



Micro Frontends IN ACTION

Michael Geers

 MANNING

Micro Frontends in Action

MICHAEL GEERS



MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

©2020 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

- ⊗ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.



Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Tricia Louvar
Technical development editor: Louis Lazaris
Review editor: Ivan Martinović
Production editor: Deirdre S. Hiam
Copy editor: Ben Berg
Proofreader: Melody Dolab
Technical proofreader: Mayur Patil
Typesetter and cover designer: Marija Tudor

ISBN 9781617296871
Printed in the United States of America

contents

<i>preface</i>	<i>ix</i>
<i>acknowledgments</i>	<i>xi</i>
<i>about this book</i>	<i>xiii</i>
<i>about the author</i>	<i>xvi</i>
<i>about the cover illustration</i>	<i>xvii</i>

PART 1 GETTING STARTED WITH MICRO FRONTENDS 1

1	<i>What are micro frontends?</i>	3
1.1	The big picture	4
	<i>Systems and teams</i>	4
	<i>The frontend</i>	7
	<i>Frontend integration</i>	10
	<i>Shared topics</i>	11
1.2	What problems do micro frontends solve?	12
	<i>Optimize for feature development</i>	12
	<i>No more frontend monolith</i>	13
	<i>Be able to keep changing</i>	14
	<i>The benefits of independence</i>	16
1.3	The downsides of micro frontends	17
	<i>Redundancy</i>	17
	<i>Consistency</i>	18
	<i>Heterogeneity</i>	18
	<i>More frontend code</i>	19
1.4	When do micro frontends make sense?	19
	<i>Good for medium-to-large projects</i>	19
	<i>Works best on the web</i>	19
	<i>Productivity versus overhead</i>	20
	<i>Where micro frontends are not a great fit</i>	20
	<i>Who uses micro frontends?</i>	21

2	<i>My first micro frontends project</i>	23
2.1	Introducing The Tractor Store	24
	<i>Getting started</i>	24
	<i>Running this book's example code</i>	25
2.2	Page transition via links	27
	<i>Data ownership</i>	28
	<i>Contract between the teams</i>	28
	<i>How to do it</i>	29
	<i>Dealing with changing URLs</i>	32
	<i>The benefits</i>	32
	<i>The drawbacks</i>	32
	<i>When do links make sense?</i>	33
2.3	Composition via iframe	33
	<i>How to do it</i>	34
	<i>The benefits</i>	35
	<i>The drawbacks</i>	35
	<i>When do iframes make sense?</i>	36
2.4	What's next?	36

PART 2 ROUTING, COMPOSITION, AND COMMUNICATION 39

3	<i>Composition with Ajax and server-side routing</i>	41
----------	--	-----------

3.1	Composition via Ajax	42
	<i>How to do it</i>	43
	<i>Namespace styles and scripts</i>	45
	<i>Declarative loading with h-include</i>	47
	<i>The benefits</i>	48
	<i>The drawbacks</i>	49
	<i>When does an Ajax integration make sense?</i>	50
	<i>Summary</i>	50
3.2	Server-side routing via Nginx	51
	<i>How to do it</i>	53
	<i>Namespace resources</i>	55
	<i>Route configuration methods</i>	56
	<i>Infrastructure ownership</i>	57
	<i>When does it make sense?</i>	58

4	<i>Server-side composition</i>	59
----------	--------------------------------	-----------

4.1	Composition via Nginx and Server-Side Includes (SSI)	60
	<i>How to do it</i>	61
	<i>Better load times</i>	63
4.2	Dealing with unreliable fragments	64
	<i>The flaky fragment</i>	65
	<i>Integrating the Near You fragment</i>	66
	<i>Timeouts and fallbacks</i>	67
	<i>Fallback content</i>	68
4.3	Markup assembly performance in depth	69
	<i>Parallel loading</i>	69
	<i>Nested fragments</i>	70
	<i>Deferred loading</i>	71
	<i>Time to first byte and streaming</i>	71
4.4	A quick look into other solutions	73
	<i>Edge-Side Includes</i>	73
	<i>Zalando Tailor</i>	73
	<i>Podium</i>	75
	<i>Which solution is right for me?</i>	81

4.5 The good and bad of server-side composition 82

The benefits 82 ▪ *The drawbacks* 83 ▪ *When does server-side integration make sense?* 83

5 Client-side composition 85

5.1 Wrapping micro frontends using Web Components 86

How to do it 87 ▪ *Wrapping your framework in a Web Component* 92

5.2 Style isolation using Shadow DOM 93

Creating a shadow root 93 ▪ *Scoping styles* 94 ▪ *When to use Shadow DOM* 96

5.3 The good and bad of using Web Components
for composition 96

The benefits 96 ▪ *The drawbacks* 97 ▪ *When does client-side integration make sense?* 97

6 Communication patterns 99

6.1 User interface communication 100

Parent to fragment 101 ▪ *Fragment to parent* 104

Fragment to fragment 107 ▪ *Publish/Subscribe with the Broadcast Channel API* 111 ▪ *When UI communication is a good fit* 112

6.2 Other communication mechanisms 113

Global context and authentication 113 ▪ *Managing state* 114

Frontend-backend communication 115 ▪ *Data replication* 115

7 Client-side routing and the application shell 118

7.1 App shell with flat routing 120

What's an app shell? 121 ▪ *Anatomy of the app shell* 122

Client-side routing 123 ▪ *Rendering pages* 124 ▪ *Contracts between app shell and teams* 127

7.2 App shell with two-level routing 128

Implementing the top-level router 129 ▪ *Implementing team-level routing* 130 ▪ *What happens on a URL change?* 131

App shell APIs 133

7.3 A quick look into the single-spa meta-framework 134

How single-spa works 135

7.4 The challenges of a unified single-page app 140

Topics you need to think about 140 ▪ *When does a unified single-page app make sense?* 142

8	Composition and universal rendering	145
8.1	Combining server- and client-side composition	147
<i>SSI and Web Components</i>	148	▪ <i>Contract between the teams</i> 152
<i>Other solutions</i>	153	
8.2	When does universal composition make sense?	153
<i>Universal rendering with pure server-side composition</i>	153	
<i>Increased complexity</i>	154	▪ <i>Universal unified single-page app?</i> 154

9	Which architecture fits my project?	156
9.1	Revisiting the terminology	157
<i>Routing and page transitions</i>	157	▪ <i>Composition techniques</i> 158
<i>High-level architectures</i>	159	
9.2	Comparing complexity	161
<i>Heterogeneous architectures</i>	162	
9.3	Are you building a site or an app?	162
<i>The Documents-to-Applications Continuum</i>	163	▪ <i>Server, client, or both</i> 164
9.4	Picking the right architecture and integration technique	165
<i>Strong isolation (legacy, third party)</i>	166	▪ <i>Fast first-page load/progressive enhancement</i> 167
<i>Instant user feedback</i>	167	▪ <i>Soft navigation</i> 168
<i>Multiple micro frontends on one page</i>	168	▪ <i>Multiple micro frontends on one page</i> 168

PART 3 **HOW TO BE FAST, CONSISTENT, AND EFFECTIVE 171**

10	Asset loading	173
10.1	Asset referencing strategies	174
<i>Direct referencing</i>	174	▪ <i>Challenge: Cache-busting and independent deployments</i> 175
<i>Referencing via redirect (client)</i>	176	▪ <i>Referencing via include (server)</i> 178
<i>Challenge: Synchronizing markup and asset versions</i>	180	
<i>Inlining</i>	183	▪ <i>Integrated solutions (Tailor, Podium, ...)</i> 183
<i>Quick summary</i>	185	
10.2	Bundle granularity	186
<i>HTTP/2</i>	186	▪ <i>All-in-one bundle</i> 187
<i>Team bundles</i>	187	
<i>Page and fragment bundles</i>	187	

- 10.3 On-demand loading 188
Proxy micro frontends 188 ▪ *Lazy loading CSS* 189

11 Performance is key 190

- 11.1 Architecting for performance 191
Different teams, different metrics 191 ▪ *Multi-team performance budgets* 192 ▪ *Attributing slowdowns* 193 ▪ *Performance benefits* 195
- 11.2 Reduce, reuse... vendor libraries 196
Cost of autonomy 196 ▪ *Pick small* 197 ▪ *One global version* 199 ▪ *Versioned vendor bundles* 200 ▪ *Don't share business code* 211

12 User interface and design system 213

- 12.1 Why a design system? 214
Purpose and role 215 ▪ *Benefits* 215
- 12.2 Central design system versus autonomous teams 216
Do I need my own design system? 216 ▪ *Process, not project* 217
Ensure sustained budget and responsibility 217 ▪ *Get buy-in from the teams* 218 ▪ *Development process: Central versus federated* 219 ▪ *Development phases* 221
- 12.3 Runtime versus build-time integration 222
Runtime integration 222 ▪ *Versioned package* 224
- 12.4 Pattern library artifacts: Generic versus specific 226
Choose your component format 227 ▪ *There will be change* 230
- 12.5 What goes into the central pattern library? 231
The costs of sharing components 231 ▪ *Central or local?* 231
Central and local pattern libraries 233

13 Teams and boundaries 236

- 13.1 Aligning systems and teams 237
Identifying team boundaries 238 ▪ *Team depth* 240
Cultural change 242
- 13.2 Sharing knowledge 243
Community of practice 243 ▪ *Learning and enabling* 244
Present your work 245
- 13.3 Cross-cutting concerns 245
Central infrastructure 245 ▪ *Specialized component team* 246
Global agreements and conventions 247

13.4 Technology diversity 247

- Toolbox and defaults* 247 ▪ *Frontend blueprint* 247
- Don't fear the copy* 248 ▪ *The value of similarity* 249

14 Migration, local development, and testing 251

14.1 Migration 252

- Proof of concept and building a lighthouse* 252 ▪ *Strategy #1: Slice-by-slice* 254
- Strategy #2: Frontend first* 255
- Strategy #3: Greenfield and big bang* 256

14.2 Local development 258

- Don't run another team's code* 258 ▪ *Mocking fragments* 259
- Fragments in isolation* 260 ▪ *Pulling other teams micro frontends from staging or production* 262

14.3 Testing 262

index 265

preface

I've been developing applications for the web for over 20 years now. On this journey, I've seen a variety of different-sized projects. I've built tiny side-projects all by myself, have been part of smaller projects with a couple of people, and have also worked on larger projects that involved more people than can comfortably fit around our kitchen table.

In 2014, my colleagues at neuland Büro für Informatik and I had the task of rebuilding an e-commerce system for a department store chain. The existing monolithic shop not only suffered from performance issues. The major pain-point was an organizational one: adding new features took a long time and often broke unrelated parts of the system. Increasing the development team made this even worse. Our client not only wanted a cleaner-structured new software, but they also wanted to architect the new system so that multiple teams could work on it independently without stepping on each other's toes. This parallel feature development was crucial to their plan of digitally expanding their business. We opted for an architecture we called *verticalization*: the establishment of different cross-functional teams that build and evolve a specific area of the shop from database to user interface. The individual team applications were able to work autonomously and only integrated in the frontend. This frontend integration looked easy on paper, but we had to learn a lot to do it effectively. In later projects, we had the chance to refine our techniques and learn from this experience.

At the same time, other companies were already building their software this way. However, there was no unique name for this architecture. What search term should I use if I want to learn about the challenges involved in building a web application with multiple autonomous teams? In November 2016, the ThoughtWorks Technology Radar changed this by coining the term *micro frontends*. The introduction of this name

made it possible for the development community to share best practices, techniques, and tools around this architecture.

The following summer, I was able to dedicate some time to write down our experiences. I distilled the techniques we were using into standalone sample projects and published the content at <https://micro-frontends.org>. From that point, things took on a life of their own: people from across the internet invited me to speak at their conferences. Magazines asked me to write articles. Developers from the community volunteered to translate the site into different languages.

To top things off, I was approached by Nicole and Brian from Manning at the beginning of last year. They asked if I could see myself writing a book on this topic. My first thought was, “What a hilarious idea—I’m not a book writer! I don’t even enjoy reading texts. I much prefer listening, writing code, building systems, and solving problems.” But since this seemed like a once-in-a-lifetime opportunity, I gave my reply some thought. I had long consultations with friends and family and some sleepless nights, but in the end, I accepted the challenge. After all, I like explaining things. Doing it in book form, with diagrams (I love good diagrams) and code examples would be a venture where I could learn a lot. In retrospect, I’m happy with this decision—and the final result you are looking at right now.

acknowledgments

The cover of the book prominently features my name as the author. But this is not a single-person effort. It takes a village to create a book like this. I would like to thank

- Emma, Noah, and Finn for your patience and understanding. In the last year, I spent much less time with you than I'd like.
- Sarah, my wonderful wife, for your repeated encouragement and your fresh perspectives. You jumped in when I did long evenings or worked through weekends. You rock!
- Tricia Louvar, my editor at Manning. You guided me through this journey, aggregated feedback, challenged my decisions, and pointed me to sections that needed more clarity.
- Dennis Reimann, Fabricius Seifert, Marco Pantaleo, and Alexander Knöller for bouncing ideas, reading my drafts, and iterating on graphics.
- The team at Manning who worked with me to plan, develop, review, edit, produce and promote this book. Thanks to Ana Romac, Brian Sawyer, Candace Gillhoolley, Christopher Kaufmann, Ivan Martinović, Lana Klasic, Louis Lazaris, Matko Hrvatin, Mayur Patil, Nicole Butterfield, Radmila Ercegovac, Deirdre Hiam, Ben Berg, and Melody Dolab.
- My folks at neuland Büro für Informatik for giving me the ability to continuously learn on new projects and providing space to create this book. Thank you, Jens and Thomas, and thanks to all the others who encouraged me to do this.
- All book reviewers who read my manuscript in various stages. Your feedback helped me to improve my chapters and adjust the focus. Thanks to Adail Reta mal, Alan Bogusiewicz, Barnaby Norman, David Osborne, David Paccoud, Dwight Wilkins, George Onofrei, Ivo Sánchez Checa Crosato, Karthikeyarajan

Rajendran, Luca Mezzalira, Luis Miguel Cabezas Granado, Mario-Leander Reimer, Matt Ferderer, Matthew Richmond, Miguel Eduardo Gil Biraud, Mladen Đurić, Potito Coluccelli, Raushan Jha, Richard Vaughan, Ryan Burrows, Tanya Wilke, and Tony Sweets.

- All MEAP readers. Receiving encouragement from good friends is one thing. Seeing that people from all over the world put in real money to get early access felt remarkable. You motivated me to pull through this, even if I'd sometimes rather have spent the evening on the couch.
- Samantha, macOS's text-to-speech voice, for relentlessly reading back every version of every paragraph I've written. Take that, dyslexia! A toast to accessibility.

about this book

I've written *Micro Frontends in Action* to explain the concepts and motivations for adopting a micro frontends architecture. You'll learn a series of practical techniques to accomplish frontend integration and communication. Since the landscape is pretty new and use cases can be very different, I decided not to go with one specific micro frontends library, tool, or platform. Instead, you'll learn the fundamental mechanisms by building directly on existing web standards wherever possible. At the end of the book, we'll address overarching topics like how to ensure good performance, coherent design, and knowledge sharing in a distributed team structure.

Who should read this book

This book has the word *frontend* in its title, and in most chapters, we work at some aspect of the user interface. However, this is not only a book for frontend developers. If your expertise is more on the backend-side, or you are a software architect, you won't be lost. As long as you have a basic understanding of HTML, CSS, JavaScript, and networking, you're good to go. You don't need to be familiar with specific libraries or frontend frameworks to understand the techniques described in this book.

How this book is organized: a roadmap

This book has three parts and a total of 14 chapters.

Part 1 explains what micro frontends are and when it's a good idea to use them:

- Chapter 1 paints the big picture. It explains what micro frontends are and goes through the benefits and drawbacks of this architecture.
- Chapter 2 walks you through your first micro frontends project. We'll start simply and won't use fancy techniques—just plain old links and iframes. In this chapter, we create a solid basis to iterate upon.

Part 2 focuses on frontend integration techniques. It gives answers to the question “How do user interfaces from different teams come together in the browser?” You’ll learn approaches for routing and composition for server- and client-rendered applications:

- Chapter 3 illustrates how to do composition using Ajax calls and implement server-based routing with a shared Nginx web server.
- Chapter 4 dives deep into server-side composition. You’ll learn how to compose markup from different applications via Nginx’s SSI feature. We’ll shine a light on some techniques to ensure proper performance even if something goes wrong. We’ll also discuss some alternative implementations like ESI, Tailor, and Podium.
- Chapter 5 addresses composition for client-rendered applications. You’ll learn how to compose UIs written in different technologies into a single view by leveraging the power of Web Components.
- Chapter 6 covers communication strategies. We focus on in-browser communication between different micro frontends. At the end of the chapter, we also address topics like backend communication and how to share information like a login status across teams.
- Chapter 7 introduces the concept of the application shell. The shell enables you to build a full client-rendered user experience that consists of single-page applications built by different teams. You learn how to develop an application shell from scratch, and we finish by taking a look at the popular single-spa library.
- Chapter 8 describes how you can accomplish universal rendering in a micro frontends architecture. We do this by combining server- and client-side integration techniques you’ve already learned in the preceding chapters.
- Chapter 9 rounds off the second part by putting the learned techniques into context. It provides you with a set of questions and tools to decide which micro frontend architecture is the best one for your project.

Part 3 explains practices to ensure good end-user performance and a consistent user interface. It guides how to organize your teams to get the most value out of the micro frontends architecture:

- Chapter 10 dives into asset-loading strategies to deliver the required JavaScript and CSS code to the customer’s browser in a performant way without introducing inter-team coupling.
- Chapter 11 describes how techniques like performance budgets can work even if code from multiple teams are active on a single page. We discuss methods to reduce the amount of vendor code like framework runtimes.
- Chapter 12 illustrates how to design systems that can help to deliver a consistent user interface to your customers, even if different teams build it. You’ll learn some organizational patterns that have proven valuable. We compare different ways of integrating a pattern library with the micro frontends and discuss their technical implications.

- Chapter 13 focuses on the organization. It answers the questions “How cross-functional should my teams be?” and “How do I identify good system boundaries?” You’ll learn about ways to effectively share knowledge and organize cross-cutting concerns and shared infrastructure components.
- Chapter 14 highlights some migration strategies for moving from a monolithic application to a micro frontends architecture. It also addresses the challenges of local development and testing.

About the code

All source code in the book is presented in a monospaced typeface like this, which sets it off from the surrounding text. In many listings, the code is annotated to point out key concepts, and numbered bullets are used in the text to provide additional information about the code. Throughout this book, we’ll build an e-commerce application. We start small and expand on it chapter by chapter. Most listings are shortened [...] to avoid repeating code.

The full source code is available for download from the publisher’s website at <https://www.manning.com/books/micro-frontends-in-action> and GitHub at <https://github.com/naltatis/micro-frontends-in-action-code>. I recommend downloading and running the code on your machine as you move through the chapters. You can also find a hosted version at <https://the-tractor.store>. There you can interact with all book examples and look at the associated code directly in your browser.

The applications in this book are built using static files. You don’t need to know a specific backend language like Java, Python, C#, or Ruby. To start the applications we use ad hoc web servers which require Node.js to be installed on your machine. In the chapters that cover server-side routing and composition, we use Nginx. You’ll find the installation instructions in the first chapter that requires it.

Online resources

Purchase of *Micro Frontends in Action* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum, go to <https://livebook.manning.com/book/micro-frontends-in-action>. You can also learn more about Manning’s forums and the rules of conduct at <https://livebook.manning.com/#/discussion>.

