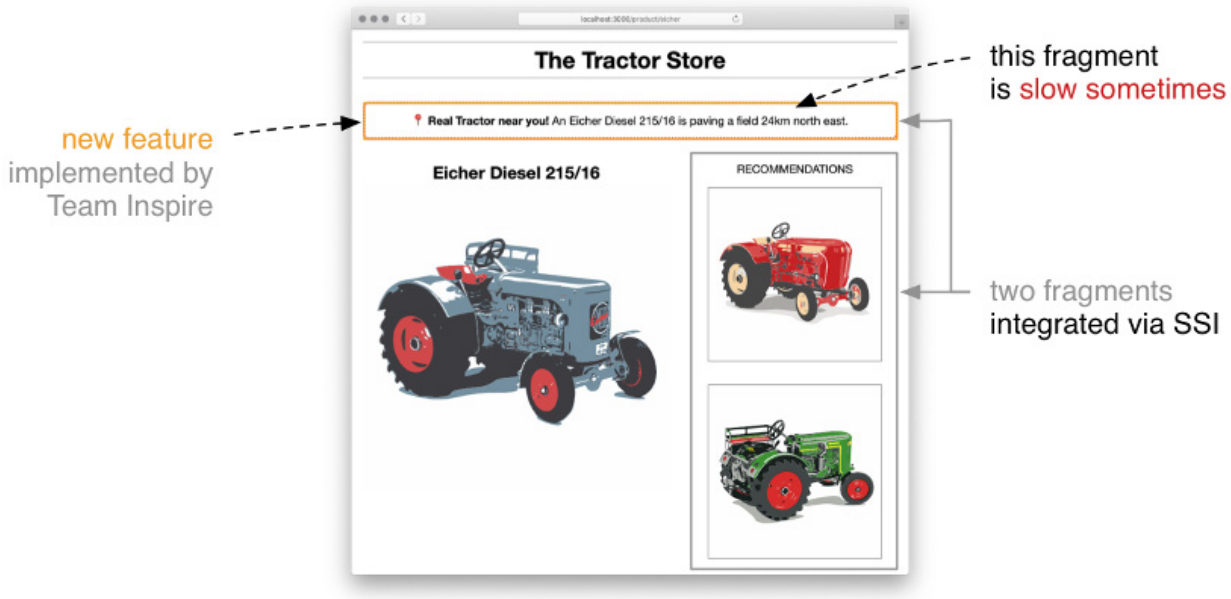


Figure 4.4. The "Near You" feature is added as a banner on top of the page.

Team Inspires says that the URL pattern of the fragment will be `/inspire/fragment/near_you/<sku>`. But before both teams separate to start working one of *Team Inspires* developer raises an issue: "Our data processing stack still has a few problems. Sometimes our response times go up to over 500ms for a couple of minutes. During our last tests, the servers also crashed and rebooted sometimes."

This unreliability, indeed, is an issue. 500ms is quite a long time for a single fragment. It will slow down the markup generation for the complete product page. But since this feature is not crucial for the site to work, they agree on leaving it out when it takes to long.

4.2.2 Integrating the "Near You" fragment

TIP	You can find the sample code for this task in the `06_timeouts` folder.

Let's have a look at *Team Inspires* new fragment.

Listing 4.3. team-inspire/inspire/fragment/near_you/eicher.html

```
<link href="/inspire/static/fragment.css" rel="stylesheet" />  (1)
<div class="inspire_near_you">                                 (2)
  🔑 <strong>Real Tractor near you!</strong>                    (2)
  An Eicher Diesel 215/16 is paving                            (2)
  a field 24km north east.                                     (2)
</div>                                                         (2)
```

(1) fragment stylesheet

(2) fragment conten

At the moment, only *Eicher Diesel 215/16* tractors are GPS equipped. The fragments for the other tractors (`porsche.html`, `fendt.html`) are just a blank file. To display the fragment, *Team Decide* inserts the associated SSI directive to their product pages.

Listing 4.4 team-decide/product/eicher.html

```
...
<h1 class="decide_header">The Tractor Store</h1>
<div class="decide_banner">
  <!--#include virtual="/inspire/fragment/near_you/eicher" -->
</div>
...
```

But since we are serving static HTML files, the response time for the fragment would always be fast. Let's simulate a slow fragment.

You can find the source code in `06_timeouts`. This time we have three scenarios we can test with this example:

1. *Team Inspire* has a **short delay** (~300ms)
2. *Team Inspire* has a **long delay** (~1000ms)
3. *Team Inspire* is down

I've created an NPM run script for each scenario. Let's have a look at the first one: the 300ms delay. Run the following command:

```
npm run 06_timeouts_short_delay
```

Now the page takes considerably longer to load. In the `05_ssi` example, the HTML document loaded in single-digit milliseconds. With the slow fragments from *Team Inspire*, it takes more than 300ms before the browser receives any data from the server. These potential delays are an inherent problem of server-side composition. The composition service has to wait for all the required fragments.

In contrast to the AJAX integration, where we fetch fragments asynchronously, one single fragment can slow down the complete page in a server-side integration. On the server, **the slowest fragment defines the total response time**. All teams need to monitor the response times of their fragments to achieve excellent performance. Let's look at the other two scenarios: long delays and broken upstream.

4.2.3 Timeouts & fallbacks

Even if everything is fast, most of the time, it's still a good idea to have a safety-net in place. In a micro frontends architecture, you want to decouple your user interface as well as possible. An error in one system should not break the others. Nginx comes with basic mechanisms to define timeouts for upstreams. When an upstream becomes slow or doesn't respond at all, Nginx stops waiting and delivers the site without the includes.

Let's have a look at how Nginx behaves when a team's application doesn't respond at all. Run the following command to simulate what happens if *Team Inspire* is down.

```
npm run 06_timeouts_down
```

Our page loads pretty quick, but *Team Inspire's* fragments are missing. Since Nginx couldn't connect to *Team Inspire's* application, it did not have to wait.

But in reality, it's not always that black and white. Sometimes a server accepts new connections but responds slowly. With the property `proxy_read_timeout` you can configure a timeout after which Nginx categorizes an upstream as non-functional. The default timeout is `60s`, which is pretty high for our use-case. We could set the `proxy_read_timeout` to `500ms` for all requests starting with `/inspire/`. 500 milliseconds is the maximum response time both teams agreed upon earlier. The Nginx configuration looks like this:

Listing 4.5 webserver/nginx.conf

```
...
  location /inspire/ {
    proxy_pass  http://team_inspire;
    proxy_read_timeout 500ms;          ❶
  }
...
```

❶ *Team Inspire's* upstream has a maximum of 500ms to produce an answer for incoming requests.

You need to keep in mind that this is a per upstream and not a per request setting. When requests exceed the configured timeouts, Nginx marks the corresponding upstream as failed and stops even trying to contact it for 10 seconds.[35]

Let's test our configured timeout by running the following command:

```
npm run 06_timeouts_long_delay
```

In this scenario, we delay all calls to *Team Inspire* by 1000ms. Since this exceeds our configured timeout, Nginx omits *Team Inspire's* fragments. Watch your network view to see that the HTML document takes ~500ms to load for the first time. Also, notice that Nginx answers all subsequent

requests to the product detail page instantly (< 10ms). Nginx doesn't even try to contact *Team Inspire's* application for at least 10s. After that 10s, Nginx will try again.

> **NOTE** It's not possible to configure a timeout in Nginx that only aborts sporadic long-running requests. When some requests take to long, Nginx marks the complete upstream as non-functional. Later in this chapter, we'll look at alternative server-side integration techniques that provide more flexibility when it comes to timeouts.

4.2.4 Fallback content

You might have noticed that Nginx omits the "Near You" fragment when it takes to long. But the recommendation strip wasn't completely removed. Instead, the page shows a "Show Recommendations" in its place.

Nginx has a builtin mechanism to deal with failed includes. The SSI command has a parameter called `stub`. It lets you define a reference to a block. Nginx uses the content of the block when something goes wrong with the include. We can define the fallback content by wrapping it in `block` and `endblock` comments. Here's the fallback markup *Team Decide* has configured for the recommendations.

Listing 4.6 team-decide/product/eicher.html

```
...
<aside class="decide_recos">
  <!--# block name="reco_fallback" -->      ❶
    <a href="/recommendations/eicher">      ❶
      Show Recommendations                  ❶
    </a>                                    ❶
  <!--# endblock -->                        ❶
  <!--#include
      virtual="/inspire/fragment/recommendations/eicher"
      stub="reco_fallback" -->              ❷
</aside>
...
```

❶ Defining the fallback content as `reco_fallback`.

❷ Assigning the `reco_fallback` block as fallback/stub to the includes.

But you don't always have a meaningful fallback. In production, it's common to use an empty block for content that is optional for the site to work.

Listing 4.7 team-decide/product/eicher.html

```
...
<div class="decide_banner">
  <!--# block name="near_you_fallback" --><!--# endblock -->    ❶
  <!--#include
      virtual="/inspire/fragment/near_you/eicher"
      stub="near_you_fallback" -->                              ❶
</div>
...
```

❶ Empty fallback content named `near_you_fallback`.

❷ Assigning the `near_you_fallback` block as a fallback.

> **NOTE** The placement in the document does not matter. However, you must define the block before you reference it via the `stub`.

Thinking about fallbacks and timeouts is crucial when you implement server-side composition. Otherwise, a misbehaving fragment can harm the complete page. The Nginx way you just learned is by far not the only way to deal with it, but the concepts are transferable to most other solutions.

4.3 Markup assembly performance in depth

In the examples before, we've seen that one fragment can slow down the complete page. We'll now look deeper into the topic of loading multiple fragments at once, dealing with nested fragments, and how to implement deferred loading. After that, we'll dig a little bit deeper better to understand the response behavior of Nginx and other solutions.

4.3.1 Parallel loading

We already observed how Nginx resolves and replaces an SSI include. But what happens when there is more than one fragment to fetch? Figure 4.5 shows the network diagram for our two fragment product page.

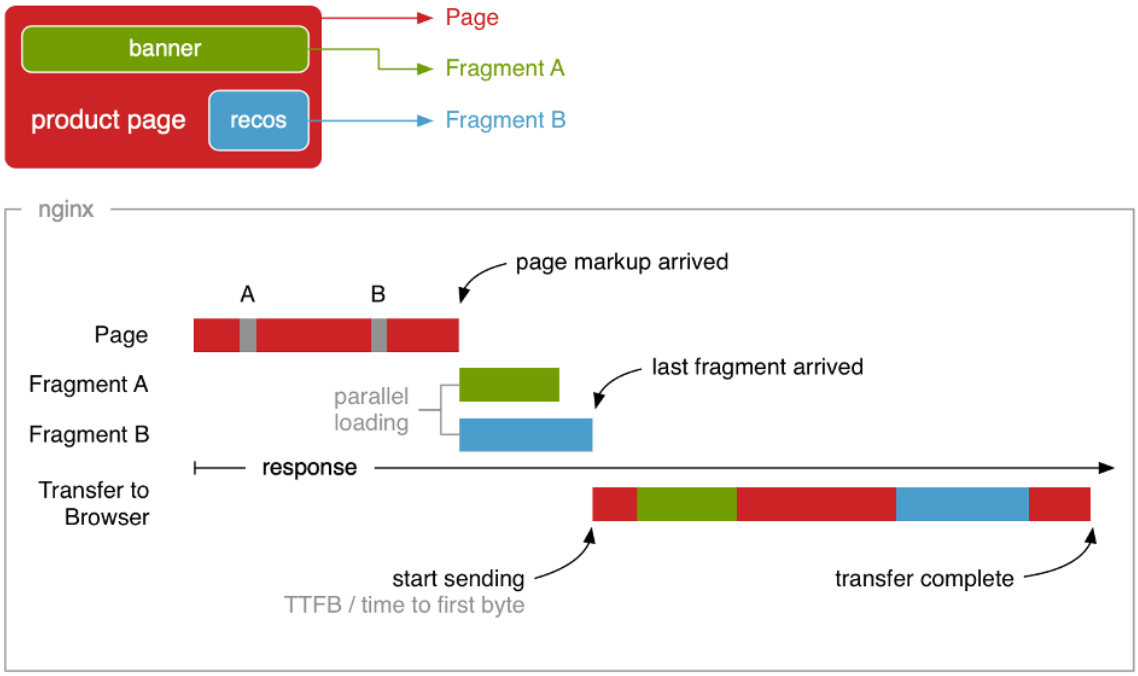


Figure 4.5 Nginx fetches multiple SSI includes in parallel

After Nginx receives the HTML for the product page, it parses the content and finds two SSI directives (A & B) that it must resolve. Then it goes ahead and requests all fragments in parallel. When the last fragment arrives, Nginx assembles the complete markup and sends the response back to the client.

So SSI processing is a two-step process:

1. fetching the page markup
2. fetching all fragments in parallel

The response time for the complete markup, also called time to first byte (TTFB), is defined by the time it takes to generate the page markup and the time of the slowest fragment.

4.3.2 Nested fragments

It's also possible to nest SSI includes: having a fragment that contains another fragment. Nginx checks all responses, even included ones, for SSI directives and executes them. In the projects I've worked on, we always tried to avoid nesting includes. Every additional level of nesting adds to the load time. The two-step process quickly becomes a three, four, or five-step process. If this nesting is acceptable or not depends on your performance target and the time it takes to generate a fragment.

A scenario where nesting always came up was the page header. Many pages include the header fragment. But the header itself is assembled out of different other fragments, for example, the

mini-cart, navigation, or login-status. Figure 4.6 illustrates this nesting.

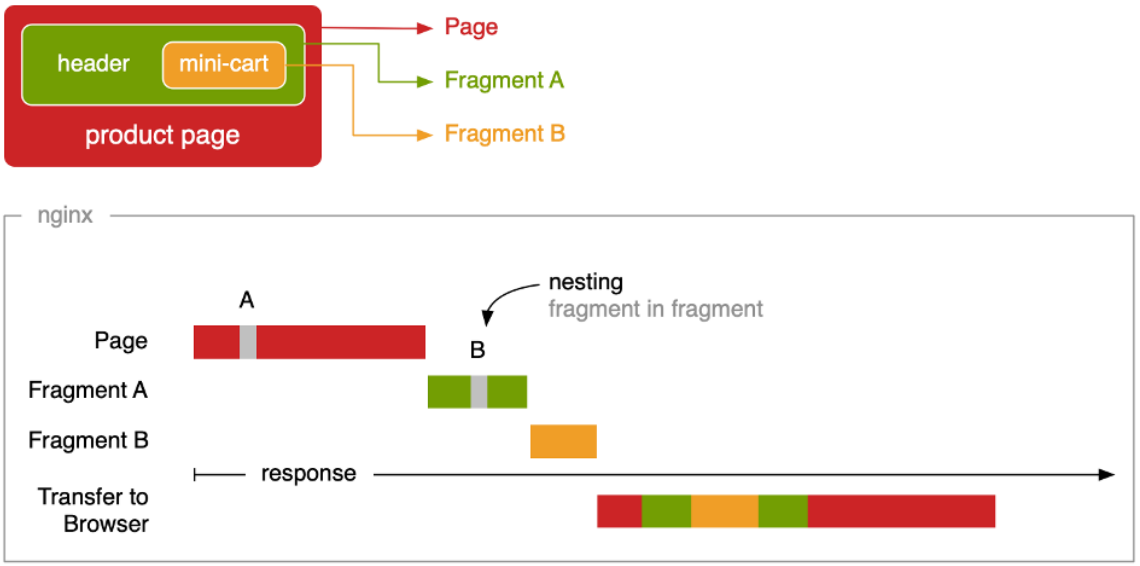


Figure 4.6 Product page includes the header fragment which itself includes the mini-cart fragment.

Since the parts for the header were either quite static and cacheable (navigation) or small and quick to produce (mini-cart, login-status), we usually accepted this indirection.

4.3.3 Deferred loading

Server-side integration is a great tool to improve the load time of your page. But you have to be careful when creating large pages. It's often a good practice to use server-side integration for the essential parts of your page - usually everything in the upper part (viewport). Additional fragments that are farther down the page or are optional for your site to work (newsletter signup, promotions) can be lazy-loaded, e.g., via AJAX. Lazy loading reduces the size of the initial markup the client needs to load and enables the browser to start rendering the page earlier.

If you want the fragment in the initial markup, you specify it as an SSI directive.

```
<div class="banner">
  <!--#include virtual="/fragment-a" -->
</div>
```

If you want to lazy load it, you can omit the `include` directive and fetch the content using an AJAX call via client-side JavaScript instead.

```
const banner = document.querySelector(".banner"):
window
  .fetch("/fragment-a")
  .then(res => res.text())
  .then(html => { banner.innerHTML = html; });
```

Since the fragment endpoints for an SSI or AJAX-based integration can be the same, it's easy to switch between those integrations and test the results.

4.3.4 Time to first byte & streaming

Let's look at some optimization techniques a composition service can implement to speed up the page load time. We've seen how Nginx works. It loads the main document and waits until all referenced fragments have arrived. It **sends** the response to the client **after it has assembled the page**.

But there are better ways. A composition service could start sending the first chunks of data earlier. It could, for example, already send the beginning of the page template up until the first fragment, then send the remaining chunks as fragments arrive. This **partial sending** would be beneficial for performance because the browser can start loading assets and render the first parts of the page earlier. The ESI mechanism in the Varnish Enterprise works like this. You'll learn more about ESI in the next section.

The idea of **streaming** templates takes this one step further. With this model, the upstreams generate and send their markup as a stream. The product page would immediately send out the first parts of its template while looking up the required data for the rest of the page (name, image, price) in parallel. The composition server can directly pass this data to the client and start fetching fragments even if the page's markup from the other upstream hasn't completely arrived yet. The two steps (loading page, loading fragments) overlap, which can reduce overall load time and improves time to first byte significantly. In the next section, we'll have a look at Tailor and Podium, which both support streaming composition.

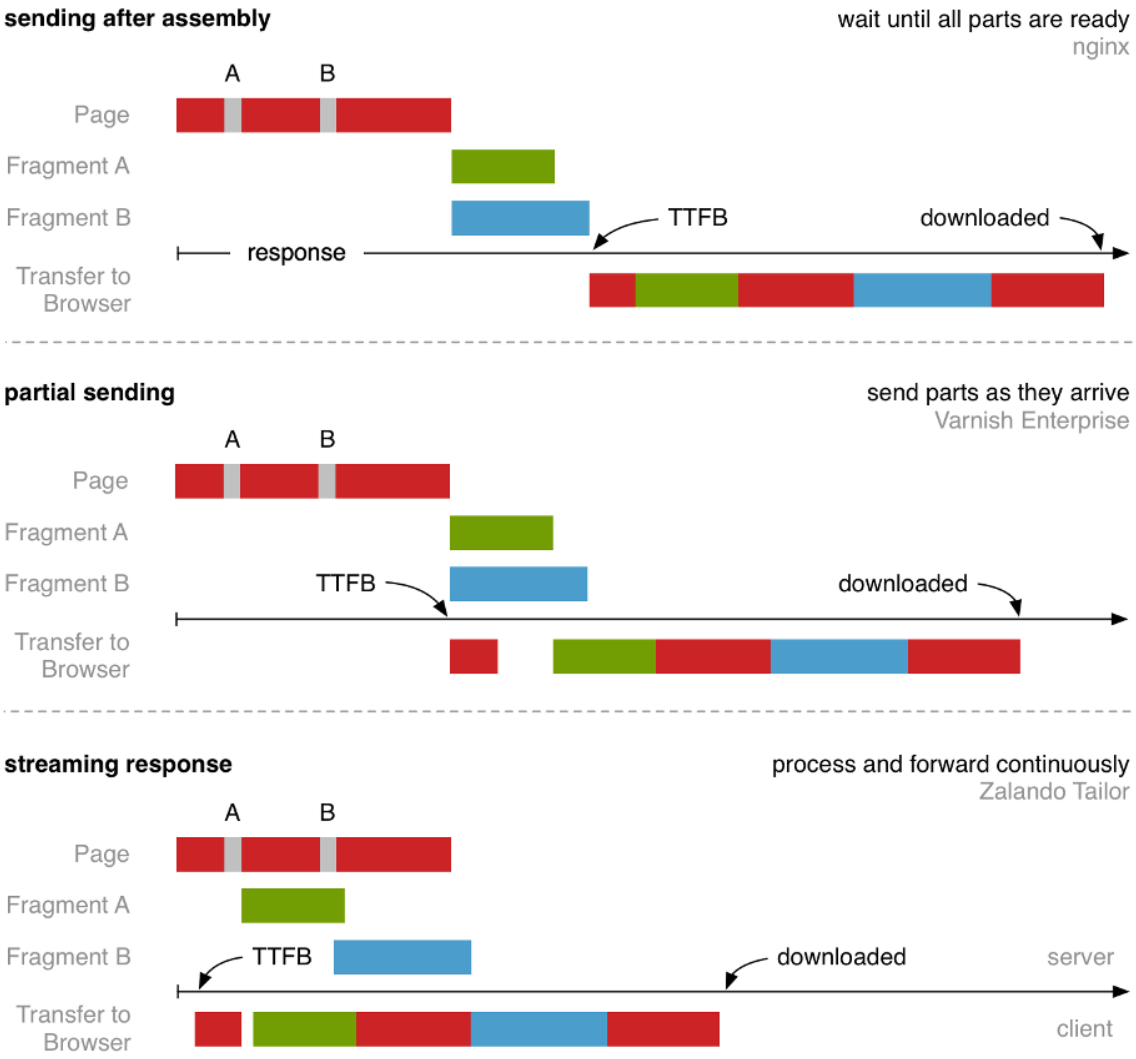


Figure 4.7 Different ways of how a server-side integration solution can handle fragment loading and markup concatenation internally. The partial sending and streaming approach provides a better time to first byte. This way, the browser receives the content earlier and can start rendering quicker.

Figure 4.7 shows a diagram of the three approaches. There are a few simplifications made that you need to keep in mind:

- The diagram does not take into account that the user's bandwidth is limited.
- The streaming model includes the assumption that the response generation is a linear process. This assumption is only valid if you are serving up static documents. Most applications usually fetch data from a database before templating is started at all. Data fetching typically takes a significant part of the response time.

4.4 A quick look into other solutions

Up until now, we focused on how to integrate using SSI and looked at Nginx's implementation in specific. Let's examine a few alternatives. We'll focus on their main benefits.

4.4.1 Edge Side Includes

Edge Side Includes, short ESI, is a specification.[36] that defines a unified way for markup assembly. Content delivery network providers like Akamai and proxy servers like Varnish, Squid, and Mongrel support ESI. Setting up an ESI integration solution would look similar to our example. Instead of putting an Nginx between the browser and our applications, we could swap it with a Varnish server. An edge side include directive looks like this:

```
<esi:include src="https://tractor.example/fragment" />
```

FALLBACKS

The `src` needs to be an absolute URL, and it's also possible to define a link for a fallback URL by adding an `alt` attribute. This way, you can set up an alternative endpoint that hosts the fallback content. The associated code would look like this:

```
<esi:include
  src="https://tractor.example/fragment"
  alt="https://fallback.example/sorry" />     ❶
```

❶ If the fragment (`src`) fails to load the content from the fallback URL (`alt`) will be shown instead.

TIMEOUTS

Like SSI, standard ESI has no way to define a timeout for individual fragments. Akamai added this feature with their non-standard extensions.[37] There you can add a `maxwait` attribute. When the fragment takes longer, the service will skip it.

```
<esi:include
  src="https://tractor.example/fragment"
  maxwait="500" />                    ❶
```

❶ Fragment is skipped if it takes longer than 500ms to load

TIME TO FIRST BYTE

The response behavior varies between implementations. Varnish fetches the ESI includes in series - one after another. Parallel fragment loading is available in the commercial edition of the software. This version also supports partial sending, which starts responding to the client early - even when it hasn't resolved all fragments yet.

4.4.2 Zalando Tailor

Zalando.[38] moved from a monolith to a micro frontends style architecture with *Project Mosaic*.[39] They published parts of their server-side integration infrastructure. *Tailor* .[40] is a Node.js library that parses the pages HTML for special `fragment`-tags fetch the referenced content and puts it into the page's markup.

We won't go into full detail on how to set up a tailor based integration. But here are some parts of the code to give you an impression. Tailor is available as a package (`node-tailor`). You can install it via NPM.

Listing 4.8 team-decide/index.js

```
const http = require('http');
const Tailor = require('node-tailor');
const tailor = new Tailor({ templatesPath: './views' });    ❶
const server = http.createServer(tailor.requestHandler);    ❷
server.listen(3001);                                        ❷
```

❶ Creating a tailor instance and setting it's template folder to `./views`. Consult the documentation.[41] for other options.

❷ Attaching tailor to a standard Node.js server, which listens on port 3001.

An associated template could look like this:

Listing 4.9 team-decide/views/product.html

```
...
<body>
  <h1>The Tractor Store</h1>
  ...
  <fragment src="http://localhost:3002/recos" />   ❶
</body>
...
```

❶ the `fragment`-tag will be replaced by the content fetched from the `src`

This example is a simplified version of our product page. Team Decide runs the Tailor service in their Node.js application. Their tailor server will handle a call to localhost:3001/product. It uses the `./views/product.html` template to generate a response. Tailor replaces the `<fragment … />`-tag with the HTML content that the localhost:3002/recos endpoint returns. *Team Inspire* operates this endpoint.

FALLBACKS & TIMEOUTS

Tailor has builtin support for handling slow fragments. It lets you define a per-fragment timeout like this:

```
<fragment
```

```
  src="http://localhost:3002/recos"
  timeout="500"                                          ❶
  fallback-src="http://localhost:3002/recos/fallback"    ❷
/>
```

❶ Sets a 500ms timeout for this fragment.

❷ Tailor loads the fallback content in case of an error or timeout.

When the loading fails, or the timeout exceeds, the `fallback-src` URL gets called to show fallback content.

TIME TO FIRST BYTE & STREAMING

Tailor's most prominent feature is the support for streaming templates. They send the result to the browser as the page template (they call it layout) is parsed, and fragments arrive. This streaming approach leads to a good time to first byte.

ASSET HANDLING

Besides the actual markup, a fragment endpoint can also specify associated styles and scripts that go with this fragment. Tailor uses HTTP headers for this.

```
$ curl -I http://localhost:3002/recos                                    ❶
HTTP/1.1 200 OK
Link: <http://localhost:3002/static/fragment.css>; rel="stylesheet",     ❷
      <http://localhost:3002/static/fragment.js>;  rel="fragment-script" ❷
Content-Type: text/html
Connection: keep-alive
```

❶ Requesting the response headers of the fragment.

❷ Associated assets (CSS, JS) are listed in the `Link` header of the fragment.

Tailor reads these headers and adds the scripts and styles to the document. Transfering the references alongside the markup is great and enables optimizations like not referencing the same resource twice and moving all script tags to the bottom of the page.

But Tailor's implementation makes some assumptions that might not be generally applicable. Teams must wrap all JavaScript in an AMD module, which will be loaded by the require.js module loader. You also can't easily control the service adds script and style tags to the markup.

4.4.3 Podium

Finn.no.[42] is a platform for classified ads and Norway's largest website by the number of page views. They are organized in small autonomous teams and also assemble their pages out of fragments, which they call *podlets*. They released their Node.js based integration library called Podium.[43] beginning of 2019. It takes the concepts from Tailor and improves them. In Podium, fragments are called **podlets**, and pages are **layouts**.

PODLET MANIFEST

Podium's central concept is the *Podlet Manifest*. Every podlet comes with a JSON structured metadata endpoint. This file contains information like name, version, and the URL for the actual content endpoint.

Listing 4.10 http://localhost:3002/recos/manifest.json

```
{
  "name": "recos",
  "version": "1.0.2",
  "content": "/",                          ❶
  "fallback": "/fallback",                 ❷
  "js": [                                  ❸
    { value: "/recos/fragment.js" }        ❸
  ],                                       ❸
  "css": [                                 ❸
    { value: "/recos/fragment.css" }       ❸
  ]                                        ❸
  ...
}
```

❶ Endpoint for the actual HTML markup.

❷ Cacheable fallback content.

❸ Associated JS and CSS assets.

It can also specify where to find cacheable fallback markup and references to the CSS, JS assets. As you can see in figure 4.8, the podium manifest acts as a machine-readable contract between the owner of the podlet and its integrator.

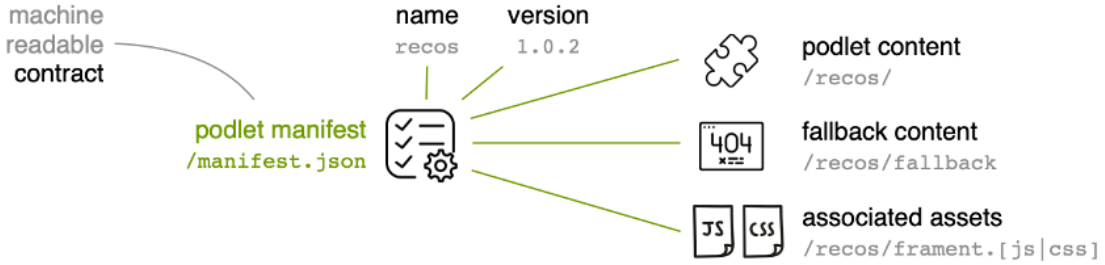


Figure 4.8 Each podlet has its manifest.json, which contains basic metadata but can also include references to fallback content and asset files. The manifest acts as the technical contract between the different teams.

PODIUM'S ARCHITECTURE

Podium consists of two parts:

- The **layout library** works in the server that delivers the page. It implements everything needed to retrieve the podlet contents for this page. It reads the `manifest.json` endpoints for all used podlets and also implements concepts like caching.

- The **podlet library** is used by the team, which provides a fragment. It generates a `manifest.json` for each fragment.

Figure 4.9 illustrates how the libraries work together. *Team Decide* uses `@podium/layout` and registers *Team Inspire's* manifest endpoint. *Team Inspire* implements `@podium/podlet` to provide the manifest.

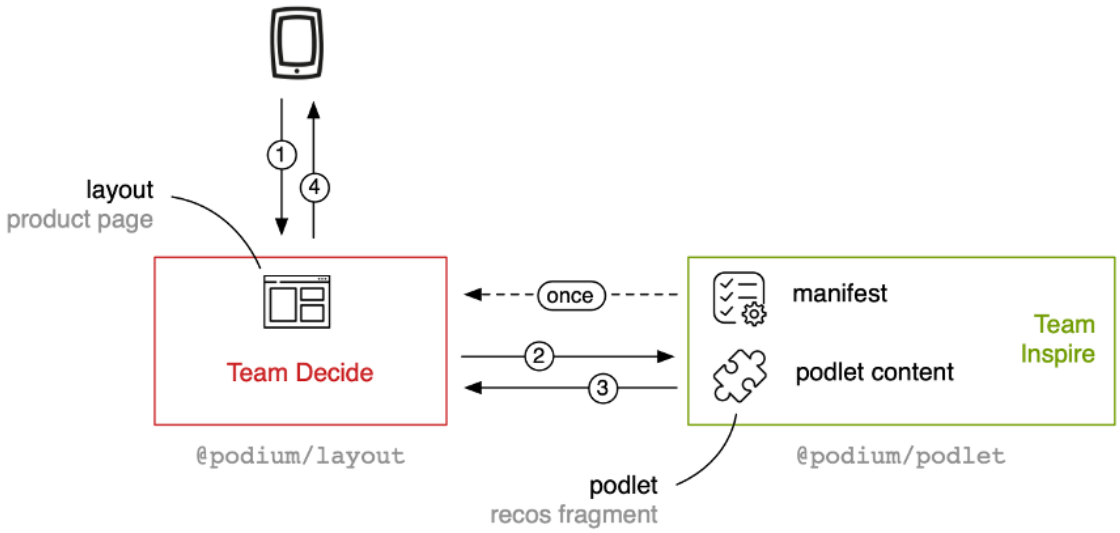


Figure 4.9 Simplified overview of Podium's architecture. The team which delivers a page (layout) communicates with the browser. It fetches fragment content (podlet) directly from the team that generates it. Associated manifest information is only requested once, not on every request.

Team Decide **reads the manifest** for the recommendation fragment **only once** to obtain all metadata needed for integration. Let's follow the numbered steps to see the processing of an incoming request:

1. *Browser* asks for the product page. *Team Decide* receives the request directly.
2. *Team Decide* needs the recommendation fragment from *Team Inspire* for its product page. It requests the podlet's content endpoint.
3. *Team Inspire* responds with the markup for the recommendation. The response is a plain HTML like in the Nginx examples.
4. *Team Decide* puts the received markup to its product page and adds the required JS/CSS references from the manifest file. *Team Decide's* application sends the assembled markup to the browser.

IMPLEMENTATION

We can't go into full detail on how to use Podium. But we'll briefly look at the key parts required to make this integration work.

Each of the teams creates it's own Node.js based server. We are using the popular express.[44]

framework as a web-server, but other libraries also work.

These are *Team Decide's* dependencies:

Listing 4.11 team-decide/package.json

```
...
"dependencies": {
  "@podium/layout": "^4.5.0",
  "express": "^4.17.1",
}
...
```

The Node.js code necessary to run the server and configure podiums layout service looks like this:

Listing 4.12 team-decide/index.js

```
const express = require("express");
const Layout = require("@podium/layout");

const layout = new Layout({                                    ❶
  name: "product",                                             ❶
  pathname: "/product",                                        ❶
});                                                            ❶

const recos = layout.client.register({                         ❷
  name: "recos",                                               ❷
  uri: "http://localhost:3002/recos/manifest.json"             ❷
});                                                            ❷

const app = express();                                         ❸
app.use(layout.middleware());                                  ❸

app.get("/product", async (req, res) => {                      ❹
  const recoHTML = await recos.fetch(res.locals.podium);       ❺

  res.status(200).podiumSend(`                                 ❻
    ...                                                        ❻
    <body>                                                     ❻
      <h1>The Tractor Store</h1>                               ❻
      <h2>Porsche-Diesel Master 419</h2>                       ❻
      <aside>${recoHTML}</aside>                               ❻
    </body>                                                    ❻
    </html>                                                    ❻
  `);                                                          ❻
});

app.listen(3001);
```

❶ Configuring the layout service. It's responsible for the communication with the podlets. It also sets HTTP headers and transfers context information.

❷ Registering the recommendation podlet from *Team Inspire*. The application fetches metadata from the `manifest.json`. The `name` is for debugging in internal reference.

❸ Creating an express instance and attaching podiums layout middleware to it.

❹ Defining the route `/product` that delivers the product page.

❺ `recos` is the reference to the podlet we registered before. `.fetch()` retrieves the markup from *Team Inspire's* server. It returns a Promise and takes a context object as its parameter. The context `res.locals.podium` is provided by the layout service and may contain information such as locale, country code, or user status. We pass this context to *Team Inspire's* podlet server.

❻ Returns the markup for the product page. The `recoHTML` contains the plain HTML returned by the `.fetch()` call.

As said before, we won't go into full detail on this code. The code annotations should give you a pretty good idea of what's happening here. Open up `07_podium` in the example code to see the full applications. You can start them via this command:

```
npm run 07_podium
```

The most interesting fact you can observe in this code is that Podium is pretty unopinionated when it comes to templating. You can use your Node.js template solution of choice. Podium just provides a function to retrieve the markup of a fragment: `await recos.fetch()`. How you place the result into your layout is entirely up to you. For simplicity, we are using a plain template-string here. This `fetch()` call also encapsulates timeout and fallback mechanisms.

Let's switch teams and look at the code *Team Inspire* needs to write to implement their podlet. These are their dependencies:

Listing 4.13 team-inspire/package.json

```
...
"dependencies": {
  "@podium/podlet": "^4.3.2",
  "express": "^4.17.1",
}
...
```

And this is the application code:

Listing 4.14 team-inspire/index.js

```
const express = require("express");
const Podlet = require("@podium/podlet");

const podlet = new Podlet({                          (1)
  name: "recos",                                     (1)
  version: "1.0.2",                                  (1)
  pathname: "/recos",                                (1)
});                                                  (1)

const app = express();                               (2)
app.use("/recos", podlet.middleware());              (2)

app.get("/recos/manifest.json", (req, res) => {      (3)
  res.status(200).json(podlet);                      (3)
});                                                  (3)

app.get("/recos", (req, res) => {                    (4)
  res.status(200).podiumSend(`                       (4)
    <h2>Recommendations</h2>                         (4)
    <img src=".../fendt.svg" />                      (4)
    <img src=".../eicher.svg" />                     (4)
  `);                                                (4)
});                                                  (4)

app.listen(3002);
```

(1) Defining a podlet. `name`, `version` and `pathname` are required parameters.

(2) Creating an express instance and attaching our podlet middleware to it.

(3) Defining the route for the `manifest.json`.

(4) Implementing the route for the actual content. `podiumSend` is comparable to express' normal `send` function but adds an extra version header to the response. It also comes with a few features that make local development easier.

You have to define the podlet information, a route for the `manifest.json`, and the `/recos` route that produces the actual content. In our case, we use express' standard `app.get` method for that.

FALLBACKS AND TIMEOUTS

The way Podium handles fallbacks is quite interesting. With the Nginx approach, we hat do define the fallback in the template of the page. With ESI and Tailor, the page owner can provide a second URL that's tried when the actual URL does not work. In Podium it's a little bit different:

- the team owning the fragment provides the fallback
- the team including the fragment caches this fallback locally

These two properties make it much easier to create a meaningful fallback. *Team Inspire* could, for example, define a list of "evergreen recommendations" that look similar to dynamic recommendations. *Team Decide* caches it and can show it even if *Team Inspire's* server does not respond at all. Figure 4.10 shows how the fallback mechanism works.

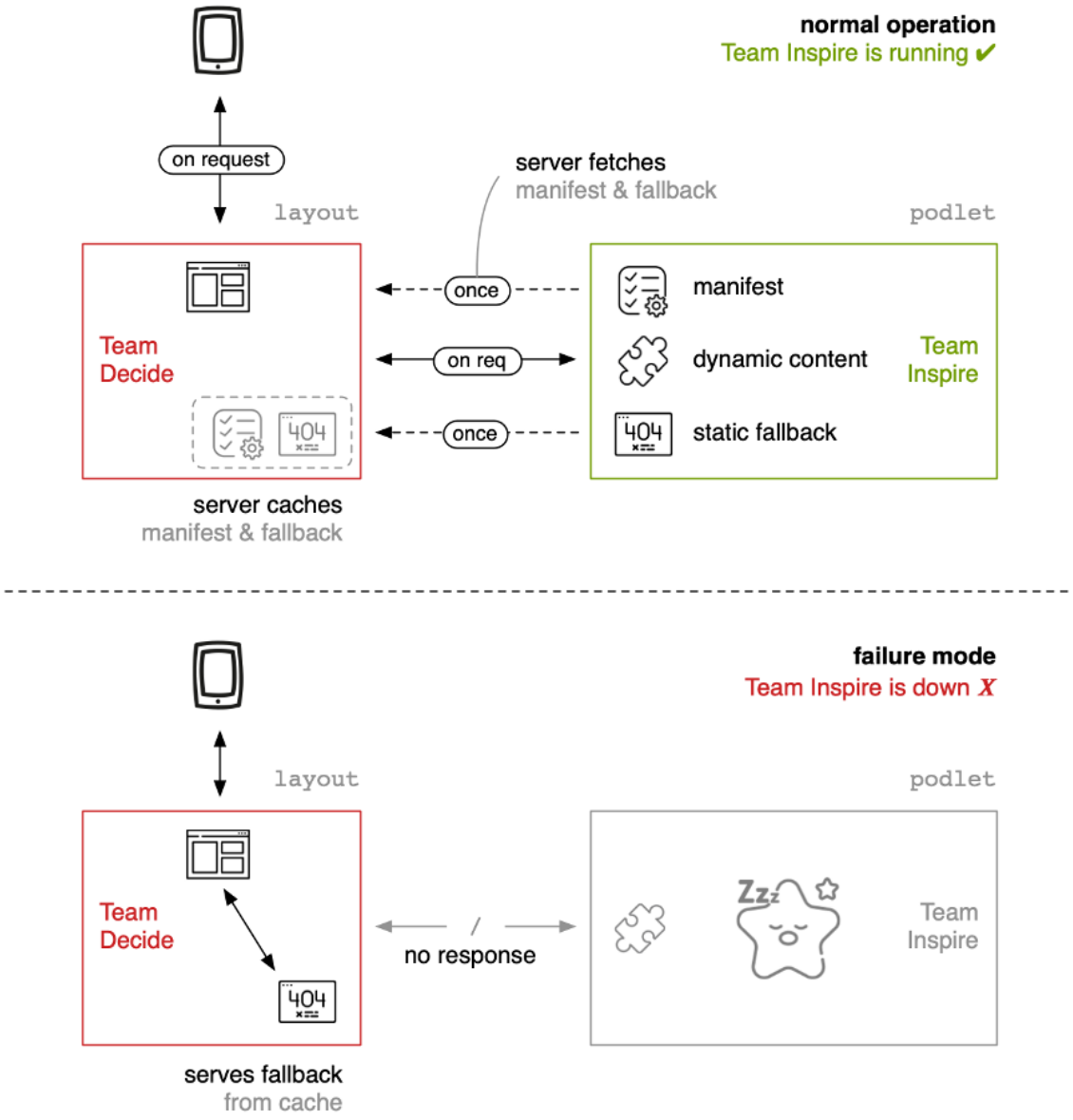


Figure 4.10 Podiums fallback handling. The podlet owner can specify the fallback content in the manifest. The layout service retrieves the fallback content once and caches it. When the podlet server goes down, the fallback is used instead of the dynamic content.

The code for specifying the fallback in the podlet server looks like this:

Listing 4.15 team-inspire/index.js

```
...
const podlet = new Podlet({
  ...
  pathname: "/recos",
  fallback: "/fallback",                          ❶
});
...
app.get("/recos/fallback", (req, res) => {        ❷
  res.status(200).podiumSend(`                    ❷
    <a href="http://localhost:3002/recos">        ❷
      Show Recommendations                        ❷
    </a>                                          ❷
  `);                                             ❷
});                                               ❷
...
```

❶ Adding the `fallback` property to the podlets configuration.

❷ Implementing the request handler for the fallback. This route is called once by the layout service. The response is then cached.

You have to add the URL to the Podlet constructor and implement the matching route `/recos/fallback` in the application.

The idea of having a `manifest.json` that describes everything you need to know for the integration of a fragment is pretty handy. The format is simple and straight forward. Even if you decide to stop using the stock `@podium/*` libraries or want to implement a server in a non-JavaScript language, you can still do it. As long as you can produce/consume manifest endpoints.

Podium also includes some other concepts like a development environment for podlets and versioning. If you want to get deeper into Podium, the official documentation.[45] is an excellent place to start.

4.4.4 Which solution is right for me?

As you might have guessed, there is no universal answer or silver bullet when it comes to choosing your composition technique. Tools like Tailor and Podium implement fragments as a first-party concept, which makes everyday tasks like fallbacks, timeouts, and asset handling much more comfortable. Teams include the composition mechanism directly into their application. There's no need for an extra piece of infrastructure. This approach is especially useful for local development since you don't need to set up a separate web-server on every developer machine to make fragments work. Figure 4.11 illustrates this. But these solutions also come with a non-trivial amount of code and internal complexity.

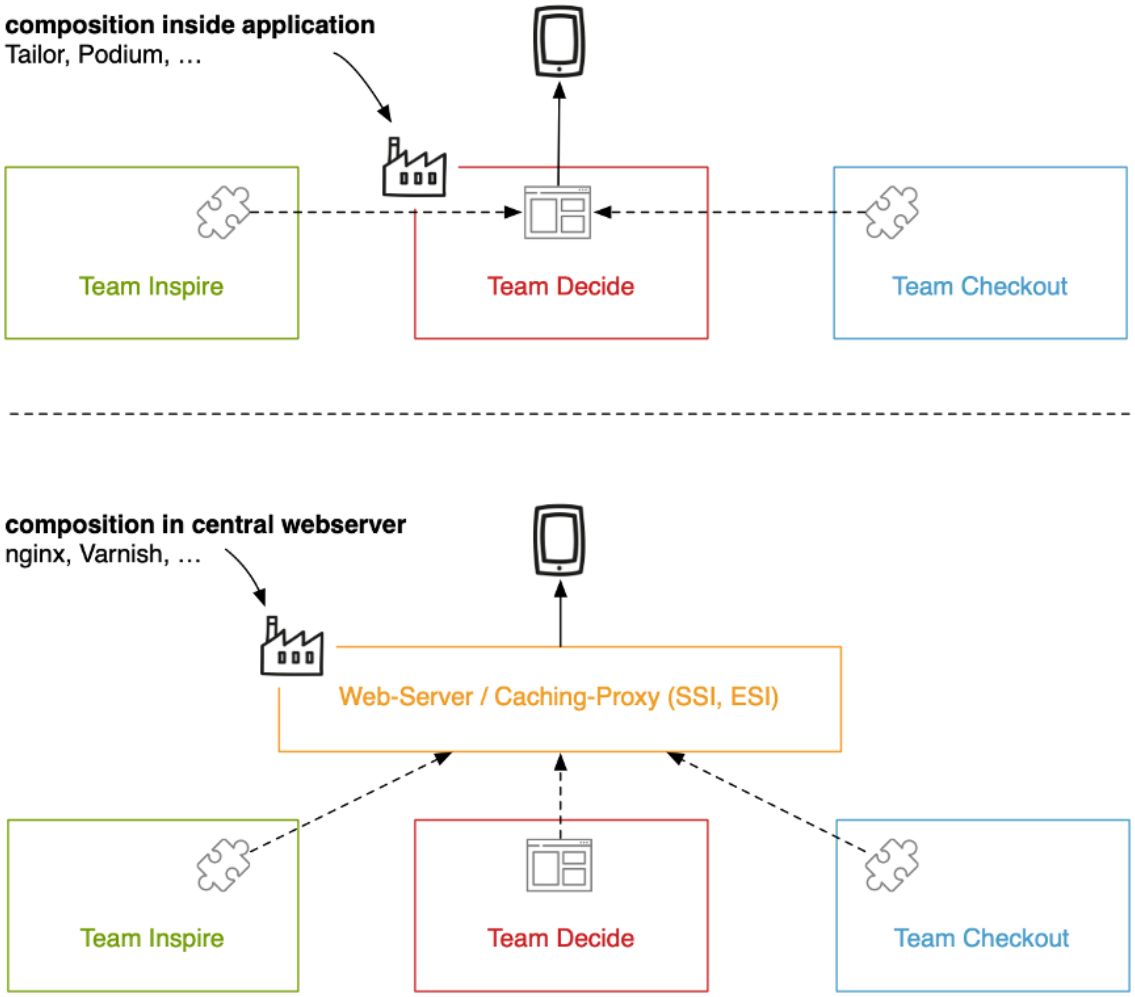


Figure 4.11 Fragment composition in the application or in a central web-server.

Techniques like SSI and ESI are old, and there is no real innovation happening. But these downsides are also their biggest strengths. Having an integration solution that is very stable, boring, and easy to understand can be a huge benefit.

Picking a composition solution is a long term decision. All teams will rely on the chosen software to do their work.

4.5 The good and bad of server-side composition

Now you know the essential aspects of server-side composition. Let's look at the advantages and disadvantages of this approach.

4.5.1 The benefits

We can achieve **excellent first load performance** since the browser receives an already assembled page. Network latency is much lower inside a data center. This way, it's also possible to integrate a lot of fragments without putting extra stress on the customer's device.

This model is a sound basis for building a micro frontends style application that embraces **progressive enhancement**. You can add interactive functionality via client-side JavaScript on top.

SSI and ESI are **proven and well-tested technologies**. They are not always convenient to configure. But when you have a working system, it runs fast and reliable without needing much maintenance.

Having the markup generated on the server is **good for search engines**. Nowadays, all major crawlers also execute JavaScript - at least in a fundamental way. But having a site that loads fast and does not require a considerable amount of client-side code to render still helps to get a good search engine ranking.

4.5.2 The drawbacks

If you are building a large, fully server-rendered page, you might get a **non-optimal time to first byte**, and the browser spends a lot of time **downloading markup instead of loading necessary assets** like styles and images for the viewport. But this is also true for server-rendered pages in a non-micro frontends architecture. Use server-side integration where it makes sense and combine it with client-side integration when needed.

As with the AJAX approach, server-side integration **does not come with technical isolation in the browser**. You have to rely on CSS class prefixes and namespacing to avoid collisions.

Depending on your choice of integration technique **local development becomes more complicated**. To test the integrated site, each developer needs to have a web-server with SSI or ESI support running on their machine. Node.js based solutions like Podium or Tailor ease this pain a bit because they make it possible to move the integration mechanism into your frontend application.

If you want to **build an interactive application** that can quickly react to user input, **a pure server-side solution does not cut it**. You need to combine it with a client-side integration approach like AJAX or web components.

4.5.3 When does server-side integration make sense?

If good loading performance and search engine ranking are a high priority for your project, there is no way around server-side integration. Even if you are building an internal application that does not require a high amount of interactivity, a server-side integration might be a good fit. It makes it easy to create a robust site that still functions even if client-side JavaScript fails.

If your project requires an app-like user interface that can instantly react to user input, server-side integration is not for you. A pure client-side solution might be easier to implement.

But you can also go the hybrid way and build a universal/isomorphic application using both server- and client-side composition. In chapter Composition and Universal Rendering you'll learn how to do it.

Figure 3.3 shows the comparison chart introduced in the last chapter. We've added server-side integration.

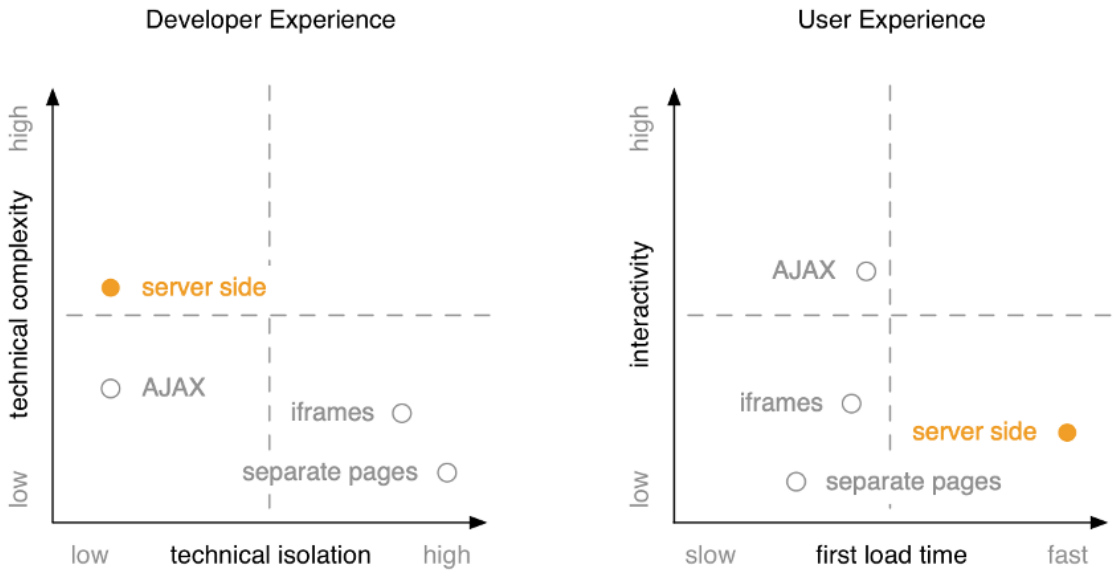


Figure 4.12 Server side integration in comparison to other integration techniques. They introduce extra infrastructure, which increases complexity. Similar to the AJAX approach, they don't introduce technical isolation. You still have to rely on manual namespacing. But they enable you to achieve good page load times.

4.6 Summary

- Integrating markup on the server usually leads to better page load performance because latency inside the datacenter is much shorter than to the client.
- You should have a plan for what happens when an application server goes down. Fallback content and timeouts help.
- Nginx loads all SSI includes in parallel, but only start sending data to the client when the last fragment arrived.
- Library based integration solutions like Tailor and Podium directly integrate into a team's application. Thereby less infrastructure is required, and local development is more comfortable. But they also are a non-trivial dependency.
- The integration solution is a central piece in your architecture. It's good to pick a solution that is solid and easy to maintain.
- Server-side composition is the basis for building a micro frontends style site that uses progressive enhancement principals.

5 Client-side Composition

This chapter covers

- Examining Web Components as a client-side composition technique
- Investigating how to use micro frontends, built with different frameworks, on the same page
- Exploring how Shadow DOM can help to safely introduce a micro frontend into a legacy system without having style conflicts

In the last chapter, you learned about different server-side integration techniques like SSI or Podium. These techniques are indispensable for websites that need to load fast. But for many applications, the first load time is not the only important thing. Users expect websites to feel snappy and react to their input promptly. No one wants to wait for the complete page to reload just because she changed an option in a product configuration. People spend more time on sites that react fast and feel app-like. Due to this fact, client-side rendering with frameworks like React, Vue.js, or Angular has gotten popular. With this model, the HTML markup gets produced and updated directly in the browser. Server-side integration techniques don't provide an answer to this.

In a traditional architecture, we would have built a monolithic frontend that's tied to one framework in one specific version. But in a micro frontends architecture, we want the user interfaces from the different teams to be self-contained and independently upgradable. **We can't rely on the component system of one specific framework.** This constraint would tie the complete architecture to a central release cycle. A framework change would result in a parallel rewrite of the complete frontend. The Web Components spec introduced a neutral and standardized component model. In this chapter, you'll learn how Web Components can act as a

technology-agnostic glue between different micro frontends. They make it possible for independent frontend applications to coexist on one page, even if their technology stack is not the same. Figure 5.1 illustrates this client side integration.

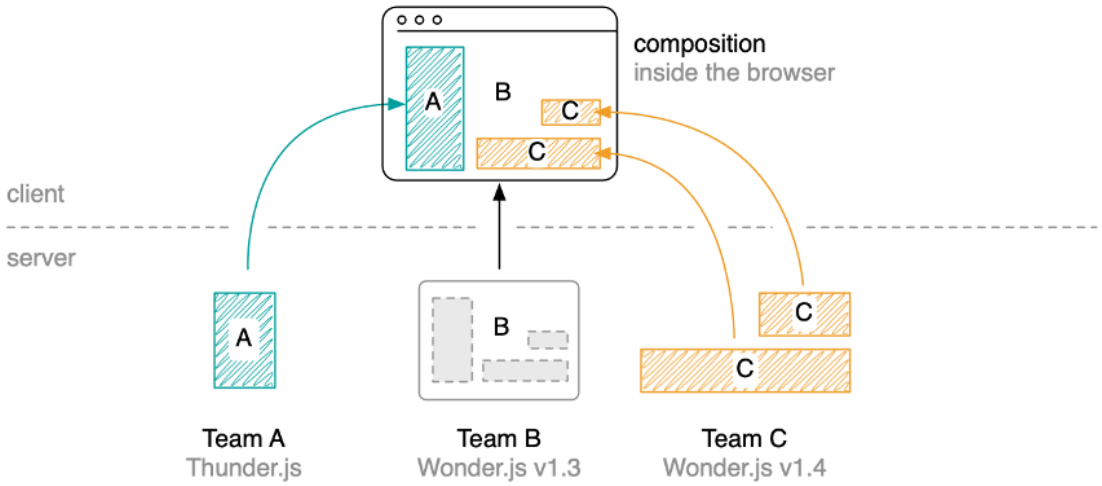


Figure 5.1 Micro frontend composition in the browser. Each fragment is it's own mini-application and can render and update it's markup independently from the rest of the page. Thunder.js and Wonder.js are placeholders for your frontend framework of choice.

5.1 Wrapping Micro Frontends using Web Components

Over the last weeks, *Tractor Models Inc.* has made an enormous splash in the tractor model community. Production is ramping up, and they were able to send out first review units. Positive press coverage and unboxing videos from YouTube celebrities lead to an enormous increase in visitor numbers.

But the online-shop still misses it's most important feature, "the buy button". Up until now, customers are only able to see the tractors, and it's recommendations. A few sprints ago, the company staffed a third team: *Team Checkout*. It has been working hard to set up the infrastructure and write the software for handling payments, storing customer data, and talking to the logistics system. Their pages for the checkout flow are ready. The last piece that's missing is the ability to add a product to the basket from the product page.

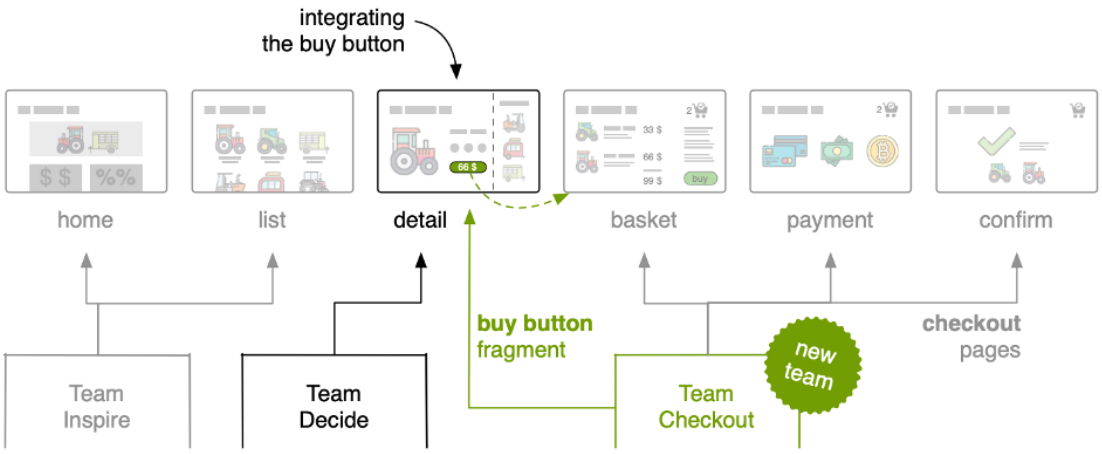


Figure 5.2 Team Checkout owns the complete checkout flow. Team Decide does not have to know about how the checkout works. But they need to integrate Team Checkouts "buy button" fragment in the detail page to make it work. Team Checkout provides this button as a standalone micro frontend.

Team Checkout chose to go with client-side rendering for their user interfaces. They've implemented the checkout pages as a single page app (SPA). The buy-button fragment is available as a standalone Web Component. Let's see what this means and how we can integrate the fragment in the product detail page. For the integration, *Team Checkout* provides *Team Decide* with the necessary information. This is the **contract between both teams**:

- **Buy Button** tag-name: `checkout-buy` attributes: `sku=[sku]` example: `<checkout-buy sku="porsche"></checkout-buy>`

Team Checkout delivers the actual code and styles for the `checkout-buy` component via a JS/CSS file. Their application runs on port `3003`.

- **required JS & CSS assets references**
  `http://localhost:3003/static/fragment.js`
  `http://localhost:3003/static/fragment.css`

Team Decide and *Team Checkout* are free to change the layout, look, or behavior of their user interfaces as long as they adhere to this contract.

5.1.1 How to do it

Team Decide has everything it needs to add the buy button to the product page. They don't have to care about the internal workings of the button. They can place `<checkout-buy sku="porsche"></checkout-buy>` somewhere in their markup, and a functional buy button will magically appear. *Team Checkout* is free to change its implementation in the future without having to coordinate with *Team Decide*. Before we go into the code, let's look at what the term Web Components means. If you're are already familiar with Web Components, you can skip the next two sections and continue with *Encapsulating business logic through DOM elements*.

WEB COMPONENTS & CUSTOM ELEMENTS

The web component spec has been long in the making. Its goal is to introduce better encapsulation and enable interoperability between different libraries or frameworks. At the time of writing this book, all major browsers have implemented v1 of the specification. It's also possible to retrofit the implementation into older browsers using a polyfill.[46].

Web Components is an umbrella term. It describes three distinct new APIs: Custom Elements, Shadow DOM, and HTML Templates.

Let's focus on Custom Elements They make it possible to provide functionality in a declarative way through the DOM. You can interact with Custom Elements the same way you would interact with standard HTML elements.

Let's look at a typical `button` element. It has multiple features builtin. You can set the text shown on the button: `<button>hello</button>`. It's also possible to switch the button into an inactive mode by setting the `disabled` attribute: `<button disabled>…</button>`. By doing this, the button is dimmed out and does not respond to click events anymore. As a developer, you don't have to understand what the browser does internally to achieve this behavior.

Custom Elements enable developers to create similar abstractions. You can **construct new generic representational elements** that are missing from the HTML spec. GitHub has published a list of such controls.[47]. Look at this "copy-to-clipboard" element.

```
<clipboard-copy value="/repo-url">Copy</clipboard-copy>
```

It encapsulates the browser-specific code and provides a declarative interface. A user of this component just needs to include GitHub's JavaScript definition for this component into his site. We can use this mechanism to create abstractions for our micro frontends.

