Micro Frontends IN ACTION

Michael Geers

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Micro Frontends in Action
Version 3

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

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 frontend scaling
- Recognizing the challenges that this architecture introduces

1.1 The Big Picture

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 amazing. 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 it's not possible for one developer to 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 decisions were made 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.

To mitigate this effect projects often get divided into multiple pieces. It became popular to divide the software, and thereby also the team structure by technology. Introducing 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 database to user interface and run by a dedicated team. The different team frontends are integrated to create a page in the customer's browser. This approach is related to the popular microservices

architecture. But the main difference is that a service is extended to include the user interface. Removing the need for a central frontend team. This comes with a lot of benefits:

- **Faster feature development:** A team includes all skills to develop a feature. No coordination between separate frontend- and backend teams is required.
- **Easier frontend upgrades:** Each team owns its complete stack from frontend to database. Teams can decide to update or switch their frontend technology independently.
- **More 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 solves and when it makes sense to use them.

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 topic. But don't worry - we'll go through this diagram from bottom to top.

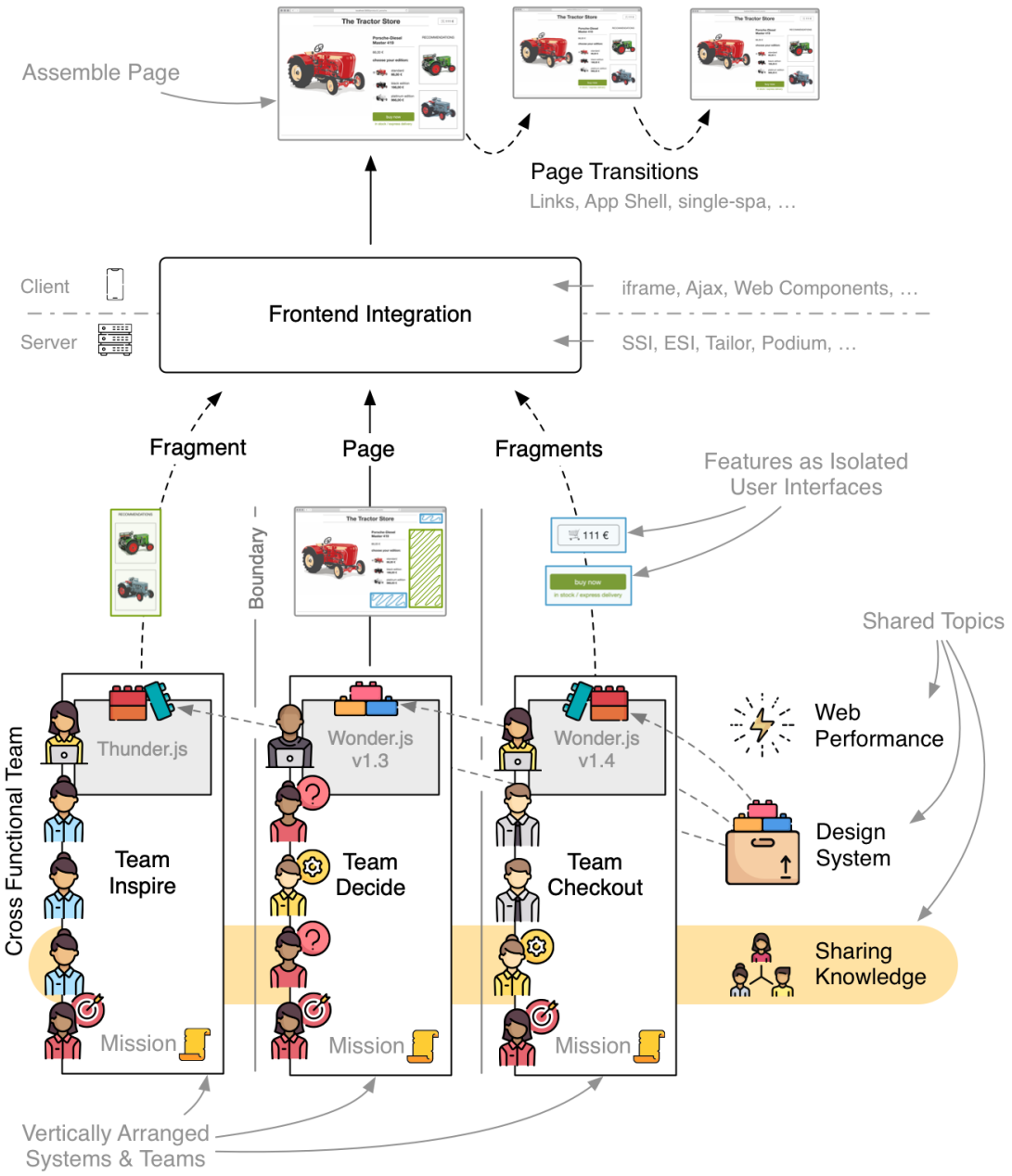

Figure 1.1 Big picture overview of the micro frontends approach. The vertically arranged teams at the bottom are the core of this architecture. They each produce features in the form of pages or fragments. These are then integrated using techniques like SSI or Web Components to an assembled page that reaches the customer.

1.1.1 Systems & Teams

The three boxes at the bottom are the vertically arranged software systems. They form the core of this architecture. Each system is autonomous, which means it is able to function even when the neighbor systems are down. To achieve this every system has its own data-store. 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'll not cover the backend aspects like data replication between these systems. Here established solutions from the microservices world apply. We'll focus on the organizational challenges and frontend integration.

TEAM MISSIONS

Each team has its own 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.

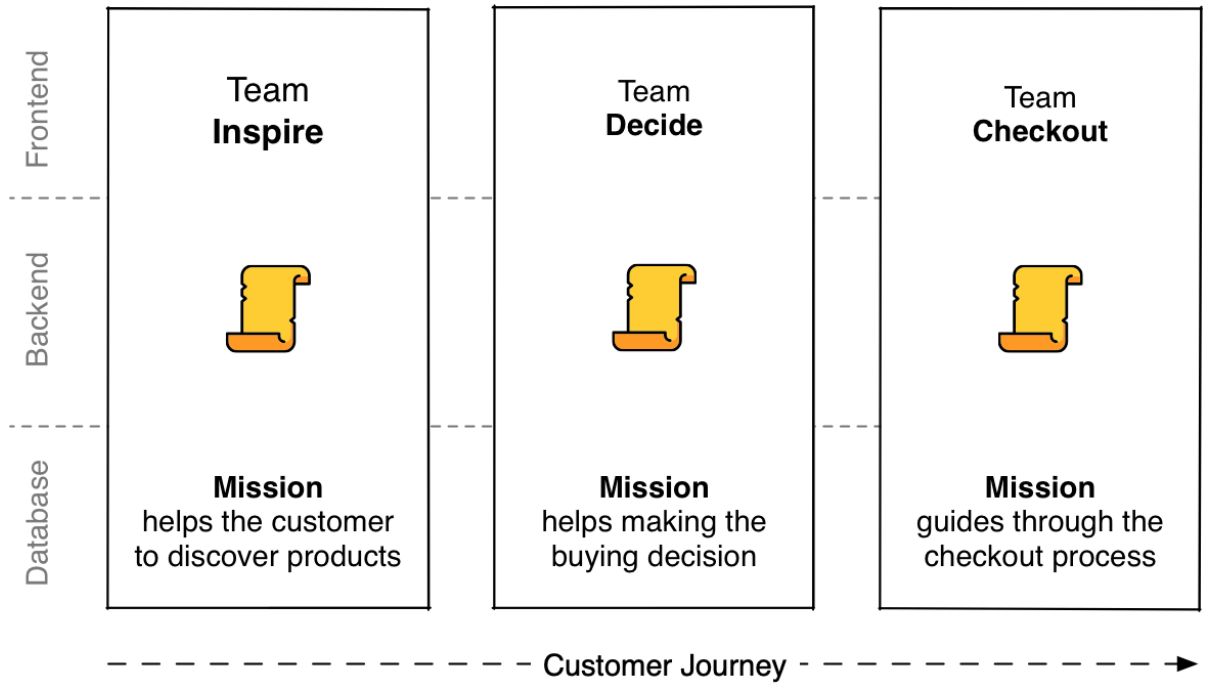

Figure 1.2 E-commerce example with three teams and their missions. Teams are formed along the customer journey.

They are formed 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 good 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.

The clear mission is important for the team. It provides focus and is the basis for dividing the

software system.

CROSS FUNCTIONAL TEAMS

The biggest difference, which micro frontends introduces 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 working on the frontend, experts in handling payment working on a payment service. Business and operations experts also form their own teams. This is a typical structure when using a microservices approach.

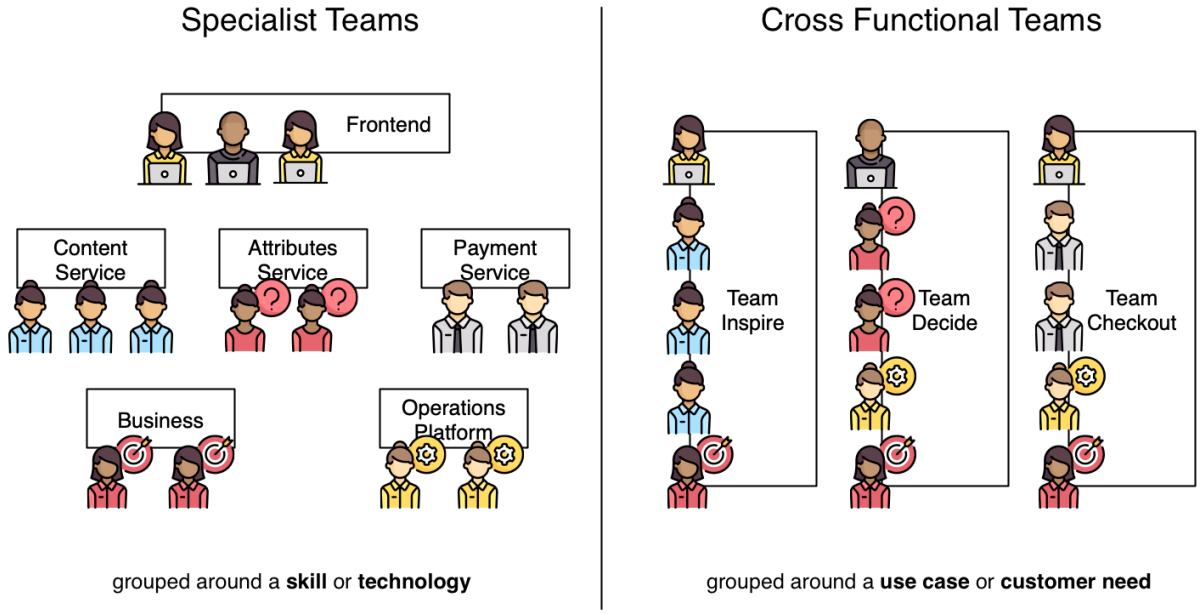

Figure 1.3 Team structure of a microservice style architecture on the left compared with micro frontends teams on the right. Here the teams are formed around a customer need and not based on technologies like frontend and backend.

It feels natural on 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 is not necessarily true. 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 teams mission. For example coming up with

solutions to present relevant product suggestions or to make the checkout process even smoother. 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 additional 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

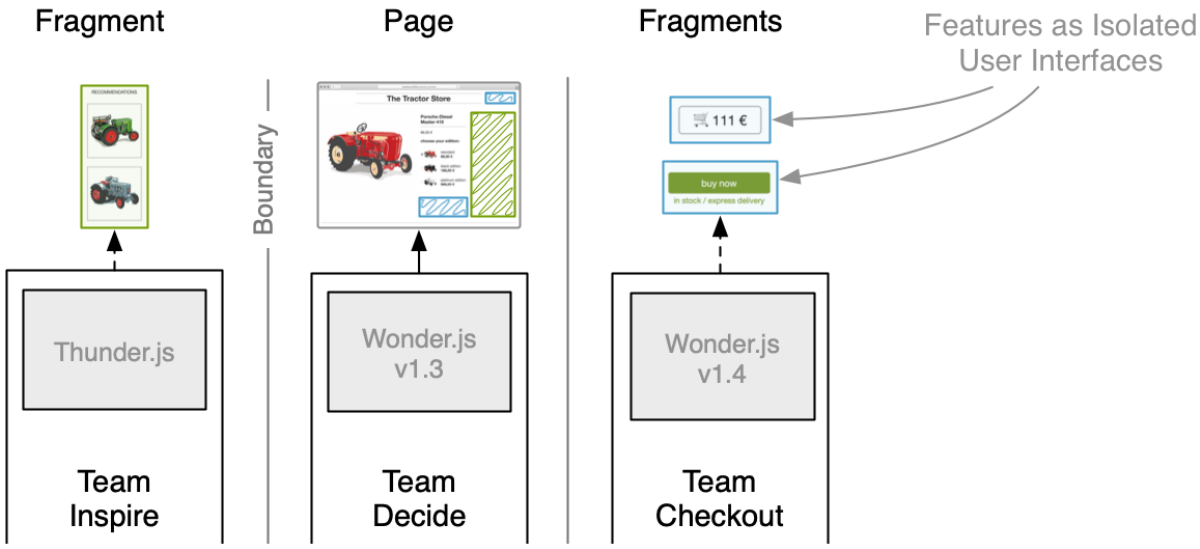

Figure 1.4 This is the middle part of the big picture diagram. Each team builds it's own user interface as a page or a fragment.

This is where the actual frontend work gets done. Each team delivers its **own frontend**. This means it 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. Library and framework code is not shared across teams. 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 Decide* uses Wonder.js v1.3 where as *Team Checkout* 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:

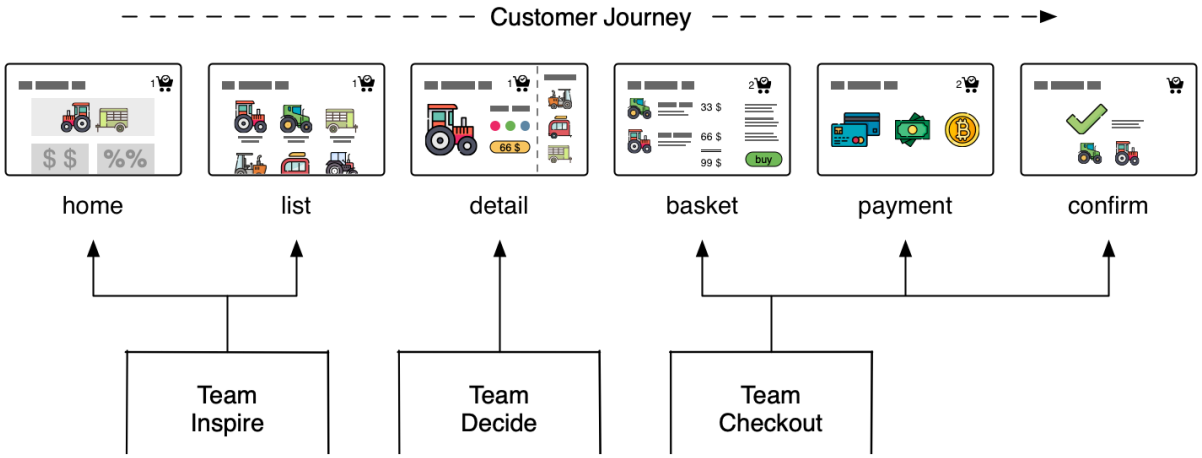

Figure 1.5 Each page is owned by one team

Because the team structure resembles the customer journey this page type mapping works well. The focus of a homepage is indeed inspiration and a product detail page is the spot where buying decisions are made.

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 autonomous 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.

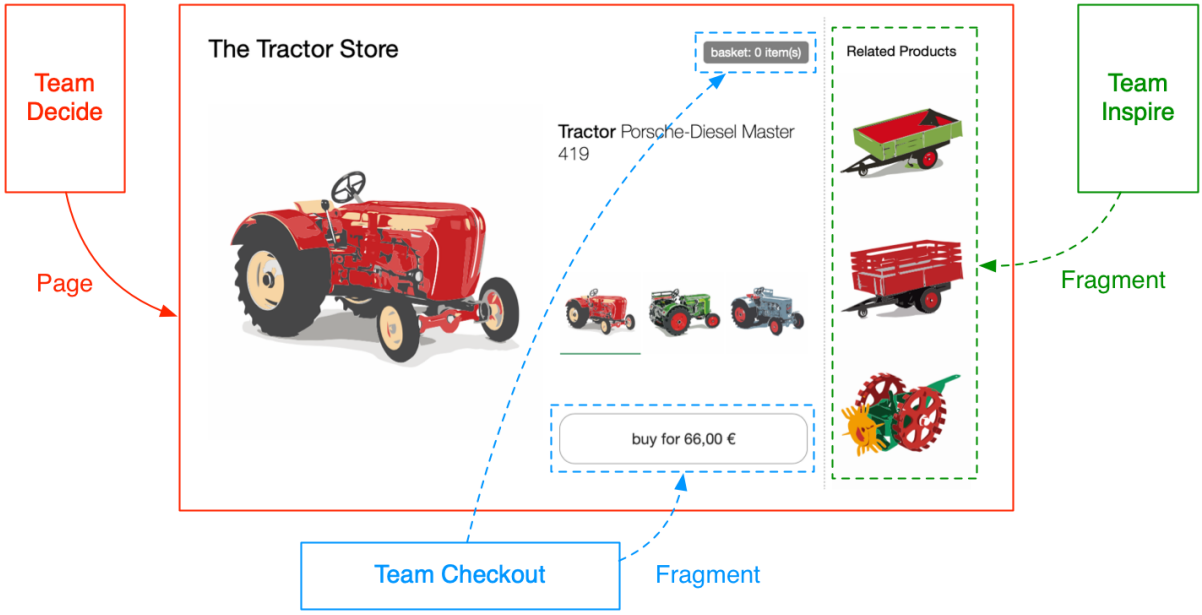

Figure 1.6 Teams are responsible for pages and fragments. You can think of fragments (dashed boarder) as embeddable mini applications that are isolated from the rest of the page.

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 "Related Products" block on the right is in inspirational element. *Team Inspire* knows how to produce those. The "Mini Basket" at the top shows the number of basket 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, the button itself is also produced by the checkout team.

A team can decide to include functionality from another team by adding it somewhere in 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 Integration

Figure 1.7 shows the upper part of our big picture diagram. This is where it all comes together.

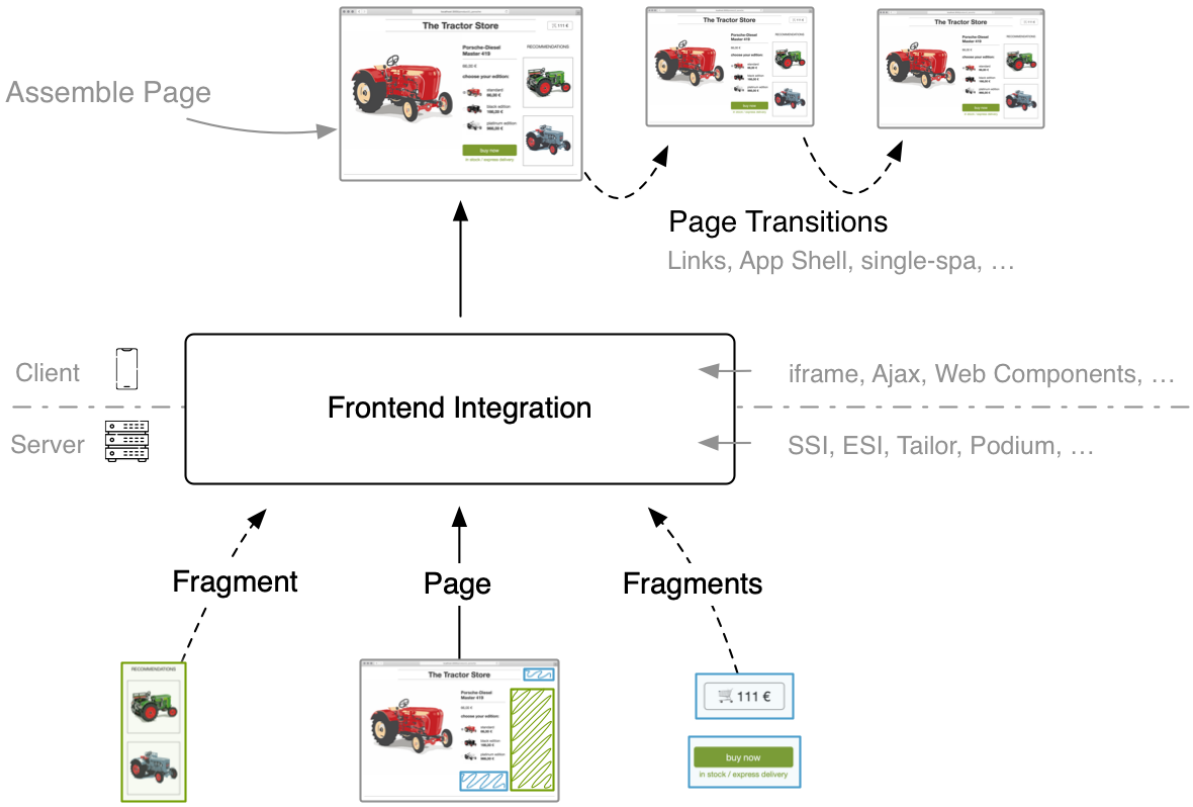

Figure 1.7 There are multiple options for integrating the frontend parts

FRONTEND INTEGRATION

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

A separate integration service or technique does the final assembly. There are different ways of achieving this. You can group the solutions into two categories:

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

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

For interactive applications you also need a model for communication. In our example the number in the "Mini Basket" should increment 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 is also part of frontend integration.

In the following four chapters of this book you'll learn about different integration techniques and the benefits and drawbacks they provide.

PAGE TRANSITIONS

With the **Assembled Page** we've reached the top of our overview diagram. The placeholders are filled with the correct fragment content. To get from the product page to the next we need **Page Transitions**. You can achieve this by simply 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 **App Shell** or use a meta framework like **single-spa**. We will look into both options in this book.

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 there are some shared topics that are important to address when working like this.

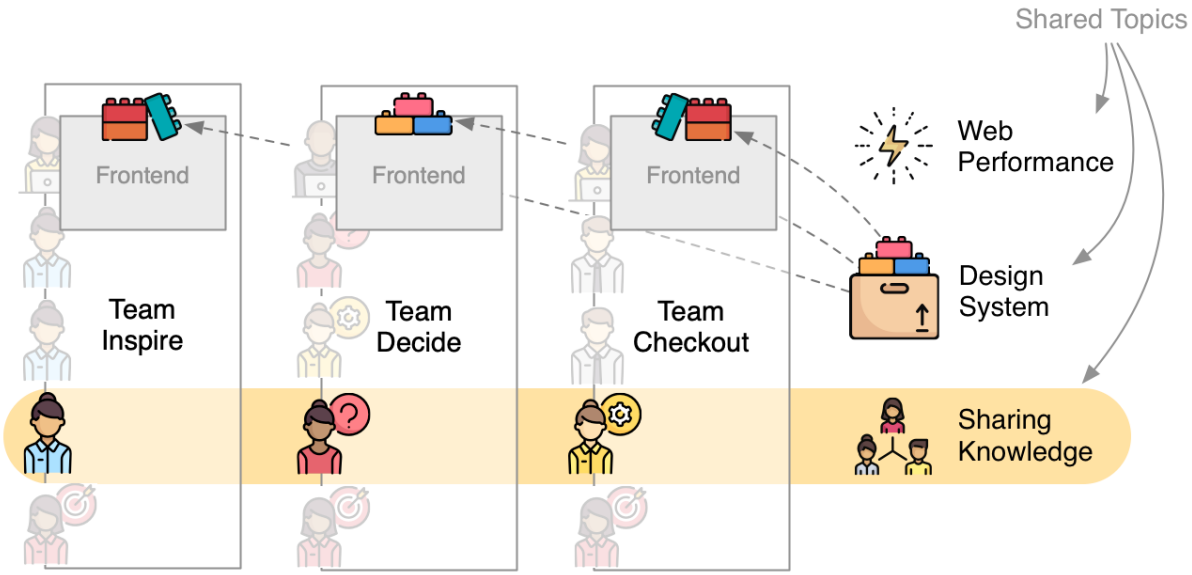

Figure 1.8 To ensure a good end-result and avoid redundant work it's important to address topics like web performance, design systems and knowledge sharing from the start.

WEB PERFORMANCE

Having a page that is assembled from fragments of multiple teams often results in more code being shipped to the browser. Its 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.

DESIGN SYSTEMS

To ensure a coherent 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 9. Coherent User Interface you'll learn different ways of implementing a design system.

SHARING KNOWLEDGE

Autonomy is important, 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 on a regular basis between teams.

1.2 What problems do Micro Frontends solve?

Now you have an idea what micro frontends are. Let's have a closer look at the organizational and technical benefits they provide.

1.2.1 Faster feature development

The number one reason why companies choose to go the micro frontend route is development speed.

In a layered microservices architecture multiple teams are involved in building a new feature. 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 coordinate how the new data should be passed to the frontend. Meetings are arranged and 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. If not, more meetings are scheduled to discuss changes.

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

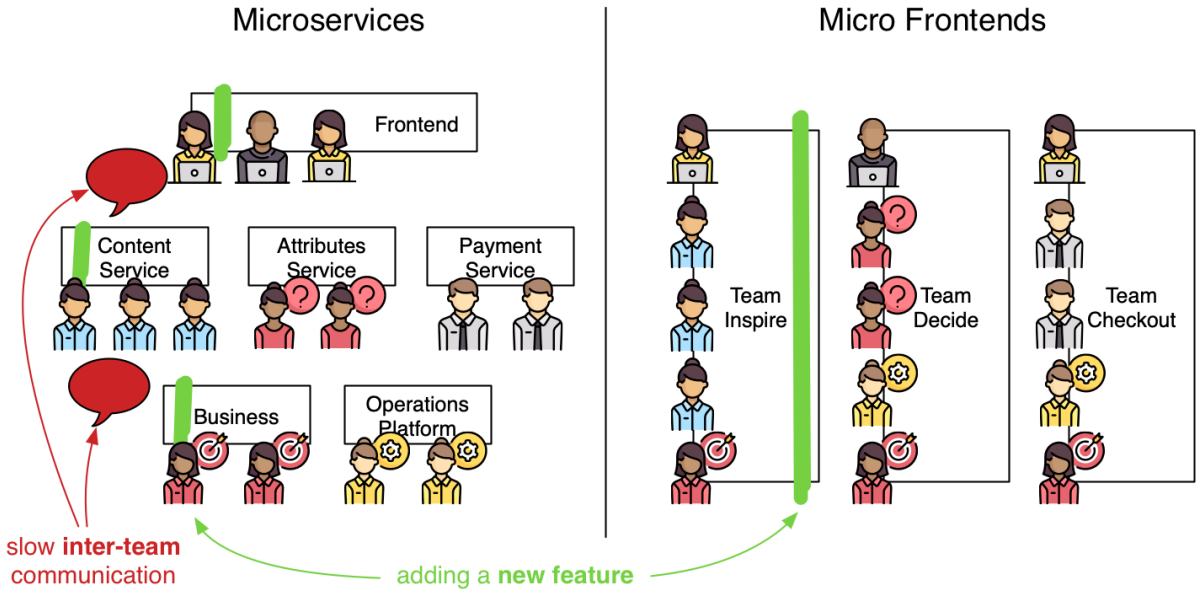

Figure 1.9 With a layered microservices multiple teams are involved to build a feature. Inter-team communication is slow. Teams need to synchronize and wait for each other. With the micro frontends approach all people needed are in the same team. Features get build faster.

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 code-base that only one team can work on sensibly.

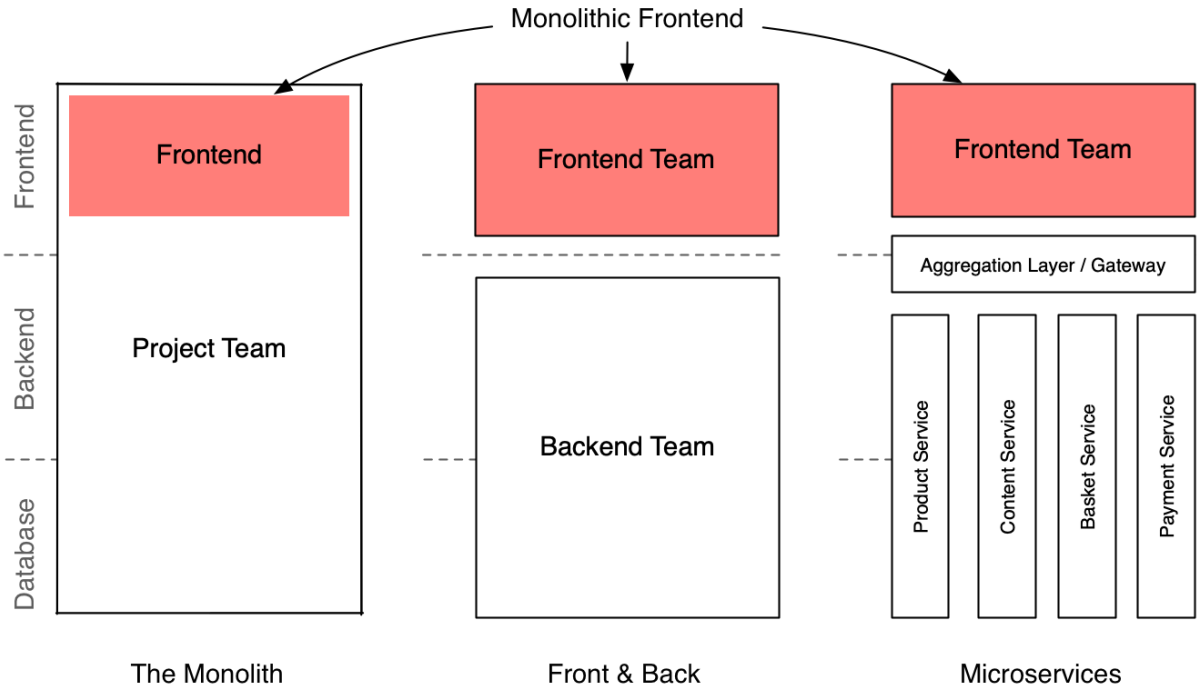

Figure 1.10 In most architectures the frontend is a monolithic system.