Manning’s commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray! The forum and the archives of previous discussions will be accessible from the publisher’s website as long as the book is in print

about the author

MICHAEL GEERS is a software developer specializing in building user interfaces. He has written software for the web since he was a teenager. In the last few years, he has worked on various customer projects with verticalized architectures. He shares his experiences on this topic at international conferences, and in a series of magazine articles, and runs the site <https://micro-frontends.org>.

about the cover illustration

The figure on the cover of *Micro Frontends in Action* is captioned “Habitante de la Calabre,” or a Woman from Calabria. The illustration is taken from a collection of dress costumes from various countries by Jacques Grasset de Saint-Sauveur (1757-1810), titled *Costumes de Différents Pays*, published in France in 1797. Each illustration is finely drawn and colored by hand. The rich variety of Grasset de Saint-Sauveur’s collection reminds us vividly of how culturally apart the world’s towns and regions were just 200 years ago. Isolated from each other, people spoke different dialects and languages. In the streets or in the countryside, it was easy to identify where they lived and what their trade or station in life was just by their dress.

The way we dress has changed since then and the diversity by region, so rich at the time, has faded away. It is now hard to tell apart the inhabitants of different continents, let alone different towns, regions, or countries. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Grasset de Saint-Sauveur’s pictures.

Part 1

Getting started with micro frontends

F

rontend development has evolved a lot over the last decade. The web applications we are building today have to load quickly, run on a broad range of devices, and should react swiftly to user interactions. For a lot of businesses, the web frontend is the prime interaction surface for their users. So it's natural to put a lot of thought and attention to detail into its development.

When your project is small, and you're working with a handful of developers, building a nice web application is a straightforward task. However, if your business has a large web application and wants to improve and add new features continually, a single team will quickly be overwhelmed. This is where the micro frontend architecture comes in. There we slice the application into pieces that multiple teams can work on independently. In chapter 1, you'll learn the core concepts, understand the reasoning behind this architecture, and know what types of projects can benefit the most from it. In the second chapter, we'll jump right into the code and build a minimal viable micro frontends project from scratch: The Tractor Store. This e-commerce project functions as the basis for the more advanced techniques you'll unlock later in the book.

1

What are micro frontends?

This chapter covers

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

I've worked as a software developer on many projects over the last 15 years. In this time, I've had multiple chances to observe a pattern that repeats itself throughout our industry: working with a handful of people on a new project feels fantastic. Every developer has an overview of all functionality. Features get built quickly. Discussing topics with your coworkers is straightforward. This changes when the project's scope and the team size increases. Suddenly one developer can't know every edge of the system anymore. Knowledge silos emerge inside your team. Complexity rises—making a change on one part of the system may have unexpected effects on

other parts. Discussions inside the team are more cumbersome. Before, team members made decisions at the coffee machine. Now you need formal meetings to get everyone on the same page. Frederick Brooks described this in the book *The Mythical Man-Month* back in 1975. At some point, adding new developers to a team does not increase productivity.

Projects often are divided into multiple pieces to mitigate this effect. It became fashionable to divide the software, and thereby also the team structure, by technology. We introduced horizontal layers with a frontend team and one or more backend teams. Micro frontends describes an alternative approach. It divides the application into vertical slices. Each slice is built from the database to the user interface and run by a dedicated team. The different team frontends integrate in the customer's browser to form the final page. This approach is related to the microservices architecture. But the main difference is that a service also includes its user interface. This expansion of the service removes the need for a central frontend team. Here are the three main reasons why companies adopt a micro frontends architecture:

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

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

1.1 **The big picture**

Figure 1.1 is an overview of all the parts that are important when implementing micro frontends. Micro frontends are not a concrete technology. They're an alternative organizational and architectural approach. That's why we see a lot of different elements in this chart—like team structure, integration techniques, and other related topics. We'll go through the complete figure step by step. We start with the three teams above the dashed line and work our way up. When we reach the magic lamp at the top, we'll discuss *frontend integration*. At the bottom of this diagram, you can see the contents of this box zoomed in. It illustrates the three different aspects we need to address to create an *integrated application*. Our diagram journey ends at the three *shared topics* at the right.

1.1.1 **Systems and teams**

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

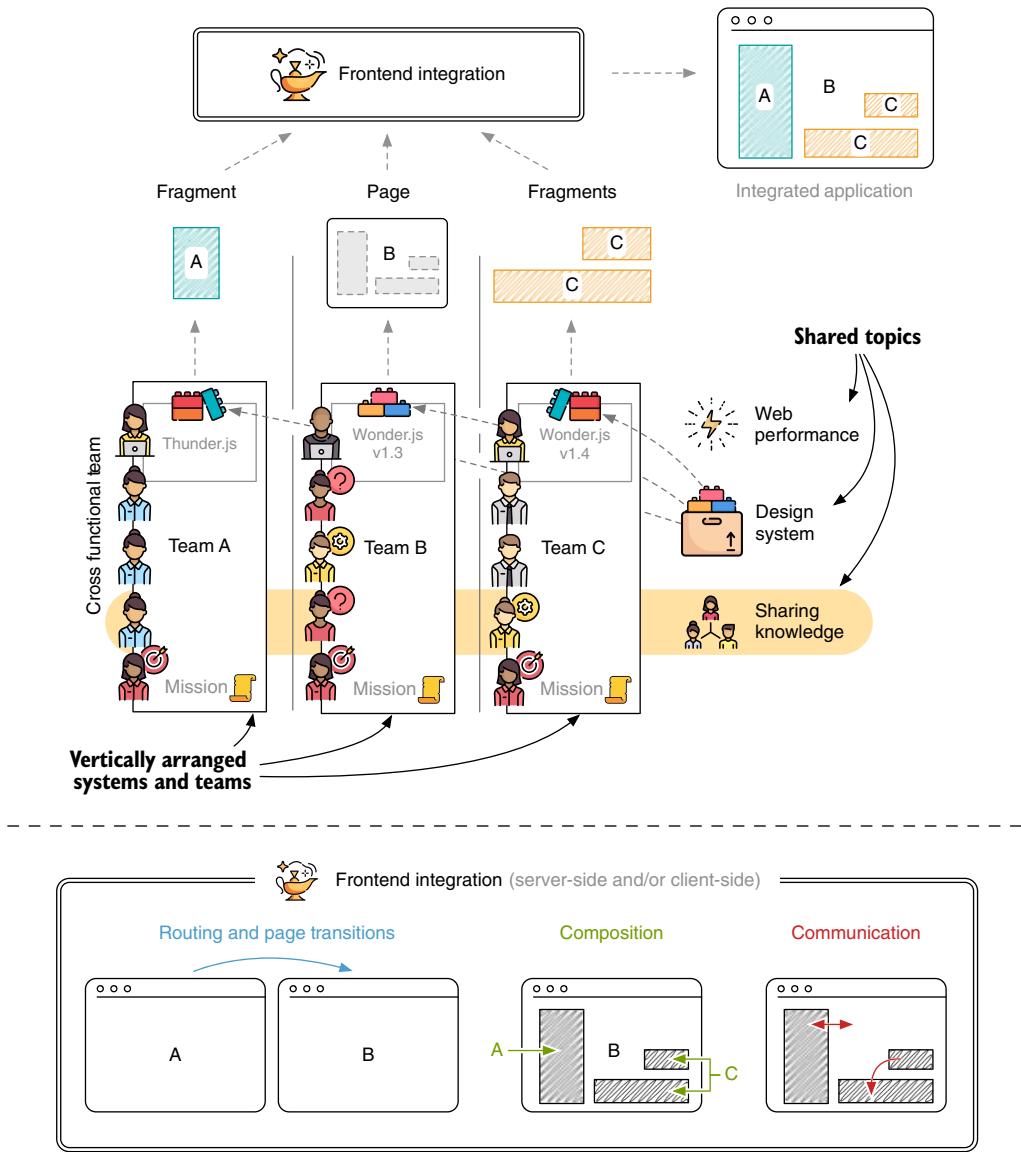


Figure 1.1 Here is the big picture overview of the micro frontends approach. The vertically arranged teams at the bottom are the core of this architecture. They each produce features in the form of pages or fragments. You can use techniques like SSI or Web Components to integrate them into an assembled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

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

TEAM MISSIONS

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

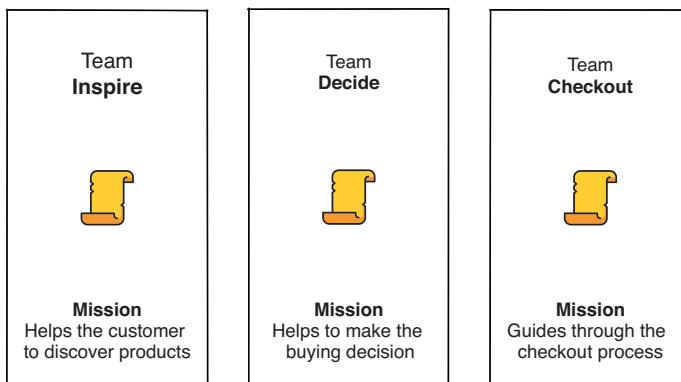


Figure 1.2 An e-commerce example with three teams. Each team works on a different part of the e-commerce shop and has its mission statement that clarifies their responsibility.

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

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

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

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

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

CROSS-FUNCTIONAL TEAMS

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

It feels natural at first sight, right? Frontend developers like to work with other frontend developers. They can discuss the bugs they are trying to fix or come up with ideas on how to improve a specific part of the code. The same is true for the other teams which specialize in a specific skill. Professionals strive for perfection and have

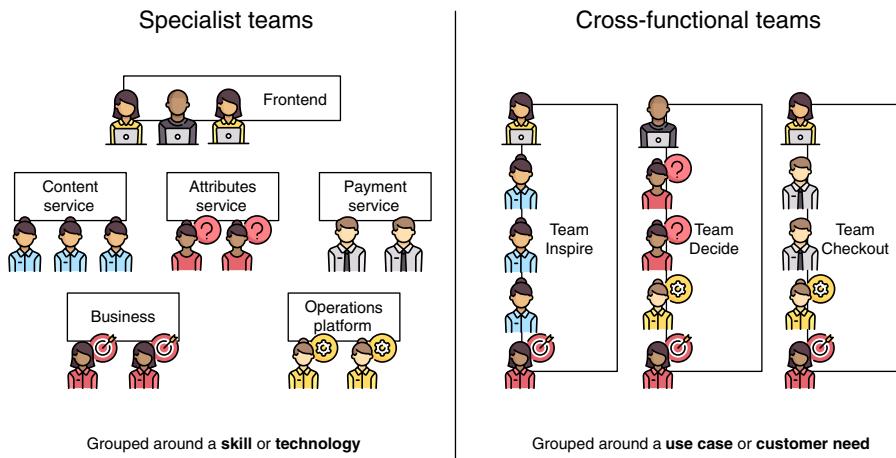


Figure 1.3 Team structure of a microservice-style architecture on the left compared with micro frontends teams on the right. Here the teams are formed around a customer need and not based on technologies like frontend and backend.

an urge to come up with the best solution in their field. When each team does a great job, the product as a whole will also be great, right?

This assumption is not necessarily valid. Building interdisciplinary teams is becoming more and more popular. You have a team where frontend and backend engineers, but also operations and business people, work together. Due to their different perspectives, they come up with more creative and effective solutions for the task at hand. These teams might not build the best-in-class operations platform or frontend layer, but they specialize in the team's mission. For example, they are working on becoming experts in presenting relevant product suggestions or building a seamless checkout experience. Instead of mastering a specific technology, they all focus on providing the best user experience for the area they work on.

Cross-functional teams come with the added benefit that all members are directly involved in feature development. In the microservice model, the services or operations teams are not involved directly. They receive their requirements from the layer above and don't always have the full picture of 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

Now we're getting to the aspect that makes the micro frontends approach different from other architectures. It's the way we think about and build features. Teams have end-to-end responsibility for a given functionality. They deliver the associated user

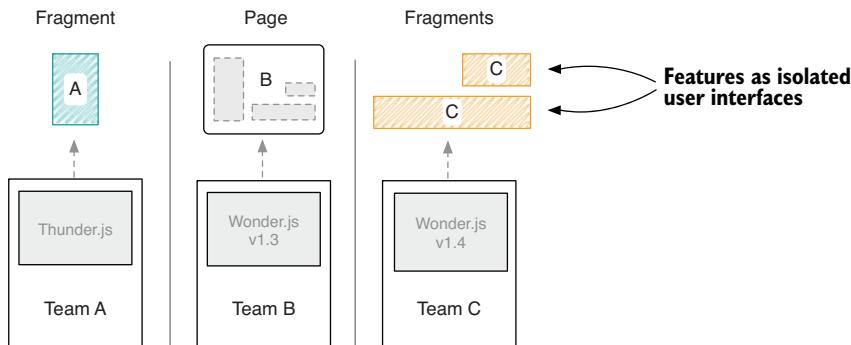


Figure 1.4 This is the middle portion of the big picture as detailed in its entirety in figure 1.1. Each team builds its own user interface as a page or a fragment.

interface as a micro frontend. A micro frontend can be a complete page or a fragment that other teams include. Figure 1.4 illustrates this.

A team generates the HTML, CSS, and JavaScript necessary for a given feature. To make life easier, they might use a JavaScript library or framework to do that. Teams don't share library and framework code. Each team is free to choose the tool that fits best for their use case. The imaginary frameworks *Thunder.js* and *Wonder.js* illustrate that.¹ Teams can upgrade their dependencies on their own. Team B uses *Wonder.js* v1.3, whereas Team C already switched to v 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 figure 1.5.

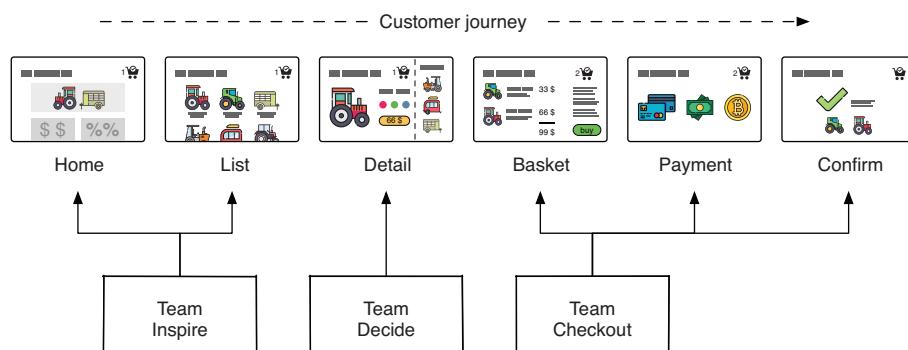


Figure 1.5 Each page is owned by one team.

¹ Yes, I'm aware that there probably is a JavaScript framework for all dictionary words registered on npmjs.org, including *Thunder* and *Wonder*. But since both projects have over six years of inactivity and single-digit weekly downloads, let's stick to them. :)

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

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

- Teams can work autonomously in their field of expertise.
- Teams can choose the technology stack that fits best for the job at hand.
- The applications are loosely coupled and only integrate in the frontend (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 re-implement them. This is where *fragments* come in.

A page often serves more than one purpose, and might show information or provide functionality that another team is responsible for. In figure 1.6, you see the product page of *The Tractor Store*. Team Decide owns this page. But not all of the functionality and content can be provided by them.

The Recommendations block on the right is an inspirational element. Team Inspire knows how to produce those. The Mini Basket at the bottom shows all selected items. Team Checkout implements the basket and knows its current state. The customer can add a new tractor to the basket by clicking the Buy button. Since this action modifies the basket, Team Checkout also provides this button as a fragment.

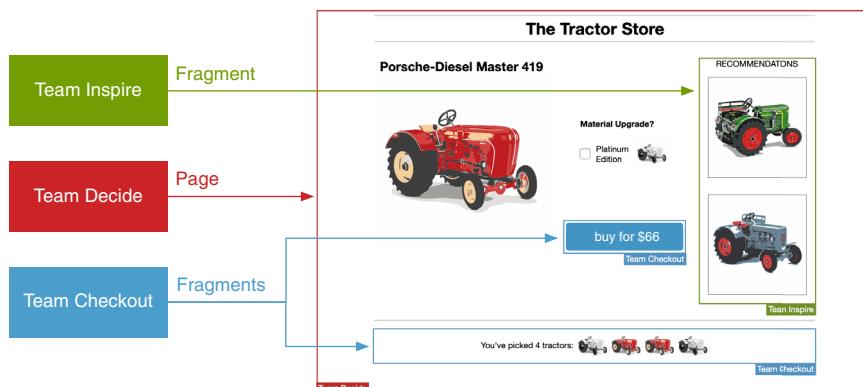


Figure 1.6 Teams are responsible for pages and fragments. You can think of fragments as embeddable mini applications that are isolated from the rest of the page.

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

1.1.3 Frontend integration

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

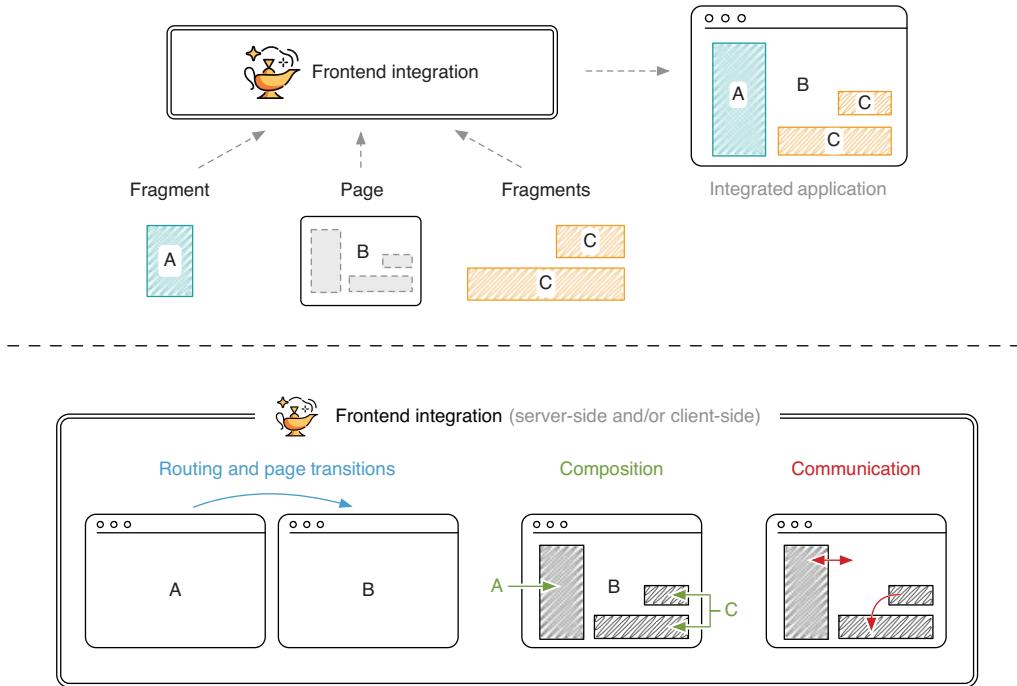


Figure 1.7 The term *frontend integration* describes a set of techniques you use to assemble the user interfaces (pages and fragments) of the teams into an integrated application. You can group these techniques into three categories: routing, composition, and communication. Depending on your architectural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

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

ROUTING AND PAGE TRANSITIONS

Here we are talking about integration on page level. We need a system to get from a page owned by Team A to a page owned by Team B. The solutions can be straightforward. You can achieve this by merely using an *HTML link*. If you want to enable

client-side navigation, which renders the next page without having to do a reload, it gets more sophisticated. You can implement this by having a shared *application shell* or using a meta-framework like *single-spa*. We will look into both options in this book.