WEB COMPONENT AS A CONTAINER FORMAT

You can also use Web Components to **encapsulate business logic**. Let's go back to our example at *The Tractor Store*. *Team Checkout* owns the domain knowledge around product prices, inventory, and availabilities. *Team Decide*, owner of the product page, doesn't have to know these concepts. Their job is to provide the customer with all product information he needs to make a good buying decision. The business logic needed for the product page is encapsulated in the `checkout-buy` component, as shown in figure 5.3.

```
<checkout-buy sku="porsche"></checkout-buy>
```

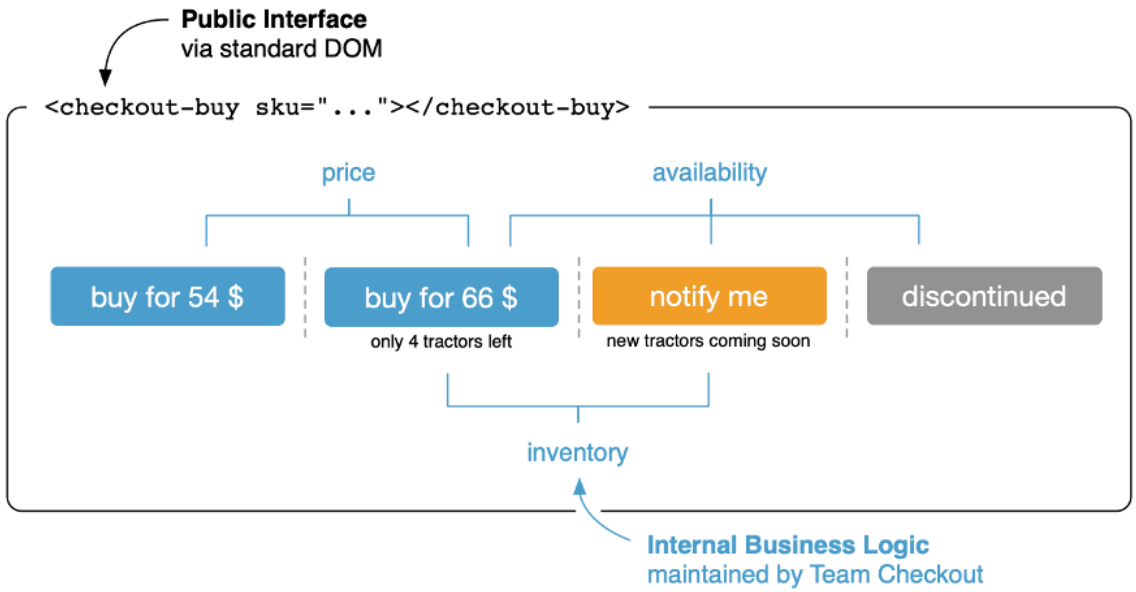


Figure 5.3 A Custom Element can encapsulate business logic and provide the associated user interface. The buy button can look differently depending on the specified SKU but also due to internal pricing and inventory information. A team that uses this fragment does not have to know these concepts.

DEFINING A CUSTOM ELEMENT

Let's look at the implementation of the buy button.

Listing 5.1 team-checkout/static/fragment.js

```
class CheckoutBuy extends HTMLElement {                    ❶
  connectedCallback() {                                    ❶
    this.innerHTML = "<button>buy now</button>";           ❷
  }                                                        ❸
}
window.customElements.define("checkout-buy", CheckoutBuy); ❸
```

❶ Defines an ES6 class for the Custom Element.

❷ This function gets called for every buy-button found in the markup and renders a simple button element.

❸ Registers the Custom Element under the name `checkout-buy`.

The above code shows a minimal example of a Custom Element. We have to use an ES6 class for the Custom Elements implementation. This class gets registered via the globally available `window.customElements.define` function. Every time the browser comes across a `checkout-buy` element in the markup, a new instance of this class gets created. The `this` of the class instance is a reference to the corresponding HTML element.

> **NOTE** The `customElements.define` call does not need to come before the browser has parsed the markup. Existing elements are *upgraded* to Custom Elements as soon as the definition is registered.

You can choose any name you want for your Custom Element. The only requirement specified in the spec is that it has to contain at least one hyphen (`-`). This way, you won't run into future issues when the HTML specification adds new elements.

In our projects we've used the pattern `[team]-[fragment]` (example: `checkout-buy`). This way, you've established a namespace, avoid inter-team naming collisions, and ownership attribution is easy.

USING A CUSTOM ELEMENT

Let's add the component to our product page. The markup for the product page now looks like this.

Listing 5.2 team-decide/product/porsche.html

```
...
<link                                                    ❶
  href="http://localhost:3003/static/fragment.css"       ❷
  rel="stylesheet" />                                    ❸
...
<div class="decide_details">
  <checkout-buy sku="porsche"></checkout-buy>            ❹
</div>
...
<script                                                  ❸
  src="http://localhost:3003/static/fragment.js" async>  ❸
</script>                                                ❸
```

❶ Including fragment styles.

❷ Placing the buy button.

❸ Including fragment scripts.

Keep in mind that Custom Elements can not be self-closing. They always need a dedicated closing tag like `</checkout-buy>`. Since the fragment is fully client-rendered, *Team Checkout* only needs to host two files: `fragment.css` and `fragment.js`. *Team Inspire* has reworked their recommendations micro frontend to work the same way. See the updated folder structure in figure 5.4.

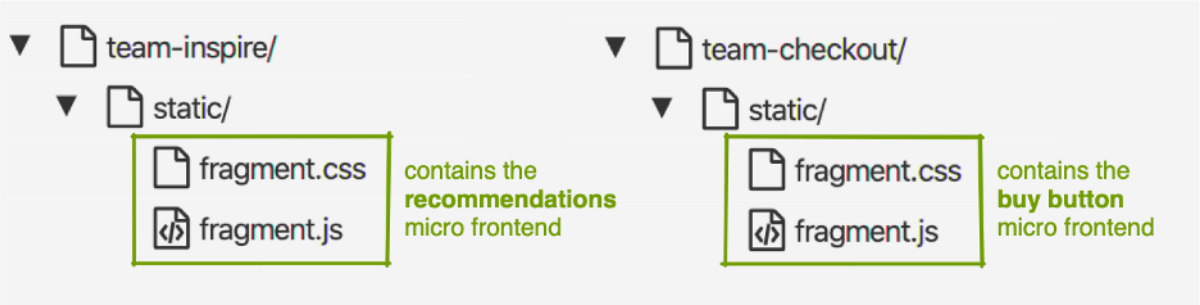


Figure 5.4 Both teams expose their micro frontends via a CSS and JavaScript file that Team Decide can reference.

Start the applications from all three teams by running this command.

```
npm run 08_web_components
```

Opening localhost:3001/product/porsch e shows you the product page with the client-side rendered buy button like in figure 5.5.

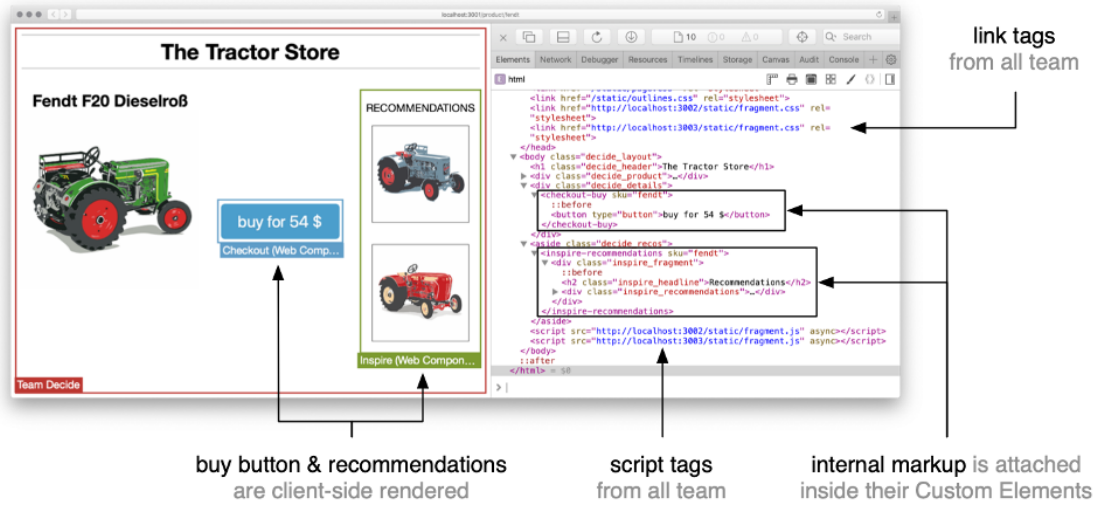


Figure 5.5 The Custom Element renders itself in the browser via JavaScript. It generates it's internal markup and attaches it as children to the tree via this.innerHTML = "...".

PARAMETRIZATION VIA ATTRIBUTES

Let's make the buy button component a little bit more useful. It should also display the price and provide the user with a simple feedback dialog after he has clicked. The following example shows different prices depending on the specified SKU attribute.

Listing 5.3 team-checkout/static/fragment.js

```
const prices = { porsche: 66, fendt: 54, eicher: 58 };      ❶

class CheckoutBuy extends HTMLElement {
  connectedCallback() {
    const sku = this.getAttribute("sku");                   ❶
    this.innerHTML = `
      <button type="button">
        buy for ${prices[sku]} $                            ❶
      </button>
    `;
  }
}
```

❶ List of tractor prices.

❷ Reading the SKU from the Custom Elements attribute.

❸ Looking up and rendering the price on the button.

For simplicity, we define the prices inside the JavaScript code. In a real application, you would probably fetch them from an API endpoint, which is owned by the same team.

Adding user feedback to the button is also straight forward. We attach a standard event listener that reacts to click events and shows a success message as an alert.

Listing 5.4 team-checkout/static/fragment.js

```
this.innerHTML = "...";
this.querySelector("button")            ❶
  .addEventListener("click", () => {    ❷
    alert("Thank you ❤");               ❸
  });
```

❶ Getting the reference to the button.

❷ Add a click handler.

❸ Display a success message on click.

Again, this is a simplified implementation. In real life, you'd probably persist the cart change to the server by calling an API. Depending on that API's response, you would show a success or error message. You get the gist.

5.1.2 Wrapping your framework in a Web Component

In our examples we use standard DOM API like `innerHTML` and `addEventListener`. In a real application, you would probably use higher-level libraries or frameworks instead. They often make developing more comfortable and come with features like DOM diffing or declarative event handling. The Custom Element (`this`) acts as the root of your mini-application. This application has its state and doesn't need other parts of the page to function.

Custom Elements introduce a set of lifecycle methods like `constructor`, `connectedCallback`, `disconnectedCallback` and `attributeChangedCallback`. When you implement them, you get notified when someone added your micro frontend to the DOM, removed it, or changed one of its attributes. It's straight forward to connect these lifecycle methods to the (de-)initialization code of the framework or library you are working with. Figure 5.6 illustrates this. The component hides the implementation details of the specific framework. This way, its owner can change the implementation without changing its signature. **The Custom Element acts as a technology-neutral interface.**

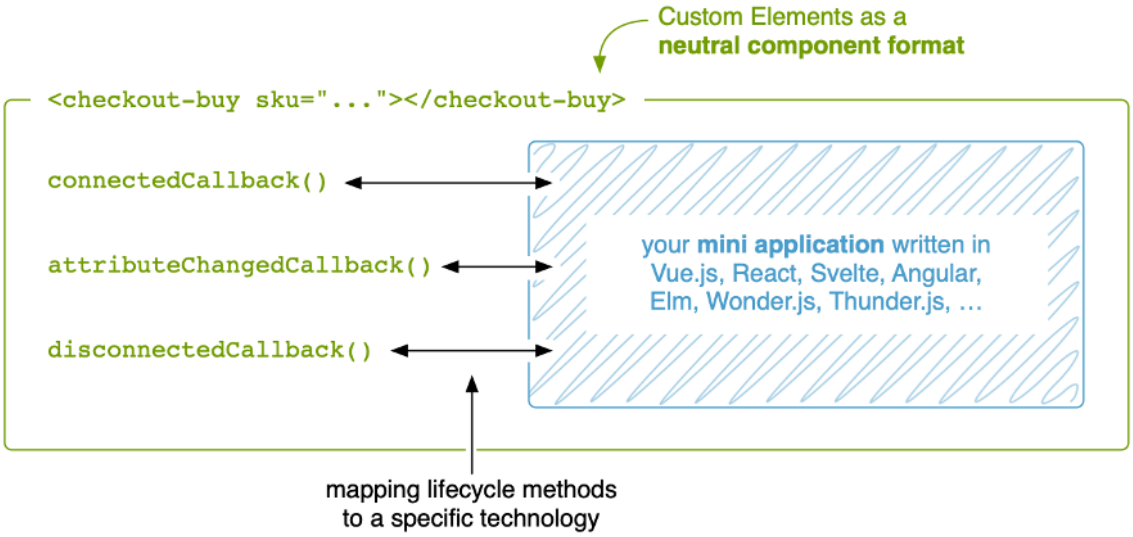


Figure 5.6 Custom Elements introduce lifecycle methods. You need to map these to the specific technology of your micro frontend.

Some newer frameworks like Stencil.js.[48] already use Web Components as their primary way to export an application. Angular comes with a feature called Angular Elements.[49]. This feature will automatically generate the code necessary to connect the app with the Custom Elements' lifecycle methods and also supports Shadow DOM. Vue.js provides a similar solution via the official `@vue/web-component-wrapper` package.[50] Since Web Components are a web standard, there are comparable libraries or tutorials for all popular frameworks out there.

The example code of this chapter is deliberately kept simple and doesn't include a frontend

framework. You can checkout the examples `20_shared_vendor_rollup_absolute_imports` from chapter 11. Performance is Key to see a React application wrapped in a Custom Element.

5.2 Style isolation using Shadow DOM

Another part of the Web Components spec is Shadow DOM. With Shadow DOM, it's possible to isolate a subtree of the DOM from the rest of the page. We can use it to eliminate the chance of leaked styles and thereby increases robustness for our micro frontends application.

Currently, *Team Checkout's* `fragment.css` file is included globally in the head. All styles in this file have the potential to affect the complete page. Teams have to adhere to CSS namespacing rules to avoid conflicts. The concept of Shadow DOM provides an alternative where no prefixing or explicit scoping is required.

5.2.1 Creating a Shadow Root

You can create an isolated DOM sub-tree via JavaScript by calling `.attachShadow()` on an HTML element. Most people use Shadow DOM in combination with a Custom Element, but it doesn't have to. You can also attach a Shadow DOM to many standard HTML elements like a `div`.[51].

Here is an example of how to create and use Shadow DOM:

```
class CheckoutBuy extends HTMLElement {
  connectedCallback() {
    const sku = this.getAttribute("sku");
    this.attachShadow({ mode: "open" });      ❶
    this.shadowRoot.innerHTML = "buy ..."     ❶
  }
}
```

❶ Creating an "open" shadow tree.

❷ Writing content to the newly created `shadowRoot`.

`attacheShadow` initializes the Shadow DOM and returns a reference to it. The reference to an open Shadow DOM is also accessible through the `shadowRoot` property of the element. You can work with it like any other DOM element.

SIDEBAR **open vs. closed**

You can choose between an `open` and `closed` mode when creating a Shadow DOM. A `mode: "closed"` hides the `shadowRoot` from the outside DOM. This guards against unwanted DOM manipulation via other scripts. But it also prevents assistive technologies and crawlers from seeing your content. Unless you have special needs, it's recommended to stick to the `open` mode.

5.2.2 Scoping styles

Let's move the styling from the `fragment.css` into the actual component. We do this by defining a `<style>…</style>` block inside the Shadow Root. Styles that are defined in the Shadow DOM stay in the Shadow DOM. Nothing leaks out and can affect other parts of the page. It also works the other way around. CSS definitions from the outside document don't work inside the Shadow DOM.[52].

Look at the code for the "buy button" fragment below:

Listing 5.5 team-checkout/static/fragment.js

```
...
class CheckoutBuy extends HTMLElement {
  connectedCallback() {
    const sku = this.getAttribute("sku");
    this.attachShadow({ mode: "open" });      ❶
    this.shadowRoot.innerHTML = `             ❶
      <style>                                 ❶
        button {}                             ❷
        button:hover {}                       ❷
      </style>                                ❷
      <button type="button">
        buy for ${prices[sku]} $
      </button>
    `;
    ...
  }
  ...
}
...
```

❶ Creating a Shadow DOM for the `buy-button` element.

❷ Writing content to the `shadowRoot` instead of directly attaching it to the `buy-button`.

❸ Defining styles as inline CSS. They only apply inside the shadowRoot.

Run the following command to play with this code in the browser:

```
npm run 09_shadow_dom
```

You should see the familiar product page. Have a look at the DOM structure with your browser's developer tools. In figure 5.7, you can see that each micro frontend now renders its internal markup and styles inside its shadow root.

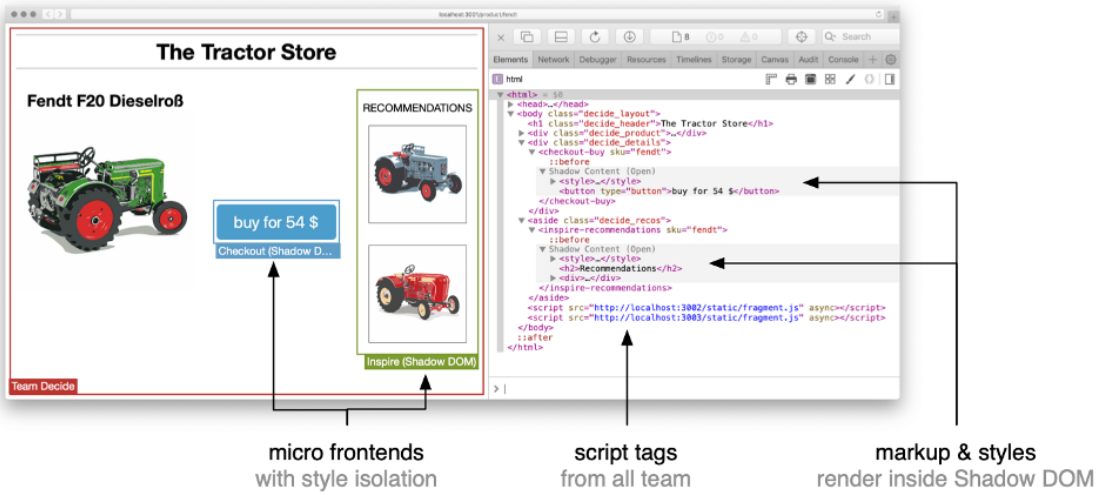


Figure 5.7 Micro frontends can render their internal markup and styles inside a shadow root. This improves isolation and reduces the risk of conflicting or leaking styles.

We've eliminated the risk of style collisions. Figure 5.8 illustrates the effect of the virtual border the `shadowRoot` introduces. This border is called **shadow boundary**.

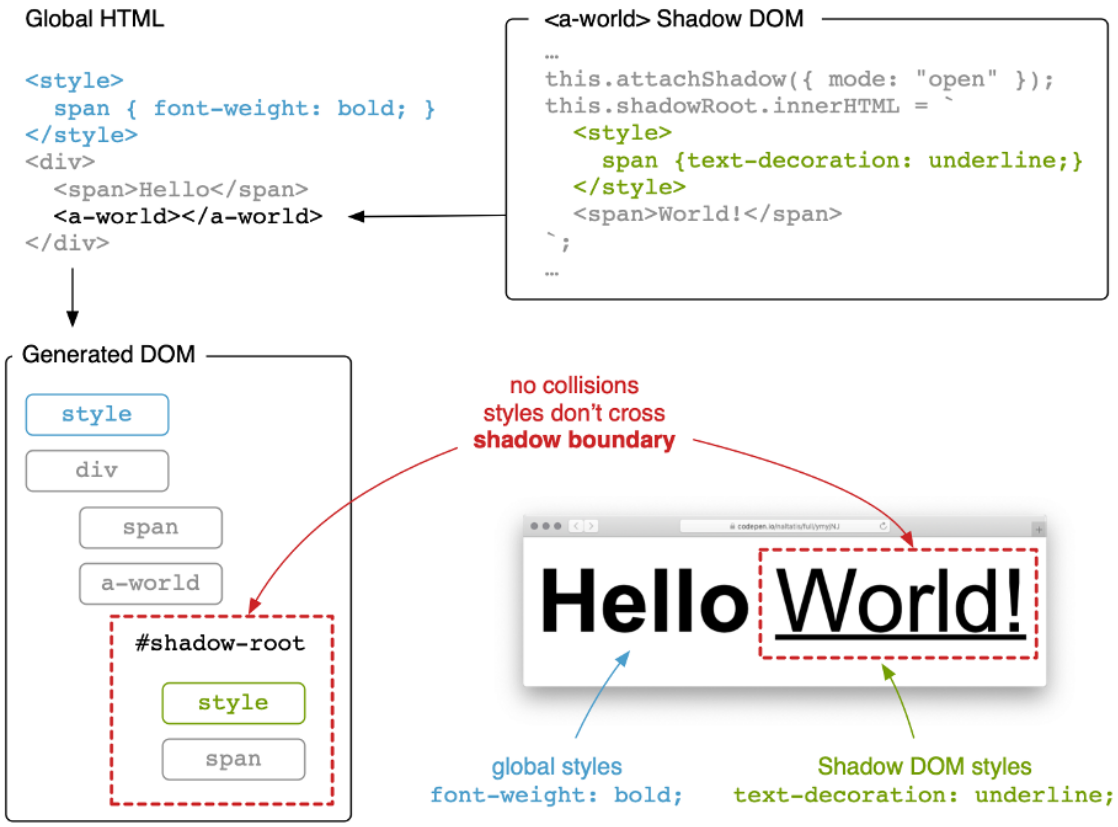


Figure 5.8 The Shadow Root creates a border called shadow boundary. It provides isolation in both ways. Styles don't leak out of the component. Styles on the page also don't affect the Shadow DOM.

If you've used CSS Modules or any other CSS-in-JS solution, this way of writing CSS should feel familiar. These tools let you write CSS code without having to worry about scope. They automatically scope your code by generating unique selectors or inline styles. Shadow DOM makes it possible to have **guaranteed style isolation between the micro frontends** of different teams. No conventions or extra toolchain required.

5.2.3 When to use Shadow DOM

There are a lot of details you can learn about Shadow DOM.[53]. Events behave differently when they bubble from the Shadow DOM into the regular DOM (also called Light DOM). But since this is a book about micro frontends and not Web Components, we won't go deeper into this topic. Here is a list of pros and cons for using Shadow DOM in a micro frontends context.

- **Pros**
  - Strong **iframe-like isolation**. No namespacing required.
  - Prevents global styles from leaking into a micro frontend. **Great when working with legacy applications.**
  - Potential to reduce the need for CSS toolchains.

- Fragments are self-contained. No separate CSS file references.

- **Cons**
  - Not supported in older browsers. Polyfills exist but are heavy and rely on heuristics.
  - **Requires JavaScript** to work.
  - **No progressive enhancement** or server rendering. Shadow DOM can't be defined declaratively via HTML.
  - Hard to share common styles between different Shadow DOMs. Theming is possible via CSS properties.
  - Does not work with styling approaches that use on global CSS classes - like Twitter Bootstrap.

5.3 The good and bad of using Web Components for composition

Using Web Components for client-side integration is one of many options. There are meta-frameworks or custom implementations to achieve a similar result. Let's discuss the strength and weaknesses of this approach.

5.3.1 The benefits

The **most significant benefit of using Web Components** as an integration technique is that **they are a widely implemented web standard**. It's often not very convenient to work with browser APIs directly. But abstractions exist that make developing easier. **Web standards evolve slowly and always in a non-breaking, backward compatible way.** That's why they are an excellent fit for a common basis.

Custom Elements and Shadow DOM both **provide extra isolation features** that were not possible to achieve before. This isolation makes your micro frontends applications more robust. It's not required to use both techniques together. You can pick and choose depending on your project's needs.

The **lifecycle methods** introduced by Custom Elements make it possible to **wrap the code of different applications in a standard way**. These applications **can then be used declaratively**. Without this standard, teams would have to agree upon home-grown initialization, deinitialization, and updating schemas.

5.3.2 The drawbacks

One of Web Components' most prominent points of criticism is that **they require client-side JavaScript** to function. You might say that this is also true for most web frameworks these days. But all major frameworks provide a way to server render the content. Not being able to server-render is **an issue when you need a fast first-page load** and want to develop by the principals of **progressive enhancement**. There are some proprietary ways to declaratively render Shadow DOM from the server and hydrate it on the client, but there is no standard.

Browser support for Web Components has dramatically improved over the last years. It's easy to add Custom Elements support to older browsers. **Polyfilling Shadow DOM is more tricky.** If you are targeting newer browsers, it's not an issue. But if your application also needs to run on older browsers that don't support Shadow DOM, you might consider going with an alternative like manual namespacing.

5.3.3 When does client-side integration make sense?

If you are building **an interactive, app-like application** where **user interfaces from different teams** must be integrated **on one screen** Web Components are a reliable basis.

The interesting question is what interactive means. We'll discuss this topic in 9.3. Are you building a site or an app?. For simpler use-cases like the catalog or a content-heavy site, a server-rendered approach that uses SSI or AJAX often works fine and is easier to handle.

Using Web Components does not mean that you have to go all-in on client-side rendering. We've successfully used Custom Elements as the contract between different teams. These Custom Elements implemented AJAX-based updating - fetching generated markup from the server. When a specific use-case required more interactivity, a team could switch from AJAX to a more sophisticated client-side rendering for this fragment. Since the Custom Element acts as the point of communication, other teams didn't care about the inner workings of this fragment.

It's also possible to **combine Custom Elements** (not Shadow DOM) **with a server-side integration technique**. We'll explore this in chapter Composition and Universal Rendering.

If your use-case requires you to build **a full client rendered application**, you should consider using **Web Components as the neutral glue** between team UIs.

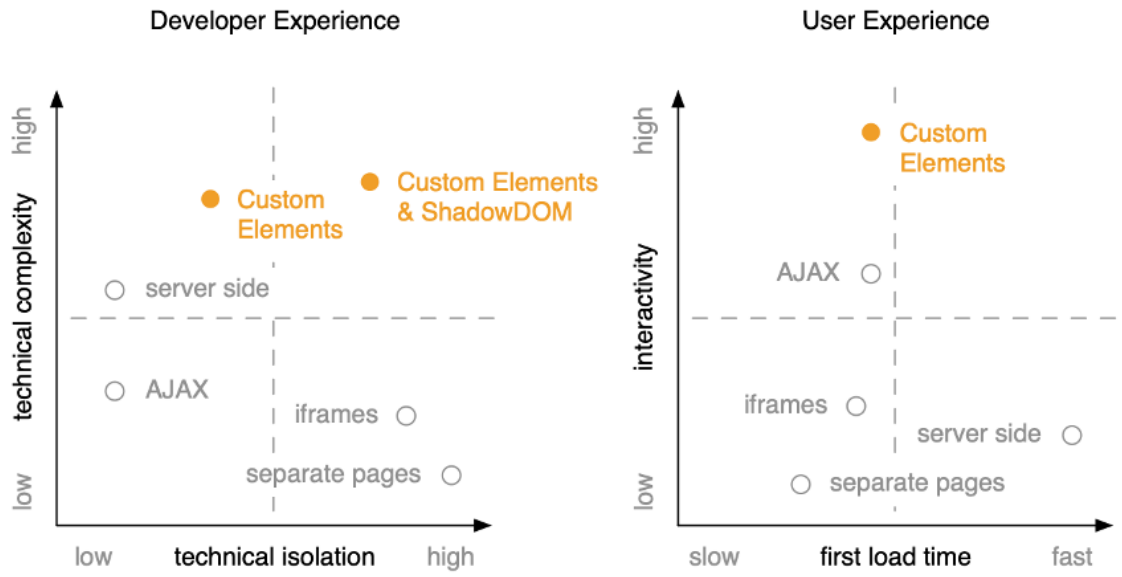


Figure 5.9 Custom Elements provide a good way to encapsulate your JavaScript application and make it accessible in a standard way. Shadow DOM introduces an extra isolation mechanism and lowers the risk of conflicts. You can build highly interactive, client-side rendered applications using Custom Elements. But since they require JavaScript to function, a server-rendered solution will usually be quicker on the first load.

5.4 Summary

- You can encapsulate a micro frontend application in a Web Component. Other teams can interact with it declaratively by using the browsers DOM API. The Web Component encapsulates business logic and implementation details.
- Most modern JavaScript frameworks have a canonical way to export an application as a Web Component. This makes creating a client-side micro frontend easier.
- Shadow DOM introduces strong, iframe-like isolation for CSS styles. This reduces the risk of conflicts between different team UIs.
- Shadow DOM does not only prevent styles from leaking out. They also guard against global styles leaking in. This styling boundary makes them an excellent fit for integrating a micro frontend into a legacy application.

6 Communication Patterns

This chapter covers

- Examining user interface communication patterns to exchange events between micro frontends.
- Inspecting ways to manage state and discussing the issues of shared state.
- Illustrating how to organize server communication and data fetching in a micro frontends architecture.

Sometimes user interface fragments owned by different teams need to talk to each other. When a user adds an item to the basket by clicking the buy button, other micro frontends like e.g., the mini basket wants to be notified to update their content accordingly. We'll have a more in-depth look at this topic in the first part of this chapter. But there are also other forms of communication going on in a micro frontends architecture, as you can see in figure 6.1.

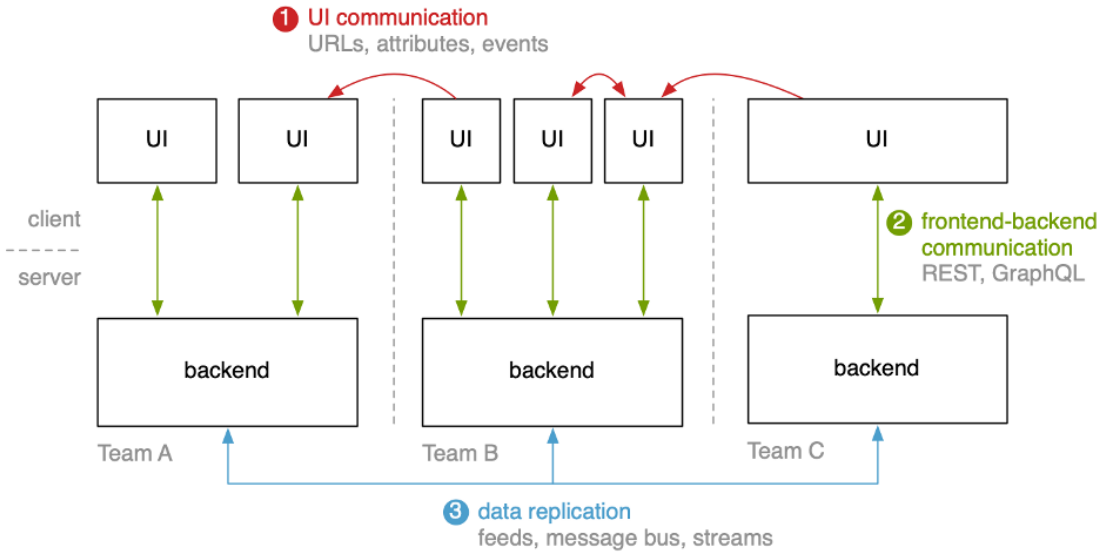


Figure 6.1 An overview of different communication mechanisms in a typical micro frontends architecture. The frontend applications in the browser need a way to talk to each other. We call this UI communication (1). Each frontend fetches data from its own backend (2), and in some cases, it's required to replicate data between the backends of the teams (3).

In the second part of this chapter, we'll explore how these types of communications play together. We'll discuss how to manage state, distribute necessary context information, and replicate data between the team's backends.

6.1 User interface communication

How can UIs from different teams talk to each other? If you've chosen good team boundaries, you'll learn more about how to do it in 13. Teams & Boundaries, there should be little need for extensive cross UI communication in the browser. To accomplish a task, a customer is ideally only in contact with the user interface from one team.

In our e-commerce example, the process the customer goes through is pretty linear. Finding a product, deciding whether to buy it, and doing the actual checkout. We've aligned our teams along these stages. *Some inter-team communication might be required at the handover points when a customer goes from one team to the next.*

This communication can be simple. We've already used **page-to-page communication** in 2. My First Micro Frontends Project. Moving from the product page to another team's recommendation page via a simple link. In our case, we transferred the product reference, the SKU, via the URL path or the query string. *In most cases cross-team communication happens via the URL.*

If you are building a richer user interface that combines multiple use cases on one page, a link isn't sufficient anymore. You need a standard way for the different UI parts to talk to each other.

Figure 6.2 illustrates three common communication patterns.

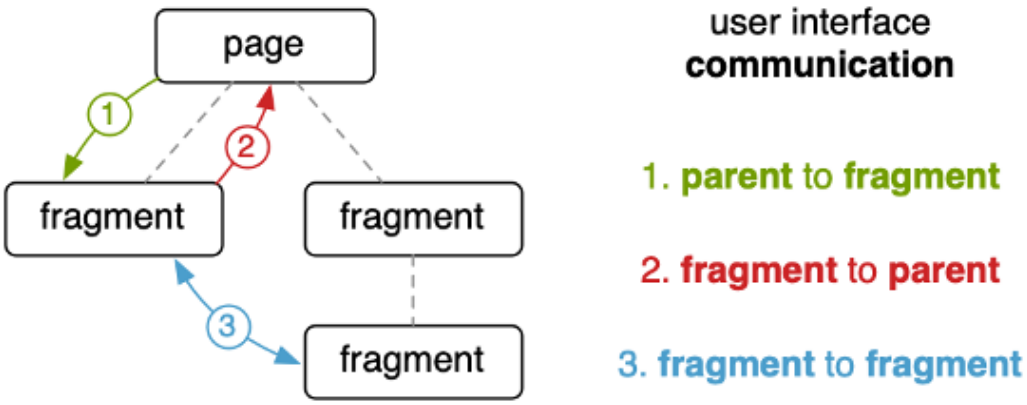


Figure 6.2 There are three different ways of communication that can happen between the different team's UIs inside a page.

We'll go through all three forms of communication with a real use case on our product page. We'll focus on native browser features in the examples.

6.1.1 Parent to fragment

The introduction of the buy button on the product page resulted in a considerable amount of tractor sales over one weekend. But Tractor Model Inc has no time to rest. CEO Ferdinand was able to hire two of the best goldsmiths. They've design special platinum editions for all tractors.

To sell these premium edition tractors *Team Decide* needs to add a **platinum upgrade option** to the detail page. Selecting the option should change the standard product image to the platinum version. *Team Decide* can implement that inside their application. But most importantly, the buy button from *Team Checkout* also needs to update. It must show the premium price of the platinum edition.

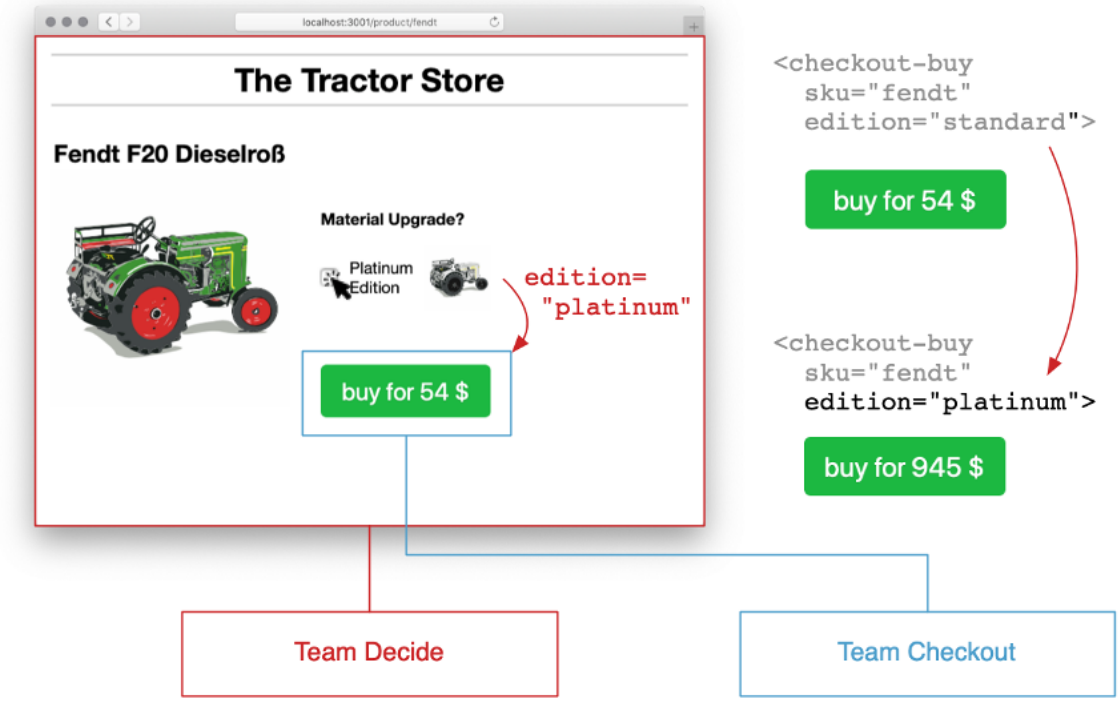


Figure 6.3 Parent-child communication. A change in the parent page (selection of platinum option) needs to be propagated down to a fragment so it can update itself (price change in the buy button).

Both teams talk and come up with a plan. *Team Checkout* will extend the buy button by another attribute called `edition`. *Team Decide* sets this attribute and update it accordingly when the user changes the option.

- **Updated Buy Button** tag-name: `checkout-buy` attributes: `sku=[sku]`, **`edition=[standard|platinum]`** example: `<checkout-buy sku="porsche" edition="platinum"></checkout-buy>`

IMPLEMENTING THE PLATINUM OPTION

The added option in the product pages markup looks like this:

Listing 6.1 team-decide/product/fendt.html

```
...
<img class="decide_image"
  src="https://mi-fr.org/img/fendt_standard.svg" />
...
<label class="decide_editions">
  <input type="checkbox" name="edition" value="platinum" />     ❶
  <span>Platinum Edition</span>
</label>
<checkout-buy sku="fendt" edition="standard"></checkout-buy>     ❶
...
```

❶ checkbox for selecting the platinum option

❷ buy button has a new `edition` attribute

Team Decide introduced a simple checkbox input element for choosing the material upgrade. The buy button component also received an `edition` attribute. Now they need to write a little bit of JavaScript glue-code to connect both elements. Changes to the checkbox should result in changes to the `edition` attribute. The main image on the site also needs to change.

Listing 6.2 team-decide/static/page.js

```
const option = document.querySelector(".decide_editions input");    ❶
const image = document.querySelector(".decide_image");              ❷
const buyButton = document.querySelector("checkout-buy");           ❸

option.addEventListener("change", e => {                            ❹
  const edition = e.target.checked ? "platinum" : "standard";       ❸
  buyButton.setAttribute("edition", edition);                       ❸
  image.src = image.src.replace(/(standard|platinum)/, edition);    ❸
});
```

❶ selecting the DOM elements that need to be watched or changed

❷ reacting to checkbox changes

❸ determining the selected edition

❹ updating the `edition` attribute on *Team Checkout's* buy button custom element

❺ updating the main product image

That's everything *Team Decide* needs to do. Now it's up to *Team Checkout* to react to the changed edition attribute and update the component.

UPDATING ON ATTRIBUTE CHANGE

The first version of the buy button custom element only used the `connectedCallback` methods. But custom elements also come with a few lifecycle methods.

The most interesting one for our case is `attributeChangedCallback(name, oldValue, newValue)`. This method is triggered every time someone changes an attribute of your custom element. You receive the name of the attribute that changed (`name`), the attribute's previous value (`oldValue`), and the updated value (`newValue`). For this to work, you have to register the list of attributes that should be observed upfront. The code of the custom element now looks like this:

Listing 6.3 team-checkout/static/fragment.js

```
const prices = {
  porsche: { standard: 66, platinum: 966 },          ❶
  fendt: { standard: 54, platinum: 945 },            ❶
  eicher: { standard: 58, platinum: 958 }            ❶
};

class CheckoutBuy extends HTMLElement {
  static get observedAttributes() {                  ❶
    return ["sku", "edition"];                       ❶
  }                                                  ❶
  connectedCallback() {
    this.render();                                   ❸
  }
  attributeChangedCallback() {                       ❹
    this.render();                                   ❹
  }                                                  ❹
  render() {                                         ❺
    const sku = this.getAttribute("sku");            ❻
    const edition = this.getAttribute("edition");    ❻
    this.innerHTML = `
      <button type="button">
        buy for ${prices[sku][edition]} $            ❼
      </button>
    `;
    ...
  }
}
```

❶ added new prices for platinum versions

❷ watching for changes to the `sku` and `edition` attribute

❸ extracted the rendering to a separate method

❹ calling `render()` on every attribute change

❺ extracted render method

❻ retrieves the current SKU and edition value from the DOM

❼ renders the price based on SKU and edition

NOTE The function name `render` has no special meaning in this context. We could have also picked another name like `updateView` or `gummibaer`.

```
npm run 10_parent_child_communication
```

Now the buy button updates itself on every change to the `sku` or `edition` attribute. Run the above code and go to localhost:3001/product/fendt in your browser and open up the DOM tree in the developer tools. You see that the `edition` attribute of the `checkout-buy` element changes every time you check and uncheck the platinum option. As a reaction to this, the component's internal markup (`innerHTML`) of it also changes.

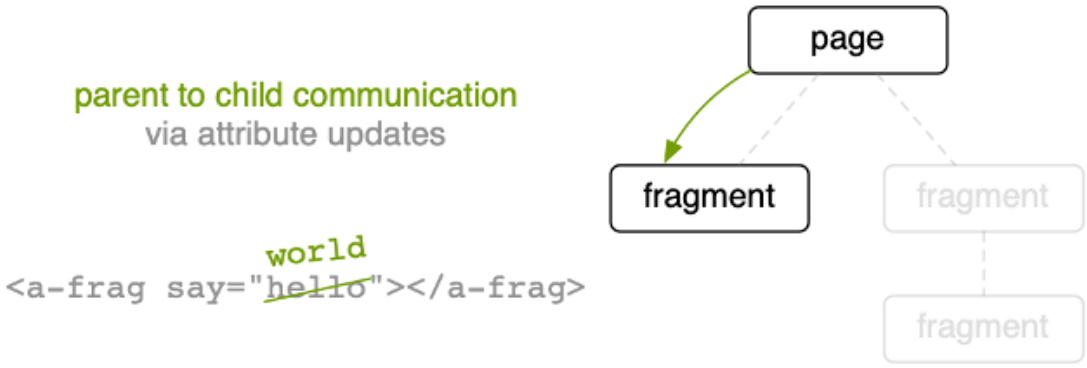


Figure 6.4 You can achieve parent-child communication by explicitly passing needed context information down as an attribute. The fragment can react to this change.

The way we propagate changed state of the outer application (product page) to the nested application (buy button) is similar to the *unidirectional dataflow* .[54] pattern. React, and Redux popularized the "props down, events up" approach. The updated state is passed down the tree via attributes to child components as needed. Communication, in the other direction, is done via events. We'll cover this next.

6.1.2 Fragment to parent

The introduction of the platinum editions resulted in a lot of controversial discussions in the *Tractor Model Inc.* user forum. Some users complained about the premium prices. Others asked for additional black, crystal, and gold editions. They shipped the first hundred platinum tractors within one day.

Emma is *Team Decide's* UX designer. She loves the new buy button but isn't entirely happy about how the user interaction feels. In response to a click, the user gets a system `alert` dialog, which he must dismiss to move on. Emma wants to change this. She has a more friendly alternative in mind. An animated green checkmark should confirm the add-to-cart interaction on the main product image.

This request is a little bit problematic. *Team Checkout* owns the add-to-cart action. Yes, they know when a user successfully added an item to the cart. It would be easy for them to show a confirmation message inside the buy button fragment. Or maybe animate the buy button itself to provide feedback. But they can't introduce a new animation in a part of the page they don't own, like the main product image.

Ok, technically they can because their JavaScript has access to the complete page markup, but they shouldn't. It would introduce a significant coupling of both user interfaces. *Team Checkout* would have to make a lot of assumptions on how the product page works. Future changes to the product page could result in breaking the animation. Nobody want's to maintain such a construct.

For a clean solution, the animation has to be developed by *Team Decide*. To accomplish this, both teams have to work together through a clearly defined contract. *Team Checkout* must notify *Team Decide* when a user has successfully added an item to the cart. *Team Decide* can trigger its animation in response to that.

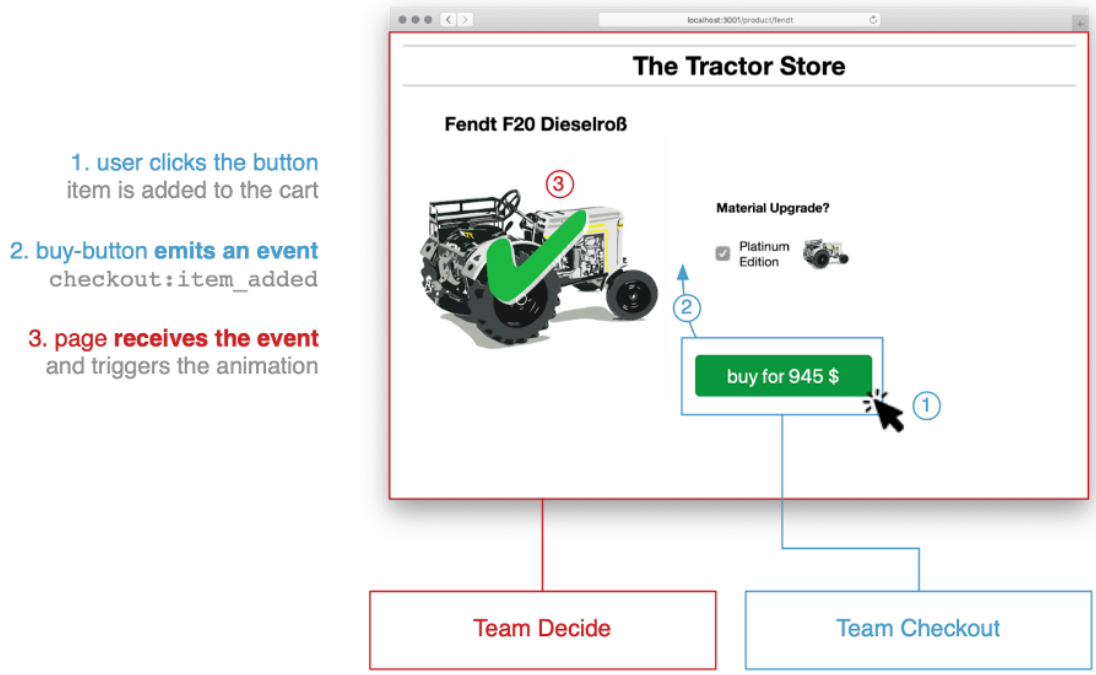


Figure 6.5 Team Checkout's buy button emits an event when the user adds an item to the cart. Team Decide reacts to this event and triggers an animation on the main product image.

Teams agree on implementing this notification via an event on the buy button. The updated contract for the buy button fragment looks like this:

- **Updated Buy Button** tag-name: `checkout-buy` attributes: `sku=[sku]`, `edition=[standard|platinum]` **emits event: `checkout:item_added`**

Now the fragment can emit a `checkout:item_added` event to inform others about a successful add-to-cart action.

EMITTING CUSTOM EVENTS

Let's look at the code that's needed to make the interaction happen. We'll use the browsers native `CustomEvents` API. The feature is available in all browsers, including older Internet Explorers. It enables you to emit events that work the same as native browser events like `click` or `change`. But you are free to choose the events name.

The following code shows the buy button fragment with the event added.

Listing 6.4 team-checkout/static/fragment.js

```
class CheckoutBuy extends HTMLElement {
  ...
  render() {
    ...
    this.innerHTML = ...;
    this.querySelector("button").addEventListener("click", () => {
      ...
      const event = new CustomEvent("checkout:item_added");        ❶
      this.dispatchEvent(event);                                   ❷
    });
  }
}
```

❶ creates a custom event named `checkout:item_added`

❷ dispatches the event at the custom element

NOTE We've used a team prefix (`[team_prefix]:[event_name]`) to clarify which team owns the event.

Pretty straight forward, right? The `CustomEvent` constructor has an optional second parameter for options. We'll discuss two options in the next example.

LISTENING FOR CUSTOM EVENTS

That's everything *Team Checkout* needed to do. Let's add the checkmark animation when the event occurs. We'll not get into the associated CSS code. It uses a CSS keyframe animation, which makes a prominent green checkmark character (✓) fade in and out again. We can trigger the animation by adding a `decide_product–confirm` class to the existing `decide_product` div element.

Listing 6.5 team-decide/static/page.js

```
const buyButton = document.querySelector("checkout-buy");          ❶
const product = document.querySelector(".decide_product");         ❶
buyButton.addEventListener("checkout:item_added", e => {           ❶
  product.classList.add("decide_product--confirm");                ❶
});                                                                ❶
product.addEventListener("animationend", () => {                   ❷
  product.classList.remove("decide_product--confirm");             ❺
});                                                                ❺
```

❶ selecting the buy button element

❷ selecting the product block where the animation should happen

❸ listening to *Team Checkout's* custom event

❹ trigger the animation by adding the `confirm` class

❺ cleanup - removing the class after the animation finished

Listening to the custom `checkout:item_added` event works the same way as listening to a `click` event. Select the element you want to listen on (`<checkout-buy>`) and register an event handler: `.addEventListener("checkout:item_added", () {…})`. Run the following command to start the example:

```
npm run 11_child_parent_communication
```

Go to localhost:3001/product/fendt in your browser and try the code yourself. Clicking the buy button triggers the event. *Tean Decide* receives it and adds the `confirm` class. The checkmark animation starts.

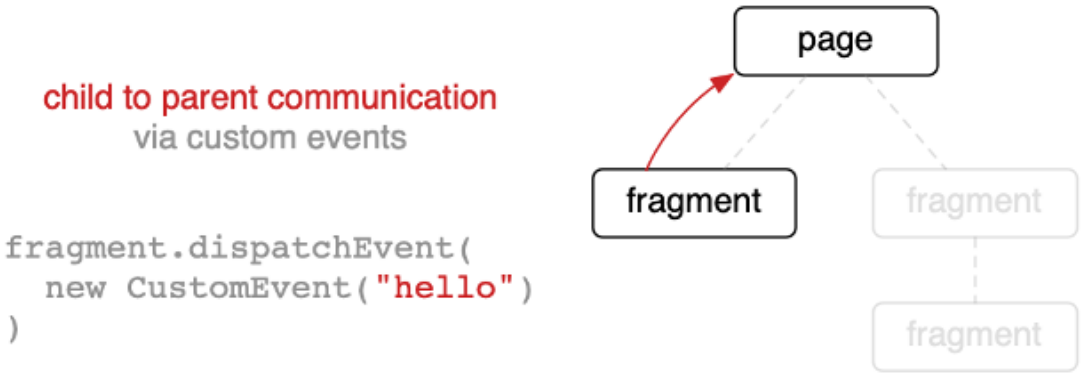


Figure 6.6 Child-parent communication can be implemented by using the browsers builtin event mechanism.

Using the browsers event mechanism has multiple benefits:

- Custom events can have high-level names that reflect your domain language. Good event names make them easier to understand than technical names like `click` or `touch`.
- Fragments don't need to know their parents.
- All major libraries and frameworks support browser events.
- Access to all native event features like `.stopPropagation` or `.target`
- Easy debugging via browser developer tools

Let's get to the last form of communication: fragment to fragment.

6.1.3 Fragment to fragment

Replacing the alert dialog with the friendlier checkmark animation had a measurable positive effect. The average cart size went up by 31%, which directly resulted in higher revenue. The support staff reported that some customers accidentally bought more tractors than they intended.

Team Checkout wants to add a mini-cart to the product page to reduce the number of product returns. This way, customers always see what's in their basket. *Team Checkout* provides the

mini-cart as a new fragment for *Team Decide* to include on the bottom of the product page. The contract for including the mini-cart looks like this:

- **Mini-Cart** tag-name: `checkout-minicart` example: `<checkout-minicart></checkout-minicart>`

It does not receive any attributes and emits no events. When added to the DOM, the mini-cart renders a list of all tractors that are in the cart. Later the team will fetch the state from its backend API. For now, the fragment holds that state in a local variable.

That's all pretty straight forward, but the mini-cart also needs to be notified when the customer adds a new tractor to the cart via the buy button. So an event in fragment A should lead to an update in fragment B. There are different ways of implementing this:

1. **Direct communication**: A fragment finds the fragment it wants to talk to and directly calls a function on it. Since we are in the browser, a fragment has access to the complete DOM tree. It could search the DOM for the element it's looking for and talk to it. **Don't do this. Directly referencing foreign DOM elements introduces tight coupling.** A fragment should be self-contained and not know about other fragments on the page. Direct communication makes it hard to change the composition of fragments later on. Removing a fragment or duplicating one can lead to strange effects.
2. **Orchestration via a parent**: We can combine the *child-parent* and *parent-child* mechanisms. In our case, *Team Decide's* product page would listen to the `item_added` event from the buy button and directly trigger an update to the mini-cart fragment. This is a clean solution. We've explicitly modeled the communication flow in the parent's system. But to make a change in communication, two teams must adapt their software.
3. **Event-Bus / broadcasting**: With this model, you introduce a global communication channel. Fragments can publish events to the channel. Other fragments can subscribe to these events and react to them. The publish/subscribe mechanism reduces coupling. The product page, in our example, wouldn't have to know and care about the communication between the buy button and the mini-basket fragment. You can implement this with Custom Events. Most browsers.[55] also support the new **Broadcast Channel API** .[56], which creates a message bus that also spans across browser windows, tabs, and iframes.

The teams decide to go with the event-bus approach using Custom Events. Figure 6.7 illustrates the event flow between both fragments.

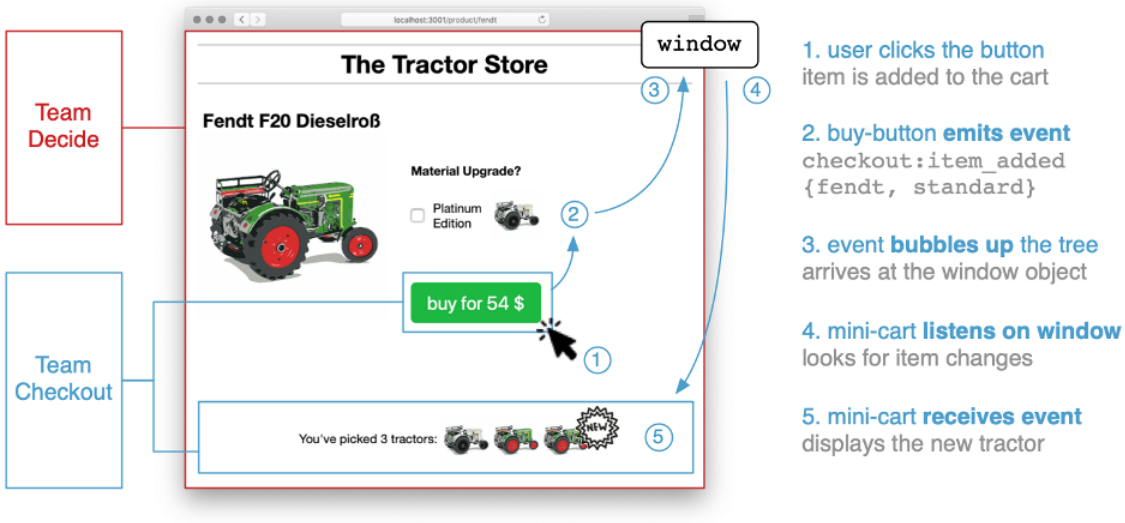


Figure 6.7 Fragment to fragment communication via a global event. The buy button emits the item_added event. The mini-cart listens for this event on the window object and updates itself. We use the browser's native event mechanism as an event-bus.

The mini-cart doesn't only need to know **if** the user added a tractor, it also must know **what** tractor the user added. So we need to add the tractor information (`sku`, `edition`) as a payload to the `checkout:item_added` event. The updated contract for the buy button looks like this:

- **Updated Buy Button** tag-name: `checkout-buy` attributes: `sku=[sku]`, `edition=[standard|platinum]` emits event:
  - name: `checkout:item_added`
  - **payload: `{sku: [sku], edition: [standard|platinum]}`**

> **WARNING** Be careful with exchanging data structures through events. They introduce extra coupling. Keep payloads to a minimum. Use events primarily for notifications and not to transfer data.

Let's look at the implementation of this.

EVENT-BUS VIA BROWSER EVENTS

The Custom Events API also specifies a way to add a custom payload to your event. You can pass your payload to the constructor via the `detail` key in the options object.

Listing 6.6 team-checkout/static/fragment.js

```
...
const event = new CustomEvent("checkout:item_added", *{
  bubbles: true,                 ❶
  detail: { sku, edition }       ❶
}*);
this.dispatchEvent(event);
...
```

❶ enables event bubbling

❷ attaching a custom payload to the event

By default, Custom Events don't bubble up the DOM tree. We need to enable this behavior to make the event rise to the window object.

That's everything we needed to do to the buy button. Let's look at the mini-cart implementation. *Team Checkout* defines the custom element in the same `fragment.js` file as the buy button.

Listing 6.7 team-checkout/static/fragment.js

```
...
class CheckoutMinicart extends HTMLElement {
  connectedCallback() {
    this.items = [];                                                    ❶
    window.addEventListener("checkout:item_added", e => {               ❷
      this.items.push(e.detail);                                        ❸
      this.render();                                                    ❹
    });                                                                 ❷
    this.render();
  }
  render() {
    this.innerHTML = `
      You've picked ${this.items.length} tractors:
      ${this.items.map(({ sku, edition }) =>
        `<img src="https://mi-fr.org/img/${sku}_${edition}.svg" />`
      ).join("")}
    `;
    ...
  }
}
window.customElements.define("checkout-minicart", CheckoutMinicart);
```

❶ initializing a local variable for holding the cart items

❷ listening to events on the window object

❸ reading the event payload and adding it to the item list

❹ updating the view

The component stores the basket items in the local `this.items` array. It registers an event-listener for all `checkout:item_added` events. When an event occurs, it reads the payload (`event.detail`) and appends it to the list. At last, it triggers a refresh of the view by calling `this.render()`.

To see both fragments in action, *Team Decide* has to add the new mini-cart fragment to the bottom of the page. The team doesn't have to know anything about the communication that's going on between `checkout-buy` and `checkout-minicart`.

Listing 6.8 team-decide/product/fendt.html

```
...
<body>
  ...
  <div class="decide_details">
    <checkout-buy sku="fendt" edition="standard"></checkout-buy>
  </div>
  <div class="decide_summary">                          (1)
    <checkout-minicart></checkout-minicart>             (1)
  </div>                                                (1)
  <script src="http://localhost:3003/static/fragment.js" async></script>
</body>
...
```

(1) adding the new mini-cart fragment to the bottom of the page

Figure 6.8 shows how the event is bubbling up to the top. You can test the example by running this command.

```
npm run 12_fragment_fragment_communication
```

Figure 6.8 Custom events can bubble up to the window of the document where other components can subscribe to it.

DISPATCHING EVENTS DIRECTLY WINDOW

It's also possible to directly dispatch the Custom Event to the global `window` object: `window.dispatchEvent` instead of `element.dispatchEvent`. But dispatching it to the DOM element and letting it bubble up comes with a few benefits. The origin of the event (`event.target`) is maintained. Having this extra information can help with debugging. It's also possible for a parent to intercept the event (`event.stopPropagation`) on it's way up to the `window`.

6.1.4 When UI communication is a good fit

Now you've seen three different types of communication, and you know how to tackle them with basic browser features. You can, of course, also use custom implementations for communicating and updating components. A shared JavaScript publish/subscribe module which all teams import at runtime can do the trick. But your goal when setting up a micro frontends integration should be to have as little shared infrastructure as needed. Going with a standardized browser specification like Custom Events or the Broadcast Channel API is a good choice when you get started.

USE SIMPLE PAYLOADS

In the last example, we've transferred the actual cart line-item (`{sku, edition}`) via an event from one fragment to another. In the projects I've worked on, we've had good experiences with keeping events as lean and straightforward as possible. Events should not function as a way to transfer data. Their purpose is to act as a nudge to other parts of the user interface. You should only exchange view-models and domain objects inside team boundaries.

INTENSE NEED FOR UI COMMUNICATION CAN BE A SMELL

As stated earlier: when you've picked your team boundaries well, there shouldn't be a lot of need for inter-team communication. That said, the amount of communication increases when you are adding a lot of different use-cases to one view.

When implementing a new feature requires two teams to work closely together, passing data back and forth between their micro frontends, we have a reliable indicator of non-optimal team boundaries. Reconsider your boundaries and maybe increase the scope or shift the responsibility for a use-case from one team to another.