With micro frontends the application including the frontend gets split into smaller vertical systems. Compared to a monolith there are multiple benefits. A vertical system …

- is easier to test.
- is narrow in scope.
- has a smaller codebase.
- is easier to refactor.
- is independently deployable.
- is easier to understand and maintain.
- can use the technology that fits best.
- isolates the risk of failure to a smaller area.
- 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. Starting 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 is closely related to the rising expectations for a frontend.

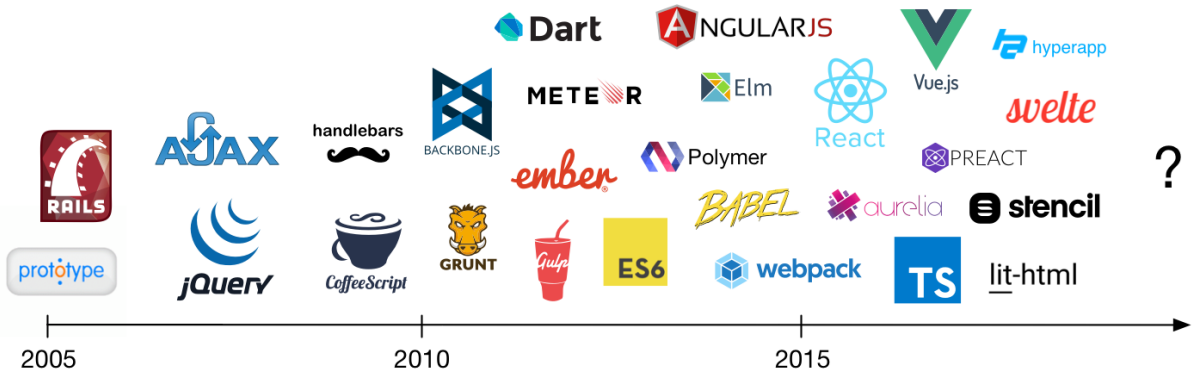

Figure 1.11 Tools in frontend development are changing fast. Being able to adapt, when it makes sense is important.

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 to maintain 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 a big 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 its impossible for one developer to 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 certain size and want to stay competitive it's important to be able to move to new technologies when they provide value for your team. This 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 its crucial to have a checkout process thats as bug free as possible the Team decides to switch to Elm which is a new 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 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.

Since they are in full control of their technology stack (*Micro Architecture*), its easy for the team 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.

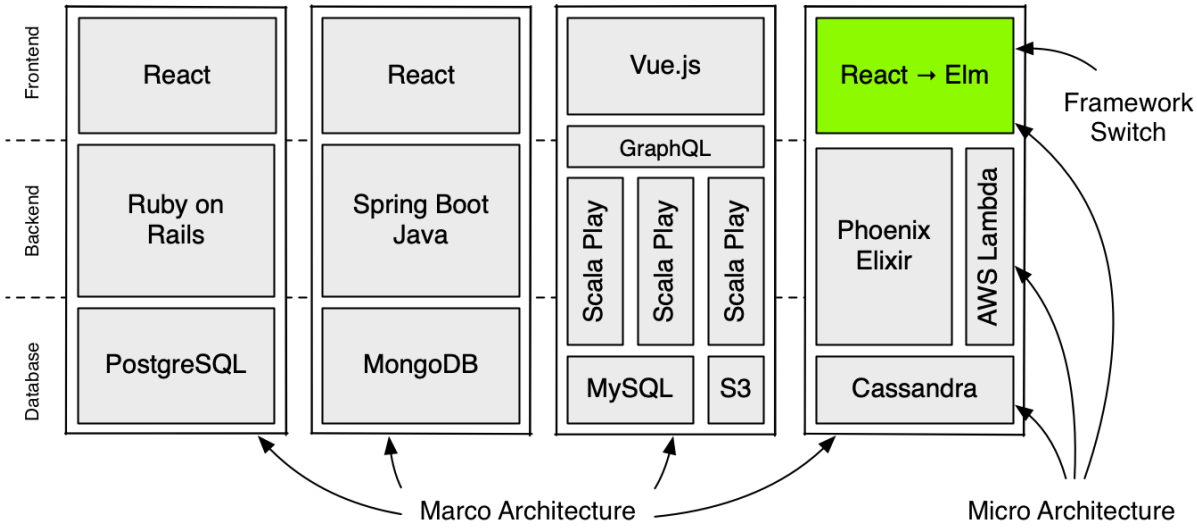

Figure 1.12 Teams can decide about their internal architecture (micro architecture) on their own as long as they stay in the boundaries of the agreed upon macro architecture.

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 key benefits of microservices and also of micro frontends. It comes in handy when teams decide to make bigger 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 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.

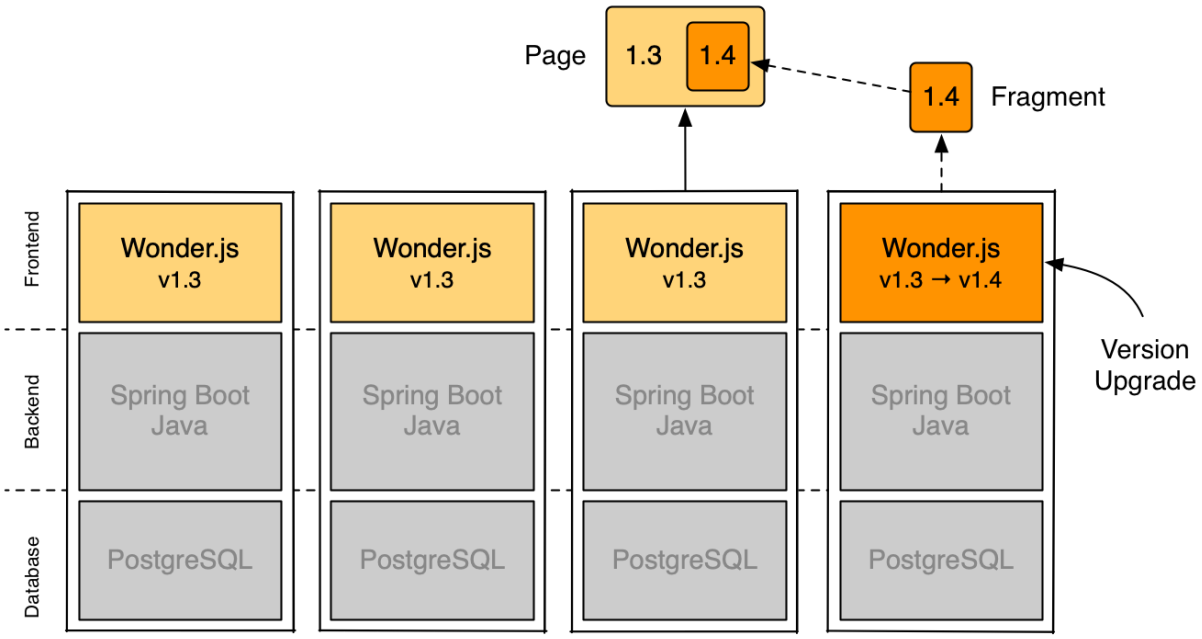

Figure 1.13 Fragments are self-contained and upgradable independently of the page they are embedded in.

On first sight it sounds wasteful that every team brings their own assets. Especially when you know that other 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 actually 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 bigger server instances.

This does not apply to frontend code. The bandwidth and computing resources available for loading and running your frontend are limited by the connection and device of your customer. This does not mean, that a setup with three teams comes with three times the amount of code in the browser. In chapter 8. 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 people 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.

To enable faster feature development, the goal is to share as little as possible. 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. They are common parts like web fonts that a save to share between teams. You'll learn more about making these tradeoffs in chapter 7. Which integration fits my project?

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 is 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 setup 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 benefit from this change. This team has to share this information with the others and they have to implement the same optimization on their own.

The reasoning behind this shared nothing architecture is, that the costs associated with this 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 for this. All teams need to know what products the shop actually 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 on a regular basis. 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 product that's promoted with a discount on the homepage might not have that 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.

The tradeoff that's made her is to favor robustness over guaranteed consistency.

HETEROGENEITY

Free technology choice is one of the biggest 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've 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 build 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 linear with the number of teams or fragments. But it's extra important 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 important 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 hand full of people scaling is probably not your main issue. The *Two-Pizza Team Rule* Amazon CEO Jeff Bezos propagates, is an indicator for a good team size.[5] It says that a team is to big when it can't be fed by two large pizzas. In larger groups communication overhead increases and decision making gets complicated. In practice this means that a perfect team size is between 5 to 10 people.

When the team exceeds that size it is worthwhile considering a team split. Doing a vertical, micro frontend style, split is definitely 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 and DAZN use this end-to-end approach in 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 Squats*. They act as support teams that build tools like A/B-testing for the feature teams to make them more productive.

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 Apples or Googles review process. One possible way around this is to load parts of the application from the web via an embedded browser or WebView and 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 others toes.

It's of cause always possible that every vertical team could have a web frontend and also expose their functionality through a REST API. This API can be used to add other user interfaces like native apps on top. A native app would than reuse the existing business logic provided by the API. But it would still form a horizontal monolithic layer that sits above the vertical teams. 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 its 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 build the associated help desk functionality for the customer hotline or the WebView-based version of the checkout that can be embedded in a native app.

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, setup the systems and implement an integration strategy. You need to establish common rules that all teams agree on like using namespaces. Its 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 own 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 bigger than the added organizational complexity.

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, there are reports that the teams who run its e-commerce site have been working like this for a long time. Amazon also uses a UI integration technique which 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*. Canopy[11], an American tax software startup and the sports streaming service DAZN[12] use micro frontends to build its applications.

1.4.1 Where micro frontends are not a great fit

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

To make good vertical cuts it's very important to know the domain you are working in well. 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 cause 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. The 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.

1.5 The scope of this book

This book consists of three parts. In the next two chapters of part 1 we will build the foundation for this book. Setting up a simple e-commerce shop developed by two independent teams. You'll get into the right mindset, learn the concepts of coupling and do first integrations using links, iframes and Ajax.

In part 2 we will do a deep dive into different integration techniques. Focusing on server side integration with SSI, client side integration with Web Components and simple page transitions. We'll also shine a light on some alternative solutions and end with a comparison that helps you pick a technology that's appropriate for your purpose.

The last part of the book includes a variety of topics that go along with micro frontends. We'll cover design systems, web performance, the organizational aspects and explain how testing works with this approach.

1.6 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 important to address web performance from the start.
- There are multiple frontend integration techniques that work either on the client or on 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 afterwards works but creates friction.

2 Loosely Coupled

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 larger application with multiple teams in parallel is the key feature of micro frontends. But the 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 very little coupling between the teams. No shared infrastructure, libraries or code conventions are required. This 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*. This project will be used 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. But competition does not sleep. A Scandinavian company already sells wooden tractor models worldwide and reports say that they are planning to build tin models in the near future as well. So time is of the essence to conquer the hearts of tractor fans and become market leader.

The company want’s 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 off with two teams.

We’ll set up the software project for both teams. *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. These pages are connected 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 needs to be done to get their pages ready.

SOFTWARE STACKS

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. The’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 car configurator.

SIDEBAR **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.

It's tempting to take this blueprint approach one step further and make it into a dependency that all teams must use. But this would introduce a shared piece of code that someone needs to own and maintain. Try to avoid this and share as little as possible to maintain team independence. Teams should have the possibility to deviate from the norm if they have a compelling reason to do so. In chapter 7.2. Full autonomy vs. unified platform we'll go deeper into this topic.

INDEPENDENT DEPLOYS

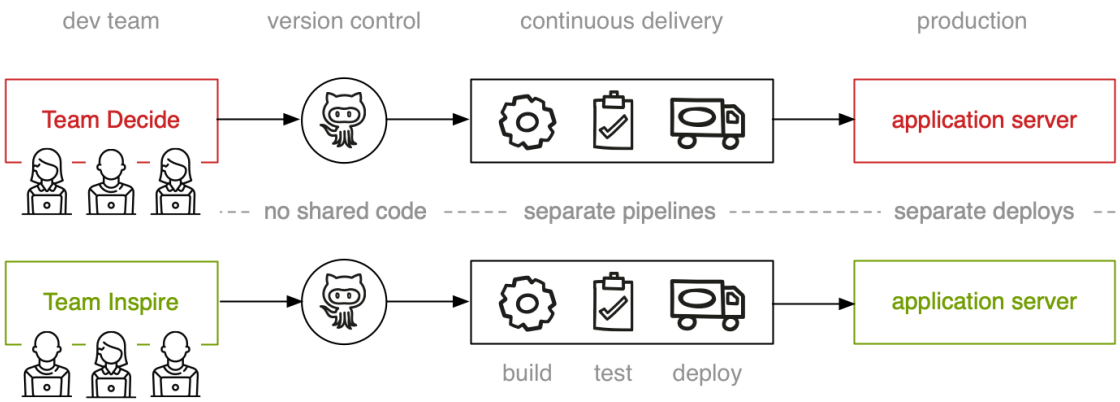

Figure 2.1 Teams work in their own source code repository, have separate integration pipelines and can deploy independently.

Both teams create their own source code repository and setup 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 softwares correctness and deploys the new version of the application to the teams production server. These pipelines run independently. A software change in *Team Decide* will never cause *Team Inspire's* pipeline to break.

2.1.2 This books 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** The source code for each chapter is also available at the-tractor.store. So if you don't want to type or copy the example code by hand. Feel free to download it and move along.

We'll create two team folders side by side. Each folder represents a teams application. Create the following structure in a place you like:

```
sample-code/
  team-decide/
    static/
  team-inspire/
    static/
```

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. After that you should be able to run these commands in your terminal.

```
node -v
v12.4.0
npm -v
6.9.0
```

If your version number is equal or higher you are good to go.

> **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.

Run the following command to install the ad-hoc web-server globally.

```
npm install @microfrontends/serve -g
```

> **NOTE** The `@microfrontends/serve` package is based on the popular `zeit/serve` web-server but with a few features like logging, custom headers and support for delaying request added. We'll need these features in the following chapters.

Now navigate into your `sample-code` folder and start a server that listens on port `3001` and

serve the content of the `team-decide` folder.

```
cd sample-code
mfserve --listen 3001 team-decide
```

A green box with the word **Serving!** should appear in your terminal. Opening localhost:3001 in your browser shows you a directory listing which currently only includes the `static` folder. 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 Integration via links

In the first iteration of their development the teams decide to keep it as simple as possible. No fancy integration technique. Every team builds their feature as a standalone page. This page is served directly from the teams application and brings its own HTML and CSS.

They are starting out 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. The images have been uploaded to a content delivery network (CDN). They can be directly used from there.

Table 2.1 List of all products

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

Since they don't have purchasing histories or other customer data yet the product recommendations will be hard coded for now. Later in the development process *Team Inspire* plans to implement machine learning and proper prediction algorithms. Table 2.2 shows the product relations.

Table 2.2 Recommendations for each SKU

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

To make the links between the pages work, each team has to communicate their URL-pattern for their page to the other team:

- Product Page URL-pattern: `http://localhost:3001/product/<sku>` example: `http://localhost:3001/product/porsche`
- Recommendation Page URL-pattern:

`http://localhost:3002/recommendations/<sku>` example:
`http://localhost:3002/recommendations/porsche`

Because we are going to build 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 2.2. The product page shows the name and image of the tractor and links to the corresponding recommendation page. This shows a list of recommendations as images. Each image links to the matching product page.

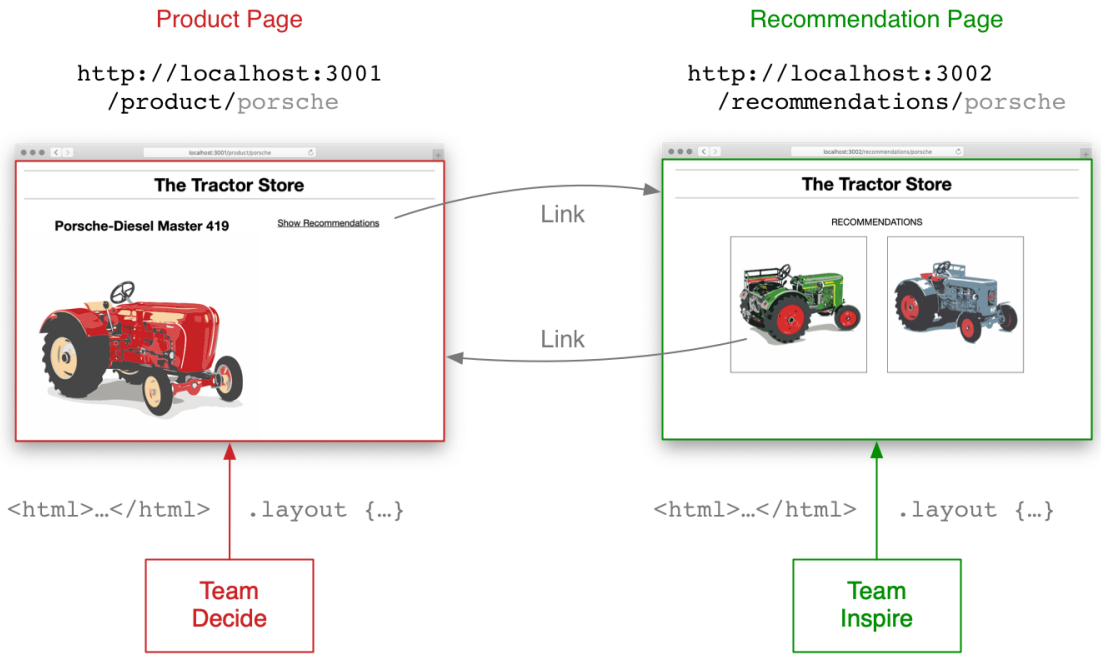

Figure 2.2 A product and recommendations page connected via links

Now the teams have everything they need to get started.

2.2.1 How to do it

Let's build both of these pages. Create the following folders and files inside the `sample-code` directory we've created earlier.

```
team-decide/
  product/
    eicher.html
    fendt.html
    porsche.html
  static/
    page.css
team-inspire/
  recommendations/
    eicher.html
    fendt.html
    porsche.html
```

```
  static/
    page.css
```

The HTML files represent the server generated output of the teams. Note that the ad-hoc web-server defaults to a `.html` extension when looking up a file. This means that requesting `/product/porsche` will serve the `./product/eicher.html` file.

MARKUP

The markup for a product page looks like this:

Listing 2.1 team-decide/product/porsche.html

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

❶ brings it's own styles

❷ tractor name

❸ tractor image

❹ link to Team Decide's matching recommendation page

The markup for the other product pages looks identical but with updated name, image and recommendation link. This link is generated according the URL-pattern which *Team Inspire* provided for the recommendation page.

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

❶ brings it's own styles

❷ link to *Team Decide's* product page

Again, the markup for the other pages is similar. Only the product page links and images need to be adjusted based on the definition from table 2.2.

So the markup from both teams looks quite similar. Both bring a header and their own styles. But the actual content is different.

STYLES

In discussions around architecture, styling is often pushed aside as an afterthought. But for most applications styles are an integral part of the software. In a micro frontends context where decoupling and autonomy are key, it's important that we view the software as a whole. That's why we don't pull in a global UI library like Twitter Bootstrap now. Instead we've prepared a small set of styles that go into the `static/page.css` file for each team. The HTML markup and the CSS are delivered directly from the teams applications.

These are the styles for the **product page**:

Listing 2.3 team-decide/static/page.css

```
* {...}            ❶
html {...}         ❶

.layout {...}      ❷
.header {...}      ❷

h2 {...}           ❸
.image {...}       ❸
```

❶ global styles

❷ product page styles

❸ product content styles

The styling rules itself are not that interesting, that's why they are left out in the listing. But notice that the file is divided into three sections. The **global styles** include normalization styles and define basic font styles for the application. The second part are the **product page styles** which defines the overall layout of the page and the header. Lastly the **content styles** for the actual content. We'll come back to this later.

These are the styles for the **recommendation page**:

Listing 2.4 team-inspire/static/page.css

```
* {...}                          ❶
html {...}                       ❶

.layout {...}                    ❷
.header {...}                    ❷

h2 {...}                         ❸
.recommendations {...}           ❸
```

❶ global styles

❷ recommendation page styles

❸ recommendation content styles

As before the CSS file is divided into three parts and contains global, page level and content level styling rules. The global styles are identical with *Team Decide*. The page styles including layout and header are also quite similar and since both teams deliver different content their content styles differ completely.

Now we've introduced redundant code. When the company want's to change the look of the header or its basic font-face all two teams need to become active and modify their Markup and CSS accordingly. This doesn't feel correct, right? But hold your horses. We'll address these topics later.

STARTING THE APPLICATIONS

To test our example we can now start out ad-hoc web-server for both teams. Run the following commands in separate terminals.

Starting the product page application.

```
npx mfserve --listen 3001 team-decide
```

Starting the recommendation page application.

```
npx mfserve --listen 3002 team-inspire
```

TIP You can run both servers in one terminal by using a NPM script called `concurrently`. After installing it globally via `npm install concurrently -g` you can pass it a list of commands in double-quotes. They will run in parallel. The complete command for our setup looks like this:

```
npx concurrently "mfserve --listen 3001 team-decide" "mfserve
--listen 3002 team-inspire"
```

Opening localhost:3001/product/porsche in your browser brings you in front of the red Porsche Diesel Master tractor. The result should look like the screenshot in figure 2.3.

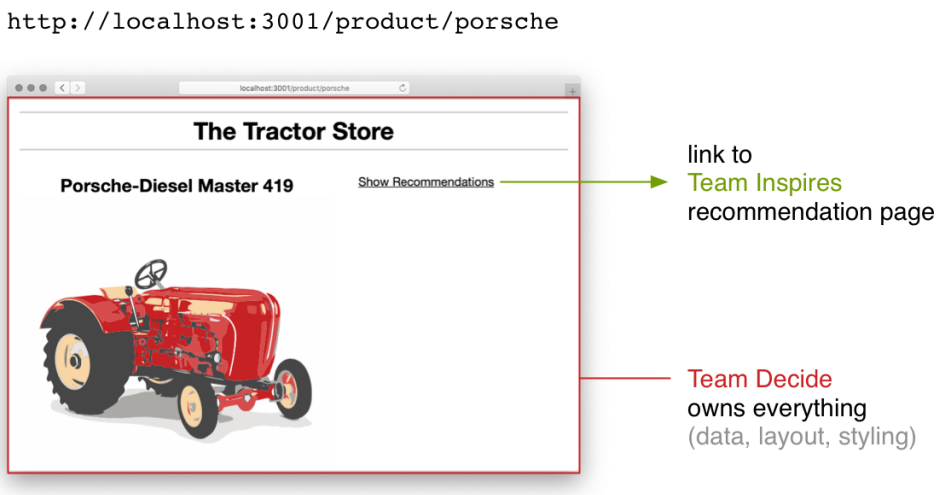

Figure 2.3 Team Decides product detail page. The team owns everything on this page.

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 finished the first task and created a web-server for each team that delivers their pages. 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.

2.2.2 Dealing with changing URLs

The integration works because both teams exchanged their URL-patterns beforehand. This is 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. Manually notifying all other teams works. 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 a great fit for a lot of use cases. A more powerful 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 on a regular basis and update their links if needed.

SIDEBAR **Implementing an URL directory**

The **URI-templates** format[13] is a good solution to specify URLs and documenting dynamic aspects. It's a popular format and URL generation libraries exist for all major programming languages.

The **json-home** spec[14] defines a JSON format that documents all available resources of a service. It uses URI-templates to do this.

You could also go with other REST API documentation formats like **Swagger OpenAPI** [15].

2.2.3 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 teams 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. This is a robust solution because the applications share nothing. They bring everything they need to deliver their own content. An error in one system can not affect the other teams system.

2.2.4 The drawbacks

The fact that the teams share nothing does come with a cost. From a user experience point of view an integration via links-only is not always optimal. The user has to click a link to see information which is owned by another team. In our case he bounces between the product and 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.

2.2.5 When do links make sense?

When you are building a somewhat complex site an integration that relies on links-only is not sufficient in most cases. Oftentimes 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 Integration via iframe

The whole company staff is extremely happy 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 obvious 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. Replacing 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 integration solutions against each other. They quickly realized that an integration 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 major drawbacks which we’ll also discuss in this chapter.

Only a few lines of code have to be change by each team. Figure 2.4 illustrates how the recommendation should be presented on the product page. It also shows the team responsibilities. The complete recommendation page gets included into the product page.

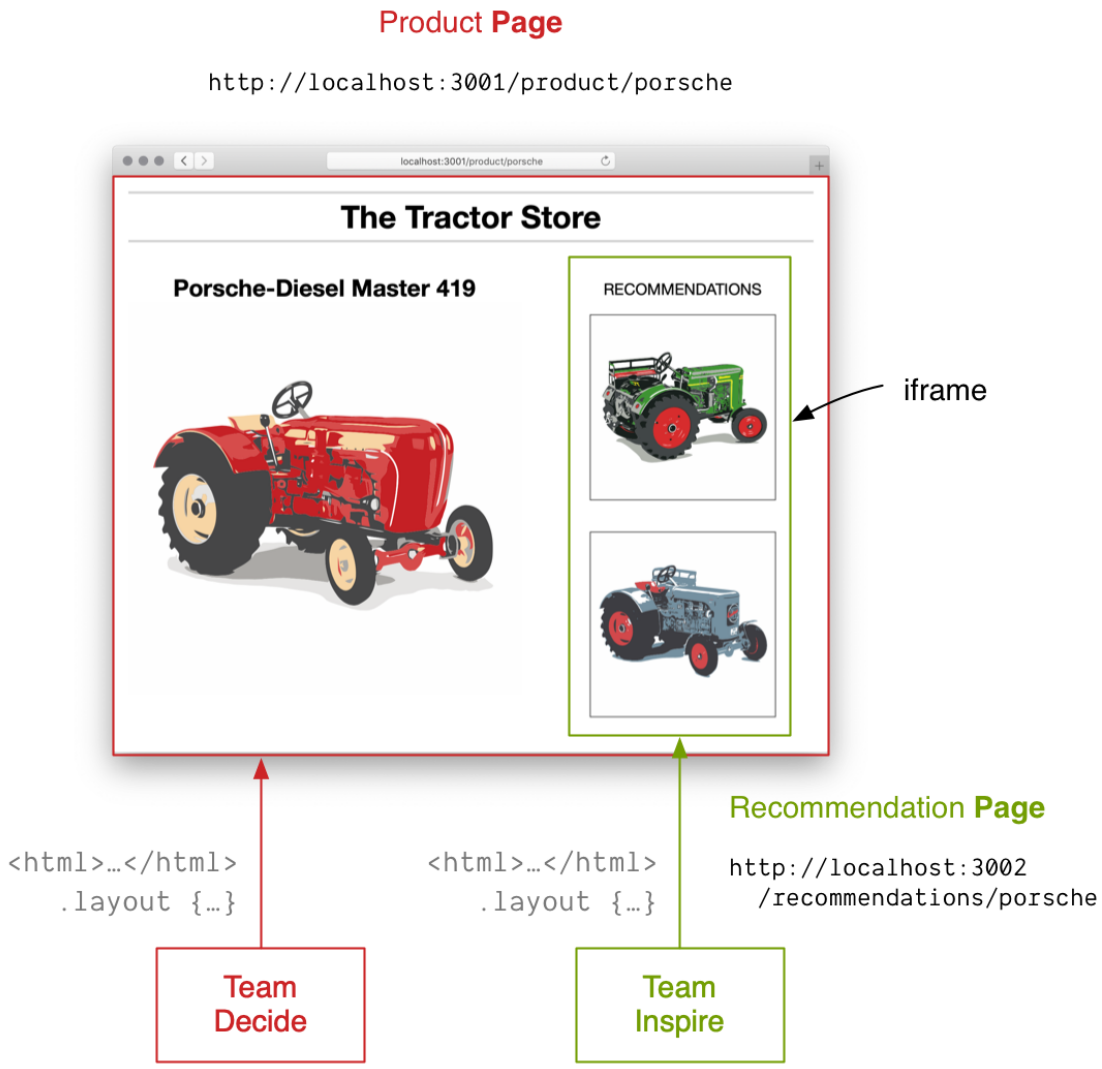

Figure 2.4 Integrating the recommendation page into the product page via iframe. These pages don't share anything. Both are standalone HTML documents with their own styling.

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 2.5 team-decide/product/porsche.html

```
...
<aside class="recos">
-  _<a href="http://localhost:3002/recommendations/porsche">...</a>_
+  <iframe src="http://localhost:3002/recommendations/porsche"></iframe>
</aside>
...
```

After that we remove the "The Tractor Store" header from *Team Inspire's* recommendation page template because we don't need it in the iframe.

Listing 2.6 team-inspire/recommendations/porsche.html

```
...
<body class="layout">
-  _<h1 class="header">The Tractor Store</h1>_
  <h2>Recommendations</h2>
...
```

Let's open the product page localhost:3001/product/porsche and see what we've got. Figure 2.5 shows the page with integrated recommendations.