COMPOSITION

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

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

- 1 *Server-side composition*, for example with SSI, ESI, Tailor or Podium
- 2 *Client-side composition*, for example with iframes, Ajax, or Web Components

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

COMMUNICATION

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

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

1.1.4 Shared topics

The micro frontends architecture is all about being able to work in small autonomous teams that have everything they need to create value for the customer. But some shared topics are essential to address when working like this (figure 1.8).

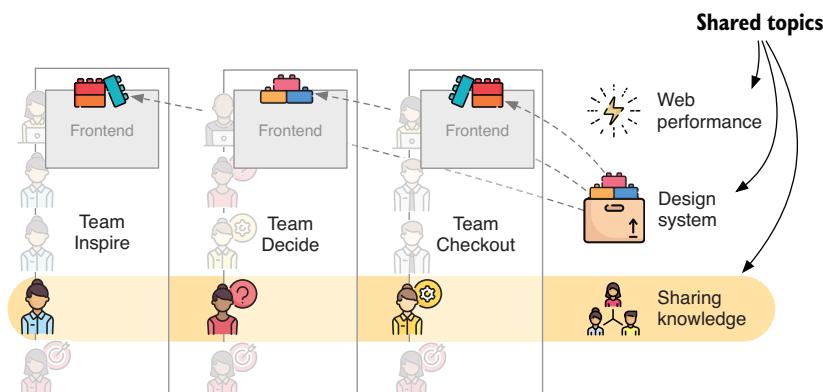


Figure 1.8 To ensure a good end result and avoid redundant work, it's important to address topics like web performance, design systems, and knowledge sharing from the start.

WEB PERFORMANCE

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

DESIGN SYSTEMS

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

SHARING KNOWLEDGE

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

1.2 **What problems do micro frontends solve?**

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

1.2.1 **Optimize for feature development**

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

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

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

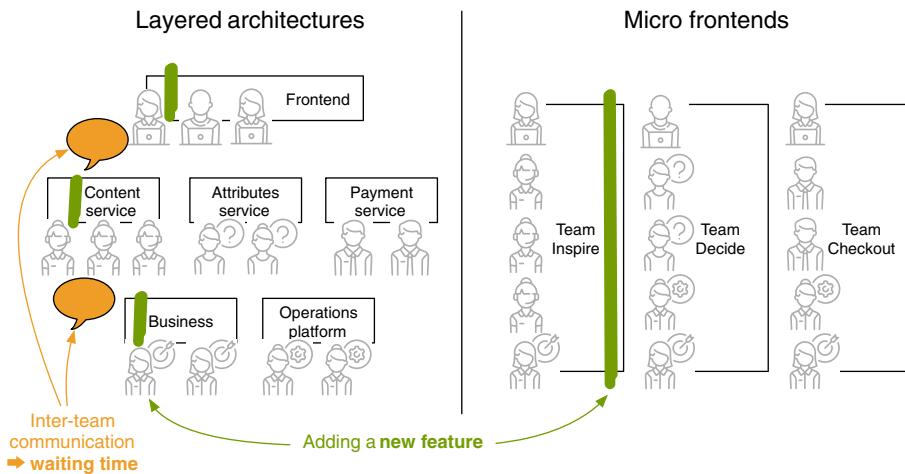


Figure 1.9 This diagram shows what it takes to build a new feature. On the left side, you see a layered architecture. Three teams are involved in building it. These teams have to coordinate and potentially wait for each other. With the micro frontends approach (right), one team can build this feature.

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

1.2.2 No more frontend monolith

Most architectures today don't have a concept for scaling frontend development. In figure 1.10 you see three architectures: the monolith, frontend/backend-split, and

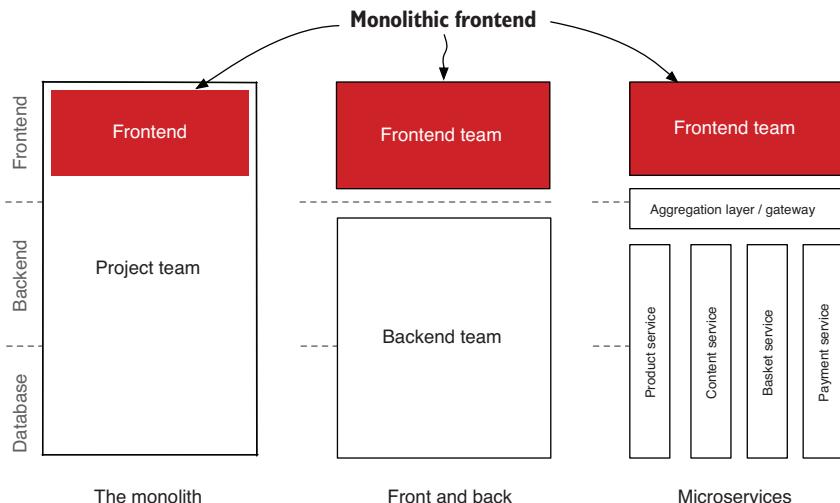


Figure 1.10 In most architectures, the frontend is a monolithic system.

microservices. They all come with a monolithic frontend. That means the frontend comes from a single codebase that only one team can work on sensibly.

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

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

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

1.2.3 **Be able to keep changing**

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

But a lot has changed since then. Frontend development transformed from “making the HTML pretty with CSS” to a professional field of engineering. To deliver good work, a web developer nowadays needs to know topics like responsive design, usability, web performance, reusable components, testability, accessibility, security, and the changes in web standards and their browser support. The evolution of frontend tools, libraries, and frameworks enabled us to build higher-quality and more capable web applications to meet the rising expectations of our users. Tools like Webpack, Babel, Angular, React, Vue.js, Stencil, and Svelte play a vital role today, but, likely, we haven't reached the end of this evolution yet. Being able to adopt a new technology when it makes sense is an essential asset for your teams and your company.

LEGACY

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

- *GitHub* did a multi-year migration to remove their dependency on jQuery.²
- *Trivago*, a hotel search engine, made an enormous effort with Project Ironman to rework their complex CSS to a modular design system.³

² See “Removing jQuery from GitHub.com frontend,” *The GitHub Blog*, <https://github.blog/2018-09-06-removing-jquery-from-github-frontend/>.

³ See Christoph Reinartz, “Large Scale CSS Refactoring at trivago,” *Medium*, <http://mng.bz/gynn>.

- Etsy is getting rid of their JavaScript legacy baggage to reduce bundle size and increase web performance. The code has grown over the years, and one developer can't have an overview of the complete system. To identify dead code, they've built an in-browser code coverage tool that runs in the customer's browser and reports back to their servers.⁴

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

LOCAL DECISION MAKING

Being able to introduce and verify a technology in an isolated part of your application without having to come up with a grand migration plan for everything is a valuable asset. The micro frontends approach enables this on a team level. Here is an example: Team Checkout is experiencing a lot of JavaScript runtime errors lately, due to references to undefined variables. Since it's crucial to have a checkout process that's as bug-free as possible, the team decides to switch to Elm, which is a statically typed language that compiles to JavaScript. The language is designed to make it impossible to create runtime errors. But it also comes with drawbacks. Developers have to learn the new language and its concepts. The open source ecosystem of available modules or components is still small. But for the use case of Team Checkout, the pros outweigh the cons.

With the micro frontends approach, teams are in full control of their technology stack (*micro architecture*). This autonomy enables them to make the decision and switch horses. They don't have to coordinate with other teams. The only thing they have to ensure is that they stay compatible with the previously agreed upon inter-team conventions (*macro architecture*). (See figure 1.11.) These might include adhering to

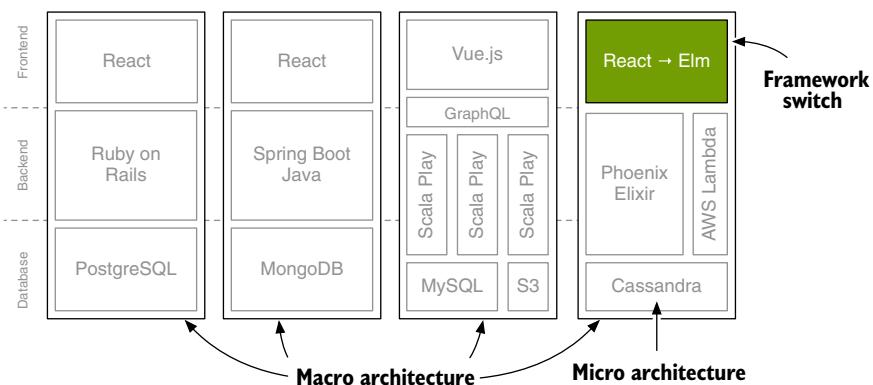


Figure 1.11 Teams can decide about their internal architecture (micro architecture) on their own as long as they stay in the boundaries of the agreed upon macro architecture.

⁴ See “Raiders of the Fast Start: Frontend Perf Archeology, <http://mng.bz/5aVD>.

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

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 trade-offs might not be the same in different parts of the application. The process of making a decision at this scale is often so painful, unproductive, and tiresome that most developers shy away from bringing it up in the first place. The micro frontends approach makes it easier to evolve your application over time in the areas where it makes sense.

1.2.4 The benefits of independence

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

SELF-CONTAINED

Pages and fragments are self-contained. That means they bring their own markup, styles, and scripts, and should not have shared runtime dependencies. This isolation makes it possible for a team to deploy a new feature in a fragment without having to consult with other teams first. An update may also come with an upgraded version of the JavaScript framework they are using. Because the fragment is isolated, this is not a big deal. (See figure 1.12.)

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

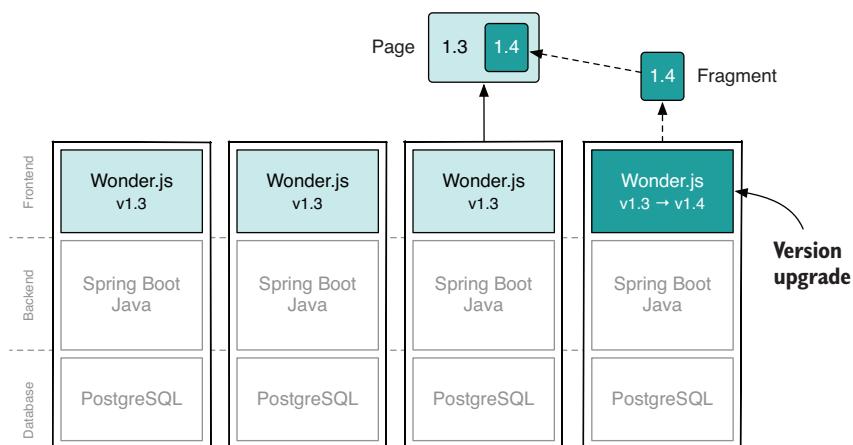


Figure 1.12 Fragments are self-contained and upgradeable independently of the page they are embedded in.

TECHNICAL OVERHEAD

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

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

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

SHARED NOTHING

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

The goal is to share as little as possible to enable faster feature development. Every shared piece of code or infrastructure has the potential for creating a non-trivial amount of management overhead. This approach is also called *shared nothing architecture*. The *nothing* sounds a bit harsh, and in reality, it's not that black and white. But in general, micro frontend projects have a strong tendency to accept redundancy in favor of more autonomy and higher iteration speeds. We'll touch on this principle at various points in this book.

1.3 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.

1.3.1 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 set up and maintain its own application server, build process and continuous integration pipeline, and might ship redundant JavaScript/CSS code to the browser. Here are two examples where this is an issue:

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

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

1.3.2 Consistency

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

1.3.3 Heterogeneity

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

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

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

1.3.4 More frontend code

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

1.4 When do micro frontends make sense?

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

1.4.1 Good for medium-to-large projects

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

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

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

1.4.2 Works best on the web

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

NATIVE MONOLITH

Native applications for controlled platforms like iOS or Android are monolithic by design. Composing and replacing functionality on the fly is not possible. For updating a native app, you have to build a single application bundle that's then submitted to Apple's or Google's review process. A way around this is to load parts of the application from the web. Embedded browsers or WebViews can help to keep the native part

⁵ See Janet Choi, "Why Jeff Bezos' Two-Pizza Team Rule Still Holds True in 2018," *I Done This Blog*, <http://blog.idonethis.com/two-pizza-team/>.

of the app to a minimum. But when you have to implement native UI, it's hard to have multiple end-to-end teams working on it without stepping on each other's toes.

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

MULTIPLE FRONTENDS PER TEAM

A team is also not limited to only one frontend. In e-commerce, it's common to 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, for example, also make the associated help desk functionality for the customer hotline. They might also build the WebView-based version of the checkout that a native app can embed.

1.4.3 **Productivity versus overhead**

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

SETUP

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

ORGANIZATIONAL COMPLEXITY

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

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

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

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

1.4.4 **Where micro frontends are not a great fit**

But of course, micro frontends are not perfect for every project. As stated earlier, they are a solution for scaling development. If you only have a handful of developers

and communication is no issue, the introduction of micro frontends won't bring much value.

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

I've spoken to people working in startups that have tried this model. Everything worked fine up until the point the company needed to pivot its business model. It is of course possible to reorganize the teams and the associated software, but it creates a lot of friction and extra work. Other organizational approaches are more flexible.

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

1.4.5 Who uses micro frontends?

The concepts and ideas described here are not new. Amazon does not talk a lot about its internal development structure. However, several Amazon employees reported that their e-commerce site has been developed like this for many years now. Amazon also uses a UI integration technique that assembles the different parts of the page before it reaches the customer.

Micro frontends are indeed quite popular in the e-commerce sector. In 2012 the Otto Group,⁶ a Germany-based mail-order company and one of the world's largest e-commerce players, started to split up its monolith. The Swedish furniture company IKEA⁷ and Zalando,⁸ one of Europe's 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. Spotify¹⁰ organizes itself in autonomous end-to-end teams called *Squads*. SAP published a framework¹¹ to integrate different applications. Sports streaming service DAZN¹² also rebuilt their monolithic frontend as a micro frontends architecture.

Summary

- Micro frontends are an architectural approach and not a specific technique.
- Micro frontends remove the team barrier between frontend and backend developers by introducing cross-functional teams.

⁶ See "On Monoliths and Microservices," <http://mng.bz/6Qx6>.

⁷ See Jan Stenberg, "Experiences Using Micro Frontends at IKEA," *InfoQ*, <http://mng.bz/oPgv>.

⁸ Project Mosaic | Microservices for the Frontend, <https://www.mosaic9.org/>.

⁹ See Markus Gruber, "Another One Bites the Dust" (written in German), <http://mng.bz/nPa4>.

¹⁰ See "Spotify engineering culture," <http://mng.bz/vx7r>.

¹¹ SAP Luigi, <https://luigi-project.io>.

¹² See "DAZN—Micro Frontend Architecture," <http://mng.bz/4ANv>.

- 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 therefore 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.
- Setting the team boundaries along the user journey and customer needs is a good pattern.
- A team should have a clear mission like "Help the customer to find the product they are looking for."
- A team can own a complete page or deliver a piece of functionality via a fragment.
- A fragment is a mini-application that is self-contained, which means it brings everything it needs with it.
- The micro frontends model typically comes with more code for the browser. It's vital to address web performance from the start.
- There are multiple frontend integration techniques. They work either on the client or the server.
- 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 company's domain well. Changing responsibilities afterward works but creates friction.

My first micro frontends project



This chapter covers

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

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

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

In this chapter, we'll also build the foundation of our example project *The Tractor Store*. We'll expand on this project throughout the book. You will learn different integration techniques and their benefits and drawbacks. Spoiler alert: there is no “gold standard” or “best integration technique.” It's all about making the right trade-offs for your use case. But this book will highlight the different aspects and properties you should look for when picking a technique. We'll start with simple scenarios in this chapter and work our way through more sophisticated ones after that.

2.1 ***Introducing The Tractor Store***

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

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

To achieve maximum flexibility in development, the company decided to build the software from scratch and not go with an off-the-shelf solution. The company wants to evaluate their ideas and features quickly. That is why they decided to go with the micro frontends architecture. Multiple teams can work in parallel, independently build new features, and validate ideas. They are starting with two teams.

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

2.1.1 ***Getting started***

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

FREEDOM OF CHOOSING TECHNOLOGY

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

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

Technology diversity and blueprints

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

It can also save up-front costs because you could implement the basic application setup, including folder structure, error reporting, form handling, or the build process once. Every team can then copy this blueprint application and build on it. This way, teams can get productive a lot quicker, and the software stacks are more similar. In chapter 13, we'll go deeper into this topic.

INDEPENDENT DEPLOYS

Both teams create their own source code repository and set up a continuous integration pipeline. This pipeline runs every time a developer pushes new code to the central version control system. It builds the software, runs all kinds of automated tests to ensure the software's correctness, and deploys the new version of the application to the team's production server. These pipelines run independently. A software change in Team Decide will never cause Team Inspire's pipeline to break. (See figure 2.1.)

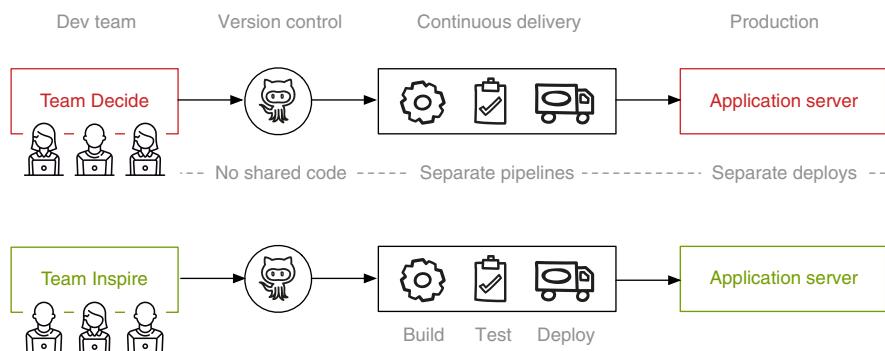


Figure 2.1 Teams work in their own source code repository, have separate integration pipelines, and can deploy independently.

2.1.2 Running this book's example code

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

TIP You can browse the source code of this book on GitHub¹ or download a ZIP from the Manning website². If you don't want to run the code locally, you can go to <https://the-tractor.store>. There you can see and inspect all examples directly in your browser.

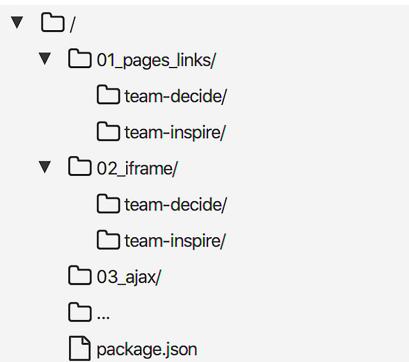


Figure 2.2 The directory structure of the example code bundle. The main directory (shown as /) contains a sub-folder for each example project. The top-level package .json file contains the run commands for all examples.

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

INSTALL DEPENDENCIES

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

```
npm install
```

STARTING AN EXAMPLE

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

```
npm run 01_pages_links
```



¹ Sample code on GitHub: <http://mng.bz/QyOQ>.

² *Micro Frontends in Action*, <http://mng.bz/XPjp>.