EVENTS VS. ASYNC LOADING

When using events or broadcasting, you have to keep in mind that not all other micro frontends might have finished loading yet. Micro frontends are unable to retrieve events that happened before they finished initializing themselves.

When you use events in response to user actions (like add-to-cart) this is not a big issue in practice. But if you want to propagate information to all components on the initial load, standard events might not be the right solution.

6.2 Other communication mechanisms

6.2.1 Global context & authentication

Each micro frontend addresses a particular use-case. However, in a non-trivial application, these frontends need some context information to do their job. *What language does the user speak, where does she live, and which currency does she prefer? Is the user logged in or anonymous? Is the application running in the staging or live environment?* These necessary details are often called **context information**. They are read-only by nature. You can see context data as infrastructure boilerplate that you want to solve once and provide it to all the teams in an easily consumable way. Figure 6.9 illustrates how to distribute this data to all user interface applications.

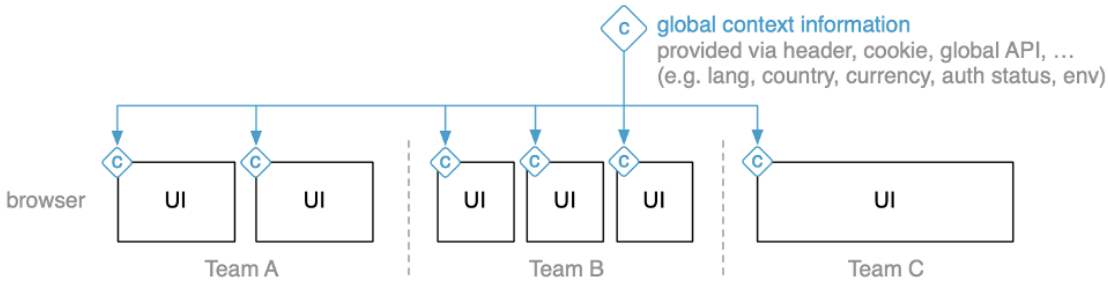


Figure 6.9 You can provide general context information globally to all micro frontends. This puts common tasks like language-detection in a central place.

PROVIDING CONTEXT INFORMATION TO ALL MICRO FRONTENDS

We have to answer two questions when building a solution for providing context data:

1. **Delivery:** How do we get the information to the teams micro frontends?
2. **Responsibility:** Which team determines the data and implements the associated concepts?

Let's start with **delivery**. If you're using server-rendering HTTP headers or cookies are a popular solution. A frontend-proxy or composition service can set them to every incoming request. If you're running an entirely client-side application, HTTP headers are not an option. As an alternative, you can provide a global JavaScript API via which every team can retrieve this information. In the next chapter, we'll introduce the concept of an *application shell*. When you decide to go that route, putting the context information into the application shell is a typical pattern.

Let's talk about **responsibility**. If you have a dedicated platform team, it's the perfect candidate also to provide the context. In a decentralized scenario with no platform team, you'd pick one of the teams to do the job. If you already have a central infrastructure like a frontend-proxy and an

application shell, the owner of this infrastructure is a good candidate for also owning the context data.

AUTHENTICATION

Managing language preferences or determining the origin country are tasks that don't require much business logic. For topics like *authenticating a user*, it's harder. You should answer the question, "Which team owns the login process?" by looking at the team's *mission statements*.

From a technical integration standpoint, the team owning the login process becomes the authentication provider for the other teams. It provides a login page or fragment that other teams can use to redirect an unauthenticated user towards. You can use standards like OAuth.[57] or JSON Web Tokens (JWT) to securely provide the authentication status to the teams that need it.

6.2.2 Managing state

If you're using a state management library like Redux, each micro frontend or at least each team should have its local state. Figure 6.10 illustrates this.

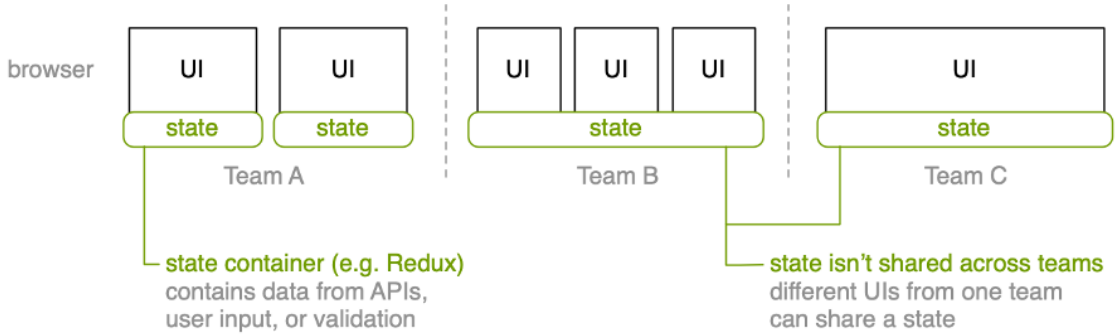


Figure 6.10 Each team has its own user interface state. Sharing state between teams would introduce coupling and make the applications hard to change later on.

It's tempting to reuse state from one micro frontend in another to avoid loading data twice. But this shortcut leads to coupling and makes the individual applications harder to change and less robust. It also introduces the potential that a shared state gets misused for inter-team communication.

6.2.3 Frontend-backend communication

To do its work, a micro frontend should only talk to the backend infrastructure of its team. A micro frontend from Team A would never directly talk to an API endpoint from Team B. This would introduce coupling and inter-team dependencies. Even more important, you give up isolation. To run an test your system, the system from the other team needs to be present. An error in Team B would also affect fragments from Team C.

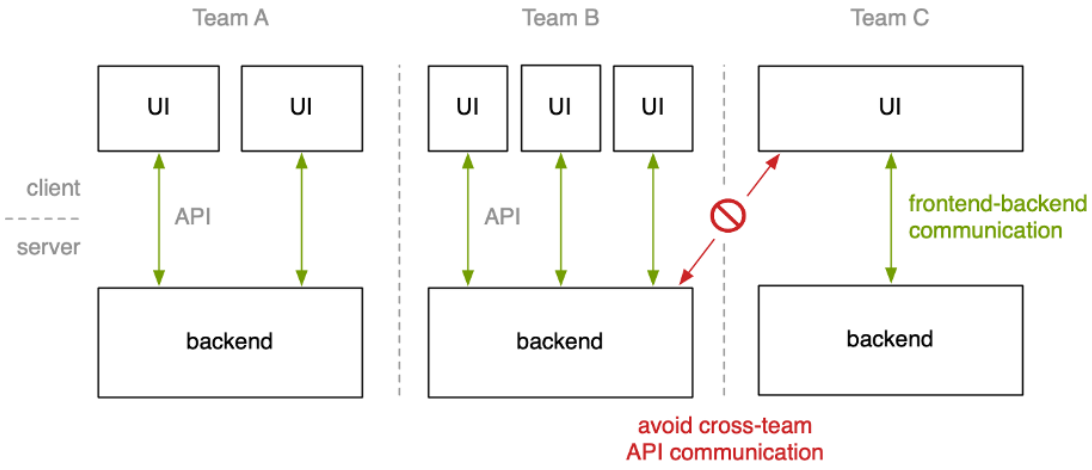


Figure 6.11 API communication should always stay inside team boundaries.

6.2.4 Data Replication

If your teams should own everything from the user interface to the database, each team needs its own server-side data store. *Team Inspire* maintains its database of manually crafted product recommendations, whereas *Team Checkout* stores all baskets and orders the users created. *Team Decide* has no direct interest in these data structures. They include the associated functionality (like recommendation strip or minicart) via UI composition in the frontend.

But for some applications, UI composition is not feasible. Let's take the product data as an example. *Team Decide* owns the master product database. They provide back-office functionality, which employees of *The Tractor Store* can use to add new products. But the other teams also need some product data. *Team Inspire* and *Team Checkout* need at least the list of all *SKUs*, the associated names, and image URLs. They've no interest in more advanced information like editing history, video files, or customer reviews.

Both teams could retrieve this information via API calls to *Team Decide* at runtime. However, this would violate our autonomy goals. If *Team Decide* goes down, the other teams aren't able to do their job anymore. We can solve this with data replication.

Team Decide provides an interface that the other teams can use to retrieve a list of all products. The other teams use this interface to replicate the needed product information regularly in the background. You'd implement this via a feed mechanism. Figure 6.12 illustrates this.

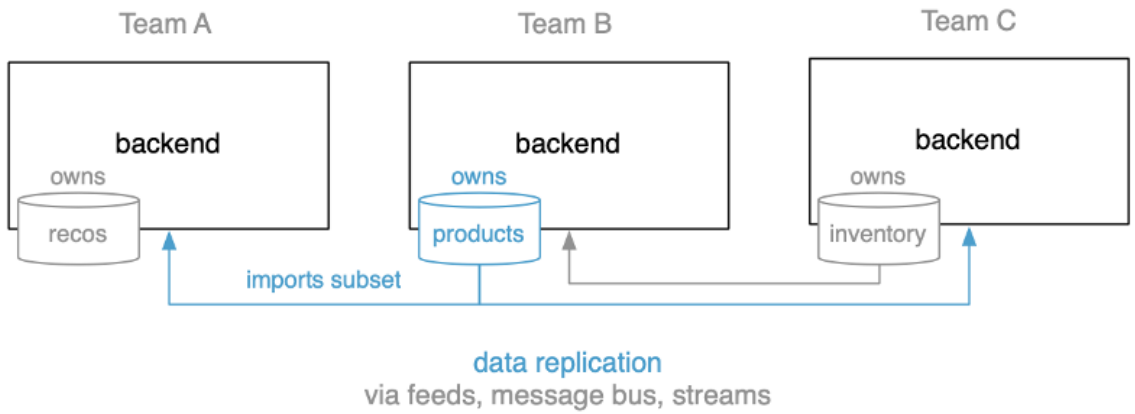


Figure 6.12 Teams can replicate data from other teams to stay independent. This replication increases robustness. If one team goes down, the others can still function.

When *Team Decide's* application goes down, *Team Inspire* still has its local product database it can use to serve recommendations. We can apply this concept to other kinds of data.

Team Checkout owns the inventory. They know how many tractors are in stock and can estimate when new supplies arrive. If another team has an interest in this inventory data, they have two options: replicate the needed data to their application or ask *Team Checkout* to provide an includable micro frontend that presents this information directly to the user.

Both approaches are valid options that have their benefits and drawbacks. *Team Decide* can choose to replicate the inventory data if they want to build business logic that builds upon it. As an example, they might want to experiment with an alternative product detail layout for products that run out of stock soon. To do this, they must know the inventory in advance, have to understand *Team Checkout's* inventory format, and build the associated business rules.

Alternatively, if they just want to show the inventory information as a simple text as part of the buy button, UI composition is much more comfortable. *Team Decide* doesn't have to understand *Team Checkout's* inventory data model at all.

6.3 Summary

- Communication between different micro frontends is often necessary at the handover points in your application. When the user moves from one use-case to the next. You can handle most communication needs by passing parameters through the URL.
- When multiple use-cases exist on one page, it might be necessary for the different micro frontends to communicate with each other.
- You can use the "props down, events up" communication pattern on a higher level between different team UIs.
- A parent passes updated context information down to its child fragments via attributes.
- Fragments can notify other fragments higher up in the tree about a user action using native browser events.
- Different fragments that are not in a parent-child relationship can communicate using an event-bus or broadcasting mechanism. Custom Events and the Broadcast Channel API are native browser implementations that can help.
- You should use UI communication only for notifications, not to transfer complex data structures.
- You can resolve general context information like the user’s language or the country in a central place (e.g., frontend-proxy or application shell) and passed it to every micro frontend. HTTP headers, cookies, or a shared JavaScript API are ways to implement it.
- Each team can have its own user interface state (e.g., a Redux store). Avoid sharing state between teams. It introduces coupling and makes applications hard to change.
- A team’s micro frontend should only fetch data from its backend application. Exchanging larger data structures across team UIs leads to coupling and makes applications hard to evolve and test.

7

Client-side Routing & The Application Shell

This chapter covers

- Applying the concepts of inter-team routing to a single-page app.
- Constructing a shared application shell as a single entry point for the user.
- Exploring different approaches for client-side routing.
- Discover how the micro frontends meta-framework single-spa can make integration easier.

In the last two chapters, we focused on composition and communication. We integrated user interfaces from different teams into one view. You learned server- and client-side techniques for doing this. In this chapter, we'll take a step back and look at page-level integration.

In chapter 2.2. Page transition via links we already covered the most basic page-integration technique: the plain old link. Later in chapter 3.2. Server-side routing via Nginx you saw how to implement a common router that forwards an incoming page-request to the responsible team. Now we'll take these concepts and apply them to client-side routing and single-page apps.

Most JavaScript frameworks come with a dedicated routing solution like `@angular/router` or `vue-router`. They make it possible to navigate through different pages of an application without having to do a full page refresh on every link click. Because the browser does not have to fetch and process a new HTML document, a client-side page transition feels snappier and leads to a better user experience. The browser only needs to rerender the parts of the page that changed. It doesn't have to evaluate referenced assets like JavaScript and stylesheets again. We'll use the terms **hard navigation** and **soft navigation** in this chapter:

- **Hard navigation** describes a page-transition where the browser needs to load the complete HTML for the next page from the server.
- **Soft navigation** refers to a page-transition that's entirely client-side rendered. Typically

by using a client-side router.

In a monolithic frontend application, it's typically a binary decision. Either you build an application with server-rendered pages, or you choose to implement a SPA. In the first case, you use hard navigations for everything. In the second case, the SPA, you have one client-side router that enables soft navigations. In a micro frontends context, it doesn't have to be that black and white. Figure 7.1 shows two simple ways to integrate pages.

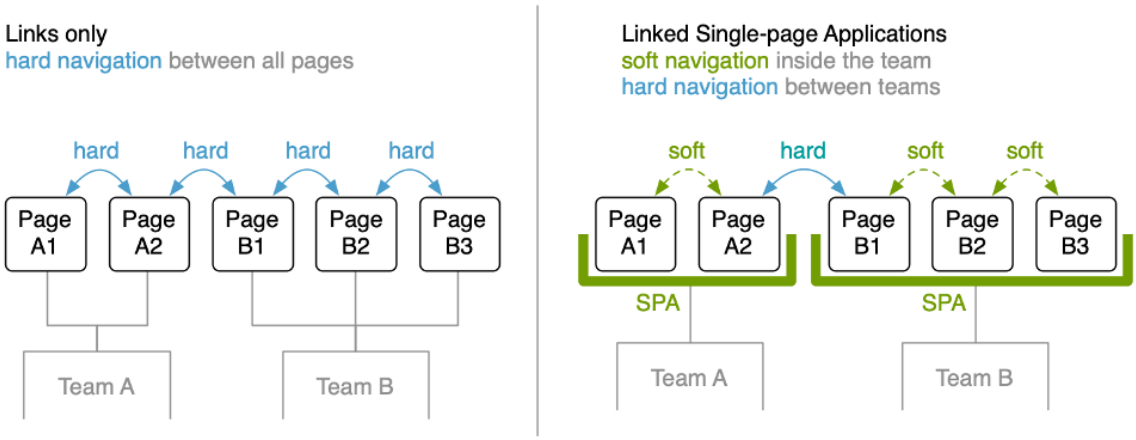


Figure 7.1 Two different approaches for page transitions in a micro frontends context. The "Link only" model is simple. Page transitions happen via plain links, which result in a full refresh of the page. Nothing special is needed - Team A must know how to link to the pages of Team B and vice versa. With the "Linked Single Page Apps" approach, all transitions inside team boundaries are soft. Hard navigation happens when the user crosses team boundaries. From an architectural perspective, its identical to the first approach. The fact that a team uses a SPA for its pages is an implementation detail. As long as it responds correctly to URLs, the other team doesn't have to care.

In these options, the link is the only contract between the teams. There is no other technical requirement or shared code needed to make it work. However, both versions include hard navigations. If this is acceptable depends on your use-case and especially the number of teams. When your goal is a setup with many teams that are each responsible for only one page, you end up with a lot of hard navigations.

Figure 7.2 shows a third option where all page transitions are soft.

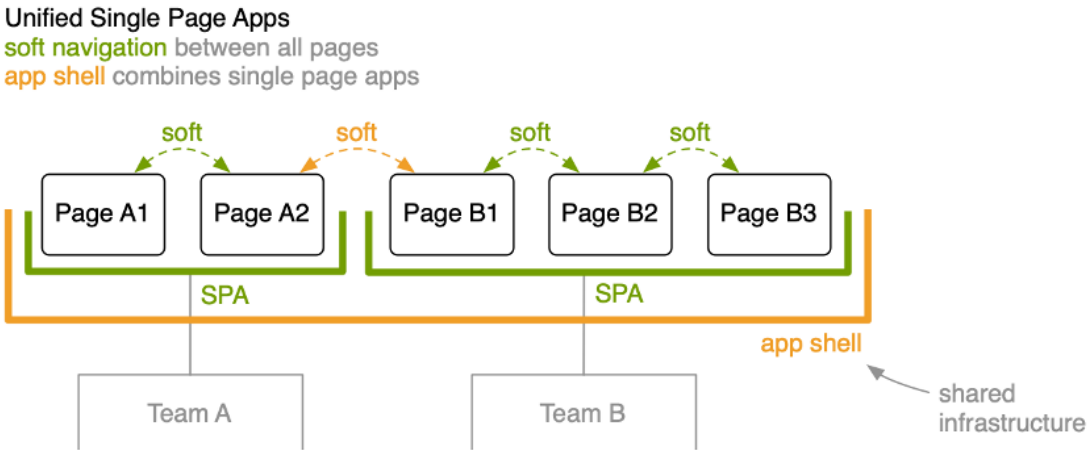


Figure 7.2 The Unified Single Page Apps approach introduces a central application container. It handles page transitions between the teams. Here all navigations are soft.

We'll focus on the "Unified Single Page App" model in this chapter. You'll implement an application shell (short app shell) yourself. It contains a simple router that we later upgrade to a more sophisticated and maintainable version. At the end of the chapter, we look at the micro frontends meta-framework single-spa, which is an out-of-the-box app shell solution.

7.1 App shell with flat routing

The micro frontends architecture had many great benefits for *Tractor Models Inc.* so far. They were able to build their online shop in a short amount of time. The three teams are highly motivated and eager to evolve their slice of the system to deliver a perfect customer experience.

In a company-wide meeting, they discussed the idea of moving to a full client-rendered user interface. Soft navigation should be possible across all pages, not only inside team boundaries. In a monolithic world, this would be straightforward: use the router of your favorite JavaScript framework - you're done. However, they don't want to introduce stronger coupling between the teams. Independent deployments and dependency upgrades should continue to be possible to ensure fast iteration. Moving to one shared framework would do the opposite.

The teams are confident that it's possible to build a technology-agnostic client-side router to enable page transitions. They know that similar ready-to-use implementations already exist. However, since this central router would become a fundamental part of their architecture, they decide to build a prototype version of it from scratch first. This way, they fully understand how all moving parts play together.

7.1.1 What's an app shell?

The app shell acts as a parent application for all micro frontends. All incoming requests arrive there. It selects the micro frontend the user wants to see and renders it in the <body> of the document. Figure 7.3 illustrates this.

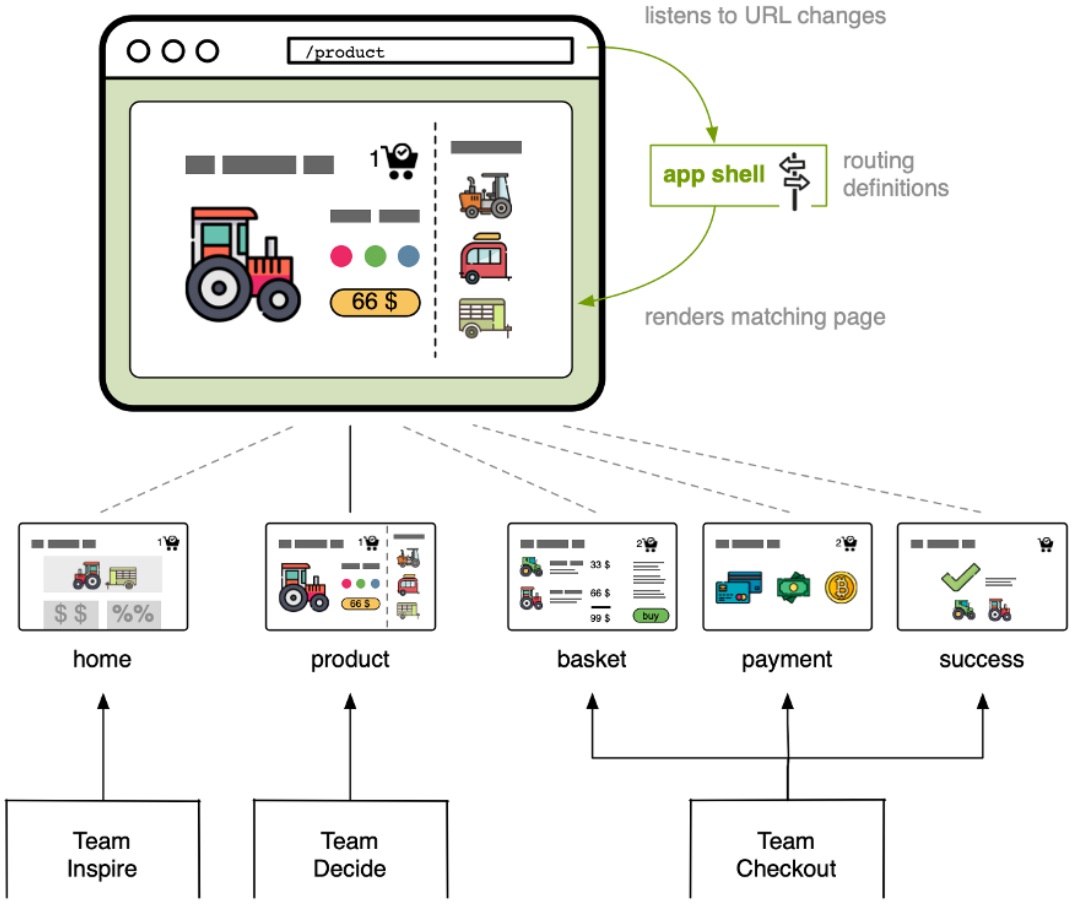


Figure 7.3 The app shell acts as a central client-side router. It watches for URL changes, determines the matching page (micro frontend) and renders it.

Since this container application is a shared piece of code, it's a good idea to keep it as simple as possible. **It should not contain any business logic.** Sometimes topics that affect all teams like authentication or analytics are also built into the app shell. However, we'll stick to the basics for now.

7.1.2 Anatomy of the app shell

The four essential parts of a micro frontends app shell are:

1. Provide a shared HTML document.
2. Map URLs to team pages (client-side routing).
3.

3. Render the matching page.
4. (De)initialize the previous/next page on a navigation.

Let's build them in this order. Since the app shell is a central infrastructure, it's code lives next to the team's applications. You can see the folder structure of the sample code in figure 7.4.

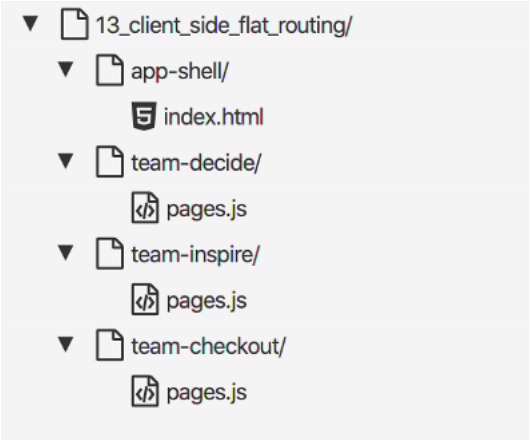


Figure 7.4 The app shell's code is located beside the team's code. It provides a shared HTML document. The teams just deliver page components via JavaScript.

As in the last chapters, each folder represents an application that's developed and deployed independently. In the example, the app-shell listens on port `3000`, and the team applications run on ports `3001`, `3002`, and `3003`.

If you are building a fully client-rendered application, it's typical to have a single `index.html` file. It acts as the entry point for all incoming requests. The actual routing happens in the browser via JavaScript.

To make this happen, we need to configure our web-server to return the `index.html` when it encounters an unknown URL. In Apache or Nginx, you can do this by specifying rewrite rules. Fortunately, our ad-hoc web-server (`mfserve`) has an option to enable this behavior. We are adding the `--single` parameter to do the trick. Start the app-shell and the three applications by running this command.

```
npm run 13_client_side_flat_routing
```

Now, the server answers all incoming requests like `/`, `/product/porsche`, or `/cart` with the content of the `index.html`.

Let's have a look at the markup:

Listing 7.1 app-shell/index.html

```
<html>
  <head>
    <title>The Tractor Store</title>
    <script src="https://unpkg.com/history@4.9.0"></script>          ❶
    <script src="http://localhost:3001/pages.js" async></script>     ❶
    <script src="http://localhost:3002/pages.js" async></script>     ❷
    <script src="http://localhost:3003/pages.js" async></script>     ❸
  </head>
  <body>
    <div id="app-content">                                           ❸
      <span>rendered page goes here<span>                            ❸
    </div>                                                           ❸
    <script type="module">                                           ❹
      /* routing code goes here */                                   ❹
    </script>                                                        ❹
  </body>
</html>
```

❶ a dependency we'll use in the router code

❷ the application code for all teams

❸ container for the actual page content

❹ place for the app shell's routing code

Now we have our HTML document. It references the JavaScript code of all teams. These files contain the code for the page components. The document also has a container for the actual content (`#app-content`). That's pretty straight forward. Let's get to the exciting part: the routing.

7.1.3 Client-side routing

There are many ways to build a client-side router. We could use a full-featured existing routing solution like `vue-router`. However, since we want to keep it simple, we'll build our own that's based on the `history` library.[58]. This library is a thin wrapper around the browser's History API. Many higher-level routers like `react-router` use it under the hood. Don't worry if you haven't used `history` before. We'll only use two features: `listen` and `push`.

Listing 7.2 app-shell/index.html

```
...
const appContent = document.querySelector("#app-content");

const routes = {                                                    ❶
  "/": "inspire-home",                                              ❷
  "/product/porsche": "decide-product-porsche",                     ❸
  "/product/fendt": "decide-product-fendt",                         ❹
  "/product/eicher": "decide-product-eicher",                       ❸
  "/checkout/cart": "checkout-cart",                                ❸
  "/checkout/pay": "checkout-pay",                                  ❸
  "/checkout/success": "checkout-success"                           ❶
};

function findComponentName(pathname) {                              ❷
  return routes[pathname] || "not found";                           ❷
}                                                                   ❷

function updatePageComponent(location) {                            ❸
  appContent.innerHTML = findComponentName(location.pathname);      ❸
}                                                                   ❸

const appHistory = window.History.createBrowserHistory();          ❹
appHistory.listen(updatePageComponent);                             ❺
updatePageComponent(window.location);                               ❻

document.addEventListener("click", e => {                           ❼
  if (e.target.nodeName === "A") {                                  ❼
    const href = e.target.getAttribute("href");                     ❼
    appHistory.push(href);                                          ❼
    e.preventDefault();                                             ❼
  }                                                                 ❼
});                                                                 ❼
...
```

❶ maps an URL path to the component name

❷ looks up a component based on a pathname

❸ writes the component name into the content container

❹ instantiates the history library

❺ registers a history listener that's called every time the URL changes either through a push/replace call or by pressing the browsers back/forward controls

❻ calling the update function once on start to render the first page

❼ registers a global click listener that intercepts link-clicks, passes the target URLs to the history and prevents a hard navigation

This is quite a bit of code. Let's see what it does.

KEEPING URL AND CONTENT IN SYNC

The central piece is the `updatePageComponent(location)` function. It keeps the displayed content in sync with the browser's URL. It's called once on initialization and every time the browser history changes (`appHistory.listen`). The change can be due to a navigation request through the JavaScript API via `appHistory.push()` or when the user clicks the back or forward button in the browser. The `updatePageComponent` function looks up the page component that matches the current URL. For now it puts the component name into the `div#app-content` element via `innerHTML`. This way, the browser shows one line of text which contains the matched name. The name acts as a placeholder for us. We'll upgrade this to rendering a real component in a minute.

MAPPING URLS TO COMPONENTS

The `routes` object is a simple pathname (`key`) to component name (`value`) mapping. Here is an excerpt from the code you saw before.

Listing 7.3 app-shell/index.html

```
...
const routes = {
  "/": "inspire-home",
  "/product/porsche": "decide-product-porsche",
  ...
  "/checkout/pay": "checkout-pay",
  "/checkout/success": "checkout-success"
};
...
```

So every page is a component. The component's name starts with the name of the responsible team. For the URL `/checkout/success`, the app shell should render the `checkout-success` component, which *Team Checkout* owns.

7.1.4 Rendering pages

The app shell includes a JavaScript file from each team. Let's have a look inside these files. As you might have guessed, we are using Web Components as a neutral component format. A team exposes their page as a Custom Element. The app shell needs to know the name of this component. It doesn't care what technology the page component uses internally. We use the same approach discussed in the last chapter 5.1. Wrapping Micro Frontends using Web Components. However, now on a page- and not on a fragment-level.

The following code shows *Team Inspire's* homepage component:

Listing 7.4 team-decide/pages.js

```
class InspireHome extends HTMLElement {
  connectedCallback() {
    this.innerHTML = `
      <h1>Welcome to The Tractor Store!</h1>
      <strong>Here are three tractors:</strong>
      <a href="/product/porsche">Porsche</a>                    (1)
      <a href="/product/eicher">Eicher</a>                      (1)
      <a href="/product/fendt">Fendt</a>                        (1)
    `;
  }
}

window.customElements.define("inspire-home", InspireHome);     (1)
```

(1) links to the product page owned by *Team Decide*

(2) adds the Custom Element to the global registry

It's a simplified example. In a real-world implementation, we would also see data-fetching, templating, and styling here. The `connectedCallback` is the entry point for the teams to display their content. The code for the other pages looks similar. Here an example for a product page:

Listing 7.5 decide/pages.js

```
class DecideProductPorsche extends HTMLElement {
  connectedCallback() {
    this.innerHTML = `
      <a href="/">&lt; home</a> -                               (1)
      <a href="/checkout/cart">view cart &gt;</a>               (2)
      <h1>Porsche-Diesel Master 419</h1>
      <img src="https://mi-fr.org/img/porsche.svg" width="200">
    `;
  }
}
window.customElements.define(                                  (3)
  "decide-product-porsche",                                     (3)
  DecideProductPorsche                                          (3)
);                                                              (3)
...
```

(1) link to *Team Inspire's* homepage

(2) link to *Team Checkout's* cart page

(3) adds the Custom Element to the global registry

The structure is the same as with *Team Inspire's* homepage. Only the content is different. Let's enhance the `updatePageComponent` implementation so that it instantiates the correct Custom Element and not only displays the component name.

Listing 7.6 app-shell/index.html

```
...
function updatePageComponent(location) {
  const next = findComponentName(location.pathname);    ❶
  const current = appContent.firstChild;                ❶
  const newComponent = document.createElement(next);    ❶
  appContent.replaceChild(newComponent, current);       ❶
}
...
```

❶ looks up the component name for the current location

❷ reference to the existing page component

❸ instantiates the Custom Element

❹ replacing the existing component with the new one (`disconnectedCallback` of the old one and `connectedCallback` of the new one are triggered)

The above code is all standard DOM API. Creating a new element and replacing an existing one with it. Our app shell is a straightforward broker that listens to the History API and updates the page via simple DOM modification. The teams can hook into the Custom Elements lifecycle methods to get the right hooks for initialization, deinitialization, lazy loading, and updating. No framework or fancy code needed.

LINKING BETWEEN MICRO FRONTENDS

Let's look at navigation. That's the whole point of this exercise. We want to achieve fast client rendered page transitions. You might have noticed that both pages have links that point to other teams. The app shell handles these links. It contains a global click listener. Here is an excerpt from the code you saw before:

Listing 7.7 app-shell/index.html

```
...
document.addEventListener("click", e => {                ❶
  if (e.target.nodeName === "A") {                       ❶
    const href = e.target.getAttribute("href");          ❶
    appHistory.push(href);                               ❷
    e.preventDefault();                                  ❷
  }
});
...
```

❶ adds a click listener to the complete document

❷ only cares about a-tags

❸ extracts the link target from `href`

❹ pushes the new URL to the history

❺ stops the browser from performing a hard navigation

> **NOTE** This is a shortened version of a global click handler. In production, you'd also want to watch for modifier keys to make *opening in a new tab* possible. You might also want to detect external links. But you get the gist.

This click handler intercepts clicks on links that are rendered by the individual micro frontends. Instead of triggering a full page load the browser performs a soft navigation:

- the target URL becomes the latest entry in the history stack (`appHistory.push(href)`)
- the `appHistory.listen(updatePageComponent)` callback triggers
- `updatePageComponent` matches the new URL against the routing table to determine the new component name
- `updatePageComponent` replaces the existing component with the new one
- the `disconnectedCallback` of the old component triggers (if implemented)
- the `constructor` and `connectedCallback` of the new component triggers

When you start the example code and open localhost:3000/, you can see this code in action. Click on the links to navigate between the pages. All page transitions are entirely client-side. The app shell document doesn't reload at any time. Figure 7.5 illustrates the links between the pages in the example project.

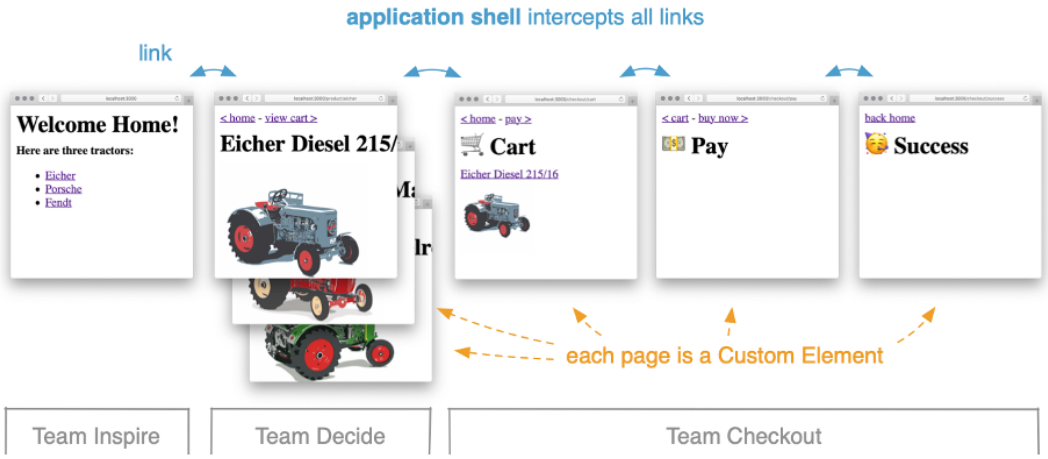


Figure 7.5 The pages in the example project are connected via links. The application shell intercepts these links and performs a soft navigation to the requested page. Teams expose their pages as Custom Elements. On navigation, the app shell replaces the existing page component with the new one.

You should take some time and play around with the code. Add log statements or debugger breakpoints to the app shell and page component code. It gives you a feeling of how our routing code plays together with the (de)initialization of the pages.

7.1.5 Contracts between app shell and teams

Let's take a step back and look at the contracts between the teams and the app shell. Each team needs to publish a list of URLs it's managing. Other teams can use these URLs to link to a specific page. However, these teams don't need to know the component name of a specific team. The application shell encapsulates this information. When a team wants to change the name of a component, it must only update the app shell.

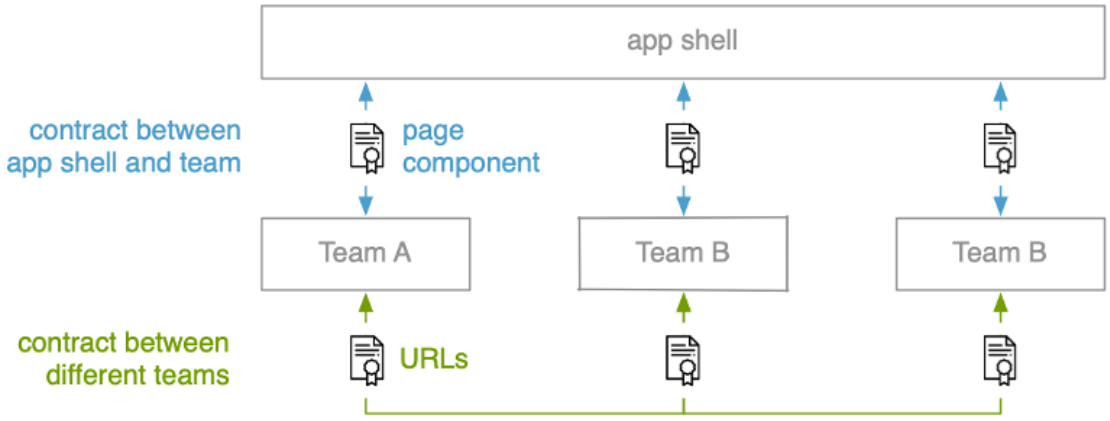


Figure 7.6 Contracts between the systems. Teams need to expose their pages to the app shell in a defined component format (e.g., Web Components). A team needs to know the URL of another team if it wants to link to it.

7.2 App shell with two-level routing

The teams are happy with their first app shell prototype. It required less code than expected. However, they already spotted a significant downside. **The flat routing approach requires that the app shell must know all URLs of the application.** When a team wants to change an existing or add a new URL, they also need to adjust and redeploy the app shell. This coupling between the feature teams and the app shell does not feel right. **The shell should be a piece of infrastructure that's as neutral as possible.** It shouldn't need to know every URL that exists in the application.

The concept of two-level routing circumvents this. Here the app shell only routes between teams. Each team can have its router that maps the incoming URL to a specific page. It's the same concept you learned in 3.2.3 Route configuration methods but moved from the web-server to JavaScript in the browser.

For this to work, the app shell needs a reliable way to tell which team owns a specific URL. The easiest way to achieve this is by using a prefix. Figure 7.7 illustrates how this works.

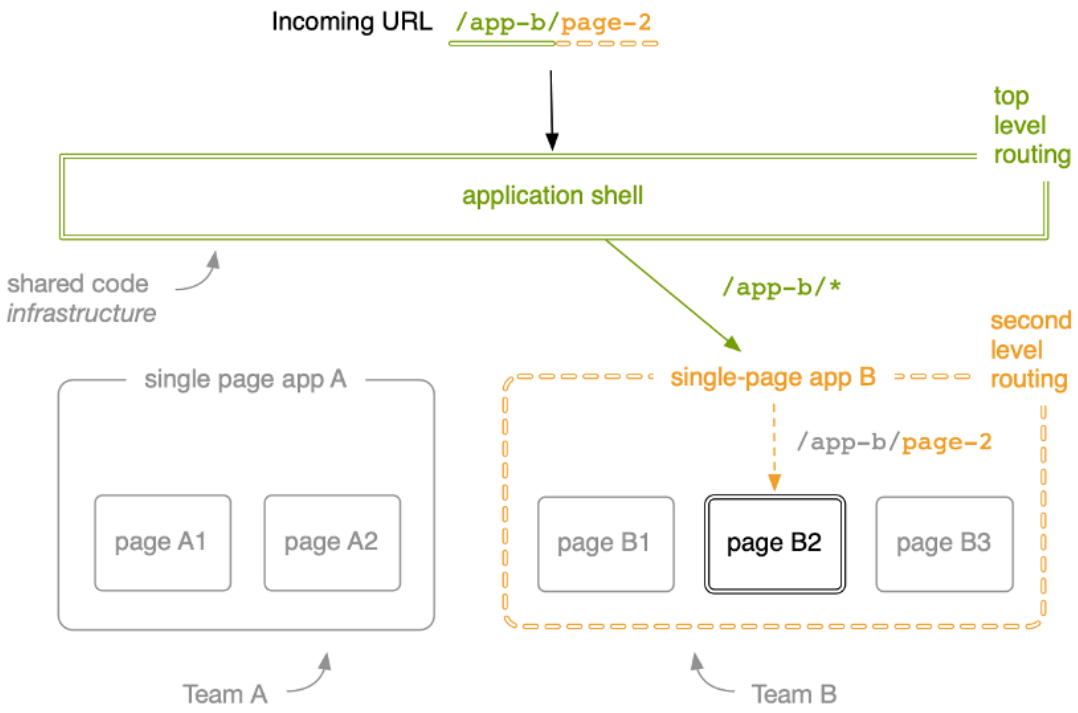


Figure 7.7 Two level routing. The app shell looks at the first part of the URL to determine which team is responsible (top-level routing). The router of the matched team processes the complete URL to find the correct page inside its single-page application (second level routing).

With this model, we have multiple single-page apps (per team) wrapped in another single-page app (app shell). It has the benefit that the routing rules inside the app shell become minimal. Its top-router decides which team is responsible. The actual route definitions move into the responsible team applications. A team can add new URLs inside its application without changing the app shell. They can add it to their router. The app shell only needs to change if you want to introduce a new team or change a team prefix.

Let's go ahead and implement these changes.

7.2.1 Implementing the top-level router

The app shell script can stay the same. We only have to change the routing definitions.

Listing 7.8 app-shell/index.html

```
...
const routes = {                                          ❶
  "/product/": "decide-pages",                            ❷
  "/checkout/": "checkout-pages",                         ❸
  "/": "inspire-pages"                                    ❹
};                                                        ❷

function findComponentName(pathname) {                    ❷
  const prefix = Object.keys(routes).find(key =>          ❷
    pathname.startsWith(key)                              ❷
  );                                                      ❷
  return routes[prefix];                                  ❷
}                                                         ❷
...
```

❶ The routes object now maps a url prefix to a team-level component

❷ To look up a component the function compares the route prefixes against the current pathname. It returns the component name of the first route that matches.

The `routes` object is compacter than before. In the flat routing version, it mapped specific URLs like `/checkout/success` to a page specific component `checkout-success`. The new routing combines all routes of a team in one definition and does not differentiate between pages.

Before, `findComponentName` did a simple object lookup via the pathname. Now it matches the incoming pathname against all prefixes and returns the first component name that matches. All URLs starting with `/checkout/` trigger a render of component `checkout-pages`. It's the job of *Team Checkout* to process the rest of the pathname and show the correct page.

That's it. The other code we saw in the flat routing model before can stay the same.

7.2.2 Implementing team-level routing

Let's look inside the `checkout-pages` component to see the second-level routing. This new component takes the role of the `checkout-cart`, `checkout-pay`, and `checkout-success` components from the previous example. Here is *Team Checkout's* code for handling the pages:

Listing 7.9. checkout/pages.js

```
const routes = {                                                    (1)
  "/checkout/cart": () => `                                         (2)
    <a href="/">&lt; home</a> -                                     (3)
    <a href="/checkout/pay">pay &gt;</a>                            (2)
    <h1>🛒 Cart</h1>                                                (2)
    <a href="/product/eicher">...</a>`,                             (2)
  "/checkout/pay": () => `
    <a href="/checkout/cart">&lt; cart</a> -
    <a href="/checkout/success">buy now &gt;</a>
    <h1>💲 Pay</h1>`,
  "/checkout/success": () => `
    <a href="/">home &gt;</a>
    <h1>✅ Success</h1>`
};

class CheckoutPages extends HTMLElement {
  connectedCallback() {                                             (4)
    this.render(window.location);                                   (5)
    this.unlisten = window.appHistory.listen(location =>            (6)
      this.render(location)                                         (6)
    );                                                              (6)
  }
  render(location) {                                                (7)
    const route = routes[location.pathname];                        (8)
    this.innerHTML = route();                                       (9)
  }
  disconnectedCallback() {                                          (10)
    this.unlisten();                                                (10)
  }                                                                 (10)
}

window.customElements.define("checkout-pages", CheckoutPages);     (11)
```

(1) Contains all of *Team Checkout's* routes.
(2) Maps the URL of the cart page to a templating function.
(3) The template for the cart page.
(4) Triggers when the app shell appends the `<checkout-pages>` component to the DOM.
(5) Renders content based on the current location.
(6) Listens to changes in the history and re-renders on change (notice that we are using the appHistory instance provided by the app shell).
(7) Responsible for rendering the content.
(8) Looks up the page template via the incoming pathname.
(9) Executes the route template and writes the result into `innerHTML`.
(10) Triggers when the app shell removes the component from the DOM and unregisters the before added history listener.
(11) Exposes the component as `checkout-pages` to the global Custom Elements registry.

This code contains the template of all three pages. The `connectedCallback` method triggers

when the app shell appends the component to the DOM. It renders the pages based on the current URL. Then it listens for URL changes (`window.appHistory.listen`).

When a location changes, it updates the view accordingly. For simplicity, we use a simple string-based template. In a real application, you'd probably go for a more sophisticated option.

CLEANUP IS KING

It's always good to clean up after you've finished. However, it is extra vital in this micro frontend setup. Running the app shell model is like sharing an apartment with other people. Global variables, forgotten timers, and event listeners may get in the way of other teams or cause memory leaks. These problems are often hard to track down.

It's essential to do proper cleanup when the component isn't in use anymore to avoid issues. That's why in our example the `disconnectedCallback()` removes the history listener via the `unlisten()` function the `appHistory.listen()` call returned.

Be careful when using third party code. Older jQuery plugins or frameworks like AngularJS (v1) are known for lousy cleanup behavior. However, most modern tools behave well when you unmount them correctly.

That's everything we need to make our two-level routing work. In the first level, the application shell decides which team is responsible. In the second level, the team selects the appropriate page. Take some time and run the example locally for a more in-depth analysis.

```
npm run 14_client_side_two_level_routing
```

7.2.3 What happens on a URL change?

Let's examine what happens when a URL changes. We'll look at three scenarios: first page-load, navigation inside team boundaries, and navigation across boundaries.

SCENARIO 1: FIRST VIEW

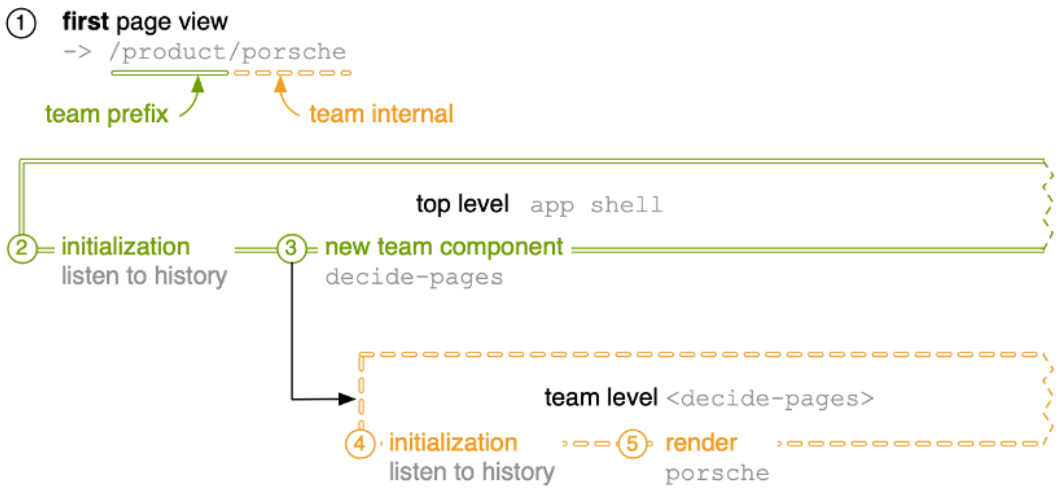


Figure 7.8 First page view in a two-level routing approach. The top-level router looks at the team prefix to determine the responsible team. The team router at the second level looks at the last part of the URL to render the actual page.

Figure 7.8 shows the first page load.

1. The app shell code runs first and does everything needed for initialization. It starts watching the URL for changes.
2. The current URL starts with the team prefix `/product/`. This prefix maps to *Team Decides* `<decide-pages>` element. The app shell inserts this component to the DOM.
3. The team level component initializes itself. It also starts listening to the URL.
4. It looks at the current URL and renders the product page for the Porsche tractor.

In short - the app shell picks the team that's responsible for the current URL, and this team renders the page. Both have registered a listener to the URL. In the next scenarios, we'll see the listeners in action.

SCENARIO 2: INSIDE TEAM NAVIGATION

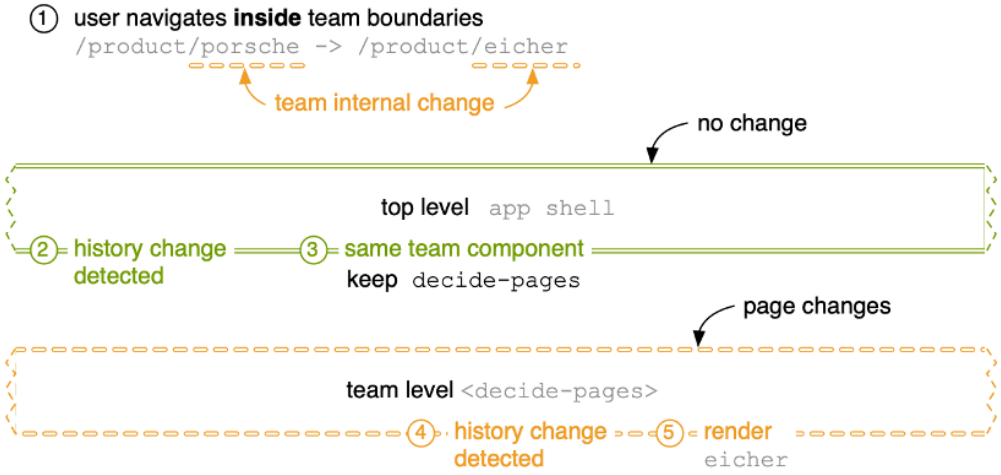


Figure 7.9 When the user navigates to another page controlled by the same team, the app shell has nothing to do. The team level component needs to update the page according to the URL.

Figure 7.9 shows what happens when the user is on page `/product/porsche` and clicks on a link to `/product/eicher`. The app shell intercepts the link and pushes the new URL to the front of the history.

1. The app shell detects a history change and notices that the team prefix did not change.
2. The team level component can stay the same. The app-shell has nothing to do.
3. The team component registers the URL change too.
4. It updates its content and switches from the Porsche to the Eicher tractor.

Since team responsibility did not change (same team prefix), the app shell has nothing to do. *Team Decides* handles the page change on its own.

Now to the exciting part: inter-team navigation.

SCENARIO 3: INTER-TEAM NAVIGATION

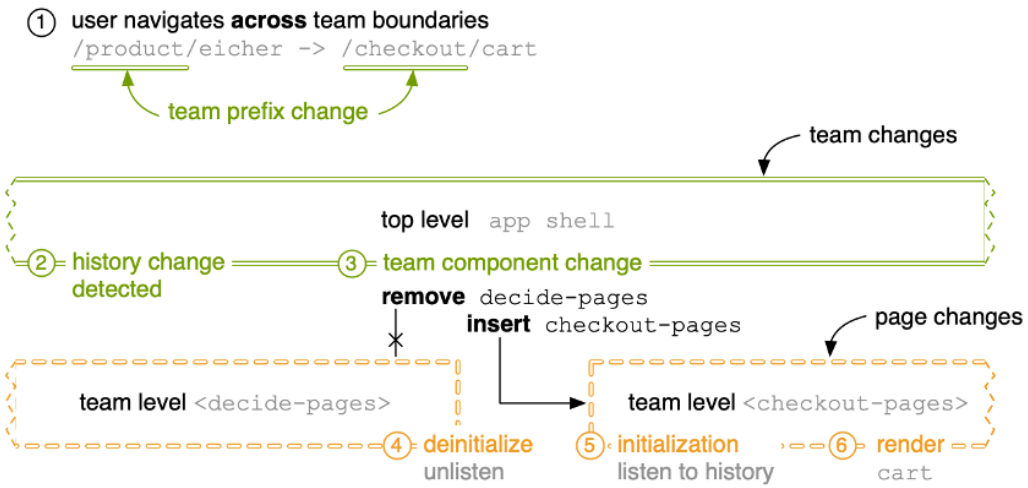


Figure 7.10 On an inter-team navigation, the responsibility changes. The app shell replaces the existing team component with a new one. This new component takes over and is in charge of rendering the page.

When the user moves from the product page to the checkout page, he crosses a team boundary. *Team Checkout* owns the cart page. In figure 7.10 you see how the app shell handles this transition.

1. The app shell recognizes a change in history.
2. Since the team prefix changed from `/product/` to `/checkout/`, the app shell replaces the existing `<decide-pages>` component with the new `<checkout-pages>` component.
3. *Team Decide* receives the request to deinitialize itself before the app shell removes it from the DOM. It cleans up behind itself and stops listening for history events.
4. *Team Checkout's* component initializes itself and starts listening to the history.
5. It renders the cart page.

In this scenario, the app shell swaps the team level components. Thereby it hands over control from one team to another. The team components deal with their initialization and deinitialization.

7.2.4 App shell APIs

You've learned about the app shells most essential tasks:

- Loading the team's application code
- Routing between them based on the URL

Here is a list of additional topics that an app shell might be responsible for:

- Context information (like language, country, tenant)

- Meta-data handling (updating tag, crawler hints, semantic data)
- Authentication
- Polyfills
- Analytics & Tag-Managers
- JavaScript error reporting
- Performance monitoring

Some of these functionalities are not interesting to the application code. Performance monitoring is, for example, often done by adding a script in a central place. The monitoring can work without the applications knowing about it.

However, other functionalities can require interaction between the app shell and the applications. The following code shows how a function for tracking events from inside an application could look.

```
window.appShell.analytics({ event: "order_placed" });
```

The app shell can also pass information to the applications. In a web component based model it can look like this:

```
<inspire-pages country="CH" language="de"></inspire-pages>
```

It's a good idea to keep this API as lean as possible. Having a stable interface reduces friction. The rollout of breaking API changes comes with inter-team coordination. All teams need to update their code to keep functioning correctly.

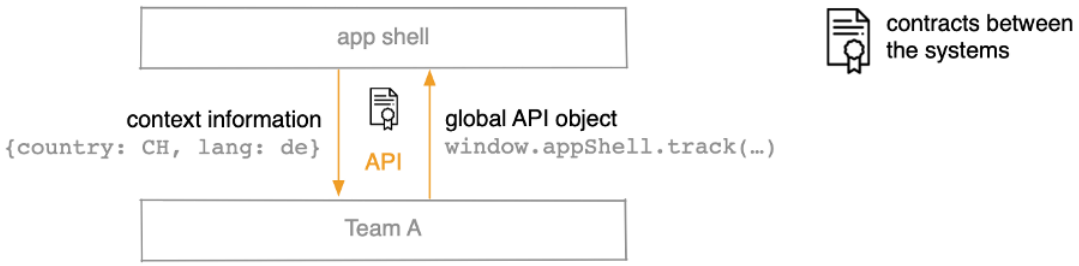


Figure 7.11 Adding shared functionality to the app shell leads to tighter coupling. The API between app shell and team applications acts as a contract between the systems. It should be lean and stable.

Business logic should be in the teams' application code - not in the shared application shell. A good indicator for too tight coupling is this: **A feature deployment from a team should not require the app shell to change.**

Now we've created a minimal application shell from scratch. Next up, we'll have a quick look into an existing and ready-to-use solution: single-spa.

7.3 A quick look into the single-spa meta-framework

After building and evolving the app shell prototype, *Tractor Models Inc.'s* development teams have a pretty good understanding of how the pieces work together. They know for sure that they want to go with the two-level routing model. However, there are still some features missing. Lazy loading of JavaScript code and proper error handling are two of them.

To avoid reinventing the wheel, they check for existing solutions that fit their needs. They come across single-spa.[59], which is a popular micro frontends meta-framework. In essence, it's an application shell - similar to the application shell we just built. But it comes with some more advanced features. It has built-in **on-demand loading of application code** and comes with a **broad ecosystem of framework bindings**. You can find examples and helper libraries to hook up a React, Vue.js, Angular, Svelte, or Cycle.js application with few efforts. These make it easy to expose an application in a unified way so that single-spa can interact with them.

The teams want to take their prototype and migrate it to single-spa. To test out the limits, each team chooses another JavaScript framework for their part of the shop. *Team Inspire* implements the Homepage using Svelte.js, *Team Decide* renders the product pages using React, and *Team Checkout* opted for the Vue.js framework. Figure 7.12 illustrates this. They don't plan to go to production with this technology mix, but it's an excellent exercise to see how the integration works.

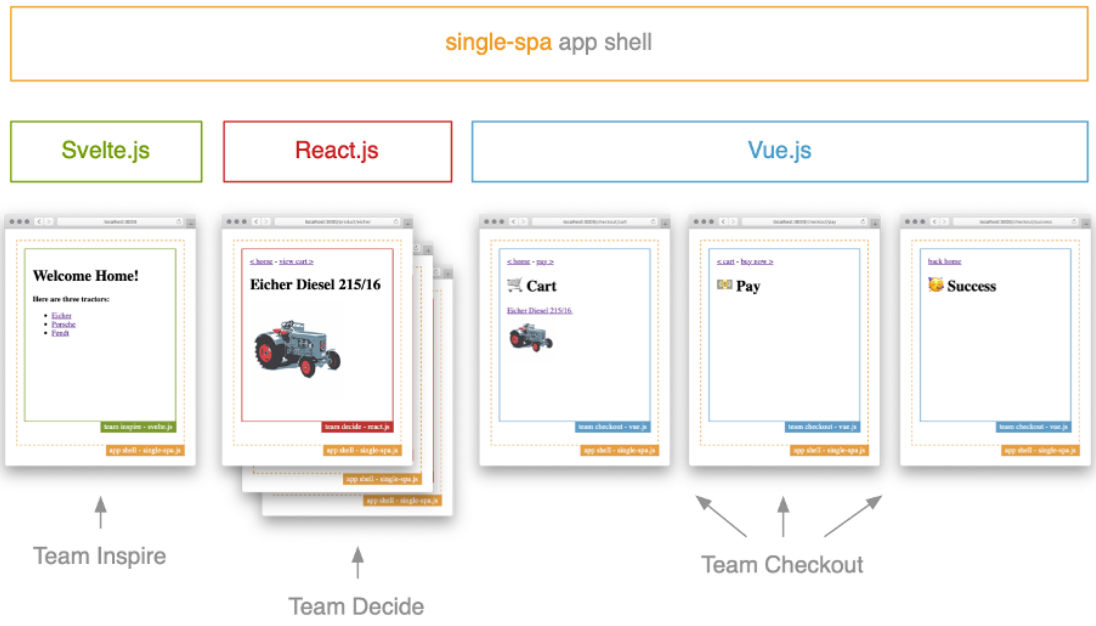


Figure 7.12 Single-spa acts as the application shell which routes between the applications. In our example, all teams have picked a different frontend framework for their application code.

Let's have a look at how single-spa works.

7.3.1 How single-spa works

> **TIP** You can find the sample code for this task in the `15_single_spa` folder.

The basic concepts are the same as in our previous prototype. We have a single HTML file that acts as the starting point. It includes the single-spa JavaScript code and maps URL prefixes to the code of a specific application. The main difference is that it does not use Web Components as the component format. Instead, the teams expose their micro frontends as a JavaScript object that adheres to a specific interface. We'll look at this in a minute. Let's look at the initialization code first.

Listing 7.10 app-shell/index.html

```
<html>
  <head>
    <title>The Tractor Store</title>
    <script src="/single-spa.js"></script>                    ❶
  </head>
  <body>
    <div id="app-inspire"></div>                              ❷
    <div id="app-decide"></div>                               ❷
    <div id="app-checkout"></div>                             ❷

    <script type="module">
      singleSpa.registerApplication(                          ❸
        "inspire",                                            ❹
        () => import("http://localhost:3002/pages.min.js"),   ❺
        ({ pathname }) => pathname === "/"                    ❻
      );
      singleSpa.registerApplication(
        "decide",
        () => import("http://localhost:3001/pages.min.js"),
        ({ pathname }) => pathname.startsWith("/product/")
      );
      singleSpa.registerApplication(
        "checkout",
        () => import("http://localhost:3003/pages.min.js"),
        ({ pathname }) => pathname.startsWith("/checkout/")
      );
      singleSpa.start();                                      ❼
    </script>
  </body>
</html>
```

❶ imports the single-spa library

❷ each micro frontend has it's own DOM element which acts as the mount point

❸ registers a micro frontend with single-spa

❹ name of the applications which makes debugging easier

❺ loading function for the applications which fetches the associated JavaScript code when needed

❻ the activity function receives the location and determines if the micro frontend should be active or not

❼ initializes single-spa, renders the first page and starts listening for history changes

In this example, the `single-spa.js` library gets included globally. Notice that you have to create a DOM element for every micro frontend (`<div id="app-inspire"></div>`). The application code of the micro frontend looks for this element in the DOM and mounts itself underneath this element.