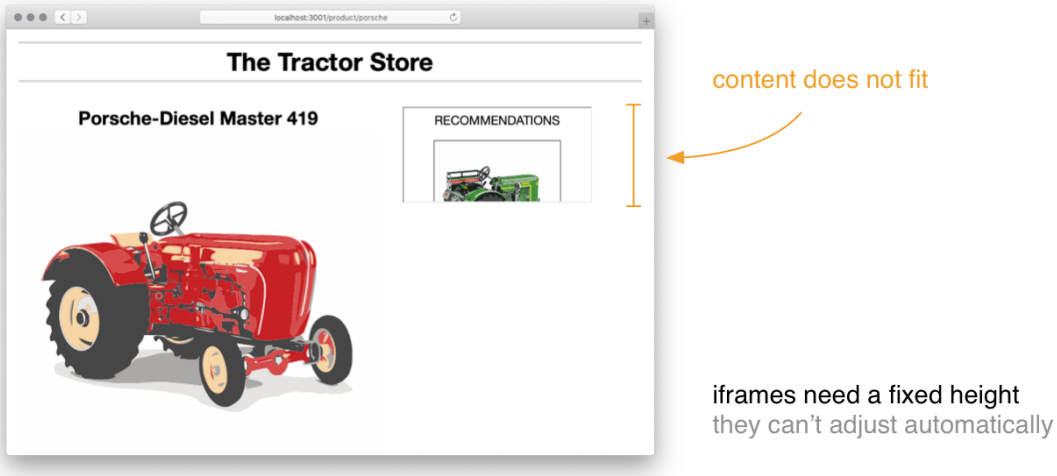

Figure 2.5 Iframes have a fixed size by default

But the recommendations are cut off at the bottom and the scrollbar is visible. This is because the iframe itself is not high enough for its content. Thats why the browser makes the content scrollable. When we fix the height issue, the scrollbar should disappear automatically.

2.3.2 Layout coupling

Let's talk about iframe sizing. This is a surprisingly complicated topic. By default browsers render an iframe with 300px width and 150px height.[16] Since we are building a responsive site we want the iframe to always adjust its width to the container its placed in. We can achieve this behavior with `width: 100%`. The height is trickier, there is no way to set a dynamic height to the iframe so that it always takes the space of its content. We can of cause set it to a fixed height. In our case setting it to `height: 750px` would work. But this is not optimal and leads to unwanted white-space on smaller screens, but it works for now. Let's add the appropriate styling to the bottom of *Team Decide's* `page.css` and see what happens.

Listing 2.7 team-decide/static/page.css

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

❶ Remove the browsers default iframe border

❷ iframe should be as wide as its parent container

❸ fixed height to make enough space for the content

After a reload the page should look as expected and the iframe has enough space for it's content.

NO INDEPENDENT LAYOUT CHANGES

We've introduced a coupling between *Team Decide* and *Team Inspire*. *Team Decide* needs to know the exact height of the recommendation view. When *Team Inspire* wants to change their view and e.g. show three instead of two recommendations, they have to coordinate with *Team Decide* first to avoid scrollbars or unwanted whitespace.

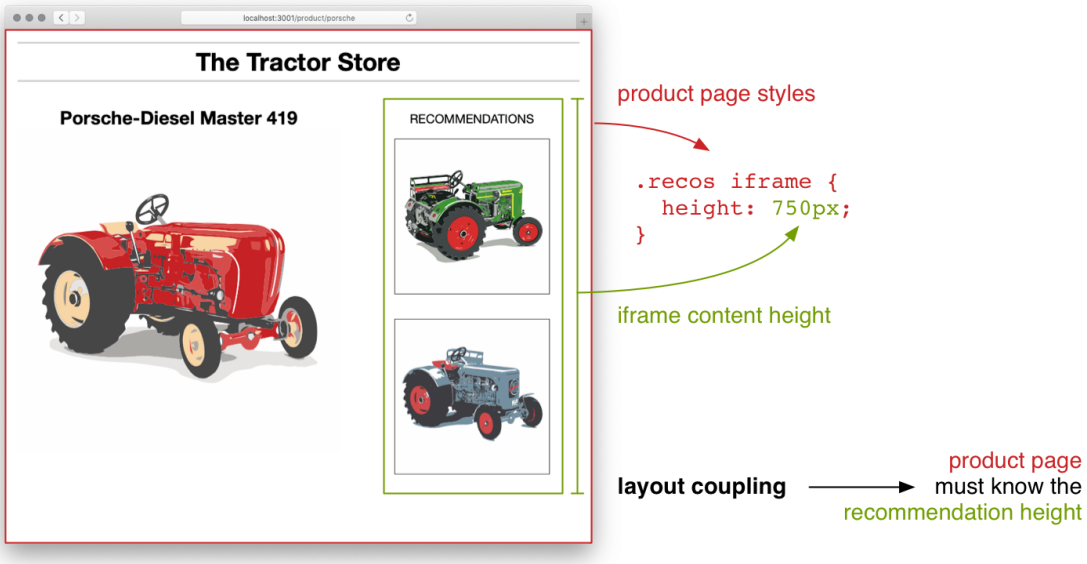

Figure 2.6 The outer page needs to know the size of the iframes content. This introduces layout coupling. Product page styles need to be updated every time the recommendation layout changes.

SIDEBAR **Automatic height calculation workaround**

There are ways to work around this with JavaScript by using a library like *iframe-resizer* [17]. But it's generally not a good idea to do layout in JavaScript. Users will see janks and reflows when loading the page. But depending on the use case this might be an option.

2.3.3 Interactions

A document that's embedded via an `iframe` runs in its own small browser window. This is great, because we get full isolation between the outer and the inner page. No leaking CSS or JavaScript. But its also possible to do interactions between these two pages.

NAVIGATION

When you click on a recommendation, you navigate to the detail page. But this detail page is displayed inside of the recommendation `iframe` by default. This is not the behavior we intended. We would have expected the outer page to update. Adding `target="_parent"` to the product detail link makes this happen.

Listing 2.8 team-inspire/recommendations/porsche.html

```
<a href="..." target="_parent">...</a>
```

Using the page and switching between tractors now feels natural. It's more user friendly than the separate pages solution we had before.

COMMUNICATION

You can also exchange messages between the iframe and the page which includes it via JavaScript. This makes it possible to implement interactive features where the iframe can react to events that happened in the page and vice versa. We won't go into details on this but the MDN documentation of the `window.postMessage()` API[18] is a good starting point to learn more.

2.3.4 Error scenarios

Let's see what happens when one of the teams goes offline due to an error. To simulate this we can stop *Team Inspires* server. When we reload the product page the recommendations on the right disappear. Instead we see a an embedded browser error page like the one in figure 2.7.

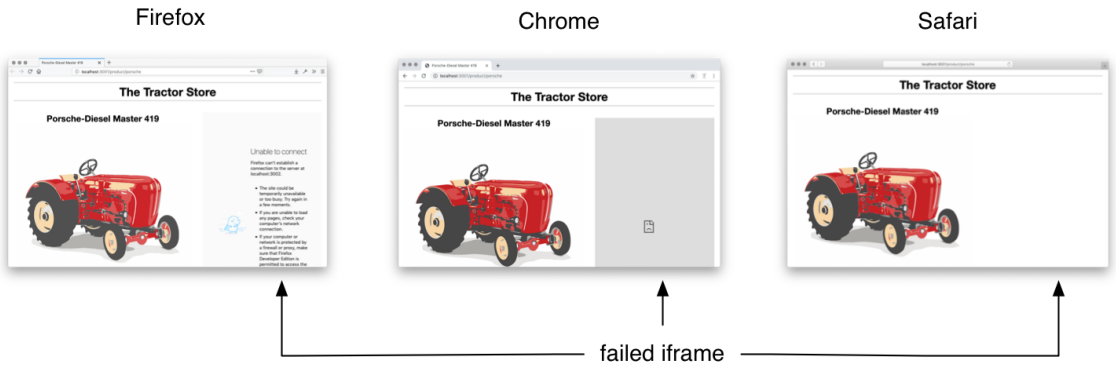

Figure 2.7 Product page with broken recommendations in different browsers. The outer page is still functional.

The presentation of this "iframe-failed-to-load" state varies from browser to browser. Safari shows blank, Chrome has a grey background with an icon and Firefox and Edge display a longer explanation text.

Technically this solution is fault tolerant and robust. The outer page is still functional and serves its purpose. But from a user's perspective the illusion of using one coherent site is broken. There is no simple way to replace the generic browser message with a more meaningful one or at least remove the broken iframe from the layout.

2.3.5 The drawbacks

While iframes provide great isolation and are easy to implement they also have a lot of negative properties which has led to the iframes bad reputation in web development.

LAYOUT CONSTRAINTS

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

PERFORMANCE OVERHEAD

Heavy use of iframes is bad for performance. Adding an iframe to a page is a costly operation from a browsers 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 your pages content in a semantic way is not only a hygienic factor. It enables assistive technologies like screen readers to analyze the pages content 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 seamlessly blend in with the rest of the page. But tools like screen readers have a hard time conceptualizing whats going on. They see multiple documents which 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. Isolation, the biggest benefit in a micro frontends context, is the biggest drawback from a search engines perspective. A crawler would index our product page as two distinct pages. The outer page and the included inner page. The fact that one includes the other is not represented in the search index. Our page would not show up for the search term "tractor recommendations". Because even though the user sees both words in her browser window, these words do not exist in the same document.

2.3.6 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 and issue this was an acceptable tradeoff for them.

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

2.4 Summary

In this chapter you've learned how to build a basic micro frontends setup where each team delivers one part of the application in form of an independent page.

- Teams should be able to develop, test and deploy independently. To achieve this it's important 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 introduces extra coupling.
- Iframes provide a strong isolation between the teams. No shared code conventions or name-spacing for CSS or JavaScript is required.
- Iframes are suboptimal for performance, accessibility and search engine compatibility.

3 Deeper Integration with AJAX & Routing

This chapter covers

- Integrating fragments into a page via AJAX
- 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 Integrating via AJAX

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

But Waldemar, responsible for online 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. In an online forum he read something about link juice which is flowing in the wrong direction and suboptimal cross product linking. He talks to the development teams to discuss options to improve the ranking.

The developers have never heard the term link juice before and doubt that this is a real thing. But they are aware that the iframe integration has issues. Especially when it comes to semantic structure. Since good search engine ranking is important 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.

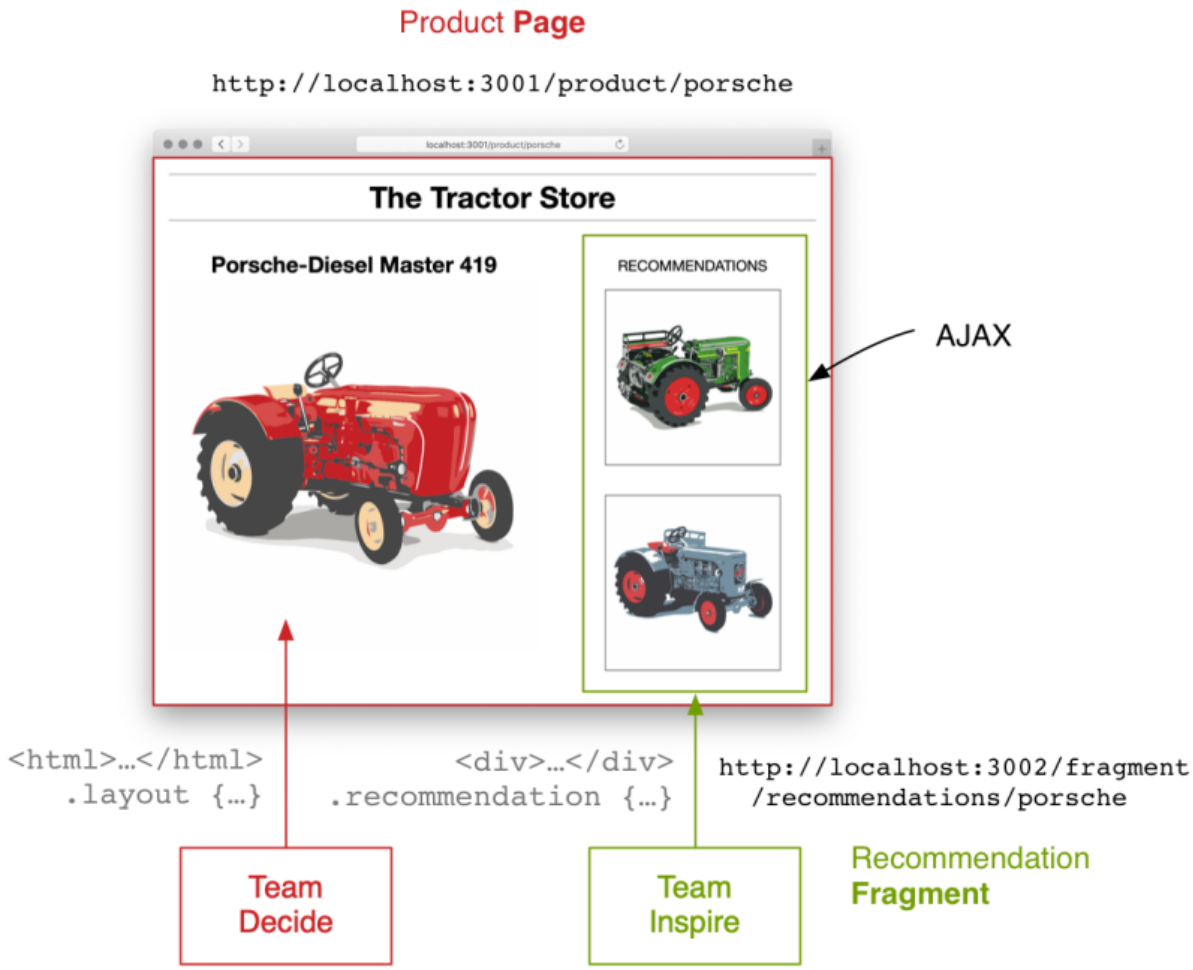

Figure 3.1 Integrating the recommendations into the product page's DOM via AJAX.

Two tasks have to be completed 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>`. This way 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 straight forward for *Team Inspire*. All data and styles are already there from the iframe integration. The have to remove the enclosing `html` and `body` from the template. They'll also have to create a new CSS file which contains only the styles needed for the fragment.

This is the folder structure which represents their applications output:

```
team-inspire/
  *fragments/
    recommendations/
      eicher.html
      fendt.html
      porsche.html*
  recommendations/
    eicher.html
    fendt.html
    porsche.html
  static/
    page.css
    *fragment.css*
```

MARKUP

The HTML under `/fragment/recommendations/<sku>` looks like this:

Listing 3.1 team-inspire/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 are included into the markup

Note that the associated CSS file is referenced from the fragments markup. The URL has to be absolute (`http://localhost:3002/…`), because this markup will be inserted into the DOM of *Team Decide* which is served from port `3001`.

STYLES

Now we have to create the `static/fragment.css` that goes with this fragment. Lets quickly revisit the contents of *Team Inspire's* `page.css`. It is already grouped into three sections:

- *Global Styles:* resets styles, definition of root font-size and font-family
- *Page Styles:* styles for the pages layout and header
- *Content Styles:* styles for the actual content

It's not a good idea to have multiple reset styles and root font definitions on one page. They affect the complete page and contradictory statements will cause strange issues. Since the

fragment does not include the header and outer layout, we can skip the *page styles* as well. Let's copy the `page.css` file to `fragment.css` and remove the first two sections so that it only contains the *content styles*.

Listing 3.2 team-inspire/static/fragment.css

```
h2 {...}
.recommendations {...}
```

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 by creating a file called `static/page.js`.

Listing 3.3 team-decide/static/page.js

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

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

❶ finding the element where the fragment should be inserted

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 `.recos` element. The product page markup now looks like this:

Listing 3.4 team-decide/view.js

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

❶ *Team Inspire's* recommendation fragment URL

❷ referencing the JavaScript file which will do the AJAX request

We are nearly done, but browser security stands in our way.

CORS

In development the teams applications currently both run on `localhost`. But since they use different ports `:3001` and `:3002` the browser considers them as different origins and enforces stricter security rules. That's why we need to allow CORS (cross-origin resource sharing) for the fragment endpoint. You can achieve this by setting the `--cors` flag when starting *Team Inspire's* server via `mfserve`.

```
npx mfserve --listen 3002 *--cors* team-inspire
```

This will set the `Access-Control-Allow-Origin: *` header to all of the servers responses and thereby allowing cross-domain AJAX requests. We'll eliminate the need for this header by adding a shared web-server in front of our applications later in this chapter.

But our AJAX integration is now functional. Opening localhost:3001/product/porsche shows you the product page with the recommendations loaded via AJAX. Let's look a little bit deeper into the micro frontends specific properties of the AJAX integration.

3.1.2 Not self-contained

An important paradigm for micro frontends is that teams must be able to develop and test their part of the frontend independently. With the link or iframe approach this was easy. You can open up the URL of the fragment, resize your browser window to fit the expected width of the slot the fragment should fit into. The result of this standalone view will be exactly the same as the integrated view.

With the AJAX integration this is different. Opening the fragment URL localhost:3002/fragment/recommendations/porsche will present you with the standalone fragment. It looks quite similar but isn't the same as the integrated view. Look at the "RECOMMENDATIONS" headline in figure 3.2.

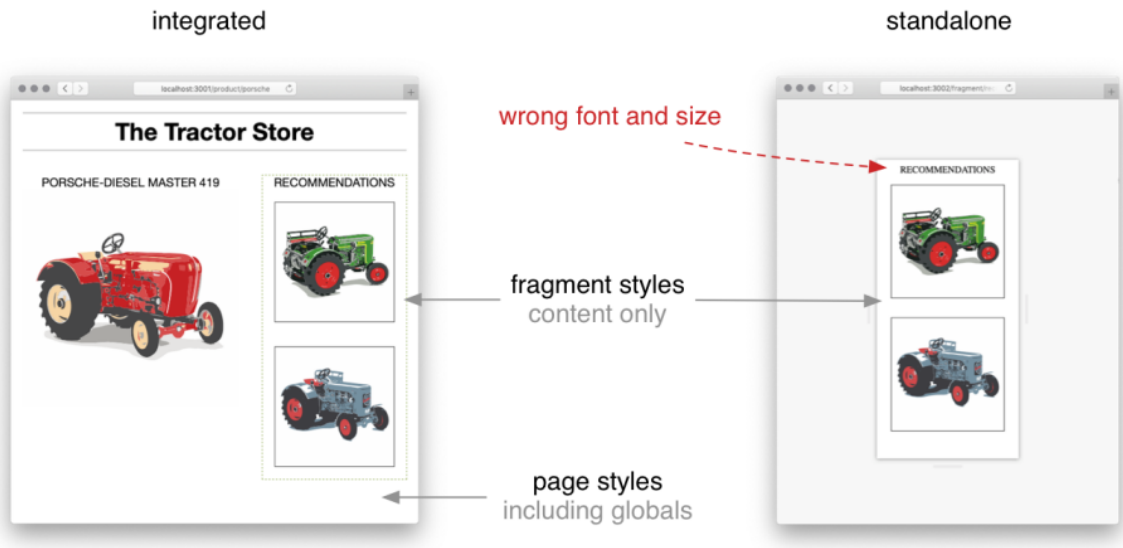

Figure 3.2 Recommendation fragment in integrated and standalone view. It brings its own styles but is not fully self-contained because it needs the global styles from the outer page.

The fragment relies on the CSS resets and global styles of the page that includes it. Otherwise it's not displayed correctly. The styling rules for `font-family` and `font-size` are missing and we see the browsers default font. To get a realistic preview for development and automated testing it's a good idea to set up a mocked integration page. For *Team Inspire* this could be a blank page which contains the global styles and has the AJAX loading mechanism implemented.

SHARED GLOBAL STYLES

To reduce friction and avoid errors it's important that all teams agree on a same basic set of global styles. This introduces coupling between the teams. Changing something in the global styles may have an affect on the included fragments. So all changes there should be well considered and the other teams need to be informed upfront.

In practice this coupling is not a huge issue when you keep global styles to a minimum. They typically change rarely.

Here are the global styles for our example project:

```
/* GLOBAL STYLES */

* {
  box-sizing: border-box;
}

html {
  font-family: 'Helvetica Neue', Arial, sans-serif;
  font-size: 20px;
}
```

```
a {
  color: inherit;
}
```

Granted, this is a very minimal set of styles. In a real project it would be more. But you should make a conscious decision on how much global styles you need. Accepting redundancy in favor of autonomy and thereby increasing development speed is one of the micro frontends mantras.

When you are using an existing design system like Bootstrap or SemanticUI via one global CSS file the coupling between the teams gets quite tight because you share a lot of styles. Updating this design system becomes a complicated task. All teams will need to do the adjustments to their code and then deploy their changes all together at the same time. In *chapter 9* we'll go deeper into this topic and discuss alternative options and their tradeoffs.

STYLE COLLISIONS

When we look a little bit closer and compare the iframe and the AJAX version of the product page we notice another difference. The name of the tractor "Porsche-Diesel Master 419" is not formatted correctly in the AJAX version. In figure 3.3 you see the correct styles on the left and the wrong styles on the right side.

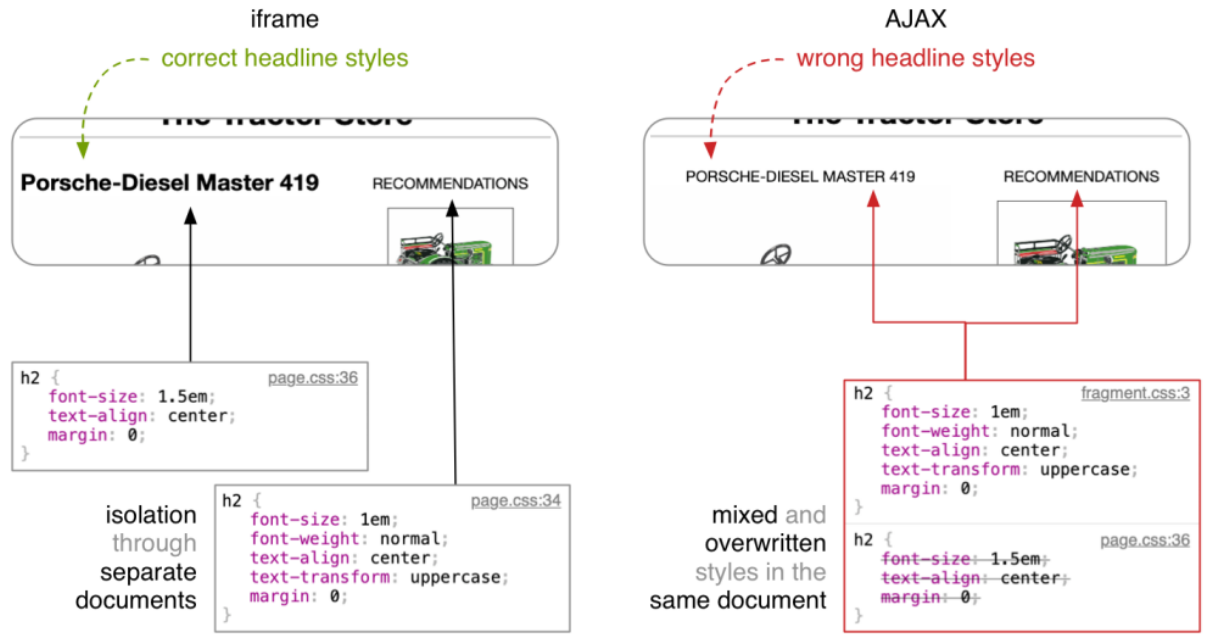

Figure 3.3 Collision of CSS styles from different teams. Styling selectors need to be unique do avoid issues.

Opening up the browser developer tools quickly reveals the reason. Both elements are styled by the css rule `h2 { … }`. The styles for the product name are different from the recommendation styles. In the iframe context these style definitions did not conflict, because they exist in different documents. With the deeper AJAX integration these styles mix and overwrite each other. Because *Team Inspire's* `static/fragment.css` is included after *Team Decide's*

`/static/page.css` stylesheet it takes precedence. Luckily this is the only problematic css selector in our example. We can fix this issue by making the scope of the selectors more specific or use a unique class-name instead of a tag-name selector.

3.1.3 Scoping styles and scripts

To avoid these kinds of collisions you need to have a shared concept for dealing with styles. Browsers don't offer much native support for developers to isolate styles. A CSS selector you write can have effect on the complete page. These kinds of errors are hard to detect, especially when the project grows.

NO NATIVE STYLE SCOPING

The *Scoped CSS* specification would have been a great fit for our use-case. It allowed you to mark a `style`- or `link`-tag with the attribute `scoped`. The effect was that all styles defined in this CSS would only be applied to the DOM-subtree the styles are referenced 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 a later chapter.

ISOLATING STYLES

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. Thats 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	
Inspire	inspire	
Checkout	checkout	

> **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 classname based selectors the issue of overwriting styles should be solved.

Prefixing does not have to be done manually. There are tools that can help here. CSS-Modules, PostCSS or SASS are a good start. Most CSS-in-JS solutions can also be configured to add a prefix to each classname.

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 they are scoped into an anonymous function. Most build tools already do this automatically.

For the `static/page.js` of *Team Decide* it would look like this:

Listing 3.5 team-decide/static/page.js

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

❶ Immediately invoked function expression

❷ Variable is not added to the global scope

But sometimes you need a global variable. A common example is, that you want to ship structured data in 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-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. This is the case for cookies, storage, events or unavoidable global variables You can use the same prefixing rules we've introduced for css classnames 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 a large 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.

3.1.4 Progressive enhancement

An important thing you have to consider when using an AJAX based integration is that it relies on JavaScript. When JavaScript fails, your integration fails. The web is a wild place. There are many reasons why this could happen:

- a few people have JavaScript disabled
- AdBlockers use heuristics to block the loading of suspicious files
- your JavaScript might be faulty
- an error occurs in a browser version you haven't tested

- some third-party script might introduce an error that also affects your code
- your JavaScript has not been loaded yet because of poor network conditions

It's considered best practice to model your web application by the principals of *Progressive Enhancement*. You build the features with the native building blocks of the web: links, forms, images, … JavaScript is only used to improve the experience. E.g. showing the recommendations inline and not on a separate page. In our example we already did that. When JavaScript fails (disable JavaScript to test that), we don't get the integrated recommendations. Instead we see the "Show Recommendations" link that brings us to the separate page. The integrated experience is clearly better, but the page model works - always.

Progressive Enhancement is like building a safety net. You can think of it as Technical Credit.
– Jeremy Keith

If you want to learn more about this, Jeremy Keith has published a lot of material on this topic at resilientwebdesign.com/.

3.1.5 Declarative loading with h-include

The loading of the fragment content via AJAX in our example is triggered by *Team Decides* `static/page.js`. The code is specific for our current use case. It looks for an element with the class `.recos` and fetches the content from an URL that's attached to an attribute. This works, but there is potential for generalizing this code to make inclusion of more fragments easier.