Each run command performs three actions:

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

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

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

Your terminal output should look like this:

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

[decide ] INFO: Accepting connections at http://localhost:3001
[inspire] INFO: Accepting connections at http://localhost:3002
[2] wait-on http://localhost:3001/product/porsche && opener
    http://localhost:3001/product/porsche exited with code 0"
[decide ] :3001/product/porsche
[decide ] :3001/static/page.css
[decide ] :3001/static/outlines.css
```

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

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

Opening the example page in your default browser

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

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

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

2.2 Page transition via links

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

2.2.1 Data ownership

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

Table 2.1 Team Decide's product database

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

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

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

Table 2.2 Team Inspire's recommendations

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

Team Decide doesn't have to know anything about these relations. Nor do they need to know about the underlying algorithms and data sources.

2.2.2 Contract between the teams

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

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

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

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

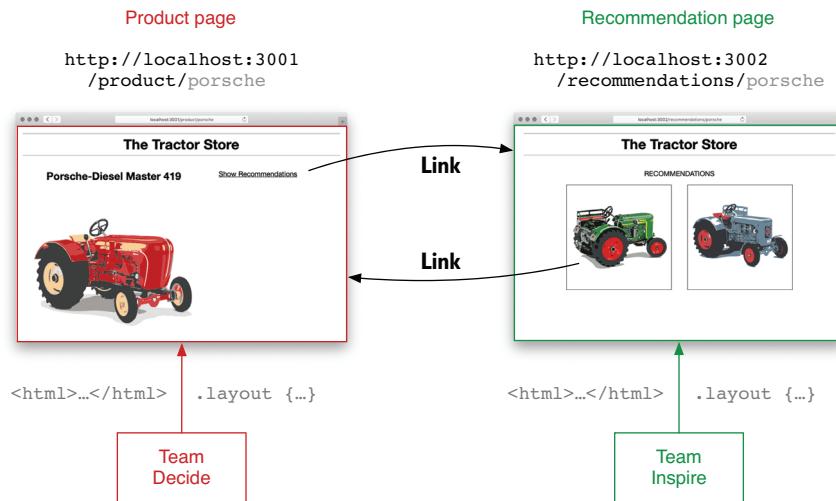


Figure 2.3 A product and recommendations page connected via links

Let's take a quick look at the code that's involved in making this happen.

2.2.3 How to do it

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

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

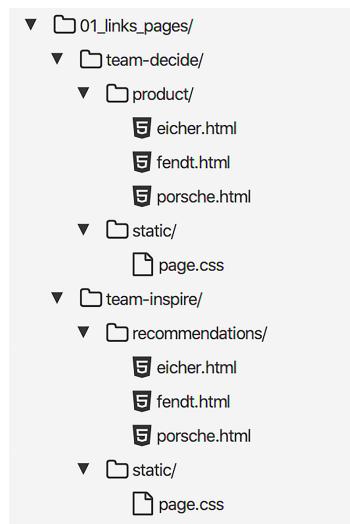


Figure 2.4 A product and recommendations page connected via links

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

MARKUP

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

Listing 2.1 team-decide/product/porsche.html

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

The link to Team Inspire's
matching recommendation page

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

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

Listing 2.2 team-inspire/recommendations/porsche.html

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

Links to Team Decide's
product pages

```

</div>
</body>
</html>

```

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

STYLES

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

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

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

STARTING THE APPLICATIONS

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

```
npm run 01_pages_links
```



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

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.

<http://localhost:3001/product/porsche>

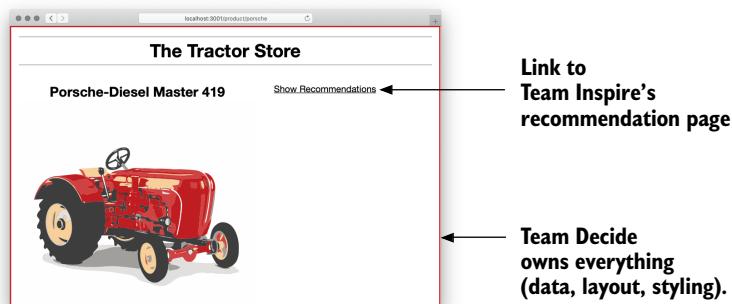


Figure 2.5 Team Decide’s product detail page. The team owns everything on this page.

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

2.2.4 Dealing with changing URLs

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

A team that wants to change its URL could provide an HTTP redirect to solve this. However, letting your end users jump through redirect chains is not always optimal. A more robust mechanism that has proven valuable for the projects we've worked on is that every team provides a machine-readable directory of all their URL patterns. A JSON file in a known location usually does the trick. This way, all applications can look up the URL patterns regularly and update their links if needed. Standards like URI templates,³ json-home,⁴ or Swagger OpenAPI⁵ can help here.

2.2.5 The benefits

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

LOOSE COUPLING

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

HIGH ROBUSTNESS

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

2.2.6 The drawbacks

The fact that the teams share nothing does come with a cost. An integration via links only is not always optimal from the user's point of view. They have to click a link to see

³ See <https://tools.ietf.org/html/rfc6570>.

⁴ See "Home Documents for HTTP APIs," <https://mnnot.github.io/I-D/json-home/>.

⁵ See the Swagger OpenAPI Specification, <https://swagger.io/specification/>.

the information that is owned by another team. In our case, the user bounces between the product and the recommendation page. With this simple integration, we have no way of combining data from two different teams into one view.

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

2.2.7 When do links make sense?

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

2.3 Composition via iframe

The whole company staff is pleased about the progress both teams made in this short amount of time. But everyone agrees that we have to improve the user experience. Discovering new tractors via the Show Recommendations link works, but is not obvious enough for the customer. Our 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.

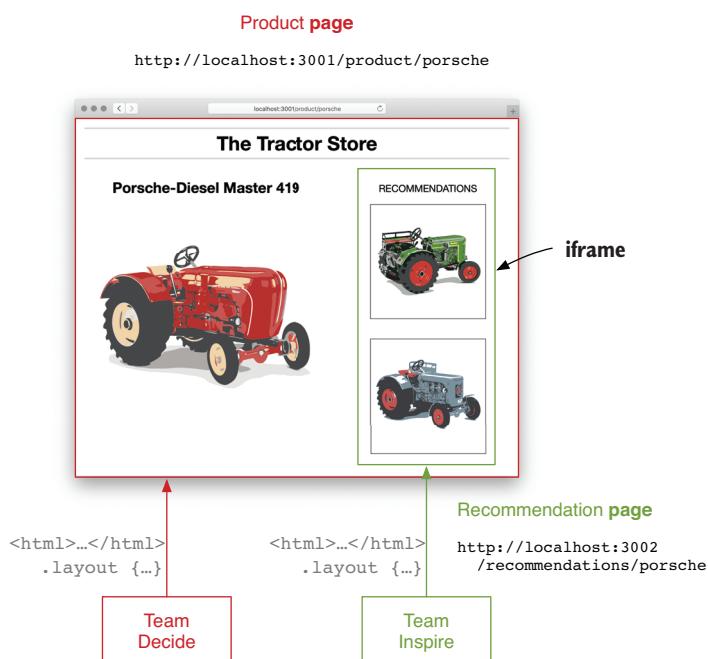


Figure 2.6 Integrating the recommendation page into the product page via iframe. These pages don't share anything. Both are standalone HTML documents with their own styling.

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

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

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

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

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, as the following code shows.

Listing 2.3 team-decide/product/porsche.html

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

After that, Team Inspire removes the “The Tractor Store” header from the recommendation page’s markup because we don’t need it in the iframe.

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

```
npm run 02_iframe
```



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

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

Listing 2.4 team-decide/static/page.css

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

Remove the browser's default iframe border.

Fixed height to make enough space for the content

Iframe should be as wide as its parent container.

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

Another issue is that Team Inspire is now bound to the height Team Decide has defined. They can’t, for example, experiment by adding a third recommendation image without having to talk to the other team. JavaScript libraries⁶ exist to automatically update the *iframe* size when its content changes.

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

2.3.2 **The benefits**

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

2.3.3 **The drawbacks**

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

LAYOUT CONSTRAINTS

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

PERFORMANCE OVERHEAD

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

BAD FOR ACCESSIBILITY

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

BAD FOR SEARCH ENGINES

When it comes to search engine optimization (SEO), *iframes* also have a bad reputation. A crawler would index our product page as two distinct pages: the outer page and the included inner page. The search index does not represent the fact that one

⁶ See *iframe-resizer*, <https://github.com/davidjbradshaw/iframe-resizer>.

includes the other. Our page would not show up for the search term “tractor recommendations.” The user sees both words in their browser window, but these words do not exist in the same document.

2.3.4 When do iframes make sense?

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

Spotify, for example, implemented a micro frontends architecture early on for their desktop application.⁷ Their integration technique relied on using iframes for the different parts of the application. Since the overall layout of their application is quite static, and search engine indexing is not an issue, this was an acceptable trade-off for them.

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

2.4 What’s next?

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

Take a look at the integration techniques in figure 2.7. You’ve already seen these three types of integration in the big-picture diagram in chapter 1.

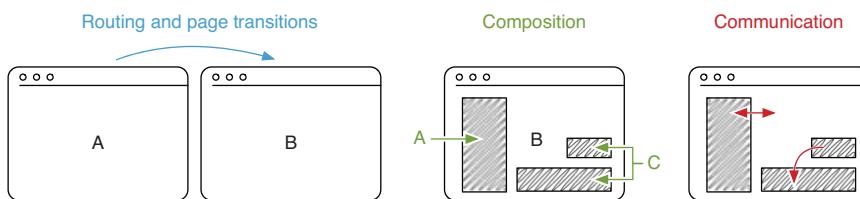


Figure 2.7 We can divide *frontend integration techniques* into three categories: routing, composition, and communication.

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

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

⁷ See Mattias Peter Johansson, “How is JavaScript used within the Spotify desktop application?” Quora, <http://mng.bz/Mdmm>.

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

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

Summary

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

Part 2

Routing, composition, and communication

Now you know the basics of the micro frontends architecture. In part 2 of this book, we'll do a deep dive into the techniques you'll need to build a more sophisticated project. Most of these techniques don't require special tools.

Adoption of the micro frontends approach is spreading in the software industry. That's why we see a lot of meta-frameworks and helper libraries being open-sourced by businesses and individuals. These tools address common pain points and provide additional abstractions to improve the developer experience. Since this support software landscape is still shifting, we won't go deep into any of these solutions. However, we'll touch on a few of them along our journey.

Throughout the following chapters, we'll focus on existing web standards and leverage native browser features wherever possible. This stick-to-the-fundamentals approach has proven stable and valuable in the projects I've been a part of over recent years. Understanding the core concepts is vital for building a successful project—even if you decide to pick up a micro frontends library later.

In chapters 3–8, you'll learn techniques for routing, composition, and communication, both for server-side and client-side rendered web applications. I've arranged the chapters by complexity. We start with simple techniques and work our way up to more sophisticated ones. Chapter 9 is an architecture overview that puts the learned techniques into context. It helps you to make the right architectural decision for your next project.



Composition with Ajax and server-side routing

This chapter covers

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

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

3.1 Composition via Ajax

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

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

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

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

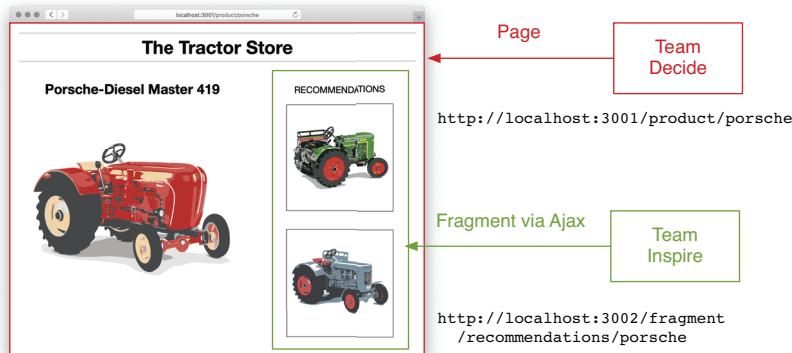


Figure 3.1 Integrating the recommendations into the product page's DOM via Ajax

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

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

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

3.1.1 How to do it

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

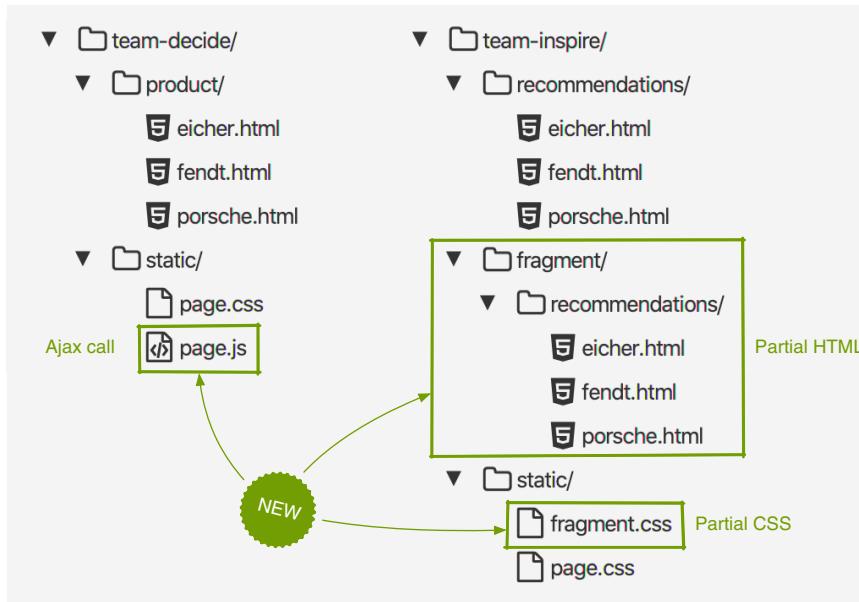


Figure 3.2 Folder structure of the Ajax example code 03_ajax.

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

MARKUP

The fragment markup looks like this.

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

```

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

Reference to the
recommendation styles

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

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

AJAX REQUEST

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

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

Listing 3.2 team-decide/static/page.js

```
const element = document.querySelector(".decide_recos");
const url = element.getAttribute("data-fragment"); ← Finding the element to
                                                insert the fragment in

window
  .fetch(url) ← Retrieving the fragment URL from an attribute
  .then(res => res.text())
  .then(html => {
    element.innerHTML = html; ← Inserting the loaded markup
  });
```

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

Listing 3.3 team-decide/view.js

```
...
<aside
  class="decide_recos"
  data-fragment="http://localhost:3002/fragment/recommendations/porsche" ← Team Inspire's
>                                         recommendation fragment URL
  <a href="http://localhost:3002/recommendations/porsche">
    Show Recommendations
  </a>
</aside>
<script src="/static/page.js" async></script> ← Link to the recommendation
                                                page. In case the Ajax call
                                                failed or hasn't finished
                                                yet, the customer can use
                                                this link as a fallback:
                                                Progressive Enhancement.
</body>
...

```

Referencing the JavaScript file, which will make the Ajax request

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

```
npm run 03_ajax
```



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

3.1.2 Namespacing styles and scripts

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

ISOLATING STYLES

Let's look at CSS first. Sadly, browsers don't offer much help here. The deprecated Scoped CSS specification would have been an excellent fit for our use case. It allowed you to mark a style or link tag with the attribute `scoped`. The effect was that these styles would only be active in the DOM subtree they're defined in. Styles from higher up in the tree would still propagate down, but styles from within a scoped block would never leak out. This specification did not last long, and browsers which already supported it pulled their implementation.¹ 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 browsers² it's possible to get strong style scoping today via JavaScript and the ShadowDOM API, which is part of the Web Components specification. We'll talk about this in chapter 5.

Since CSS rules are global by nature, the most practical solution is to namespace all CSS selectors. Many CSS methodologies like BEM³ use strict naming rules to avoid unwanted style leaking between components. But two different teams might come up with the same component name independently, like the headline component in our example. That's why it's a good idea to introduce an extra team-level prefix. Table 3.1 shows what this namespacing might look like.

Table 3.1 Namespacing all CSS selectors with a team prefix

Team name	Team prefix	Example selectors
Decide	decide	.decide_headline .decide_recos
Inspire	inspire	.inspire_headline .inspire_recommendation_item
Checkout	checkout	.checkout_minicart .checkout_minicart-empty

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

¹ See Arly BcBlain, "Saving the Day with Scoped CSS," *CSS-Tricks*, <https://css-tricks.com/saving-the-day-with-scoped-css/>.

² See "Can I Use Shadow DOM?" <https://caniuse.com/#feat=shadowdomv1>.

³ BEM, <http://getbem.com/naming/>.

When every team follows these naming conventions and only uses class-name-based selectors, the issue of overwriting styles should be solved. Prefixing does not have to be done manually. Some tools can help here. CSS Modules, PostCSS, or SASS are a good start. You can configure most CSS-in-JS solutions to add a prefix to each class name. It does not matter which tool a team chooses, as long as all selectors are prefixed.

ISOLATING JAVASCRIPT

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

A popular way is to wrap your script in an *IIFE* (*immediately invoked function expression*).⁴ This way, the declared variables and functions of your application are not added to the global window object. Instead, we limit the scope to the anonymous function. Most build tools already do this automatically. For the `static/page.js` of Team Decide it would look like this.

Listing 3.4 `team-decide/static/page.js`

```
(function () {
  const element = ...;
  ...
})();
```

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

Instead of writing this

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

you could express it declaratively and avoid creating a global variable:

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

Accessing the data can be done by looking up the `script` tag in your part of the DOM tree and parsing it:

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

⁴ See <http://mng.bz/Edoj>.

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

Table 3.2 Some JavaScript functionalities also need namespacing.

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

Namespacing helps with more than just avoiding conflicts. Another valuable factor in day-to-day work is that they also indicate ownership. When an enormous cookie value leads to an error, you just have to look at the cookie name to know which team can fix that.

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

3.1.3 Declarative loading with *h-include*

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

The JavaScript library *h-include* provides a declarative approach for fragment loading.⁵ Including a fragment feels like including an iframe in the markup. You don't have to care about finding the DOM element and making the actual HTTP request. The library introduces a new HTML element called `h-include`, which handles everything for you. The code for the recommendations would look like this.

Listing 3.5 `team-decide/product/porsche.html`

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

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

⁵ See <https://github.com/gustafnk/h-include>.

```
</h-include>
</aside>
...
```

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

3.1.4 **The benefits**

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

NATURAL DOCUMENT FLOW

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

SEARCH ENGINES AND ACCESSIBILITY

Even though integration happens in the browser and the fragment is not present in the page's initial markup yet, this model works well for search engines. Their bots execute JavaScript and index the assembled page.⁶ Assistive technologies like screen readers also support this. It's essential, though, that the combined markup semantically makes sense as a whole. So make sure that your content hierarchy is marked up correctly.

PROGRESSIVE ENHANCEMENT

An Ajax-based solution typically plays well with the principles of progressive enhancement.⁷ Delivering server-rendered content as a fragment or as a standalone page doesn't introduce a lot of extra code.

You can provide a reliable fallback in case JavaScript failed or hasn't executed yet. On our product page, users with broken JavaScript will see the Show Recommendations link, which will bring them to the standalone recommendations page. Architecting for failure is a valuable technique that will increase the robustness of your application. I recommend checking out Jeremy Keith's publications⁸ for more details on progressive enhancement.

FLEXIBLE ERROR HANDLING

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

⁶ For details on the Googlebot's JavaScript support, see <https://developers.google.com/search/docs/guides/rendering>.

⁷ See https://en.wikipedia.org/wiki/Progressive_enhancement.

⁸ See <https://resilientwebdesign.com/>.

3.1.5 The drawbacks

The Ajax model also has some drawbacks. The most obvious one is already present in its name: it's asynchronous.

ASYNCHRONOUS LOADING

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

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

MISSING ISOLATION

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

SERVER REQUEST REQUIRED

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

NO LIFECYCLE FOR SCRIPTS

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

The team that owns the fragment must know when their code should run. There are multiple ways to implement this. MutationObserver,⁹ annotation via data-* attributes, custom elements, or custom events can help here. But you have to implement these mechanisms manually. In chapter 5 we'll explore how Web Components can help here.

⁹ See Louis Lazaris, "Getting To Know The MutationObserver API," *Smashing Magazine*, <http://mng.bz/Mdao>.