The `singleSpa.registerApplication` function maps the application code to a specific URL. It takes three parameters:

- `name` must be a unique string which makes debugging easier
- `loadingFn` returns a promise that loads the application code. We are using the native `import()` function in the example.
- `activityFn` gets called on every URL change and receives the `location`. When it returns true, the micro frontend should be active.

On start, single-spa matches the current URL against all registered micro frontends. It calls their activity functions to detect which micro frontends should be active. When an application becomes active for the first time, single-spa fetches the associated JavaScript code through the loading function and initializes it. When an active application becomes inactive, single-spa calls its `unmount` function, instructing it to uninitialize itself.

More than one application may be active at the same time. A typical use-case for this is the global navigation. It can be a dedicated micro frontend that gets mounted at the top and is active on all routes.

JAVASCRIPT MODULES AS THE COMPONENT FORMAT

In contrast to our Web Component based prototype, single-spa uses a JavaScript interface as the contract between app shell and team application. An application has to provide three asynchronous functions. It looks like this:

Listing 7.11 team-a/pages.js

```
export async function bootstrap() {...}
export async function mount() {...}
export async function unmount() {...}
```

These functions (`bootstrap`, `mount`, `unmount`) are similar to the Custom Elements lifecycle functions (`constructor`, `connectedCallback`, `disconnectedCallback`). Single-spa calls `bootstrap` when a micro frontend becomes active for the first time. It invokes `(un)mount` every time the application is (de)activated.

All lifecycle functions are asynchronous. This fact makes lazy loading and data-fetching inside an application a lot easier. Single-spa ensures that `mount` is not called before `bootstrap` has completed.

The Custom Elements lifecycle methods are synchronous. Implementing asynchronous initialization with Custom Elements is possible. However, it requires some extra work on top of what the standard specifies.

FRAMEWORK ADAPTERS

Single-spa comes with a list of framework adapters. Their job is to wire the three lifecycle methods to the appropriate framework calls for (de)initialization. Let's have a look at the code for *Team Inspire*, which delivers the homepage. They've chosen the framework Svelte.js. Don't worry if you've never Svelte before. It's a simple example.

Listing 7.12 team-decide/pages.js

```
import singleSpaSvelte from "single-spa-svelte";                    ❶
import Homepage from "./Homepage.svelte";                           ❷

const svelteLifecycles = singleSpaSvelte({                          ❸
  component: Homepage,                                              ❸
  domElementGetter: () => document.getElementById("app-inspire")    ❸
});                                                                 ❸

export const { bootstrap, mount, unmount } = svelteLifecycles;      ❹
```

❶ import single-spa's Svelte adapter

❷ import the Svelte component for rendering the homepage

❸ call the adapter with the root component and a function that retrieves the DOM element to render it in

❹ exports the lifecycle functions returned by the adapter call

First, we import the adapter library `single-spa-svelte` and the `Homepage.svelte` component containing the actual template. We'll look at the homepage code in a second. The adapter function `singleSpaSvelte` receives a configuration object with two parameters: the root component and a function that looks up *Team Inspire's* DOM element. The adapters have different parameters that are specific to the associated framework. In the end, we export the lifecycle methods returned by the adapter function.

NOTE In the example code, each team has a Rollup-based build process to generate the `pages.min.js` file in the ES module format. However, there is nothing Rollup specific. You can do the same with Webpack or Gulp.

NAVIGATING BETWEEN MICRO FRONTENDS

Let's have a look at the homepage component:

Listing 7.13 team-decide/Homepage.svelte

```
<script>
  function navigate(e) {                                    ❶
    e.preventDefault();                                     ❷
    const href = e.target.getAttribute("href");             ❶
    window.history.pushState(null, null, href);             ❶
  }                                                         ❶
</script>

<div>
  <pre>team inspire - svelte.js</pre>
  <h1>Welcome Home!</h1>
  <strong>Here are three tractors:</strong>
  <a on:click={navigate} href="/product/eicher">Eicher</a>      ❷
  <a on:click={navigate} href="/product/porsche">Porsche</a>    ❷
  <a on:click={navigate} href="/product/fendt">Fendt</a>        ❷
</div>
```

❶ function that intercepts link clicks by pushing the URL to the history and preventing a reload

❷ links to *Team Decide's* product page

In the above example, you see three links to product pages. They have a `navigate` click handler attached that prevents hard navigation (`e.preventDefault()`) and writes the URL to the native history API instead (`window.history.pushState`). Single-spa monitors the history and updates the micro frontends accordingly.

A click on this product link would trigger the termination (`unmount`) of *Team Inspire's* micro frontend. After that, single-spa loads *Team Decide's* application and activates it (`mount`). This behavior is similar to the inter-team navigation scenario you saw in the previous section.

RUNNING THE APPLICATION

You can fire up the sample code by running the following command:

```
npm run 15_single_spa
```

It starts four webservers (app-shell & three applications) and opens your browser at localhost:3000/. Have a look at the developer tools when navigating through the shop using the links.

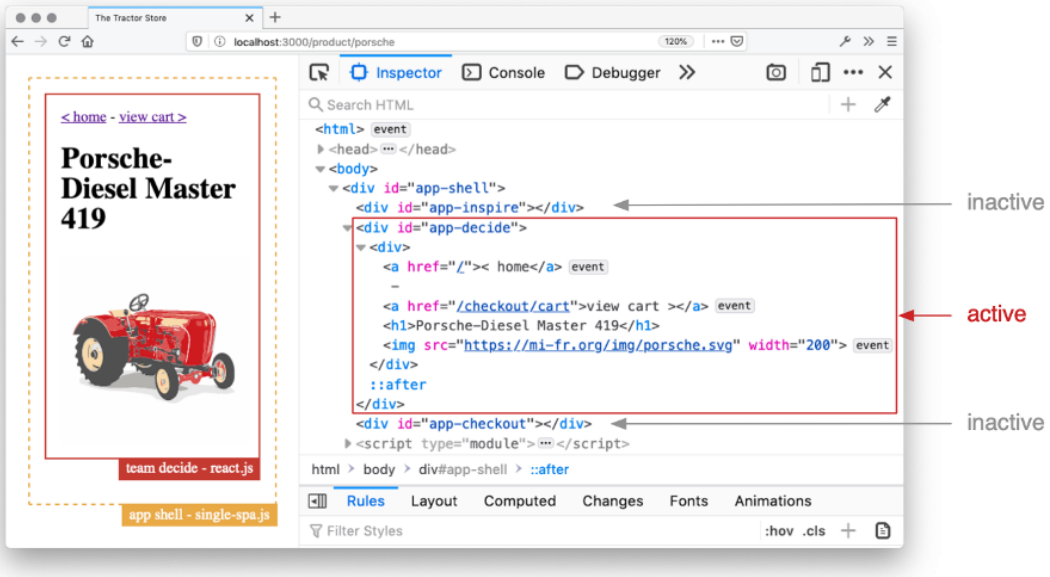


Figure 7.13 With single-spa each micro frontend has it's own DOM node to render its content. In this example, the micro frontend for Team Decide's product page is active. It shows it's content inside the #app-decide element. The other micro frontends are inactive, and their corresponding DOM elements are empty.

- See how the `#app-inspire`, `#app-decide`, and `#app-checkout` DOM nodes of the app shell get to life. When the user moves from one micro frontend to the next, the content changes. The old micro frontend removes its markup. The new micro frontend fills its DOM node with the new content. You can see this in figure 7.13.
- Open the network tab and also notice that single-spa loads the JavaScript bundles (`pages.min.js`) as they are needed and not all up front.
- Have a look at the code of *Team Decide's* and *Team Checkout's* micro frontends. They both include a framework level router (`react-router` & `vue-router`). The application code is not special. It's straight from the respective "Getting started" guides. Client-side navigation works via the stock `<Link>`- and `<router-link />` components from the routers.

NESTING MICRO FRONTENDS

In our current example, there is precisely one micro frontend active at a time. The app shell instantiates the micro frontends at the top level. One micro frontend is active at a time. This is how single-spa gets used most often. As said before, it's possible to implement some navigation micro frontend that is always present and sits next to the other applications. However, single-spa also allows nesting. This concept goes by the name *Portals*. Portals are pretty much the same as what we've called fragments in the last chapters.

DIVING DEEPER INTO SINGLE-SPA

You've seen the underlying mechanisms of single-spa. It offers more functionality than we've covered here. Besides the mentioned portals, there are status events, the ability to pass down context information, and ways to deal with errors.

The official documentation.[60] is an excellent place to start to get deeper into single-spa. They have a lot of good examples showcasing how to use single-spa with different frameworks.

7.4 The challenges of a unified single-page app

Now you have a good understanding of what's necessary to build an app shell that connects different single-page apps. The unified model makes it possible for the user to move through the complete app without encountering a hard navigation. All page transitions are client rendered, which in general results in quicker responses to user interactions.

7.4.1 Topics you need to think about

However, the improvement in user experience does not come for free. Here are a couple of topics you need to address when going the unified single-page app route.

SHARED HTML DOCUMENT AND META DATA

The teams have no control over the surrounding HTML document. A micro frontend may only change the content inside of its root DOM node in the body.

You most always want to set a meaningful `title` for the individual pages. Providing a global `appShell.setTitle()` method would be one way of dealing with this. Each micro frontend could also directly alter the `head` section via DOM API.

However, if your site is accessible on the open web setting, the `title` is often not enough. You want to provide crawlers and preview generators like Facebook or Slack with machine-readable information like canonicals, href-langs, schema.org-tags, and indexing hints. Some of these might be the same for the complete site. Others are highly specific to one page-type.

Coming up with a mechanism to effectively manage meta tags across all micro frontends comes with some extra work and complexity. Think of Angular's meta-service.[61], vue-meta.[62] or react-helmet.[63] but on an app shell level.

ERROR BOUNDARIES

If the code of different teams runs inside one document, it can sometimes be tricky to find out where an error originated. In the composition approach from the last chapter, we have the same problem. Code from inside a fragment has the potential to cause unwanted behavior on the complete page. However, the unified single-page app model widens the debugging area from page level to the complete application. A forgotten scroll listener from the homepage can introduce a bug on the confirmation page in the checkout. Since these pages are not owned by the same team, it can be hard to make the connection when looking for the error.

In practice, these types of problems are rather rare. Also, error reporting and browser debugging tools have gotten pretty good over the last years. Identifying which JavaScript file caused the error helps in finding the responsible team.

MEMORY MANAGEMENT

Finding memory leaks is more complicated than tracking back a JavaScript error. A common cause for memory leaks is inadequate cleanup: removing parts of the DOM without unregistering event listeners or writing something to a global location and then forgetting about it. Since the micro frontend applications get initialized and deinitialized regularly, even smaller problems in cleanup can accumulate to a bigger problem.

Single-spa has a plugin called `single-spa-leaked-globals` which tries to clean up global variables after a micro frontend unmounted. However, there is no universal magic cleanup solution. It's essential to raise awareness in your developer teams that proper unmounting is as important as proper mounting.

SINGLE POINT OF FAILURE

The app shell is, by nature, the single first point of contact. Having a severe error in the app shell can bring down the complete applications. That's why your app shell code should be of high quality and well tested. **Keeping it focused and lean** helps in achieving this.

COMMUNICATION

Sometimes micro frontend A needs to know something that happened in micro frontend B. The same communication rules we discussed in chapter 6. Communication Patterns also apply here:

- avoid inter-team communication when possible
- transport context information via the URL
- stick to simple notifications when needed
- prefer API communication to your backend

Don't move state to the app shell. It might sound like a good idea to not load the same information twice from the server. However, misusing the app shell as a state container creates

strong coupling between the micro frontends. In the backend world, it's a best practice that microservices don't share a database. One change in a central database table has the potential to break another service. The same applies to micro frontends. Here your state container is equivalent to a database.

BOOT TIME

Code splitting has become best practice in web development. When implementing an app shell, you should consider this as well. In the single-spa example, you saw how the library loads the actual micro frontend code on-demand. It's crucial to think about optimizations to deliver an excellent overall performance.

7.4.2 When does a unified single-page app make sense?

This model plays it's strength when **the user needs to switch frequently between user interfaces owned by different teams**. In e-commerce, the jump between the search result and the product details page is a good example. The user looks at a list of products, clicks on one, jumps back to the list and repeats the process until he finds something he likes. In this case, using a soft navigation makes a noticeable difference in the user experience.

For web applications **where providing a high amount of interactivity is more important than initial page load time**, the unified single-page app approach is a good fit. Sites that require the user to log in before using it and classical back-office applications are prime candidates.

However, as already discussed, this approach does not come for free and introduces a considerable amount of shared complexity. If you want to split your existing single-page application into smaller ones, the unified single-page application approach is not necessarily the way to go. For many use-cases, it's totally fine to have a hard navigation between two linked single-page apps.

Imagine a content management application with an area for writing long-form articles and another area to moderate comments. These can be two independent single-page applications. Since a typical user would not always switch from moderating comments to writing an article, it might be perfectly fine to build this as two distinct applications that both include the same header fragment via composition.

Figure 7.14 shows the tradeoff between providing the best user experience and having a simple setup with low coupling.

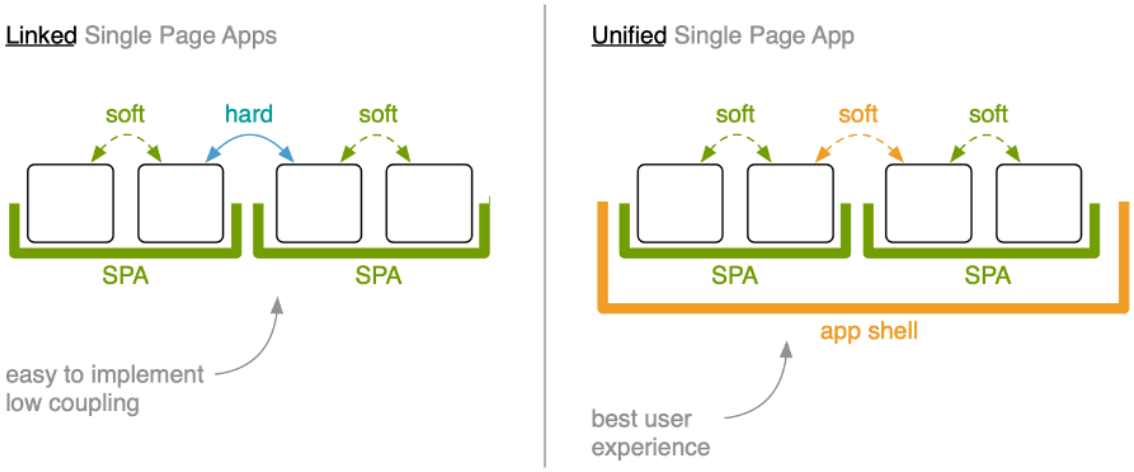


Figure 7.14 Linked single-page apps are easy to build and introduce low coupling. However, they require a hard navigation when moving from one app to the other. The unified single-page app approach solves this and provides a better user experience. But this enhancement does come with some major complexity.

As always, there are no right or wrong solutions. Both models have their benefits. Let's close this chapter by placing the unified single-page app model into the comparison chart we've built over the last chapters. See the result in figure 3.3.

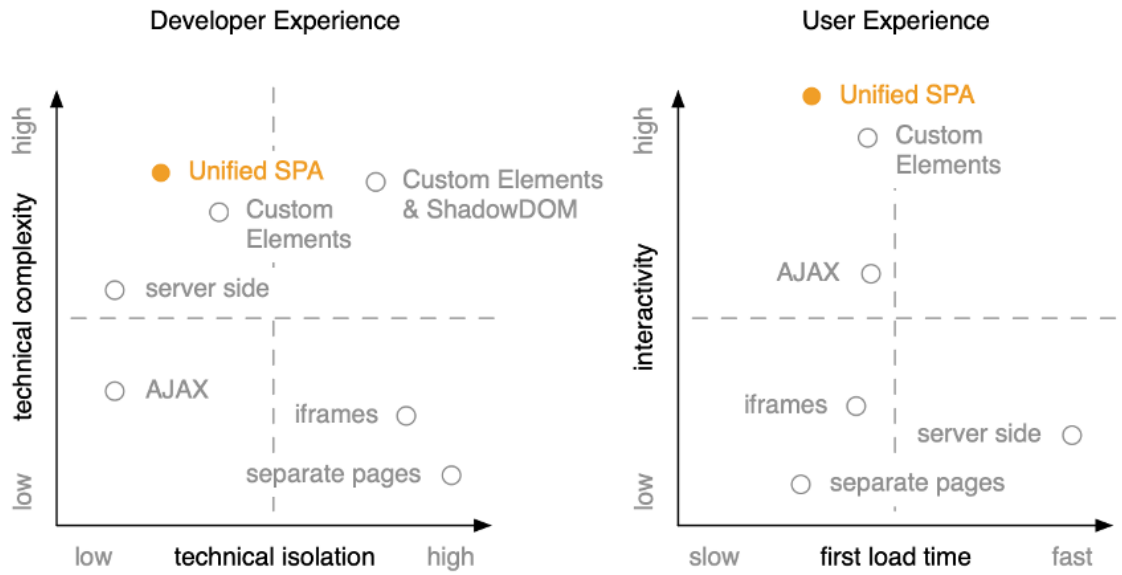


Figure 7.15 Setting up and running a unified single-page app in production is not trivial. Existing libraries like single-spa make it easy to get started. Since all application code lives in the same HTML document, there is no technical isolation. We also have the risk that an error in app A can affect app B. Since a unified single-page application is client rendered and needs additional app shell code, it has a higher startup time. However, if your goal is to create a product with a perfect user experience, the unified single-page approach is the way to go.

7.5 Summary

- Combining multiple single-page apps requires a share app shell that handles routing.
- This approach makes it possible to use soft navigations across all pages.
- The app shell is a shared piece of infrastructure and should not contain business logic.
- Deploying a team feature should never require an app shell deployment.
- Having a two-level routing approach where the app shell performs a simple team match and the team's single-page determines the actual page is a useful model for keeping the app shell lean.
- Teams must expose their single-page applications in a framework-agnostic component format. Web Components are a great fit for this. But you can also use a custom interface like single-spa does.
- It might be necessary to establish additional APIs between the app shell and the application. Analytics, authentication, or meta-data handling are popular reasons for this. These APIs introduce new coupling. Keep them as simple as possible.
- With this approach, all applications must deinitialize and clean up correctly. Otherwise, you risk running into memory leaks and unexpected errors.

8 Composition & Universal Rendering

This chapter covers

- Employing universal rendering in a micro frontends architecture.
- Applying server- and client-side composition in tandem to combine their benefits.
- Discovering how to leverage the server-side rendering (SSR) capabilities of modern JavaScript frameworks in a micro frontends context.

In the last chapters, we focused on various integration techniques and discussed their strengths and weaknesses. We grouped them into two categories: server- and client-side. Integration on the server makes it possible to *ship a page that loads fast* and adheres to the principals of *progressive enhancement*. Client-side integration enables *building rich user interfaces* where the page can *react to user input instantly*.

Broad framework support for *universal rendering* made building applications that run server- and client-side a lot easier for developers. But what do we need to do to integrate multiple universal applications into a big one?

SIDEBAR **Terminology: Universal, Isomorphic & SSR**

The terms *Universal Rendering* .[64], *Isomorphic JavaScript* .[65] and *Server-side Rendering (SSR)* essentially refer to the same concept: Having a single code-base that makes it possible to render and update markup on the server and in the browser. Their meaning or perspective varies in detail. However, in this book, we'll go with the term *universal rendering*.

You've already acquired the necessary building blocks. We can combine the client- and server-side composition and routing techniques from the last chapters to make this happen. Figure 8.1 illustrates how our puzzle pieces fit together.

NOTE In this chapter, we assume that you're already familiar with the concept of universal rendering and know about *hydration*. If not I can recommend checking out the book "Isomorphic Web Applications".[66]

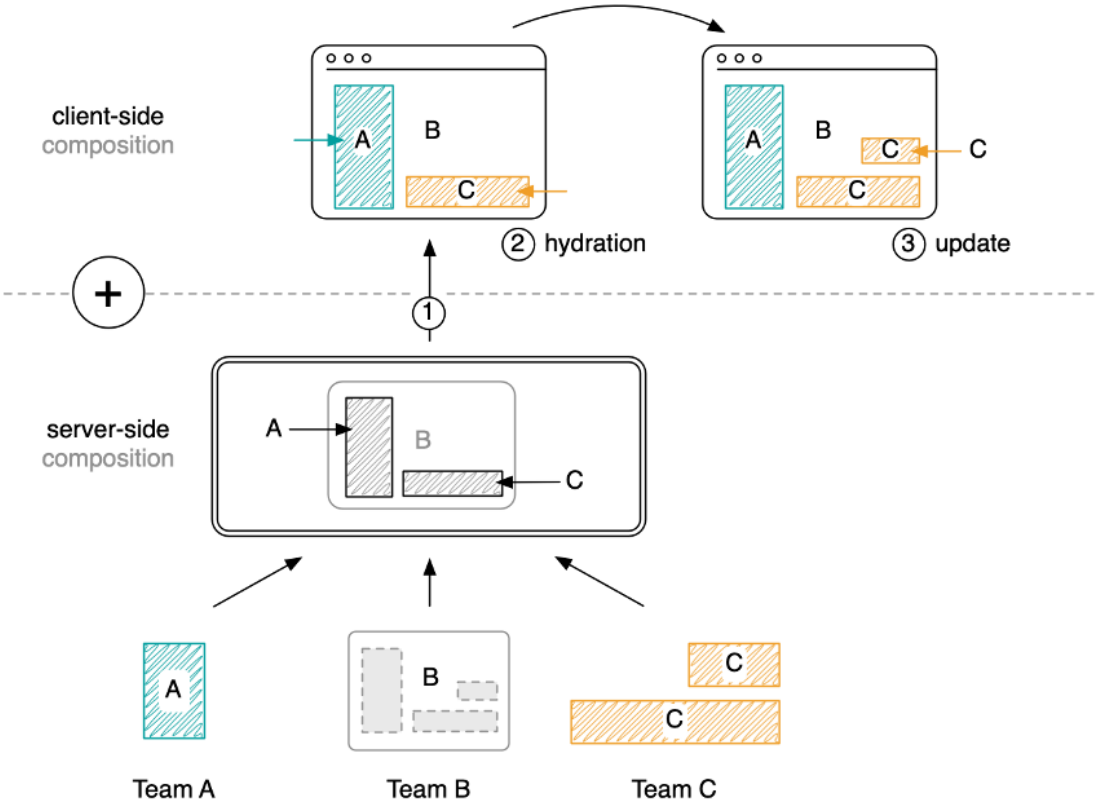


Figure 8.1 Universal composition is the combination of a server- and a client-side composition technique. For the first request, a technique like SSI, ESI, or Podium assembles the markup of all micro frontends server-side. The complete HTML document gets sent to the browser (1). In the browser, each micro frontend hydrates itself and becomes interactive (2). From there on, all user interactions can happen fully client-side. The micro frontends update the markup directly in the browser (2).

In this chapter, we'll upgrade our product detail page. We'll implement universal rendering for all micro frontends and than apply the required integration techniques to make the site work as a whole.

8.1 Combining server- and client-side composition

Since *Team Decide* added *Team Checkout's* buy button to the product page, tractor sales skyrocketed. Now hundreds of orders from all over the world arrive every hour. The team behind *The Tractor Store* was pretty overwhelmed by this success. The had to ramp up their production and logistics capabilities to keep up with the demand. But not everything has been rosy since then. Over the last weeks, the development teams struggled with some serious issues. One day *Team Checkout* shipped a release of their software, which triggered a JavaScript error in all Microsoft Edge browsers. Due to this bug, the buy button was missing on the page. Sales for that day were down by 34%. This incident showed a significant quality issue, and the team took measures so that this kind of problem wouldn't strike again.

But this is not the only problem. The product page integrates the buy button micro frontend using client-side composition via Web Components. The buy button is not part of the initial markup. Client-side JavaScript renders it. For the time loading, the user looks at an empty spot where the buy button will come in with a delay. In local development, this delay is not noticeable. But in the real world, on lower-end smartphones and non-optimal network conditions, it takes a considerable amount of time. Adding new features to the buy button made this effect even worse. You can see how the product page looks when JavaScript fails or hasn't finished loading yet.

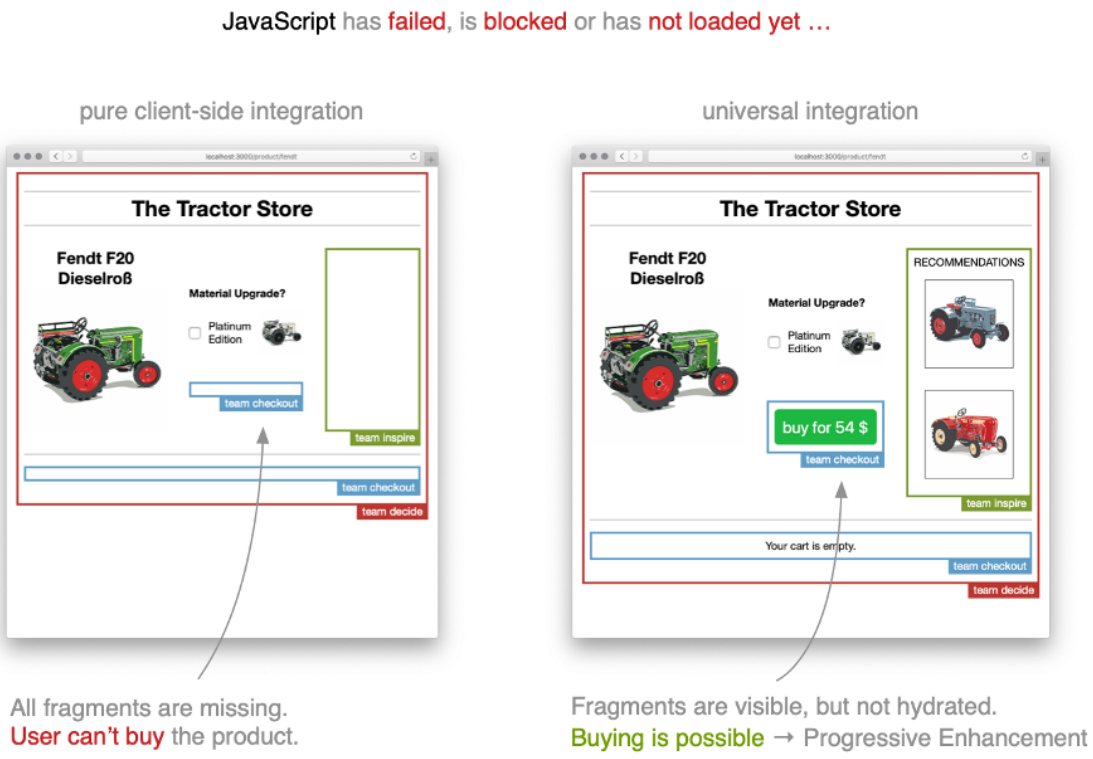


Figure 8.2 Client-side composition requires JavaScript to work. If it fails or takes a long time to load the included micro frontends are not shown. For the product page, this means that the user can't buy a tractor. Universal composition makes it possible to use progressive enhancement in a micro frontends context. That way, the buy button can render instantly, and you can make it function even without JavaScript.

The teams decide to switch to a hybrid integration model. Using SSI for server-side composition and also keeping the Web Components composition. This way, the first-page load can be fast, and client-side updating and communication is still possible. Let's have a look at this combination.

8.1.1 SSI & Web Components

In chapter 5, *Team Checkout* wrapped its buy-button micro frontend into a Custom Element. The browser receives this HTML markup:

Listing 8.1 team-decide/product/fendt.html

```
...
<checkout-buy sku="fendt"></checkout-buy>
...
```

Since `checkout-buy` is a custom HTML tag, the browser treats it as an empty inline element. At first, the user sees nothing. Client-side JavaScript creates the actual content (a button with a price) and renders it as a child. Then the final DOM structure in the browser looks like this:

```
...
<checkout-buy sku="fendt">
  <button type="button">buy for 54 $</button>
</checkout-buy>
...
```

It would be great if we could ship the `button` content already with the initial markup. Sadly Web Components don't have a standard way to render server-side.[67]

> **TIP** You can find the sample code for this task in the `16_universal` folder. It essentially combines the example code from `05_ssi` with `08_web_components`.

Since there is no standard way of doing it, we need to be creative. In this example, we will use the SSI technique you learned in chapter 4 for adding server-side composition to the Web Components approach. This way, we prepopulate the Web Components' internal markup. Figure 8.3 shows our folder structure.

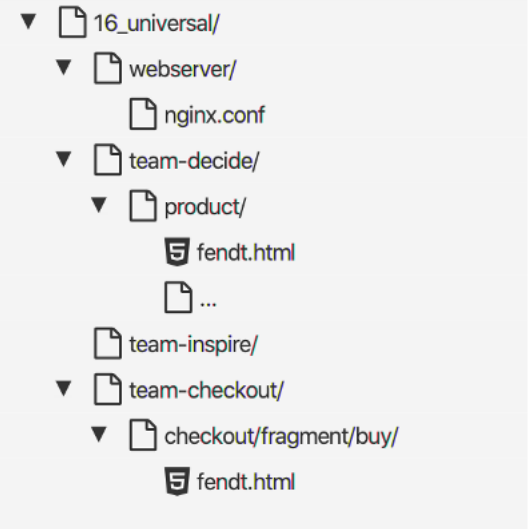


Figure 8.3 Nginx acts as the shared frontend-proxy and handles the markup composition on the server-side. Note that this is an excerpt of the complete folder structure.

Team Decide adds an SSI directive as the child to the buy buttons Custom Element.

Listing 8.2 team-decide/product/fendt.html

```
...
<checkout-buy sku="fendt">                                    (1)
  <!--#include virtual="/checkout/fragment/buy/fendt" -->     (1)
</checkout-buy>
...
```

(1) Client-side Custom Element definition owned by *Team Checkout*. The associated code runs in the browser and renders/hydrates the micro frontend.

❷ Nginx replaces this SSI directive with the content that's returned by the endpoint specified in `virtual`. *Team Checkout* owns this endpoint.

The above code of *Team Decide's* product page now combines client- and server-side composition. The Nginx web-server replaces the SSI directive with the `<button>` markup, which *Team Checkout* generates when calling the `/checkout/fragment/buy/fendt` endpoint. Our example simulates this by serving a static HTML file:

Listing 8.3 team-checkout/fragment/buy/fendt.html

```
<button type="button">buy for 54 $</button>
```

In practice, you'd use a library with server-rendering capabilities to dynamically generate a response in a Node.js environment. For a React based application you'd call `ReactDOMServer.renderToString(<CheckoutBuy />)` and return it's result. Here `<CheckoutBuy />` would be the React-based micro frontend application.

The assembled product page markup that reaches the browser looks like this:

```
...
<checkout-buy sku="fendt">
  <button type="button">buy for 54 $</button>     ❶
</checkout-buy>
...
```

❶ Nginx replaced the SSI directive with the actual content.

The browser is now able to show the button instantly. The associated Custom Element code runs when the JavaScript finishes loading. It hydrates the micro frontend - making sure that the markup is correct and attaching events for further interaction.

Team Checkout's client-side code for the buy button looks like this:

Listing 8.4 team-checkout/checkout/static/fragment.js

```
const prices = {
  porsche: 66,
  fendt: 54,
  eicher: 58
};

class CheckoutBuy extends HTMLElement {
  connectedCallback() {
    const sku = this.getAttribute("sku");
    this.innerHTML = `                                                    ❶
      <button type="button">buy for ${prices[sku]} $</button>             ❶
    `;                                                                    ❶
    this.querySelector("button").addEventListener("click", () => {       ❶
      ...                                                                 ❷
    });                                                                   ❶
  }
  ...
}
window.customElements.define("checkout-buy", CheckoutBuy);
...
```

❶ Renders the markup client-side. This is a "dumb" implementation which replaces all existing markup even if it might already be correct. In a real application, you'd use something more clever and performant like DOM-diffing.

❷ Adding event listeners to be able to react to user input.

The code is identical to the examples we've used in chapter 5. The component renders it's internal markup inside itself and attaches all required event handlers. We again use a simplified implementation here. No client-server code reuse, no DOM diffing. But you get the picture.

When you use something like React, this is the place where you'd call `ReactDOM.hydrate(<CheckoutBuy />, this)`, where `<CheckoutBuy />` is the React application for the button and `this` is the reference to the Custom Element. The call instructs the framework to pick up the existing server-generated markup and hydrate it.

Figure 8.4 shows the complete process we went through. Starting with the server-side markup generation at the bottom and ending with the initialization of the buy button's Custom Element in the DOM.

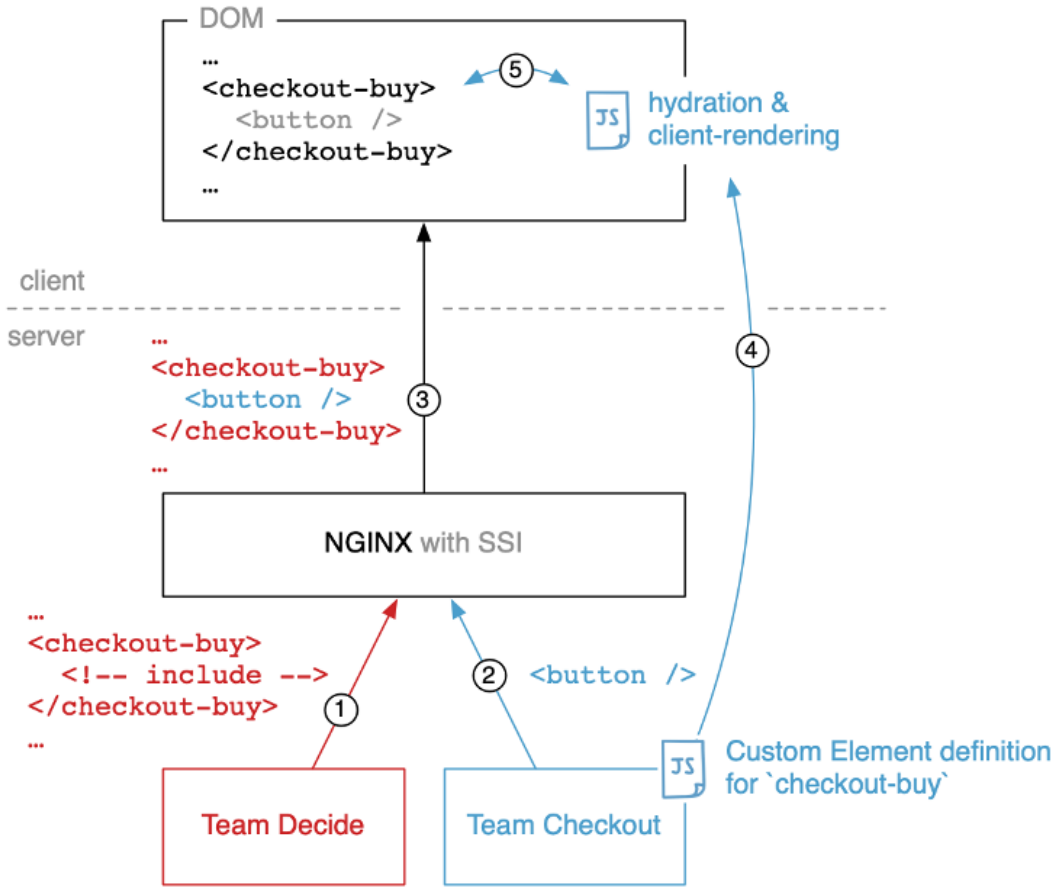


Figure 8.4 Prerendering the contents of a Web Component based micro frontend using SSI. The markup of Team Decide's product page contains a Custom Element for Team Checkout's buy button. It has an SSI include directive as it's content (1). Nginx replaces the include directive with the internal buy button markup generated by Team Checkout (2). The browser receives the assembled markup and displays it to the user (3). The browser loads Team Checkout's JavaScript containing the Custom Element definition for the buy button (4). The Custom Element's initialization code (constructor, connectedCallback) runs. It hydrates the server-generated markup and can react to user input from this point on (5).

The integration works. Run the example with `npm run 16_universal` on your machine and open localhost:3000/product/fendt in your browser to see it working.

- Notice *Team Checkout's* mini cart and *Team Inspire's* recommendation fragment. The integration for these fragments works the same way as for the buy button.
- Have a look at the server-logs in the console. You can see how Nginx requests the individual SSI fragments needed for the page.
- See how the price on the buy button updates client-side when you select the platinum edition.
- Clicking the button triggers the checkmark animation and updates the mini-cart.
- Disable JavaScript in your browser to simulate how the page looks when the client-side code failed or isn't loaded yet.

SIDEBAR **Progressive enhancement**

You've noticed that the buy button now appears even with JavaScript disabled. But clicking it does not perform any action. This is because we are attaching the actual add-to-cart mechanics via JavaScript. But it's straight forward to make it work without JavaScript by wrapping the button inside an HTML form element like this:

```
<form action="/checkout/add-to-cart" method="POST">
  <input type="hidden" name="sku" value="fendt">
  <button type="submit">buy for 54 $</button>
</form>
```

In the case of failed or pending JavaScript, the browser performs a standard POST to the specified endpoint provided by *Team Checkout*. After that, *Team Checkout* would redirect the user back to the product page. On that page, the updated mini-cart presents the newly added item.

Building an application with progressive enhancement principals in mind requires a little more thinking and testing than relying on the fact that JavaScript always works. But in practice, it boils down to a handful of patterns you can reuse throughout your application. This way of architecting creates a more robust and failsafe product. It's a good thing to work with the paradigms of the web and not reinvent your ones on top of it.

8.1.2 Contract between the teams

Let's have a quick look at the contract for including a fragment from another team. Here is the definition *Team Checkout* provides.

- **Buy Button** Custom Element: `<checkout-buy sku=[sku]></…>` HTML endpoint: `/checkout/fragment/buy/[sku]`

Since we are combining two integration techniques, the team offering the micro frontend needs to provide both: the Custom Element definition and den SSI endpoint, which delivers the server-side markup. The team using the micro frontend also needs to specify both. In our example *Team Decide* uses this code:

```
<checkout-buy sku="fendt">
  <!--#include virtual="/checkout/fragment/buy/fendt" -->
</checkout-buy>
```

These two lines include a lot of redundancy. To reduce friction, it's a good idea to establish a project-wide naming schema. This way, tag-names and endpoints all look alike, and teams can use a generic template for including a fragment. Figure 8.5 shows how a schema could look.

Figure 8.5 This schema shows how you could generate the universal integration markup in a standardized way. When offering or integrating a fragment, teams need to know three properties: the name of the team which owns it, the name of the micro frontend itself, and the parameters it takes.

8.1.3 Other solutions

This is, of course, not the only way to build a universal integration. Instead of SSI and Web Components, you can also combine other techniques. Integrating server-side with ESI or Podium and adding your client-side initialization on top would also work.

Are you looking for a batteries-included solution? Then you could try the Ara Framework.[68]. Ara is a relatively young micro frontends framework, but it's built with universal rendering in mind. It brings its own SSI-like server-side assembly engine written in Go. Client-side hydration works through custom initialization events. Examples for running a universal React, Vue.js, Angular, or Svelte application exist.

8.2 When does universal composition make sense?

Does your application need to have a fast first-page load? Your user interface should be highly interactive, and your use case requires communication between the different micro frontends? Then there is no way around a universal composition technique like you've seen in this chapter.

8.2.1 Universal rendering with pure server-side composition

But the fact that one team wants to use universal rendering does not mean that you need a client-side composition technique. Let me give you an example.

Team Decide owns the product page and includes a header micro frontend (fragment), which *Team Inspire* owns. The two applications (product page & header) do not need to communicate with each other. Here a simple server-side composition is sufficient. Both teams can adopt universal rendering inside of their micro frontends if it helps their goal. But they don't have to. If the header has no interactive elements, a pure server rendering is sufficient. They can add client-side rendering later on if their use-case changes. The other team does not have to know

about it. From an architectural perspective, universal rendering inside a team is a team-internal implementation detail.

8.2.2 Increased complexity

Universal composition combines the benefits of server- and client rendering. But it also comes with a cost. Setting up, running, and debugging a universal application is more complicated than having a pure client- or server-side solution. Applying this concept on an architecture level with universal composition doesn't make it easier. Every developer needs to understand how integration on the server and hydration on the client works. Modern web frameworks make building universal applications easier. Adding a new feature is usually not more complicated. But the initial setup of the system and onboarding of new developers takes extra time.

8.2.3 Universal Unified Single-Page App?

Is it possible to combine the application shell model from chapter 7. with universal rendering? Yes, in this chapter, we combined client- and server-side composition techniques to run multiple universal application in one view. You could also combine client- and server-side routing mechanisms to create a *universal application shell*. However, this is not a trivial undertaking, and I haven't seen production projects that are doing this right now.

The single-spa project plans to add server-side rendering support. But at the time writing this book this feature hasn't been implemented yet.[69].

Let's take a look at our beloved comparison chart in figure 3.3 for the last time. As stated before, running a universal composition setup is not trivial and introduces extra complexity. Since it builds on the existing client- and server-side composition techniques, it also does not introduce extra technical isolation. But this approach shines when it comes to user experience. It's possible to achieve the page load speeds of server-rendered solutions, and it also enables building highly interactive features that directly render in the browser.

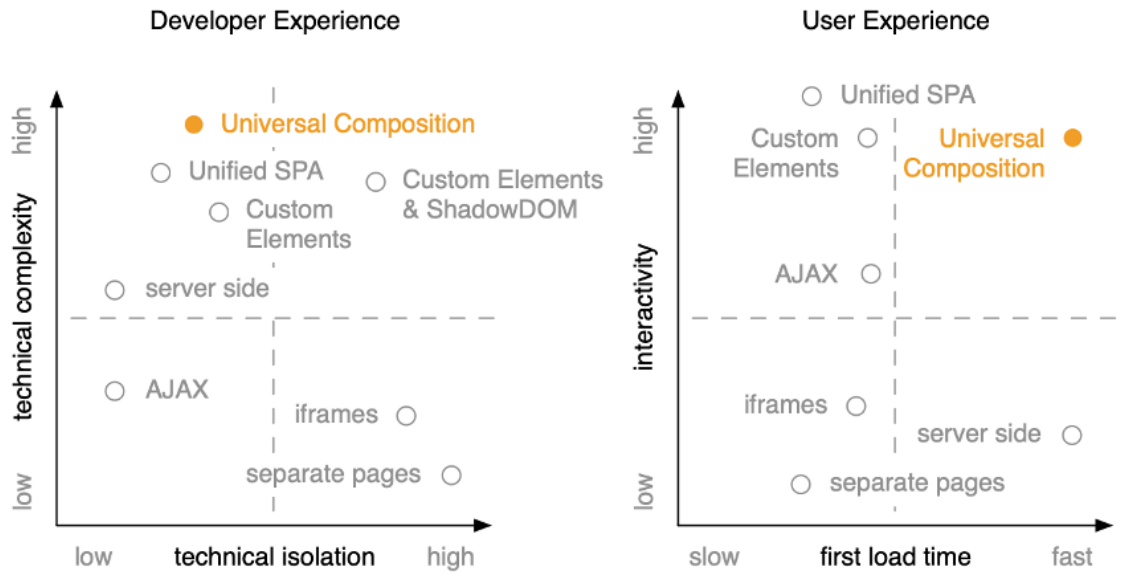


Figure 8.6 To run a micro frontends integration that supports universal rendering for all teams, we need to combine server- and client-side composition techniques. Both have to work together in harmony. This makes this approach quite complex. Regarding user experience, it's the gold standard since it delivers a fast first-page load while also providing a high amount of interactivity. It also enables developers to build their features using progressive enhancement principals.

To keep this chart readable, I've omitted the theoretical *Universal Unified SPA* option. It's by far the most complicated approach, but it would rank even higher on the interactivity scale since it eliminates all hard page transitions.

8.3 Summary

- Universal rendering combines the benefits of server- and client-rendering: fast first-page load and quick response to user input. To leverage this potential in a micro frontends project, you need to have a server- and client-side composition solution.
- You can use SSI together with Web Components as a composition pattern.
- Each team must be able to render it's micro frontend via an HTTP endpoint on the server and also make it available via JavaScript in the browser. Most modern JavaScript frameworks support this.
- On the first page load, a service like Nginx assembles the markup for all micro frontends and sends it to the browser. In the browser, all micro frontends initialize themselves via JavaScript. From that point on, they can react to user input entirely client-side.
- Currently, there's no web standard to server render a Web Component. But there are custom solutions to define ShadowDOM declaratively. In our example, we use the regular DOM to prepopulate the Web Components content on the server.
- It's possible to implement a universal application shell to enable client- and server-side routing. However, this approach comes with a lot of complexity.

9 Which Architecture Fits My Project?

This chapter covers

- Contrasting different micro frontend architectures you can build with the learned integration techniques.
- Comparing the benefits and challenges of the different high-level architectures.
- Figuring out the best architecture and composition technique for your project's needs.

In the last six chapters, you've learned different techniques to integrate user interfaces owned by different teams. We started with simple ones like links, iframes, and AJAX, but also more sophisticated ones like server-side integration, Web Components, and the app shell model. These chapters all ended with a simplified comparison chart indicating how the newly learned technique compares to the previous ones. In this chapter, we'll put all the puzzle pieces together and also make a more in-depth comparison. First, we revisit the terminology and highlight the key advantages of the different techniques and architectures. After that, you'll learn about the Documents-to-Applications Continuum, which can help to decide if you should go for a server- or client-side integration. This distinction is crucial because it determines which architectures and integration patterns are suitable for your use-case. We'll end this chapter with an architecture decision guide. You'll learn how you can make a sound choice based on a handful of questions. These questions will lead you through the different options.

9.1 Revisiting the terminology

When you are setting up a micro frontends project with different teams, everyone must use the same vocabulary. That's why we take a step back and sort the terms you've learned in the previous chapters. We'll start with the basic building blocks: the **integration techniques**. Then we'll look at different **high-level architectures** that you can build with them.

Figure 9.1 shows all the **integration techniques** we've covered in this book.

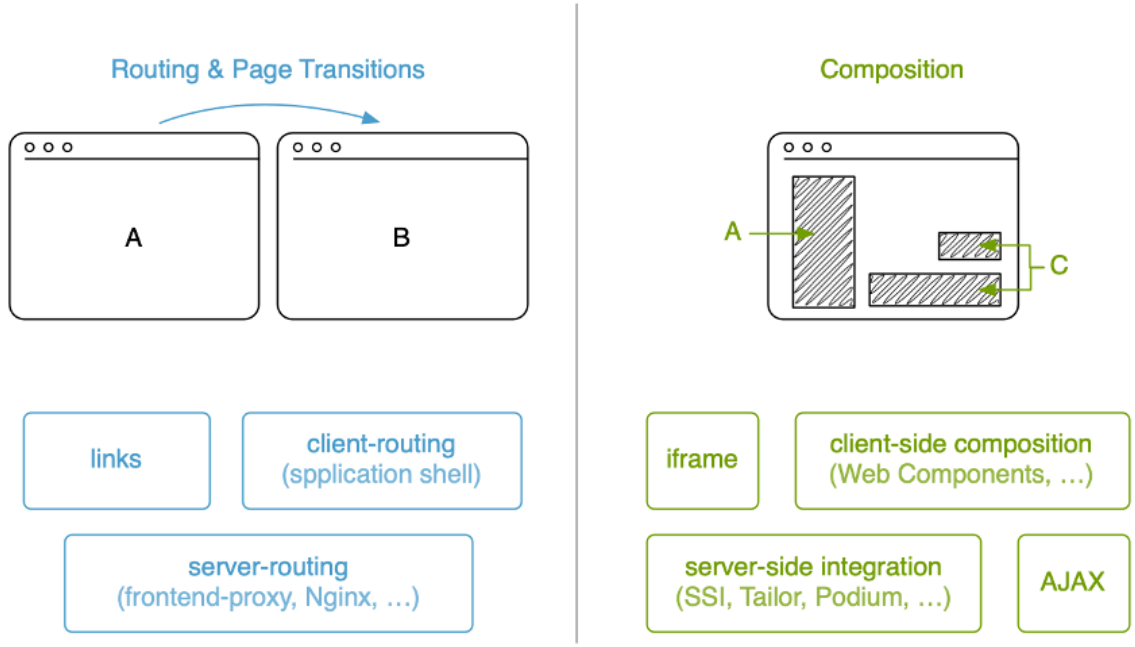


Figure 9.1 The integration techniques required for a micro frontend architecture. On the left, we see two techniques for handling cross-team page transitions. The right side shows a list of methods to compose different user interfaces onto one page.

We can group them into two categories: **Page-Transition** and **Composition**. Let's look at the transitions first.

9.1.1 Page transitions

When we talk about page transitions as an integration technique, we technically always mean **inter-team page transitions**. How does a user get from a page owned by Team A to a page owned by Team B? From an architectural standpoint, it's not essential to know how a team handles transitions between its pages. This is an implementation detail.

LINKS

The plain old hyperlink is the most basic form for doing a micro frontends integration. Each team is responsible for a set of pages. Handing over the user to another part of the application is as easy as placing a link to the other team's work. In its cleanest form, there's no extra coordination needed. Teams could even host their part of the applications under different domains. We covered the link in 2.2. Integration via links.

APPLICATION SHELL

Clicking on classical hyperlinks forces the browser to fetch the target markup from a server and then replace the current page with the new one. Having a reload is fine for a lot of use-cases. But the evolution of the browsers History API and the rise of single-page app frameworks enabled developers to build entirely client-side page transitions. Its main benefit is the opportunity to render the layout for the target page instantly. That way, the user gets a quick response, even if the content data requested from the server is still pending. Implementing client-side page transitions across team boundaries requires a central piece of JavaScript in the browser. It's typically called **app shell**. The central app shell acts as a parent application to the single-page applications built by the different teams. It determines which team-applications should be active based on the browser's URL. When the URL changes, it passes the responsibility for the page from Team A to Team B. You find more details on this in chapter 6. Communication Patterns.

9.1.2 Composition techniques

In practice, you often want to show user interface parts from different teams on one page. A typical example of this is a header or navigation micro frontend. One team builds and owns it. All the other teams integrate it on their page. It could also be functionality like the buy button or mini basket on our product page.

In this book, we often called an **includable micro frontend** a **fragment**. To make the integration happen, we need a shared format. The owner of the fragment must provide it in a standardized format. The fragment consumer uses this format to integrate the desired micro frontend on his page.

We can broadly group the composition techniques into two buckets: **server-side integration** and **client-side integration**. We've also included the **iframe** and **AJAX** technique since they are a little bit of a hybrid between server and client.

SERVER-SIDE INTEGRATION

Implementing a server-side integration technique makes sense when teams generate their markup server-side. The markup for all fragments of a page gets assembled before it reaches the customer's browser. A central piece of infrastructure like a web-server will perform the markup assembly. In chapter 4. Server Side Integrations we used the SSI technique in the Nginx to perform this task. An alternative approach is that the team owning the page fetches the required fragments directly from the other teams. The server-side integration libraries Tailor and Podium work like this.

CLIENT-SIDE INTEGRATION

If teams generate their markup in the browser, you need a client-side integration technique. The most popular approach is leveraging the Custom Elements API from the Web Components spec. The API defines (de)initialization hooks. The team that owns the fragment implements them. This way, the integration happens directly through the browsers DOM API. No special libraries or custom JavaScript APIs required.

An essential part of client-side integration is communication. How can fragment A inform fragment B about an event that might be interesting? Micro frontends can communicate via Custom Events or an event-bus/broadcasting solution. We covered this in chapter 5. Client-side Composition.

IFRAME

The iframe is the weird but somewhat powerful stepchild of web development. It fell out of favor years ago for various reasons. Using iframes in responsive design doesn't work without JavaScript, and having a lot of iframes on a site is resource-intensive. But it's secret superpower is that it provides a high level of technical isolation. In a micro frontends context, this is a desirable feature. This way, faults in micro frontend A can not negatively affect micro frontend B. Communication across iframes is also possible through the `window.postMessage` API. We briefly talked about the iframe in chapter 2.3. Integration via iframe.

AJAX

Fetching a snippet of markup from a server-endpoint via JavaScript is the technique that enabled the Web 2.0 revolution back in the days. You can also use AJAX as an integration technique for micro frontends. Client-side JavaScript triggers the actual AJAX call to fetch server-side generated HTML. AJAX is a bit of a hybrid approach that does not fit into one of our client- or server-side integration buckets. It is often used in tandem with a server-side integration technique - incrementally updating the markup of an embedded micro frontend. It does not come with a canonical way to handle (de)initialization and communication. Using Web Components together with AJAX for internal updating is also a good fit.

These are the basic integration techniques you've learned so far. Let's zoom out a bit and have a look at different architectural styles.

9.1.3 High-level architectures

One of the micro frontends benefits is that teams are free to use the technology that fits their slice of the application best. However, before you start setting up a micro frontends project, all teams need to be on the same page when it comes to the high-level architecture. Are we building static pages that integrate solely via links, or is the goal to create a highly dynamic and tighter integrated single-page app? You should consciously make this decision together with all teams. Figure 9.2 shows six different architectures.

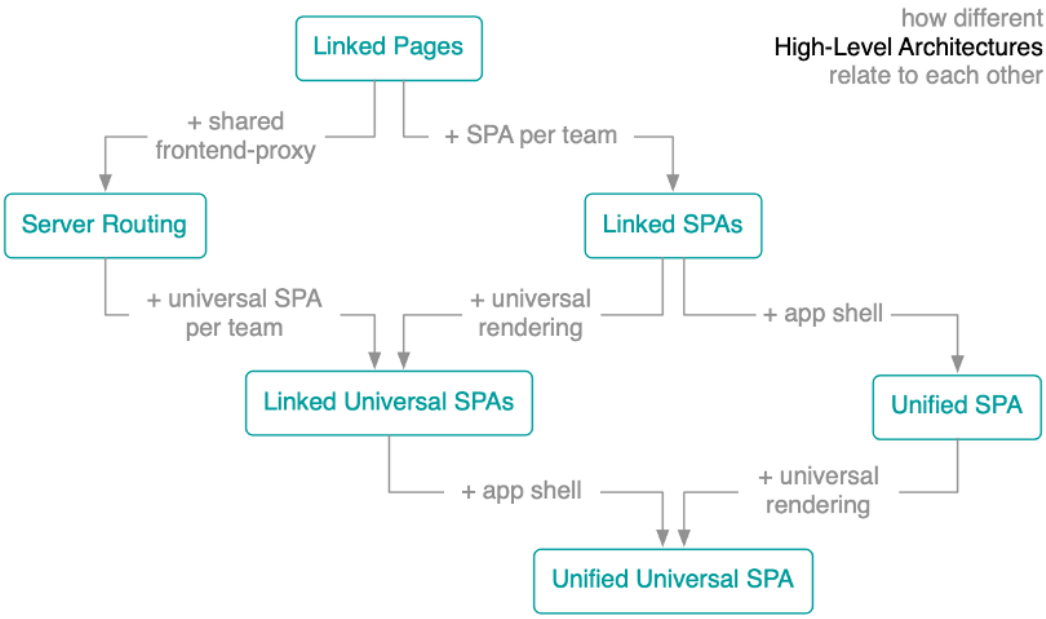


Figure 9.2 Different architectural styles to build a micro frontends project. This chart starts with the simplest form, the linked pages approach, and shows how you can extend this with extra features like single-page applications, universal rendering, or a shared app-shell.

We'll go through them from top to bottom.

LINKED PAGES

The most simple architecture. Every team serves its pages as complete server-rendered HTML documents. Clicking a link reloads the complete page and shows the desired content. This hard navigation happens if you are moving between pages from the same team or if you are navigating across team boundaries. The simplicity of this approach is its main benefit: no central infrastructure or shared code required, debugging is straight forward, and new developers instantly understand what's going on. But from a user experience point of view, there's room for improvement.

SERVER ROUTING

It's identical to the Linked Pages approach, but with the difference, that all requests pass through a shared web-server or reverse proxy. This server sits in front of the team's applications. It has a set of routing rules to identify which team should handle an incoming request. The routing is often done via URL prefixes associated with a specific team. We talked about this in chapter 3.2. Routing via a shared web-server.

LINKED SPAS

To improve the user experience and react to input faster, a team can decide to switch from delivering static server-generated pages to implementing a client rendered single-page app for the pages they own. This way, all link clicks for pages of this team result in a fast soft navigation. The transitions between team boundaries are still hard navigations. Technically the adoption of a single-page app architecture inside one team can be seen as an implementation detail. As long linking to a specific page from the outside still works, teams can decide to change its internal architecture. But the distinction between Linked Pages and Linked Single-Page Apps is essential when we talk about more advanced architectures and suitable integration techniques later.

LINKED UNIVERSAL SPAS

Teams can also decide to adopt universal rendering. The markup for the first request gets rendered on the server. It enables a pretty fast first page-load experience. From there on, the application behaves like a single-page app - incrementally updating the user interface as needed. From a team's point of view, this is a more complicated setup, which requires some additional development skills. But from an architectural view, this approach is identical to the other "linked" architectures. The contract between the teams is still a set of shared URL patterns. A navigation across team boundaries results in a reload of the page. But when implemented well, these reloads should be more seamless compared to a Linked SPA architecture, where the browser needs to execute a bunch of JavaScript before the user can see the content. Chapter XREF discusses universal rendering, its benefits, and its challenges.

UNIFIED SPA

The Unified SPA describes a single-page application composed out of other single-page applications. In chapter 6. Communication Patterns we introduced this concept. It requires all teams to build their software as a single-page app. These single-page apps are then unified by a parent application, which is often called the **app shell**. The shell typically does not render any user interface. Its job is to listen to changes in the browsers address bar and pass control from one single-page app to another if necessary. With the Unified SPA architecture, all page transitions are soft navigations. This leads to a snappier and more app-like user interface. However, the app shell is a central piece of code. It introduces a non-trivial amount of coupling and complexity.

UNIFIED UNIVERSAL SPA

When we take the Unified SPA model and introduce universal rendering, we arrive at something we call Unified Universal SPA. With this model, each team builds a single-page app with universal rendering capabilities. To make this work, the parent application (app shell) also needs to be universal. It needs to be able to run on the server and the client. This is a pretty challenging architecture. It promises to combine the best of all worlds but comes with the most complexity.

9.2 Comparing complexity

The architecture and the level of integration you choose has a considerable effect on your complexity. This complexity manifests itself in different aspects:

- Initial infrastructure work that's required to get started.
- Number of moving parts (services, artifacts) that need maintenance.
- Amount of coupling: Which changes require more than one team to become active?
- Developer skill level: What concepts do new developers need to understand?
- Debugging: How easy is it to attribute a bug to a specific team.

Figure 9.3 sorts the described architectures in four complexity groups. Starting from very simple and ending with very complex.

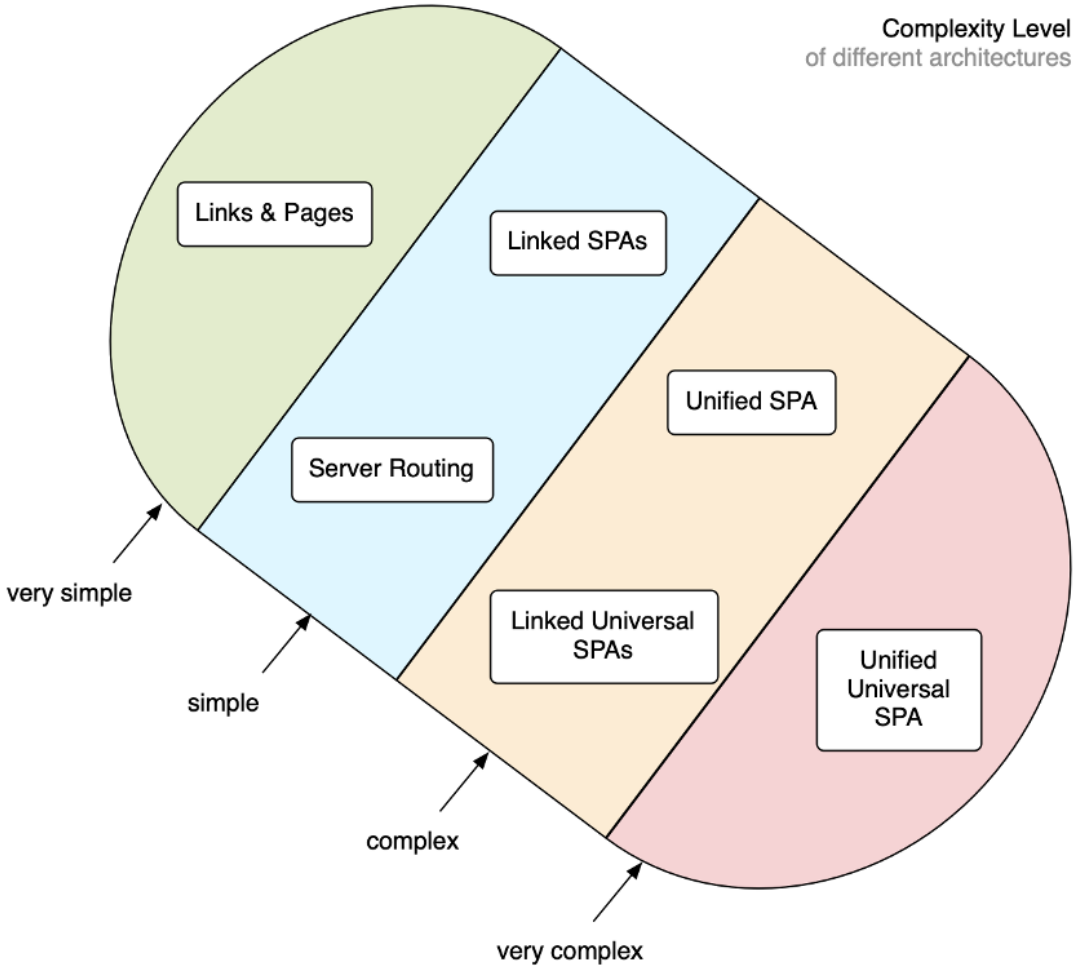


Figure 9.3 Micro frontend architectures sorted by complexity. The Links & Pages approach is the simplest one to build and run. The complexity rises as you move to more sophisticated architectures. The Unified Universal SPA approach requires a lot of development skills to get it right. You also need shared infrastructure and code to make it happen.