The JavaScript library *h-include* provides a good abstraction for this[24]. It's a declarative approach. Including a fragment feels like including an iframe in the markup. You don't have to care about finding the DOM element and doing the actual HTTP request. The code for the recommendations would look like this:

Listing 3.6 team-decide/view.js

```
...
<aside class="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 a extra features like defining timeouts, reducing reflows by bundling the insertion of multiple fragments together and lazy loading.

3.1.6 The benefits

The AJAX integration is a technique which 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, all content is integrated into one DOM. Fragments are now part of the pages document flow. This means, that they take exactly the space in the layout they need. 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. No JavaScript based height calculations or workarounds are needed.

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 completely or display a static alternative content you've prepared for this case.

SEARCH ENGINES AND ACCESSIBILITY

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

PROGRESSIVE ENHANCEMENT

As said before, it's typically pretty straight forward to implement progressive enhancement with the AJAX model. The markup is generated on the server and delivering it as a fragment or as a standalone page does not introduce a lot of extra code.

3.1.7 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 its loading. This is because the fragment is fetched asynchronously via JavaScript. 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 delay the page rendering and 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 integrations.

MISSING ISOLATION

The AJAX model does not come with any isolation. To avoid conflicts teams have to agree on inter-team conventions for namespacing. This is 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. This is fine for many applications, but sometimes you have the 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 readded 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[26], annotation via `data-*` attributes, custom elements or custom events can help here. But you have to implement these mechanisms manually and share them across teams.

3.1.8 When does an AJAX integration make sense?

Integration via AJAX is straight forward. It's **robust and easy** to implement. It comes with the benefit, that all content is located in the same document. It also comes 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.

3.1.9 Summary

Lets revisit the three integration techniques we've touched so far. Figure 3.4 shows how the links, iframe and AJAX approach compare to each other from a developers and users perspective.

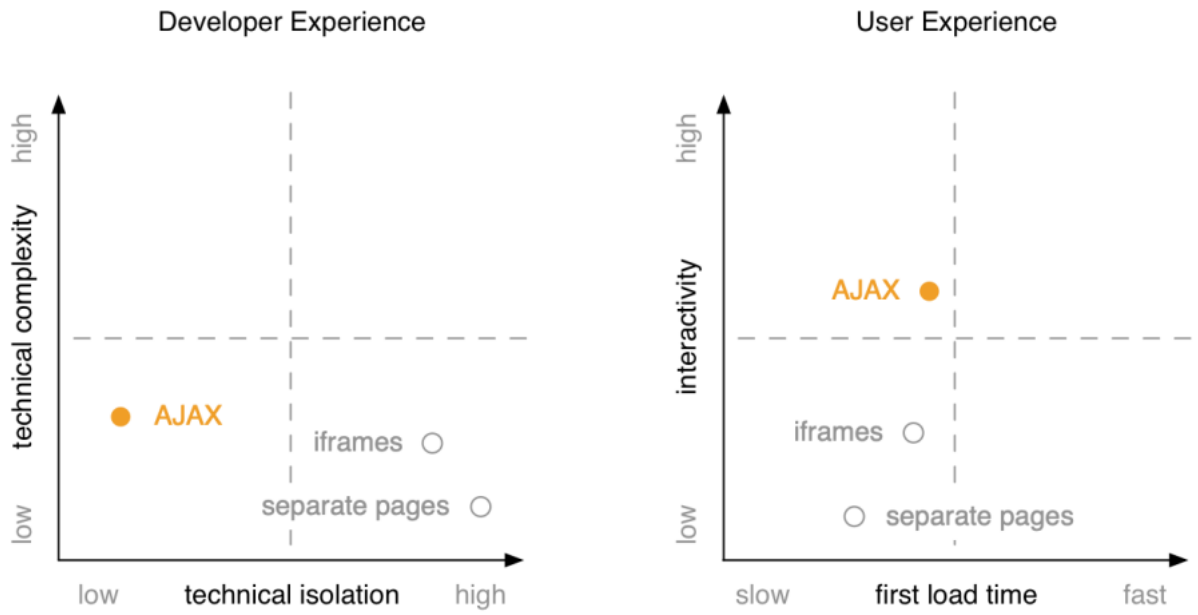

Figure 3.4 Comparison of different integration techniques. Compared to the iframes or links approach its possible to build more performant and usable solutions with AJAX. But you loose technical isolation and need to rely on inter-team conventions like using css-prefixes.

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 quickly 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 Routing via a shared web-server

The switch from iframe to AJAX had measurable positive effects. Search engine ranking improved and we received mails 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, so their pages were hosted at `tractor-inspirations.firebaseapp.com/`. The domain switch in the browsers address bar while navigating the page confused some users.

Ferdinand, the CEO of Tractor Models Inc., took this request seriously. He decided, that all of the companies web properties should be accessible from one domain. After long 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. They were happy with this assignment. This would also solve the cross-origin issues they encountered in the last iteration.

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 3.5 illustrates this concept.

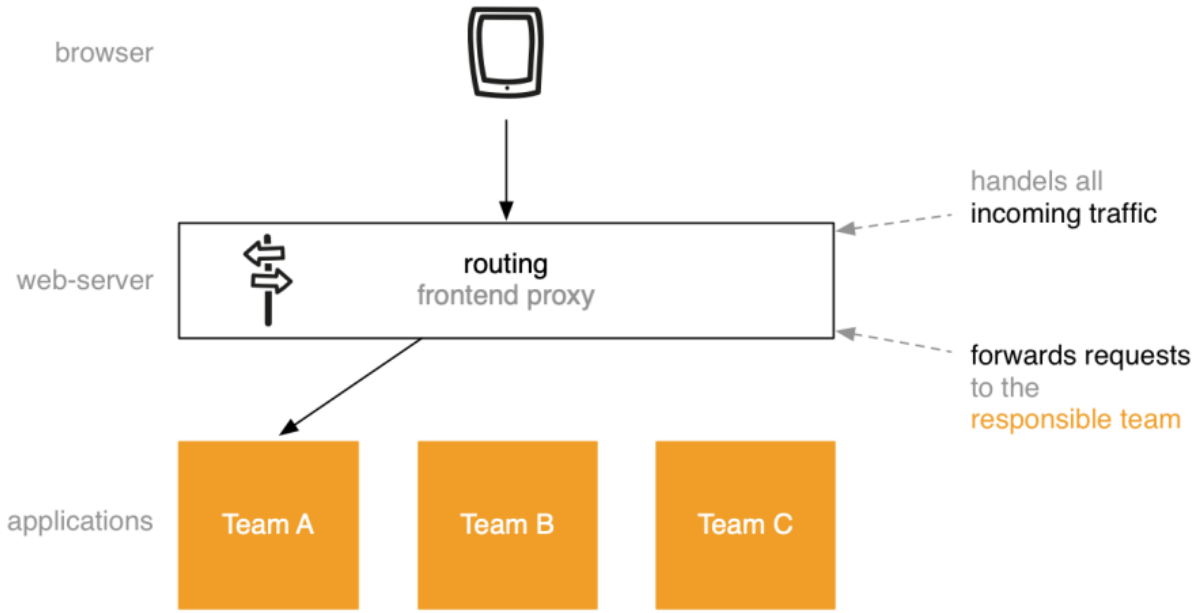

Figure 3.5 The shared web-server is inserted between the browser and the team applications. It acts as a proxy and forwards the requests to the responsible teams.

From the web-server the requests must be routed to the application of the responsible team. The routing decision is made based on the incoming path. Request starting with `/product/` go to

Team Decide, the ones with `/recommendations/` are passed to *Team Inspire*. After checking the applications again, they discovered that this is not sufficient. Both applications have a static route `/static/` for their CSS and JavaScript files. *Team Inspire* delivers their fragments under `/fragment`, To avoid collisions these URLs need to be changed. They chose to go with team prefixes here. *Team Decide* adds `/decide/` and *Team Inspire* adds `/inspire/` in front of these endpoints.

In our development environment we again use different port numbers instead of configuring actual domain names. The shared web-server we will setup listens on port `3000`. This is what the routing table of the web-server needs to look like:

Table 3.3 Web-server routes incoming requests to the teams applications

Requests starting with ___	are forward to	Team ___	Application
`/product/`		Decide	`localhost:3001`
`/decide/`		Decide	`localhost:3001`
`/recommendations/`		Inspire	`localhost:3002`
`/inspire/`		Inspire	`localhost:3002`

Figure 3.6 illustrates how an incoming network request is processed. Let's follow the numbered steps:

1. customer enters the-tractor.store/product/porsche and hits enter. The request is send to the shared web server
2. web server matches the path `/product/porsche` against his routing table. Rule 1 `/product/` is a match.
3. web server passes the request to *Team Decides* application.
4. application generates a response and passes it back to the web server.
5. web server passes the response to the client

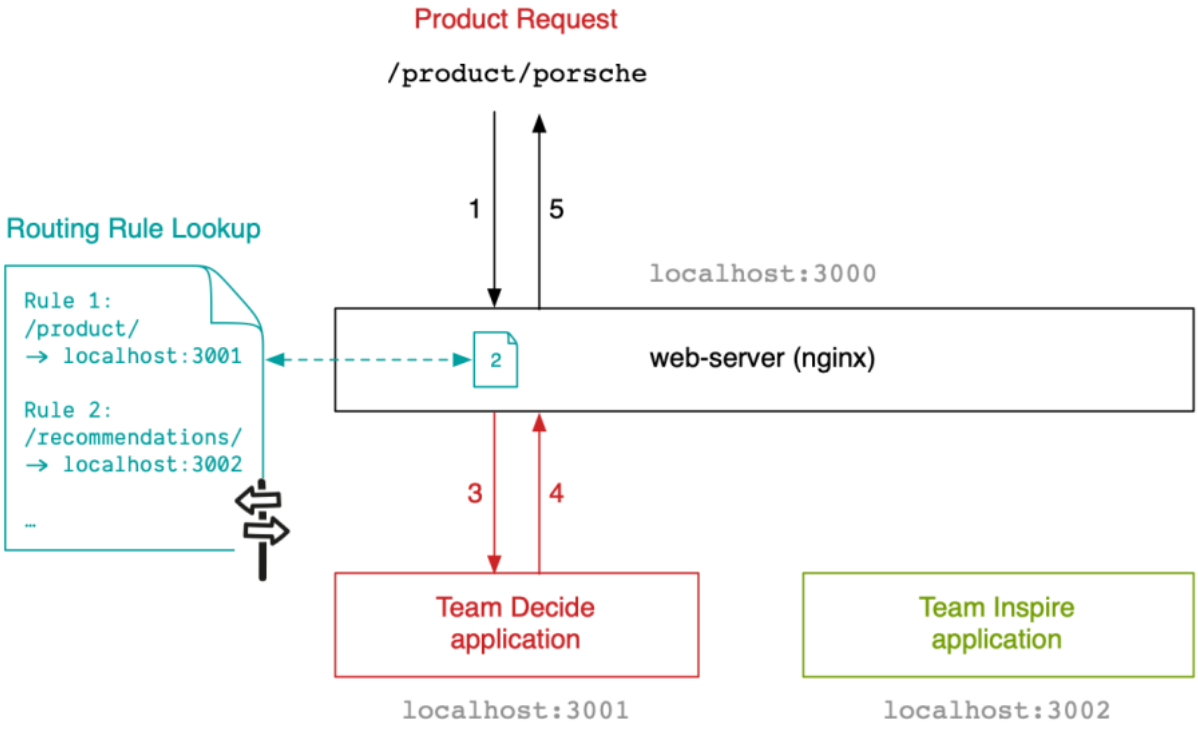

Figure 3.6 Flow of a request. The web-server decides which application should handle an incoming request. The decision is based on the URL-path and the configured routing rules.

Now we have everything we need to implement this. Let's get started.

3.2.1 How to do it

TIP	You can find the sample code for this task in the `5_routing` folder.

The teams picked *nginx* as their web server of choice. It's a popular, easy-to-use and pretty fast solution.

Don't worry if you haven't worked with nginx before. We'll explain be basic concepts necessary to make our routing work. If you want to run the example-code locally you need to install nginx on your machine. You can find an installation instruction in Appendix A Setting up nginx for local development.

ROUTE CONFIGURATION

You'll need to understand two nginx concepts for this:

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

In nginx you can specify different *location*-blocks for a server. A location-block has a matching

rule which gets compared against every incoming request. When the rule matches, the body of the block `{..}` describes how the request is processed.

Here the "hello world" example from our `nginx.conf`.

Listing 3.7 webserver/nginx.conf

```
location / {
  return 200 "hello world!";
}
```

This rule matches all requests where the *path* starts with `/` (matcher) and returns status code 200 with the text `hello world!`.

> **NOTE** It's possible to use more advanced regex-matchers. But we will stick to simple prefix-matching for our examples,

For our use-case we don't want nginx to answer the request directly. Instead the request should be forwarded to another server. To do this you have to register all servers you want to forward traffic to as an *upstream*. Each upstream as a unique name. With the directive `proxy_pass` you can tell nginx to forward a request to a registered upstream.

Look at the following example. It forwards all incoming request starting with `/product/` to *Team Decides* application running on `localhost3001`.

Listing 3.8 webserver/nginx.conf

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

server {
  ...
  location /product/ {                ❷
    proxy_pass  http://team_decide;   ❸
  }
}
```

❶ creates a new upstream definition named `team_decide` for *Team Decide*

❷ handles all request starting with `/product/`

❸ passes the requests to the `team_decide` upstream

These are the basics you need to know to use nginx as a proxy server. Let's expand this configuration by specifying all forwarding rules we've identified in table 3.4 before.

Table 3.4 Web-server routes incoming requests to the teams applications

Requests starting with ___	are forward to	Team ___	Application
`/product/`		Decide	`localhost:3001`
`/decide/`		Decide	`localhost:3001`
`/recommendations/`		Inspire	`localhost:3002`
`/inspire/`		Inspire	`localhost:3002`

Listing 3.9 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/ {                    ❷
      proxy_pass  http://team_decide;      ❷
    }                                      ❷
    location /recommendations {
      proxy_pass  http://team_inspire;
    }
    location /inspire/ {
      proxy_pass  http://team_inspire;
    }*
}
```

❶ registers *Team Decides* application as an upstream called *team_decide*

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

NOTE In our example we use a local setup. The upstream is served from `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 Request routing

Let's test this configuration. Stop your nginx if it's still running. Start the server and both teams applications in separate terminals.

```
nginx -c "`pwd`/webserver/nginx.conf"
npx mfserve --listen 3001 team-decide
```

```
npx mfserve --listen 3002 team-inspire
```

When we open localhost:3000/product/porsche our request now goes to the nginx (port `3000`), the fact that our path starts with `/product/` tells nginx to forward it to *Team Decide* (port `3001`). You should see the red Porsche-Diesel Master.

NAMESPACING RESOURCES

Now that both applications run under the same domain it's important that their URL structure doesn'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.

```
team-decide/
  product/
  *decide/*
    static/
team-inspire/
  recommendations/
  *inspire/*
    fragment/
      recommendations/
    static/
```

To make this work the internal references to the CSS and JS files are adjusted. But also the URL-patterns both teams agreed upon need to be updated. With the central web-server in place a team does not have to know the domain of the other teams application any more. 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 web-server the domain can be removed 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>`

Find these references in the HTML files and also update them.

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

Voila - reloading the browser should show us our beloved product page as we knew it. It looks the same as in our AJAX version before but under the hood all requests are delivered from `localhost:3000` - the nginx.

Introducing URL-namespaces is an important 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>`. But it also helps with debugging in the browser. Looking at the path of a CSS file that's causing an issue reveals which team owns it. This easy attribution if resources is especially helpful when your application gets more complex and the number of teams working on it grows.

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 for dealing with this. 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 a new team is added to the project. The configuration looks like this.

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

For URLs the customer does not see like APIs, assets or fragments the prefixing is not an issue. 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 (decide, inspire, …) which a search engine bot would read and add to their index.

Choosing 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 web-servers routing.

```
/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 grow quickly. 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 their routing solution called Skipper[27]. They claim that it has been built to handle more then 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. They must be considered in every decision that is made. Teams should have all the power and tools they need to accomplish their job as best as possible. To do this redundancy is accepted 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. Therefor you should keep central components like this to a minimum. Only introduce them, when there is no reasonable alternative.

To ensure that these central components run stable and get the attention they need its important to have a clear ownership. In classical software projects it would be run by a dedicated platform team. The goal of this team is to provide and maintain these shared services. But in practice these teams create a lot of friction and are horizontal in nature.

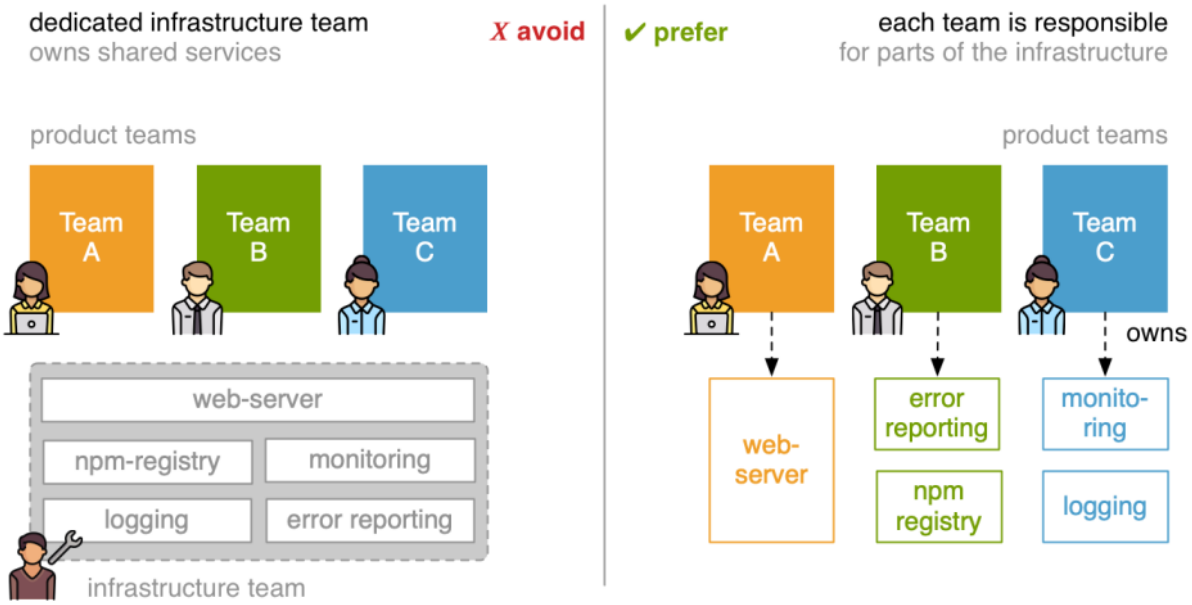

Figure 3.7 Avoid introducing pure infrastructure teams. Distributing responsibility for shared services to the product teams is a good alternative model.

A more light weight alternative is to assign the responsibility for a central service like this to one of the vertical teams. This keeps development focused on creating value for the customer and reduces inter-team communication. In our example *Team Decide* could take responsibility for running and maintaining the nginx.

3.2.5 When does it make sense?

Delivering the contents of multiple teams through a single domain is pretty common. 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 which 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. Integration can be done via links and cross-origin AJAX requests.

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.
- CSS collisions can be avoided by introducing team namespaces for CSS-classes.
- Progressive enhancement is a valuable philosophy to build robust applications.
- You can route the content of multiple applications through one web-server which serves all content through a unified domain.
- Using team prefixes in the URL path is a good 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.

With links, iframes and AJAX we've covered the most basic integration techniques. In the next chapter we'll look at how to integrate the markup on the server side. With nginx we already have the infrastructure available for this.

4 Server Side Integrations

This chapter covers

- Examining fragment integration via Server Side Includes (SSI)
- Identifying the performance impact of server-side integration
- Investigating how timeouts and fallbacks can help dealing with faulty or slow fragments
- Exploring alternative integration solutions like Tailor, Podium, and ESI
- Determining when a server-side integration technique makes sense for your project

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 this 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.

Integration is typically done by a service that sits between the browser and the actual application servers as illustrated in figure 4.1.

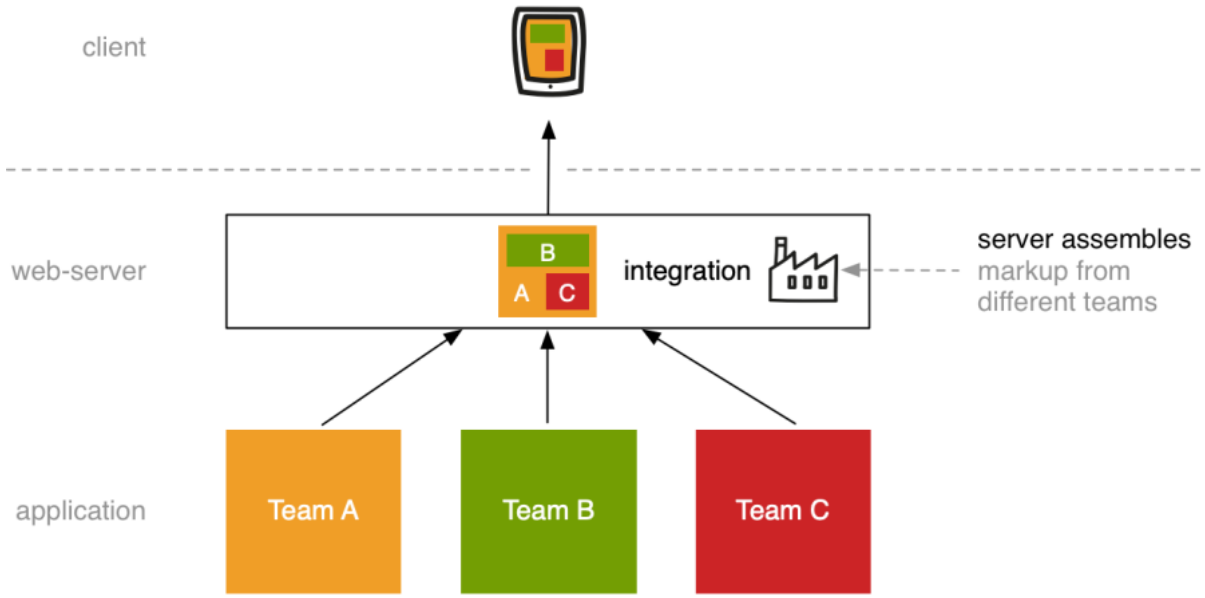

Figure 4.1 Integration of fragments happens on the server. The client receives an already assembled page.

A key benefit is that integration has already happened before the page reaches your customer's browser. You can achieve an incredibly good first-page load speeds that can't be matched with pure client-side integration techniques.

Another key factor is robustness. Server-side integration does not rely on client-side JavaScript at all. This means it's much easier to develop by the principals of progressive enhancement. Teams can decide to add JavaScript to the fragments where it improves the user experience.

4.1 Integration via Server Side Includes (SSI)

Last iteration the teams switched their integration from iFrames to an AJAX-based solution. This 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 pop in with a considerable 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 town 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 certain 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 the 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**

The server-side includes mechanism dates back to the 1990s. Beside including other resources it brings features like defining variables and running conditionals. It's a turing-complete scripting language, but writing programs in SSI was never a popular thing. Back in the days, it was often used to embed the current date into an otherwise static page or embed the contents of one file into another one. Webservers like Apache, IIS or LiteSpeed also come with SSI implementations. In this book, we will focus on SSI's `include` directive in the nginx server.

The specification is pretty old and has not evolved over the last years. But this is a good thing. 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 already deliver everything that's needed to make server-side integration work. We can reuse the recommendations fragment endpoint from our AJAX integration.

Team Decide needs to make two changes

1. Activate nginx' SSI support in the web-servers configuration
2. Adding 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 is structured like this:

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

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

Figure 4.2 shows how the HTML for the product page is generated 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.

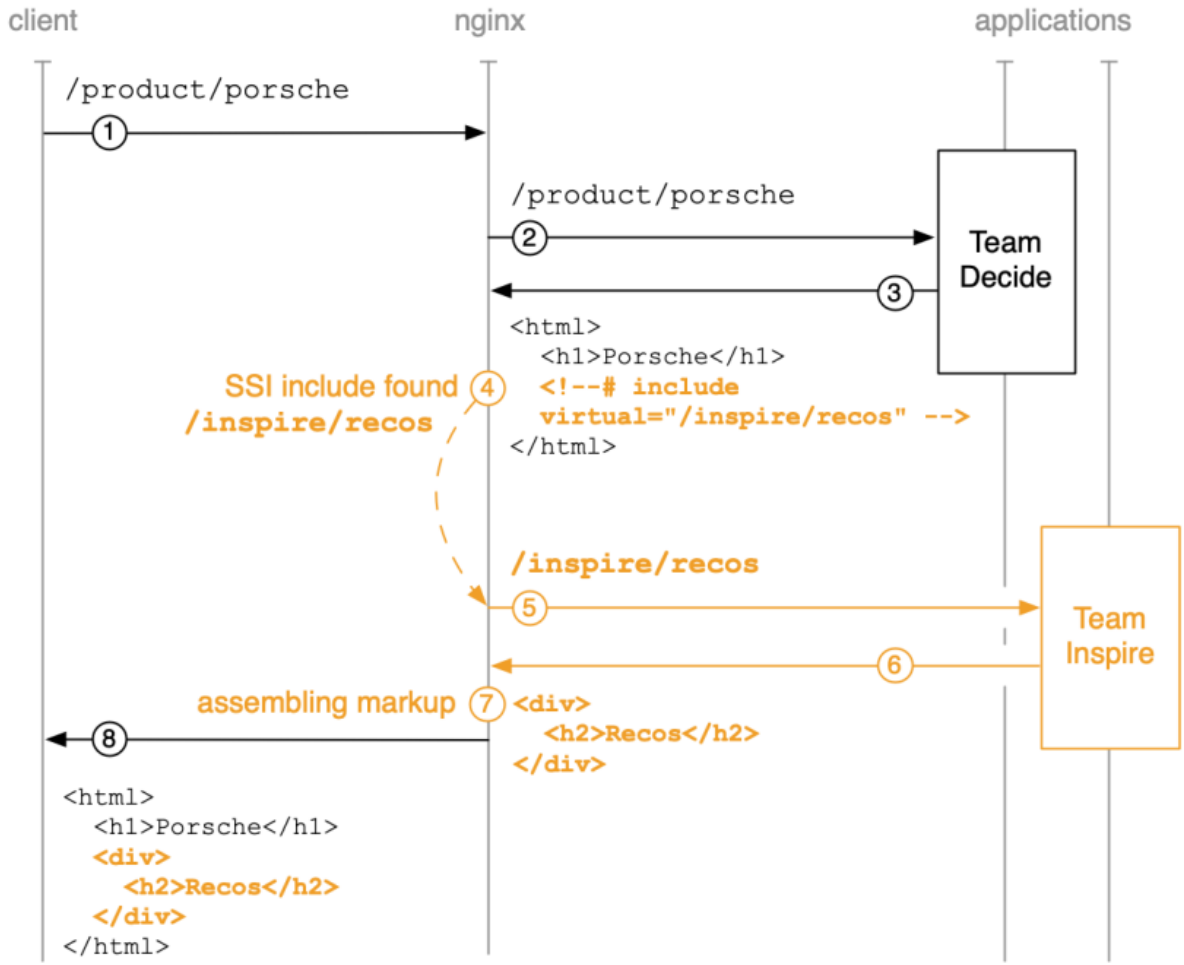

Figure 4.2 SSI processing inside nginx

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 sends 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

> **TIP** You can find the sample code for this task in the `6_ssi` folder.

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" -->*
    </aside>
</body>
...
```

Voila - we are done. Opening localhost:3000/product/porsche now shows you the tractor page as we know it. But we don't need client-side JavaScript for the integration anymore. The markup is already integrated when it reaches the browser.

4.1.2 Better load times

Let's open up the browser developer tools and see how our changes affected the page load speed. We activate the same 3G network throttling from before. Figure 4.3 shows the result.

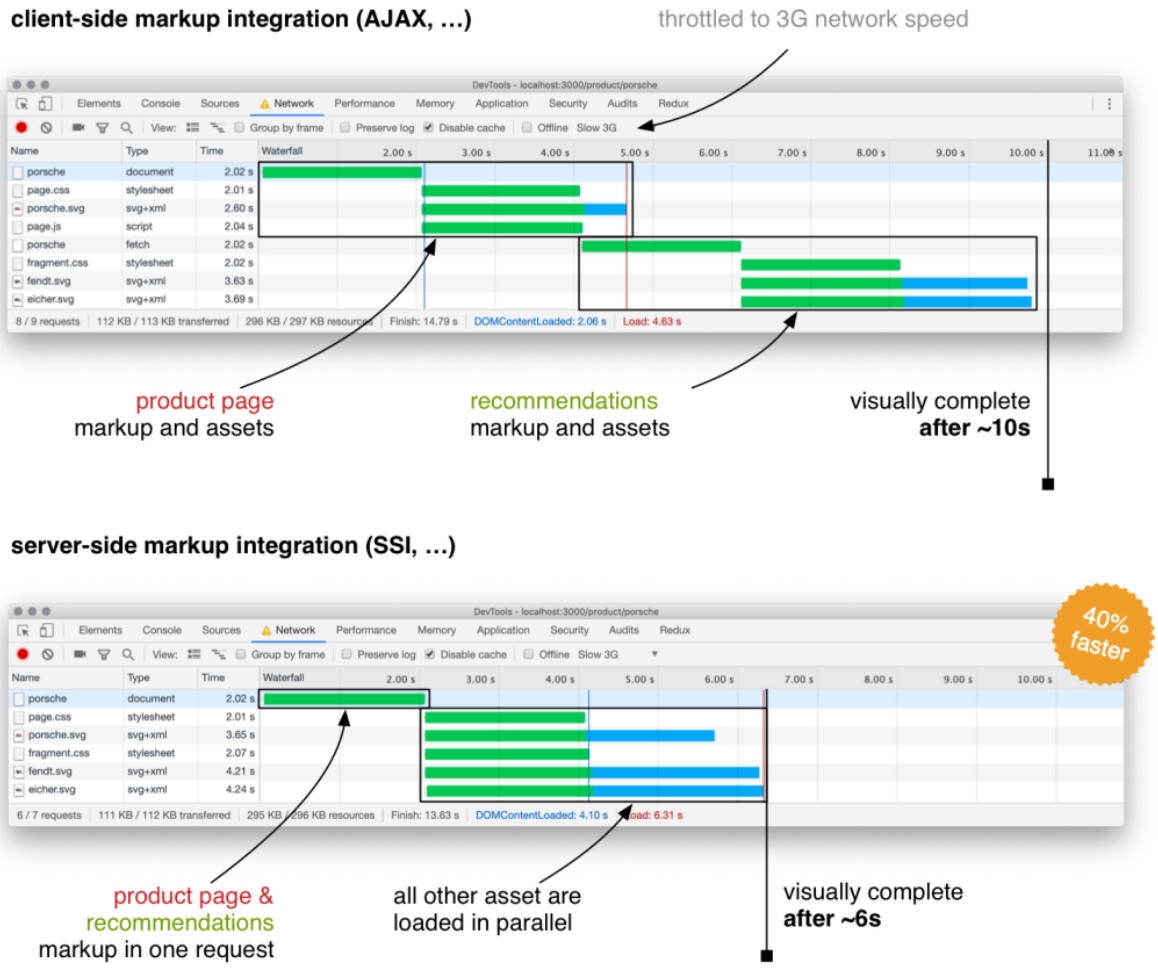

Figure 4.3 Page-load speed for the product page with client-side and server-side integration. Server-side integration optimizes the critical paths and is there by considerably faster.

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 deeper 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 now already bundled into the initial markup. This saves us two seconds. The other factor is that the loading of the AJAX call was triggered by JavaScript. The browser had to wait for the JavaScript file to finish before he was able to load the fragment. This 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 important to give the browser the information about all essential parts of the page like images and styles as early as possible. Server-side integration is essential in making this happen.

The key 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 [28] 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 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 a little information box should be displayed on the product page. The first version of this feature will be limited to Europe and Russia and localizes the users by his IP-address. This 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 slot in the product pages layout. They agree, that the near you fragment should be displayed as a long banner underneath the header on the product page. When no nearby tractor is found the fragment is not shown at all. 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.

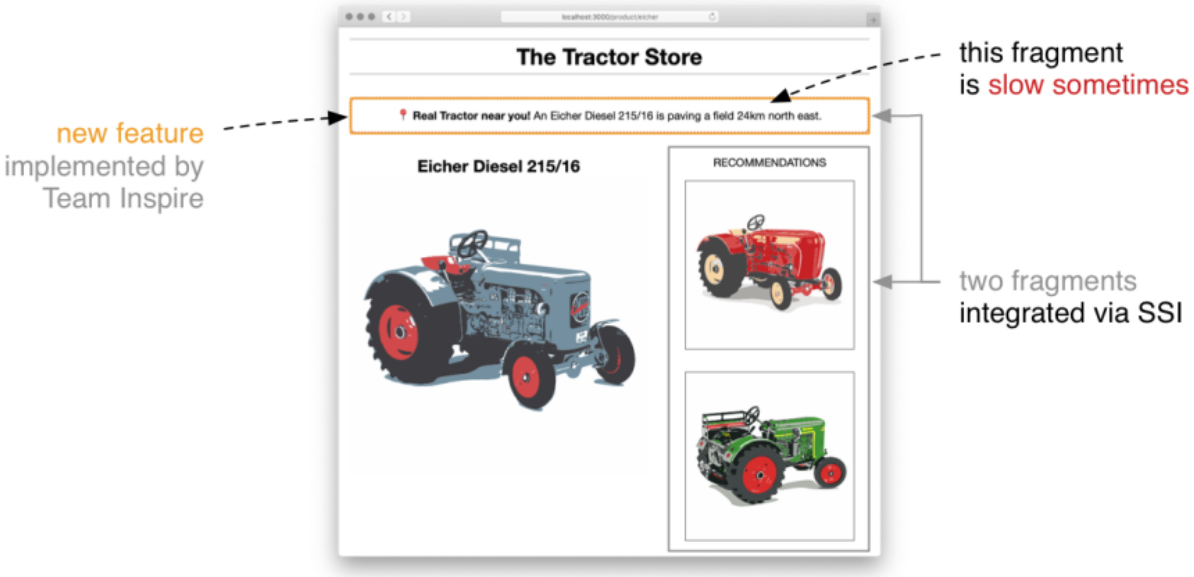

Figure 4.4 The near you feature is added as a banner on top of the page.

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 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 `7_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" />     ❶
<div class="inspire_near_you">                                    ❷
   <strong>Real Tractor near you!</strong>                        ❷
  An Eicher Diesel 215/16 is paving                               ❹
  a field 24km north east.                                        ❷
</div>                                                            ❷
```

❶ fragment stylesheet

❷ fragment content

Note that the "Near You" fragment references the same CSS file (`fragment.css`) as the recommendation fragment. The required styles have been added to this file. We'll address asset loading in more depth later in this book.