3.1.6 When does an Ajax integration make sense?

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

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

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

3.1.7 Summary

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

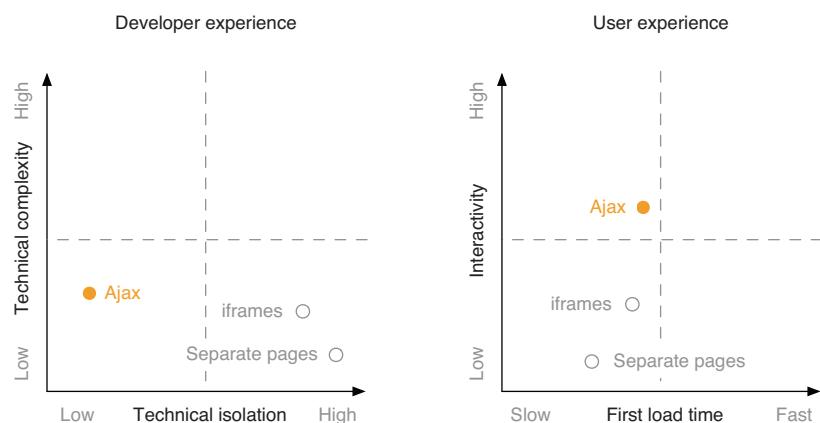


Figure 3.3 Comparison of different integration techniques. Compared to the iframes or links approach, it's possible to build more performant and usable solutions with Ajax. But you lose technical isolation and need to rely on inter-team conventions like using CSS prefixes.

I decided to compare them along four properties:

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

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

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

3.2 Server-side routing via Nginx

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

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

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

The server routes all requests to the responsible application. It does not contain any business logic besides this. This routing web server is often called a *frontend proxy*. Each team should receive its own path prefix. The frontend proxy should route all

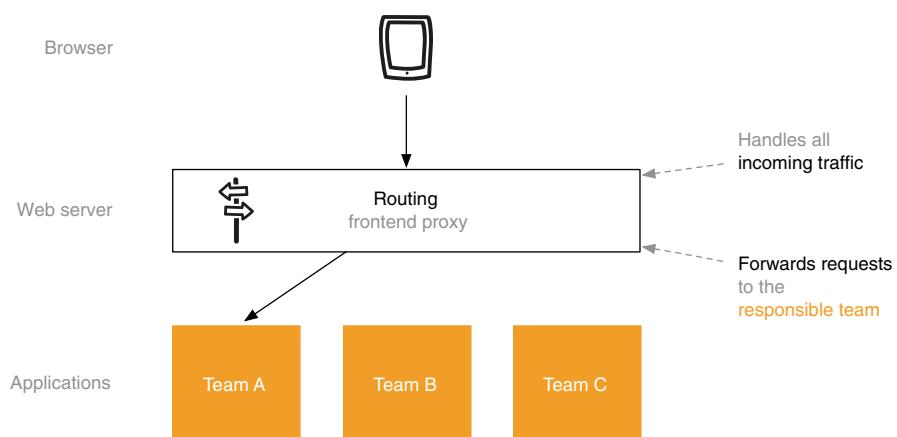


Figure 3.4 The shared web server is inserted between the browser and the team applications. It acts as a proxy and forwards the requests to the responsible teams.

requests starting with `/decide/` to Team Decide's server. They also require additional routing rules. The frontend proxy passes all requests starting with `/product/` to Team Decide; the ones with `/recommendations/` go to Team Inspire.

In our development environment, we again use different port numbers instead of configuring actual domain names. The frontend proxy we will set up listens on port 3000. Table 3.2 shows the routing rules our frontend proxy should implement.

Table 3.3 Frontend proxy routes incoming requests to the teams applications

Rule #	Path prefix	Team	Application
per team prefixes (default)			
#1	<code>/decide/</code>	Decide	<code>localhost:3001</code>
#2	<code>/inspire/</code>	Inspire	<code>localhost:3002</code>
per page prefixes (additional)			
#3	<code>/product/</code>	Decide	<code>localhost:3001</code>
#4	<code>/recommendations/</code>	Inspire	<code>localhost:3002</code>

Figure 3.5 illustrates how an incoming network request is processed. Let's follow the numbered steps:

- 1 The customer opens the URL `/product/porsche`. The request reaches the frontend proxy.
- 2 The frontend proxy matches the path `/product/porsche` against its routing table. Rule #3 `/product/` is a match.

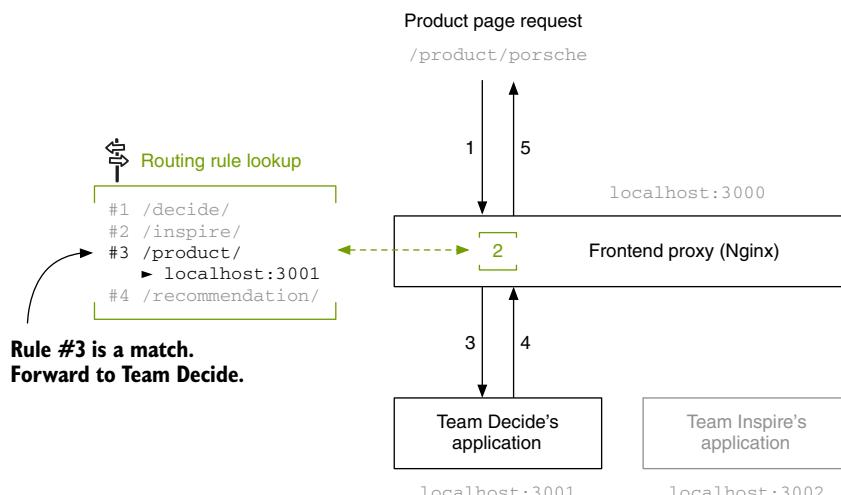


Figure 3.5 Flow of a request. The frontend proxy decides which application should handle an incoming request. It decides based on the URL path and the configured routing rules.

- 3 The frontend proxy passes the request to Team Decide's application.
- 4 The application generates a response and gives it back to the frontend proxy.
- 5 The frontend proxy passes the answer to the client.

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

3.2.1 How to do it

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

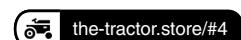
INSTALLING NGINX LOCALLY

If you want to run the example code locally, you need Nginx on your machine. For Windows users, it should work out of the box because I've included the Nginx binaries in the sample code directory. If you're running macOS, the easiest option is to install Nginx via the Homebrew package manager.¹⁰ Most Linux distributions offer an official Nginx package. On Debian- or Ubuntu-based systems you can install it via `sudo apt-get install nginx`. There's no need for extra configuration. The example code only needs the Nginx binary to be present on your system. The `npm` script will automatically start and stop the Nginx together with the team's applications.

STARTING THE APPLICATIONS

Start all three services by running this:

```
npm run 04_routing
```



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

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

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

¹⁰ Get the Homebrew package manager at <https://brew.sh> and install Nginx by running `brew install nginx`.

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

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

- Forwarding a request to another server (`proxy_pass/upstream`)
- Differentiating incoming requests (`location`)

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

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

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

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

See the `proxy_pass` directive in the location block? It advises Nginx to forward all matched requests to the `team_decide` upstream. You can consult the Nginx documentation¹¹ for a more in-depth explanation, but for now we have everything we need to understand our `./webserver/nginx.config` configuration file.

Listing 3.6 `webserver/nginx.conf`

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

Registers Team Decide's application as an upstream called "team_decide"

Handles all request starting with /product/ and forwards them to the team_decide upstream

¹¹ See https://nginx.org/en/docs/beginners_guide.html#proxy.

```

}
location /recommendations {
    proxy_pass  http://team_inspire;
}
location /inspire/ {
    proxy_pass  http://team_inspire;
}
}
}

```

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

3.2.2 Namespacing resources

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

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

- **product page**

```
old: http://localhost:3001/product/<sku>
new: /product/<sku>
```

- **recommendation page**

```
old: http://localhost:3002/recommendations/<sku>
new: /recommendations/<sku>
```

- **recommendation fragment**

```
old: http://localhost:3002/fragment/recommendations/<sku>
new: /inspire/fragment/recommendations/<sku>
```

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

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

3.2.3 Route configuration methods

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

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

STRATEGY 1: TEAM ROUTES ONLY

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

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

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

Choosing 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, putting the information about which team owns which page into your frontend proxy's routing table is unavoidable:

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

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

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

There are multiple technical solutions for handling your routing. Nginx is only one option. Zalando open-sourced its routing solution called Skipper.¹² They've built it to handle more than 800,000 route definitions.

3.2.4 Infrastructure ownership

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

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

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

Distributing infrastructure responsibility across the product teams can help to keep the focus on customer value (see figure 3.6). In our example, Team Decide could take responsibility for running and maintaining Nginx. They, and the other

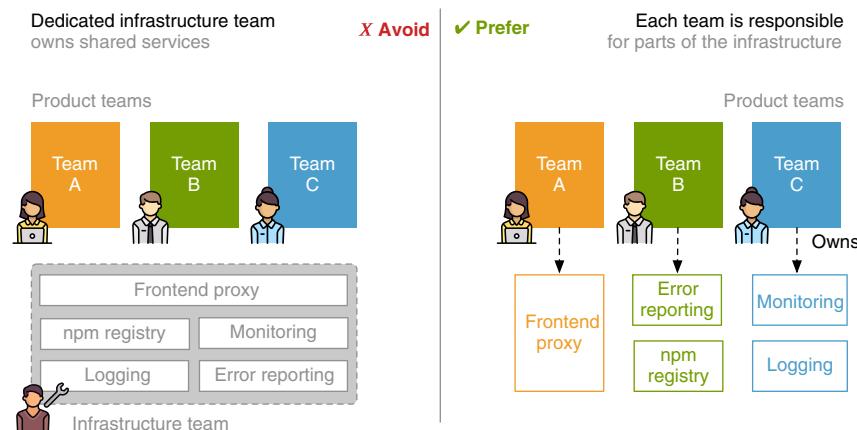


Figure 3.6 Avoid introducing pure infrastructure teams. Distributing responsibility for shared services to the product teams can be a good alternative model.

¹² See <https://opensource.zalando.com/skipper/>.

teams, have a natural interest in this service being well maintained and running stably. The feature teams have no motivation to make a shared service fancier than it needs to be. In our projects we've had good experiences with this approach. It helped to maintain customer focus, even when we were working deeper in the stack. In chapters 12 and 13, we'll go deeper into the centralized vs. decentralized discussion.

3.2.5 When does it make sense?

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

It also has technical benefits:

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

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

Now we've discussed routing on the server side. Nginx is only one way to do it; other tools like Traefik¹³ or Varnish¹⁴ offer similar functionality. In chapter 7 you'll learn how to move these routing rules to the browser. Client-side routing enables us to build a unified single-page app. But before we get there, we'll stay on the server and look at more sophisticated composition techniques.

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, it's possible to have style collisions.
- You can avoid CSS collisions by introducing team namespaces for CSS classes.
- You can route the content of multiple applications through one frontend proxy, which serves all content through a unified domain.
- Using team prefixes in the URL path is an excellent way to make debugging and routing easier.
- Every piece of software should have clear ownership. When possible, avoid creating horizontal teams like a platform team.

¹³ See <https://docs.traefik.io>.

¹⁴ See <https://varnish-cache.org>.

Server-side composition

This chapter covers

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

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

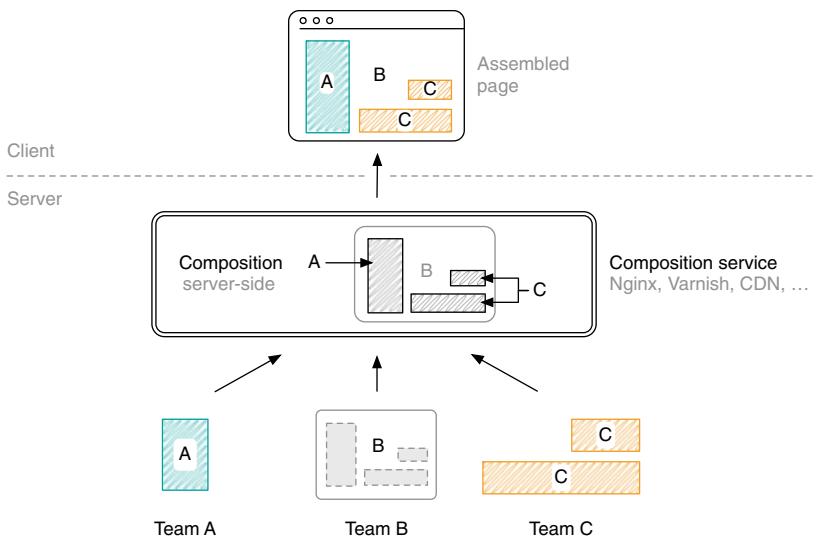


Figure 4.1 Composition of fragments happens on the server. The client receives an already assembled page.

Server-side composition is typically performed by a service that sits between the browser and the actual application servers, as illustrated in figure 4.1. The most significant benefit of server-side integration is that the page is already fully assembled when it reaches the customer's browser. You can achieve incredibly good first-page load speeds that are hard to match using pure client-side integration techniques.

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

4.1 **Composition via Nginx and Server-Side Includes (SSI)**

In the 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 10 languages. She talks to enthusiasts from around the world to get their opinions and feedback.

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

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

To test the site under suboptimal network conditions, one developer opens his browser's developer tools and loads the page with the network throttled to 3G speeds. He is quite surprised to see it take 10 seconds to load.

The developers are confident that there is room for improvement. They plan to move to a server-side integration technique. This way, the first HTML response would already include the references for all assets the site needs. The browser has a complete picture of the page much sooner. It can load the needed resources earlier and in parallel. Since they already have an Nginx in place, the teams choose to use its Server-Side Includes (SSI) feature to do the integration.

4.1.1 How to do it

SSI history

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

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

Let's get to work. This time Team Inspire can lean back. They can reuse the recommendation fragment endpoint from the previous chapter (Ajax).

Team Decide needs to make two changes:

- 1 Activate Nginx's SSI support in the web server's configuration.
- 2 Add an SSI directive to their product template. The SSI's URL must point to Team Inspire's existing recommendation endpoint.

How SSI WORKS

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

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

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

Figure 4.2 shows how our systems generate and compose the HTML for the product page using Server-Side Includes. Let's follow the arrows from the initial request to the final response, from top to bottom. All the steps happen in sequential order.

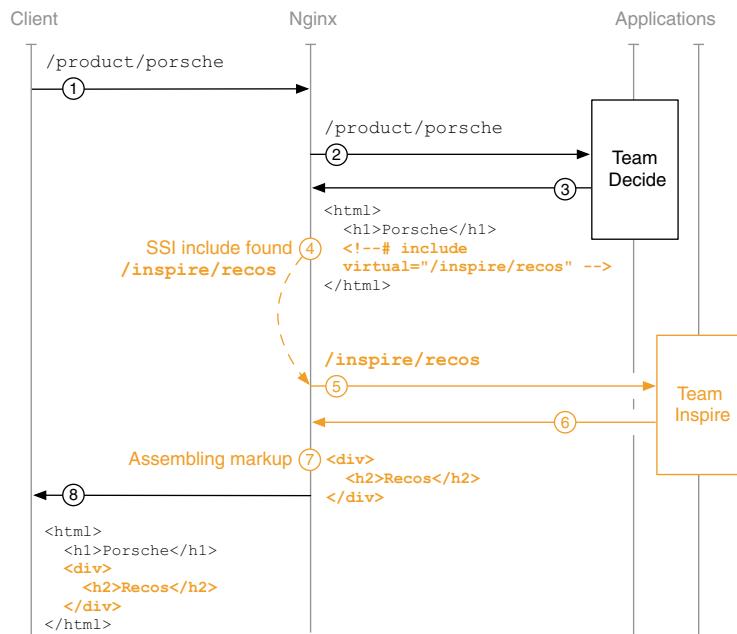


Figure 4.2 SSI processing inside Nginx

- 1 The 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 page's markup with the fragments markup.
- 8 Nginx sends the combined markup to the browser.

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

INTEGRATING A FRAGMENT USING SSI

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

Listing 4.1 webserver/nginx.conf

```
...
server {
    listen 3000;
    ssi on;           ← Activates Nginx's
    ...
}                         server-side include feature
```

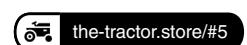
Now we must add the SSI include directive to the product page's markup. It follows a simple structure: `<!--#include virtual="/url-to-include" -->`. We can use the same URL for the fragment as we did with the Ajax example before.

Listing 4.2 team-decide/product/porsche.html

```
...
<aside class="decide_recos">
    <!--#include virtual="/inspire/fragment/recommendations/porsche" --> ←
</aside>
...
Nginx will replace this SSI directive
with the contents of the URL.
```

Start the example by running the following command:

`npm run 05_ssi`



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

4.1.2 Better load times

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

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 take a more in-depth look. The 3G throttling mode limits the available bandwidth, but also delays all requests by around two seconds. We removed the need for the separate fragment Ajax call. The recommendations are already bundled into the initial markup. This bundling saves us two seconds. The other factor is that JavaScript triggered the loading of the Ajax call. The browser had to wait for the JavaScript file to finish before it was able to load the fragment. This waiting for JavaScript accounted for another two seconds.

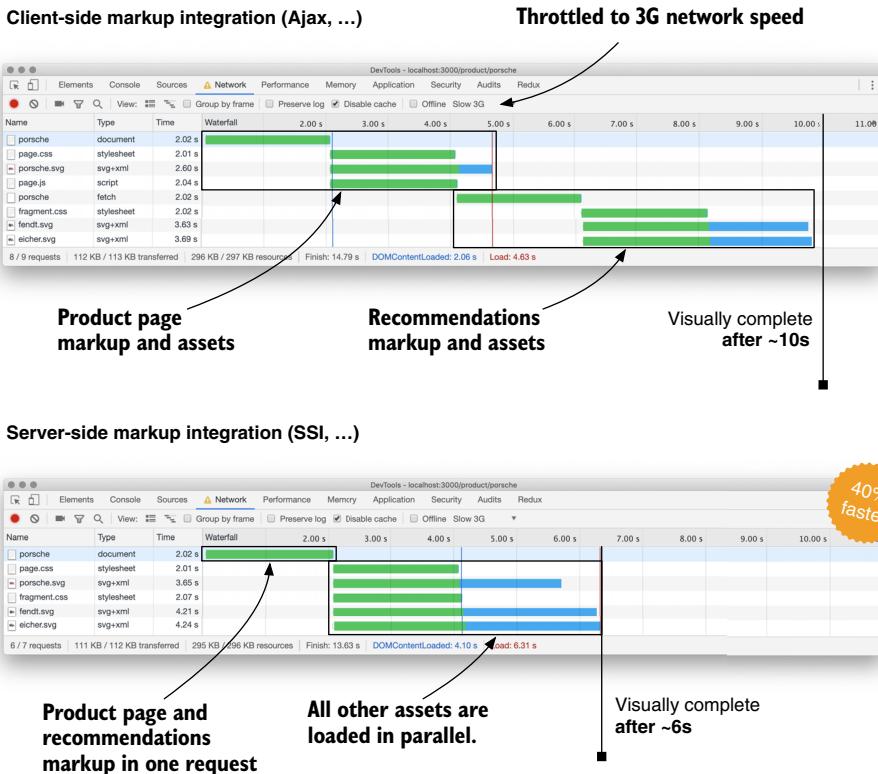


Figure 4.3 Page-load speed for the product page with client-side and server-side composition. Server-side integration optimizes the critical paths.

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

The critical difference is that latency inside one data center is magnitudes smaller and more predictable. We're 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 < 50 ms on good connections to multiple seconds for bad ones.