This is, of course, only general guidance. The real cost associated with an architecture depends on your team's experience and the use-case. **As a rule of thumb you should always opt for the most simple architecture you can responsibly get away with.**

Sure, it's nice to have a Unified SPA with no hard page transitions, but does the extra work required to achieve and maintain this justify the potential benefits?

9.2.1 Heterogeneous architectures

In the descriptions above, we always assumed that all teams use the same architecture. But you can also mix and match to create a heterogeneous architecture. For a team that builds fast-loading landing pages, the Links & Pages approach might be sufficient. But for a seamless browsing experience, you want to create a Unified SPA that integrates the team which owns the product list and the team managing the product pages. These architectures can work side-by-side: Some teams are doing Links & Pages, some other teams share an app shell to deliver a Unified SPA. This way, you only increase the complexity in the areas where it's needed.

But having a heterogeneous architecture also has drawbacks:

- There is no go-to architecture for a new team. Teams need to analyze and discuss their use-cases beforehand. (This is not necessarily a drawback.)
- Integrating fragments from different teams might get harder. Teams need to deliver their includable micro frontend in a format that works for the page that includes it.

9.3 Are you building a site or an app?

As you've seen throughout the book so far, it makes a significant difference if you render your markup on the client or server. Its a general question that everyone who's setting up a new web project has to answer. But in a micro frontends context, this decision is essential. It defines which integration techniques are suitable.

In this section, you'll learn about the Documents-to-Applications Continuum. I've found this concept helpful in architecture discussions. It creates an excellent mental model that helps you pick the right tools and techniques for the job. It provides a counterweight to the "Let's use the hot new JavaScript framework!"-reflex many developers (me included) have when they're confronted with a greenfield project. After explaining the concept, we'll look at how the high-level architectures fit into this continuum.

9.3.1 The Documents-to-Applications Continuum

What purpose does the project we are building serve? Do people come to our site to **consume content**, or do they want to **use a specific functionality** we provide? For better visualization, it helps to look at extreme examples:

- **content-centric:** Imagine a simple blog. A user can browse the list of posts and read the complete content on a dedicated article page.
- **behavior-centric:** Imagine an online drawing application. People can go to the site and draw beautiful sketches with their fingers and export them as an image.

The first one is a prototypical web site where the content is essential. The second one is a pure application. It does not bring any content. It's all about the functionality it provides to the user.

In a non-trivial project, it's typically not that black and white. This is where the Documents-to-Applications Continuum.[70] comes in. The idea is that both examples are at different ends on a spectrum, as illustrated in figure 9.4. Positioning your micro frontends project on this scale can help to set the right priorities and select an appropriate high-level architecture.

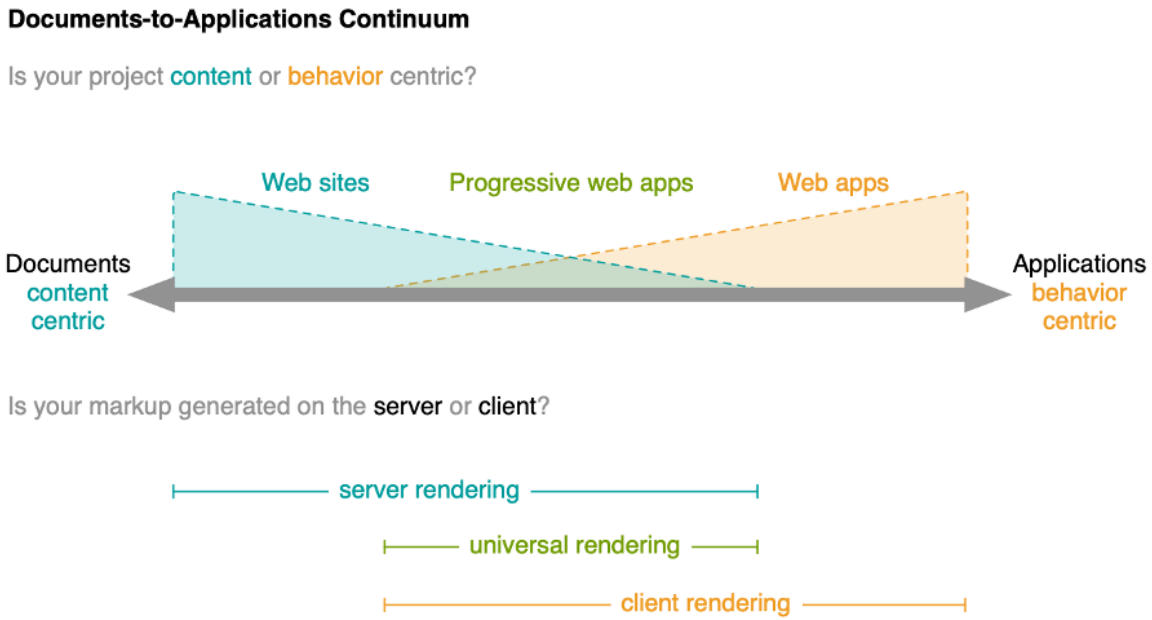


Figure 9.4 The Documents-to-Applications Continuum provides a mental model to help you think about if your project is more of a web site or a web application. It's a gradual scale and not a black and white decision.

Let's look at two examples. Where would amazon.com fit on this scale? They provide a lot of functionality. You can search, sort, and filter through product lists, rate products, manage your returns, or have a live chat with their customer service. But at its core, it's a content-centric site. A good question to ask is: **Would the site still be useful if we'd strip away all behavior?** For amazon.com, we can answer this with a definite yes. No doubt, the extra functionality is also important, but without products, the features would be pretty useless. We would put their site somewhere on the left part of the continuum. Starting with server-site composition with the option to upgrade it to a universal composition is a safe bet when picking a micro frontends architecture.

Now to our second example. The site CodePen.io let's web developers and designers put together HTML, CSS, and JS to get a live preview in the browser. Developers use the online code editor to sketch out ideas or isolate bugs. CodePen also has an active community of people who showcase their work and share code with others. You can go to the site and discover new exciting techniques by browsing the public catalog. How does CodePen fit on our continuum? It's a harder question to answer because its strong on both aspects: the online editor (behavior-centric) and the public catalog (document-centric). If we'd strip away all behavior, the online editor would vanish. If we remove all content, the catalog disappears, but the editor is still

there. That's why we'd probably put CodePen in the middle of the spectrum. If we rebuilt CodePen in a micro frontends architecture, we would establish two teams. The *Team Editor* would pick a client-side approach. *Team Catalog* would likely go the server-side route. This is a good starting point. To decide which micro frontend integration techniques they we'd have to go a step further and analyze the use-cases. Does one team need to include content from the other? How does the user move through the site?

9.3.2 Server, client or both

Classifying your product onto the continuum is a good starting point to identify if your templating should live on the server or in the browser. If your product has a strong content focus, server-side rendering should be your first choice. Using progressive enhancement to add functionality should feel natural.

If your building an application where it's all about interaction and not about content, a purely client-rendered solution will be the best fit. Here the concept of progressive enhancement doesn't help you at all because there is no enhanceable content, to begin with.

For a project that resides in the middle of the spectrum, you need to make a choice. Server- and client-side templating are valid options. But in this area, both have their advantages and disadvantages. If you aren't afraid of the extra complexity, you can also pick both and go with the universal rendering option.

Let's revisit our high-level architectures. Figure 9.5 highlights which of them use server-, client- and universal rendering.

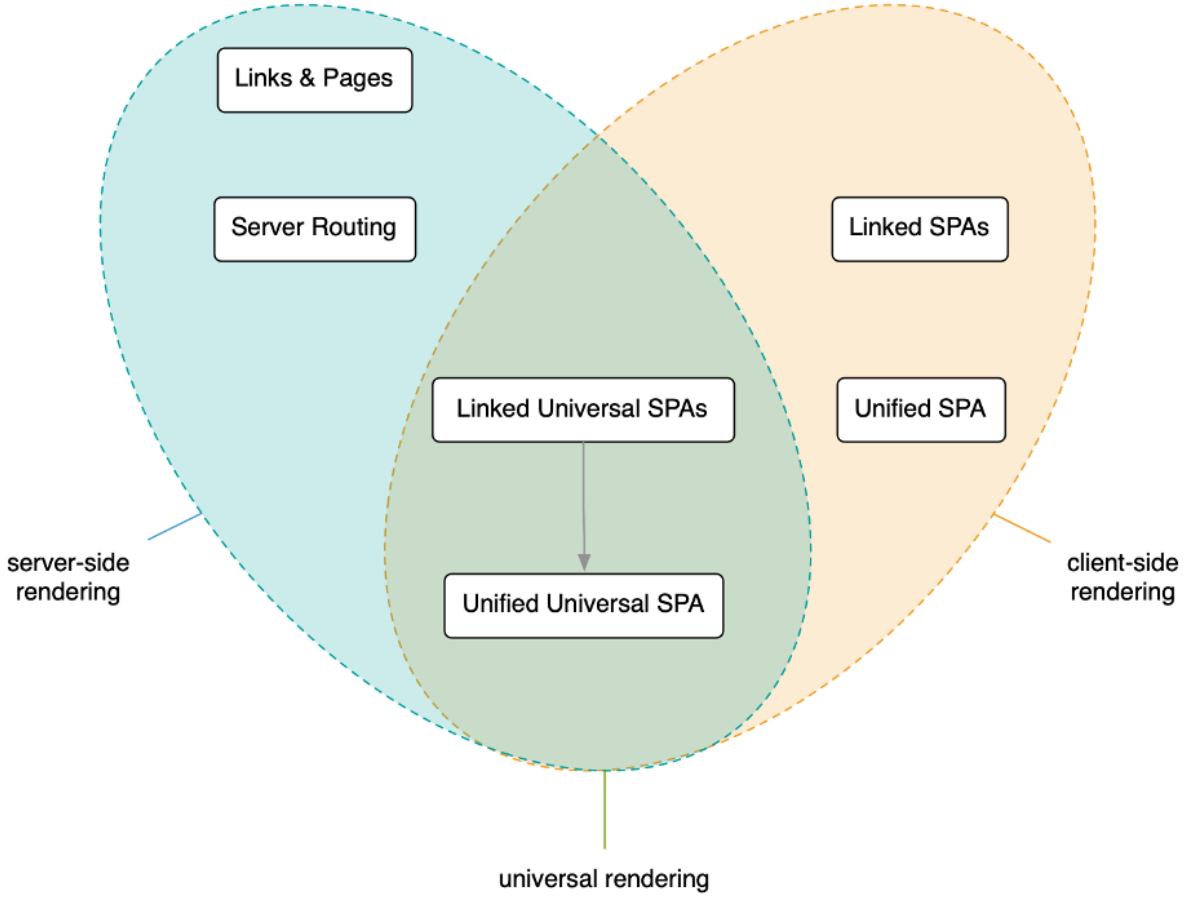


Figure 9.5 Illustrates which architectures feature server-side, client-side or universal rendering.

Make sure the architecture you choose aligns with the nature of your project and the business. Templating and complexity considerations are two significant factors in making a decision. Next up, we'll take another angle on this decision using a decision tree.

9.4 Picking the right architecture and integration technique

Now we've sharpened our vocabulary and have a mental model to pinpoint what kind of product we are building. Let's look at a concrete way to determine which architecture and integration your project needs. Figure 9.6 shows a decision tree that helps with this question. It's inspired by Manfred Steyer's work.[71] for creating Angular based frontend microservices.

Take some time to understand what's going on in this diagram. Follow the lines from top to bottom by answering the questions until you reach your **high-level architecture**. From there on, you can follow the dotted line to get to the **compatible Composition Technique**. If your use-case does not require you to have different micro frontends to be active at the same time (fragments or nested micro frontends), you can skip this step.

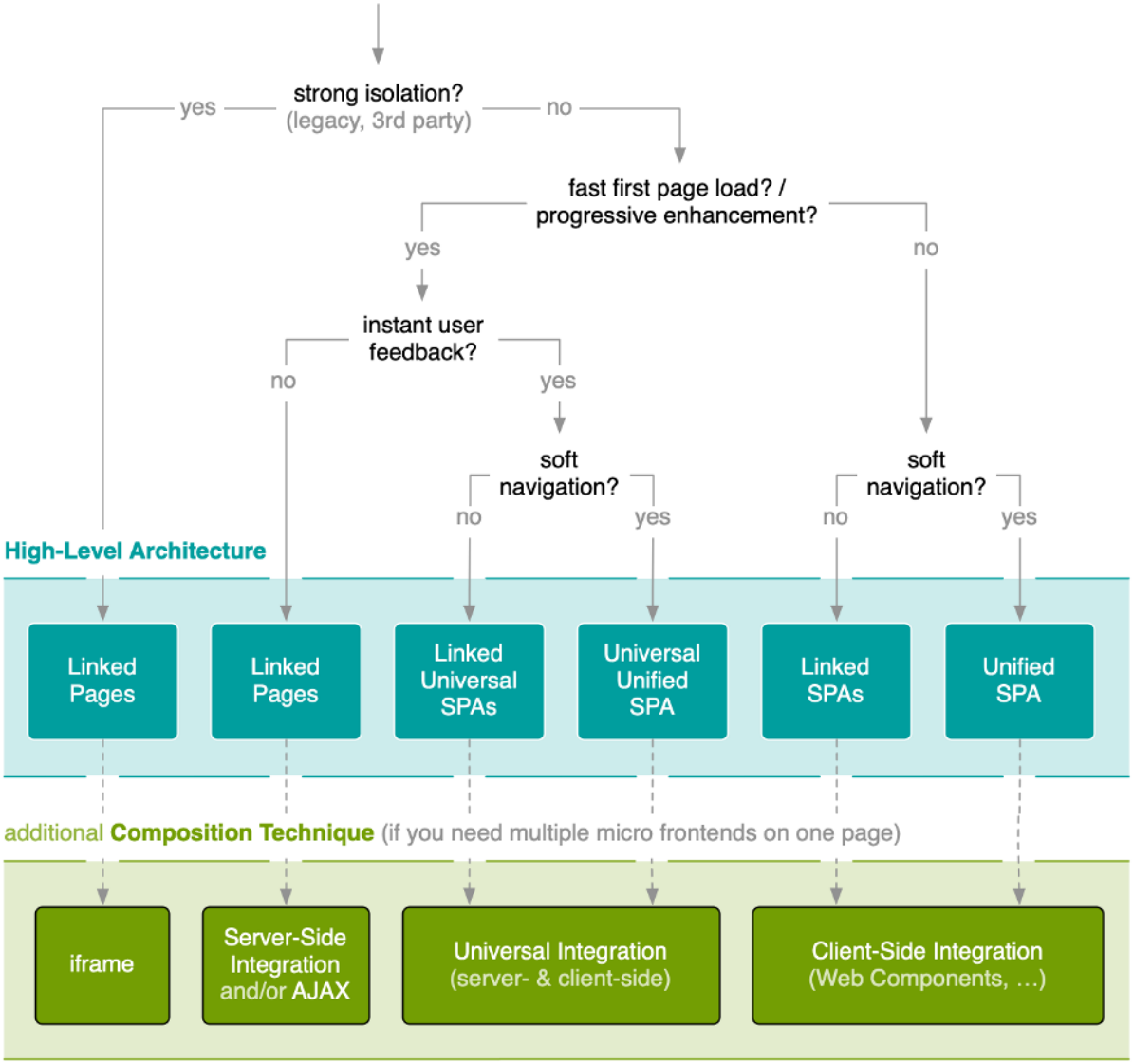


Figure 9.6 The decision tree helps to pick a micro frontends architecture based on your project's requirements. It also shows which kind of composition technique is appropriate for your use-case.

Let's have a look at the questions in this decision tree.

9.4.1 Strong isolation (legacy, 3rd party)

Do you want strong technical isolation between the code of the teams? Of course you want, why wouldn't you? Isolation and encapsulation generally lead to less unforeseen effects and reduces bugs. But sadly, opting for strong isolation eliminates a lot of other possibilities. So the right question to ask is: Do you **need** strong isolation? This is typically true if you integrate a legacy system that doesn't respect namespacing rules and requires global state to work correctly. Another reason is security. If you are integrating with an untrusted 3rd party solution or one part of your application has high-security requirements (e.g., it handles credit card data), it can be necessary to better shield the micro frontends against each other.

9.4.2 Fast first page load / Progressive enhancement

This is a double question. If you need one of these properties, you should follow the **yes** arrow.

Having a **fast first-page view** is always pleasant, but the importance of this property heavily depends on your business. If you want your site to rank high in search results, first-page load performance is nothing you can ignore. **Search engines** like Google increasingly **favor fast-loading sites** in their ranking.[72]. Even if search ranking us not your primary goal, there are a lot of case-studies.[73] that show how better web performance increases business metrics.

We talked about the benefits of **progressive enhancement** in chapter Progressive enhancement. If you'd locate your project on the middle or left side of the Documents-to-Applications Continuum, I'd highly recommend adopting progressive enhancement practices. You should encourage all developer teams to learn about this approach. For developers that started their web career with frameworks like React or Angular, the concepts might sound strange at first sight. However, architecting features with progressive enhancement in mind and embracing the primitives of the web will lead to more maintainable, easier to understand, and more stable software. If you're on the far right in the continuum and building a pure web application, there is typically no content to enhance. Then progressive enhancement won't help you at all.

9.4.3 Instant user feedback

In the last question, we talked about the first-page load performance. **But how does your site react to further interactions from the user?** The classical "click a link" and "fetch generated markup from the server" works for a lot of cases. Primarily when you use AJAX techniques to avoid a full page reload. In this model, the complete templating resides on the server. **This means that at least one server-roundtrip is necessary to update the UI in response to a user input.**

If you need to be faster than this, you must adopt client-side rendering. Fetching data will be a little bit faster since JSON data is more compact than rendered HTML, but the network latency itself stays the same. However, the most significant advantage of client-side rendering is that it enables us to provide instant feedback. Even if the data the user wants to see is still in transit, you can update the view and show **placeholders and skeleton screens**.[74]

It also enables you to adopt **Optimistic UI patterns**.[75] With Optimistic UI, you try to increase the perceived performance by **instantly rendering the result that's most likely**. Let's look at a shopping cart example: When a user wants to delete an item, she clicks on the delete button. The browser calls the associated API on the server, and when it comes back, the item is deleted from the visual shopping cart list. With Optimistic UI, you assume that the delete API call works in most of the cases. That's why you remove the line item directly and don't wait for the API call to return. If this assumption turns out not to be true (item remove failed), you restore the item in the

user interface and show an appropriate error message. This technique is powerful, but since you are effectively lying to your user, you should use it with care.

Being able to render a response to a user input instantly improves user experience and makes your site feel more app-like.

9.4.4 Soft navigation

In the "instant user feedback" question, we talked about improving the user experience inside a micro frontend. Now let's look at what happens when the user transitions across team boundaries. This question differentiates the **linked** architectures from the **unified** architectures. We talked about this in chapter 6. Communication Patterns. How important is it that inter-team page transitions are client-side rendered?

Answering this question **depends heavily on** the **team boundaries** you establish, the **number of teams** and the **usage pattern** of your application.

If you **create your team boundaries along with the user's tasks and needs** they don't cross team boundaries that often.

Say you are building a website for a bank. It has to distinct areas developed by two teams: users can **check their account balance** (Team A), they can also **calculate and request a housing loan** (Team B). For a good user experience, it might be essential to provide a high amount of interactivity inside these areas. This is something Team A and Team B can decide independently. But since **users seldom switch** between balance checking and loan requesting in one session **it might be fine to have a hard navigation between these areas**.

Let's pick another example. We are building a call-center application. The agents use the application to **manipulate orders** (Team A) and **make personalized recommendations** (Team B). Since the agent switches between these two micro frontends frequently, it might be a good idea to implement soft-navigation. It makes using the application faster and positively impacts the agent's workflow.

9.4.5 Multiple micro frontends on one page

If you've answered all questions on your way down the tree and arrived at your high-level architecture, there is one last bonus question: "Do you need composition?" Answering "yes" brings you straight down to the associated composition technique. If you are building a pure SPA, you need to integrate client-side. If you opt for server-generated pages, you should use a server-side integration.

Having a composition technique is optional. When we look at our banking example from the section before, we might not need a composition technique at all. The account area and the housing loan area could be two distinct sections of the site that link to each other.

The most common example of a composition is a header and navigation fragment. Usually, one team owns it, and the others include it on their page.

9.5 Summary

- Establishing a shared vocabulary across all teams avoids misunderstandings. Differentiating between transition techniques, composition techniques, and your high-level architecture helps every one to get a clear picture of what you are building towards.
- The Documents-to-Applications Continuum is a good mental model to identify if your project is more content- or behavior-centric. This distinction helps to make good technology choices.
- There are no right or wrong solutions. Whether a solution fits or not depends on the nature of your project, its usage patterns, the amount of coupling and complexity you are willing to accept, and your team's size and experience level.
- Not all teams need to adopt the same architecture. Some parts of your application might be document-centric, others more behavior-centric. With micro frontends, it's possible to mix and match. But when you need composition, you must find an integration technique that works for all teams.
- Try to pick the simplest architecture, which is reasonable for your business.

10 Asset Loading

This chapter covers

- Solving common asset loading challenges in a micro frontends context.
- Comparing techniques to deal with cacheability and synchronization when loading assets of different teams.
- Deciding what bundling strategy is appropriate: Many smaller bundles or fewer large ones.
- Understanding how on-demand loading can be effectively used with micro frontends.

In the last chapters, we covered a lot of different integration techniques. But we always focused on the content - integrating markup on the server and in the client. A topic we only discussed in passing is: *How to load the assets associated with a micro frontends?* In this chapter, we'll dive deeper into this significant side-topic. There are at least a handful of aspects that you must consider. How can we ensure that *teams can deploy* a micro frontend and the needed assets *on their own*? How do you *implement cache-busting to improve cacheability* without introducing tight coupling? How do you ensure that the loaded CSS and JS always fits the server-generated markup? How *coarse or fine-grain* should your *bundles* be? Do you want one big bundle for your application, one per team, or even smaller ones? How can *on-demand loading techniques* help in reducing the upfront asset data the browser needs to process?

10.1 Asset referencing strategies

We'll start with some techniques to integrate the assets into a page. For simplicity, we will stick to traditional `<link>` and `<script>` tags in the following scenarios. Module loaders like RequireJS.[76] (AMD) or CommonJS.[77] are popular and provide programmatic loading functionality. But nowadays, ES Modules are supported in all significant browsers.[78]. They are a web standard that solves most of our JavaScript loading needs without having an extra library or a custom module format.

Later in this chapter, we'll talk about bundle granularity. For now, let's assume that every team that provides an includable micro frontend (a fragment) generates one JavaScript and CSS file. The including team must add the references for both files to its page.

10.1.1 Direct referencing

The concept is pretty straightforward. If you want to integrate a micro frontend from another team, you have to add their references to make it work. You can think of the associated assets like adding an `import` at the top of your source code file in languages like Java, C#, or JavaScript.

If you are going the app shell route, it's different. There you have **one single HTML document**. It's the responsibility of the shared app shell to load the code for all micro frontends. The simplest way is to include all assets from all teams upfront. A smarter way would be to load assets just in time when the user needs them. The meta-frameworks single-spa implements on-demand loading. You can flip back to chapter 7.3. A quick look into the single-spa meta-framework to see the dynamic `import()` based JavaScript registration code. We'll talk more about on-demand loading later in this chapter.

Let's go back to *The Tractor Store* and revisit how we dealt with assets loading in the last chapters. There Team Decide referenced the assets directly. The other teams publish the URLs of the associated assets as part of their documentation.

Here is an example from chapter 5. Client-side Composition. Team Checkout specifies the Custom Element details for their buy button and the files that contain the associated initialization code and styling.

- **Custom Element:** `<checkout-buy sku="{sku}"></checkout-buy>`
- **Required Assets:** `/checkout/fragment.js`, `/checkout/fragment.css`

To ensure fast rendering, it's best practice.[79] to include stylesheets in the `<head>` and script-tags asynchronously at the end of the `<body>`. Team Decide directly includes these references in their product page's markup:

Listing 10.1 08_web_components/team-decide/product/porsche.html

```
<html>
  <head>
    <link href="/decide/page.css" rel="stylesheet" />
    <link href="/checkout/fragment.css" rel="stylesheet" />      ❶
  </head>
  <body>
    <h1>The Tractor Store</h1>
    <checkout-buy sku="porsche"></checkout-buy>                  ❶

    <script src="/decide/page.js" async></script>
    *<script src="/checkout/fragment.js" async></script>*        ❷
  </body>
</html>
```

❶ Styles for Team Checkout's fragments.

❷ Team Decide includes the "buy button" micro frontend of Team Checkout. It relies on Team Checkout's assets to be present.

❸ Scripts for Team Checkout's fragments.

10.1.2 Challenge: Cache-busting & independent deployments

One day CEO Ferdinand walks into Team Decide's office space - laptop under his arm. He grabs a chair, opens his laptop, and points at his screen. "I've read an article about the importance of web performance in e-commerce. I ran a tool called Lighthouse.[80] on our product pages. It measures performance and checks if our site uses best practices. We score 94 points. This score is way better than our competitors! However, Lighthouse shows one piece of advice. We seem to use an inefficient cache policy on static assets." .[81]

The current best practice for performant asset loading is to ship static assets (JavaScript, CSS) in separate files with a **one-year cache header**. This way, you ensure that the browser does not redownload the same file twice. Adding this cache header is not complicated. In most applications, web-servers, or CDNs it's a simple configuration entry. However, **you need a cache invalidation strategy**. If you've deployed a new CSS file, you want all users to stop using their cached version and download the updated one. An effective invalidation strategy is adding a fingerprint to the filename of the asset. The fingerprint is a checksum based on the contents of the file. A filename could look like this `fragment.a98749.css` and only changes when the file is modified.

We call this cache-busting. Most frontend build tools like Webpack, Parcel, or Rollup support it. They generate fingerprinted filenames at build time and provide a way to use these filenames in your HTML markup. You might already see the issue. **Cache-busting approach does not play nice with our distributed micro frontend setup.**

In the example before, *Team Decide* needs to know the path to *Team Checkout's* JavaScript and

CSS files. Yes, *Team Checkout* could update their documentation:

Required Assets: `/checkout/fragment.a62c71.js`, `/checkout/fragment.a98749.css`

But with the current process, *Team Decide* would have to manually update these references in their product pages markup every time *Team Checkout* deploys a new version. In this scenario a team is not able to deploy without coordinating with another team. This coordination is the kind of coupling we want to avoid. Let's explore some better alternatives.

10.1.3 Referencing via redirect (client)

You can circumvent this problem by using HTTP redirects. The idea is the following:

1. *Team Decide* references *Team Checkout's* assets as before without fingerprint. The URLs are stable and do not change (e.g., `/checkout/fragment.css`).
2. *Team Checkout* responds with an HTTP Redirect to the fingerprinted file (`/checkout/fragment.css /checkout/static/fragment.a98749.css`)

This way, *Team Checkout* can configure the directly referenced file (`/checkout/fragment.css`) with a short cache header or set it to `no-cache`. They can give the fingerprinted file which contains the actual content a long cache lifetime (e.g., one year). The benefit is that the registration code can stay the same, and the user only downloads the big asset file when it changes.

In our example `17_asset_client_redirect`, we've added a redirect configuration and caching header to the team's web-servers. You can look at the `serve.json` file in each team's folder. The `mfserve` library picks up this file. In a real application your built tool or bundler would create the fingerprints and redirect rules for you.

Listing 10.2 team-checkout/serve.json

```
{
  "redirects": [                                                    ❶
    {                                                               ❷
      "source": "/checkout/fragment.css",                           ❸
      "destination": "/checkout/static/fragment.a98749.css"         ❹
    },                                                              ❸
    ...                                                             ❸
  ],                                                                ❸
  "headers": [                                                      ❶
    {                                                               ❷
      "source": "/checkout/static/**",                              ❷
      "headers": [                                                  ❷
        { "key": "Cache-Control", "value": "max-age=31536000000" }  ❸
      ]                                                             ❸
    }                                                               ❸
  ]                                                                 ❹
}
```

❶ Configuring a redirect from the public asset path to the latest fingerprinted version.

❷ Setting a one year cache header to all fingerprinted assets. By default all other resources are served with `Cache-Control: no-cache`.

Start the application by running `npm run 17_asset_client_redirect`. The network requests for *Team Checkout's* fragment styles look like this.

```
# Request                                          ❶
GET /checkout/fragment.css                         ❶

# Response (redirect)                              ❶
HTTP/1.1 301 Moved Permanently                     ❶
Cache-Control: no-cache                            ❶
Location: /checkout/static/fragment.a98749.css     ❶

# Request                                          ❸
GET /checkout/static/fragment.a98749.css           ❹

# Response (actual content)                        ❹
HTTP/1.1 200 OK                                    ❹
Content-Type: text/css; charset=utf-8              ❺
Content-Length: 437                                ❻
Cache-Control: max-age=31536000000                 ❻
```

❶ Browser requests *Team Checkout's* fragment styles.

❷ *Team Checkout* responds with a 301 redirect to the fingerprinted resource. The redirect is not cacheable.

❸ Browser requests the fingerprinted resource.

❹ *Team Checkout* serves the asset file with a one-year cache header.

Figure 10.1 shows the example code with the network panel of the browser. You can see that each *registration file* redirects to the fingerprinted and cacheable version.

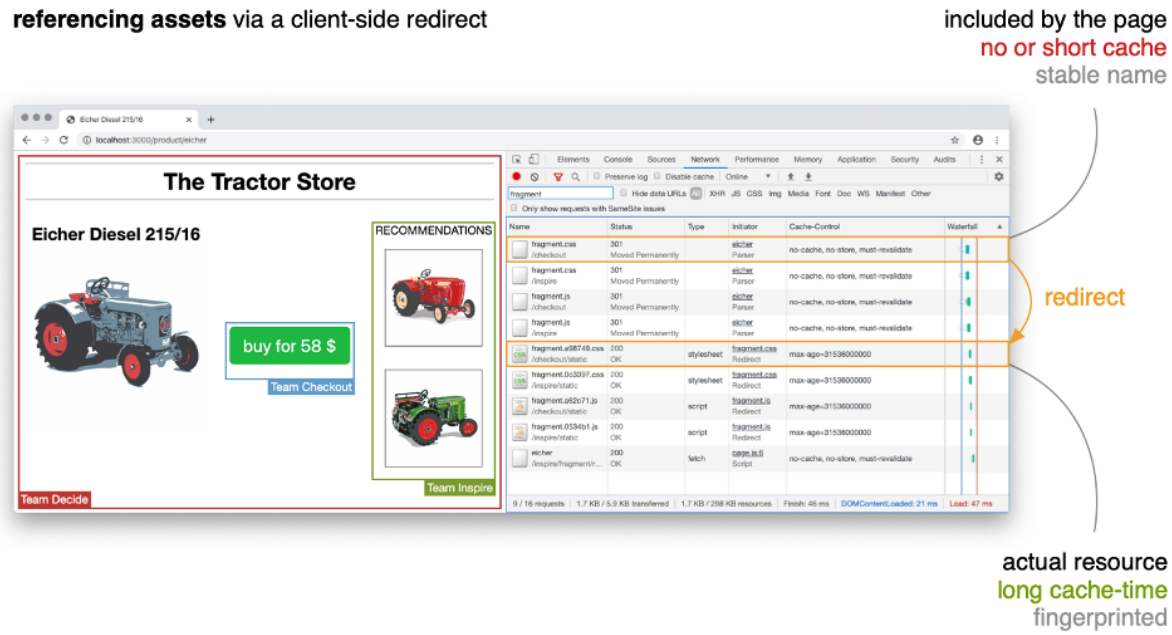


Figure 10.1 Shows the network requests for loading the fragments styles and script. There's a fragment.css and fragment.js for both Team Checkout and Team Inspire. Each resource redirects to the latest version of the file.

The main benefit compared to the direct referencing approach is the decoupling. A team that provides a micro frontend can ship versioned assets with **long cache times**. **They can update their code without having to notify the team that includes it.** It's also simple to build, and users only have to redownload an asset if it changed.

> **NOTE** It's possible to achieve similar decoupling and caching results by using `Cache-Control: must-revalidate` together with the `ETag` header. But using filename based versioning and long cache headers comes with a few other benefits we'll discuss later.

But there are drawbacks. The browser **can't cache the initial resource** that returns the redirect. It has to make **at least one network request** to make sure the redirect still points to the same resource.

A second problem is **missing synchronization**. The **redirect always points to the latest version**. When you do rolling deployments, you may have different versions of your software running at the same time. Then you might want to ensure that the CSS, JavaScript, and HTML are all from the same build.

We'll address the **additional network request** first and later talk about **synchronization**.

10.1.4 Referencing via include (server)

The teams are happy with the improved caching. Ferdinand reruns the Lighthouse test. It shows a 98 score - up 4 points. But a new potential improvement message appeared: **Minimize Critical Requests Depth**.[82]

With the redirect approach, we achieve the decoupling and caching benefits at the cost of more network requests. The browser has to make an extra lookup request before it knows the URL of an actual resource. Under poor network conditions, this can introduce a noticeable delay. Let's move this lookup request to the server. Server to server communication is much faster. There we talk about latencies on the realm of single-digit milliseconds.

If you are already using a server-side markup integration, we can also use this mechanism to register the assets. The idea is pretty straightforward. At the spot where *Team Decide* referenced the other team's assets using a `link` or a `script` tag, they include a piece of markup that's generated by the respective team.

We'll again use Nginx's SSI feature in our example. You can check back 4. Server-Side Composition for more details on how this works.

The product page markup looks like this:

> **NOTE** For simplicity, we haven't fingerprinted *Team Decide's* `page.css` and `page.js` files.

Listing 10.3 team-decide/product/eicher.html

```
<html>
  <head>
    <title>Eicher Diesel 215/16</title>
    ...
    <link href="/decide/static/page.css" rel="stylesheet" />
    <!--#include virtual="/checkout/fragment/register_styles" -->  ❶
    <!--#include virtual="/inspire/fragment/register_styles" -->  ❷
  </head>
  <body>
    <h1>The Tractor Store</h1>
    ...
    <script src="/decide/static/page.js" async></script>
    <!--#include virtual="/checkout/fragment/register_scripts" -->  ❸
    <!--#include virtual="/inspire/fragment/register_scripts" -->  ❸
  </body>
</html>
```

❶ SSI directive will resolve into the link tag markup from the teams.

❷ SSI directive will resolve into the script tag markup from the teams.

Team Checkout and *Team Inspire* have to provide the registration endpoints for scripts and styles. These endpoints are now part of the contract between the teams. This example shows the

content of one of these registration includes:

Listing 10.4 team-checkout/checkout/fragment/register_styles.html

```
<link href="/checkout/static/fragment.a98749.css" rel="stylesheet" />
```

It's a link tag that points to the fingerprinted asset file. Figure 10.2 illustrates the assembly process. Nginx replaces the directive with the contents of the registration include.

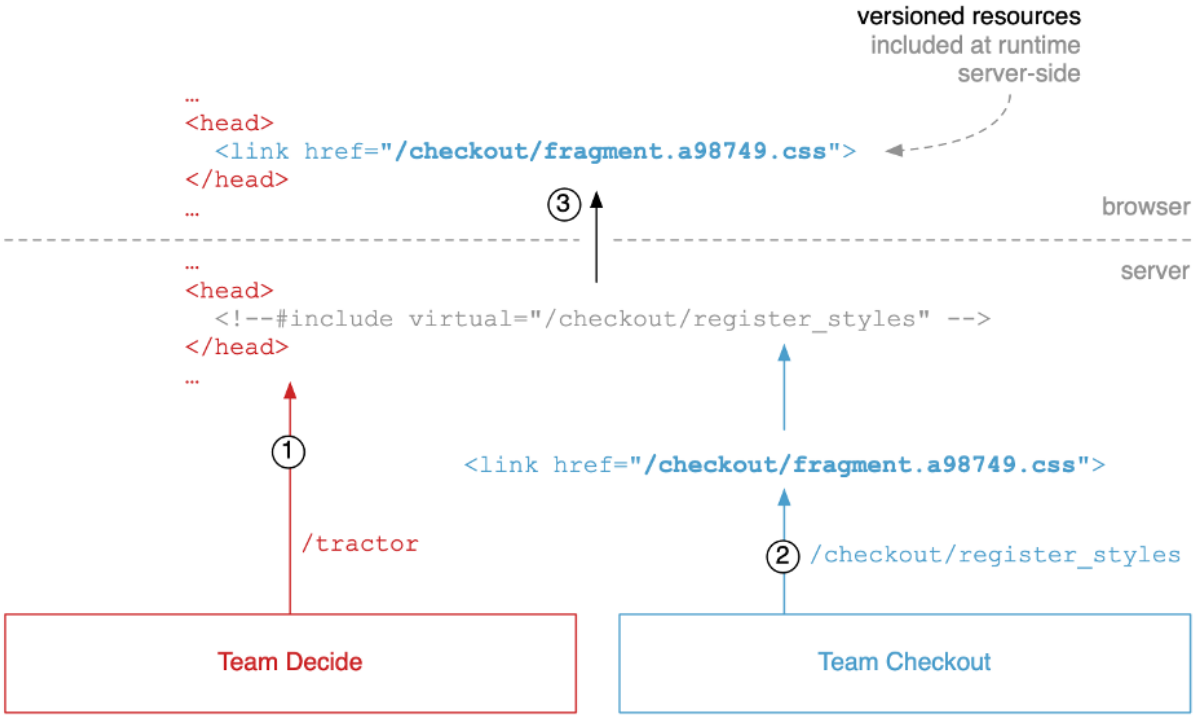


Figure 10.2 Team Decide doesn't reference Team Checkout's assets directly. Instead they add an SSI include directive which is pointing to Team Checkout's register_style endpoint (1). This endpoint returns HTML markup with the fingerprinted assets. Team Checkout can update the fingerprints on the fly without having to coordinate with Team Decide (2). The browser receives the already assembled markup with the link to the fingerprinted asset (3).

The markup that reaches the browser already contains the resolved include. The browser can instantly start to download the asset. If it's present in the disk cache, the browser can use its local copy without having to make another network request or revalidation. Start the example by running `npm run 18_asset_registration_include`. Figure 10.3 shows how a first page visit looks in the browser.

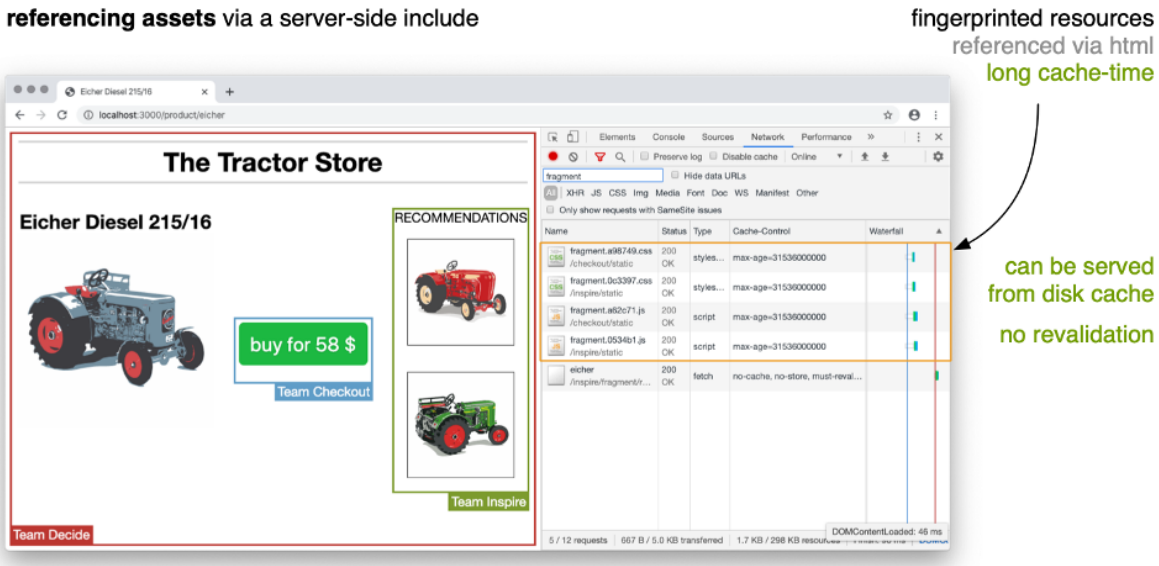


Figure 10.3 Shows the browsers network tab with the fragment assets required for this page. The HTML directly links to the fingerprinted files from the other teams. The assets are cacheable for a long time.

This approach **provides good decoupling**. Teams can change their asset URLs without having to notify another team. Because we don't need a client-side redirect or a revalidation request, this is also a **perfect solution from a web performance standpoint**.

If you don't already have a server-side integration mechanism in place **this approach requires some extra work and shared infrastructure**.

10.1.5 Challenge: Synchronizing markup and asset versions

Registering the assets server-side improved the Lighthouse score significantly. *Team Decide's* product page now scores the full 100 points. The developers and CEO Ferdinand are delighted. They rolled out the change to their production servers.

A week later, Noah, *Team Checkout's* DevOps guy, recognized a strange pattern while going through the server logs. From time to time, the application servers report a 404 error on one of their fingerprinted asset files. It seems like the browser is requesting a file which the application does not know. First, he suspected a bug, but after closer examination, he realized that these **issues appeared at times when *Team Checkout* deployed a new version** of their software.

After consulting his teammates, Noah was pretty confident about the cause of these issues: **Rolling deployments**. To deal with the high amount of traffic *Team Checkout* runs ten instances of their application. The application contains everything from database communication to rendering markup and shipping the assets. They use Kubernetes for automatic deployments. During a deployment, Kubernetes incrementally replaces the old applications with new instances. It does this step by step: creating a new instance, waiting until its operational, redirect traffic to

it, and then killing an old instance. Kubernetes repeats this process until all ten applications are updated. A full deployment can take a few minutes. During this time, old and new versions of the application run side by side. This running side by side is the cause of the 404 issues.

> **NOTE** The problem gets even worse when you use **canary deployments**. With canary deployments, you roll out the new version to a small percentage of your instances and monitor them for a while. If the new instances perform well, all instances will be updated. If they have performance issues, the team rolls back the deployment. With canary deployments, old and new versions run side by side for a much longer time. The risk of inconsistencies increases.

A load balancer routes an incoming request randomly to one of the ten application servers to distribute the work-load evenly. Imagine the freshly deployed instance serves the registration fragment (`/checkout/register_styles`) but the actual asset request (`/checkout/static/fragment.[new-fingerprint].css`) reaches an old instance which only knows the file with an old fingerprint. This scenario results in a 404 error and an unhappy user who's looking at a page with an unstyled buy button. Figure 10.4 illustrates this.

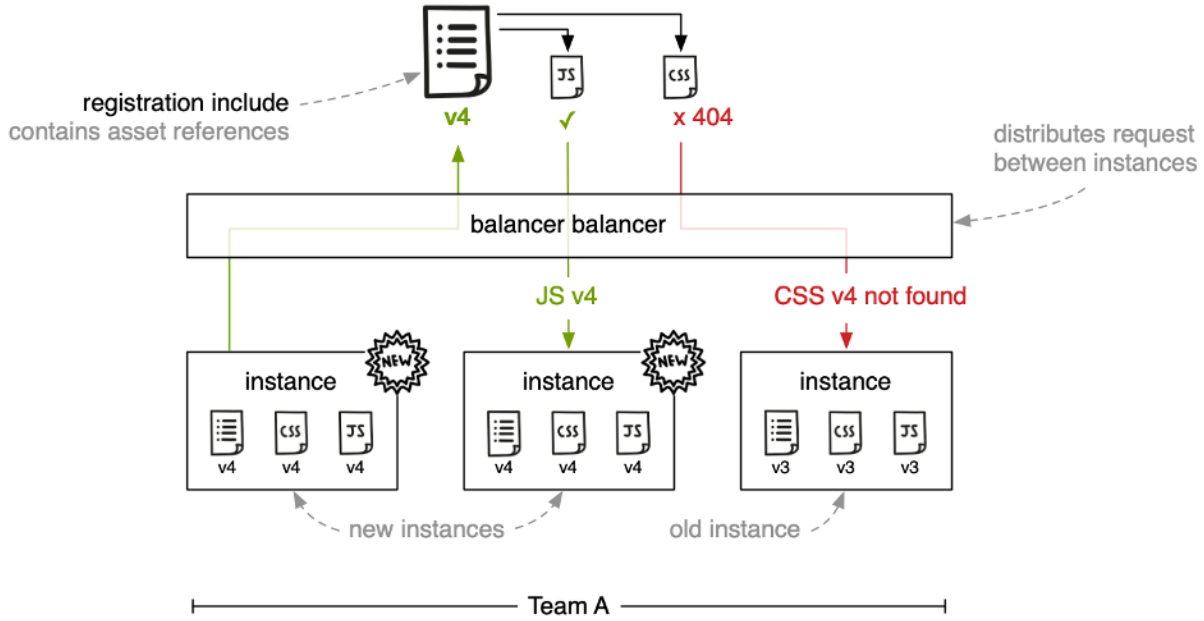


Figure 10.4 Using fingerprinted asset references can lead to issues during rolling deployments. When the registration include comes from an application server with a new version and the actual asset request reaches an old instance that doesn't know this files, the browser receives a 404 error.

There are two quick fixes to avoid this issue:

1. **Enable sticky sessions** in the load balancer to ensure that all requests from one user go to the same application server.[83]
2.

2. **Serve all assets from a CDN.** Teams push new assets to the CDN before an application deployment. The CDN contains new and old assets.

These fixes reduce the likelihood of the above-described error, but they aren't perfect mitigations. **Sticky sessions are not a guarantee.** When an application server goes down due to a fault or redeployment, users must switch to another application.

The CDN solution also doesn't solve all problems. Not only do you need to ensure that all asset files are present. You also have to guarantee that the fragment markup is compatible with the loaded JavaScript and CSS files. If you ship the new markup with the fancy x-mas teaser, but an old stylesheet is loaded that doesn't contain the associated styles, the site will not look christmasy but broken. We have to find a way to ensure that markup and asset references always fit together.

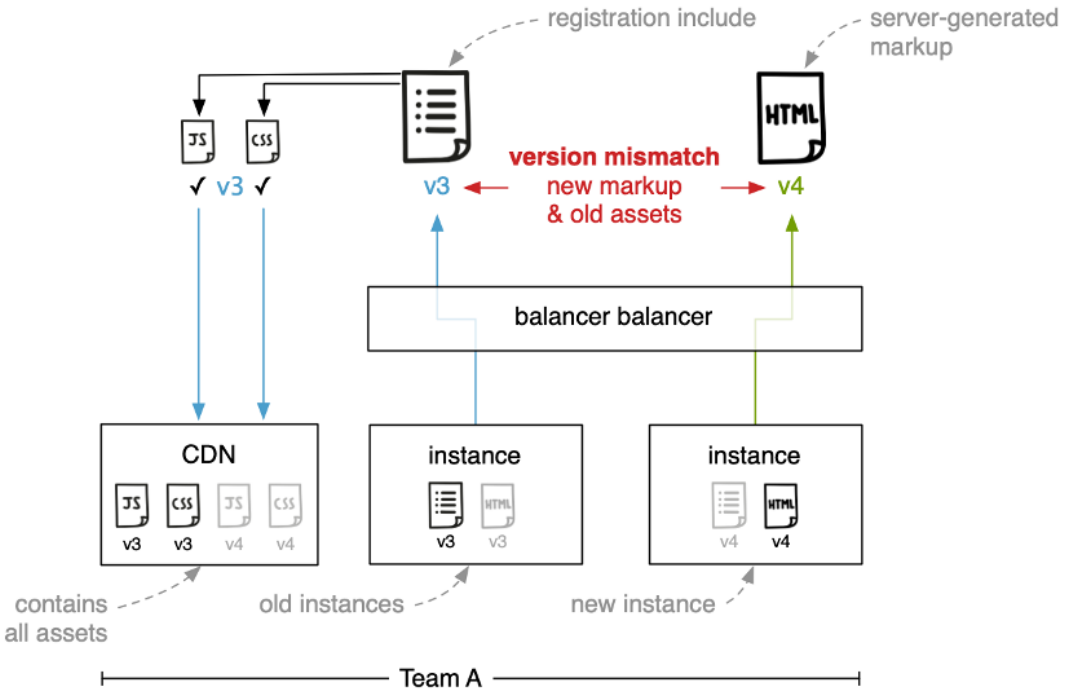


Figure 10.5 In this diagram we use a CDN that contains old an new assets. The CDN ensures that fingerprinted asset requests can always be resolved. However, since the registration include and the actual server-generated markup are retrieved in two separate request a version mismatch can occure. Here the old instance (v3) serves the registration include but the actual content is generated by a new version of the application (v4). This results in a version mismatch which may lead to errors in the browser.

NOTE Synchronization is primarily an issue for server-generated markup. When you run a fully client-rendered application, the HTML template is part of the JavaScript file anyway. If you are using a CSS-in-JS solution, the styles are most likely also part of the JavaScript bundle. Otherwise, you can ship the `script` and `link` tag via the same registration fragment to ensure that they are compatible with each other.

10.1.6 Inlining

The easiest way to ensure synchronization is to embed them into the markup of the fragment itself. Let's say *Team Checkout* generates the buy button markup server-side. Then they could ship the link and style tags directly into the requests that respond with the button markup. It could look like this:

Listing 10.5 team-checkout/fragment/buy-button.html

```
<link href="/checkout/static/fragment.a98749.css" rel="stylesheet" />
<button>buy now</button>
<script src="/checkout/static/fragment.a62c71.js" async></script>
```

Inlining works but comes with a few issues:

- **Redundant link/script tags:** If you have a page that includes the buy button five times, you will get five identical link and script tags. If the resources are cacheable, browsers are smart and download the files only once.
- **More JavaScript execution:** Even though the browser would download the JavaScript once it would execute it again for every script tag. Double execution may introduce unforeseen issues and a higher CPU load.
- **Works for server-side integration only:** Since the style and script references are part of the server-generated markup, this solution won't work for client- or universal-rendered micro frontends.

If these tradeoffs are acceptable for you, inlining could be a viable and easy to build option.

10.1.7 Integrated solutions (Tailor, Podium, ….)

Most micro frontend libraries come with a solution to deal with assets. In 4.4. A quick look into other solutions we introduced Tailor and Podium. Let's see how they handle JavaScript and CSS.

TAILOR'S ASSET HANDLING

Zalando's Tailor transfers the asset references via an HTTP header. A team can specify the associated assets to a piece of server-generated markup via a `Link` entry. A response can look like this:

```
$ curl -v http://.../checkout/fragment/buy-button
HTTP/1.1 200 OK
Link: </checkout/static/fragment.a98749.css>; rel="stylesheet",   ❶
 </checkout/static/fragment.a62c71.js>; rel="fragment-script"     ❶
Content-Type: text/html
Connection: keep-alive

<button>buy now</button>                                          ❶
```

❶ List of required CSS and JS files.

❷ The HTML content.

Because references and markup are in the same request, we have no synchronization issues. The Tailor service assembles the page and keeps track of all references. In the final markup, it creates a `link` tag for all unique CSS files and loads the JavaScript via the require.js module loader.

PODIUM'S ASSET HANDLING

With Podium, a team defines it's asset references in a `manifest.json` file. The manifest also contains a version number. *Team Checkout's* manifest for the buy button could look like this:

```
$ curl http://.../checkout/fragment/buy-button/manifest.json
{
  "name": "buy-button",
  "version": "4",                                          ❶
  "content": "/checkout/fragment/buy-button",              ❶
  "css": [                                                 ❶
    { value: "/checkout/static/fragment.a98749.css" }      ❷
  ],                                                       ❷
  "js": [                                                  ❷
    { value: "/checkout/static/fragment.a62c71.js" }       ❸
  ]                                                        ❸
}
```

❶ Deployed version of the software. Typically a build number or commit hash.

❷ Endpoint that returns the markup.

❸ List of associated assets.

Team Decide would use Podium's layout library and provide it with the manifest.json URLs for all micro frontends the product page needs. On startup podium, downloads all manifest files to determine the endpoints for the `content`. These endpoints respond with a plain HTML:

```
$ curl -v http://.../checkout/fragment/buy-button/
HTTP/1.1 200 OK
Content-Type: text/html
Connection: keep-alive
podlet-version: 4   ❶

<button>buy now</button>   ❷
```

❶ Version number of the application.

❷ The HTML content.

The response also includes a `podlet-version` header. It does not indicate the version of the podium library. It is a string that uniquely identifies the deployed version of the software. The owner of the fragment (aka podlet) has to set it explicitly. It can be a build number or a commit hash. In our example, the version number is `"4"`. It's the same number you find in the manifest.json above.

Every time Podium fetches the HTML content, it compares the `podlet-version` header to the version number in its cached manifest.json file. If the versions match, it can use the assets files specified in the current manifest. A difference in version numbers indicates that the owner of the

fragment has deployed a new software version. Podium will redownload the manifest.json to get a link to the updated assets.

Listing 10.6 team-decide/server.js

```
...
const buyButton = layout.client.register({                          ❶
  name: 'buy-button',                                               ❶
  uri: 'http://.../checkout/fragment/buy-button/manifest.json'      ❶
});                                                                 ❸

app.get("/product/eicher", async (req, res) => {
  const button = await buyButton.fetch(res.locals.podium);          ❸
  console.log(button);                                              ❸
  console.log(button.css);                                          ❹
  console.log(button.js);                                           ❺
  res.send(`<h1>Eicher<h1>${button}`);
});
```

❶ Registering the manifest file for *Team Checkout's* buy button.

❷ Fetching the content for the button via a promise.

❸ The markup for *Team Checkout's* button (`<button>buy now</button>`)

❹ Array of the required styles (`[{href: "/checkout/static/fragment.a98749.css", …}]`).

❺ Array of the required scripts (`[{src: "/checkout/static/fragment.a62c71.js", …}]`).

The above code shows how *Team Decide* registers *Team Checkout's* buy button micro frontend and fetches the content. Podium performs the synchronization and manifest updating under the hood. *Team Decide* waits for the Promise (`buyButton.fetch`) to resolve and receives the HTML and the assets in one object (`button`). This object contains the HTML as well as the associated asset reference. *Team Decide* can use it to construct its page markup.

10.1.8 Quick Summary

Now you've seen a couple of strategies to pull the required assets for all included micro frontends into your page. As with the markup integration strategies, there are no right and wrong solutions. It depends on your use case. How important is performance and caching? Do you need perfect synchronization, or is it practical to write your CSS and JS in a backward-compatible manner? For the projects I've worked on, the server included registration fragments with assets served from a CDN worked out fine for us. Table 10.1 summarizes the loading methods and lists their features.

Table 10.1 Properties of registration strategies

Method	Team Autonomy	Caching & Performance	Synchronization
Direct	✘	✘	✘
Redirect (client)	✓	✓	✘
Include (server)	✓	✓ ✓	✘
Inlining	✓	✘	✓
Integrated (Tailor, Podium, ….)	✓	✓ ✓	✓

Whichever concrete solution you choose, it's essential to define a uniform way that all teams use. A producer of a micro frontend must be able to count on the fact that the page owner references his assets correctly. He also needs to be able to update his assets without manually notifying other teams. Table 10.2 shows the technical contracts between a micro frontend owner und user. What does *Team A* have to know about *Team B's* micro frontend to use it?

Table 10.2 Contract for loading required assets

Method	Inter-team Contract	Example
Direct	asset file URLs	`/checkout/fragment.js /checkout/fragment.css`
Redirect (client)	asset file URLs	`/checkout/fragment.js /checkout/fragment.css`
Include (server)	endpoints with registration markup	`/checkout/register_scripts` `/checkout/register_styles`
Inlining	none *(only for server markup)*	
Tailor	HTTP-Header	`Link: <fragment.css>; <fragment.js>`
Podium	manifest.json	`/checkout/manifest.json`

10.2 Bundle granularity

We've talked about how to load the assets for your micro frontends. Now let's have a look at the files itself. Which granularity should the asset files have? One per micro frontend, one per team, or even a single huge one for the complete project?

10.2.1 HTTP/2

Best practices change over time. Some years ago, it was crucial to load as few resources as possible to keep the number of network requests down. Bundling up everything in one file and combining multiple images into one (spriting) was widespread. A couple of years later, tools like Google PageSpeed heavily rewarded inlining the CSS for the viewport into the HTML to ship everything needed for the first render in only a few TCP packets.

With the introduction of HTTP/2, these best practices became bad practices. The protocol reduced the overhead cost of loading multiple resources from the same domain. It's builtin multiplexing and server push features removed the need to manually inline assets into the page, which reduces complexity in the application and is also great for cacheability.

These HTTP/2 features come in handy when you are building a micro frontends style application.

10.2.2 All-in-one bundle

In 2014 I worked on my first project with vertically organized teams. Back then, we had lengthy discussions about the necessity to build an overarching asset bundling process. This bundling service would collect the scripts and styles from all teams to combine them into one single file for delivery. Luckily we decided not to introduce such a central service, but I know of other projects that did that.

A central asset bundler introduces a significant amount of coupling and friction. Someone needs to build and maintain that service. Deployments have to be synchronized between the asset service and the applications to ensure that the markup always matches the delivered assets. Today it's an anti-pattern to deliver all-in-one bundles for most use cases:

1. The cost of **shipping a lot of unused code** outweighs the gains of using fewer requests.
2. The **chance of cache invalidation** is high. The complete bundle needs to be redownloaded even if only a small part changed.

But even today, a central bundler can provide one valuable feature: **elimination of redundant code**. When two teams use the same JavaScript library or button styling, the central service could remove one instance of it to make the bundle smaller. In the next chapter 11 we'll discuss options on how to remove redundancy without introducing a shared service.

10.2.3 Team bundles

In our example application, each team has one **page- and fragment-bundle**. For the product page, *Team Decide* loads their page-bundle. If they want to include *Team Checkouts* buy button micro frontend, they would also need to add *Team Checkout's* fragment-bundle.

Since HTTP/2 makes additional requests very cheap but not free, **you should still use bundling inside your team and not confront the browser with your raw component- and dependency-tree.** In the projects I worked on, the bundle-per-team approach has proven itself as a reasonable tradeoff between bundle size, over-fetching, and reusability across pages.

But as always, it depends on your use case. If one team provides a fragment that requires a lot of CSS code, but only one niche page uses it, it might make sense to create a separate bundle for it.

10.2.4 Page & fragment bundles

The **one bundle per micro frontend** approach takes granularity one step further. There every fragment or page has its script and style bundle. You can think of this as adding an import statement to the top of your file before you can use the actual component in your code.

This more fine-grained way of bundling ensures that your only download code the customer needs on the page. But depending on your page structure and the number of fragments you include, it could lead to quite a few assets that need to be loaded.