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>*
<div class="decide_product">
...
```

Opening up your browser on localhost:3000/product/eicher shows the product page of the blue Eicher Diesel including the new fragment at the top. But since we are serving static HTML files the response time for the complete page is pretty fast. The markup is ready in a few milliseconds.

Let's change this and introduce an artificial delay for the fragments. The `mfserve` command has a `--delay` option which we can use to slow down response times. Stop *Team Inspire's* server and restart it with a 1000 millisecond delay. The command then looks like this:

```
npx mfserve --listen 3002 --delay 1000 team-inspire
```

Now the page takes considerably longer to load. Since nginx waits until all fragments have arrived more than one second goes by until the markup is transferred to the browser.

In contrast to the AJAX integration, where fragments are fetched asynchronously, one single fragment can slow down the complete page in a server-side integration. This means **the slowest fragment defines the total response time**. To achieve good performance its important for all teams to monitor the response times of their fragments.

4.2.3 Timeouts

But 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 good as possible. An error in one system should not break the others. Nginx comes with basic mechanisms to define timeouts for upstreams. So when an upstream becomes slow or doesn't respond at all nginx stops waiting for its fragments and delivers the site without the includes.

Let's have a look at how nginx behaves when the application doesn't respond at all. Testing this is easy. We can stop *Team Inspire's* application and reload the product page. Now both

fragments are missing and we get two nasty "502 Bad Gateway" messages in their place. We'll deal with those later. On the plus side, the page loads pretty quick. 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`. For our example, we could set the `proxy_read_timeout` to `500ms` for all requests starting with `/inspire/`. 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.

One thing 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.[29]

Let's test our configured timeout. We'll need to restart our nginx and start *Team Inspire's* application with the `--delay 1000` option. The first load of the product page takes around 500ms. This is our configured time nginx is willing to wait. After that, the page is sent to the browser with the "Bad Gateway" message as seen before. Subsequent requests to the product detail page are answered instantly (< 10ms) because nginx doesn't even try to contact *Team Inspire's* application for at least 10s.

NOTE It's not possible to configure a timeout in nginx that only aborts sporadic long-running requests. When some requests take to long the complete upstream is marked 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

We don't want to show the "Bad Gateways" error message to the customer. Luckily 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. The content of the block is used when something goes wrong with the include. This means that everything inside the `block` and `endblock` comment is the fallback content.

> **NOTE** It does not matter where the block is placed in the document but it has to be defined before it is referenced via the `stub`.

In this example, we'll show the "Show Recommendations" link instead of the integrated fragment in case of an error or slow response.

Listing 4.6 team-decide/product/eicher.html

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

❶ Defining the fallback content

❷ Assigning the 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"* -->                                ❷
</div>
...
```

❶ Empty fallback content

❷ Assigning the block as a fallback

> **NOTE** Even though the SSI commands look like HTML comments nginx does not parse HTML. It does a raw string-based search and replace in the markup. With `ssi_types` it's even possible to enable SSI for other mime types beside `text/html`. You could use it to concatenate JavaScript or CSS files.

After reloading localhost:3000/product/eicher the "Bad Gateway" messages are gone. The "near your" banner is left blank and the recommendations are replaced by the fallback "Show Recommendations" link.

Restarting *Team Inspires* application without the artificial delay makes both fragments appear again.

4.3 Tuning for performance

In the examples before we've seen that one fragment can slow down the complete page. Let's dig a little bit deeper to understand the performance implications of server-side integration techniques.

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.

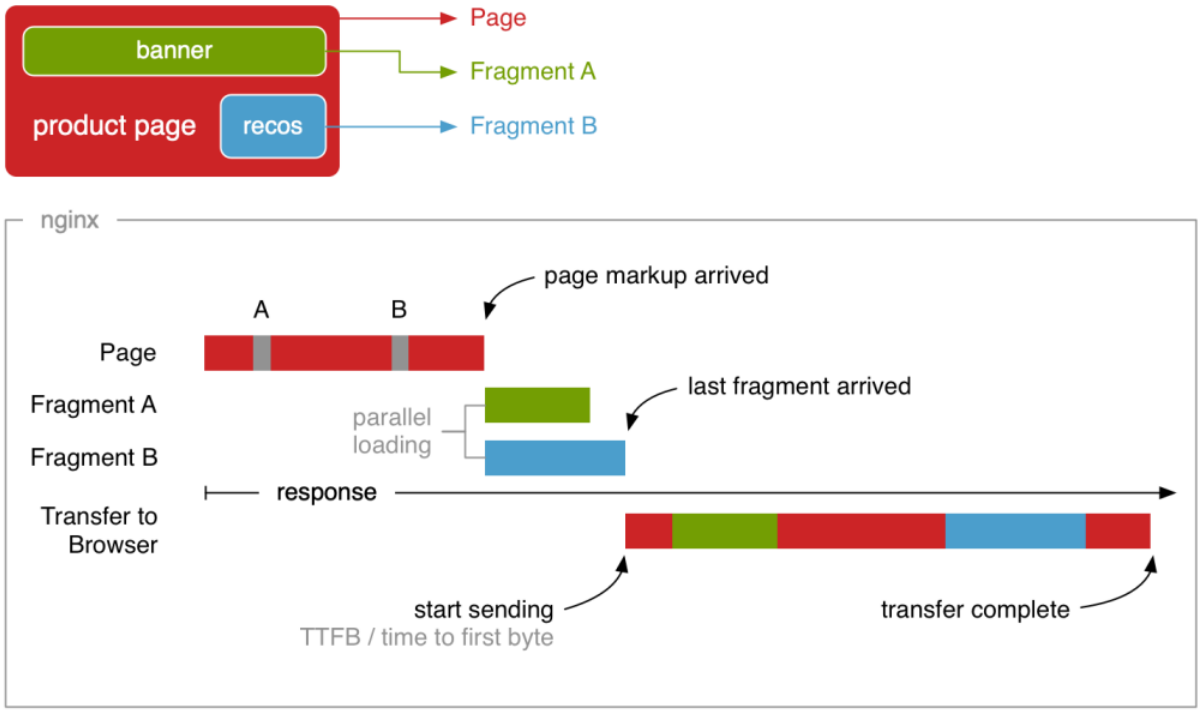

Figure 4.5 Nginx fetches multiple SSI includes in parallel

After nginx receives the HTML for the product page it parses the content and finds two SSI directives (A & B) that need to be resolved. 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 includes 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. The header fragment is included by many pages. 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.

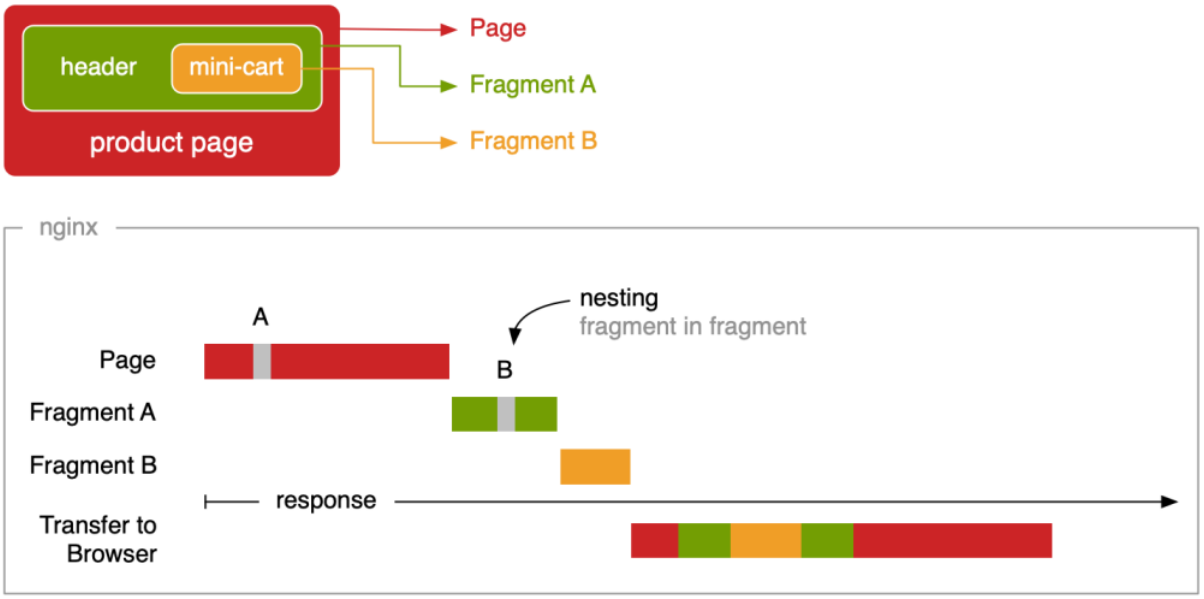

Figure 4.6 Product page includes the header fragment which itself includes the mini-cart fragment

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.

> **NOTE** It's not possible to crash your web-server by accidentally building an infinite loop. After a nesting depth of 50, nginx stops resolving more SSI includes.

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 most important parts of your page. This is usually everything in the upper part of the page (viewport). Additional fragments that are farther down the page or are optional for your site to work (newsletter signup, promotions, …) can be lazy-loaded. This reduces the size of the initial markup the client needs to load and enables the browser to start rendering the page earlier.

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

If a fragment should be included in the initial markup you specify it as an SSI directive.

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

Or defer the loading of its content by fetching it using an AJAX call via client-side JavaScript.

```
window
  .fetch("/fragment-a")
  .then(res => res.text())
  .then(html => { target.innerHTML = html; });
```

4.3.4 Time to first byte & streaming

As you've seen before nginx waits for all fragments before it starts sending data to the client. In theory, it 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 SSI include. 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. But unfortunately nginx doesn't work this way. It only starts the **sending after assembly** is finished.

The idea of **streaming** templates takes this one step further. With this model, the markup from the upstreams is generated and sent as a stream. This means, that 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 integration server can directly pass this data to the client and start fetching fragments even if the pages markup from the other upstream is not fully generated. The two steps (loading page, loading fragments) overlap which reduces overall load time and improves time to first byte significantly.

Figure 4.7 Different ways 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 bytes. This way the browser receives the content earlier and can start rendering quicker.

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

- This diagram does not factor in the actual bandwidth limitations of the browser.
- The streaming model includes the assumption that the response generation is a linear process. This is only true 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.

Remember when integrating on the server it's important that all fragments get produced quickly. One slow fragment can impact the overall response time. Implementing timeouts can help to mitigate this effect. You learned how to implement fallback content for a fragment in case a fragment fails or an upstream does down. We also examined nginx's SSI handling in more depth.

How parallel and nested fragment loading works and what performance implications come with this behavior. You can compare nginx's fragment aggregation mechanism with the concepts of partial or streaming assembly when it comes to TTFB and absolute download 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 have a look at a few alternatives. We'll focus on their main benefits.

4.4.1 Edge Side Includes

Edge Side Includes, short ESI, is a specification[30] that defines a unified way for markup assembly. It's most often implemented by content delivery network providers like Akamai and supported in proxy servers like Varnish, Squid, and Mongrel. 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://example.org/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://example.org/fragment"
  alt="https://fallback.org/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.[31] There you can add a `maxwait` attribute. When the fragment takes longer it will be skipped.

```
<esi:include
  src="https://example.org/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 not all fragments are resolved yet.

4.4.2 Zalando Tailor

Zalando[32] moved from a monolith to a micro frontends style architecture with *Project Mosaic*.[33] They published parts of their server-side integration infrastructure. *Tailor* [34] is a node.js library that parses the pages HTML for special `fragment`-tags fetch the referenced content and puts it into the pages 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 that can be installed via npm.

```
npm install node-tailor
```

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[35] for other options.

❷ Attaching tailor to a standard node.js webserver which listens on port 3001.

An associated template could look like this:

Listing 4.9 team-decide/views/product.html

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

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

This is a simplified version of our product page example. Team Decide runs the Tailor service in their node.js application. A call to localhost:3001/product will be handled by the tailor server. It uses the `./views/product.html` template to generate a response. Tailor replaces the

`<fragment … />`-tag with the HTML content that is returned from the localhost:3002/recos endpoint. This endpoint is operated by Team Inspire.

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 the this fragment

❷ fallback content is loaded 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. Data is sent to the browser as the page template (they call it layout) is parsed and fragments arrive. This 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.

```
$ 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. This 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. All JavaScript has to be wrapped in an AMD module which will be loaded by the require.js module loader. You also have no control over how script and style tags are added to the markup.

NOTE At the time of writing this book Zalando is working on a successor to Tailor called Interface Framework.[36] Its goal is to simplify the development of new fragments and enable data-driven layout decisions based on user behavior.

4.4.3 Podium

Finn.no[37] is a platform for classified ads and Norway's largest website by 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[38] 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, the URL for the actual content endpoint.

Listing 4.10 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 it's 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.

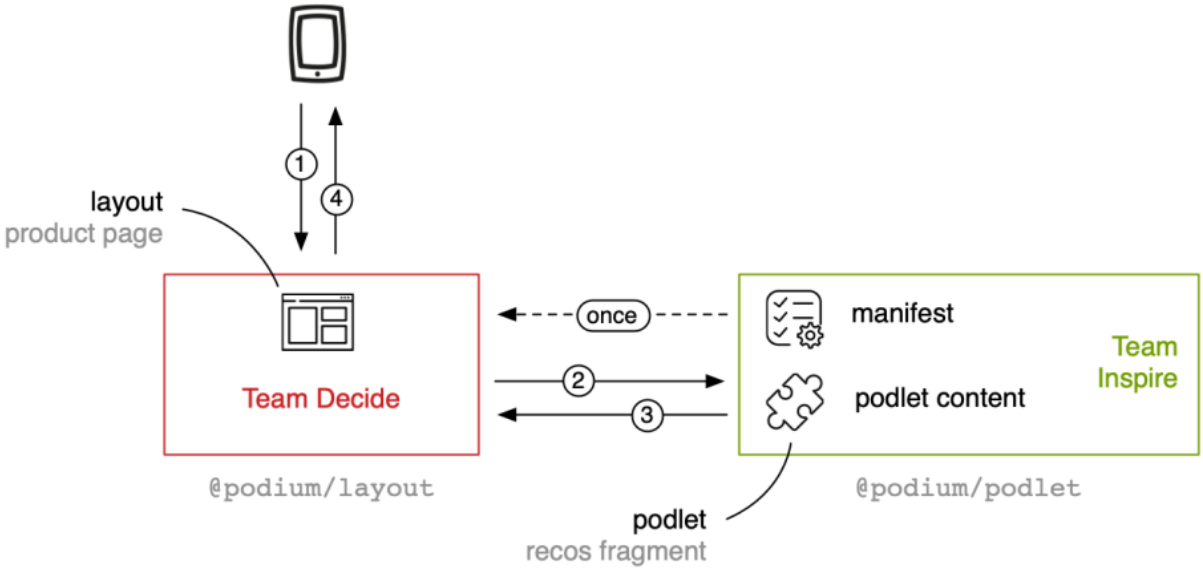

Figure 4.9 Simplified overview of Podiums architecture. The team which delivers a page (layout) communicates with the browser. It fetches fragment content (podlet) directly from the team that generates it. Associated manifest information is only requested once, not on every request.

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 it's product page. It requests the podlet's content endpoint.
3. *Team Inspire* responds with the markup for the recommendation. This is a plain HTML response 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. The assembled markup is sent 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.

> **NOTE** You can find a working version of *The Tractor Store* in the sample code `8_podium`.

Each of the teams creates it's own node.js based server. We are using the popular express[39] framework as a web server but Podium is not tied to that.

These are *Team Decide's* dependencies:

Listing 4.11 team-decide/package.json

```
...
"dependencies": {
  "@podium/layout": "^4.2.0",
  "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.send(`                                                   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*. Metadata is fetched 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. The context is passed down to the 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.

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 way to retrieve the markup of the fragment that needs to be included: `await`

`recos.fetch()`. How you place it into your layout is completely 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": "^4.2.0",
  "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.

Pretty straight forward right? Defining the podlet information, a route for the `manifest.json` and the `/recos` route that produces the actual content.

We are using express in this example but Podium is not tied to that. You can use it with the node.js plain HTTP server or other server frameworks.

> **NOTE** To make the listing more readable I've shortened the HTML code and omitted logging and asset handling code. You can get the full picture in the sample code that comes along with this book.

FALLBACKS AND TIMEOUTS

The way Podium handles fallbacks is quite interesting. With the nginx approach, the fallback had to be defined in the template that makes the include. With ESI and Tailor, you can provide a second URL that's tried when the actual URL does not work. In Podium it's a little bit different:

- the fallback is **provided by the team owning the fragment**
- the fallback is **cached** by the team which integrates it

These two properties make it much easier to create a meaningful fallback. *Team Inspire* could, for example, define a list of "evergreen recommendations" that are styled in the same way as the 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.

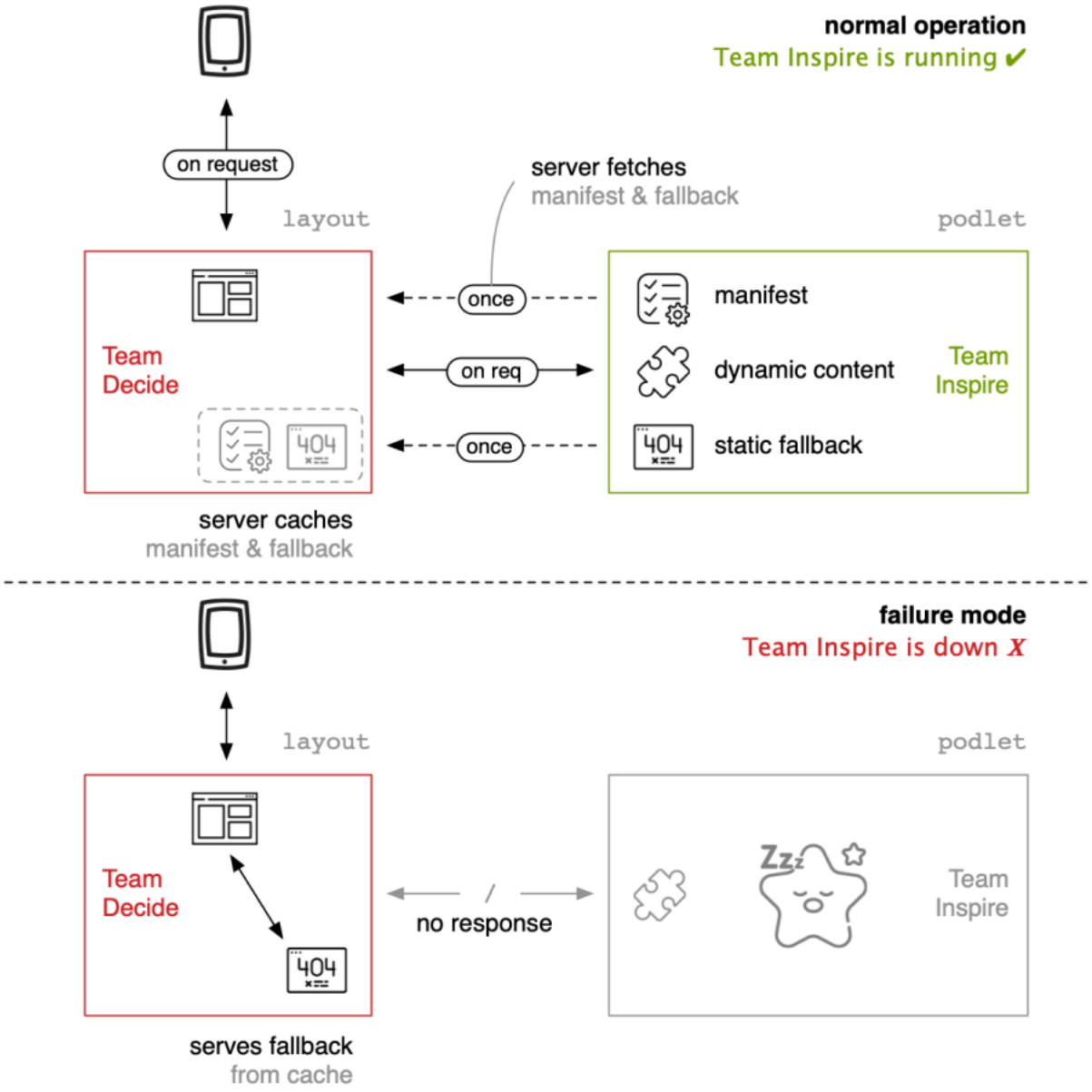

Figure 4.10 Podiums fallback handling. The fallback content for the podlet can be specified in the manifest. The layout service retrieves the fallback content once and caches it. When the podlet server goes down the fallback is used instead of the dynamic content.

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[40] is a good 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 integration technique.

Tools like Tailor and Podium implement fragments as a first-party concept which makes common needs like fallbacks, timeouts and asset handling much easier. They also eliminate the need for an extra piece of infrastructure because the integration is done from inside the node.js application. This 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.

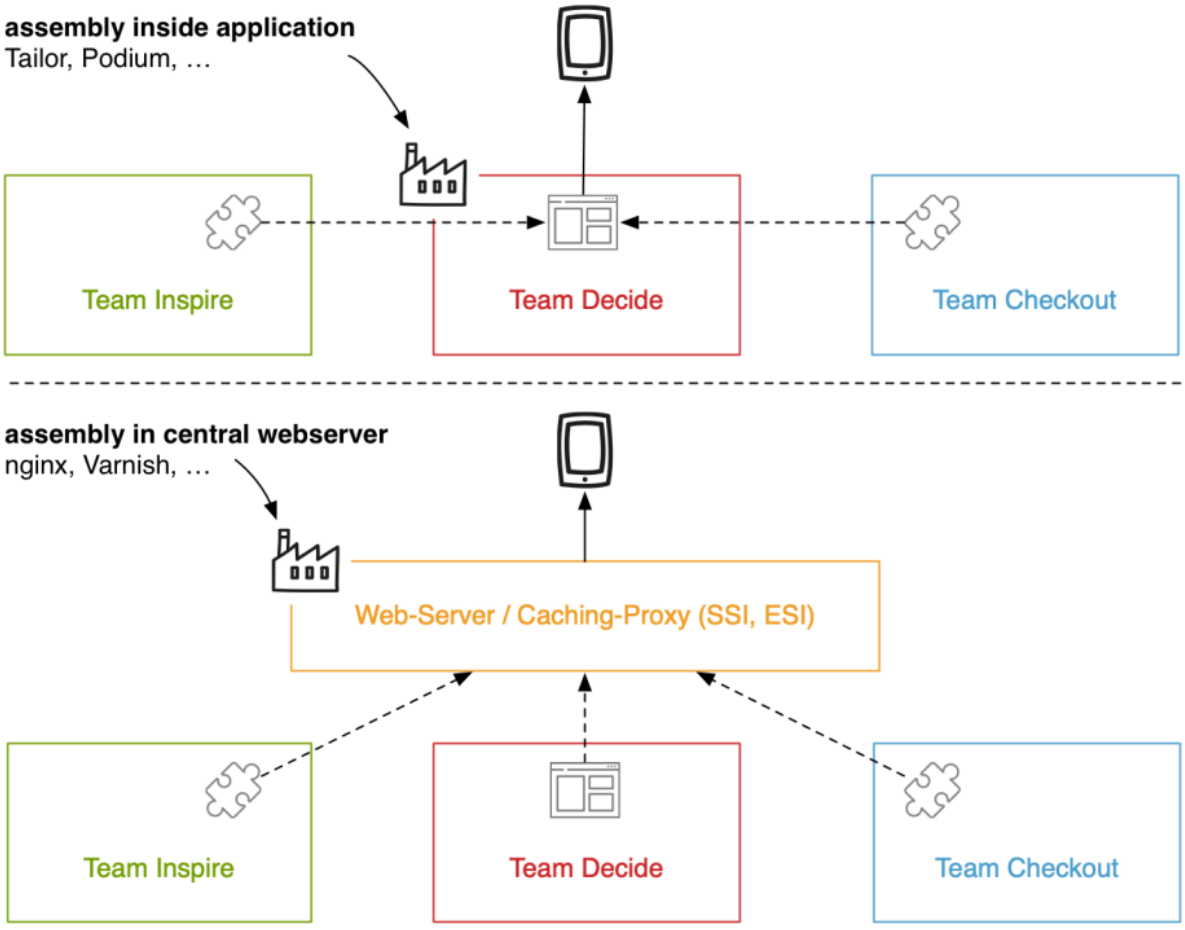

Figure 4.11 Fragment integration in the application or in a central web-server.

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.

Since this is the piece of software that all teams rely on it's important to make a well-considered decision.

4.5 The good and bad of server-side integration

Now you know the most important aspects of server-side integration. Let's look at the advantages and disadvantages of server-side integration in general.

4.5.1 The benefits

Since the assembly is already done before the site reaches the browser you can achieve **good first load performance**. 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 good 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 basic way. But having a site that loads fast and does not require a huge 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 because 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/universal way. In *chapter 7* we'll go deeper into the discussion on client- vs. server-side rendering.

Figure 3.4 shows the comparison chart introduced in the last chapter. We've added server-side integration.

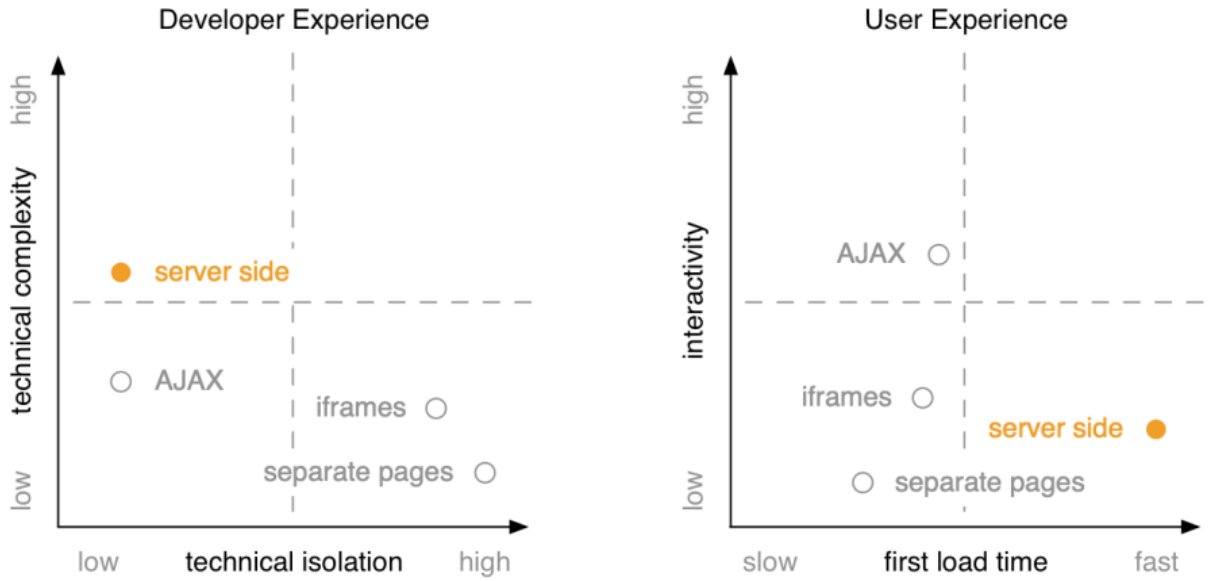

Figure 4.12 Server side integration in comparison to other integration techniques. They introduce extra infrastructure which increases complexity. Similar to the AJAX approach they don't introduce technical isolation, you still have to rely on manual namespacing. But they enable you to achieve good page 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 starts 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 easier. 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 integration techniques can be combined with client-side approaches. They are the basis for building a micro frontends style site that uses progressive enhancement principals.

5 Client Side Integrations and Communication

This chapter covers

- Examining Web Components as a client-side integration technique
- Investigating how to use micro frontends that are implemented with different frameworks on the same page
- Comparing different communication patterns between micro frontends
- 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 an indispensable for websites that need to load fast. But for many applications, 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 he 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 for 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 upgradable on their own. **We can't rely on the component system of one specific framework.** This would tie the complete architecture to the release cycle of this framework. 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. Making it possible for micro

frontends to coexist on one page, even if their technology stack is not the same. Figure 5.1 illustrates this client side integration.

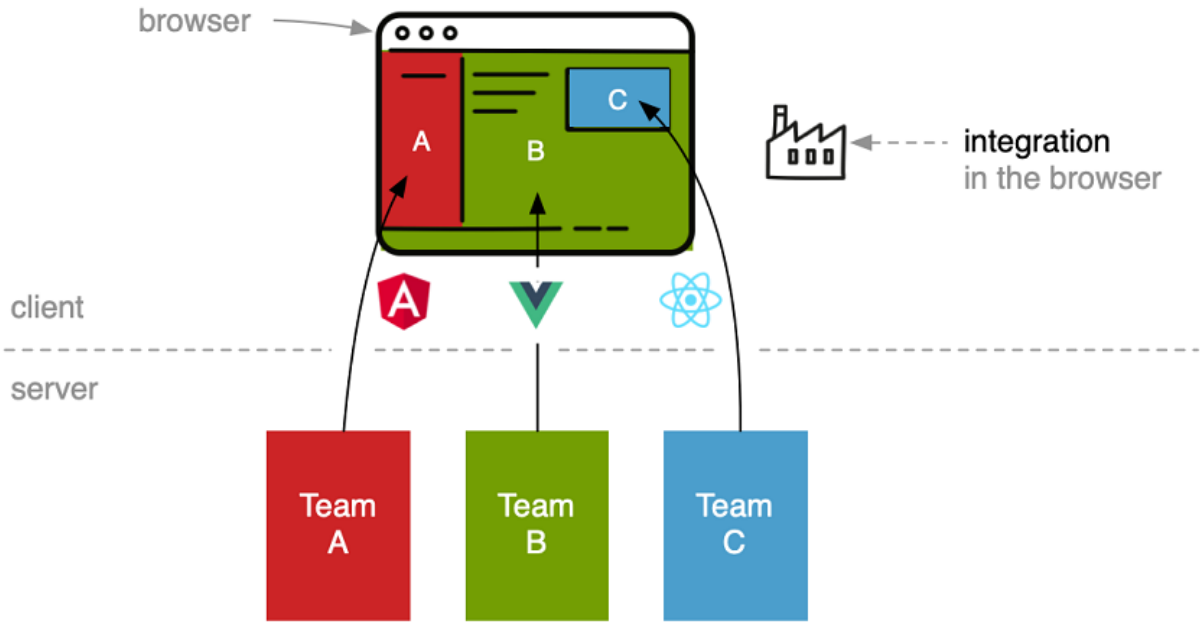

Figure 5.1 Fragments integrate in the browser. Each fragment is it's own mini-application and can render and update it's markup independently from the rest of the page.

Sometimes user interface fragments owned by different teams need to talk to each other. When a user logs into your application via a Login fragment owned by Team A, other fragments on this page likely want to be notified and update their interface accordingly. In-browser communication between the team's user interfaces is a topic we need to address. There are a lot of ways to realize this. Later in this chapter, we'll focus on communication methods that are close to web standards like the DOM-API or CustomEvents.

5.1 Wrapping Micro Frontends using Web Components

Over the last weeks *Tractor Models Inc.* has made a huge 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.

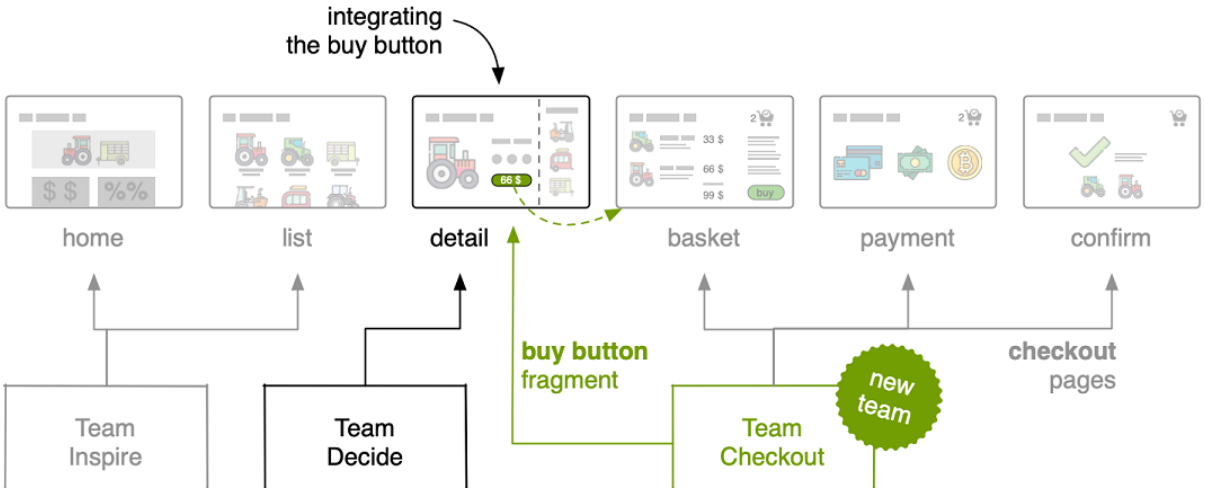

Figure 5.2 Team Checkout owns the complete checkout flow. Team Decide does not have to know about how the checkout works. But they need to integrate Team Checkouts "buy button" fragment in the detail page to make it work. Team Checkout provides this button as a standalone micro frontend.

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`

Both teams are free to make changes to layout, look or behavior of their user interfaces as long as they adhere to this contract.

5.1.1 How to do it

> **TIP** You can find the sample code for this task in the `9_web_components` folder.

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 it's 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*.

WHAT ARE WEB COMPONENTS?

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[41].

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

CUSTOM ELEMENTS

With custom elements, it's 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 normal `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 let you similarly create your abstraction. You can **construct new generic representational elements** that are missing from the HTML spec. GitHub has published a list of such controls[42]. This is an example of a "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.

ENCAPSULATING BUSINESS LOGIC THROUGH DOM ELEMENTS

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>
```