4.2 Dealing with unreliable fragments

The developers of Team Decide generated a comparison video showing the real-time page load before and after the server-side integration¹ and posted it to the company's Slack channel. As expected, the responses were extremely positive.

¹ WebPageTest is an excellent open-source tool for doing this: <https://www.webpagetest.org/>.

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

4.2.1 The flaky fragment

While Team Decide worked on the server-side integration, Team Inspire was also busy. They were able to build the prototype of a new feature called “Near You.” It informs the tractor fan when a real-world version of one of their favorite models is working in a field nearby. Making this work wasn’t easy—talking to farmer associations, distributing GPS kits to the farmers, and making the real-time 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 100 km radius nearby, we show a little information box on the product page. The first version of this feature will be limited to Europe and Russia, and locates the user by their IP address. The location is not always accurate, and Team Decide plans to leverage the browser’s native geolocation and notifications APIs in the future.

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

Team Inspire 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 Inspire’s developer raises an issue: “Our data processing stack still has a few



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

problems. Sometimes our response times go up to over 500 ms for a couple of minutes. During our last tests, the servers also crashed and rebooted sometimes.”

This unreliability, indeed, is an issue. A response time of 500 ms 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 too long.

4.2.2 Integrating the Near You fragment

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

Let’s take a look at Team Inspire’s new fragment.

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

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

At the moment, only Eicher Diesel 215/16 tractors are GPS-equipped. The fragments for the other tractors (`porsche.html`, `fendt.html`) are just blank files. To display the fragment, Team Decide inserts the associated SSI directive into their product pages, as shown in the following listing.

Listing 4.4 `team-decide/product/eicher.html`

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

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

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

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

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

```
npm run 06_timeouts_short_delay
```



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

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

4.2.3 **Timeouts and fallbacks**

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

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

```
npm run 06_timeouts_down
```

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

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

Listing 4.5 `webserver/nginx.conf`

```
...  
location /inspire/ {  
    proxy_pass  http://team_inspire;  
    proxy_read_timeout 500ms; ←  
}  
...  
  
Team Inspire's upstream has a  
maximum of 500 ms to produce  
an answer for incoming requests.
```

You need to keep in mind that this setting is per upstream and not per request. When requests exceed the configured timeouts, Nginx marks the corresponding upstream as failed and stops even trying to contact it for 10 seconds.²

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

```
npm run 06_timeouts_long_delay
```

In this scenario, we delay all calls to Team Inspire by 1000 ms. Since this exceeds our configured timeout, Nginx omits Team Inspire's fragments. Watch your network view to see that the HTML document takes ~500 ms to load for the first time. Also, notice that Nginx answers all subsequent requests to the product detail page instantly (< 10 ms). Nginx doesn't even try to contact Team Inspire's application for at least 10 seconds. After that 10 seconds, Nginx will try again.

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

4.2.4 **Fallback content**

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

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

Listing 4.6 `team-decide/product/eicher.html`

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

Defining the fallback content as `reco_fallback`

← **Assigning the `reco_fallback` block as fallback/stub to the includes**

² You can change this behavior by setting the `max_fails` and `fail_timeout` options in an upstream configuration. See the Nginx documentation for more details on this: <http://mng.bz/aRro>.

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 -->
    <!--#include
        virtual="/inspire/fragment/near_you/eicher"
        stub="near_you_fallback" -->
</div>
...

```

Empty fallback content
named near_you_fallback

Assigning the
near_you_fallback
block as a fallback

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

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

4.3 Markup assembly performance in depth

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

4.3.1 Parallel loading

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

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

So SSI processing is a two-step process:

- 1 Fetching the page markup
- 2 Fetching all fragments in parallel

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

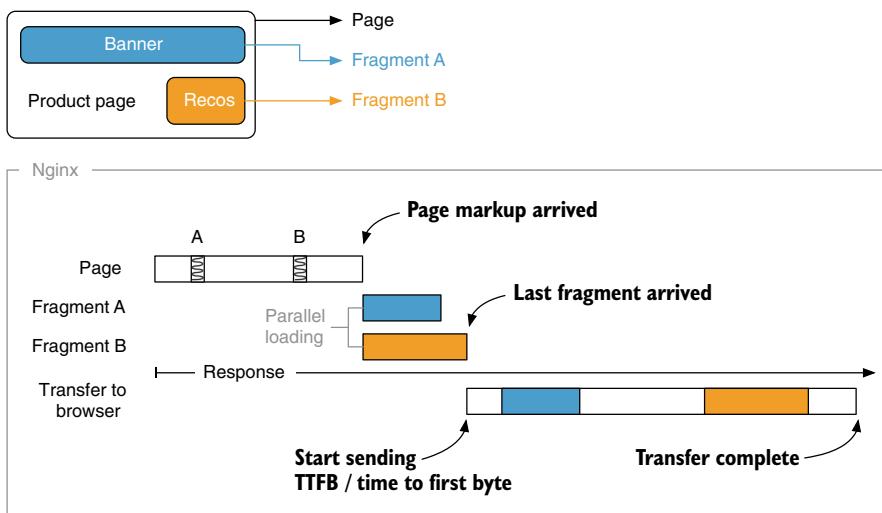


Figure 4.5 Nginx fetches multiple SSI includes in parallel.

4.3.2 Nested fragments

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

A scenario where nesting always came up was the page header. Many pages include the header itself. 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.

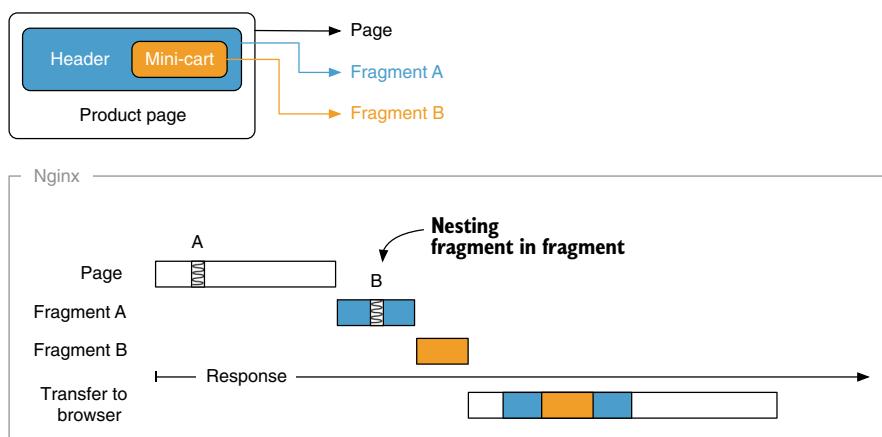


Figure 4.6 Product page includes the header fragment which includes the mini-cart fragment

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

4.3.3 Deferred loading

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

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

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

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

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

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

4.3.4 Time to first byte and streaming

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

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

The idea of streaming templates takes this one step further. With this model, the upstreams generate and send their markup as a stream. The product page would immediately send out the first parts of its template while looking up the required data for the rest of the page (name, image, price) in parallel. The composition server can directly pass this data to the client and start fetching fragments even if the page's

markup from the other upstream hasn't completely arrived yet. The two steps (loading page and loading fragments) overlap, which can reduce overall load time and improves time to first byte significantly. In the next section, we'll look at Tailor and Podium, which both support streaming composition.

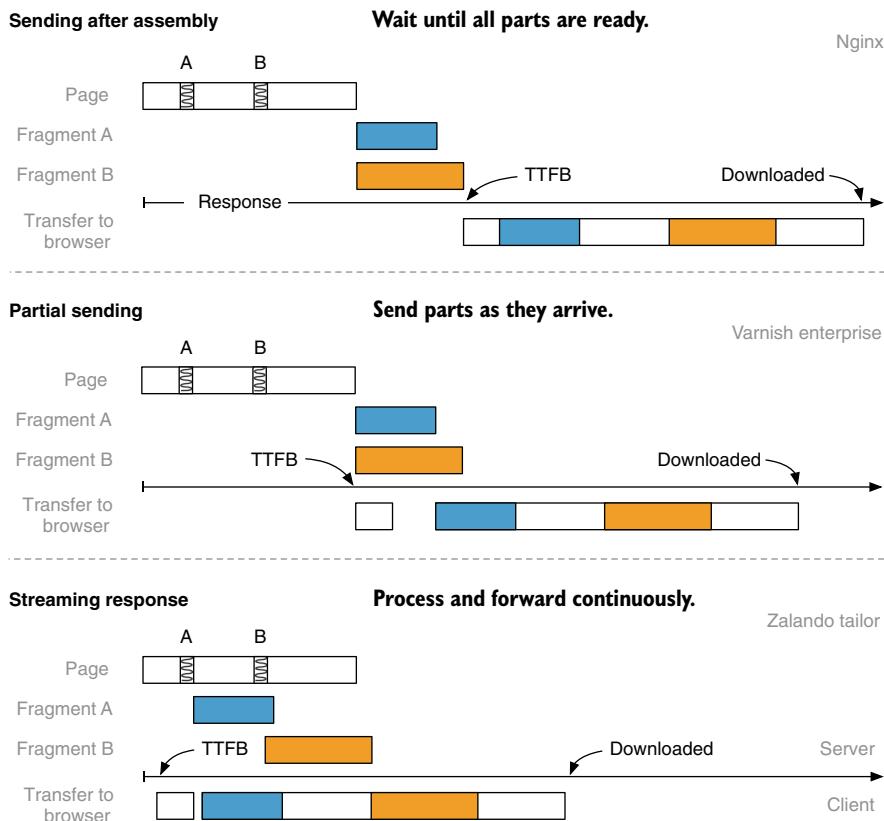


Figure 4.7 Different ways a server-side integration solution can handle fragment loading and markup concatenation internally. The partial sending and streaming approach provides a better time to first byte. This way, the browser receives the content earlier and can start rendering sooner.

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

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

4.4 A quick look into other solutions

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

4.4.1 Edge-Side Includes

Edge-Side Includes, or *ESI*, is a specification³ that defines a unified way of markup assembly. Content delivery network providers like Akamai and proxy servers like Varnish, Squid, and Mongrel support ESI. Setting up an ESI integration solution would look similar to our example. Instead of putting Nginx between the browser and our applications, we could swap it with a Varnish server. An edge side include directive looks like this:

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

FALLBACKS

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

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

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.⁴ There you can add a `maxwait` attribute. When the fragment takes longer, the service will skip it.

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

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

TIME TO FIRST BYTE

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

4.4.2 Zalando Tailor

Zalando⁵ moved from a monolith to a micro frontends-style architecture with *Project Mosaic*.⁶ They published parts of their server-side integration infrastructure. *Tailor*⁷ is a

³ ESI language specification 1.0, <https://www.w3.org/TR/esi-lang>.

⁴ See <https://www.akamai.com/us/en/multimedia/documents/technical-publication/akamai-esi-extensions-technical-publication.pdf>.

⁵ See <https://en.wikipedia.org/wiki/Zalando>.

⁶ See <https://www.mosaic9.org/>.

⁷ See <https://github.com/zalando/tailor>.

Node.js library that parses the page's HTML for special fragment tags, fetches the referenced content, and puts it into the page's markup.

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

Listing 4.8 team-decide/index.js

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

Creating a tailor instance and setting its template folder to `./views`. Consult the documentation for other options.

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

An associated template could look like this.

Listing 4.9 team-decide/views/product.html

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

The fragment tag will be replaced by the content fetched from the src.

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

FALLBACKS AND TIMEOUTS

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

```
<fragment
  src="http://localhost:3002/recos"
  timeout="500"
  fallback-src="http://localhost:3002/recos/fallback"
/>
```

Sets a 500 ms timeout for this fragment

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

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

TIME TO FIRST BYTE AND STREAMING

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

ASSET HANDLING

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

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

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

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

4.4.3 Podium

Finn.no⁸ is a platform for classified ads and Norway's largest website, when ranked by the number of page views. The company is organized into small, autonomous development teams that assemble their pages out of fragments, which they call *podlets*. Finn.no released its Node.js-based integration library called Podium⁹ at the beginning of 2019. It takes concepts from Tailor and improves them. In Podium, fragments are called *podlets*, and pages are *layouts*.

PODLET MANIFEST

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

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

```
{
  "name": "recos",
  "version": "1.0.2",
  "content": "/",
```

⁸ See <https://en.wikipedia.org/wiki/Finn.no>.

⁹ See <https://podium-lib.io/>.

```

"fallback": "/fallback",
"js": [
  { value: "/recos/fragment.js" }
],
"css": [
  { value: "/recos/fragment.css" }
]
...
}
  
```

← Cacheable fallback content
Associated JS and CSS assets

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

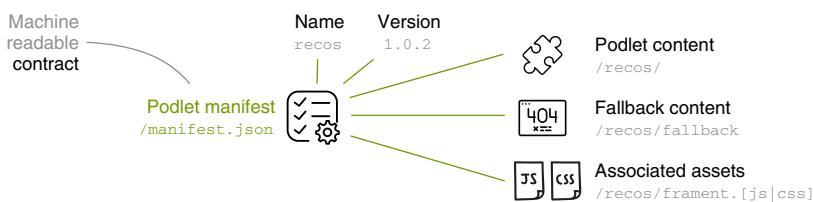


Figure 4.8 Each podlet has its own `manifest.json`, which contains basic metadata but can also include references to fallback content and asset files. The manifest acts as the technical contract between the different teams.

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.

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 The browser asks for the product page. Team Decide receives the request directly.
- 2 Team Decide needs the recommendation fragment from Team Inspire for its product page. It requests the podlet's content endpoint.

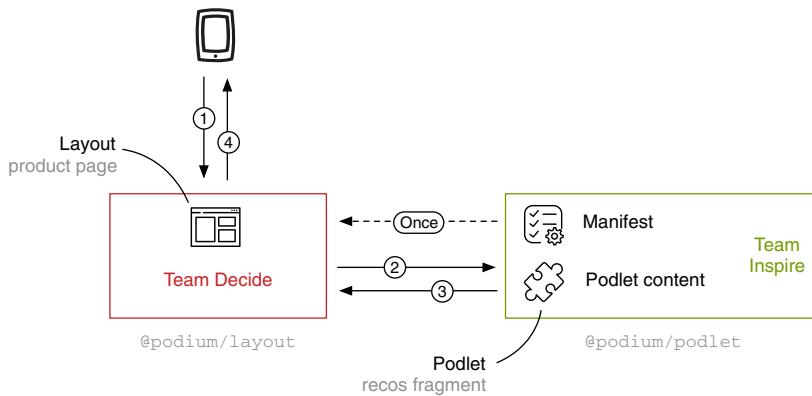


Figure 4.9 Simplified overview of Podium’s architecture. The team that delivers a page (layout) communicates with the browser. It fetches fragment content (podlet) directly from the team that generates it. Associated manifest information is only requested once, not on every request.

- 3 Team Inspire responds with the markup for the recommendation. The response is plain HTML like in the Nginx examples.
- 4 Team Decide puts the received markup into its product page and adds the required JS/CSS references from the manifest file. Team Decide’s application sends the assembled markup to the browser.

IMPLEMENTATION

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

Each of the teams creates its own Node.js-based server. We are using the popular Express¹⁰ framework as a web server, but other libraries also work.

These are Team Decide’s dependencies.

Listing 4.11 team-decide/package.json

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

¹⁰ Express—web framework for Node.js: <https://expressjs.com/>.

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({
  name: "product",
  pathname: "/product",
});

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

const app = express();
app.use(layout.middleware());
app.get("/product", async (req, res) => {
  const recoHTML = await recos.fetch(res.locals.podium);

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

app.listen(3001);
```

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

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

Creating an express instance and attaching podiums layout middleware to it

Defining the route /product that delivers the product page

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

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

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

```
npm run 07_podium
```

 the-tractor.store/#7

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

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

Listing 4.13 team-inspire/package.json

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

And this is the application code.

Listing 4.14 team-inspire/index.js

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

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

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

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

app.get("/recos", (req, res) => {
  res.status(200).podiumSend(`<h2>Recommendations</h2>


`);
}

app.listen(3002);
```

Defining a podlet. name, version, and pathname are required parameters.

Creating an express instance and attaching our podlet middleware to it

Defining the route for the manifest.json

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

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

FALLBACKS AND TIMEOUTS

The way Podium handles fallbacks is quite interesting. With the Nginx approach, we had to define the fallback in the template of the page. With ESI and Tailor, the page

owner can provide a second URL that's tried when the actual URL does not work. In Podium it's a bit different:

- The team that owns the fragment provides the fallback.
- The team including the fragment caches this fallback locally.

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

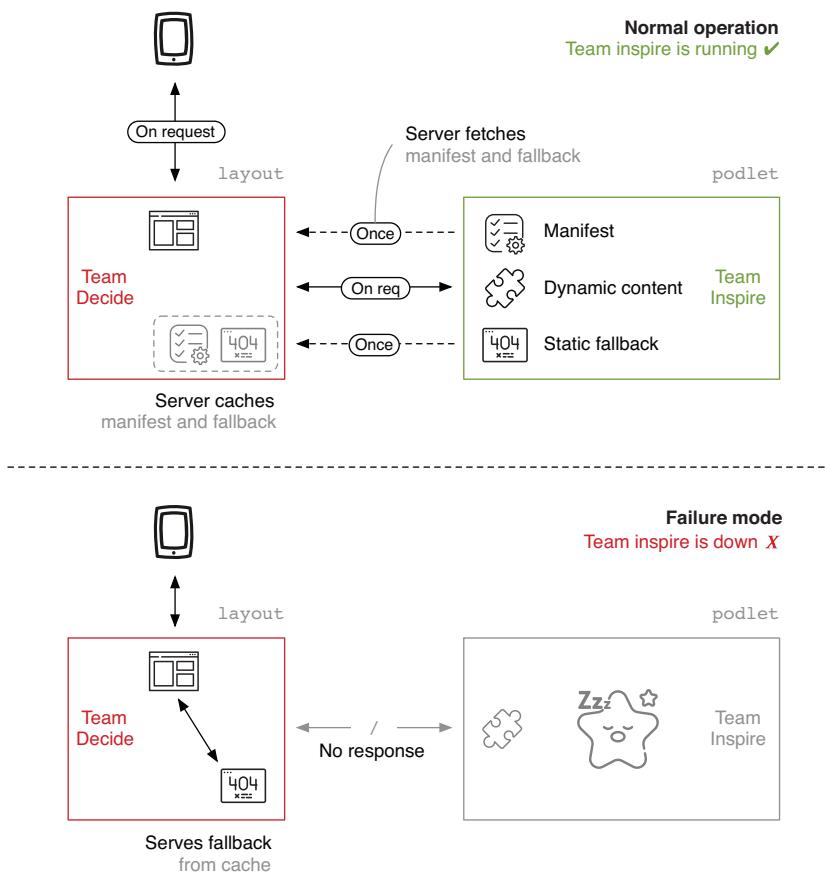


Figure 4.10 Podium's fallback handling. The podlet owner can specify the fallback content in the manifest. The layout service retrieves the fallback content once and caches it. When the podlet server goes down, the fallback is used instead of the dynamic content.

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

Listing 4.15 team-inspire/index.js

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

```

Adding the fallback property to the podlets configuration

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 a fragment's integration is pretty handy. The format is simple and straightforward. 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 documentation¹¹ is an excellent place to start.

4.4.4 Which solution is right for me?

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

¹¹ Podium documentation: https://podium-lib.io/docs/podium/conceptual_overview/.