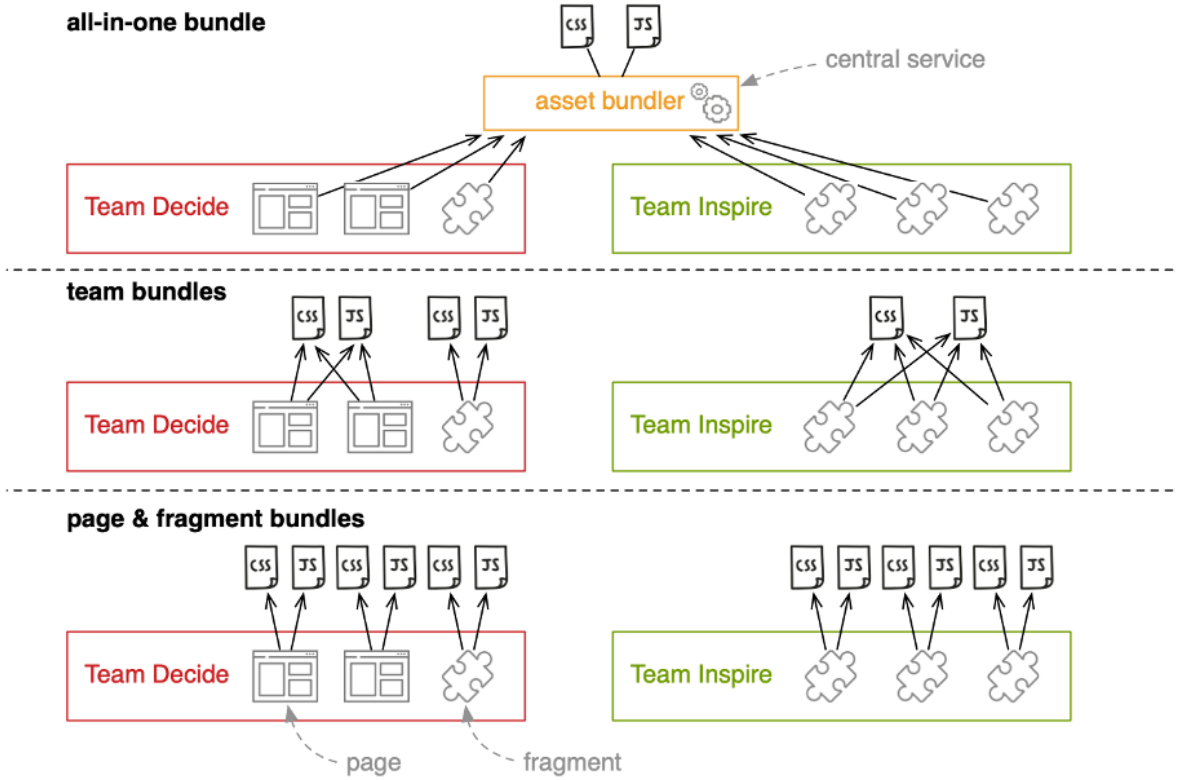


Figure 10.6 Different asset bundle granularities.

Figure 10.6 illustrates the three bundling strategies. Pick the strategy that fits your needs best.

10.3 On-demand loading

Picking the bundle granularity is essential because it affects the contracts between the teams. **But not all code has to live directly in this bundle.**

A team can adopt techniques like code-splitting inside of their bundle to further improve the loading behavior: Reducing initial download size and fetching parts of the code when the user needs them.

Say *Team Checkout's* buy button would open a fancy layer that requires a bunch of JavaScript. They could take the layer code out of the initial fragment bundle and fetch it when the user hovers over the button.

10.3.1 Proxy micro frontends

But we could reduce the bundle size even further. Say your asset file contains the code for five different micro frontends, which are rarely used together on one page. Instead of putting the code directly into the file, you can **setup proxy components that fetch the real code when it's needed the first time**. If you are using Custom Elements the code could look like this:

Listing 10.7 team-checkout/static/fragment.js

```
class CheckoutBuyProxy extends HTMLElement {
  constructor() {
    import("./real-buy-button.js").then(...);                    ❶
  }
}
window.customElements.define("checkout-buy", CheckoutBuyProxy);
```

❶ Dynamically loads the real implementation of the buy button when needed for the first time.

> **WARNING** Proxying a Custom Element is more complex than this example. It's currently not possible to update a Custom Element definition solely by registering a new class, and the lifecycle methods are synchronous. But we can't go more in-depth into Web Component land in this book. You'll find resources for doing this properly on the internet.

Shipping micro frontend proxies in your asset bundle can reduce initial download, which is good.

10.3.2 Lazy loading CSS

If you are using plain CSS, lazy loading is not that easy because there is no native browser support to split and load CSS files dynamically. But many CSS-in-JS solutions, CSS Modules, and most bundlers come with mechanisms to enable lazy loading for CSS without a bunch of manual work.

These are standard performance optimization techniques you would also use in a monolithic frontend. In a micro frontend architecture, each team can adopt them for their part of the system.

10.4 Summary

- Teams must be able to update their assets without having to coordinate with other teams.
- The asset paths must be part of the contract between the teams. A team that uses a micro frontend needs to add the associated assets.
- There are different ways for communicating the asset URLs: via documentation, through a redirect or registration include, via HTTP headers or a machine readable manifest file.
- If you render on the server you need to ensure that the JavaScript and CSS files match the version of the generated markup. For pure client-side rendering this is less of an issue since the template is part of the JavaScript itself.
- The development teams must implement performance optimization techniques like on-demand loading inside their applications. Try to avoid overarching optimizations like a shared asset bundling service. They introduce extra coupling and complexity.

11 Performance is Key

This chapter covers

- Examining how to measure performance when multiple micro frontends exist on one page.
- Pointing out how to find regressions and bottlenecks and attributing them to the right team.
- Illustrating the typical performance drawbacks that are consequential to the micro frontends architecture.
- Reducing the amount of required JavaScript by sharing larger vendor libraries across teams.
- Examining multiple techniques to implement library sharing without compromising team independence.

In 2014 my colleague Jens handed me an article.[84] written by a company that implemented a vertical style architecture. Back then, the term Micro Frontends didn't exist. Being a frontend developer who takes pride in delivering fast user experiences, my first gut reaction to this idea was rejection - strong rejection: *"Five teams that all roll their own frontend? This sounds like a lot of overhead. The result will surely be inefficient and slow."*

Today, when I introduce Micro Frontends to developers, I often get a similar reaction. They understand the concepts, and it's benefits, but sacrificing performance for increased development speed seems like a compromise that's hard to swallow. Having worked in Micro Frontend projects over the last years, my initial worries faded quickly. This does not mean that my concerns were unfounded or magically resolved themself. **Autonomy inherently comes with the cost of accepting redundancy.** But I learned to focus on the bottlenecks that have a real impact for our users instead of reflexively fighting code duplications.

The Micro Frontend projects we built all outperformed the monolith they replaced. Resulting in faster responses, less code shipped to the browser, and better total load times. One

factor these projects have in common was that architecting for excellent performance was a top priority from the start and not an afterthought. Another significant benefit I experienced while working with Micro Frontends is that the architecture makes it easier to optimize the user experience in the places where it makes the most significant difference. But more on this later.

In this chapter, you'll learn how to address performance in your Micro Frontends project. We'll start with the "definition of fast". What does "performant" mean for the different parts of your project? Measuring performance and acting upon the results is tricky when the frontend includes code from different teams. I'll show you some strategies that have proven valuable when architecting for excellent performance. At the end of the chapter, you'll learn how to keep your JavaScript overhead to a healthy minimum. Avoiding large redundant framework downloads while still being able to deploy autonomously.

11.1 Architecting for performance

Early on in the project, Finn, Model Tractor Inc.'s lead architect, arranged a meeting with developers from all three teams. Together they defined some performance guidelines that act as the default for all pages of the shop. They decided that the total weight of a page should never exceed 1MB of data. The viewport of a page must render in 1 second under good and 3 seconds under 3G network conditions.

11.1.1 Different teams, different metrics

They arrived at these values by looking at their competitor's websites. The team knows that excellent performance is vital for e-commerce. Users enjoy sites that feel fast. They spend more time browsing and have a higher chance of actually buying a tractor. But what does **feel fast** actually mean? Figure 11.1 illustrates this. Different parts of the site have different performance requirements:

- A user that opens the **homepage** for the first time mainly cares about **seeing the content without waiting**.
- On the **product page**, the **main image (aka hero image) is most important** and should be one of the first items to load.
- When the user enters the **checkout process**, it's all about interaction. Trusting the system while entering personal data. For that, **the software must react reliably and swiftly**.

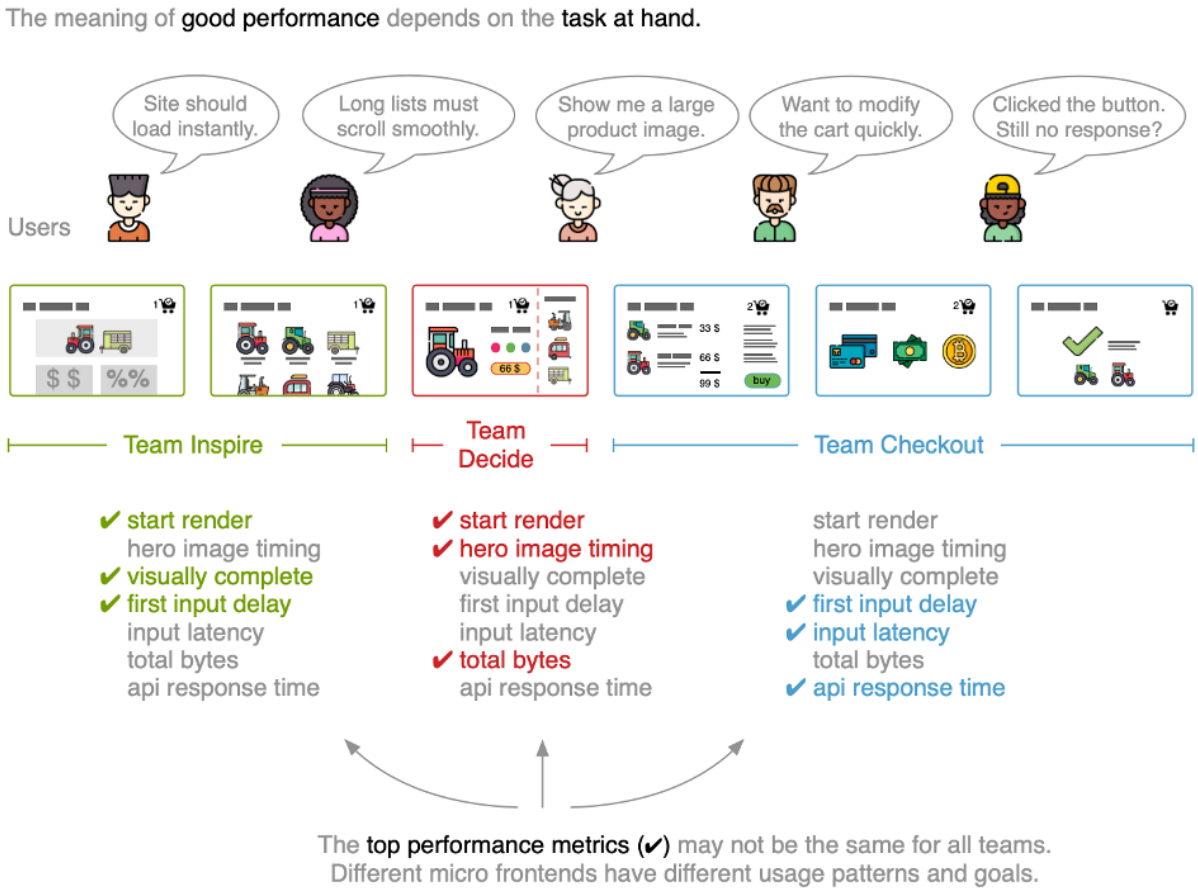


Figure 11.1 The metrics a team should optimize to depend on their use-case. The performance expectations of the homepage are not the same as the performance goals for the checkout process.

Having some overarching rules that act as a performance baseline is good. You can see them as basic hygienic requirements. But if you want to optimize further, the metrics a team should focus on will vary depending on the context the user is in. **Each team must understand the performance requirements of its subdomain and pick their own metrics.**

11.1.2 Multi-team performance budgets

Picking a metric and defining a concrete limit is also called a performance budget.[85]. Performance budgets are a perfect tool to establish a performance-oriented culture inside a team. The mechanism is simple:

- Your **team defines a concrete budget** for a specific metric. Say your site should never be bigger than 1MB.
- You **continuously measure** this metric to ensure your site stays in budget. Lighthouse CI, sitespeed.io, Speedcurve, Calibre, Google Analytics, etc. are useful tools for that.
- **If a new feature breaks your site's budget** the development stops. Developers investigate the cause of the degradation. Then the complete team, including product managers, discusses options to get back into budget: rolling back the change, implementing an optimization, or even removing another feature from the page.

Budgets are a powerful mechanism for fostering performance discussions on a regular basis. But how can budgets work for a site with Micro Frontends from multiple teams? **Should Team A stop their development because Team B's Micro Frontend slows down the page?** Maybe!

You can address this in different ways:

1. **Dividing the budget** to all Micro Frontends. The analytical approach. This would mean that a page containing five Micro Frontends could e.g., use 500kb for the content of the page itself and grant each included Micro Frontend a budget of 100kb. Adding everything up will get us to our 1MB (500kb + 5 * 100kb) size budget. In theory, this works, and for metrics like bytes and server-response times, it's possible to measure and sum up the pieces like that. But for metrics like Load Time, Lighthouse Score, or Time to Interactive, it's not that linear.
2. **Page owners are responsible.** The social approach. Here budgets are always on page level. A page owner is in charge of staying in that budget. In our example, *Team Decide* would be responsible for the product page. The team's goal is to provide the user with the best experience possible. Should an included Micro Frontend use an unreasonable amount of resources *Team Decide* contacts the owner, explains, and discusses options. You can view an includable Micro Frontend as a guest that tries to be as well-behaved as possible.

We've had good experiences with the latter approach. It avoids getting into the weeds with too fine grain budgets: Should a recommendation strip consume 100kb or are 150kb more reasonable? **The responsibility is very clear.** When the product page is slow, it's *Team Decide* that needs to become active. Yes, the team might not have caused the issue, but it's their task to get back to a performant state by finding the cause of the problem and informing the right team.

For the page owning team, this might feel cumbersome at first. But in practice, this worked well for us. No team that provides an includable Micro Frontend wants to be the "slow kid" that's holding everyone back. Teams started to measure the performance of their Micro Frontends in isolation to detect regressions before they go to production.

11.1.3 Attributing slowdowns

Team Decide installed a large dashboard screen in their office area. It shows the performance of their system with live updating charts and big green numbers. One day the team came back from lunch and noticed that the average load time of the product pages main image tripled. Before the product images rendered in around 300ms, now it takes nearly a second. The team checked their last commits but didn't find any suspicious change that could have caused such an issue. They checked the site in the browser. It didn't look broken.

They figured that an included Micro Frontend from another team might be responsible. Since they use server-side integration, this slowdown can be due to a service that has issues producing its Micro Frontends markup (see 4.2. Dealing with unreliable fragments). They open up the centralized metrics system where they can see the response times of every endpoint in the

platform. None of the endpoints that are used to assemble the product pages markup showed anomalies.

Then they check their web performance monitoring tool. This tool opens the product page on a regular basis with a real web browser. Recording a video of the process and also storing the browsers network graph. With this tool, the team was able to compare the product page from before lunch with the current slower version. This before-and-after comparison highlighted the real issue. Before the user loaded four images. *Team Decide's* big hero image and three images from the recommendation strip. Now the network graph shows 13 images. With this information and a little bit of digging, it became apparent that the recommendation strip was the cause of the slowdown.

It turned out that *Team Inspire* implemented a carousel feature for their recommendations. Now users can tap on a small arrow to see more matching products. But this simple carousel implementation did not feature any lazy loading. Even though the user only sees three recommendation images at a time, all images from the carousel load upfront. Jeremy, *Team Decide's* product owner, walked over to *Team Inspire's* office space and explained the issue. *Team Inspire* rolled back their carousel feature. They implemented lazy loading for the images and reintroduced the optimized version the next day.

OBSERVABILITY

Debugging a distributed system is a challenging task. The root of a problem is not always visible. Investing in proper monitoring will make finding issues much more manageable. If you integrate markup on the server, it's crucial to know how long the different parts of the page take to produce. Having a central view with the deployments of all teams can help to correlate a measured effect to a specific change in the system.

Monitoring the code that runs in the browser can be tricky. The software from all teams has to share bandwidth, memory, and CPU resources. Having video recordings, network graphs, and metrics you can compare over time is a vital first step. Implementing unique team prefixes for all resources the browser loads also helps (see 3.2.2. Namespacing resources). This way, the ownership of a file that looks suspicious is always evident.

ISOLATION

A popular debugging technique is isolating the issue. Imagine you've written a piece of software and notice a mysterious bug that you can't explain. A good strategy in finding the root cause is to comment out parts of your code and check if the bug is still present.

You can apply the same approach to a Micro Frontends website. The "Block URL" feature in the network tab of your favorite browser is your friend. Block the scripts or styles from a specific team. This way, you can test your site without the code of *Team B* or *Team C* and measure if the

performance degradation or error still exists.

11.1.4 Performance benefits

I often talk about performance challenges the Micro Frontends concept introduces. But there are a couple of positive properties this approach comes with.

BUILT-IN CODE SPLITTING

With the move to HTTP/2, it has become a best practice to split an application's JavaScript and CSS code into smaller pieces. We talked about this in 10. Asset loading. Delivering the code in smaller chunks (per team, per Micro Frontend) and not in one monolithic blob has benefits:

- **Cacheability:** Browsers only need to redownload the parts of the code that changed. Not everything. Micro Frontends are often used in conjunction with continuous delivery, where teams deploy to production several times a day.
- **Fewer long-running tasks:** The browser's main thread becomes unresponsive as it processes a JavaScript file. Loading multiple smaller files gives it more room to breathe and accept user input in-between processing the JavaScript resources.
- **On-demand loading:** Since the assets are often grouped by team or Micro Frontend, it's easy only to include the code a page needs or implement route based loading as we saw in the single-spa example. The user doesn't have to download the code for the cart page when he visits the homepage.

These benefits are by no means exclusive to Micro Frontends. You can achieve these optimizations also in a good architected monolithic frontend. But the way you think about and develop features in a Micro Frontends project naturally guides you towards this structure.

OPTIMIZING FOR THE USE-CASE

Developers working in a Micro Frontends team have a much more narrow scope. A team focuses on one specific set of use-cases to help the customer. It's in the interest of the team to optimize this use-case as much as possible.

Let me give you an example. Imagine *Team Inspire* is responsible for displaying promotional teasers in different areas of the shop. These images or videos are often large in size and have a considerable performance impact. Since the team controls the complete process from teaser creation, uploading, and delivery, it's easy for them to experiment with new file formats like WebP, AV1, or H.265 to speed up teaser loading.

They don't have to think about what a format switch would mean for product images or user uploaded review videos. This focus on teasers allows them to move quicker. No big meetings with everyone who has an opinion about image or video formats. No grant rollout plans. No big business case calculations. No compromises. The team has everything it needs to improve its teasers.

After the experiment, *Team Inspire* shares their learnings with the other teams. They're helping them not to fall into the same traps when they try something similar.

Being able to have this kind of focus and control is the biggest strength of a Micro Frontends architecture. It can not only lead to better web performance of the individual pieces. It improves quality and increases user focus.

EASIER CHANGES

The narrower scope makes it possible for a developer to know every aspect of the software. Something that is not possible in sizeable monolithic frontend projects. Have you ever deleted an old dependency that isn't in use anymore? Just to realize two days later, you broke an obscure marketing page you didn't even know existed? I definitely did.

Having clearly isolated Micro Frontends reduces the risk of cleaning up dramatically. This makes it easier to lose old craft and evolve the software.

11.2 Reduce, reuse … vendor libraries

The most discussed performance optimization topic for Micro Frontends is how to deal with libraries that are the same across teams. Downloading the same code twice triggers the Pavlovian reflex for all frontend developers: "This is inefficient, and we must avoid it!". But let's take a step back and question this reflex. We'll have a more analytical look at the topic of redundant code.

11.2.1 Cost of autonomy

Model Tractor Inc.'s three development teams all chose to go with the same JavaScript framework to build their frontend. Before they started the project lead architect Finn and the teams discussed three different options:

- **Alignment.** Everyone uses the same framework.
- **No constrains.** Every team can choose the framework they want.
- **Some constraints.** Free choice, as long as the framework has specific properties. Like having a runtime that's smaller than 10kb.

Each option has it's benefits and drawbacks. They went for the **everyone use the same** option for two reasons. Teams can help each other because they are all familiar with the same stack. Easier recruiting: developers switching teams get up to speed quickly, and human resources can use the same job profile for all teams.

Although all teams start with the same stack architect Finn emphasized that this decision is not set in stone. It should be possible for a new team to pick another stack if there are good reasons.

The same goes for version upgrades or migration to a newer, better framework in the future. Teams must maintain their autonomy. **All integration techniques and architecture level artifacts have to be technology agnostic.**

They use the JavaScript framework to generate server-side markup. However, for interactive features, the framework also needs to run in the browser. Each team has it's own git repository and a dedicated deployment pipeline. The team's JavaScript bundler builds an optimized asset file that's self-contained. It includes everything that the team uses. If teams use the same dependency, the client will download it multiple times. We can optimize this by providing a large framework code as a separate download from a central place. See the example in figure 11.2.

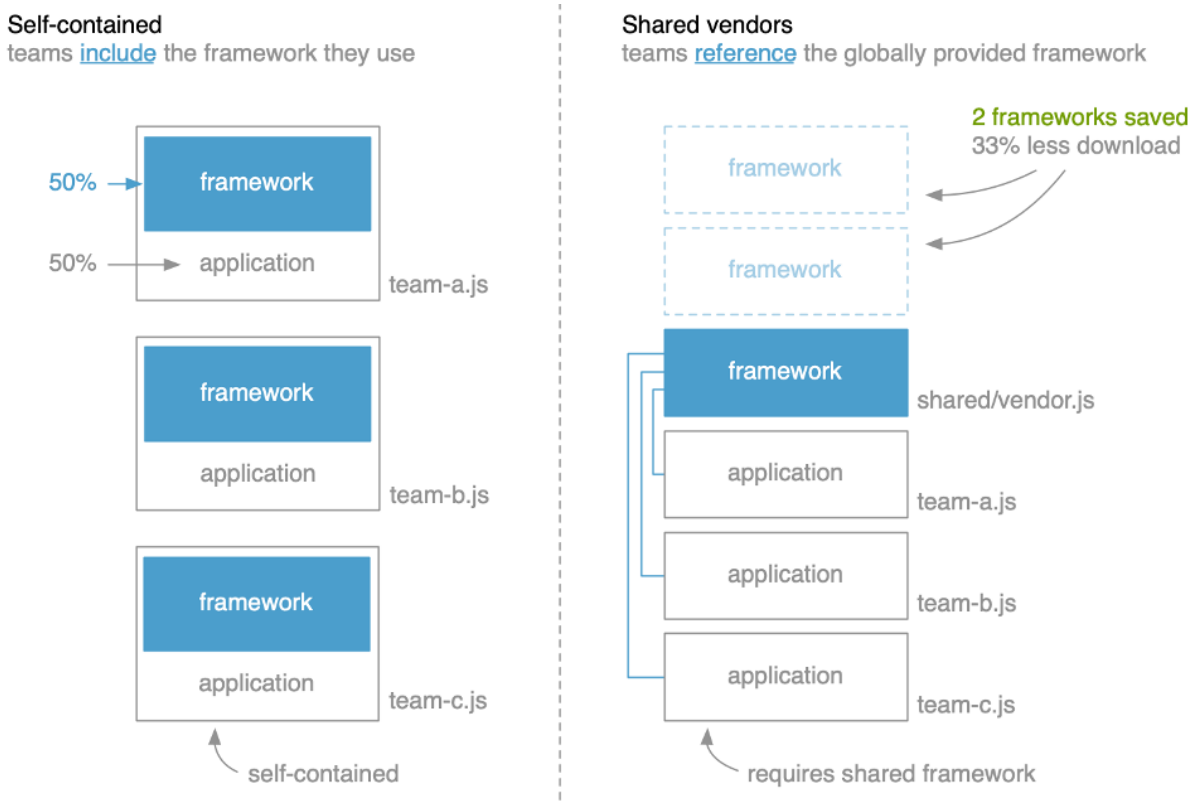


Figure 11.2 A teams JavaScript should be self-contained. It should be able to function on its own. That's why bundling all dependencies and vendor libraries is the easiest option (left side). When all teams use the same framework, it could be a worthwhile optimization to host the framework code on a central place (right side). This reduces the amount of network traffic and lowers memory footprint and CPU usage on the user's device.

It shows three teams that use the same framework. In our case, the framework code makes up for 50% of the team's bundle size. Removing the framework from the team bundles and providing it from a central place decreases the JavaScript size by 33%. The user saves two framework downloads. This sounds like a good optimization, but before we go ahead and build this, we should have a look at real numbers and the project's demands.

11.2.2 Pick small

The amount of overhead obviously depends on the framework and other libraries you choose. Going with a large framework like Angular will increase the need to centralize vendor code.[86] Even though the major big frameworks have gained a lot of popularity, you can see a trend in adopting smaller libraries and frameworks.

Picking a stack like Preact, hyperapp, lit-html, or Stencil will reduce framework overhead. Tools like Svelte go even a step further. They don't have vendor runtime code at all. The source code gets transpiled to native DOM operations. This way, your JavaScript bundle grows proportionally with the features you build. No fixed costs introduced by the framework.

No worries, we won't go into the "What's the best framework?"-discussion. Comparing a batteries-included framework like Angular to the small templating library, lit-html is an apple to oranges comparison. However, since the individual Micro Frontends are smaller in scope, it can be a viable option not to pick an all-mighty framework that has you covered for everything the future can bring. It might be worth going with a leaner option that's better tailored to your use-case. If your bundle includes little vendor code, the overhead of loading it multiple times diminishes.

The other factor you should take into consideration is the team boundaries. **How much composition does a typical page require?** If you don't use composition at all and every team manages its own set of pages, there is no overhead when loading a page. You only have the disadvantage that vendor libraries aren't cached between the pages of different teams. But the importance of minimizing redundancy increases with the number of different teams that run a Micro Frontend on one page. Figure 11.3 shows a rough calculation that gives you an idea of how much code we are talking about.

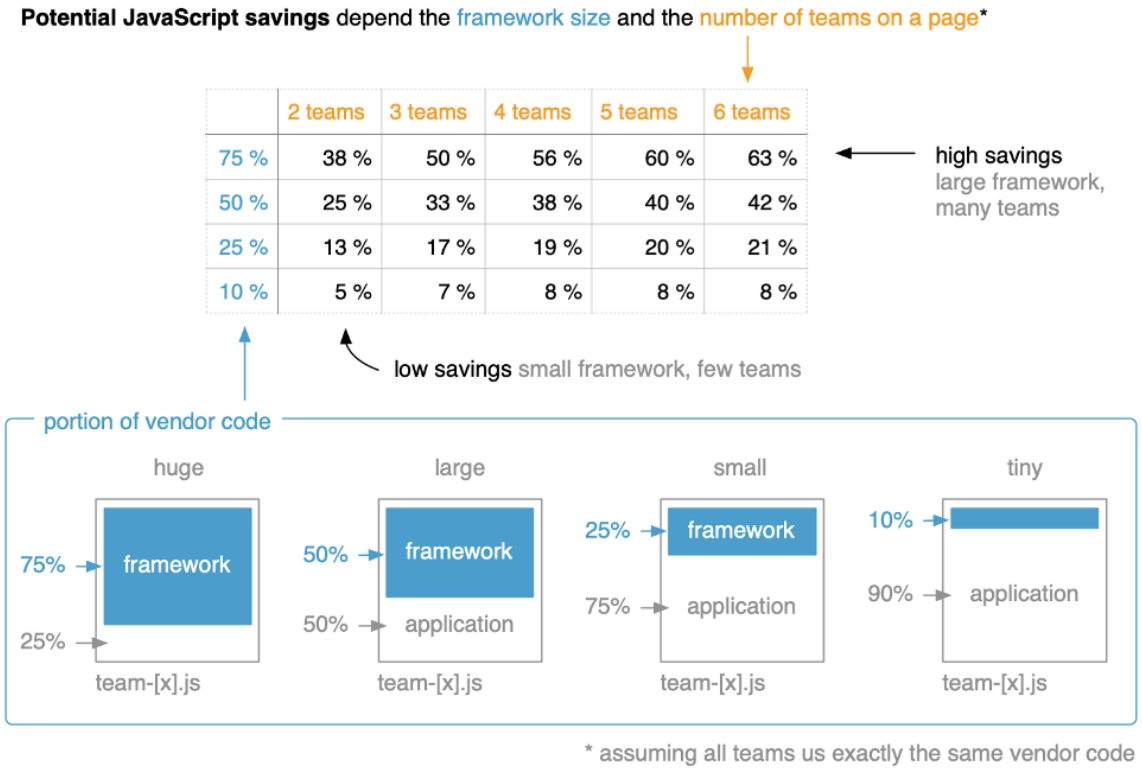


Figure 11.3 The potential savings depend on the portion of vendor code the teams include and the number of teams that are active on one page. Using small dependencies reduces the overhead noticeably. If multiple teams include larger libraries, you can save a lot of JavaScript by centralizing vendor code.

Now we have rough numbers to qualify the overhead in bytes. **It's still essential to measure the real performance implications for your use-case and target audience.** Intelligent on-demand loading and good code-splitting can make a more significant difference than shaving an extra 25kb of your JavaScript bundle.

11.2.3 One global version

The teams at Tractor Models Inc. decided that centralizing their framework code is an optimization worthwhile pursuing. They wanted to start with the most straightforward implementation possible. When we can assume that all teams are on the same version of one framework, we can use a low-tech solution:

1. Including the framework as a global script tag.
2. Excluding the framework from the team bundles and referencing it once from a global location.

The associated HTML code can look like this:

Listing 11.1 team-decide/index.html

```
<body>
  ...
  <script src="/shared/react.16.11.0.min.js"></script>
  <script src="/shared/react-dom.16.11.0.min.js"></script>

  <script src="/decide/static/bundle.js" async></script>
  <script src="/inspire/static/bundle.js" async></script>
  <script src="/checkout/static/bundle.js" async></script>
</body>
```

The React script tags attach their code to the `window` object. Teams can call it via `window.React` or `window.ReactDOM`. All bundlers provide an option to mark a library as "globally available". Webpack calls this concept `externals`. This removes the code from the bundle and replaces it with reference to a given variable. The configuration for Webpack can look like this:

Listing 11.2 team-decide/webpack.config.js

```
const webpack = require("webpack");

module.exports = {
  externals: {
    react: 'React',
    'react-dom': 'ReactDOM'
  }
  ...
};
```

Voila! That's it. We've eliminated the redundant framework code.

But **we've created a new central artifact** (`/shared/…`) which someone must maintain. Tractor Models Inc. decided not to instantiate a dedicated platform team *Team Checkout* volunteers to do the job. Making sure the files get deployed to the correct location and coordinating version upgrades with the other teams.

11.2.4 Versioned vendor bundles

The centralization worked well and improved performance measurably. Keeping the React version up-to-date was also not a big issue for *Team Checkout*. Every time a new React version came out, they informed all teams to test their software against it, deployed the new files to the `shared` folder two days later, and ensured that the markup references the new script.

But when React 17, the next major version, was announced, it became complicated. It included breaking changes requiring the teams to restructure parts of their existing software.

Team Checkout and *Team Decide* made the required changes to their codebase in the first week after the announcement. However, they were not able to deploy their migration because *Team Inspire* wasn't ready. This team was in the middle of a major rewrite of their recommendation algorithms to bring personalized product suggestions to the next level. Moving to the next

version of React at the same time was not an option for them. This task has to wait until the algorithm update shipped. So the other teams had no choice but to park their changes in a git branch and wait for the other team.

Three weeks later, *Team Inspire* managed to get their React migration done - paving the way to ship the new framework finally. The teams agreed on a day and time for the deployment of all software systems and the updated React library. Otherwise, the functionality of the site may be faulty if the central framework does not match with the application code.

This is what's often referred to as a **lock-step deployment**. If you're operating on a smaller scale, it might be fine to do such a manually orchestrated deployment from time to time. It gets extra fun when one team discovers that they need to roll back to the previous version because they found a severe bug. Then we have a lock-step rollback. These kinds of activities are exhausting and unsatisfactory. Furthermore, it contradicts the **autonomous deployments** paradigm of Micro Frontends.

A solution to this problem is to move away from one central framework to a versioned approach. Figure 11.4 shows a deployment process where two teams upgrade from Vue.js 2 to Vue.js 3:

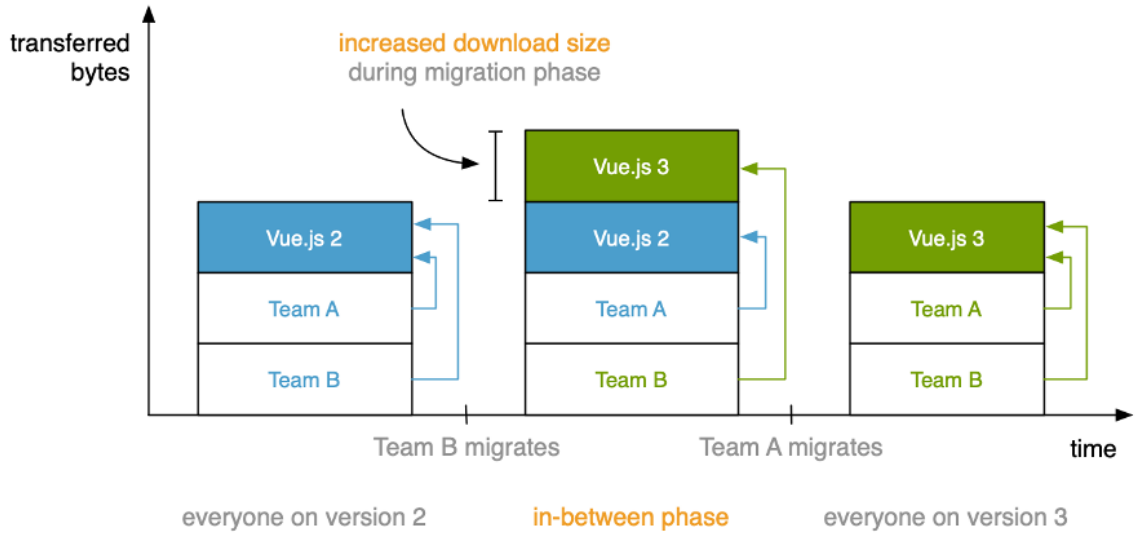


Figure 11.4 Illustrates a framework upgrade process. Team A & B both reference Vue.js v2 from a central location. The framework is loaded once. Now Team B migrates to Vue.js v3. Now the user has do download two Vue.js libraries (v2 & v3). In the last step Team A also migrates to the latest version. At this point both teams use Vue.js v3 and the user must only download one version.

1. Before the migration, both teams reference version 2.
2. Vue.js 3 gets published as a shared library.
3. *Team B* migrates first and deploys their software, which now references the new framework.
4. *Team A* migrates as well. Now both teams are on version 3. The old version 2 is not referenced any more.

With this approach, both teams can upgrade at their own pace. They control which version of the library their code references. Even a rollback is possible without having to coordinate with another team. The only drawback is that the total download size increases during the migration phase.

There are a lot of different ways of achieving this. Let's explore some possible solutions.

WEBPACK DLLPLUGIN

TIP You can find the sample code for this in the `19_shared_vendor_webpack_dll` folder.

The Webpack bundler enjoys great popularity. It includes a tool called `DllPlugin`.[87] It strangely lends its name from the dynamic link library concepts Windows users are familiar with. The plugin works in two steps:

1. You can **create a versioned bundle** with the shared dependencies. The plugin generates the **JavaScript**, which you can host statically **and a manifest** file. Think of the manifest as the table of contents for the vendor bundle.
2. You provide this manifest to the teams (e.g., via an NPM package). The teams Webpack configuration **reads that manifest**, **omits** all listed vendor **libraries** from the build and **adds references** to the versioned libraries of the central vendor bundle.

Let's have a look at the structure of the sample project:

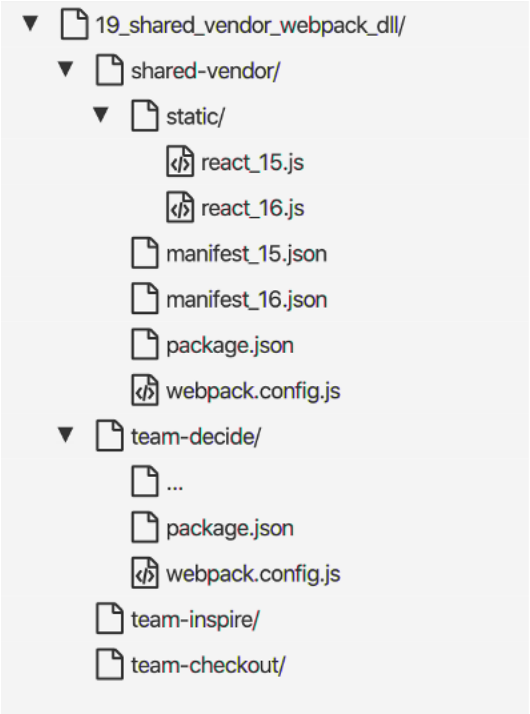


Figure 11.5 Folder structure of the Webpack DllPlugin example project. We've introduced a shared-vendor/ folder that sits beside the teams. It's the project that generates the shared vendor bundles (static/) using the DllPlugin. It also includes the manifest_[x].json files for each version. The teams use Webpack for packaging their application code. You can find the configuration in webpack.config.js.

The `shared-vendor/` folder contains the JavaScript and manifest code for version 15 and 16.

CREATING THE VERSIONED BUNDLE

We'll go through the essential pieces required to make this happen. Here an excerpt from the vendor bundles `package.json`.

Listing 11.3 shared-vendor/package.json

```
{
  "name": "shared-vendor",
  "version": "16.12.0",
  "dependencies": {              ❶
    "react": "^16.12.0",         ❶
    "react-dom": "^16.12.0"      ❷
  },                             ❸
  ...
}
```

❶ Specifying the dependencies and their version.

Here is the Webpack code for generating JavaScript and manifest.

Listing 11.4 shared-vendor/webpack.config.js

```
const path = require("path");
const webpack = require("webpack");

module.exports = {
  ...
  entry: { react: ["react", "react-dom"] },                ❶
  output: {                                                ❷
    filename: "[name]_16.js",                              ❸
    path: path.resolve(__dirname, "./static"),             ❹
    library: "[name]_[hash]"                               ❸
  },                                                       ❸
  plugins: [
    new webpack.DllPlugin({                                ❸
      context: __dirname,                                  ❶
      name: "[name]_[hash]",                               ❷
      path: path.resolve(__dirname, "manifest_16.json")    ❷
    })                                                     ❷
  ]
};
```

❶ List of dependencies to include in the vendor bundle. Here one bundled called `react` gets created. It contains the code of `react` and `react-dom`.

❷ Configuring location and name for the JavaScript code.

❸ Adding the `DllPlugin` and specifying where to write the manifest file.

USING THE VERSIONED BUNDLE

The teams must have access to the desired manifest at build-time. Publishing the `shared-vendor` project as an NPM module is one option to do this. Here is the `package.json` for a team:

Listing 11.5 team-decide/package.json

```
{
  "name": "team-decide",
  "dependencies": {
    ...
    "react": "^16.12.0",                    ❶
    "react-dom": "^16.12.0",                ❶
    "shared-vendor": "file:../shared-vendor"    ❶
  },
  ...
}
```

❶ Specifying the framework dependencies.

❷ Referencing the `shared-vendor` package. We use the `file:` syntax to make it happen locally. In the real project we would publish it as a properly named and versioned package like this: `@the-tractor-store/shared-vendor@16.12.0`.

The Webpack configuration of the team looks like this:

Listing 11.6 team-decide/webpack.config.js

```
const webpack = require("webpack");
const path = require("path");

module.exports = {
  entry: "./src/page.jsx",                                          ❶
  output: {                                                         ❷
    ...                                                             ❸
    publicPath: "/static/",                                         ❸
    filename: "decide.js"                                           ❸
  },                                                                ❸
  plugins: [
    new webpack.DllReferencePlugin({                                ❸
      context: path.join(__dirname),                                ❸
      manifest: require("shared-vendor/manifest_16.json"),          ❸
      sourceType: "var"                                             ❸
    })                                                              ❸
  ]
  ...
};
```

❶ Entry point of team decides application.

❷ Configuring where the generated files should go.

❸ Adding the `DllReferencePlugin` and pointing it to the `manifest_[x].json` of the `shared-vendor` package.

This is a pretty standard Webpack configuration. Adding the `DllReferencePlugin` is the special part. It performs the magic of omitting the code of all vendor libraries specified in the `manifest.json` and replacing it with reference to the central bundle.

Are you curious about the manifest's content? Let's have a look inside:

Listing 11.7 shared-vendor/manifest_16.json

```
{
  "name": "react_a00e3596104ad95690e8",                             ❶
  "content": {
    "./node_modules/react/index.js": {                              ❶
      "id": 0,                                                      ❶
      "buildMeta": { "providedExports": true }                      ❶
    },                                                              ❶
    "./node_modules/object-assign/index.js": {                      ❷
      "id": 1,                                                      ❺
      "buildMeta": { "providedExports": true }                      ❺
    },                                                              ❸
    ...
  }
}
```

❶ Unique internal name. Ensures that different DLLs can exist on one page.

❷ List of node modules that the bundle contains.

❸ The bundle also contains the dependencies of the dependencies.

The last step in our process is adjusting the script tags in the HTML to ensure that the bundles load in the correct order.

Listing 11.8 team-decide/index.html

```
<html>
  ...
  <body>
    <decide-product-page></decide-product-page>
    <script src="http://localhost:3000/static/react_15.js"></script>    ❶
    <script src="http://localhost:3000/static/react_16.js"></script>    ❶
    <script src="http://localhost:3001/static/decide.js" async></script>
    <script src="http://localhost:3002/static/inspire.js" async></script>
    <script src="http://localhost:3003/static/checkout.js" async></script>
  </body>
</html>
```

❶ Including the bundle for both React versions.

It's crucial that the vendor bundles execute before the team's code. Run the example locally (`npm run 19_shared_vendor_webpack_dll`) and look at the output at localhost:3001/product/fendt. Figure 11.6 shows the result. You can see that the Micro Frontends run on different React versions.

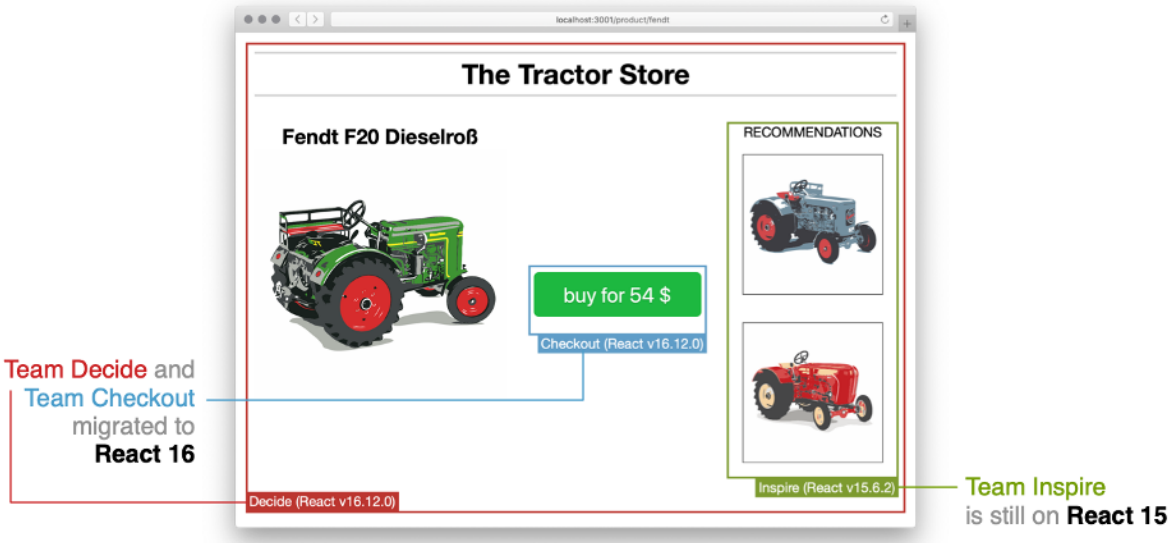


Figure 11.6 Team Decide's and Team Checkout's Micro Frontends run on React 16. Team Inspire still uses version 15. The different versions can coexist on the same page.

The DllPlugin has some **benefits** compared to the "One global version" approach:

- Safe way to globally provide **different versions of the same library.**
- A vendor bundle can contain **more than one library.**
- manifest.json is a **machine-readable and distributable documentation** of the vendor bundle.
- Works in **all** browsers.

But there are some drawbacks:

- **No on-demand or dynamic loading** of vendor assets. The vendor bundle has to be loaded before the application code that relies on it. The application code does not automatically pull in the vendor bundle it needs.
- **All teams must use Webpack.** The vendor bundle uses Webpack's internal module loading and referencing code.

> **NOTE** At the time of writing this book, there's a lot of work going on to improve Webpack's code sharing abilities across projects. Webpack 5 will introduce a technique called Module Federation.[88] addresses many micro frontends requirements.

Let's explore a third option that's based on JavaScript's new ES Modules standard.

CENTRAL ES MODULES (ROLLUP.JS)

Nowadays, relevant browsers (except Internet Explorer 11 and the popular Chinese QQ Browser).[89] support JavaScript's native modules system with the `import`/`export` syntax. This opens up new possibilities for sharing dependencies without needing a specific bundler.

> **TIP** You can find the sample code for this in the `20_shared_vendor_rollup_absolute_imports` folder.

Let's have a quick look at the capabilities of the `import` mechanism. The spec calls the dependency string, a **module specifier**. Here is a list of different specifier types:

- relative path (starts with a dot) `import Button from "./Button.js"`
- absolute path (starts with a slash) `import Button from "/my/project/Button.js"`
- bare specifier (simple string) `import React from "react"`
- URL (starts with a protocol) `import React from "https://my.cdn/react.js"`

> **TIP** If you want to learn more about ES Modules, I recommend this resource.[90] as a starting point.

In this example, we'll use the last option: the absolute URL. The concept of this example is the same as in the previous Webpack case:

1. We have a `shared-vendor` project that creates **versioned bundles** containing `react` and `react-dom`. But the files are now **standard ES Modules**.
2. We adjust the **team projects** to **reference** the vendor bundle by **using an absolute URL**.

In production, the code that runs in the browser works like this:

Listing 11.9 shared-vendor/static/react_16.js

```
export default [...react implementation...];
```

Listing 11.10 team-decide/static/decide.js

```
import React from "http://localhost:3000/static/react_16.js";
```

The central React JavaScript file is in ES Module format, and the teams point to it via a URL.

You could ship this code without using a bundler at all. For our example, we use rollup.js.[91] to ship `react` and `react-dom` in one bundle file and build and optimize our team's code for production. Rollup.js recognizes absolute URL dependencies (`http://..`) and leaves them untouched. This is something that isn't yet possible with Webpack.

We won't go through the full code but highlight the significant parts. This is the folder structure of the sample code:

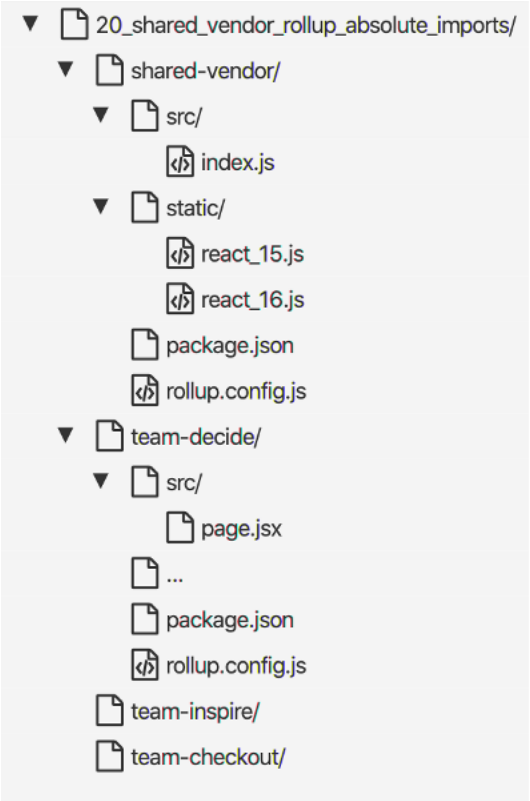


Figure 11.7 The shared-vendor project creates versioned bundles in ES Module format using rollup.js. The other teams also use rollup.js

CREATING THE VERSIONED BUNDLE

Rollup's configuration is straight forward. We define input and advise it to write the bundle as an ES Module (`esm`) to the `static/` folder.

Listing 11.11 shared-vendor/rollup.config.js

```
...
export default {
  input: "src/index.js",           ❶
  output: {                        ❷
    file: `static/react_16.js`,    ❸
    format: "esm"                  ❹
  },                               ❷
  plugins: [...]
};
```

❶ Input file specifies what should go into the vendor bundle.

❷ Output defines the target location of the bundle and sets it's format.

The `src/index.js` imports `react` and `react-dom` and exposes them as a default- and named-export. This way, Rollup will create one bundle which contains both libraries.

Listing 11.12 shared-vendor/src/index.js

```
export { default } from "react";
export { default as ReactDOM } from "react-dom";
```

That's everything we need to do to create the vendor bundle. As with the example before, the generated file will be available at `localhost:3000/static/react_16.js`. Let's look at how we configure the team's React applications to use this bundle.

USING THE VERSIONED BUNDLE

The team's rollup configuration is basically the same as the one we saw before: configuring input, output, and setting the format. It includes a few plugins to deal with JSX, Babel, and CSS, but these are all straight from the official documentation.

Listing 11.13 shared-vendor/src/index.js

```
export default {
  input: "src/page.jsx",
  output: {
    file: "static/decide.js",
    format: "esm"
  },
  plugins: [...]
};
```

Let's have a look inside the input file `src/page.jsx`. To use the globally provided vendor bundle, we need to set our imports accordingly. In a traditional react application, you would use a **bare specifier** like this.

```
import React from "react"
```

The bundler then searches for `react` in your `node_modules` and includes it. In our case, we can

specify the absolute URL.

```
import React from "http://localhost:3000/static/react_16.js";
```

Rollup.js will treat this as an external resource. Since all components in a React application need to import `react`, it's a little cumbersome always to write the absolute URL to the versioned file. In the example, I've used Rollup's alias feature.[92] to configure this in a central place. This way, the application code can stay as is, and Rollup replaces all instances of `react` with the absolute URL on build.

The absolute URL approach has two significant benefits:

1. **It's standards-based.** Asset sharing is an architecture decision that affects all teams. Changing it later on in the project will produce a non-trivial amount of work. Relying on standards makes future changes in tooling or libraries much more manageable. Want to switch your bundler? No problem, as long as it supports ES Modules.
2. **Dynamic loading of required vendor bundles.** The DllPlugin requires you to load the vendor files before the application code synchronously. With ES Modules, the application code requests the vendor bundle(s) it needs. If it's already downloaded because another Micro Frontend requested the same module, it reuses the existing one.

The dynamic loading makes the integration code a lot simpler. Here is *Team Decide's* HTML file:

Listing 11.14 team-decide/index.html

```
<html>
  ...
  <body>
    <decide-product-page></decide-product-page>
    <script src="http://localhost:3001/static/decide.js" type="module" async></script>      (1)
    <script src="http://localhost:3002/static/inspire.js" type="module" async ></script>    (1)
    <script src="http://localhost:3003/static/checkout.js" type="module" async ></script>   (1)
  </body>
</html>
```

(1) The HTML must only reference the JavaScript files from the teams. They download central bundles when needed.

Start the example locally by running `npm run 20_shared_vendor_rollup_absolute_imports`. In the first view, it looks exactly like the previous example. Two teams use React 16. One team is still on React 15. Opening up the developer tools shows a difference. In the network-tab, you see that the three application bundles load first (small parallel downloads). Then they request their associated vendor bundle (large parallel downloads). You can see this in figure 11.8. The **Initiator** column shows the team which first requested the bundle.

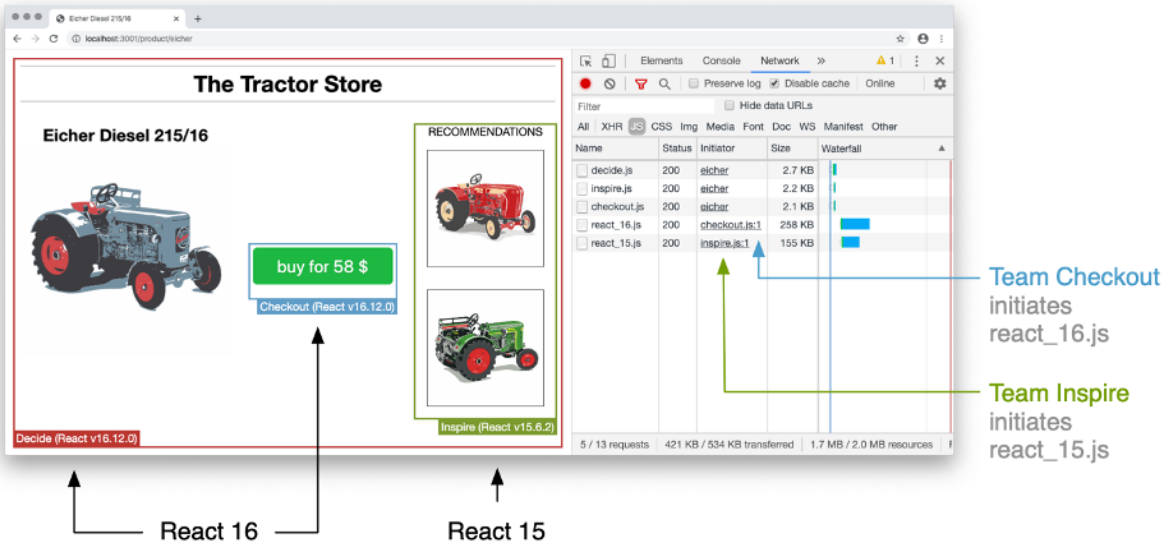


Figure 11.8 Different framework versions on one page by using ES Modules. The network-tab shows which team initiated the download of a specific vendor bundle. Team Decide and Team Checkout both reference react_16.js. Team Checkout was the first to request it. Team Inspire references the react_15.js bundle.

IMPORT-MAPS

In the previous example, we used Rollup's alias plugin to make our life easier. It saved us the hassle of using an absolute URL in all files that require `react`. The import-maps spec (editors draft).[93] might simplify this even further. It provides a declarative way to **map bare specifiers to absolute URLs**. An import-map looks like this:

```
<script type="importmap">                                    ❶
  {
    "imports": {
      "vue": "https://my.cdn/vue@2.6.10/vue.js",             ❷
      "vue@next": "https://my.cdn/vue@3.0.0-beta/vue.js"     ❷
    }
  }
</script>
```

❶ Introduces the new script-type `importmap`.

❷ Maps the bare specifier vue to the current version of the framework.

❸ Maps the bare specifier `vue@next` to the upcoming version of the framework.

The definitions from the import-map apply globally. Teams can reference the current version of Vue.js by importing vue. They don't need to know the URL of the shared bundle. The following example illustrates that:

```
<!-- Team A -->
<script type="module">
  import Vue from "vue";
  console.log(Vue.version);
  // -> 2.6.10
</script>
```

```
<!-- Team B -->
<script type="module">
  import Vue from "vue@next";
  console.log(Vue.version);
  // -> 3.0.0-beta
</script>
```

MORE ABOUT IMPORT-MAPS

Import-maps are a promising solution but not an official standard yet. Right now, the above code only works in Chrome when you've activated a feature flag.

If you want to use them today, I recommend having a look at **SystemJS**.[94] SystemJS maintainer and single-spa developer Joel Denning has published a video series.[95] on using import-maps and SystemJS with micro frontends.

Podium developer Trygve Li as written an introduction to **using import-maps in a micro frontend context**.[96] He also authored a rollup plugin.[97] that works similar to our alias approach but takes an import-map as an input.

11.2.5 Don't share business code

Extracting large pieces of vendor code is a powerful technique. You've learned a couple of ways to achieve it. But you should be careful of what to extract.

It's tempting also to share snippets of code every team uses like currency formatting, debugging functions, or API clients. But since this is business code and has a tendency to change over time, you should avoid that.

Having a similar piece of code in the codebase of multiple teams feels wasteful. However, sharing code creates coupling that you shouldn't underestimate. Someone has to be responsible for maintaining it. Changes to shared code have to be well thought out and appropriately documented. Don't be afraid of copy-and-pasting snippets of code from other teams. It can save you a lot of hassle.

If you're confident that it's a good idea to share a specific piece of code with other teams, you should instead do it as an NPM package that teams include at build time. Try to avoid runtime dependencies. They increase complexity and make your application harder to test.

In the next chapter, we'll talk about code that's often shared in micro frontends projects: the design system.

11.3 Summary

- Performance budgets are an excellent tool to foster performance discussions on a regular basis. They also form a shared baseline that all team members can agree upon.
- Having some project-wide performance targets is valuable. If teams want to optimize further, they might pick different metrics because they work on different use-cases. The performance requirements for the homepage are not the same as the requirements for the checkout process.
- Measuring performance is tricky when micro frontends from multiple teams exist on one site. Having clear responsibilities helps. The owner of a page can also be responsible for the overall page performance. If another team's micro frontend slows down the page, the page owner informs that team to fix the issue.
- It makes sense to measure the performance characteristics of a micro frontend in isolation to detect regressions and anomalies.
- Micro frontend teams have a narrower scope they are responsible for. This makes it easier for them to optimize performance in the places where it has the most significant effect on the user.
- The size of your JavaScript framework and the number of teams on a page have an impact on performance. Because teams have a smaller scope, it might be a viable solution to pick a lighter framework. This eliminates the need for vendor code centralization.
- You can improve performance by extracting large libraries from the team's application bundles and serving them from a central place.
- Sharing assets introduces extra complexity and requires maintenance.
- You should measure the real impact of redundant JavaScript code for your use case and target audience.
- Forcing all teams to run the same version of a framework can become complicated for major version upgrades. Teams have to deploy in lock-step to avoid breaking the page.
- Allowing teams to upgrade dependencies on their own pace is an important feature and can save a lot of discussions. You can achieve this by implementing versioned assets files that can work side-by-side. Use Webpack's DllPlugin or native ES Modules to implement this.
- Only centralize generic vendor code. Sharing business code introduces coupling, reduces autonomy, and can lead to problems in the future.

12 User Interface & Design System

This chapter covers

- Examining how a design system can help to deliver a consistent experience to your users.
- Discovering different approaches to develop a design system and how it can affect the autonomy of the Micro Frontends teams.
- Highlighting technical challenges when building a pattern library that should be technology agnostic.
- Distinguishing if a component should go into the central pattern library or stay under a product team's control.

In a micro frontend architecture, every team builds its part of the frontend. A team can plan, build, and ship new features without talking to its neighbors. But how do you deliver a consistent look and feel for the user? The different frontends should use the same color palette, typography, and grid layout. These measures ensure that the website does not look weird. But it typically doesn't stop there. There's also button styling, spacing rules, breakpoint definitions to support a variety of screen sizes, and a lot more.

Classical architecture discussions often dismiss these topics as unimportant. You hear sentences like: "We'll find a way to make it pretty afterward." However, in a distributed architecture like this, it's essential to have a proper plan for managing your design from the start. Throughout the book, you've learned techniques to avoid sharing code and keep teams as decoupled as possible. When it comes to design, it's not that easy. If you don't want to alienate your users, you need **a system to share your design building blocks with all teams**. A design system enables them to build interfaces that have a similar look. However, a design system also introduces coupling because every team has to be compatible with it.

In the micro frontends projects I've worked on, planning and setting up a shared design system

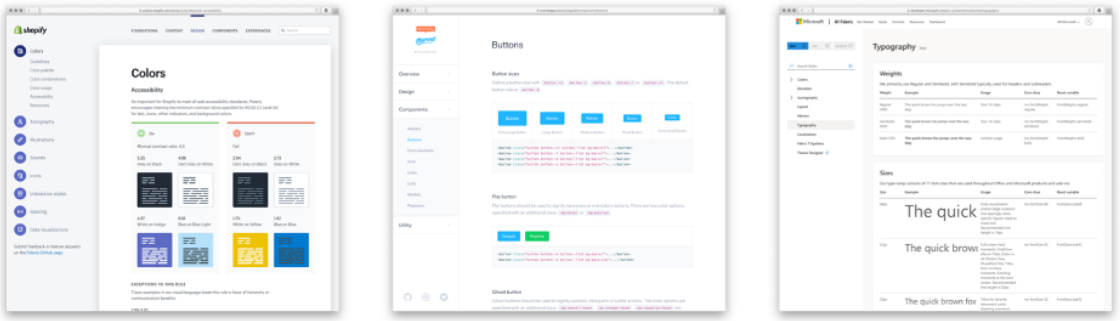
was always among the first and most important tasks. How to integrate a design system into the team's code is a much-discussed topic. It has direct implications on how teams build their frontend features. Changing these architecture decisions afterward is expensive because all user-facing features rely upon it.

In this chapter, we'll briefly introduce the concept of a design system, discuss how to organize effective development, and look at a variety of technical integration options and their trade-offs.