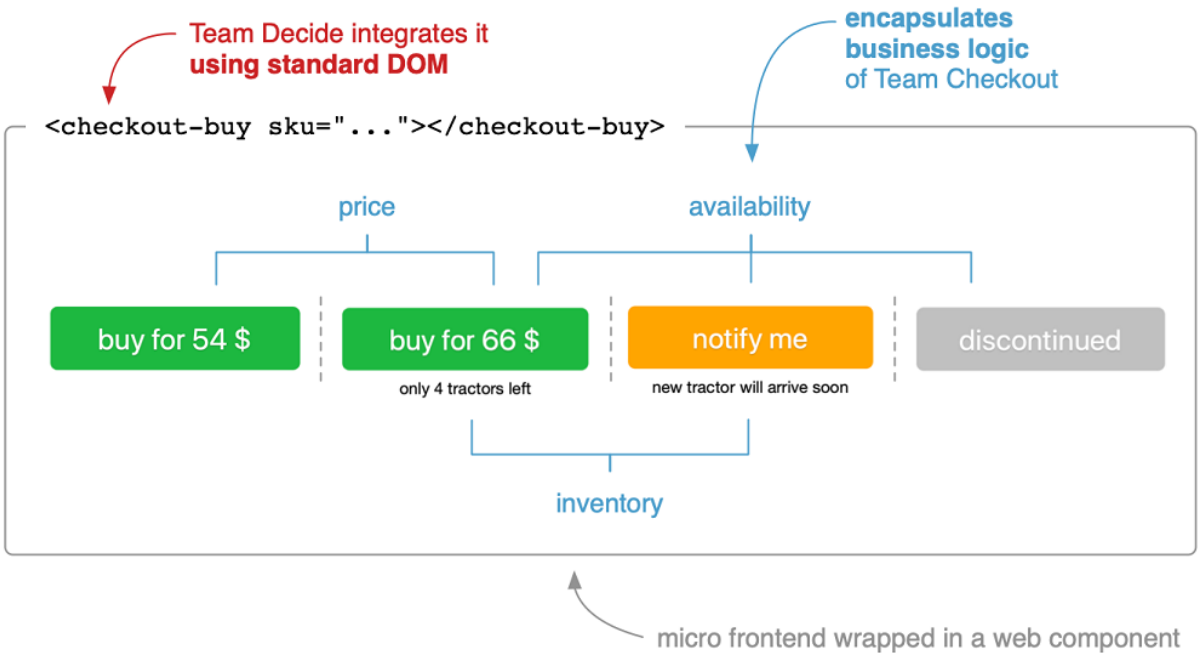

Figure 5.3 A custom element can encapsulate business logic and provide the associated user interface. The buy button can look differently depending on the specified SKU but also due to internal pricing and inventory information. A user of this fragment does not have to know these concepts.

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
  constructor() {                                 ❷
    super();                                      ❸
    this.innerHTML = "<button>buy now</button>";  ❹
  }                                               ❷
}
window.customElements.define("checkout-buy", CheckoutBuy);   ❸
```

❶ defines an ES6 class for the custom element

❷ the constructor 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`

This is a minimal example of a custom element. The implementation of the custom element must be defined as an ES6 class. 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 new elements are added to the HTML specification.

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

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

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>`.

> **NOTE** The example code is based on the AJAX integration from chapter 3 (`4_namespaces`). This way it's easier to run and play with it because we don't need the nginx web-server.

Since the fragment is fully client rendered *Team Checkout* only needs to host two files: `fragment.css` and `fragment.js`.

```
team-checkout/
  static/
    fragment.css
    fragment.js
```

The `fragment.css` is not that interesting. It contains the styles needed for the buy button.

Listing 5.3 team-checkout/static/fragment.css

```
checkout-buy button {...}
checkout-buy button:hover {...}
checkout-buy button:active {...}
```

When you start the three applications in separate terminal windows you can see the buy button in action.

```
npx mfserve --listen 3001 team-decide
npx mfserve --listen 3002 --cors team-inspire (optional)
npx mfserve --listen 3003 team-checkout
```

Opening localhost:3001/product/porsche shows you the product page with the client-side rendered buy button like in figure 5.4.

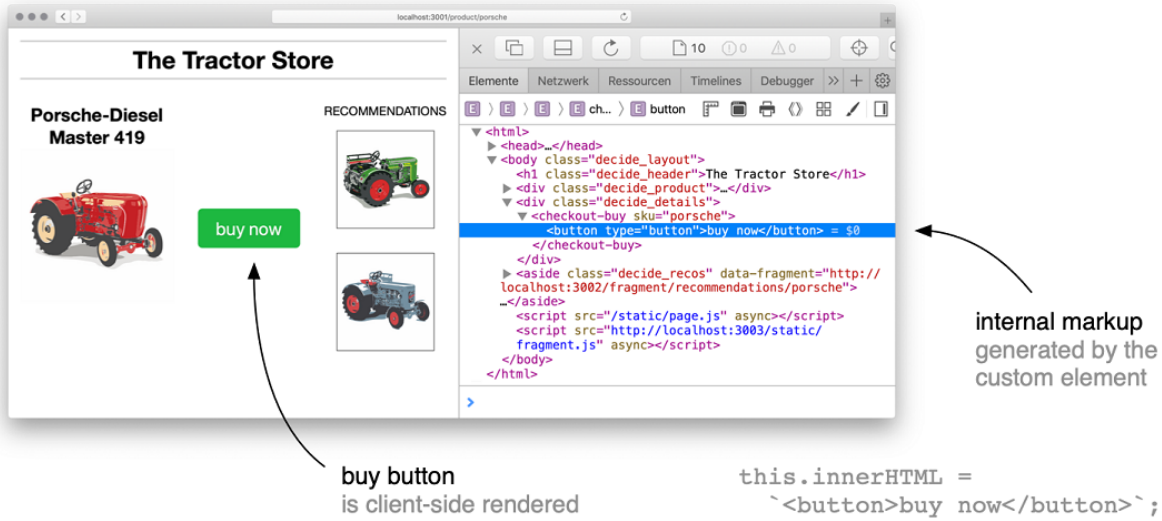

Figure 5.4 The custom element renders itself in the browser via JavaScript. It generates it's internal markup and attaches it as children to the tree via this.innerHTML = "".

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 5.4 team-checkout/static/fragment.js

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

class CheckoutBuy extends HTMLElement {
  constructor() {
    super();
    const sku = this.getAttribute("sku");                ❶
    this.innerHTML = `
      <button type="button">
        buy for ${prices[sku]} $                         ❷
      </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, the prices are defined 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 5.5 team-checkout/static/fragment.js

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

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 the APIs response a success or error message would be shown. 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 easier 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 `connectedCallback`, `disconnectedCallback` and `attributeChangedCallback`. When you implement them you get notified when your component is added to and removed from the DOM or an attribute value changed. It's straight forward to connect these lifecycle methods to the (de)initialization code of the framework or library you are working with. Figure 5.5 illustrates how to do it for a React application. The specific framework is hidden inside the component. This way its owner can change the implementation without changing how the component is used. **The custom element acts as a technology-neutral interface.**

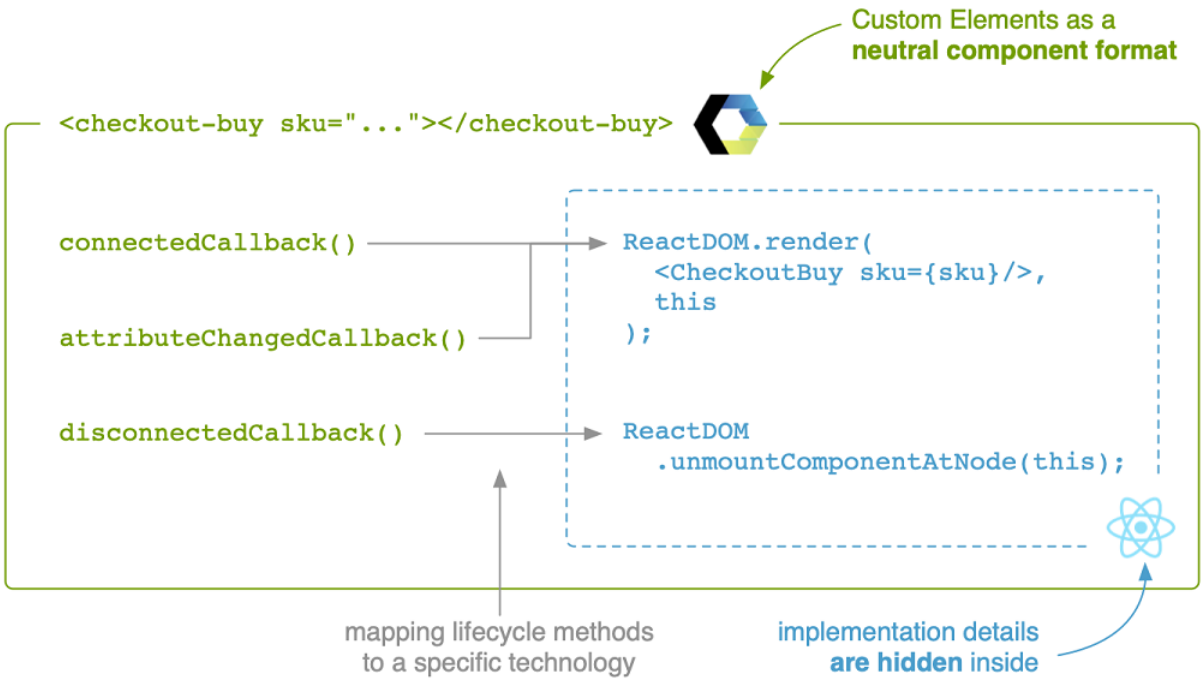

Figure 5.5 Custom Elements introduce lifecycle methods. You need to map these to the specific technology of your micro frontend.

Some newer frameworks like Stencil.js[43] already use Web Components as their primary way to export an application. Angular comes with a feature called Angular Elements[44]. You just need to add your desired custom element name as an annotation to your application. Angular will automatically generate the code necessary to connect the app with the custom elements' lifecycle methods. It also supports rendering to Shadow DOM. Vue.js provides a similar solution via the official `@vue/web-component-wrapper` package.[45] Since Web Components are a web standard there are comparable libraries or tutorials for all popular frameworks out there.

5.2 Communicating between Micro Frontends

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 10.1 Finding good 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. Our teams are formed 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. Moving from a product recommendation to the actual product page can be a plain link which has the SKU of the selected product encoded in it. **In most cases 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 5.6 illustrates three common communication patterns.

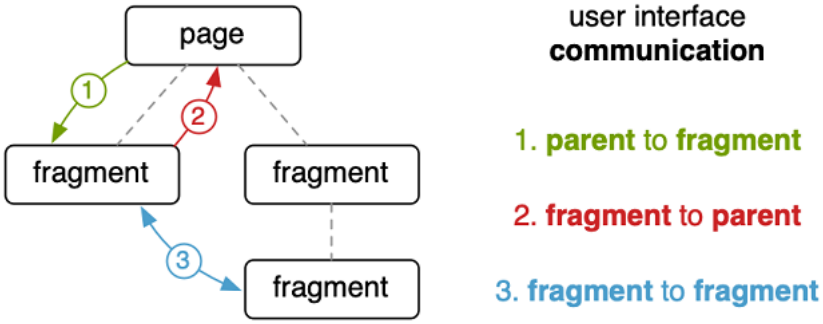

Figure 5.6 There are three different ways of communication that can happen between the different team's UIs inside a page.

We'll go through all three forms of communication with a real use case on our product page. There are a lot of JavaScript libraries that provide convenient ways for communication. But since the goal in a micro frontends architecture is to keep shared code to a minimum we'll focus on native browser features in the examples.

5.2.1 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.

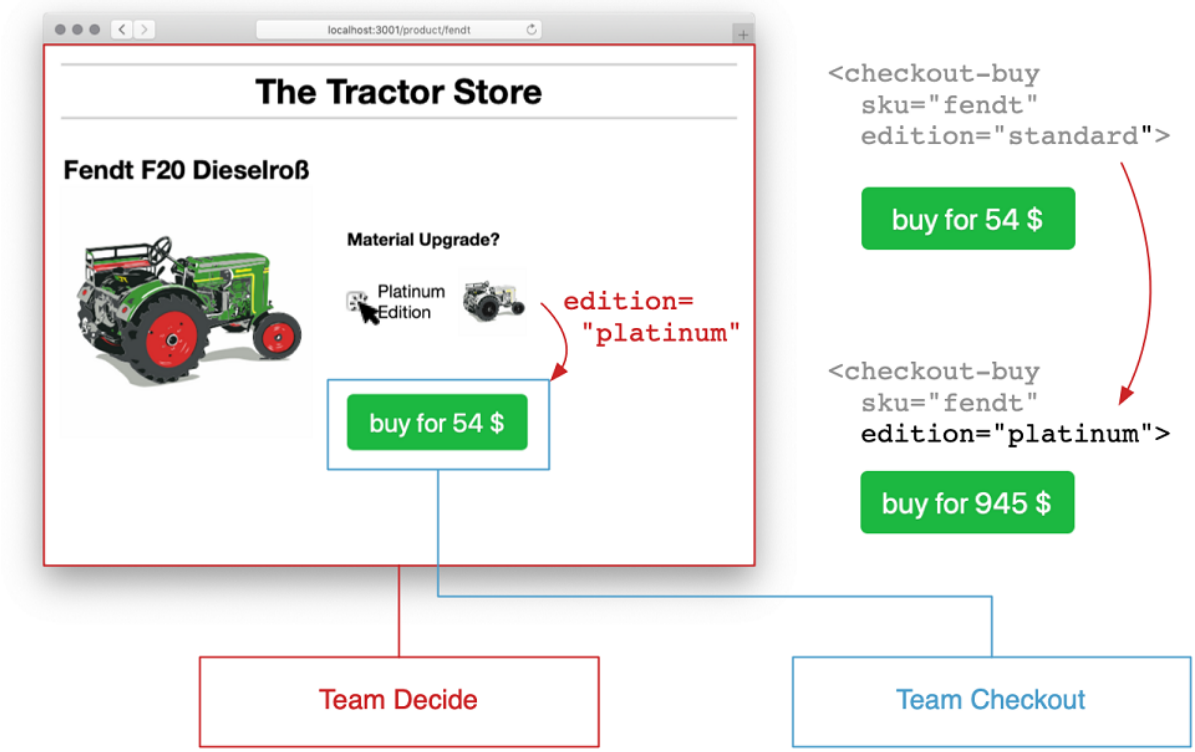

Figure 5.7 Parent-child communication. A change in the parent page (selection of platinum option) needs to be propagated down to a fragment so it can update itself (price change in the buy button).

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

> **TIP** You can find the sample code for this task in the `10_parent_child_communication` folder.

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

Listing 5.6 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>    ❷
...
```

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 5.7 team-decide/static/page.js

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

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

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 `constructor` 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 5.8 team-checkout/static/fragment.js

```
const prices = {
  porsche: { standard: 66, platinum: 966 },     (1)
  fendt: { standard: 54, platinum: 945 },       (1)
  eicher: { standard: 58, platinum: 958 }       (1)
};

class CheckoutBuy extends HTMLElement {
  static get observedAttributes() {             (2)
    return ["sku", "edition"];                  (2)
  }                                             (2)
  constructor() {
    super();
    this.render();                              (2)
  }
  attributeChangedCallback() {                  (2)
    this.render();                              (2)
  }                                             (4)
  render() {                                        (5)
    const sku = this.getAttribute("sku");           (6)
    const edition = this.getAttribute("edition");   (6)
    this.innerHTML = `
      <button type="button">
        buy for ${prices[sku][edition]} $        (7)
      </button>
    `;
    ...
  }
}
```

(1) added new prices for platinum versions

(2) watching for changes to the `sku` and `edition` attribute

(3) extracted the rendering to a separate method

(4) calling `render()` on every attribute change

(5) extracted render method

(6) retrieves the current SKU and edition value from the DOM

(7) 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`.

Now the buy button updates itself on every change to the `sku` or `edition` attribute. 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.

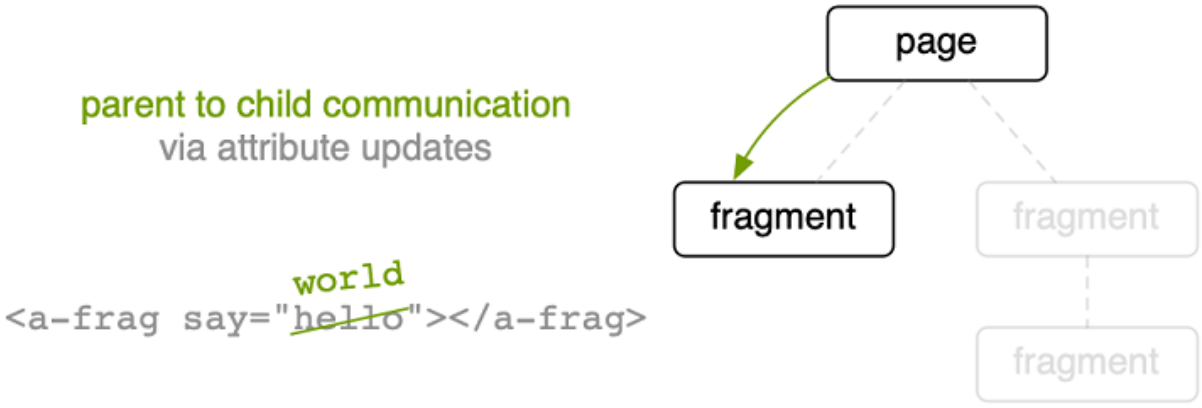

Figure 5.8 You can achieve parent-child communication by explicitly passing needed context information down as an attribute. The fragment can react to this change.

The way we propagate changed state of the outer application (product page) to the nested application (buy button) is similar to the *unidirectional dataflow* [46] pattern. The "props down, events up" approach was popularized by React and Redux. 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.

5.2.2 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. The first hundred platinum tractors were shipped within one day.

Emma is *Team Decide's* UX designer. She loves the new buy button but isn't quite 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. The add-to-cart interaction should be confirmed by an animated green checkmark on the main product image.

This request is a little bit problematic. *Team Checkout* owns the add-to-cart action. Yes, they know when an item was successfully added 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.

To make it happen 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 it's animation in response to that.

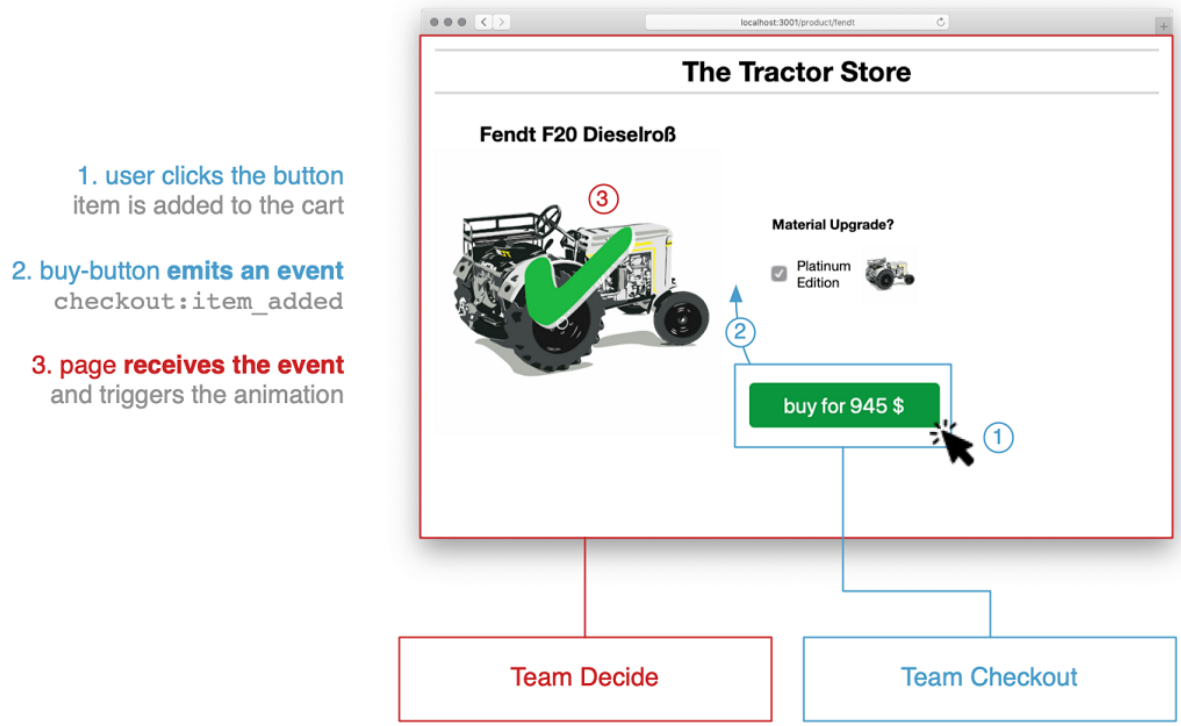

Figure 5.9 Team Checkout's buy button emits an event when an item was added to the cart. Team Decide reacts to this event and triggers an animation on the main product image.

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

> **TIP** You can find the sample code for this task in the `11_child_parent_communication` folder.

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 5.9 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 big green checkmark character () fade in and out again. The animation can be triggered by adding a `decide_product–confirm` class to the existing `decide_product` div element.

Listing 5.10 team-decide/static/page.js

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

Go to localhost:3001/product/fendt in your browser and try the code yourself. Clicking the buy button triggers the event, the `confirm` class is added and the animation starts.

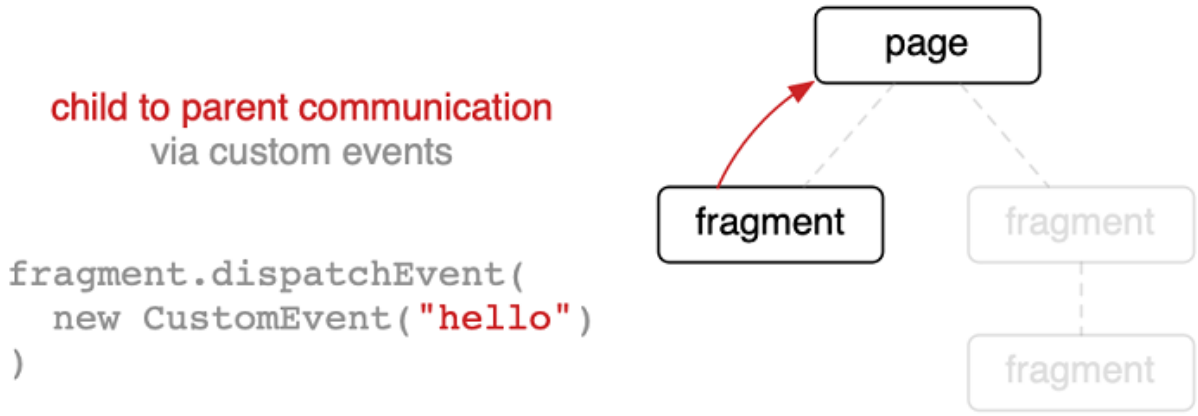

Figure 5.10 Child-parent communication can be implemented by using the browsers builtin event mechanism.

Using the browsers event mechanism has multiple benefits:

- Custom events can have high-level names that reflect your domain language. This makes 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.

5.2.3 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.

To reduce the number of product returns *Team Checkout* wants to add a mini-cart to the product page. 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. The state of the cart will later be fetched from a 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 to this. This 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. The communication flow is explicitly modeled in the parent's system. But to make a change in the 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. This 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. Some browsers also support the new **Broadcast Channel API** [47] 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 5.11 illustrates the event flow between both fragments.