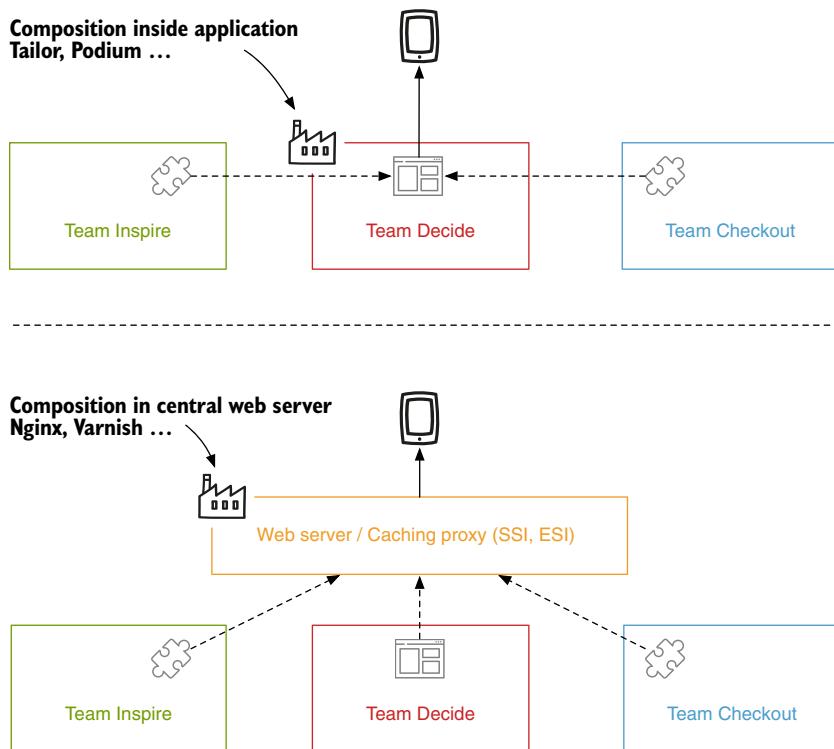


Figure 4.11 Fragment composition in the application or in a central web server

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

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

4.5 **The good and bad of server-side composition**

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

4.5.1 **The benefits**

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

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

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

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

4.5.2 **The drawbacks**

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

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

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

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

4.5.3 **When does server-side integration make sense?**

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

If your project requires an app-like user interface that can instantly react to user input, server-side integration is not for you. A pure client-side solution might be easier to implement. But you can also go the hybrid way and build a universal/isomorphic application using both server- and client-side composition. In chapter 8 you'll learn how to do so.

Figure 4.12 shows the comparison chart introduced in the previous chapter. We've added server-side integration.

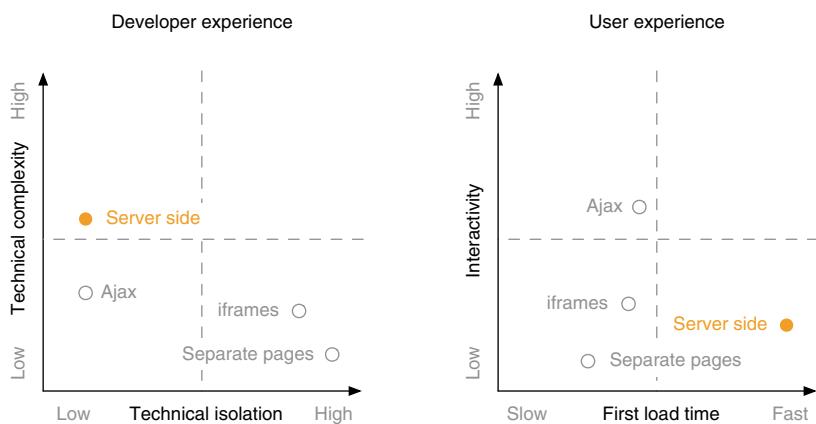


Figure 4.12 Server-side integration in comparison to other integration techniques.
Server-side integration introduces extra infrastructure, which increases complexity.
Similar to the Ajax approach, they don't introduce technical isolation. You still have
to rely on manual namespacing. But they enable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

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 arrives.
- Library-based integration solutions like Tailor and Podium directly integrate into a team's application. Therefore less infrastructure is required, and local development is more comfortable. But they also are a non-trivial dependency.
- The integration solution is a central piece in your architecture. It's good to pick a solution that is solid and easy to maintain.
- Server-side composition is the basis for building a micro-frontends-style site that uses progressive enhancement principles.

5

Client-side composition

This chapter covers

- Examining Web Components as a client-side composition technique
- Investigating how to use micro frontends, built with different frameworks, on the same page
- Exploring how Shadow DOM can help 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, including SSI and Podium. These techniques are indispensable for websites that need to load quickly. But for many applications, the first load time is not the only important thing. Users expect websites to feel snappy and react to their input promptly. No one wants to wait for the complete page to reload just because they changed an option in a product configuration. People spend more time on sites that react quickly and feel app-like. Due to this fact, client-side rendering with frameworks like React, Vue.js, or Angular has become popular. With this model,

the HTML markup gets produced and updated directly in the browser. Server-side integration techniques don't provide an answer to this.

In a traditional architecture, we would have built a monolithic frontend that's tied to one framework in one specific version. But in a micro frontends architecture, we want the user interfaces from the different teams to be self-contained and independently upgradable. We can't rely on the component system of one specific framework. This constraint would tie the complete architecture to a central release cycle. A framework change would result in a parallel rewrite of the complete frontend. The Web Components spec introduces a neutral and standardized component model. In this chapter, you'll learn how Web Components can act as a technology-agnostic glue between different micro frontends. They make it possible for independent frontend applications to coexist on one page, even if their technology stack is not the same. Figure 5.1 illustrates this client-side integration.

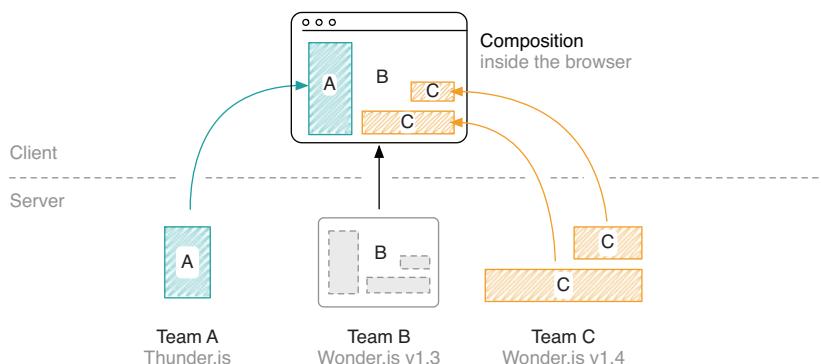


Figure 5.1 Micro frontend composition in the browser. Each fragment is its own mini-application and can render and update its markup independently from the rest of the page. Thunder.js and Wonder.js are placeholders for your frontend framework of choice.

5.1 Wrapping micro frontends using Web Components

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

But the online shop is still missing its most important feature: the Buy button. Up until now, customers have only been able to see the tractors and the site's recommendations. Recently, the company staffed a third team: Team Checkout. They have 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.

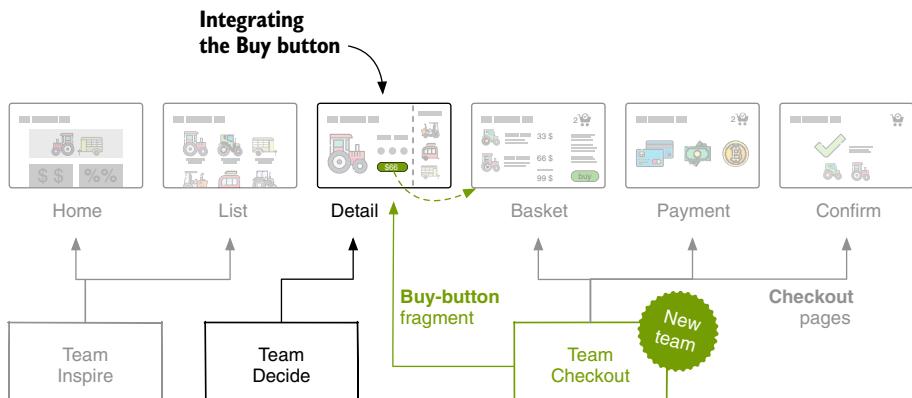


Figure 5.2 Team Checkout owns the complete checkout flow. Team Decide does not have to know about how the checkout works. But they need to integrate Team Checkout's Buy-button fragment in the detail page to make it work. Team Checkout provides this button as a standalone micro frontend.

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 (see figure 5.2). Let's see what this means and how we can integrate the fragment into the product detail page. For the integration, Team Checkout provides Team Decide with the necessary information. This is the *contract between both teams*:

■ Buy Button

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

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

■ required JS & CSS assets references

```
http://localhost:3003/static/fragment.js
http://localhost:3003/static/fragment.css
```

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

5.1.1 How to do it

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

familiar with Web Components, you can skip the next two sections and continue with encapsulating business logic through DOM elements.

WEB COMPONENTS AND CUSTOM ELEMENTS

The Web Components spec has been a long time 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 version 1 of the specification. It's also possible to retrofit the implementation into older browsers using a polyfill.¹

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

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

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

Custom Elements enable developers to create similar abstractions. You can construct new generic representational elements that are missing from the HTML spec. GitHub has published a list of such controls under the name `github-elements`.² Look at this “copy-to-clipboard” element:

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

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

WEB COMPONENTS AS A CONTAINER FORMAT

You can also use Web Components to encapsulate business logic. Let's go back to our example at The Tractor Store. Team Checkout owns the domain knowledge around product prices, inventory, and availabilities. Team Decide, owner of the product page, doesn't have to know these concepts. Its job is to provide the customer with all product information they need 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>
```

¹ See <https://www.webcomponents.org/polyfills>.

² See <https://www.webcomponents.org/author/github>.

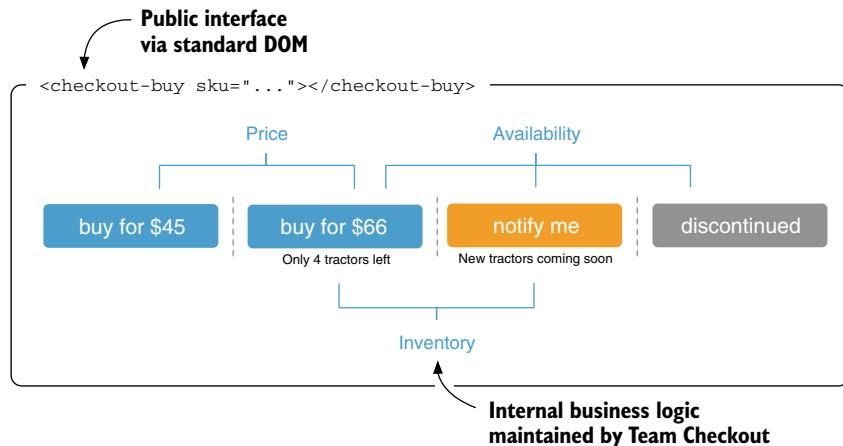


Figure 5.3 A Custom Element can encapsulate business logic and provide the associated user interface. The Buy button can look different depending on the specified SKU, but also due to internal pricing and inventory information. A team that uses this fragment does not have to know these concepts.

DEFINING A CUSTOM ELEMENT

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

Listing 5.1 team-checkout/static/fragment.js

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

Defines an ES6 class for the Custom Element

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

Registers the Custom Element under the name checkout-buy.

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

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

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

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

USING A CUSTOM ELEMENT

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

Listing 5.2 team-decide/product/porsche.html

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

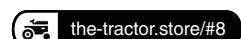
Keep in mind that Custom Elements cannot be self-closing. They always need a dedicated closing tag like </checkout-buy>. Since the fragment is fully client-rendered, Team Checkout only needs to host two files: fragment.css and fragment.js. Team Inspire has reworked their recommendations micro frontend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



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

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

```
npm run 08_web_components
```



Opening `http://localhost:3001/product/porsche` shows you the product page with the client-side-rendered Buy button like in figure 5.5.

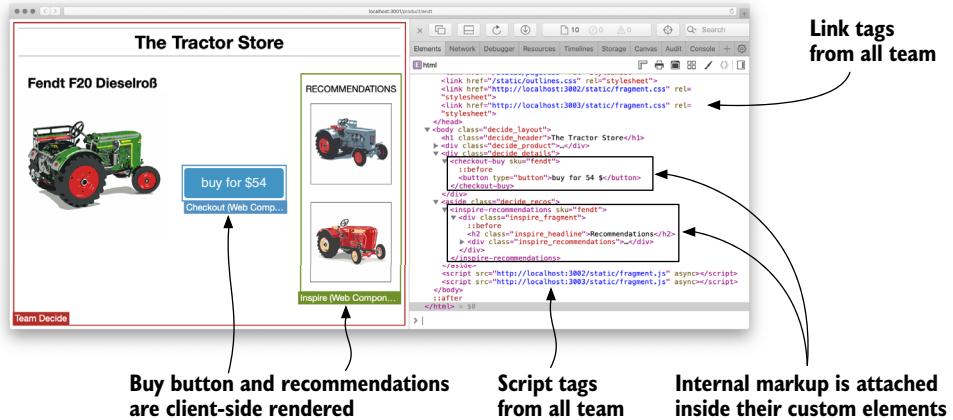


Figure 5.5 The Custom Element renders itself in the browser via JavaScript. It generates its internal markup and attaches it as children to the tree via `this.innerHTML = "..."`.

PARAMETRIZATION VIA ATTRIBUTES

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

Listing 5.3 `team-checkout/static/fragment.js`

```
const prices = { porsche: 66, fendt: 54, eicher: 58 }; ← List of tractor prices
class CheckoutBuy extends HTMLElement {
  connectedCallback() {
    const sku = this.getAttribute("sku"); ← Reading the SKU from the Custom Elements attribute
    this.innerHTML = `← Looking up and rendering the price on the button
      <button type="button">
        buy for $$${prices[sku]}
      </button>
    `;
  }
}
```

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

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

Listing 5.4 team-checkout/static/fragment.js

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

```

Get the reference to the button.
Add a click handler.
Display a success message on click.

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

5.1.2 Wrapping your framework in a Web Component

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

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

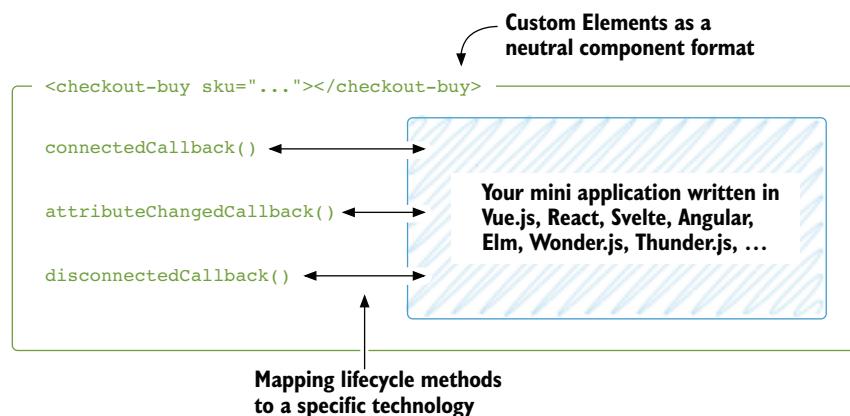


Figure 5.6 Custom Elements introduce lifecycle methods. You need to map these to the specific technology of your micro frontend.

Some newer frameworks like Stencil.js³ already use Web Components as their primary way to export an application. Angular comes with a feature called Angular Elements.⁴ This feature will automatically generate the code necessary to connect the app with the Custom Elements' lifecycle methods, and also supports Shadow DOM. Vue.js provides a similar solution via the official `@vue/web-component-wrapper` package.⁵ Since Web Components are a web standard, there are comparable libraries or tutorials for all popular frameworks out there.

This chapter's example code is deliberately kept simple and doesn't include a front-end framework. You can check out the examples `20_shared_vendor_rollup_absolute_imports` from chapter 11 to see a React application wrapped in a Custom Element.

5.2 Style isolation using Shadow DOM

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

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

5.2.1 Creating a shadow root

You can create an isolated DOM sub-tree via JavaScript by calling `.attachShadow()` on an HTML element. Most people use Shadow DOM in combination with a Custom Element, but you don't have to. You can also attach a Shadow DOM to many standard HTML elements like a `div`.⁶

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

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

Creating an "open" shadow tree

Writing content to the newly created shadowRoot

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

³ See <https://stenciljs.com/>.

⁴ See <https://angular.io/guide/elements>.

⁵ See <https://github.com/vuejs/vue-web-component-wrapper>.

⁶ See <https://dom.spec.whatwg.org/#dom-element-attachshadow> for a list of elements that support Shadow DOM.

Open versus closed

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

5.2.2 Scoping styles

Let's move the styling from the fragment.css into the actual component. We do this by defining a `<style>...</style>` block inside the shadow root. Styles that are defined in the Shadow DOM stay in the Shadow DOM. Nothing leaks out to 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.⁷ Look at the code for the Buy-button fragment.

Listing 5.5 team-checkout/static/fragment.js

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

The code listing has three annotations:

- A callout pointing to `this.attachShadow({ mode: "open" })` with the text "Creating a Shadow DOM for the Buy-button element".
- A callout pointing to the `<style>...</style>` block with the text "Defining styles as inline CSS. They only apply inside the shadowRoot".
- A callout pointing to the entire innerHTML block with the text "Writing content to the shadowRoot instead of directly attaching it to the Buy button".

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

```
npm run 09_shadow_dom
```



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

⁷ Except for a few inherited properties like `font-family` and `root font-size`.

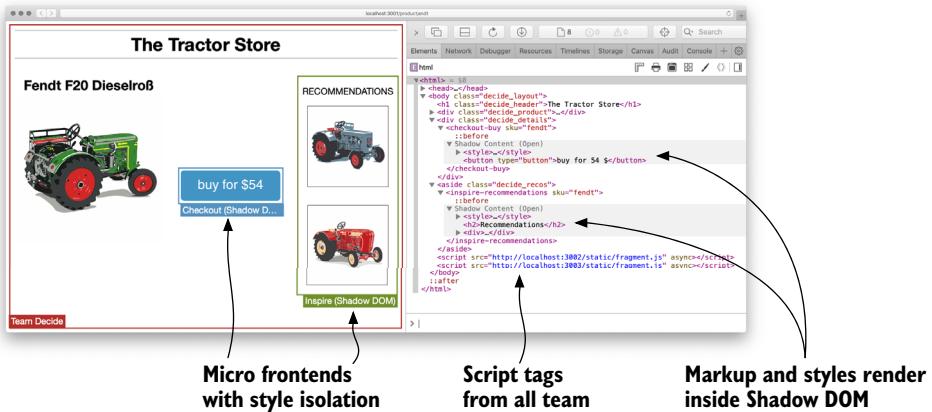


Figure 5.7 Micro frontends can render their internal markup and styles inside a shadow root. This improves isolation and reduces the risk of conflicting or leaking styles.