12.1 Why a design system?

Creating an overarching design that different teams can use is far from being a Micro Frontend specific thing. The term design system has become popular in software development in recent years. It provides a way to systematically tackle design in an era of growing web applications that must work on a multitude of devices.

A design system contains **design tokens** (fonts, colors, icons, …), **reusable interface components** (buttons, form elements, …), more **advanced patterns** (tooltips, layers, …) and most importantly a **well explained set of rules** on how to use these individual pieces together. Figure 12.1 shows some design system examples.



shopify Polaris | Marvel Styleguide | Microsoft Fabric

Figure 12.1 A lot of companies have published their design systems on the web.Design Systems Gallery designsystemsrepo.com/design-systems/. You can use them in your project or leverage them as a source of inspiration when creating your own.

Two other terms often also come up in this context: "pattern library" and "(living) style-guide". They mean the same thing: A way to modularize the complexity of the web with a component-based system. However, they have a slightly different focus.

The term **pattern library** describes a set of concrete building blocks developers can use. It is a library that contains tangible components like buttons and form inputs. It focuses more on the components than on the documentation aspect. You can say that a pattern library is a subset of a design system.

Style-guide is a traditional term from the design world. Before the internet, they came in the form of a well-crafted stack of paper describing all design rules for a company's corporate identity. The **"living"** prefix transferred this concept to the digital age, where the illustrated components use the real code.

In this chapter, we'll use the term "design system" when we talk about the broader concept and use the word "pattern library" when it comes to the technical integration with the team's applications.

In this book, we won't discuss how to build a design system. You can find a lot of excellent blog posts.[99], books.[100] and even hands-on checklists.[101] to get deeper into this topic. Instead, we'll focus on the design system aspects that are crucial to get right for running a successful micro frontends architecture.

12.1.1 Purpose and role

In a micro frontends project, all features the product teams create, are directly targeted to the end-user. These features make the user's life more enjoyable and thereby create value for the company. A centralized design system does not fit into this model.

No user signs up for Microsoft Office 360 because he thinks that Microsoft's Fabric Design System is the best. No question, the existence of the design system makes Office a more usable product. People who are familiar with using Word have an easier time understanding PowerPoint or Excel because all teams use the same UI paradigms and components.

A design system has an indirect effect that manifests itself through the product teams. The goal of a design system team should never be to create the most beautiful, best documented, or most versatile design system on the market. The objective of a design system team should be supporting the product teams as best as possible. **A design system is a product that serves other products.**

12.1.2 Benefits

A sound design system can help product development by providing these benefits:

- **Consistency:** Making user interfaces from different teams "feel familiar" to the user.
- **Shared Language:** A design system forces you to create a shared vocabulary that all involved parties understand. Proper naming is never easy, but having consistent names for your components and patterns improves communication across teams and avoids misunderstandings.
- **Development Speed:** Having clear guidance and the necessary UI components to build a new feature makes the developer's life easier.
- **Scaling:** The value of a design system increases by the number of teams using it. New teams have a solid foundation they can build upon. No redundant discussions on "if we should use a custom select box or not". Hopefully, the authors of the design system have

documented this decision before.

The benefits of a design system are mid- and long-term. Creating a robust system will take a considerable amount of time. However, if your project is of a particular size, these efforts will pay off quickly. It will also save you a lot of unsatisfactory design consolidations and eliminate chaotic redesign projects.

12.2 Central design system vs. autonomous teams

Now you know the basics of a design system, and it's benefits. Let's look at some aspects that are important in a micro frontends architecture. One question that's frequently asked is if it's indispensable to build your own design system.

12.2.1 Do I need my own design system?

Creating a design system is not an easy or cheap task. If you are building an internal product where branding is not an important aspect, it's, of course, perfectly fine to go with an off-the-shelf solution. Projects like Twitter Bootstrap.[102], Google's Material Design.[103], Semantic UI.[104] or Blueprint.[105] are great candidates. They all bring a set of generic components developers can adapt for their use case.

But you shouldn't choose a library by its appearance alone. They have different technical architectures that introduce constraints into your project. Some integrate solely via CSS classes (Bootstrap, Semantic UI), dictate a specific frontend framework (Blueprint), or provide a set of framework options (Material Design). Later in this chapter, we'll dive deeper into the possible integrations and their pros and cons.

If your product should convey a unique style and must be in line with your companies branding, it's a good idea to develop your own design system from scratch. Such a system also enables you to incorporate components that are unique to your business domain. In e-commerce, you want to have a price component that defines how reductions, sales, or the base price should render. When you are building a messaging application, you will want to include primitives like user avatars or chat-bubbles.

12.2.2 Process, not project

Having your own design system has some real benefits. We, as developers, like to focus on its technical aspects. Creating a set of usable components for all teams sounds like a worthwhile project. But in an otherwise distributed organizational structure, a design system also introduces an important social aspect. A former co-worker of mine likes to describe the design system as …

... the camp-fire by around which people from different teams and with different professions gather regularly.

– Dennis Reimann Creator of the open-source project UIengine

This quote highlights the fact that a design system is never a finished product. **It's better to think of it as a process.** A design system should be a living and evolving piece of infrastructure. The usable components and formalized design rules are the result of discussions between user-experience (UX) and design experts as well as developers and product owners from the teams. It should be the singles source of truth when it comes to design questions. Figure 12.2 illustrates this.

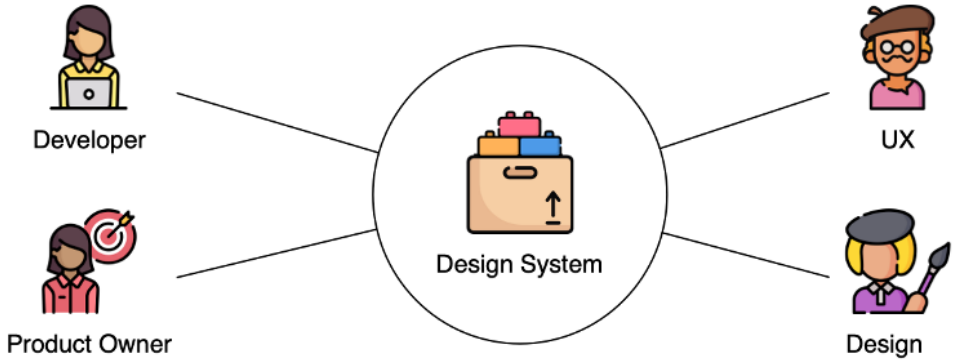


Figure 12.2 A good design system is a place where all key design decisions get documented. It's constantly refined to meet the needs of its users best.

12.2.3 Ensure sustained budget & responsibility

It's important to set appropriate expectations in management. The bulk of the design system work will be in the first months, but the work doesn't stop then. New use-cases arise, and teams develop new and more sophisticated features. You need free space to adjust and grow the design system accordingly. It's crucial to have a sustainable budget dedicated to doing this work:

- extending components
- questioning existing patterns
- refactoring areas
- refining the documentation
- fixing inconsistencies

I've seen projects with thoroughly crafted design systems that worked pretty well in the beginning. But when nobody is there who feels responsible or can maintain and evolve the system, it starts getting out of date. Teams work around existing patterns. They modify components with custom override styles to adjust them to their needs. Some components get extended several times and grow in complexity. Documentation is out of date.

From this point, it usually gets worse pretty quickly. That's what the community calls a zombie style-guide.[106]. **Don't let your design system join the zombie army.** Rebuilding and replacing a design system is expensive. The Micro Frontends architecture optimizes for feature development speed inside team boundaries (vertical). Introducing substantial changes across

teams (horizontal) requires a lot of coordination, creates friction, and can impair development for a considerable time.

Make sure to establish proper conditions in the first place. **Having a dedicated budget and strong responsibility is vital.**

12.2.4 Get buy-in from the teams

Getting the green light from management is an essential precondition, but it's even more important to have a healthy relationship with the product teams. They are the users of your design system. They are your customers. Take time to explain the design system, and it's concepts to them.

THE FIRST SPRINTS

Learn about their development roadmap and discuss wireframes to identify the components that are required first. A transparent development process helps the product teams to know when needed parts are ready. Publishing documentation, examples, and changelogs supports this.

In the early stages of a new project, the design system team is usually the bottleneck. There's a lot of technical setup to do. The team needs to build essential components for typography and interactions. Giving the design system team a head-start of a couple weeks is something we've had good experiences with. This way, the product teams can use the pattern library from the start. Nobody needs to wait or use temporary solutions that cause trouble later on.

ACCEPTANCE

Even though all teams know about the benefits of the design system, it's often tempting to work around it. Imagine *Team Decide* wants to ship a new product review feature. To build it, they need a new rating-star icon and a new smaller heading style. The team is already under pressure because the lead developer broke his arm in a sport accident a week ago. To keep the schedule, it would save time to add the icon directly to *Team Decide's* application code. They could take the standard heading and just overwrite it with a smaller font size. Yes, other teams wouldn't be able to use these components in their features. "But that's not important right now." These moves would save *Team Decide* time. They are avoiding discussions with the central UI team and then waiting until these changes are ready to use.

Micro frontend's main benefit is to empower teams to move fast by eliminating dependencies and waiting for other people. A central design system will get into the way. Having discussions around reusability and consistency doesn't help the product team's primary mission. **Make sure everyone understands this conflict of interest and recognizes the importance of the design system.** Find a way to spot technical debt, and don't let it build up.

COMMUNICATION

Establishing proper communication channels between the design system and the product teams is a crucial factor for success. There are a lot of ways to do this. It doesn't have to be regular in-person meetings. Being creative and coming up with light-weight solutions makes the process leaner and can build up acceptance.

We've experimented with a concept called "opening hours". The design team offers dedicated time-slots. Product teams can come in and discuss wireframes for upcoming features. They don't need to schedule a meeting. The goal is to identify changes for the design system in an early phase.

However, the method we found most effective is to **directly involve people from all teams in the development process** itself. Next up, we'll see how this can work.

12.2.5 Development process: central vs. federated

There's no single way to organize the development of a design system. Up until now, we implicitly talked about an organizational form that's called the **Central Model**. We have a dedicated team that plans and builds the design system and distributes it to the product teams to use. But there's another approach that's gaining popularity and fit's well into our autonomous-teams architecture: The **Federated Model**.[107] Figure 12.3 shows both models side-by-side.

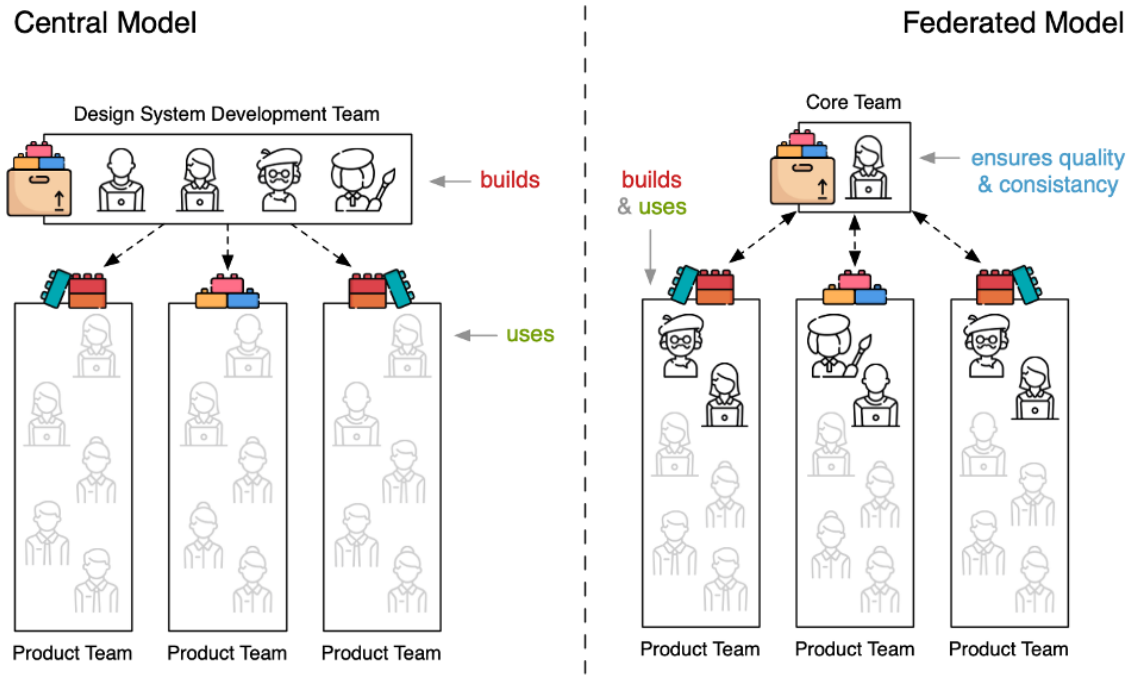


Figure 12.3 Two approaches for organizing the development of a design system. In the Central Model, a dedicated design team develops the system, and the teams use it. The Federated Model blurs the line between the design system and the product team. The members of the product teams contribute to the system and drive development.

THE CENTRAL MODEL

In the Central Model, we have a **clear division of labor**. A group of developers, designers, and UX specialists plan an build the design system. To know what they should build, they talk to the product teams. The design team has a pretty good overview of the complete system, can spot inconsistencies quickly, and works efficiently.

The **product teams are only users** of the pattern library. They make requests to the design system team and wait until their components are ready. The central team has the potential to become a bottleneck. When product teams request more changes than the design team can implement, it gets ugly. Teams have to delay their schedule or start working around the design system.

THE FEDERATED MODEL

The Federated Model changes this. **Designers and UX specialists move into the product teams. There's no real central team anymore.** Yes, we still need someone who stewards the design system and has an eye on quality and consistency. However, the product teams now drive the development of the design system themself. When a product team needs a new component, they design it, build it, and publish it to the design system for everyone to use.

This model gives the team a lot more freedom and autonomy. But since the design system is a shared project, its crucial to properly communicate changes to others. Running this model requires some skill and experience. It's most significant benefit is that UX experts and designers now work in the product teams. They bring new perspectives to the development team and can help to improve the product directly. This quote from Nathan Curtis.[108] brings it to the point:

> *We need our best designers on our most important products to work out what the system is and spread it out to everyone else. Without quitting their day jobs on product teams.*
> *– Nathan Curtis Founder of UX firm EightShapes*

12.2.6 Development phases

You might ask yourself what's the best model for your project. It's hard to give a general answer to this question. But I'll share what worked for us.

First of all, the two models aren't mutually exclusive. They blend very well. You don't have to pick one of the extremes. Running a design system with a strong central team does not mean that you can't take contributions from a product team. Figure 12.4 shows the Central-to-Federated Continuum.

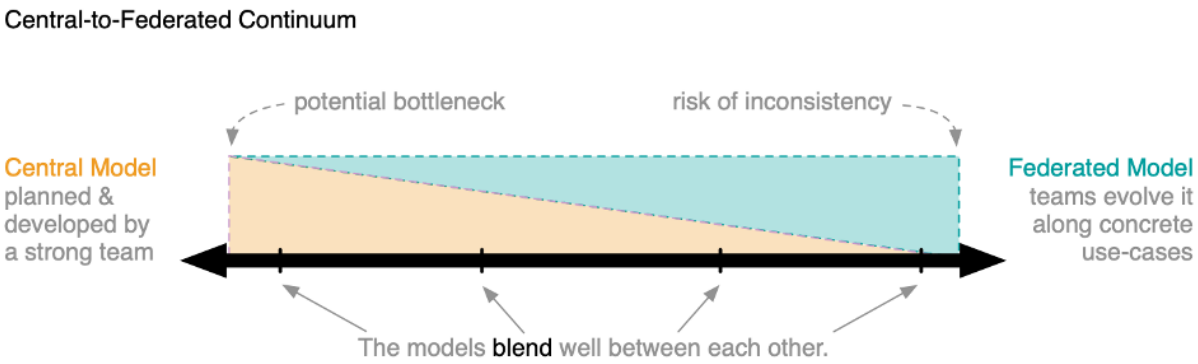


Figure 12.4 Central vs. federated is not a binary decision. The models work well together. The scale at the bottom shows the spectrum. You can run the central model with some federated aspects (left side). It's also possible to run the federated model in combination with some central planning and development (right side).

In our projects, I could observe two phases of design system development: the ramp-up phase (phase 1) and the production phase (phase 2). Figure 12.5 shows how these phases vary in focus.

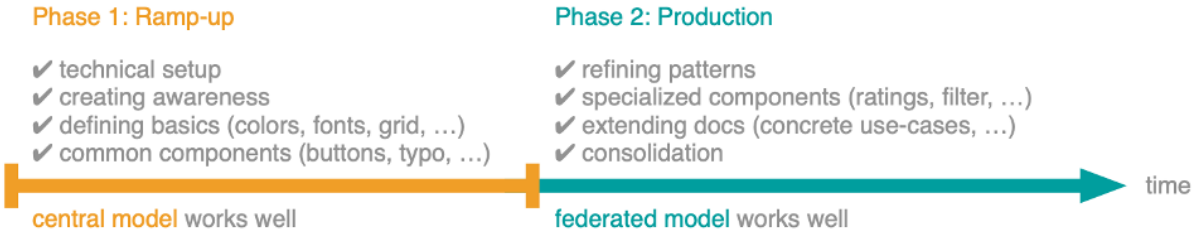


Figure 12.5 What model fits best can depend on the development phase your project is in.

When starting a new project, we've had good experiences with the central model. It's an efficient way to get a new design system off the ground. In this **ramp-up phase**, there's a lot of work to do. Setting up pipelines and tools, making initial decisions, and creating the first set of standard components. **Having a dedicated team that has no other responsibilities is valuable in this phase.**

When the dust has settled, and **teams start to get productive**, we slowly move towards the federated model. This way, we ensure that **the real use-cases drive the development**. We encourage frontend developers from the product teams to learn about the design system and contribute. Developers and designers from the design system team move towards the product teams. In this transition phase, it's common that people divide their time between two teams. A designer might spend 50% of his time on the design system and 50% on the product team. These percentages make planning easier. They can gradually shift over time.

12.3 Runtime vs. build-time integration

You've learned a lot about the organizational aspects. Let's see how we can technically integrate a pattern library with the team's applications. First, we'll talk about different strategies for rolling out changes.

Imagine you've changed the color of your button component in the central pattern library. What needs to happen so that the user can see it?

You can find two deployment approaches that people use: **Runtime Integration** and distribution as **Versioned Packages**. Figure 12.6 shows both of them side-by-side.

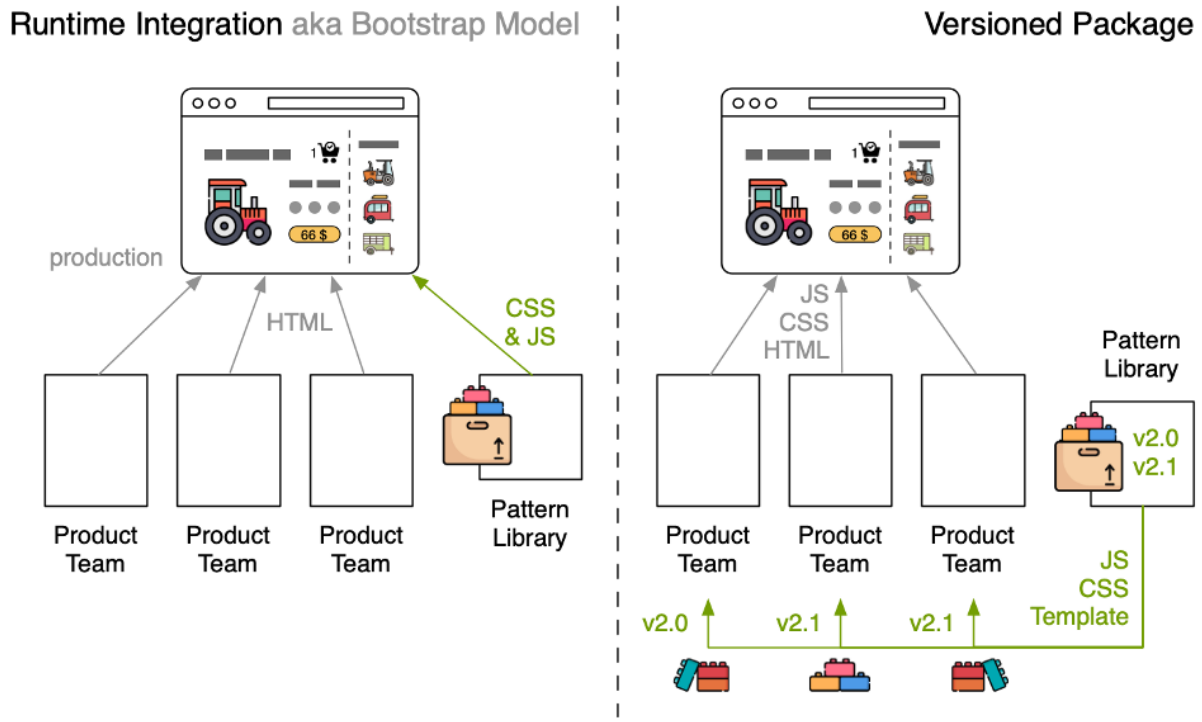


Figure 12.6 In the Bootstrap Model (left), the pattern library deploys its artifacts (JS, CSS, and possibly images) directly to production. Changes are instantly visible and distributed across teams. With versioned packages (right), the pattern library offers the components as a package (e.g., NPM) that teams can pull into their application. Teams control when to update to the latest version.

12.3.1 Runtime Integration

Twitter Bootstrap is the most famous example for a **runtime integration**. The concept is simple. Teams embed a link to a global CSS file that's maintained by the design system team. They can style their page by applying CSS classes to the markup. The same goes for the micro frontends embedded on that page. The CSS classes are globally available. Here's a code sample that shows how to embed and use global styles.

Listing 12.1 /team-decide/product/porsche.html

```
<link rel="stylesheet" href="/shared/pattern-library.css">      ❶
<button class="btn btn-call-to-action">Buy a tractor</button>  ❶
```

❶ integrating the pattern library styles

❷ using the styles via CSS classes

The runtime model is not exclusive to pure styling. If you use client-side rendering, it's also possible to provide components that encapsulate styling and internal markup. Here's an example for doing it via Web Components.

Listing 12.2 /team-decide/product/porsche.html

```
<link rel="stylesheet" href="/shared/pattern-library.js">   ❶

<tractor-store-price reduction="10%" value="66$">   ❷
```

❶ Integrating the pattern library script which contains Web Component definitions.

❷ Using the price component as markup. It renders the appropriate styled markup inside it's ShadowDOM.

Setting up a pattern library using a **runtime integration** is pretty straight forward. It's simple to develop and easy to use. Another benefit is that the design system team can roll-out changes instantly.

However, this model has some **considerable coupling and autonomy disadvantages**:

- **Testing in isolation:** Micro frontends should be self-contained. With runtime integration, a team's user interface doesn't work in isolation. For it to function, it's necessary to include the pattern library's styles and scripts. Since the design system teams can change these assets at any time a product team can't ensure that it's user-interface is looking and working correctly. They would have to run their automated test-suite on every pattern library change.
- **Single point of failure:** With runtime integration, the pattern library becomes a mission-critical part of the system. An error that slips through can bring down the complete project since all teams rely on it.
- **No tree shaking or deprecations:** Since the design system team can not know which components a specific page uses its common practice to include the styling code for all of them in one big CSS file. This file tends to only grow in size because there's no safe way to ensure that an old component is unused. When you go for the JavaScript components option, you can at least use on-demand loading strategies to avoid loading unneeded script code.
- **Breaking changes:** There's no structured way to handle breaking changes. If the team wants to refactor the button component in a significant way they need to create a new one (e.g. `.btn_v2`) and delete the old one when everyone has updated their markup.
- **Versioning & scoping:** It's hard to establish proper versioning in this model. There's also no easy way to prevent leaking styles between different micro frontends.

The lack of versioning is particularly critical. It means that all teams must use the latest version of the pattern library. **You can't restructure or upgrade the pattern library in a meaningful way.** It would require close coordination and simultaneous deployments from all teams. The fear of introducing friction incentivizes the design system team to shy away from making the necessary steps forward.

Let's look at a more flexible model to distribute a pattern library.

12.3.2 Versioned package

In the versioned model, the pattern library is not a runtime system. Instead, the design team distributes it as a package that contains all components. The lego metaphor works well here. You can think of it as a big box of bricks. The product teams can grab one of these boxes and take the required bricks out of it. The team uses these bricks Together with their special ones, they can build features for the customer.

Listing 12.3 /team-decide/static/product.jsx

```
import { Price, Button } from "@tractor-store/pattern-library";   ❶

function ProductPage() {
  return <div>
    <Price reduction="10%" value="66$" />   ❶
    <Button type="call-to-action">Buy a tractor</Button>   ❶
    ...
  </div>;
}
```

❶ Importing the required components from the pattern library package.

❷ Using the components to build the product page.

INDEPENDENT UPGRADES

The design system team can iterate on the pattern library. They produce new revisions of the lego box regularly. An updated revision might include new kinds of bricks or an updated surface finish. But teams don't have to upgrade instantly. They can upgrade at their own pace. An older revision might not look as sweet as the new one, but it still works fine.

DON'T SHIP UNUSED CODE

With this approach, each team generates its CSS file. A bundler like Webpack includes only the pattern library components that a team uses. So if the pattern library still includes an old component, but no team requires it, the browser won't have to download its code. This mechanism leads to quite small CSS files.

SELF-CONTAINED

You can instruct your bundler to prefix all CSS classes automatically. This way, it's possible to achieve proper scoping, and a page can contain Micro Frontends that use different versions of the pattern library.

Let's take an example. Imagine *Team Decide* owns the product page, which displays a price and a button. It also includes a Micro Frontend from *Team Inspire*, which also shows a button.

1. *Team Decide* uses pattern library version 4 in their applications.
2. The design system team releases a new iteration (v5) in which **the buttons have a new and rounder style**.
3.

3. *Team Inspire* immediately upgrades to this version and deploys its application.
4. *Team Decide* has other work to do. They'll update tomorrow.

Here is what the generated production code can look like:

Listing 12.4 /team-decide/dist/product.css

```
/* based on pattern library v4 */
.decide_price {...}
.decide_button { border-radius: 2px; }   ❶
.decide_[...] {}
```

❶ The old button styling from pattern library v4.

Listing 12.5 /team-inspire/dist/reco.css

```
/* based on pattern library v5 */
.inspire_button { border-radius: 10px; }   ❶
.inspire_[...] {}
```

❶ The new more rounded button styling from pattern library v5.

Listing 12.6 https://the-tractor.store/product/porsche

```
<div>
  <span class="decide_price">only 66$ (10% off)</span>
  <button class="decide_button">Buy a tractor</button>   ❶
  <aside>
    <button class="inspire_button">Show recommendations</button>   ❶
  </aside>
</div>
```

❶ *Team Decide's* button references it's own CSS class.

❷ *Team Inspire's* button also references it's own CSS class.

The two buttons on the page have different appearances. *Team Inspire's* button is already on the new rounder style, whereas *Team Decide's* button is still on the old style. Being able to use different versions, side-by-side is an essential step for independent deployments. **With this approach a Micro Frontend is fully self-contained and doesn't rely on styles from other teams.** The product teams are in control of upgrading their pattern library and testing the changes before they deploy them.

THE DRAWBACKS

This approach has a lot of advantages compared to the runtime integration. But it also has some drawbacks.

- **Redundancy:** When teams use the same component, the user has to download the associated code multiple times. You can see this in our example above. We have two versions of the button styling. This redundancy is typically not a big problem. Since the bundler only includes components that are in use, and no team uses all components at

once, the total CSS file size is usually much smaller compared to the global Bootstrap model.

- **Slower rollouts:** Changes in the pattern library take longer to be visible in production. The design system team can not push new updates. They can provide a new version and inform all the teams. The changes are only visible when all teams have updated and deployed their application. It might be necessary to encourage teams to deploy faster to rollout a critical design system bugfix quickly.
- **Eventual consistency:** Most graphic designers are not comfortable with the idea that a page can contain different versions of the same component. However, when teams update on a regular schedule, this is not a pressing issue. ProTip: In my last project, we've created a dashboard that shows which team uses which version of the pattern library. Merely showing this information in an aggregated view leads to faster upgrades by the teams. On another note: have a look at figure 12.7. It shows different generations of Amazon's buttons. These were all active at the same time in different areas of the site. The fact that Amazon does it should not be an excuse to discard consistency, but having temporal inconsistencies can be perfectly fine.

Figure 12.7 Screenshot of different Amazon button-styles from different parts of the site.

12.4 Pattern library artifacts: generic vs. specific

Now let's have a closer look at the technology of the pattern library. Its output has to be compatible with the technology stack of the teams. There's no gold standard for shipping reusable components. Different options exist, and they all have their benefits and drawbacks. Some don't support server-side rendering, some require a specific JavaScript framework and others support styling but not templating.

12.4.1 Choose your component format

User interface components consist of three parts:

1. **Styling** in the form of **CSS code**.
2. **Templating** to generate the components **internal HTML markup** based on the provided inputs. You can execute the templates on the server and/or client - depending on its format.
3. **Behavior** (optional) for components the user can interact with like tooltips or modals. They require **client-side JavaScript** to work.

Let's explore our options. Have a look at figure 12.8 and take some time to get an overview. The diagram shows different formats a pattern library can produce.

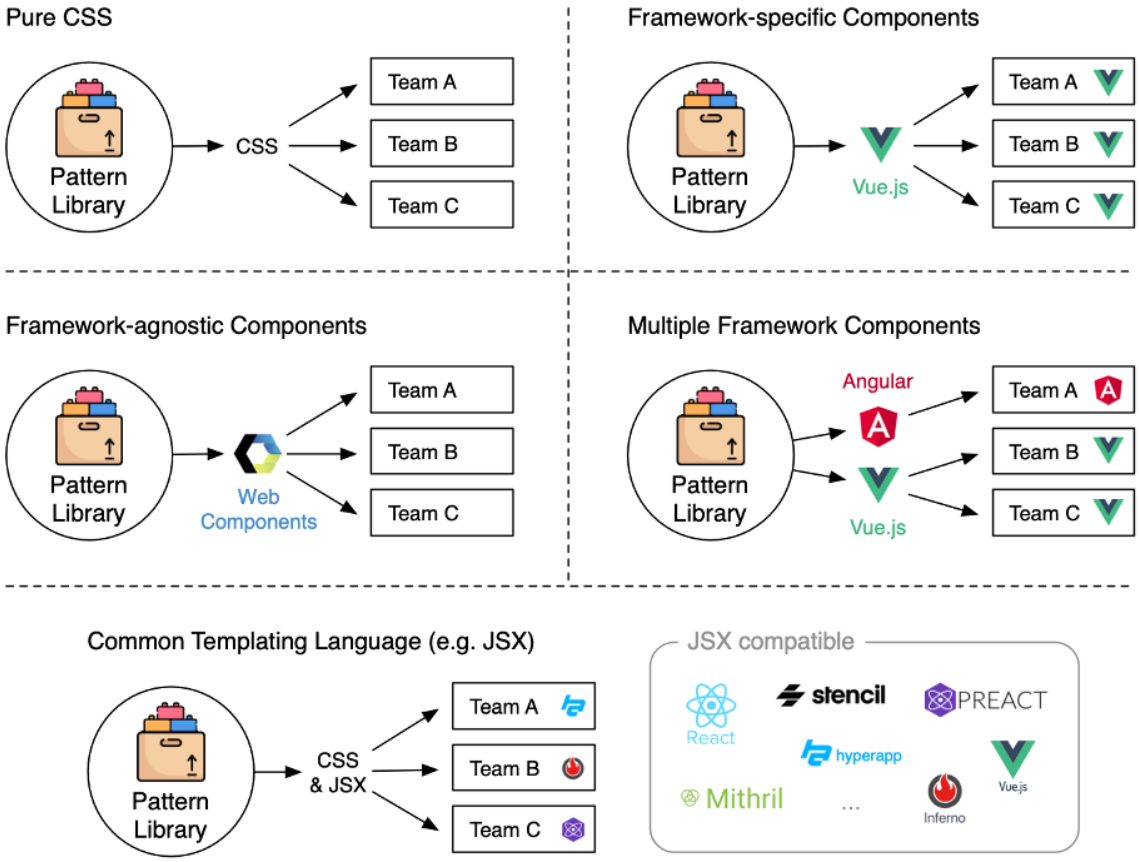


Figure 12.8 Shows different artifacts a pattern library can produce. Some output formats have technical implications for the team. When the pattern library only exports Vue.js components, all teams need to use Vue.js to be compatible.

We'll go through the diagram line by line.

PURE CSS

The pattern library provides it's component styling via CSS classes. Twitter Bootstrap is the role model in this category. Teams need to craft the components markup according to the pattern library's documentation.

- **Benefits**
  - easy to implement
  - works server- and client-side
  - compatible with all tech stacks that can generate HTML
- **Drawbacks**
  - styling only
  - teams need to know the internal markup
  - changing the component markup is hard

FRAMEWORK-SPECIFIC COMPONENTS

The pattern library uses the component format of one specific framework. An open-source example is Vuetify.[109], a component library designed for Vue.js. This model requires all teams to use the chosen JavaScript framework. The component formats of the popular frameworks have been pretty stable - even across major versions. This format stability means that teams have to use the same framework but aren't required to run the same version.

- **Benefits**
  - easy to implement
  - works server- and client-side
  - components integrate seamlessly with the team's code
  - components can use the full feature-set of the framework
- **Drawbacks**
  - all teams must use the same framework

FRAMEWORK-AGNOSTIC COMPONENTS

Web Components integrate well with all modern frameworks.[110] You can also use them on plain old HTML pages. Have a look at the Duet Design System.[111] as a useful reference. The developers built it using Stencil.[112]. In contrast to the "Pure CSS" approach, Web Components also encapsulate templating and behavior.

- **Benefits**
  - supported by all browsers
  - future-proof (web standard)
  - compatible with plain HTML and frameworks
- **Drawbacks**
  - only works client-side.[113]
  - JavaScript required (makes progressive enhancement hard)

MULTIPLE FRAMEWORK COMPONENTS

The model is related to "Framework-specific components". But instead of supporting one framework, the pattern library exports its components in different formats. Providing more than one format requires extra work because you will need to implement the framework-specific parts multiple times. **However, the concepts, component list, and the CSS styling stays the same.**

Google's Material Design is a large scale example of this. The design system itself defines styling, markup documentation, and scripts. Projects like Material UI (React) or Angular Material take the "generic" design system and transforms it into a framework-specific format.

- **Benefits**
  - works server- and client-side

  - components integrate seamlessly with the team's code
  - components can use the full feature-set of the framework
- **Drawbacks**
  - more work required

COMMON TEMPLATING LANGUAGE (E.G. JSX)

It doesn't have to be a specific component format. You can also ship HTML templates and styling (e.g., via CSS Modules). Many JavaScript libraries and frameworks support the JSX templating format. This way, it's possible to write the HTML template once and use it in a Hyperapp, Inferno, Preact or React application.

The lifecycle methods and event handling in these frameworks are not the same. This difference means that you can't include behavior. **Components have to be stateless.** But if your design system mainly includes essential UI components, this is not an issue.

Have a look at X-DASH.[114] from The Financial Times to see a real world example of this method. We are using the JSX approach in newer projects and are happy with its tradeoffs.

- **Benefits**
  - works server- and client-side
  - supports all frameworks compatible with the templating language
- **Drawbacks**
  - you can't include behavior
  - implementations might vary and speak different "dialects"

> **NOTE** You can use this model with any templating language. But be aware that implementations aren't always 100% compatible with each other. We e.g., had significant issues with using handlebar templates across languages like Scala, Python, and JavaScript. Be confident that your model works and its limitations are well understood. Create technical spikes to verify it before you roll it out company-wide.

12.4.2 There will be change

As said before, there's no clear winner. The right choice for your project or company depends on your needs.

However, if you've made a choice, you should communicate the **contract between the pattern library and the teams**. Does your integration rely on a framework component format, is it based on a specific DOM structure that's documented somewhere or do teams need to support a dedicated templating language.

This decision impacts the team's autonomy long-term. Switching to another model later is

costly and cumbersome.

BE OPEN FOR CHANGE

A good option for being open to future trends and technologies is to **have the "multiple framework components" model in mind from the beginning.** Even if you decide to start with Vue.js components. If your concepts are stable and you architect your CSS in a reusable way, it will be easier to add new output formats like Web Components, Angular, or Snowcone.js.[115] later on.

KEEP IT SIMPLE

Another tip is to **keep the central components as dumb as possible.** Try to keep the behavioral aspects to a minimum.

Let's take a navigational tree component as an example. It's a vertical list of links you can expand to see it's nested links. The pattern library could provide a fully-fledged component which includes functions like expand/collapse, text-search and has hooks for lazy loading subtrees. But getting all of these aspects right and fulfilling every team's needs is challenging.

You could also go the other route and let the pattern library only provide the building blocks and states this tree component can have: expanded/collapsed items, active state, and position of a search box. With this approach, the teams have more work because they need to build the mechanics (toggle, search, …) themselves, but they also have much more flexibility. They could decide to pick an open-source tree library that fits their needs and feed it with the styled building blocks from the pattern library.

Focusing on the visual aspects makes your pattern library more flexible and reduces feedback-loops with the teams.

Finding the right balance between centralized and distributed is not always easy. In the next section, we'll dig a little deeper into this question.

12.5 What goes into the central pattern library?

Having all user interface elements visible and documented in the central pattern library is valuable. The central documentation makes it easy to get an overview. But sharing a component comes with costs.

12.5.1 The costs of sharing components

Changing a component in a teams application code is much easier than changing a component in the central pattern library because **central components …**

- **… live in another project.** You have to publish a new version to see the change in the team's code.

- **… might be used by other teams.** You need to think about the possible consequences of these teams.
- **… must conform to higher quality standards.** You want to ensure that even people from outside your team understand a component's capability and the reasoning behind it.
- **… might require code-review.** Depending on your design system development process, you might instantiate a dual control principle to guarantee a high standard.

These aspects make changing a component in the central pattern library much harder than directly changing it in your own code. Putting all components into the pattern library would slow down the development. That's why you need to consciously decide what goes into the central pattern library and what should better be local to a team.

12.5.2 Central or local?

In a lot of cases, the decision if a component should be **central for all** or **local for one team** is easy to make.

- The definition of the *sale color* should, of course, be **global**. The same goes for an *icons set* or the *styling of an input field*.
- Advanced patterns like a *payment options box* or the *concrete layout of the product page* should be **controlled by the respective teams**.

But there's a middle-ground where these decisions are not that clear. Is the *filter navigation* or a *product tile* a central component? Let's look at some vectors that help you make your decisions.

COMPONENT COMPLEXITY

The atomic design methodology.[116] is quite popular. It uses the chemistry metaphor of atoms, molecules, and organisms to sort components by their complexity. This metaphor also highlights the fact that larger components are a composition of smaller ones. Figure 12.9 shows the atomic design categories from lowest complexity (design tokens) to highest complexity (features & pages).

Figure 12.9 The atomic design methodology organizes the design system by complexity. The central pattern library should include the basic building blocks (tokens, atoms, molecules). More sophisticated components (organisms, features, entire pages) should be under a team's control. The middle-ground around molecules and organisms is fuzzy.

This scale maps well to our central vs. local question. A good rule of thumb is to **share simple components** and **put complex ones under team control**.

But this model is fuzzy in the centrum of the scale. Developers like to argue about the fact if a

component is a molecule or an organism. But these discussions are almost always theoretical and fruitless. Comparing code complexity is not the only important factor.

REUSE VALUE

The reusability of a component is a reliable indicator. Components that different teams need might go into the central pattern library, even if they are not simple. Patterns like accordions or carousels are good examples here.

However, you should be careful with this rule. **The focus of larger components might change over time.** Here is an example:

Team Inspire uses the *product tile* component for their recommendations. *Team Decide* has built a wishlist feature. They use the same *product tile* on their wishlist page. The central component worked great at the start, but over time both teams come up with conflicting requirements. *Team Inspire* want's the component to be more compact to fit more tiles in a recommendation slot. *Team Decide* needs to add more functions and product details to it. Moving the component out of the central pattern library and let each team work on their own version of it might be an option. Another alternative can be to reduce the *product tile* component to its essence. The new component could provide dedicated slots where teams can add functionality if they need to.

These conflicts are natural because nobody can foresee the future. It's essential to reevaluate your decisions regularly and be open to revising them.

DOMAIN SPECIFIC

Domain-specific components are good candidates to be team owned. To identify this, you can ask the question: **"Which team has the interest to change this component and why?"**

When there's a team that frequently updates a component to improve its business, it's a reliable indicator that this component should be local to that team.

A *filter navigation* is a good example here. At first sight, a list of filters looks pretty simple. However, when you have a product team with the mission to "make finding products easier", this team will want to change this component frequently. They want to test different variants, collect feedback, and improve the component. Making this team jump to hoops by centralizing the component will slow them down and block innovation.

TRUST IN TEAMS

These three properties (complexity, reusability, and being domain-specific) can give you a good idea of where a component should live. Don't be afraid to give up central control and let teams own and evolve specific components.

But it's not just about control. If a component is not part of the global pattern library, it's hard for designers to keep an overview. Next up, we'll look at the concept of "local pattern libraries" to mitigate this.

12.5.3 Central and local pattern libraries

It's not a written rule that you must have a single design system. There's the concept of **tiered design systems** .[117] that perfectly fits into our micro frontend architecture. The idea is to have a central pattern library that defines the basics and other pattern libraries that build on it and add their use-case specific components. In our case, each team can have its own **local pattern library**. Figure 12.10 illustrates this.

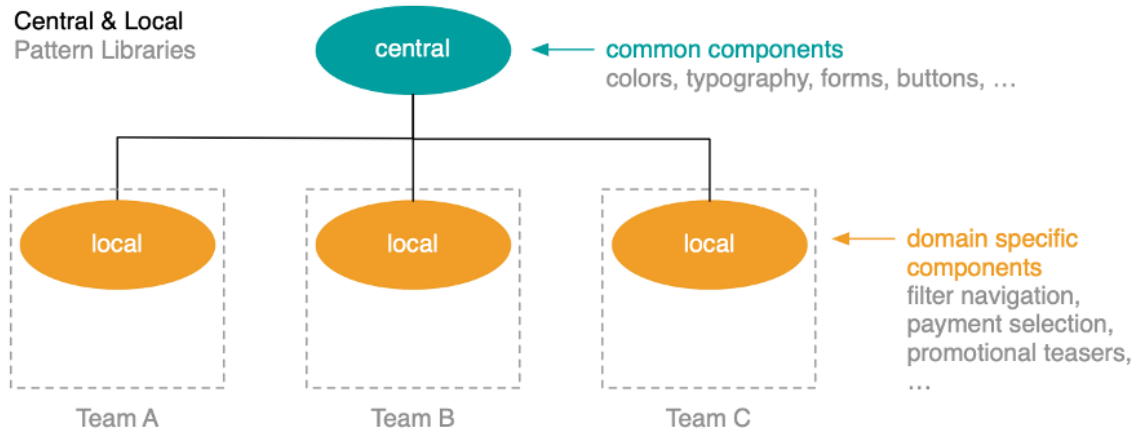


Figure 12.10 A two tiered pattern library approach. Each micro frontend team has it's own local pattern library where it develops it's domain specific components.

A team can only use the components from its own local pattern library. But **all teams can browse the component catalog of the other teams.** This visibility is a good starting point for spotting cross-team inconsistencies. It's also a solid basis to start a "central vs. local" discussion.

The central and local pattern libraries could also use the same tool to develop and generate the design system documentation site. Popular tools for this are Storybook.[118], Pattern Lab.[119] or UIengine.[120]. Using the same tool has the advantage that moving a component from the central to the local pattern library (or the other way around) is as easy as moving a component folder.

Now you've learned a lot of aspects that can help you when implementing a design system in a micro frontends project. In the next chapter, we'll broaden our view and look at other organizational implications this architecture introduces to your company.

12.6 Summary

- Every micro frontend team develops its own user interface. A central design system that all teams can use helps to deliver a consistent user experience across all micro frontends.
- A shared design system introduces coupling between the teams. All teams must work with the system and be compatible with its technology.
- All visible features the product teams produce rely on the design system. Changing the technical architecture of the design system afterward is cumbersome and costly.
- The design system exists to help the product teams ship features and be more consistent. It does not create value on its own.
- Developing a design system is a continuous process. Ensure that it’s maintained properly and doesn’t get out of date. Don’t let it become a zombie design system.
- The design system can become a bottleneck when teams request more changes than the central team can handle. Product teams might need to wait and delay features.
- You can develop the design system in a federated way. Developers and designers from each team contribute to the system. A small core team ensures quality and has an eye on consistency. This model can scale and works well with the micro frontends principals.
- The central and federated development models are not mutually exclusive. You can move between these models.
- There are two ways to integrate the pattern library into your project: Runtime integration and via versioned packages.
- With runtime integration, the design system team deploys directly to production, and all product teams must use the latest version. This model has considerable drawbacks for team autonomy since teams cannot ensure that their software is always working correctly.
- Distribution via versioned packages enables the product teams to upgrade the pattern library at their own pace. A team’s micro frontends can be self-contained because it doesn’t rely on external dependencies at runtime.
- There are different formats (CSS only, framework-specific, …) a pattern library can publish their components in. The format has technical implications for the team. Some don’t work server-side others require all teams to use the same JavaScript framework.
- Frontend tools and libraries change over time. Try to build your design system in a way that makes adapting to changes easy.
- Sharing components across teams is not free of cost because they require a higher quality standard. The central pattern library should include basic building blocks. More complex and domain-specific components should be local to the team that needs it.
- A product team can have it’s own "local pattern library". It presents all the components this team owns. It’s a good way for designers and developers to get an overview, spot inconsistencies, and start discussions.

13 Teams & Boundaries

This chapter covers

- Exploring how to structure your teams to maximize the benefits of the micro frontends architecture.
- Employing techniques and rituals to foster a healthy amount of knowledge sharing between the teams.
- Identifying common crosscutting concerns and highlighting different strategies to address them.
- Illustrating the challenges a diverse technology landscape can introduce.
- Examining techniques that can help new teams to get up and running quickly.

Throughout this book, we focused on the technical aspects of Micro Frontends. You learned techniques to integrate independent user interfaces that form a greater whole. We talked about strategies to mitigate **architecture-inherent issues** like performance and providing a seamless user interface.

But why are we doing all this?

Yes, there are **some technical benefits** that come with this architecture. Smaller software projects are simpler to build, test, understand, and rebuild than a monolith. Being able to use different tech-stacks in different areas of the product can also be a valuable asset.

However, **the most significant benefits our composable frontend architecture unlocks are the organizational ones.** It makes it possible to parallelize development. Properties like having real team ownership, and local decision making can lead to faster innovations.

You might have noticed that I've used the word **"team"** over a thousand times .[121] in this book - this is not by accident or lack of creativity. It would have been perfectly fine to use words like "micro frontend application" or "software system" in most cases to understand the described

techniques. **But it's not about the software. It's about the people designing and building it.**

I've talked to and read from a lot of smart people that successfully introduced a Micro Frontends architecture in their company. In all cases the motivation to go down this road were **the organizational and not the technical benefits**: Setting up individual and robust teams and empowering them to build and improve a specific area of the product.

That's what we'll talk about in this chapter. What organizational and cultural changes should you make to leverage the full potential of this model? How to address cross-cutting concerns without reinventing the wheel in each team? Last we'll look at the topic of technology diversity. How much freedom should a team have to pick their stack? Let's start with a little bit of theory.

13.1 Aligning systems and teams

If you have ever explored the concept of microservices before, you probably came across **Conway's Law** .[122]. In the 1960s, computer programmer Melvin Conway formulated the hypothesis that the communication structures of an organization are reflected in the technical systems they create.

This means that if you let *one* team build a product, it will likely produce a more monolithic system. If you give the same task to *four* teams, they'll probably come up with a more modular solution.

The importance of keeping the structure the organization and its technical systems in sync has been well researched.[123] and understood in modern software development. Here's a quote from the book "Organizational Patterns of Agile Software Development", published in 2004.

> *If the parts of an organization (e.g., teams, departments, or subdivisions) do not closely reflect the essential parts of the product […], then the project will be in trouble … Therefore: Make sure the organization is compatible with the product architecture.*
>
> *– James O. Coplien and Neil Harrison*

For a micro frontend architecture, this means that the team boundaries should align with the boundaries of the vertical applications that form the product. Figure 13.1 illustrates this.

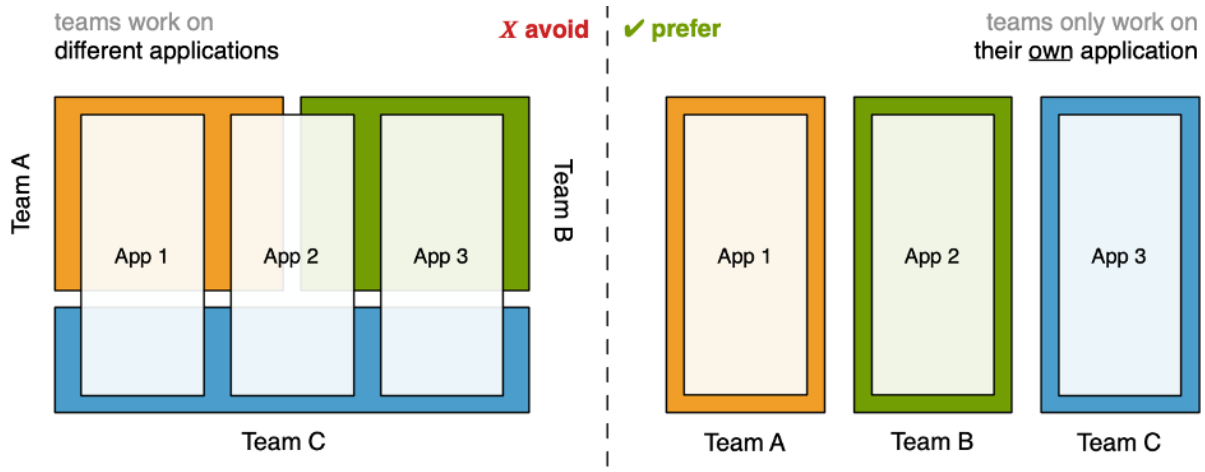


Figure 13.1 Team structure and software structure should align. Having one team working on multiple applications, or even worse, multiple teams working on the same application, can create issues. An architecture where one team owns one application will likely be more effective.

13.1.1 Team boundaries

Ok, understood! We should keep team- and software-structure aligned. But how do we find out what structure is beneficial for the product we want to create? How do we identify sound boundaries? Here are three methods that can help you.

DOMAIN-DRIVEN DESIGN (DDD)

Domain-Driven Design is a popular approach for structuring software. It acknowledges the fact that its hard to create a consistent model for a project of a specific size. It provides patterns to handle this complexity by creating smaller sub-models that have an explicit relationship with each other.

DDD provides a set of concepts and tools to identify and isolate areas in your project. It introduces the idea of analyzing the language of different experts and departments in a company: **Ubiquitous Language**. This way, it's possible to identify **Bounded Contexts**, one of DDD's core concepts. We won't go into more detail on DDD in this book. Still, there's a lot of great.[124] content.[125] you can check out if you want to learn about it. **A Bounded Context is an excellent candidate to become it's own Micro Frontend application and team.**

USER-CENTERED DESIGN

Let's set aside our IT glasses and put on our product management scarf for a minute. A critical task in product design is **to pinpoint user needs**. In day-to-day business, it's easy to get lost in optimizing our current products.

If we want a sustainable relationship with our customers, it's essential to understand their real motivations. What do they want when they come to us? How can we make their life easier?

Techniques like **Design Thinking** .[126] or **Jobs To Be Done** .[127] provide solid mental models to reason about a users motivation. The famous quote.[128] of Theodore Levitt brings the difference between our current offers, and the users needs to the point:

People don't want to buy a quarter-inch drill. They want a quarter-inch hole!
– Theodore Levitt Harvard Business School marketing professor

Modeling your teams and systems around your customers needs can be a valid choice. It gives the teams a clear goal that's focused on what matters most: your user.

EXAMINING EXISTING PAGE STRUCTURES

A more hands-on method for identifying boundaries is to have a look at the page structure of your current project. This method works when you already have a functioning business model. Print out all page types on a piece of paper. Gather a group of experienced colleagues and **group the pages by using your intuitions.**

In most cases, a page represents a specific use-case or task your user needs to do. Looking at pages is not a perfect solution. Some pages might have more than one purpose. You can use scissors to fix this. This method is an excellent entry to start more in-depth discussions.

If you've established groups, you can try to verify your hypotheses by looking at analytics data you've gathered in the past. Do the usage patterns align with your page groups?

Now that we have an idea on how to structure the teams. Let's talk about who should be one the team.

13.1.2 Team depth

The integration techniques described in this book are all frontend-related. But Micro Frontends is not an architecture limited to the frontend - on the contrary. It unfolds its full potential when it covers the complete stack. Figure 13.2 shows different depths of integration and their potential benefits.

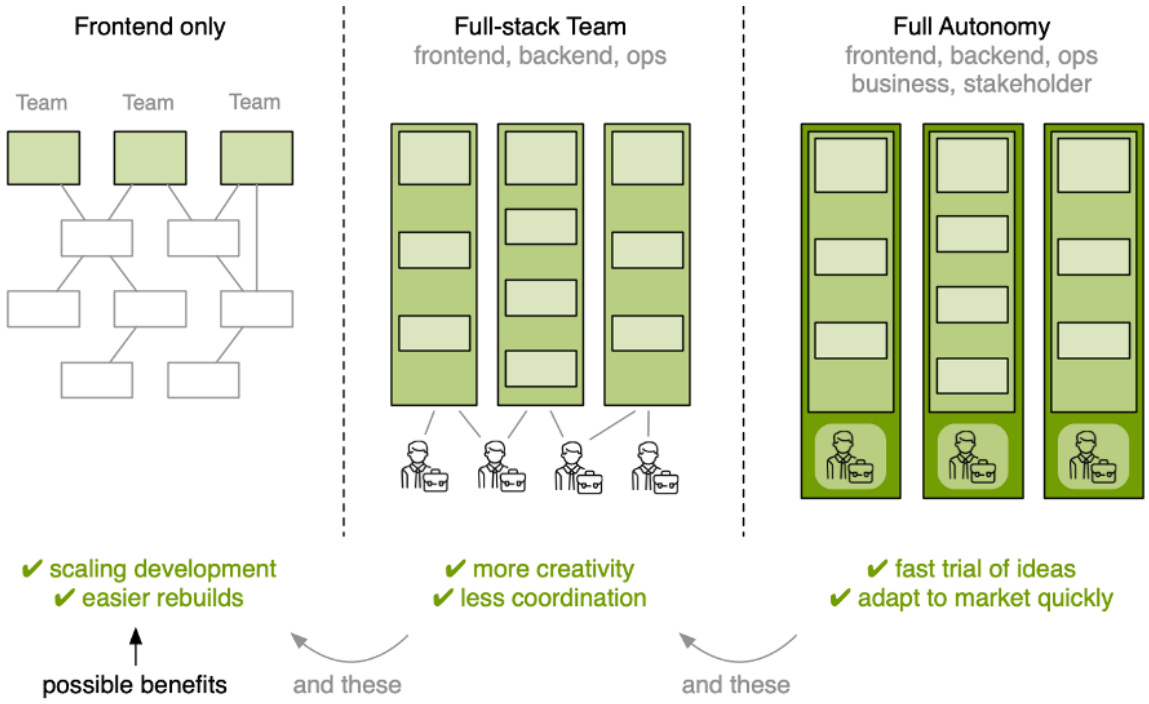


Figure 13.2 A Micro Frontend team can be limited to the frontend (left). However, when you add more disciplines like backend and operations to the team (middle) it becomes easier to ship features end-to-end. An ideal team also includes business experts and stakeholders (right). Then it's able to make all it's decisions locally to create customer value.

Let's look closer at the three approaches described in the diagram.

FRONTEND ONLY

In this model, you have a backend, be it monolithic or microservices-style. The vertical Micro Frontend teams sit on top of this backend. In case of a microservices architecture, each frontend might have its own Backend for Frontends (BFF).[129] to communicate with the services.

This approach has some real benefits compared to e.g., a monolithic single page application:

- **Scaling development:** Here the *Two-Pizza Team Rule* we talked about in chapter 1 comes into play. Assuming you've come up with good boundaries, it's more efficient to have three teams with five developers, each working on a dedicated piece of software than having a 15 person team working on a large code-base. It's easier for a developer to understand the part of the system he's responsible for. When you've established a Micro Frontends architecture with three teams, all patterns are in place to create a fourth team that develops an entirely new part of the application. The other three teams can go on with their regular business. Integrating the new Micro Frontend with the existing application is a small amount of work.
- **Easier rebuilds:** Modernizing an existing Micro Frontend is a more straightforward task. You don't have to think about the complete application. You can upgrade and rebuild team-by-team. No all-hands-on-deck big-bang migrations.

FULL-STACK TEAM

In this model, we make our Micro Frontends teams go beyond the frontend-backend line. It includes developers from the frontend, backend to operations, or data science. We form a cross-functional team that combines competences from database to user interface. Here are the benefits of the full-stack approach:

- **More creativity:** Cross-functional teams combine people with different backgrounds which provide different perspectives on a problem. This diversity can lead to better and more creative solutions.[130]
- **Less coordination** The most significant benefit of the end-to-end team model is that it **reduces waiting time**. All features that can be accomplished inside team boundaries don't require other teams to become active. This autonomy eliminates the need for organizing meetings with other teams, formalizing requirements, and global ticket prioritization.

[[TODO: move this paragraph to the new communication chapter once it exists. reference it from here.]] (Moving to this model introduces some unique challenges: How do teams share data in the backend when there are no shared services? We usually solve this by introducing a shared message bus or feed exchange mechanism. An example: *Team Decide* runs the product page but also owns the master product database. They provide a data-feed that other teams can tap into to import the data they need. *Team Checkout* might need the product name, image, and price. They replicate these product properties and store them in their database. This way *Team Checkout* is still able to function even if *Team Decide* has technical issues)

FULL AUTONOMY

We can take this one step further by also including domain experts and business people into the team. In most companies, these people typically work in departments like legal, marketing, risk, customer support, logistics, controlling, and so on. These departments specify requirements that the "IT people" must implement. Breaking up this traditional boundary and moving these experts closer to the development teams is not an easy task. It's a slow transition that must be encouraged by the top of the organization.

Having e.g., marketing, legal, or customer support expertise directly available in a development team can unlock further benefits:

- **Fast trial of ideas:** Moving from a formal requirements and prioritization process to a basis where you can exchange ideas at eye level can improve your product. Here a small-scale example: In our last project, the team developing the checkout system invited people from the call center to its sprint ritual. A developer presented the new voucher system. A call center employee interrupted and described the fact that older customers are often stressed by the minimum order value - especially if their shopping cart total is only slightly below it. In this meeting they came up with the idea to make the minimum value constrained more tolerant: communicating a minimum of 20€ but enforcing only 18€. This trivial software change had a measurable effect on customer satisfaction: fewer

support calls and a more generous company image. Ideas and changes like this can make a big difference.

- **Adapt to marked quickly:** The digital services landscape and your user's expectations can change fast. New forms of payment methods, integrations with social platforms, and communications channels emerge. When all people that are necessary for strategic decisions work in the same team, you can move quicker.

A general rule of thumb is that extending your vertical teams deeper into the organization will likely increase the speed and quality of these teams' work. If you want to get deeper into this topic, the **Agile Fluency Model** is a good starting point. The model describes four fluency zones an agile team can reach.[131] A team in the first zone *(Focusing)* leverages basic agile practices like scrum to improve its work. Running a Micro Frontends architecture with **Full-Stack Teams** aligns with the second stage: *Delivering*. The **Full Autonomy** approach maps to the third agile fluency stage: *Optimizing*.

The adoption of a Micro Frontend structure in the frontend is a decision that the frontend team can make on its own. But extending it to the entire development team or even an organization is a significant management task. Let's briefly talk about the cultural changes that come with it.

13.1.3 Cultural change

The vertical architecture plays well with having a user-focused culture. Every team delivers to the customer directly.

This mentality is often already part of the DNA of start-ups. That's why the proposed vertical team structure might feel like a natural way of growing a start-up.

Large traditional organizations have a harder time moving to a more vertical architecture. They often think in short term projects rather than long term products. Also, the concept of **ownership** plays a vital role in running this architecture successfully.

You want teams to identify with the product they create and make it more valuable for the user. Teams should be empowered to make decisions, conduct experiments, and learn from failures. Hierarchies, department structures might get in the way. I think to have an **open culture based on agile values** .[132] is a prerequisite for getting the most out of a Micro Frontends style architecture.