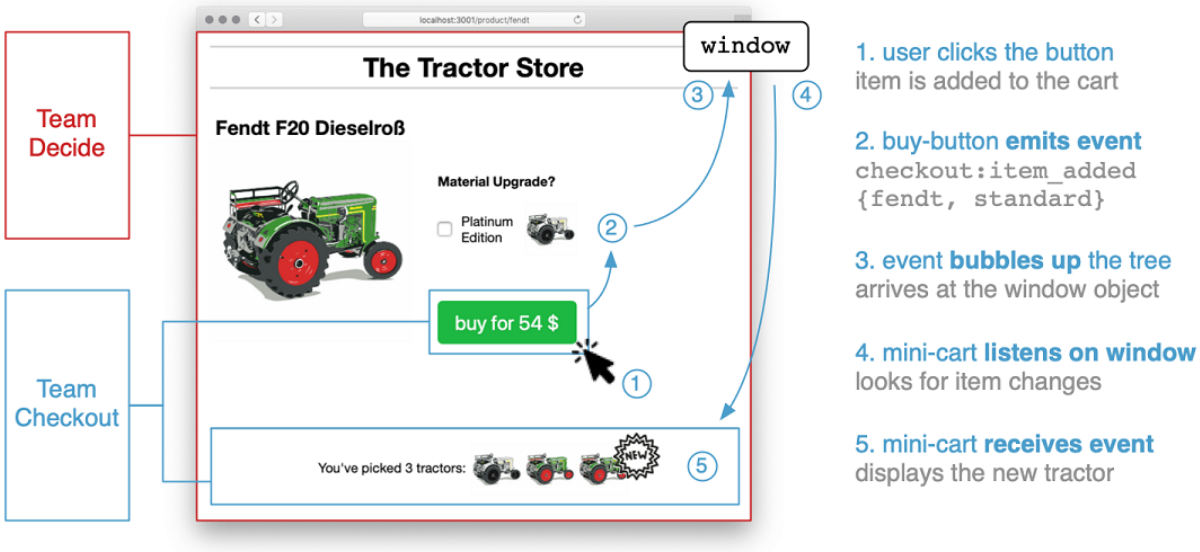

Figure 5.11 Fragment to fragment communication via a global event. The buy button emits the item_added event. The mini-cart listens for this event on the window object and updates itself. The browser's native event mechanism is used as an event-bus.

The mini-cart doesn't only need to know **if** a tractor was added, it also must know **what** tractor was 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

TIP You can find the sample code for this task in the `12_fragment_fragment_communication` folder.

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 5.11 team-checkout/static/fragment.js

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

SIDEBAR **Directly dispatching to 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. This helps with debugging. It's also possible for a parent to intercept the event (`event.stopPropagation`) on it's way up to the `window`.

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 5.12 team-checkout/static/fragment.js

```
...
class CheckoutMinicart extends HTMLElement {
  constructor() {
    super();
    this.items = [];                                                    ❶
    window.addEventListener("checkout:item_added", e => {               ❷
      this.items.push(e.detail);                                        ❷
      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 5.13 team-decide/product/fendt.html

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

❶ adding the new mini-cart fragment to the bottom of the page

Figure 5.12 shows how the event is bubbling up to the top.

Figure 5.12 Custom events can bubble up to the window of the document where other components can subscribe to it.

SIDEBAR **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.

5.2.4 Dos and don'ts of communication

Now you've seen three different types of communication and how to tackle them with basic browser features. You can of course also use custom implementations for communicating and updating components. 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.

HOW MUCH COMMUNICATION?

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.

If implementing a new feature requires two teams to work closely together, passing data back and forth between their micro frontends, you should reconsider your team boundaries. Increasing scope or shifting the responsibility for a use-case from one team to another can help.

WHAT TO TRANSFER

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 simple and lean 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. View-models and domain objects are only exchanged inside team boundaries.

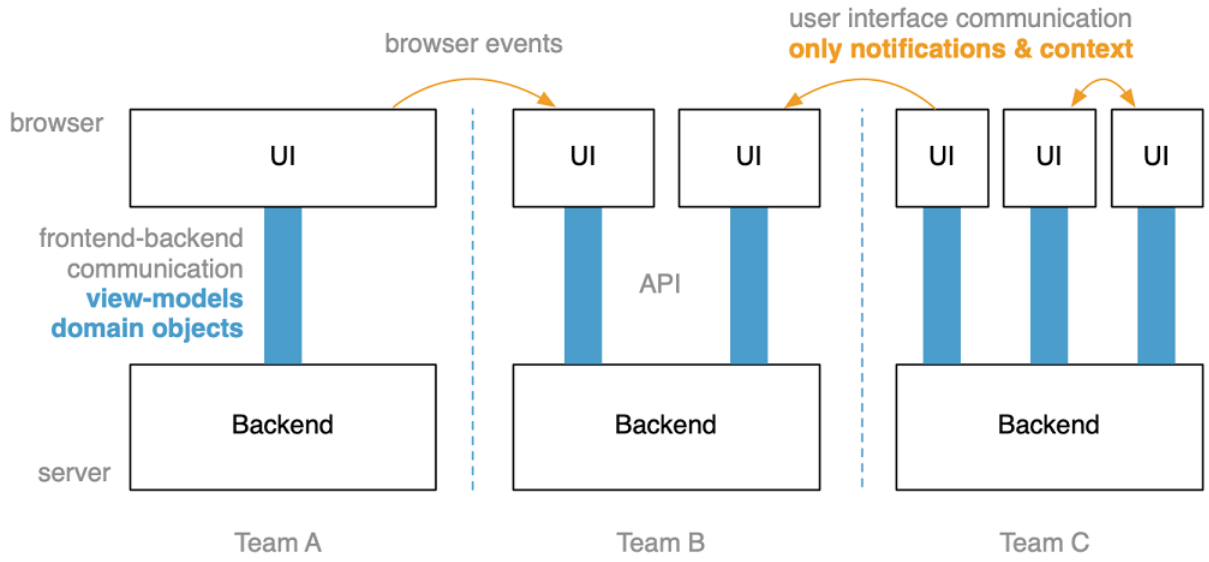

Figure 5.13 UI communication should be used for notifications and not to transfer data from one system to another. Domain objects should stay inside team boundaries. User interfaces retrieve the data they present from their backend.

DON'T SHARE STATE BETWEEN MICRO FRONTENDS

If you're using a state management library like Redux each micro frontend should have its state. It's tempting to reuse state from one micro frontend in another. 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.

NO CROSS-TEAM API 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 A.

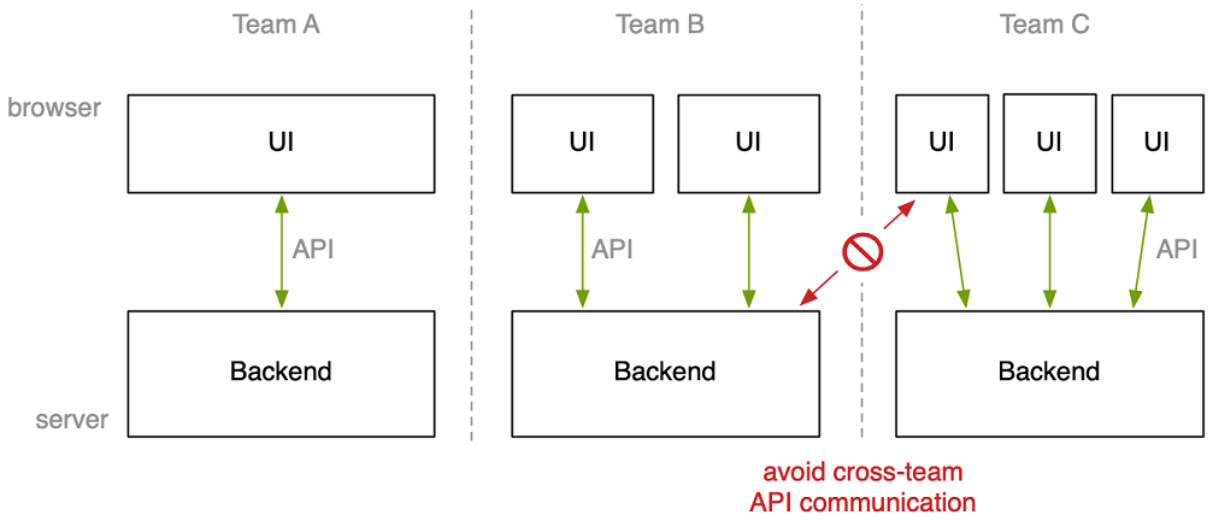

Figure 5.14 API communication should always stay inside team boundaries.

5.3 Style isolation using Shadow DOM

One 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. This makes it possible to shield the styles of different micro frontends against each other. It eliminates the chance of leaked styles and thereby increases robustness.

Let's have a look at the CSS styling introduced for the mini-cart fragment:

Listing 5.14 team-checkout/static/fragment.js

```
checkout-minicart { text-align: center; }
checkout-minicart img { display: inline-block; width: 80px; ...}
```

The styles are scoped with the name of the custom element `checkout-minicart`. This way we avoid styles leaking out to the rest of the page. But *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.

The concept of Shadow DOM provides an alternative where no prefixing or explicit scoping is required.

5.3.1 Creating a Shadow Root

TIP You can find the sample code for this task in the `13_shadow_dom` folder.

You can create an isolated DOM sub-tree via JavaScript by calling `.attachShadow()` on an HTML element. It's mostly used 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` [48].

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

```
class CheckoutMinicart extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: "open" });        ❶
    this.shadowRoot.innerHTML = "Mini Cart";    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 with 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. It's recommended to always use `mode: "open"`.

5.3.2 Scoping styles

Let's move the styling from the `fragment.css` into the actual component. This is done 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[49].

This is the code for the mini-cart fragment.

Listing 5.15 team-checkout/static/fragment.js

```
...
class CheckoutMinicart extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: "open" });
    ...
    this.render();
  }
  render() {
    this.shadowRoot.innerHTML = `                          ❶
      <style>                                              ❶
        :host { text-align: center; ... }                  ❶
        img { width: 80px; display: inline-block; ... }    ❶
      </style>                                             ❶
      You've picked 1 tractor:
      <img src="https://mi-fr.org/img/fendt_standard.svg" />
    `;
  }
}
...
```

❶ writing to the `shadowRoot` instead of the custom element directly

❷ defining styles an inline CSS

❸ `:host` selects the element that owns the Shadow DOM (`checkout-minicart`)

❹ `img` only affects images inside this subtree

After this change, the fragment still looks the same. But we've eliminated the risk of style collisions. Figure 5.15 illustrates the effect of the virtual border the `shadowRoot` introduces. This border is called **shadow boundary**.

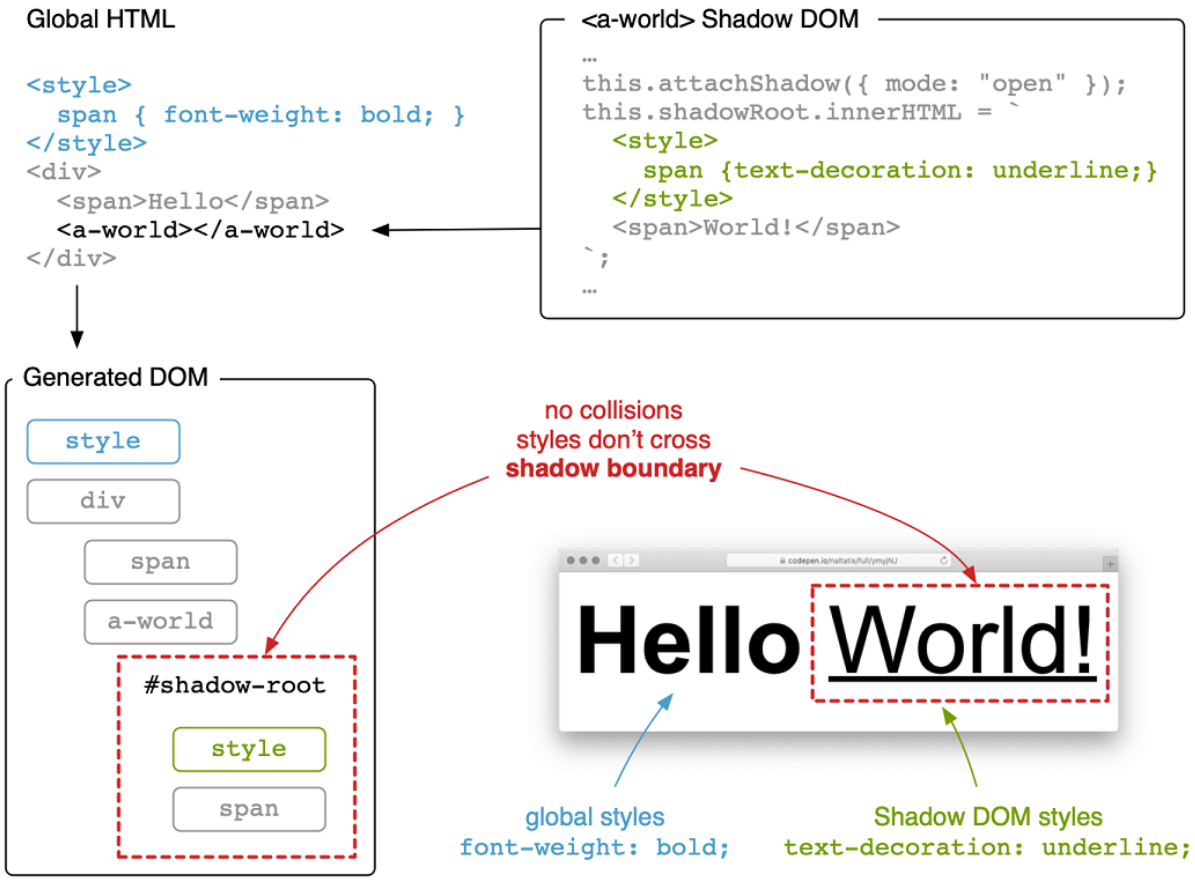

Figure 5.15 The Shadow Root creates a border called shadow boundary. It provides isolation in both ways. Styles don't leak out of the component. Styles on the page also don't affect the Shadow DOM.

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. With Shadow DOM it's possible to work like this without the need for an extra toolchain.

5.3.3 When to use Shadow DOM

There are a lot of details you can learn about Shadow DOM[50]. Events behave differently when they bubble from the Shadow DOM into the normal 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 are based on global CSS classes. Like Twitter Bootstrap.

5.4 The good and bad of integration via Web Components

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.

5.4.1 The benefits

The **biggest 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 a great fit for a common basis.

Custom Elements and Shadow DOM both **provide extra isolation features** that were not possible to achieve before. This 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**. Before you had to come up with home-grown initialization, deinitialization, and updating schemas.

5.4.2 The drawbacks

One of Web Components' biggest 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. This 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 quite easy and stable 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.

5.4.3 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 solid basis.

The interesting question is what interactive means. We'll discuss this topic in 7.1 The app-document continuum: server- or client-render?. For simpler use-cases like catalog or content-heavy sites 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 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**. You can use the techniques learned in 4 Server-side integration? to render the first-page view. When the page is loaded you can apply the client-side interaction concepts you've learned in this chapter.

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

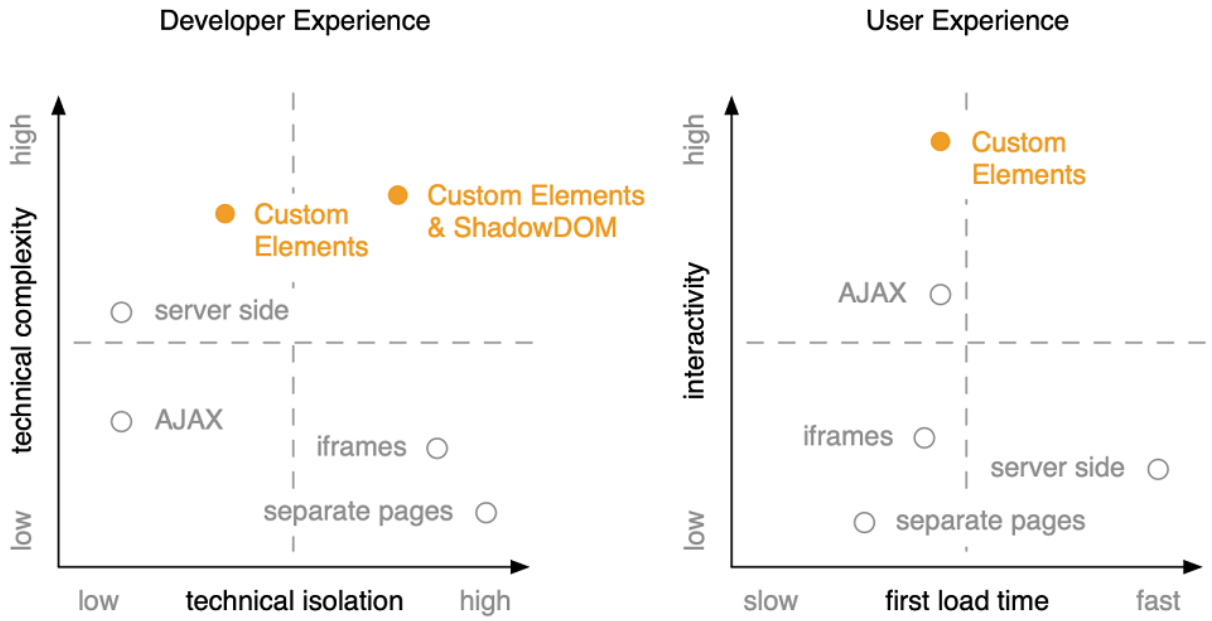

Figure 5.16 Custom Elements provide a good way to encapsulate your JavaScript application and make it accessible in a standard way. Shadow DOM introduces an extra isolation mechanism and lowers the risk of conflicts. You can build highly interactive, client-side rendered applications using Custom Elements. But since they require JavaScript to function a server-rendered solution will usually be quicker on first load.

5.5 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. Business logic and implementation details are hidden inside the component.
- 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. Most communication needs can be handled 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. - User interface communication should only be used for notification.
- 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.
- 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 makes them a great fit for integrating a micro frontend into a legacy application.

6 Unified Single-Page App

This chapter covers

- Eliminating page reloads by combining multiple single-page apps into one
- 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. 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 Integration via links we already covered the most basic page-integration technique: the plain old link. Later in chapter 3.2. Routing via a shared web-server 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** to differentiate between a **full reload from the server** and a **client-side page transition**.

In a monolithic frontend application, it's typically a binary decision. Either you run a more traditional approach and use hard navigations for everything, or you've built a single page

application, and all navigations are soft. In a micro frontends context, it doesn't have to be that black and white. Figure 6.1 shows two simple ways to integrate pages.

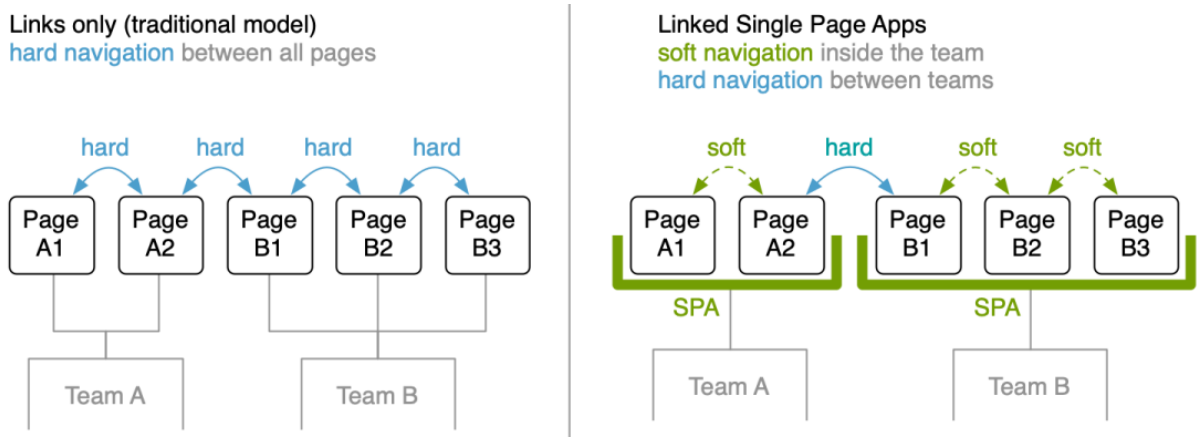

Figure 6.1 Two different approaches for page transitions in a micro frontends context. The "Link only" model is simple. Page transitions happen via plain links, which result in a full refresh of the page. Nothing special is needed - Team A must know how to link to the pages of Team B and vice versa. 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 an 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 6.2 shows a third option where all page transitions are soft.

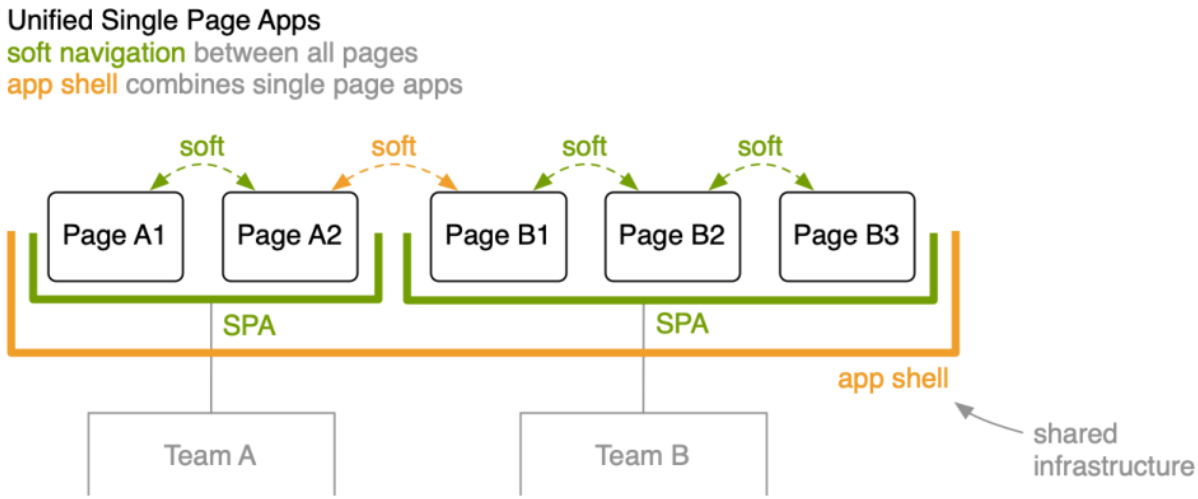

Figure 6.2 The Unified Single Page Apps approach introduces a central application container. It handles page transitions between the teams. Here all navigations are soft.

We'll focus on the "Unified Single Page App" model in this chapter. You'll implement an application shell 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.

6.1 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 straight forward: 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.

6.1.1 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 6.3 illustrates this.

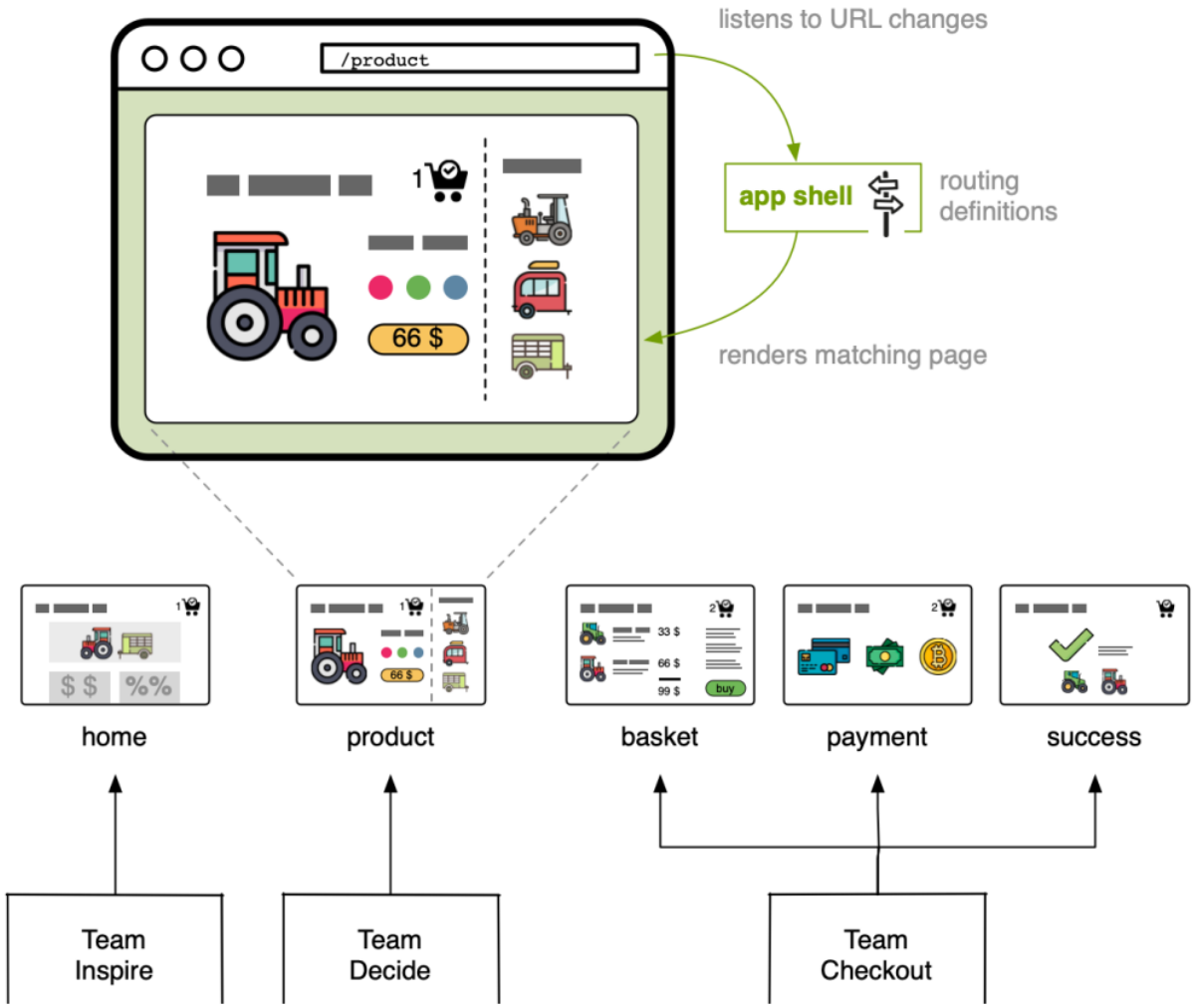

Figure 6.3 The app shell acts as a central client-side router. It watches for URL changes, determines the matching page (micro frontend) and renders it.

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.

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

1. provide a shared HTML document
2. map URLs to team pages
3. render the matching page
4. (de)initialize the page on a navigation

Let's build them in this order.

6.1.2 Anatomy of the app shell

> **TIP** You can find the sample code for this task in the `14_client_side_flat_routing` folder.

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 6.4.

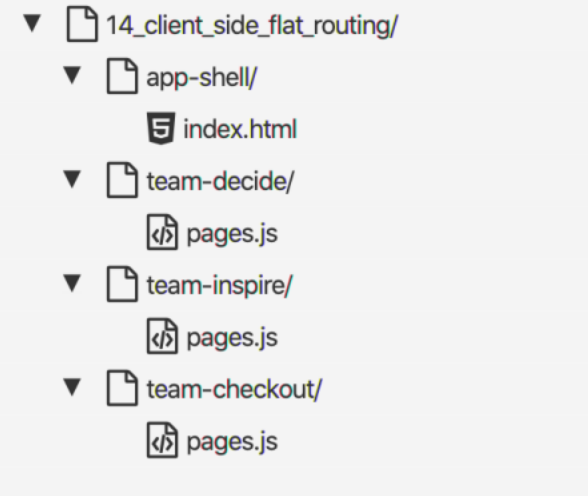

Figure 6.4 The app shell's code is located beside the team's code. It provides a shared HTML document. The teams just deliver page components via JavaScript.

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 this is typically done by specifying rewrite rules. Fortunately `mfserve`, our ad-hoc web-server, has a simple option to enable this behavior. We are adding the `--single` parameter to do the trick.

```
mfserve --listen 3000 --single app-shell
```

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 6.1 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>     ❷
    <script src="http://localhost:3002/pages.js" async></script>     ❸
    <script src="http://localhost:3003/pages.js" async></script>     ❹
  </head>
  <body>
    <div id="app-content">                                           ❷
      <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.

6.1.3 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[51]. 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 6.2 app-shell/index.html

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

const routes = {                                                    ❶
  "/": "inspire-home",                                              ❷
  "/product/porsche": "decide-product-porsche",                     ❷
  "/product/fendt": "decide-product-fendt",                         ❹
  "/product/eicher": "decide-product-eicher",                       ❷
  "/checkout/cart": "checkout-cart",                                ❷
  "/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 6.3 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.