13.2 Sharing knowledge

The cross-functional team structure optimizes communication along with a business domain (vertical). This model is good because it helps to focus on the user, but it also introduces challenges: **How do you avoid reinventing the wheel in every team?**

Ok, granted - the majority of the work in these teams is not the same. Developers building a fast

loading product list are faced with other challenges than the developers, which are architecting a registration form that has to work in all countries around the world.

But there are aspects that all teams share. What are the right strategies to automatically test the software? What's a good way to handle state inside my application? "I've encountered a strange issue. I wonder if anyone else has this problem?"

Let me give you a real-world example of insufficient cross-team communication. In my last project, we've been developing an e-commerce shop with five teams staffed from three software companies. Half a year into the project, a co-worker from one of the other companies gave a talk on debugging Node.js performance at a conference in Hamburg. I attended his talk because I was curious. On stage, he referred to a mysterious problem he's been tracking down for the last weeks. He was convinced that it had to be something in his team's application code. The behavior he described was instantly familiar to me because I've encountered something very similar in our team's application. After the talk, we spoke and shared our findings. It became apparent that it had to be an issue with the hosting infrastructure our applications ran on.

But the fact that we had to meet at a public conference to figure this out is a little disturbing. We could have saved each other a considerable amount of time and headaches if we had spoken to each other earlier.

13.2.1 Community of Practice

In the early 90s, the concept of a Community of Practice (CoP).[133] was formulated. It describes ways to spread knowledge across teams. A CoP is a group of people that share a craft or profession. In our example, all people doing frontend work could be part of the same CoP. These groups create their communication channel to exchange information on a specific technology, ask for help or share learnings.

Spotify is famous for its agile and team-focused organization structure.[134]. They also organize in end-to-end teams. They've institutionalized Communities of Practice called **guilds**. There like-minded people from different teams can exchange knowledge.

Figure 13.3 shows some example guilds that form horizontal channels across our otherwise vertical organization structure.

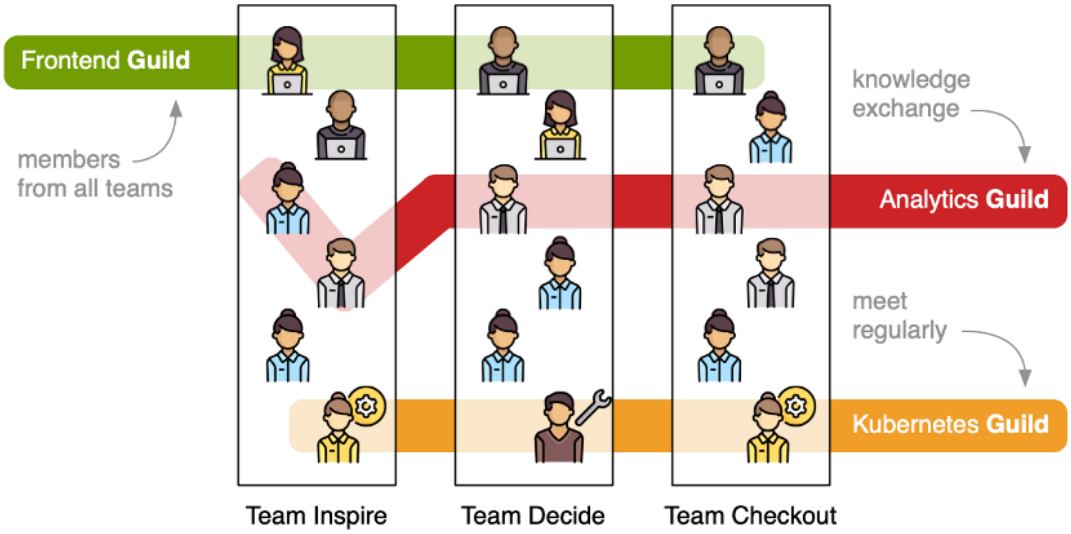


Figure 13.3 A guild creates a room for people from different teams that share an interest or profession. Their primary goal is to exchange knowledge.

Guilds typically have a dedicated communications channel like a Slack group. All guild members meet regularly. In our projects, short guild meetings happen (bi)weekly via video call to discuss recent issues. From time to time, the guild also organizes longer in-person workshops for diving deeper into a specific topic.

Typical guilds in our projects are frontend, backend, UX/design, analytics, infrastructure, data-science, coaching, security, and macro-architecture.

13.2.2 Learning & enabling

Having a cross-functional team that's able to handle the complete stack flawlessly sounds terrific on paper. In practice, it is rarely possible to assemble a team that is capable of developing customer features while also mastering all non-functional requirements such as performance, security, or testing.

Being faced with these expectations as a team can be quite intimidating.

In some areas, technical developments like *cloud hosting* play in our favor. A team can offload tasks like "managing real hardware" to a cloud provider. These services make a *You build it; you run it!* approach more realistic.

But that's not possible for all topics. **Learning and improving is an integral part of forming a cross-functional team.** Having each team identify and formalize its strengths and weaknesses can speed-up the learning process.

CoP's can play a central role in education. A developer from *Team A* that has experience in analytics can teach other guild members and help them to level up their skills. For some topics, it might also be a good option to hire an external mentor to support a guild.

13.2.3 Present your work

Another way to exchange information is by presenting your teams work. This way, teams have an idea of what the others are working on. The presenting can be in the form of a real on-stage presentation where all teams show what they've accomplished and learned in the last month. But it can also be in the form of a small internal blog-post.

Since all teams work in different areas, these presentations usually don't have an immediate value for the others. However, they enable "Wait, hasn't Team Inspire also build a feature with Apache Spark last month? Let's talk to them first."-moments.

Rituals like these can also strengthen group cohesion and avoid an "Us vs. Them" mentality.

13.3 Cross cutting concerns

Let's dig a little deeper into these cross-cutting concerns. Yes, forms like guilds can help to spread knowledge, but let's look at some concrete examples on how to address common topics.

13.3.1 Central infrastructure

Some cross-cutting concerns require dedicated infrastructure. Examples of this are your version control system, continuous delivery pipeline, analytics, dashboards, monitoring, error tracking, your hosting setup, and shared services like a load balancer. Each team could make these choices on its own. However, these are all general topics most professional software projects need. Having each team figure out a solution might not be the best use of their time.

Establishing a shared set of infrastructure all teams can use is a good idea. There are different ways to organize this.

SOFTWARE AS A SERVICE (SAAS)

For commodity products, it's often the easiest way to use an off-the-shelf product like Amazon's AWS for infrastructure and GitLab for version control and pipelines. Using standardized services does not introduce inter-team coupling. All teams communicate directly with the provider of the service. Make sure that each team has its sub-account or an explicit namespace to avoid conflicts. If a team has a strong need to switch to another provider, there should be no technical hurdles.

Sometimes going for a SaaS solution is not an option. The reasons for this might be the price or lack of functionality. If you need to run an infrastructure component yourself that all teams can use, you have two options: having it owned by one of the product teams or introducing a dedicated infrastructure team.

OWNED BY ONE PRODUCT TEAM

In this model, the product teams take responsibility for the self-hosted central services. Team A might be responsible for setting up, running, and maintaining the shared load balancer. Team B might run a private NPM registry that all teams can use. **Having clear responsibility is essential** to ensure that these services receive the attention and care they need. By spreading the services among the teams, **the burden** that each team has to carry **is reduced**.

This model works well if the number of self-hosted services is not too high and the services themself are easy to maintain.

> **WARNING** Share generic infrastructure components only. Avoid sharing business logic this way, because it creates coupling and undermines team autonomy.

CENTRAL INFRASTRUCTURE TEAM

If the above methods don't work for you, there's always the option to create a dedicated infrastructure team that takes responsibility for all shared infrastructure aspects. But this pure infrastructure team does not fit well into our otherwise vertical and customer-centric architecture. It has the potential to become a bottleneck that hinders feature development.

13.3.2 Specialized component team

Sometimes there is neither a managed service nor an open-source solution that fulfills our need. This is where the concept of *component teams* comes in.[135] Spotify calls these teams *infrastructure squads*.[136]

Say different product teams need to talk to a legacy ERP system that does not support modern APIs. Having a dedicated team that develops a service or an abstraction library might save the product teams a lot of time. Another example of a component team is the *central design system team* we talked about in the previous chapter.

Component teams don't provide direct value. Their goal is to enable the product teams to move faster. Since the introduction of a component team creates friction and inter-team dependencies, it's used should be well considered. These two questions can help with deciding if you should use a component team or not:

- Is the service it provides **required by many** teams?
- Does building the service **require specialized technical expertise** that is not present in the product teams?

If you can answer one or better both questions with a clear yes, it might be worth thinking about a component team.

13.3.3 Global agreements & conventions

Not all cross-cutting concerns manifest themself in a shared service or library. **Often an agreement to which all teams adhere is enough.** For topics like *currency formatting*, *internationalization*, *search engine optimization*, or *language detection*, it's often perfectly fine to have central documentation.

All teams agree upon the described canonical way and implement it in their applications. Yes, this may result in redundant code, but for topics that are not critical or don't change frequently, it's often the most effective way.

13.4 Technology diversity

The Micro Frontends architecture enables each team to pick and change their technology stack. We've already discussed the benefits this introduces. But just because **you "can" does not mean you "must"** use a diverse technology stack.

Having to hand-pick a technology stack can also be a burden. Let's talk about some techniques to make these decisions easier.

13.4.1 Toolbox & Defaults

The **toolbox** idea explicitly limits the technology choices by providing a list of vetted options. It's a project- or company-wide piece of documentation that may live in the wiki. The content of the toolbox might read like this: Java or Scala are the backend programming languages of choice. PostgreSQL is our go-to for relational databases. You should stick to Webpack for your frontend builds.

The toolbox should be guidance and not a set of laws. If teams have reasons to deviate from the norm, they should have the possibility to do so. For most teams, the toolbox is a **source of sensible default options**. Need an end-to-end testing framework? Let's open the toolbox and see what has worked for other teams.

Since technology is evolving, the toolbox has to be a living document that's updated regularly: adding new technologies that have proven valuable or deprecating existing ones that went out of date.

13.4.2 Frontend Blueprint

When a new team starts fresh, it has to do a lot of setup work. Creating it's a basic application, build process, and other tedious tasks that are necessary before they can get productive.

We've been using the concept of a **shared frontend blueprint** to ease this pain. The blueprint is an example project that includes all significant aspects a Micro Frontend application needs. We

can divide these aspects into two groups: technical and project-specific.

TECHNICAL ASPECTS

- Directory structure
- Testing (unit, end-to-end)
- Linting & formatting rules
- Code formatting rules
- API communication
- Performance best practices (optimizing assets)
- Build tool configuration

These general topics are necessary to have, but they are not that interesting. Most major JavaScript frameworks have a scaffolding tool that generates an example project for you. But a stock frontend setup will not be sufficient for a team to get going.

PROJECT SPECIFIC ASPECTS

Your frontend needs to integrate with the other teams and must adhere to the high-level architecture guidelines. A new frontend must also cover **project-specific** aspects. That's why our frontend blueprint also includes:

- Composition examples
  - including another Micro Frontend
  - providing an includable Micro Frontend
- Communication examples
- Team prefixing for CSS and URLs
- Template for documenting your Micro Frontends
- Integration with the central pattern library
- Setup for the local pattern library
- Wiring for shared services like error tracking or analytics
- CI/CD pipeline

New teams will copy the blueprint over to their project and adjust it to their need. Building on the existing work reduces setup time noticeably. But for us, the blueprint has another, even more, important role.

*It's the **reference implementation** for the macro architecture decisions.*

It includes running examples of integration patterns and communication strategies. This example-code helps all developers to understand high-level topics by seeing them in action in a real application.

MAKE IT OPTIONAL

Teams are not forced to use the blueprint as is, or even at all. They are free to adapt it to their needs. It's explicitly not a shared production code-base. The frontend applications are based on a copy. Making a change to the blueprint will not affect the existing frontends. Developers communicate improvements to the blueprint via the frontend guild. If a specific improvement is valuable for a team, it can look at the change and apply it manually.

13.4.3 Don't fear the copy

As you've seen with the blueprint, it's often a good idea to copy-and-paste from other applications. For everyday tasks, it's an easy solution that ensures team autonomy down the road. Copying the 15 line currency formatting algorithm from your neighbor team is a good example. This algorithm is not set in stone, but it's easy to understand and unlikely to change monthly.

We, as developers, have a trained tendency to spot and eliminate duplications. But this elimination is not free - especially when you try to centralize across teams. Maintaining a shared library that six teams depend on is not a trivial job and will come with a lot of discussions, waiting, and headaches.

For bigger use-cases, the pain a duplication introduces might be higher. Our poster-child example is the central pattern library. You don't want to copy-and-paste it on every change. There might be other pieces of code you want to share like a library that makes talking to a legacy system easier. Sharing these as a versioned library might be fine, but it should always be a conscious decision and come with the right amount of dedication. In discussions, I found this quote helpful to create the right mindset:

> *Only do it, if you are willing to run it as a successful (internal) open-source project.*

Don't underestimate the organizational overhead a shared library introduces.

13.4.4 The value of similarity

In our projects, teams often picked similar programming languages and frameworks to build their applications.

Using the same technologies as your neighbors has advantages. It makes sharing best practices more accessible. Developers that want to switch teams can get up and running quickly. The ability to browse other teams' Git repositories and see how they've solved a particular task is also valuable.

Artifacts like the toolbox and the blueprint can help in forming a shared technical direction. Finding the right balance between similarity and freedom is never easy. Technical arguments often drive discussions around this. But taking the business-, or even better, the user-perspective

can help to maintain focus. Will using Haskell instead of Scala improve the product in a noticeable way?

13.5 Summary

- Running a successful Micro Frontends architecture is not a technical decision. The team structure should align with the software systems to be most effective.
- There are different ways to identify team and system boundaries. Domain-Driven Design provides tools like analyzing expert language to identify groups of functionality. Bounded Contexts are good candidates for a Micro Frontend team.
- Organizing your teams around user needs can be a good model. Techniques like Design Thinking and Jobs-to-be-done can help to isolate these use-cases.
- The existing page structure of your site might already be a good indicator of team boundaries. The question "What purpose does this page serve?" can lead you to groups of functionalities.
- Using Micro Frontends only on the frontend has technical benefits like parallelizing work and easier rebuilds. Rolling it out to the complete development stack or extending it further also to include stakeholders and business experts can unlock further benefits like faster development and a better customer focus. The vertical team structure aligns well with the upper stages of the Agile Fluency Model.
- The vertical architecture optimizes for delivering features inside a team’s scope. Introducing horizontal groups like Communities of Practice or guilds helps to spread knowledge.
- It’s often more efficient or necessary to run a shared infrastructure. Leveraging SaaS solutions like AWS can be a good option that doesn’t introduce inter-team coupling. Sometimes the SaaS model doesn’t fit, and you need to self-host. You can distribute the responsibility for the infrastructure components across the product teams. Introducing a dedicated infrastructure team is an alternative but does not fit well into a vertical architecture.
- Since Micro Frontends are decoupled, each team can choose it’s technology stack freely. Methods like a shared toolbox or a central blueprint can help to form a common technology direction that ensures room for innovation and experimentation when needed.

14 Migration, Local Development & Testing

This chapter covers

- Illustrating strategies to migrate a monolithic application to a micro frontends architecture.
- Setting up a local development environment and examining techniques like micro frontend mocks to ensure independence.
- Implementing automated testing in a micro frontends architecture.

Micro Frontends is not the first architecture for most companies. It's something you migrate to because the old architecture has trouble keeping up with new demands like increasing team size or high demand for features.

If you are a fresh startup that needs to grow quickly, it might be a good idea to start with Micro Frontends from scratch. However, most larger companies use Micro Frontends to replace a functioning but slow or unmaintainable monolith. If you find yourself in the latter camp, this chapter will help you by highlighting some good migration strategy.

In the second part of this chapter, we'll have a closer look at the developers day-to-day life in a Micro Frontends project. A team only works on its slice of the complete application. Developing a feature locally without seeing it integrated with the rest of the software will feel strange at first. You'll learn techniques and tricks that make developing and testing easier.

14.1 Migration

Migrating a non-trivial project from one architecture to another is a scary and often costly task. You can take different roads which all have their benefits and drawbacks. On the following pages, we'll discuss **three ways to move to a Micro Frontends architecture**. This chapter will not be the "definitive guide for software migrations". Lots of publications describe the essential parts you should think about when migrating a large project. **Instead, we will focus on the Micro Frontends specific aspects.**

Having a somewhat realistic idea of the complexity and effort a migration takes is vital to set expectations and calculate costs. But when your team doesn't have experience with the target architecture, it's hard to come up with reasonable estimates. Playing around with the technology in a sandbox project helps to reduce the fuzziness. The examples in this book can be a good starting point for these experiments.

*Micro Frontend's **user-interface integration techniques** are a valuable asset for incremental migrations.*

The Micro Frontends paradigm and its frontend integration techniques lend itself well to build and integrate a proof of concept and even verify it in your production application. Before we go into the migration strategies, let's have a closer look at this proof of concept idea.

14.1.1 Proof of concept & building a lighthouse

You can adopt Micro Frontends by building a single feature as it's own end-to-end system and integrate it into your existing application. Figure 14.1 shows a two-part diagram that illustrates this.

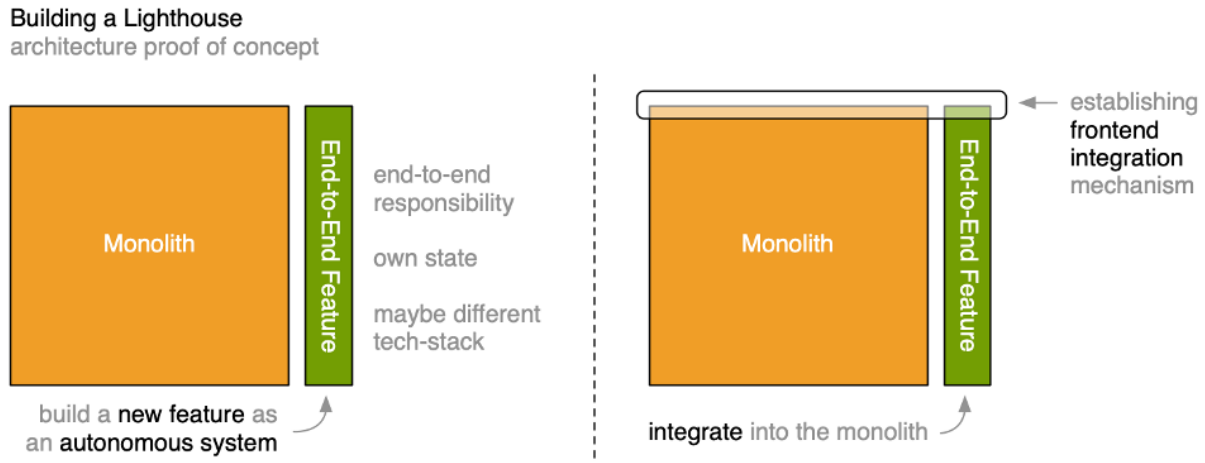


Figure 14.1 To try out the Micro Frontends architecture, you build a new feature as a dedicated application that has its own state but also includes the associated user interface. It's decoupled from the existing monolith. That's why the team responsible for this new feature can build it based on a new technology stack if it wants to (left). A frontend integration mechanism is established to integrate this new application with the monolith (right). Integration can be as simple as using hyperlinks between both applications, but it can also include a composition technique.

A REAL WORLD EXAMPLE

Let's look at a concrete example. The company *Miniature Farming Industries*, one of *Tractor Model Inc's* rivals, has a monolithic e-commerce shop that doesn't perform well. They consider moving to a Micro Frontends architecture. To test out the waters and don't lose a lot of time, they decide to develop one of their already planned features as a Micro Frontends application.

Miniature Farming Industries forms a new team dedicated to building this new feature: the wishlist. The core user-facing aspect is *the wishlist overview page*, where the user can see and manage his favorite products. Also, a user should be able to add products to the wishlist by clicking a small *heart icon button* on a product tile. **The new team builds and owns both: The wishlist page and the add-to-wishlist button.**

The wishlist page should have the same header and footer as the other pages of the shop. Since the new team doesn't want to duplicate the header and footer, they decide to include it from the existing application as a fragment. To make this possible, the team working on the monolith has to provide header and footer as standalone micro frontends. In reverse, the wishlist team provides the **add-to-wishlist button** as a fragment for the monolith to include in every product tile.

The teams must establish a shared integration technique. They go with a server-side composition using SSI. Therefore they install an Nginx server as a frontend proxy that sits in front of both applications. This server has two tasks: routing and composition. All requests starting with `/wishlist` get routed to the new application, all others hit the monolith. The web-server also handles composition. It replaces the header/footer SSI-directives of the wishlist page with the actual markup from the monolith.

That's everything required to make the integration work. Ok, not quite. The teams also needed to work on some other relevant topics. The frontend developers refactored the CSS code of both systems to ensure that the old and the new application don't over-style each other. The backend developers had to build an import for necessary product data like image, name, and price. The data import is necessary to ensure that the new system has it's own data store and doesn't depend on the monolith at runtime.

THE ROLE MODEL

If everything goes as planned, this first vertical system can act as a lighthouse for your migration project. We've established a frontend integration mechanism that new systems can use. The "proof of concept" can become the role model other teams can follow to build new features.

14.1.2 Strategy #1: Slice-by-slice

The first migration strategy **slice-by-slice** is a natural progression from the proof-of-concept idea before. Figure 14.2 shows a monolith thats migrated to a three-team micro frontend architecture.

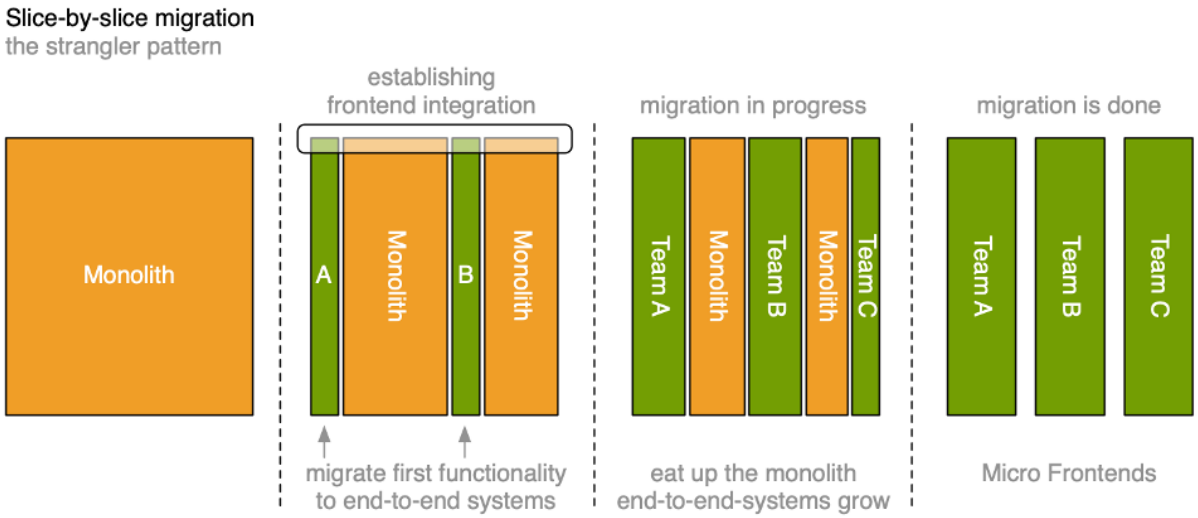


Figure 14.2 Migrating a monolithic application (left) to a three-team micro frontends architecture (right). In this diagram, we create three new applications (Team A-C), which take over functionality from the monolith step by step until the monolith has vanished (middle). A frontend integration layer handles the routing and composition of the different applications.

HOW IT WORKS

First of all, we need a shared plan for how the final team boundaries should look like. Which team owns which feature? After these decisions, the teams can go ahead, set up their new applications, and start migrating functionality from the monolith into their micro frontends application. They migrate the system feature by feature. The first feature to extract might be *product reviews*. One team moves the feature over to their application: from user-interface to the database.

The teams established a frontend integration mechanism that handles routing and composition. After migrating a feature, the team replaces the associated user interface in the monolith with the new micro frontend's UI. Then they tackle the next feature.

The teams repeat this process until the monolith has vanished. This migration follows the **Strangler Fig Pattern**.[137] This pattern describes how a new application gradually replaces the existing one. During the migration phase, both applications are still in business.

BENEFITS & CHALLENGES

The main benefit of this incremental migration approach is that it introduces **little risk**. The newly created software goes into production regularly. There's no big-bang moment when switching from the old to the new system. **The system is always in a working state.** Even if you decide to cancel the migration project in the middle of the process, you have a functioning application. All software that's written goes to production quickly.

Compared to a greenfield project this approach **requires more thought, understanding of the existing system and coordination.** Extracting features from the monolith does not mean that you have to remove them. However, you will at least have to **adapt the monolith's user interface** along the process to play nice with the new micro frontends. Depending on the software quality, the CSS code and lack of proper scoping are often the most significant tasks that you face. Web Components and Shadow DOM can be of help. Revisit chapter 5.2 Style isolation using Shadow DOM for more details on this.

14.1.3 Strategy #2: Frontend first

The frontend first approach follows a similar pattern but avoids mixing the old and the new frontend code. Not having to care about the "old frontend code" can make your life easier, especially when you are planning to do a frontend facelift on the way. Figure 14.3 shows the migration process.

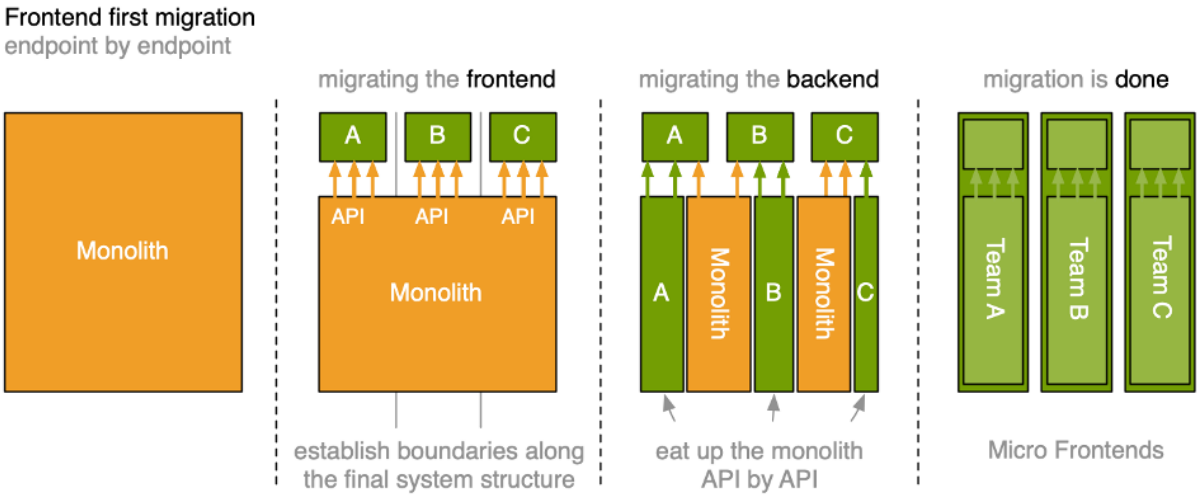


Figure 14.3 We start with a monolith (left). The migration has two phases. In the first phase, we replace the frontend of the monolith with three new frontend applications, which are each owned by one team. The frontends communicate via APIs with the old monolith. In the second phase, we migrate the backend with the slice-by-slice approach. Migrating each API endpoint into the new backend application of the responsible team. After the backend migrations, we've reached our goal. A vertically sliced application (right).

HOW IT WORKS

Here the migration is a two-phase process. We start with the frontend. It's rebuild to fit into the desired vertical structure. You need to plan team boundaries and responsibilities ahead of time. Each team build's it's own part of the frontend. Teams integrate their user interface via the known routing and composition techniques. The new frontends receive their data from the old monolith via APIs that are purpose build.

In the second phase, we start splitting up the backend. The APIs we've implemented in the previous step define the boundaries and guide the way for the backend. Each team creates a backend application that's able to replace the monolith APIs it's frontend relies on. In this phase, we can again apply the slide-by-slice pattern. The teams replace API after API until the monolith is out of job.

Now we've reached our desired state. The monolith has vanished, and each team owns a system that reaches from frontend to backend.

BENEFITS & CHALLENGES

As said before, the most significant benefit with the frontend-first approach is, that **we don't have a phase where the old and new frontend code mixes.** No issues with leaking styles or unexpected side-effects because we create a clean new frontend landscape in one step. If your frontend does not contain too much business logic and complexity this approach also has the benefit of **delivering fast results.**

We had good experiences with this approach. However, it has two disadvantages that you should consider.

The required frontend and backend work will not be distributed evenly. The first phase is more frontend heavy, and in the second phase, the backend work dominates. You can counteract this by overlapping the phases or, even better, encourage your teams to work cross-functionally.

The second aspect you should keep in mind is that **visible progress in this model is non-linear**. From an outsider's or the management's perspective, the first phase, rebuilding the frontend, will introduce a lot of improvements. Even if you don't build new features, the use of modern technology or the introduction of a new design will make the site feel faster and fresher. The second phase will, at best, not introduce any visible change to the user at all. This lack of visual progress might not be a problem, but you should manage expectations accordingly.

14.1.4 Strategy #3: Greenfield & big bang

The greenfield and big bang approach is the easiest from a conceptual standpoint. The old system stays as is, and you build a new system in parallel. Figure 14.4 illustrates this.

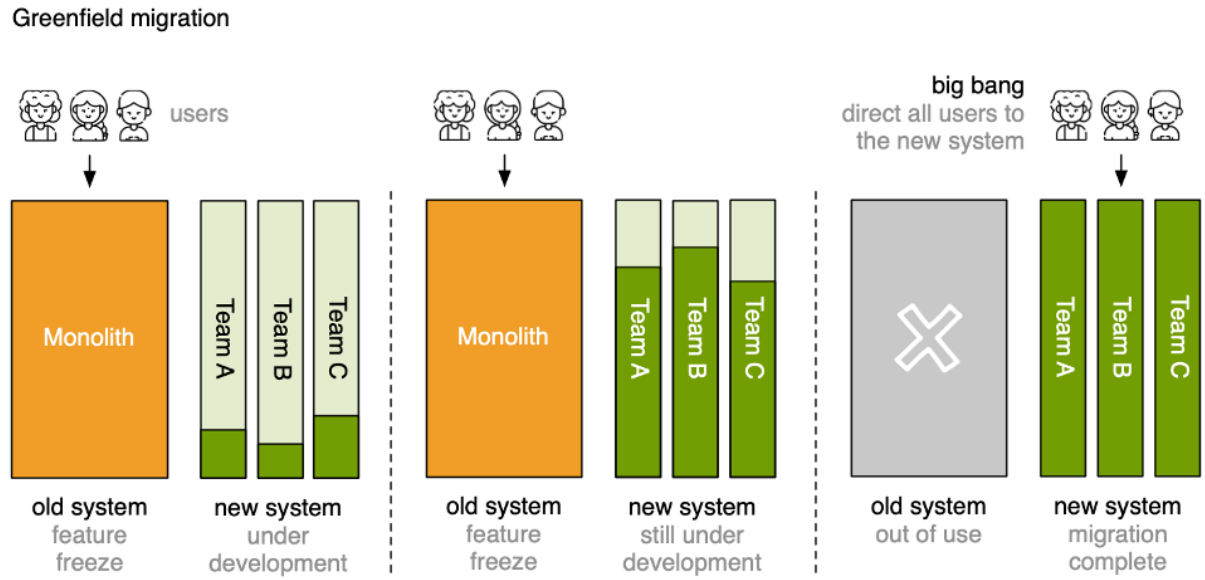


Figure 14.4 We set up our new team structure and system architecture beside the existing monolith (left). The new and old systems don't share anything. During the development phase, all incoming traffic still arrives at the monolith (middle). When the teams finished building the new system, we direct the incoming traffic to the new system, and the monolith is out of use (right).

HOW IT WORKS

We make a plan for how the new system should look like and set it up in a new environment that's separate from the existing monolith. The development of the old system is often halted to avoid extending the migration phase. The teams start building their slice of the system. When all teams finished implementing the features that are necessary for production, we route the incoming traffic to the new system and retire the old one. The old and the new systems don't mix at any time. Users are either using the old or the new system.

BENEFITS & CHALLENGES

The main benefit of a greenfield approach is the fact that we can start fresh and don't have to deal with legacy code. The clean slate makes it easy to adopt techniques like continuous delivery or introduce a new design system that can be **free of hacks and compromises**. Because teams can focus on building the new architecture and don't have to wrestle with the legacy system **development will be faster.**

We've used this migration strategy in different projects. It's **attractive when it's hard to adapt the existing monolith** during the migration process. This inflexibility may be the case when the monolith relies on proprietary technology that you can't change or when it's on a very long deployment cycle that would slow down your development.

But as the **big bang** in the title implies, there's a **considerable amount of risk** associated with this approach. The **teams develop** the new system over a long period **without receiving real user feedback**. Verifying that the system works in production is extremely valuable. **Consider moving users to your new system as early as possible.** Having actual users reduces risk and increases confidence in the system you're building. Concepts like *releasing it as a beta version* or *testing it in smaller markets* can be of help.

Now you've seen a couple of strategies to get from monolith to Micro Frontends. There's no golden way, and it always depends on the system you have and the goals you want to reach with the new architecture. But **leveraging frontend integration techniques to gradually replace the old monolith with new micro frontend applications is a powerful tool** that you should consider.

14.2 Local Development

Now we'll leave the architecture level and zoom into the day to day life of a developer working in a Micro Frontends project. Running and developing a classical monolith is pretty straight forward. You can check out one source code repository, which contains everything required to start the complete application on your local machine. Everything should work, and you can try the application in your browser from start to finish.

With a distributed architecture like Micro Frontends, this gets more complicated.

14.2.1 Don't run other teams code

Each team has its source code repository, and teams may have different tech stacks. Yes, it might be possible for a developer to not only have his team's repository checked out but also pull an up-to-date copy of the other team's source code regularly. While this might work, it can become cumbersome very quickly. **Having to know about the development environment of other teams introduces friction.**

What do you do if the other team has a bug that prevents their application from starting? Has Team B upgraded to the latest version of Node.js or are they still on the old one? You shouldn't have to care about these kinds of problems to do your job. You should be able to focus on the code your team owns. So, let's talk about how we can develop without running other people's code.

SIDEBAR **But what about monorepos?**

When you read about Micro Frontends on the web, the term monorepo.[138] sometimes appears as a solution for local development. Monorepo describes a concept where the code of independent applications or libraries live in one version control repository. A monorepo makes it easy to download and update multiple projects at once and manage shared dependencies.

If you see Micro Frontends purely as a set of integration techniques for one team to modularize its frontend, the monorepo approach is reasonable. However, if you want to take advantage of the organizational benefits of multiple independent teams that can work side-by-side without close coordination, the monorepo is an anti-pattern. The team's applications should be independent and shouldn't share code or a deployment pipeline. Separate repositories guard against unwanted inter-team dependencies.

14.2.2 Mocking fragments

TIP You can find the sample code for this chapter in the `21_local_development` folder.

Ok, so if I can't run the code from other teams, how shall I be able to develop? On page level, the answer is simple. **Replace other teams fragments with mock versions of them.** Let's have a look at *Team Decide's* product page.

Go into the sample code and run the following command.

```
npm run 21_local_development
```

Open up localhost:3001/product/porsche to see the product page in local development mode. Figure 14.5 shows the result.

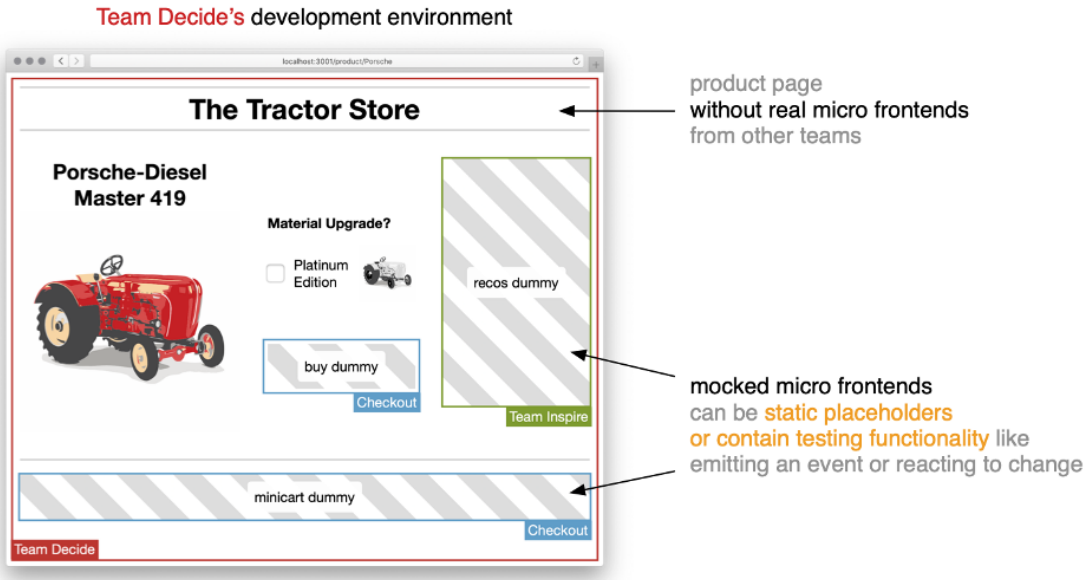


Figure 14.5 Team Decide's product page in local development mode. The fragments from the other teams are replaced by simple mock micro frontends.

We see the product page, but the fragments from the other teams (recommendations, buy button, and mini-cart) got replaced with mock versions of these micro frontends. But the page itself is working as expected. You can toggle the platinum option and the product image updates accordingly.

The product page you are seeing does not include any code from other teams. In development mode *Team Decide* omits the script and style tags to the other teams fragment definitions. Not loading these files would lead to empty blocks where the fragments should be.

To improve this, *Team Decide* created its simple mock implementations for the three fragments. You can find the associated code in `team-decide/static/mock-fragments.(css|js)`. Since we are using Custom Elements for integration, it's pretty easy to mock the fragments. Here is the code for one mock:

Listing 14.1 team-decide/static/mock-fragments.js

```
...
class CheckoutMinicart extends HTMLElement {
  connectedCallback() {
    this.innerHTML = `<div>minicart dummy</div>`;
  }
}
window.customElements.define("checkout-minicart", CheckoutMinicart);
...
```

The above code is a pretty simple mock that just shows a text. But if you expect a fragment to throw an event, you can get more sophisticated and e.g., add a button that triggers the event.

> **NOTE** The example uses client-side rendering, but the concepts are also applicable for a server-generated application. Instead of replacing Custom Element definitions, you'd route the fragments HTTP request to an endpoint that returns mock markup.

Using mock fragments instead of pulling in real components will make development more straightforward and more robust. You only have to fire up your application, and if something breaks, you can be sure that it's the fault of your code.

Each team that provides a fragment should document its interface. The interface lists the parameters it understands and the events it can emit. **The fragment documentation can be the basis for creating your local mock.**

> **WARNING** If you find yourself in a situation that requires building a lot of sophisticated mocks to develop and test your software you might have issues with your team boundaries. Make sure the responsibility for one use-case is not spread across different teams.

14.2.3 Fragments in isolation

Let's see how developing a fragment looks like. Keep the sample application running and open your browser at localhost:3003/sandbox to find *Team Checkout's* sandbox page, which shows both of their fragments. *Team Inspire* has a similar page running on port 3002. Figure 14.6 shows both sandbox pages.

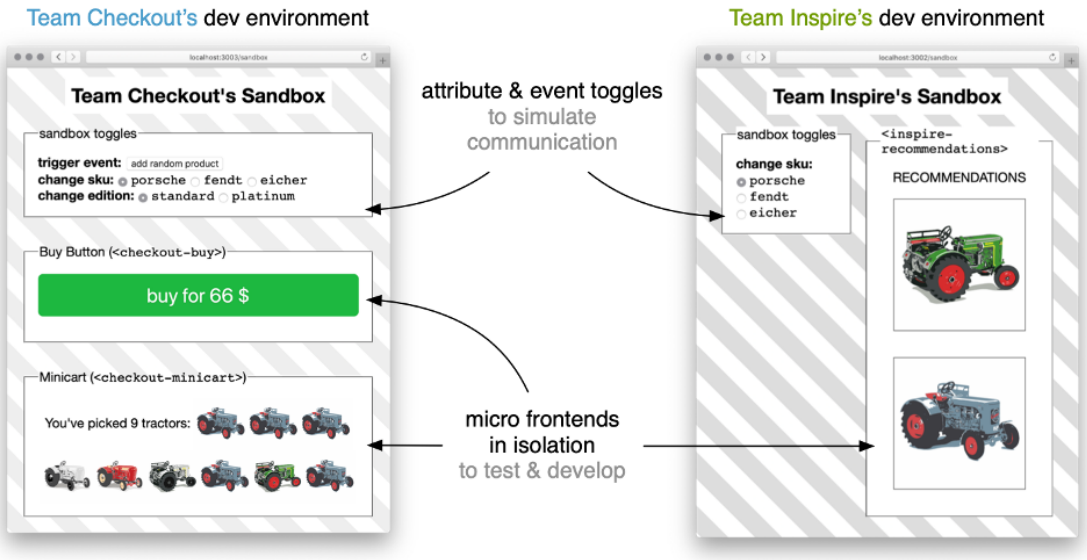


Figure 14.6 Each team has its own sandbox page where it can develop and test fragments in isolation. The sandbox page also contains some toggles to simulate communication.

DEVELOPMENT PAGE

The sandbox page acts as the development environment for fragments. It's an empty page (in this case with a stripy background) that contains a team's fragments. The page itself also includes basic global styles like root font definitions and some CSS resets since you don't want every fragment to redefine these styles itself. Tools like Podium create such a page out of the box.[139], but building this page from scratch is also not complicated.

You'd also use your favorite live-reload or hot-code-replacement solution here to make development more enjoyable.

SIMULATING INTERACTIONS

Now we have an environment to develop our fragments in, but how do you test cross Micro Frontend communication? You might have noticed the "sandbox toggles" section at the top of our pages. It contains a set of actions our fragments can react to.

You can e.g., use the "change sku" control to switch from one tractor to another. Changing the option will toggle the associated `sku` attribute of the buy button fragment, which should then update it's price accordingly. In the example the toggle mechanics are a few lines of plain JavaScript in the sandbox file.

The mini cart also updates itself when someone clicks the buy button. You can test this fragment-to-fragment communication on the sandbox page. Click the button, and the product will appear in the mini-cart. The mini-cart listens to the `checkout:item_added` event on the

window, just as it would on a fully integrated page. The sandbox page also has a dedicated *add random product* button that triggers such an event.

Investing some effort in a local development environment will make your life easier and can save a lot of time down the road.

14.2.4 Pulling other teams Micro Frontends from staging or production

But in some cases, mocking is not sufficient. If you are trying to reproduce a mysterious bug, you might want to test with the real code.

If you're doing **client-side rendering**, this can be easy. You don't have to check out and build the other team's code from scratch. **Point the associated script and style tags to your staging or production** environment and fetch the code for the other teams' fragments from there. Now you can debug how your local code plays with the released code from the others.

Single-spa even goes a step further. They've built a tool called `single-spa-inspector` that lets you do it the other way around.[140] You can open up a production page in the browser, and the inspector makes it possible to replace the released version of your code with your local development code. They use `import-maps` to do the trick.

Pulling in fragments from a remote server is also possible with **server-side rendering**. There you'd advise your HTML assembly mechanism to fetch the markup for some routes directly from production. If you're using Nginx and SSI, you can achieve this by changing the upstream configuration for the other teams to the production server but keep your upstream pointing to localhost.

14.3 Testing

Automated testing has become the centerpiece of modern software development. Having good test coverage reduces the need for manual testing and enables you to adopt techniques like continuous delivery.

How does testing look in a Micro Frontends project? It's not so different from testing in a monolithic project. Every team tests its application on different levels. They'll have a bunch of fast running unit- and service-tests and a couple of browser-based end-to-end tests.

You'll probably know about the testing pyramid.[141]. It describes that tests with a low level of integration (e.g., unit-tests) are cheap to write and run fast. Tests with a high level of integration run slow and are expensive to maintain. Figure 14.7 shows a variant of the classical testing pyramid.

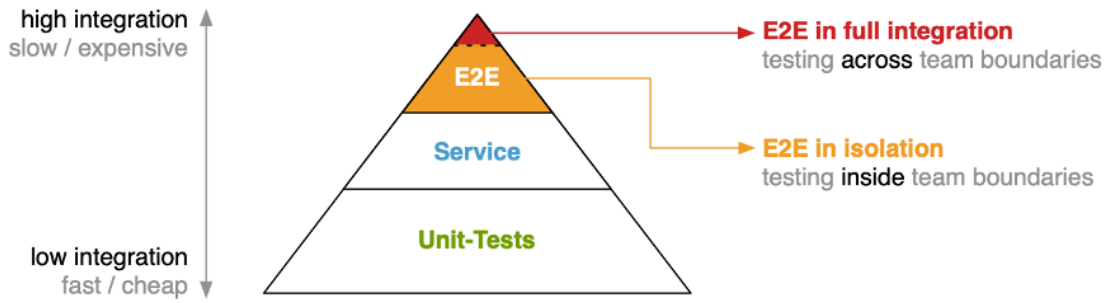


Figure 14.7 The testing pyramid shows that low-level tests are fast and cheap (bottom). Tests with a high amount of integration, like browser-based end-to-end tests, are slow and expensive to maintain. In a Micro Frontends project, we can split the end-to-end test category (top) into types of tests: those that only run on one teams UI and those that run across team boundaries.

In a Micro Frontends project, we can split the topmost category (UI or end-to-end tests) into two parts:

- **Isolation (most tests)**: A team should perform the largest part of their user interface tests in an isolated environment without the code from other teams. These tests would run against a version of the software with mocked fragments. Own fragments are tested in an isolated environment (sandbox), as shown in the previous section.
- **Full integration (very few tests):** Even if every team tests its fragments and pages accurately, there is a possibility of errors at the user interface boundaries. You should test critical transition points in full integration.

Full integration tests are hard to write because they require knowledge about the markup structure from at least two teams. **We didn't have good experiences with introducing an overarching integration test-suite** that runs against the complete software. All our attempts ended in brittle solutions with lots of false positives. Also, the question "Who owns the overarching integration tests?" is a hard one to answer if you don't want to introduce a horizontal testing team.

Instead, we go for **a distributed approach**. Every team can decide to test across the borders of their direct neighbors. *Team Checkout* could test if it's buy button micro frontend works when it's integrated on *Team Decide's* product page. *Team Decide* might check if *Team Inspire's* recommendation fragment is not empty.

14.4 Summary

- You can use the Micro Frontend's user interface integration techniques to test out this architecture with your existing project. These techniques also enable gradual migrations where new micro frontends replace the old monolith user interface slice-by-slice.
- Replacing a current system slice-by-slice introduces low risk, because you have a working application at all times. However, mixing the new frontends with the monolith's frontend can be challenging due to leaking styles. Using Shadow DOM for the new micro frontends can be of help.
- If mixing user interfaces with the monolith doesn't work, the frontend first or a greenfield approach are good alternatives, but they come with a higher risk.
- It's a good idea to disable code from other teams in your local development and testing environment. Eliminating foreign code reduces complexity and makes the environment more stable. Creating simple mock micro frontends help to get a more realistic impression of the layout.
- Mock micro frontends can be static placeholders, but they can also include simple functionality like emitting events.
- You can develop fragments on a dedicated sandbox page. It shows the fragment in isolation. This sandbox page can also contain some custom user interface to test communication (trigger events) or simulate changes in the environment (e.g., change sku).
- Nearly all your tests should run against your own team's code. Test in isolation where possible. In some cases, it might be necessary to test across team boundaries. A central testing team can be responsible for this. Another solution is that teams test the integrations point to the neighbor teams themself.

Notes

1. Yes, I'm aware that there probably is a JavaScript framework for all dictionary words registered on npmjs.org, including *Thunder* and *Wonder*. But since both projects have over six years of inactivity and single-digit weekly downloads, let's stick to them. :)

2. Removing jQuery from GitHub.com frontend github.blog/2018-09-06-removing-jquery-from-github-frontend/

3. Large Scale CSS Refactoring at trivago medium.com/@pistenprinz/large-scale-css-refactoring-at-trivago-4602113c4a26

4. Raiders of the Fast Start: Frontend Perf Archeology www.youtube.com/watch?v=qts9gPYoANU

5. Why Jeff Bezos' Two-Pizza Team Rule Still Holds True in 2018 blog.idonethis.com/two-pizza-team/

6. On Monoliths and Microservices dev.otto.de/2015/09/30/on-monoliths-and-microservices/

7. Experiences Using Micro Frontends at IKEA www.infoq.com/news/2018/08/experiences-micro-frontends

8. Project Mosaic | Microservices for the Frontend www.mosaic9.org/

9. Another One Bites the Dust (written in German) tech.thalia.de/another-one-bites-the-dust-wie-ein-monolith-kontrolliert-gesprengt-wird-teil-i/

10. Spotify engineering culture labs.spotify.com/2014/03/27/spotify-engineering-culture-part-1/

11. SAP Luigi luigi-project.io

12. DAZN - Micro Frontend Architecture www.youtube.com/watch?v=BuRB3djraeM

13. Sample Code on GitHub github.com/naltatis/micro-frontends-in-action-code

14. Manning: Micro Frontends in Action www.manning.com/books/micro-frontends-in-action

15. URI Template tools.ietf.org/html/rfc6570

16. Home Documents for HTTP APIs mnot.github.io/I-D/json-home/

17. Swagger OpenAPI Specification swagger.io/specification/

18. iframe-resizer github.com/davidjbradshaw/iframe-resizer

19. Quore: How is JavaScript used within the Spotify desktop application? answer/Mattias-Petter-Johansson

20. Saving the Day with Scoped CSS css-tricks.com/saving-the-day-with-scoped-css/

21. Can I Use Shadow DOM caniuse.com/#feat=shadowdomv1

22. BEM getbem.com/naming/

23. Immediately invoked function expression en.wikipedia.org/wiki/Immediately_invoked_function_expression

24. github.com/gustafnk/h-include

25. Details on the Googlebot's JavaScript support developers.google.com/search/docs/guides/rendering

26. Progressive enhancement en.wikipedia.org/wiki/Progressive_enhancement

27. Resilient web design resilientwebdesign.com/

28. Getting To Know The MutationObserver API www.smashingmagazine.com/2019/04/mutationobserver-api-guide/

29. The easiest option is to install it via Homebrew (brew.sh) by running `brew install nginx`.

30. Most distributions offer nginx as an official package. On Debian/Ubuntu, this works `sudo apt-get install nginx`.

31. Zalando Skipper opensource.zalando.com/skipper/

32. Open-source Edge Router docs.traefik.io

33. Varnish HTTP Cache varnish-cache.org

34. WebPageTest is an excellent open-source tool for doing this www.webpagetest.org/

35. You can change this behavior by setting the `max_fails` and `fail_timeout` options in an upstream configuration. See the Nginx documentation for more details on this. nginx.org/en/docs/http/ngx_http_upstream_module.html#upstream

36. ESI Language Specification 1.0 www.w3.org/TR/esi-lang

37. Akamai EdgeSuite 5.0 ESI Extensions to the ESI 1.0 Specification www.akamai.com/us/en/multimedia/documents/technical-publication/akamai-esi-extensions-technical-publication.pdf

38. www.zalando.com

39. www.mosaic9.org/

40. github.com/zalando/tailor

41. github.com/zalando/tailor#options

42. www.finn.no/

43. podium-lib.io/

44. expressjs.com/

45. podium-lib.io/docs/podium/conceptual_overview/

46. Web Component Polyfill www.webcomponents.org/polyfills

47. github-elements www.webcomponents.org/author/github

48. stenciljs.com/

49. Angular Elements Overview angular.io/guide/elements

50. github.com/vuejs/vue-web-component-wrapper

51. List of elements that support Shadow DOM dom.spec.whatwg.org/#dom-element-attachshadow

52. Except for a few inherited properties like font-family and root font-size.

53. Encapsulating Style and Structure with Shadow DOM css-tricks.com/encapsulating-style-and-structure-with-shadow-dom/

54. Unidirectional Data Flow en.wikipedia.org/wiki/Unidirectional_Data_Flow_(computer_science)

55. At the time of writing this, Safari is the only browser that hasn't implemented it. caniuse.com/#feat=broadcastchannel

56. Broadcast Channel API developer.mozilla.org/en-US/docs/Web/API/Broadcast_Channel_API

57. OAuth en.wikipedia.org/wiki/OAuth

58. github.com/ReactTraining/history

59. single-spa.js.org

60. single-spa.js.org/docs/getting-started-overview.html

61. angular.io/api/platform-browser/Meta

62. vue-meta.nuxtjs.org

63. github.com/nfl/react-helmet

64. Michael Jackson: Universal JavaScript cdb.reacttraining.com/universal-javascript-4761051b7ae9

65. Isomorphic JavaScript: The Future of Web Apps medium.com/airbnb-engineering/isomorphic-javascript-the-future-of-web-apps-10882b7a2ebc

66. Elyse Kolker Gordon: Isomorphic Web Applications - Universal Development with React www.manning.com/books/isomorphic-web-applications

67. There are custom solutions available in projects like Skate.js or Andrea Giammarchi's project Heresy. But since the W3C spec defines Shadow DOM as a pure client-side concept, we don't have a web standard to build upon for proper hydration.

68. Ara Framework github.com/ara-framework

69. single-spa: Figure out server rendering github.com/CanopyTax/single-spa/issues/103

70. Aral Balkan - Sites vs. Apps defined: the DocumentstoApplications Continuum. ar.al/notes/the-documents-to-applications-continuum/

71. Manfred Steyer - A Software Architect's Approach Towards Using Angular (And SPAs In General) For Microservices Aka Microfrontends www.softwarearchitekt.at/aktuelles/a-software-architects-approach-towards/

72. Addy Osmani & Ilya Grigorik - Speed is now a landing page factor for Google Search and Ads developers.google.com/web/updates/2018/07/search-ads-speed

73. WPO stats wpostats.com/

74. Luke Wroblewski - Mobile Design Details: Avoid The Spinner www.lukew.com/ff/entry.asp?1797

75. Denys Mishunov - True Lies Of Optimistic User Interfaces www.smashingmagazine.com/2016/11/true-lies-of-optimistic-user-interfaces/

76. RequireJS requirejs.org

77. CommonJS www.commonjs.org

78. Can I use - JavaScript modules: dynamic import() caniuse.com/#feat=es6-module-dynamic-import

79. Ilya Grigorik: Analyzing Critical Rendering Path Performance developers.google.com/web/fundamentals/performance/critical-rendering-path/analyzing-crp

80. Lighthouse developers.google.com/web/tools/lighthouse

81. Uses inefficient cache policy on static assets developers.google.com/web/tools/lighthouse/audits/cache-policy

82. Critical Request Chains developers.google.com/web/tools/lighthouse/audits/critical-request-chains

83. Sticky Sessions in Kubernetes medium.com/@zhimin.wen/sticky-sessions-in-kubernetes-56eb0e8f257d

84. S. Kraus, G. Steinacker, O. Wegner - Teile und Herrsche: Kleine Systeme für große Architekturen - OBJEKTspektrum 05/2013 (German) ottodev.files.wordpress.com/2013/11/teile-und-herrsche-kleine-systeme-fc3bcr-groc39fe-architekturen.pdf

85. Tim Kadlec - Setting a performance budget timkadlec.com/2013/01/setting-a-performance-budget/

86. If you are building an Angular project, you should check out Manfred Steyer's work on Angular, Micro Frontends and reducing Angular bundle size. www.softwarearchitekt.at/blog/

87. Webpack DllPlugin webpack.js.org/plugins/dll-plugin/

88. Webpack 5 Module Federation: A game-changer in JavaScript architecture indepth.dev/webpack-5-module-federation-a-game-changer-in-javascript-architecture/

89. Browser Support for ES Modules caniuse.com/#feat=es6-module

90. JavaScript for impatient programmers by Dr. Axel Rauschmayer exploringjs.com/impatient-js/ch_modules.html

91. rollup.js rollupjs.org/

92. @rollup/plugin-alias github.com/rollup/plugins/tree/master/packages/alias

93. Import maps github.com/WICG/import-maps

94. github.com/systemjs/systemjs/blob/master/docs/import-maps.md

95. YouTube playlist import-maps, SystemJS www.youtube.com/watch?v=3EUfbnHi6Wg&list=PLLUD8RtHvsAOhtHnyGx57EYXoaNsxGrTU

96. Podium ES modules prototype github.com/asset-pipe/podium-asset-prototype

97. github.com/trygve-lie/rollup-plugin-esm-import-to-url

98. Design Systems Gallery designsystemsrepo.com/design-systems/

99. Vitaly Friedman: Taking The Pattern Library To The Next Level www.smashingmagazine.com/taking-pattern-libraries-next-level/

100. Alla Kholmatova: Design Systems www.smashingmagazine.com/printed-books/design-systems/

101. Design System Checklist designsystemchecklist.com

102. Bootstrap getbootstrap.com

103. Material Design for Web material.io/develop/web/

104. Semantic UI semantic-ui.com

105. Blueprint blueprintjs.com

106. Tweet by @jina: "zombie style guides — style guides that aren't maintained and part of your process. they die and rot. they eat your brains." twitter.com/jina/status/638850299172667392

107. Nathan Curtis: Team Models for Scaling a Design System medium.com/eightshapes-llc/team-models-for-scaling-a-design-system-2cf9d03be6a0

108. Nathan Curtis: A series of blog posts on design systems. It's a goldmine. You should read them all :) medium.com/@nathanacurtis

109. Vue.js Component Framework vuetifyjs.com/

110. Custom Elements Everywhere - Making sure frameworks and custom elements can be BFFs custom-elements-everywhere.com/

111. Duet Design System www.duetds.com/

112. Stencil is a toolchain for building reusable, scalable Design Systems. stenciljs.com/

113. Yes, there are ways to render them on the server. But there's no standardized templating in the current web components spec (github.com/whatwg/html/issues/2254). That's why all current solutions require a lot of hacking and fiddling.

114. X-DASH - Shared front-end for FT.com and The App financial-times.github.io/x-dash/

115. This might be the hot new thing in a couple of years. You never know :)

116. Brad Frost: Extending Atomic Design bradfrost.com/blog/post/extending-atomic-design/

117. Nathan Curtis: Design System Tiers medium.com/eightshapes-llc/design-system-tiers-2c827b67eae1

118. Storybook Docs github.com/storybookjs/storybook/tree/master/addons/docs

119. Pattern Lab patternlab.io

120. UIengine uiengine.uix.space (This is the tool we are using in most projects.)

121. 1390 times up to this line to be exact (excluding figures). (TODO: Verify and this number after editing)

122. Wikipedia: Conway's Law en.wikipedia.org/wiki/Conway%27s_law

123. Harvard Business School: Exploring the Duality between Product and Organizational Architectures: A Test of the Mirroring Hypothesis hbswk.hbs.edu/item/exploring-the-duality-between-product-and-organizational-architectures-a-test-of-the-mirroring-hypothesi

124. Martin Fowler: Tag Domain-Driven Design martinfowler.com/tags/domain%20driven%20design.html

125. Eric Evans: Domain-Driven Design, 2003 Addison-Wesley Professional

126. Wikipedia: Design Thinking en.wikipedia.org/wiki/Design_thinking

127. Harvard Business Review: The “Jobs to be Done” Theory of Innovation hbr.org/podcast/2016/12/the-jobs-to-be-done-theory-of-innovation

128. Harvard Business School: What Customers Want from Your Products hbswk.hbs.edu/item/what-customers-want-from-your-products

129. Seam Newman: Backends For Frontends samnewman.io/patterns/architectural/bff/

130. Cross-functional team en.wikipedia.org/wiki/Cross-functional_team#Effects

131. Agile Fluency Model www.agilefluency.org

132. Manifesto for Agile Software Development agilemanifesto.org/

133. Wikipedia: Community of practice en.wikipedia.org/wiki/Community_of_practice

134. Agile Team Organisation: Squads, Chapters, Tribes, and Guilds medium.com/@achardypm/agile-team-organisation-squads-chapters-tribes-and-guilds-80932ace0fdc

135. SAFe: Organizing by Feature or Component www.scaledagileframework.com/features-and-components/

136. Spotify Spotify engineering culture labs.spotify.com/2014/03/27/spotify-engineering-culture-part-1/

137. Martin Fowler: StranglerFigApplication martinfowler.com/bliki/StranglerFigApplication.html

138. Wikipedia: Monorepo en.wikipedia.org/wiki/Monorepo

139. Podium: Local Development podium-lib.io/docs/podlet/local_development

140. single-spa Dev Tools: single-spa-inspector single-spa.js.org/docs/devtools

141. Martin Fowler: TestPyramid martinfowler.com/bliki/TestPyramid.html