6.1.4 Page components

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 6.4 inspire/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>                        (2)
    `;
  }
}

window.customElements.define("inspire-home", InspireHome);     (2)
```

(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 6.5 decide/pages.js

```
class DecideProductPorsche extends HTMLElement {
  connectedCallback() {
    this.innerHTML = `
      <a href="/">&lt; home</a> -                               (1)
      <a href="/checkout/cart">view cart &gt;</a>               (1)
      <h1>Porsche-Diesel Master 419</h1>
      <img src="https://mi-fr.org/img/porsche.svg" width="200">
    `;
  }
}
window.customElements.define(                                  (2)
  "decide-product-porsche",                                     (2)
  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 6.6 app-shell/index.html

```
...
function updatePageComponent(location) {
  const next = findComponentName(location.pathname);    ❶
  const current = appContent.firstChild;                ❷
  const newComponent = document.createElement(next);    ❸
  appContent.replaceChild(newComponent, current);       ❹
}
...
```

❶ looks up the component name for the current location

❷ reference to the existing page component

❸ instantiates the Custom Element (`constructor` is called)

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 6.7 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();                                    ❸
  }
});
...
```

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. The code does not support clicking on elements nested inside the a-tag. It also doesn't check for anchors or external URLs.

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 6.5 illustrates the links between the pages in the example project.

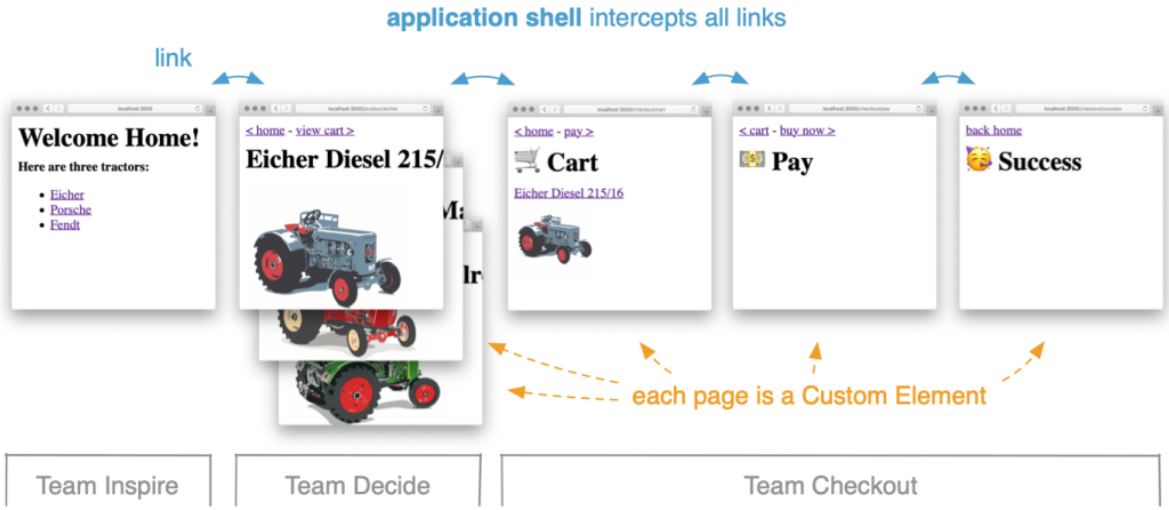

Figure 6.5 The pages in the example project are connected via links. The application shell intercepts these links and performs a soft navigation to the requested page. Teams expose their pages as Custom Elements. On navigation, the app shell replaces the existing page component with the new one.

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.

6.1.5 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.

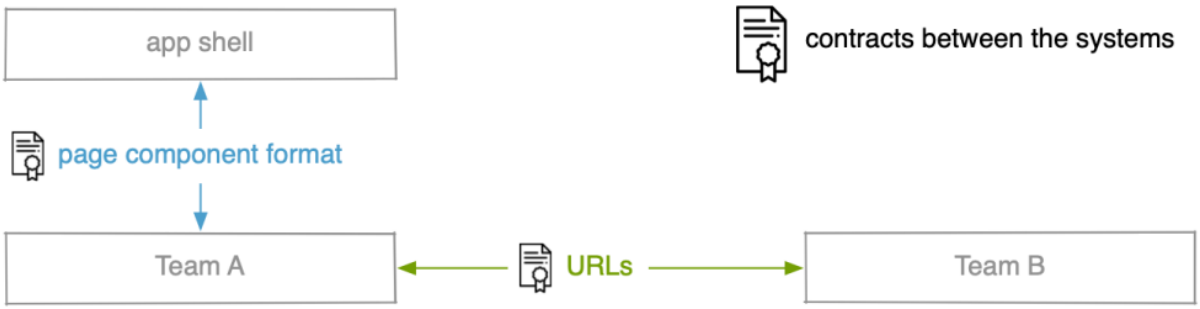

Figure 6.6 Contracts between the systems. Teams need to expose their pages to the app shell in a defined component format (e.g., Web Components). A team needs to know the URL of another team if it wants to link to it.

6.2 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 6.7 illustrates how this works.

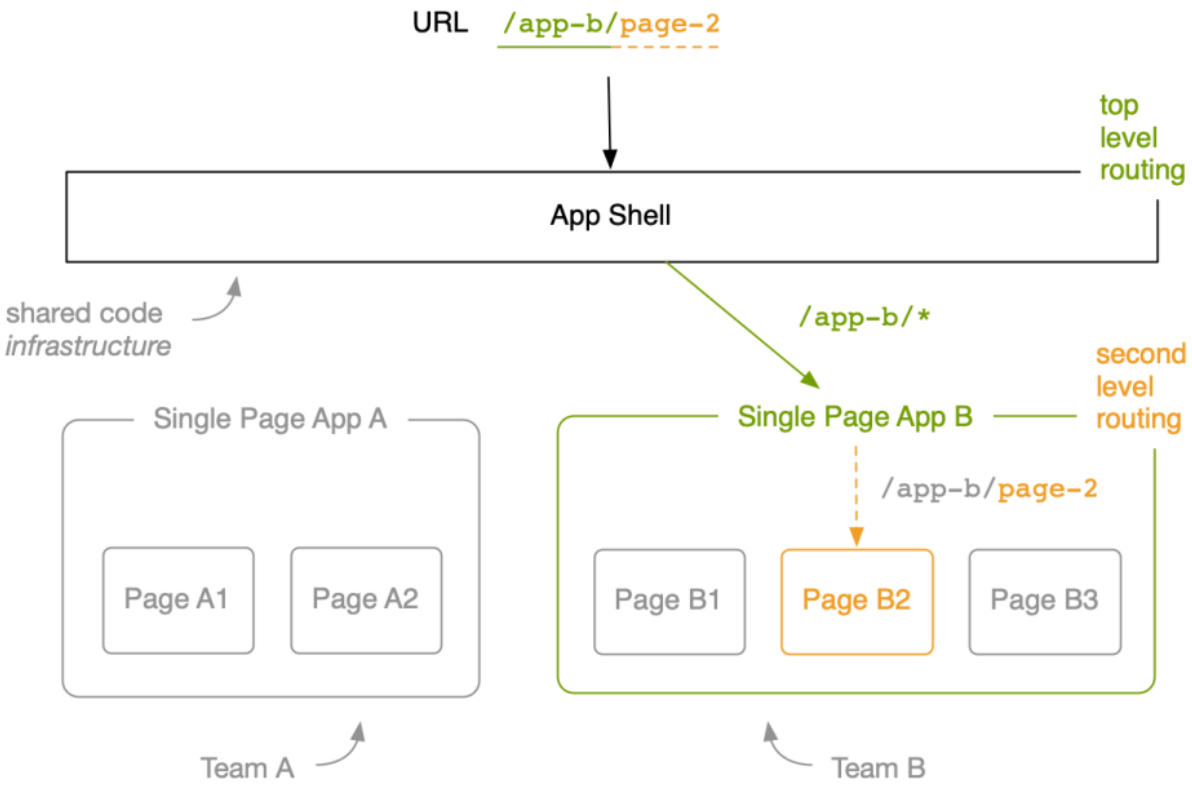

Figure 6.7 Two level routing. The app shell looks at the first part of the URL to determine which team is responsible. The router of the matched team processes the complete URL to find the correct page.

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.

6.2.1 Implementing the top-level router

> **TIP** You can find the sample code for this task in the `15_client_side_two_level_routing` folder.

The app shell script remains mostly the same. We have to change the routing definitions.

Listing 6.8 app-shell/index.html

```
...
const routes = {                                          ❶
  "/product/": "decide-pages",                            ❶
  "/checkout/": "checkout-pages",                         ❶
  "/": "inspire-pages"                                    ❷
};                                                        ❷

function findComponentName(pathname) {                    ❷
  const prefix = Object.keys(routes).find(key =>          ❷
    pathname.startsWith(key)                              ❷
  );                                                      ❷
  return routes[prefix];                                  ❹
}                                                         ❺
...
```

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.

6.2.2 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 6.9 c.heckout/pages.js

```
const routes = {                                                    (1)
  "/checkout/cart": () => `                                         (2)
    <a href="/">&lt; home</a> -                                     (3)
    <a href="/checkout/pay">pay &gt;</a>                            (3)
    <h1>🛒 Cart</h1>                                                (3)
    <a href="/product/eicher">...</a>`,                             (3)
  "/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`).

SIDEBAR **No native "history change" event**

Notice that the app shell now exposes the `History` instance it creates globally via `window.appHistory`. The app shell and team components must use the same instance. The `.listen()` function provides a unified listener that triggers when the location changes a) due to an imperative `history.push|replace` call or b) through a back navigation triggered by the browser.

The native History API only exposes a hook for the latter one via `onpopstate`. A native event like `onchangestate` does not exist in the spec. That's why we are using the history library.

Other ways to deal with this exist. It's possible to extend the native `window.history` object, intercept `pushState`/`replaceState` calls and introduce a non-standard event.

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 two-tiered routing work. The code for the other teams looks similar. You can go deeper and analyze the sample code that comes with this book.

6.2.3 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

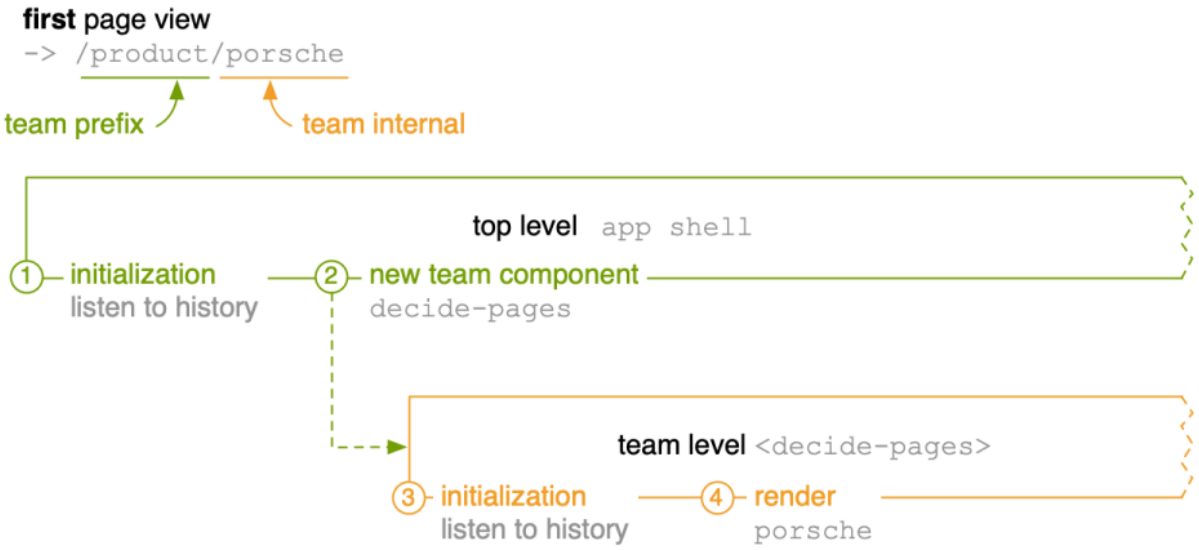

Figure 6.8 First page view in a two-tiered routing approach. The top-level router looks at the team prefix to determine the responsible team. The team router at the second level looks at the last part of the URL to render the actual page.

Figure 6.8 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

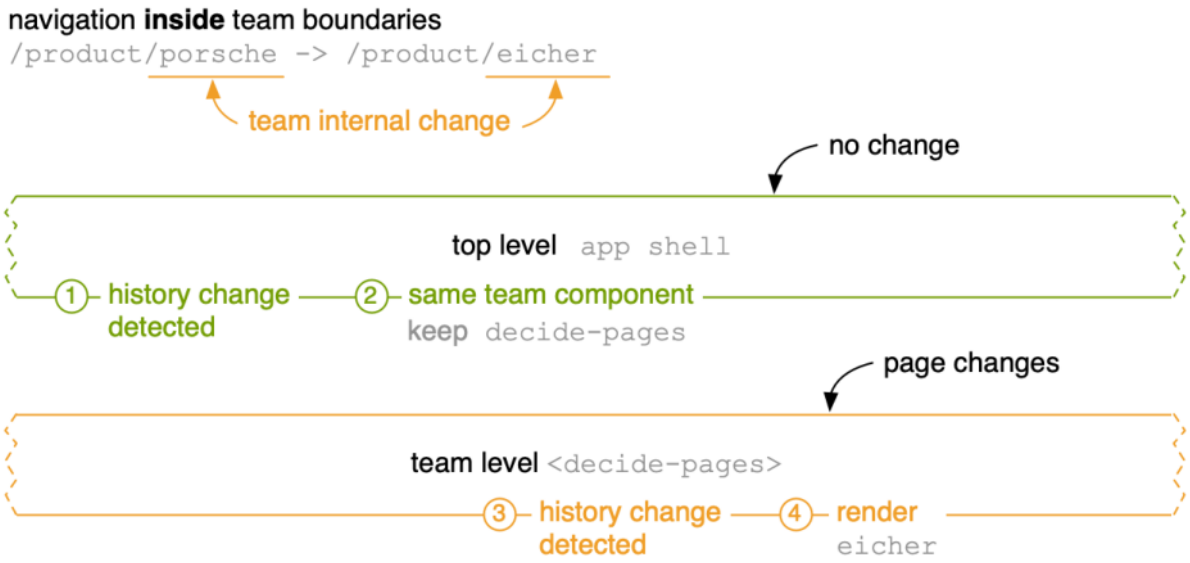

Figure 6.9 When the user navigates to another page controlled by the same team, the app shell has nothing to do. The team level component needs to update the page according to the URL.

Figure 6.9 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

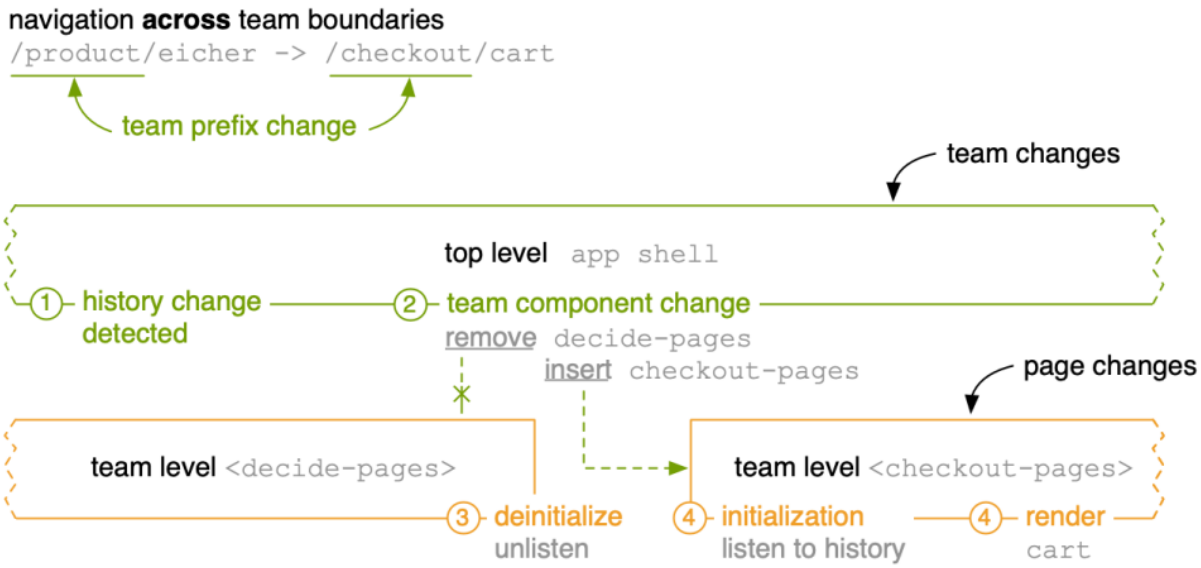

Figure 6.10 On an inter-team navigation the responsibility changes. The app shell replaces the existing team component with a new one. This new component takes over and is in charge of rendering the page.

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 6.10 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.

6.2.4 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 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.

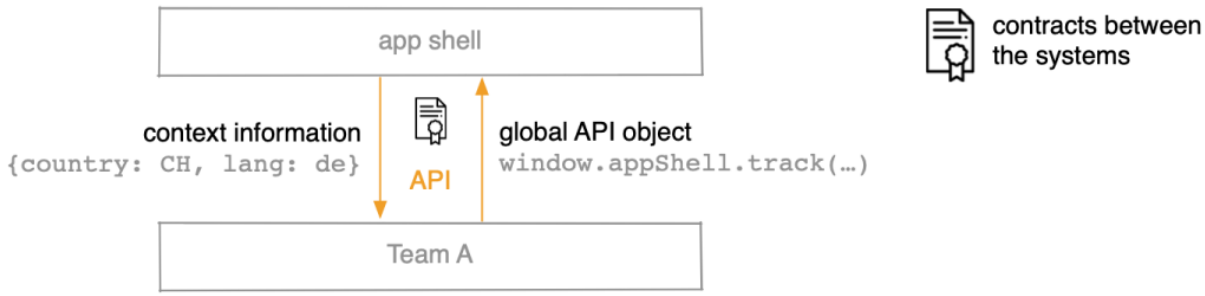

Figure 6.11 Adding shared functionality to the app shell leads to tighter coupling. The API between app shell and team applications acts as a contract between the systems. It should be lean and stable.

Business logic should be in the teams' application code - not in the app 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.

6.3 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-tiered 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 the micro frontends meta-framework single-spa[52]. In essence, it's a top-level router. It comes with **on-demand loading of application code**. The most prominent feature is its ecosystem. Bindings for popular frameworks like React, Vue.js, Angular, Svelte, or Cycle.js exist. 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's* pages are build using Vue.js. Figure 6.12 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.

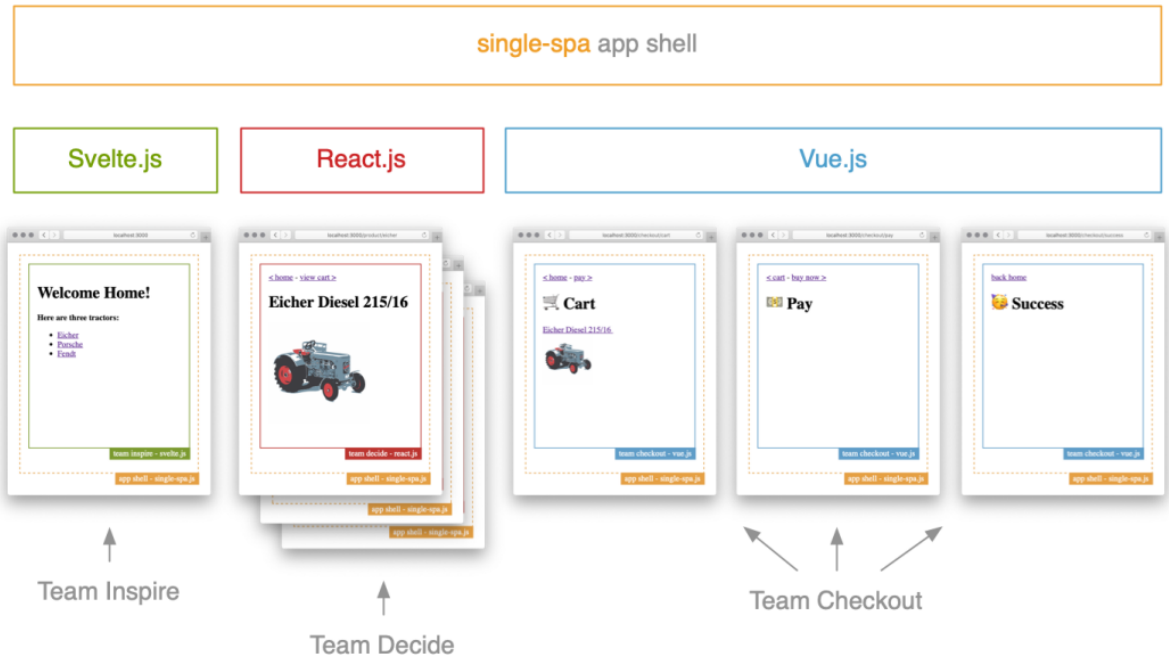

Figure 6.12 Single-spa acts as the application shell. It routes between the applications which can be implemented in different technologies.

Let's have a look at how single-spa works.

6.3.1 How single-spa works

> **TIP** You can find the sample code for this task in the `16_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 6.10 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>                               ❸
    <div id="app-checkout"></div>                             ❹

    <script type="module">
      singleSpa.registerApplication(                          ❸
        "inspire",                                            ❺
        () => import("http://localhost:3002/pages.min.js"),   ❺
        ({ pathname }) => pathname === "/"                    ❺
      );
      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 6.11 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 6.12 team-inspire/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 6.13 team-inspire/Homepage.svelte

```
<script>
  function navigate(e) {                                    ❶
    e.preventDefault();                                     ❷
    const href = e.target.getAttribute("href");             ❷
    window.history.pushState(null, null, href);             ❹
  }                                                         ❷
</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 16_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.

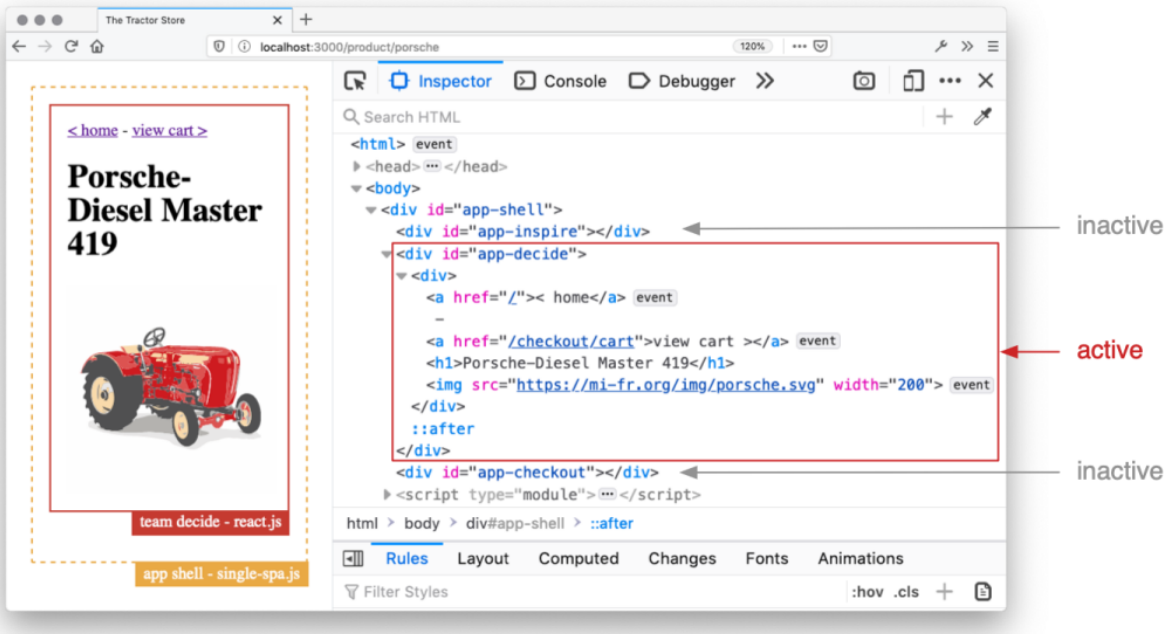

Figure 6.13 With single-spa each micro frontend has it's own DOM node to render its content. In this example, the micro frontend for Team Decide's product page is active. It shows it's content inside the #app-decide element. The other micro frontends are inactive, and their corresponding DOM elements 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 6.13.
- Open the network tab and also notice that single-spa loads the JavaScript bundles (`pages.min.js`) as they are needed and not all upfront.
- 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. 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[53] 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.

6.4 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.

6.4.1 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 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 necessary 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[54], vue-meta[55] or react-helmet[56] 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 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 laborious 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 5.2 Communicating between Micro Frontends 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 like to deliver an excellent overall performance.

6.4.2 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 brings some challenges. 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 constantly 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.

Figure 6.14 shows the tradeoff between providing the best user experience and having a simple setup with low coupling.

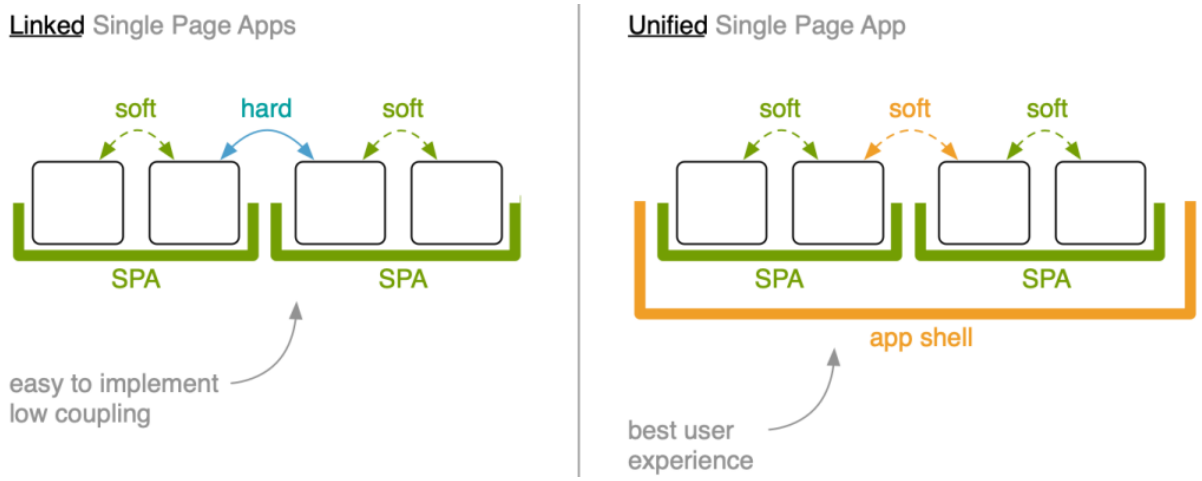

Figure 6.14 Linked single-page apps are easy to build and introduce low coupling. However, they require a hard navigation when moving from one app to the other. The unified single-page app approach solves this and provides a better user experience. However, this enhancement does come with some complexity.

SIDEBAR **The proposed `portal` spec**

Google has proposed a new specification to make navigating between different single-page apps a lot easier. The proposed `<portal>` element[57] behaves like an iframe and comes with even higher[58] technical isolation. It's main feature is, that it'll be possible to promote a page that's loaded in a portal to become the full page.

This way Team A could already instantiate a portal with the micro frontend of Team B. The portal itself might be invisible until the user wants to navigate to it. Now Team A can promote the portal and let Team B take over the complete page. This proposal could eliminate the need for a shared application shell and reduce complexity.

At the time of writing this book this proposal is in an early stage and only Chrome has implemented it behind a flag[59]. But you should keep an eye on it. If it's beomes a real standard it would play nicely with the Linked SPA approach.

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 3.4.

Figure 6.15 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.

6.5 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-tiered 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.

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 five years of inactivity and single digit weekly downloads, lets 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. Canopy's micro frontends meta framework single-spa github.com/CanopyTax/single-spa

12. DAZN - Micro Frontend Architecture www.youtube.com/watch?v=BuRB3djraeM

13. URI Template tools.ietf.org/html/rfc6570

14. Home Documents for HTTP APIs mnot.github.io/I-D/json-home/

15. Swagger OpenAPI Specification swagger.io/specification/

16. developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe#Attributes

17. github.com/davidjbradshaw/iframe-resizer

18. MDN Window.postMessage() developer.mozilla.org/en-US/docs/Web/API/Window/postMessage

19. Quore: How is JavaScript used within the Spotify desktop application? www.quora.com/How-is-JavaScript-used-within-the-Spotify-desktop-application-Is-it-packaged-up-and-run-locally-o

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 Googlebots JavaScript support developers.google.com/search/docs/guides/rendering

26. Getting To Know The MutationObserver API www.smashingmagazine.com/2019/04/mutationobserver-api-guide/

27. Zalando Skipper opensource.zalando.com/skipper/

28. WebPageTest is a good open-source tool for doing this www.webpagetest.org/

29. This behavior can be changed 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

30. ESI Language Specification 1.0 www.w3.org/TR/esi-lang

31. 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

32. www.zalando.com

33. www.mosaic9.org/

34. github.com/zalando/tailor

35. github.com/zalando/tailor#options

36. Front-End Micro Services jobs.zalando.com/tech/blog/front-end-micro-services/

37. www.finn.no/

38. podium-lib.io/

39. expressjs.com/

40. podium-lib.io/docs/podium/conceptual_overview/

41. Web Component Polyfill www.webcomponents.org/polyfills

42. github-elements www.webcomponents.org/author/github

43. stenciljs.com/

44. Angular Elements Overview angular.io/guide/elements

45. github.com/vuejs/vue-web-component-wrapper

46. Unidirectional Data Flow en.wikipedia.org/wiki/Unidirectional_Data_Flow_(computer_science)

47. Broadcast Channel API developer.mozilla.org/en-US/docs/Web/API/Broadcast_Channel_API

48. List of elements that support Shadow DOM dom.spec.whatwg.org/#dom-element-attachshadow

49. Except for a few inherited properties like font-family and root font-size.

50. Encapsulating Style and Structure with Shadow DOM css-tricks.com/encapsulating-style-and-structure-with-shadow-dom/

51. github.com/ReactTraining/history

52. single-spa.js.org

53. single-spa.js.org/docs/getting-started-overview.html

54. angular.io/api/platform-browser/Meta

55. vue-meta.nuxtjs.org

56. github.com/nfl/react-helmet

57. Portals - A proposal for enabling seamless navigations between sites or pages github.com/WICG/portals

58. Communication should only be possible through `postMessage`. No shared JavaScript objects or `WindowProxy`.

59. Hands-on with Portals: seamless navigation on the Web web.dev/hands-on-portals/

60. RequireJS requirejs.org

61. CommonJS www.commonjs.org

62. Can I use - JavaScript modules: dynamic import() caniuse.com/#feat=es6-module-dynamic-import

63. Ilya Grigorik: Analyzing Critical Rendering Path Performance developers.google.com/web/fundamentals/performance/critical-rendering-path/analyzing-crp

64. Lighthouse developers.google.com/web/tools/lighthouse

65. Uses inefficient cache policy on static assets developers.google.com/web/tools/lighthouse/audits/cache-policy

66. Critical Request Chains developers.google.com/web/tools/lighthouse/audits/critical-request-chains

67. Sticky Sessions in Kubernetes medium.com/@zhimin.wen/sticky-sessions-in-kubernetes-56eb0e8f257d