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

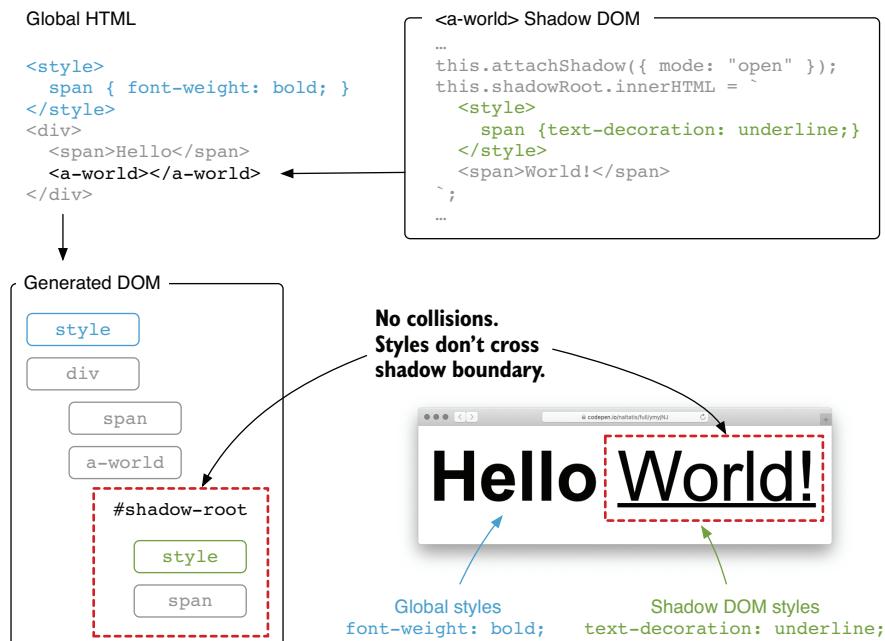


Figure 5.8 The shadow root creates a border called the *shadow boundary*. It provides isolation in both ways. Styles don't leak out of the component. Styles on the page also don't affect the Shadow DOM.

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

5.2.3 **When to use Shadow DOM**

There are a lot of details you can learn about Shadow DOM.⁸ Events behave differently when they bubble from the Shadow DOM into the regular DOM (also called *Light DOM*). But since this is a book about micro frontends and not Web Components, we won't go deeper into this topic. Here is a list of pros and cons of using Shadow DOM in a micro frontends context:

■ Pros

- Strong iframe-like isolation. No namespacing required.
- Prevents global styles from leaking into a micro frontend. Great when working with legacy applications.
- Potential to reduce the need for CSS toolchains.
- Fragments are self-contained. No separate CSS file references.

■ Cons

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

5.3 **The good and bad of using Web Components for composition**

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

5.3.1 **The benefits**

The *most significant benefit of using Web Components* as an integration technique is that *they are a widely implemented web standard*. It's often not very convenient to work with

⁸ See Caleb Williams, "Encapsulating Style and Structure with Shadow DOM," *CSS-Tricks*, <https://css-tricks.com/encapsulating-style-and-structure-with-shadow-dom/>.

browser APIs directly. But abstractions exist that make developing easier. Web standards evolve slowly and always in a non-breaking, backward compatible way. That's why they are an excellent fit for a common basis.

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

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

5.3.2 **The drawbacks**

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

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

5.3.3 **When does client-side integration make sense?**

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

The interesting question is what "interactive" means. We'll discuss this topic in chapter 9. Are you building a site or an app? For simpler use cases like a catalog or a content-heavy site, a server-rendered approach that uses SSI or Ajax often works fine and is easier to handle.

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

It's also possible to combine Custom Elements (not Shadow DOM) with a server-side integration technique. We'll explore this in chapter 8.

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

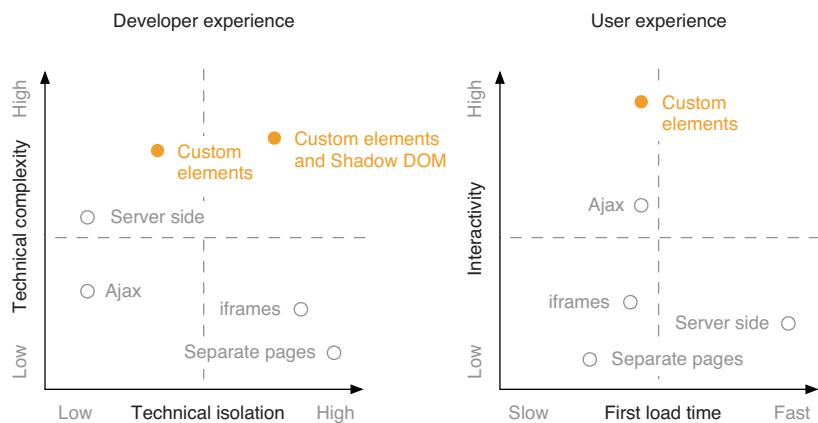


Figure 5.9 Custom Elements provide a good way to encapsulate your JavaScript application and make it accessible in a standard way. Shadow DOM introduces an extra isolation mechanism and lowers the risk of conflicts. You can build highly interactive, client-side rendered applications using Custom Elements.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

Summary

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

Communication patterns



This chapter covers

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

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

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

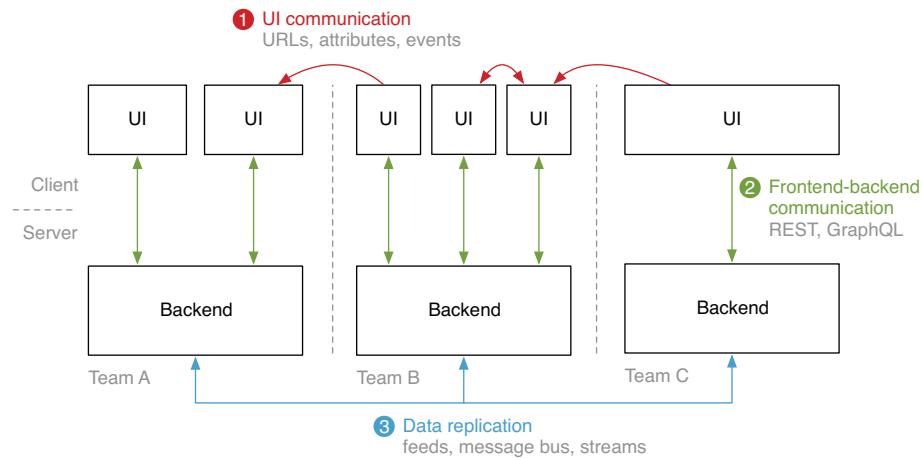


Figure 6.1 An overview of different communication mechanisms in a typical micro frontends architecture. The frontend applications in the browser need a way to talk to each other. We call this **UI communication** ①. Each frontend fetches data from its own backend ②, and in some cases, it's required to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

6.1 User interface communication

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

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

This communication can be simple. We've already used *page-to-page communication* in chapter 2—moving from the product page to another team's recommendation page via a simple link. In our case, we transferred the product reference, the SKU, via the

URL path or the query string. In most cases, cross-team communication happens via the URL.

If you are building a richer user interface that combines multiple use cases on one page, a link isn't sufficient anymore. You need a standard way for the different UI parts to talk to each other. Figure 6.2 illustrates three common communication patterns.

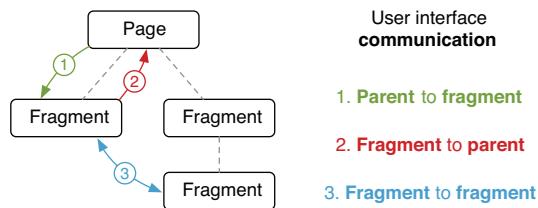


Figure 6.2 Three different forms of communication that can happen between the different teams' UIs inside a page

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

6.1.1 Parent to fragment

The introduction of the Buy button on the product page resulted in a considerable amount of tractor sales over one weekend. But Tractor Model, Inc. has no time to rest. CEO Ferdinand was able to hire two of the best goldsmiths. They've designed special platinum editions of 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. See figure 6.3.

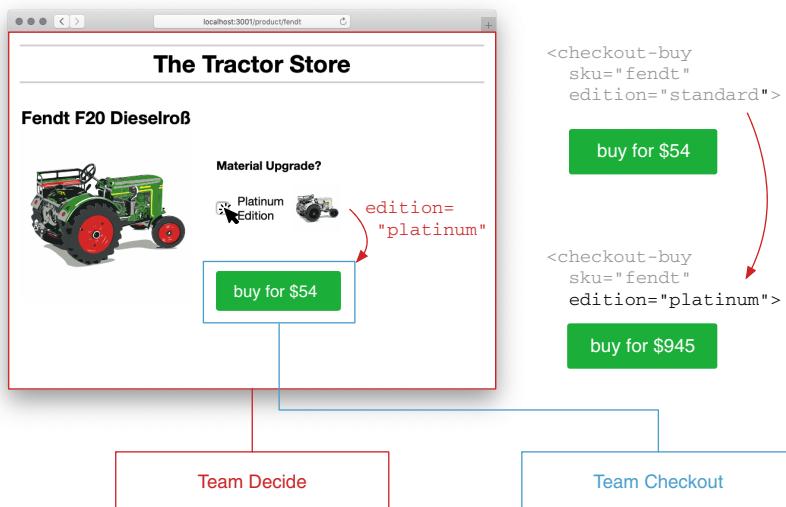


Figure 6.3 Parent-child communication. A change in the parent page (selection of platinum option) needs to be propagated down to a fragment so it can update itself (price change in the Buy button).

Both teams talk and come up with a plan. Team Checkout will extend the Buy button using another attribute called `edition`. Team Decide sets this attribute and updates it accordingly when the user changes the option:

- **Updated Buy button**

tag-name: `checkout-buy`

attributes: `sku=[sku]`, `edition=[standard|platinum]`

example: `<checkout-buy sku="porsche" edition="platinum"></checkout-buy>`

IMPLEMENTING THE PLATINUM OPTION

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

Listing 6.1 team-decide/product/fendt.html

```
...

...
<label class="decide_editions">
    <input type="checkbox" name="edition" value="platinum" />
    <span>Platinum Edition</span>
</label>
<checkout-buy sku="fendt" edition="standard"></checkout-buy>
...

```

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

Listing 6.2 team-decide/static/page.js

Selecting the DOM elements that need to be watched or changed

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

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

Updating the main product image

Updating the edition attribute on Team Checkout's Buy-button custom element

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

UPDATING ON ATTRIBUTE CHANGE

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

The most interesting one for our case is `attributeChangedCallback (name, oldValue, newValue)`. This method is triggered every time someone changes an attribute of your custom element. You receive the name of the attribute that changed (`name`), the attribute's previous value (`oldValue`), and the updated value (`newValue`). For this

to work, you have to register the list of attributes that should be observed up front. The code of the custom element now looks like this.

Listing 6.3 team-checkout/static/fragment.js

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

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

The code is annotated with several callouts:

- Added new prices for platinum versions**: Points to the addition of the `platinum` key in the `prices` object.
- Watching for changes to the sku and edition attribute**: Points to the `observedAttributes` getter.
- Extracted the rendering to a separate method**: Points to the `render` method.
- Calling render () on every attribute change**: Points to the `attributeChangedCallback` method.
- Extracted render method**: Points to the `render` method again.
- Retrieves the current SKU and edition value from the DOM**: Points to the `getAttribute` calls.
- Renders the price based on SKU and edition**: Points to the rendered HTML output.

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

```
npm run 10_parent_child_communication
```



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

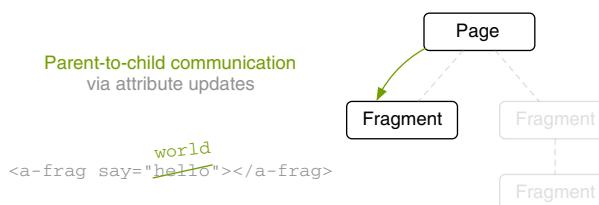


Figure 6.4 You can achieve parent-child communication by explicitly passing needed context information down as an attribute. The fragment can react to this change.

Figure 6.4 illustrates the data flow. We propagate changed state of the outer application (product page) to the nested application (Buy button). This follows the *unidirectional dataflow*¹ pattern. React and Redux popularized the “props down, events up” approach. The updated state is passed down the tree via attributes to child components as needed. Communication in the other direction is done via events. We’ll cover this next.

6.1.2 Fragment to parent

The introduction of the platinum editions resulted in a lot of controversial discussions in the Tractor Model, Inc. user forum. Some users complained about the premium prices. Others asked for additional black, crystal, and gold editions. The first 100 platinum tractors shipped within one day.

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

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

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

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

The 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. See figure 6.5.

¹ See <http://mng.bz/pB72>.

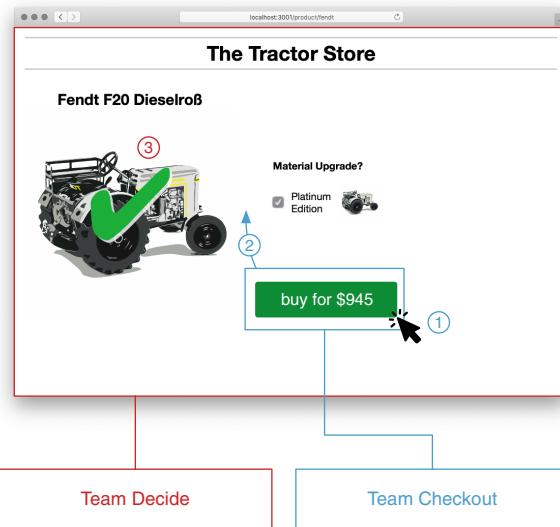


Figure 6.5 Team Checkout's Buy button emits an event when the user adds an item to the cart. Team Decide reacts to this event and triggers an animation on the main product image.

EMITTING CUSTOM EVENTS

Let's look at the code that's needed to make the interaction happen. We'll use the browser's native `CustomEvents` API. The feature is available in all browsers, including older versions of Internet Explorer. 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 event's name.

The following code shows the Buy-button fragment with the event added.

Listing 6.4 team-checkout/static/fragment.js

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

Creates a custom event named `checkout:item_added`

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 straightforward, 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 won't get into the associated CSS code. It uses a CSS keyframe animation, which makes a prominent green checkmark character (✓) fade in and out again. We can trigger the animation by adding a `decide_product-confirm` class to the existing `decide_product` div element.

Listing 6.5 team-decide/static/page.js

```

Selecting the
Buy-button element
→ const buyButton = document.querySelector("checkout-buy");
const product = document.querySelector(".decide_product");
buyButton.addEventListener("checkout:item_added", e => {
    product.classList.add("decide_product--confirm");
});
product.addEventListener("animationend", () => {
    product.classList.remove("decide_product--confirm");
});

Selecting the product block where
the animation should happen
Listening to Team
Checkout's custom event
Cleanup—removing the class
after the animation finished
Triggering the animation by
adding the confirm class

```

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

`npm run 11_child_parent_communication`



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

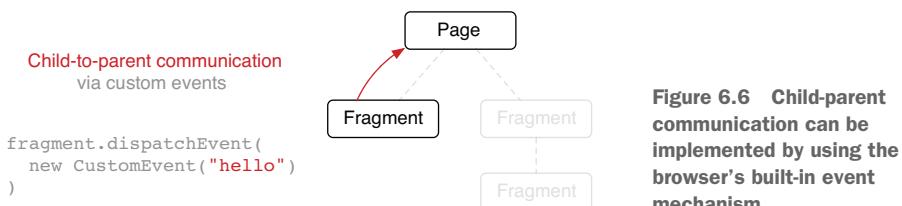


Figure 6.6 Child-parent communication can be implemented by using the browser's built-in event mechanism.

Using the browser's event mechanism has multiple benefits:

- Custom Events can have high-level names that reflect your domain language. Good event names are 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.
- It gives access to all native event features like `.stopPropagation` or `.target`.
- It's easy to debug via browser developer tools.

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

6.1.3 Fragment to fragment

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

Team Checkout wants to add a mini-cart to the product page to reduce the number of product returns. This way, customers always see what's in their basket. Team Checkout provides the mini-cart as a new fragment for Team Decide to include on the bottom of the product page. The contract for including the mini-cart looks like this:

- **Mini-Cart**

tag-name: `checkout-minicart`

example: `<checkout-minicart></checkout-minicart>`

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

That's all pretty straightforward, 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:

- *Direct communication*—A fragment finds the fragment it wants to talk to and directly calls a function on it. Since we are in the browser, a fragment has access to the complete DOM tree. It could search the DOM for the element it's looking for and talk to it. *Don't do this. Directly referencing foreign DOM elements introduces tight coupling.* A fragment should be self-contained and not know about other fragments on the page. Direct communication makes it hard to change the composition of fragments later on. Removing a fragment or duplicating one can lead to strange effects.
- *Orchestration via a parent*—We can combine the *child-parent* and *parent-child* mechanisms. In our case, Team Decide's product page would listen to the `item_added` event from the Buy button and directly trigger an update to the mini-cart fragment. This is a clean solution. We've explicitly modeled the communication flow in the parent's system. But to make a change in communication, two teams must adapt their software.
- *Event-Bus/broadcasting*—With this model, you introduce a global communication channel. Fragments can publish events to the channel. Other fragments can subscribe to these events and react to them. The publish/subscribe mechanism reduces coupling. The product page, in our example, wouldn't have to know or

care about the communication between the Buy button and the mini-basket fragment. You can implement this with Custom Events. Most browsers² also support the new *Broadcast Channel API*,³ which creates a message bus that also spans across browser windows, tabs, and iframes.

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

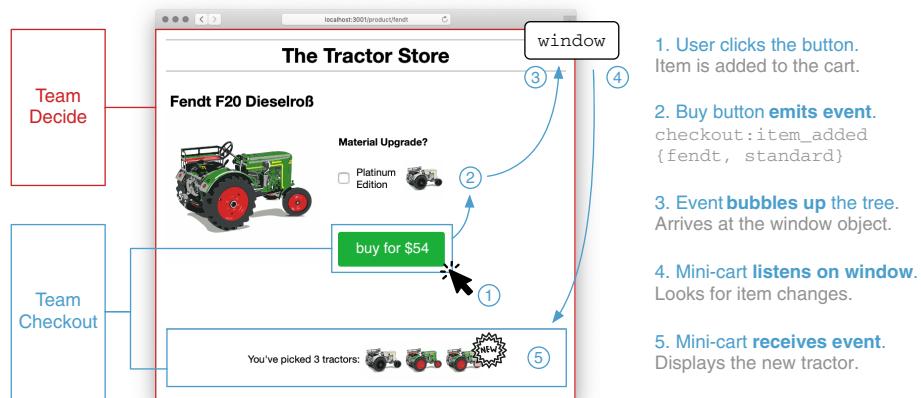


Figure 6.7 Fragment-to-fragment communication via a global event. The Buy button emits the `item_added` event. The mini-cart listens for this event on the `window` object and updates itself. We use the browser's native event mechanism as an event bus.

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

- **Updated Buy button**

tag-name: `checkout-buy`

attributes: `sku=[sku]`, `edition=[standard|platinum]`

emits event:

name: `checkout:item_added`

payload: `{sku: [sku], edition: [standard|platinum]}`

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

² At the time of writing this, Safari is the only browser that hasn't implemented it: <https://caniuse.com/#feat=broadcastchannel>.

³ See <http://mng.bz/OMeo>.

Let's look at the implementation of this.

EVENT BUS VIA BROWSER EVENTS

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

Listing 6.6 team-checkout/static/fragment.js

```
...
const event = new CustomEvent("checkout:item_added", {
  bubbles: true,
  detail: { sku, edition }           ← Attaches a custom
}__);                                ← Enables event
this.dispatchEvent(event);            ← bubbling
...

```

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

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

Listing 6.7 team-checkout/static/fragment.js

```
...
class CheckoutMinicart extends HTMLElement {
  connectedCallback() {
    this.items = [];
    window.addEventListener("checkout:item_added", e => {
      this.items.push(e.detail);           ← Initializing a local variable
      this.render();                     ← for holding the cart items
    });
    this.render();
  }
  render() {
    this.innerHTML = `
      You've picked ${this.items.length} tractors:
      ${this.items.map(({ sku, edition }) =>
        ``  

      ).join("")}
    `;
    ...
  }
}
window.customElements.define("checkout-minicart", CheckoutMinicart);
```

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. Lastly, it triggers a refresh of the view by calling `this.render()`.

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

Listing 6.8 `team-decide/product/fendt.html`

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

Adding the new mini-cart fragment to the bottom of the page

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

```
npm run 12_fragment_fragment_communication
```

 the-tractor.store#12



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

DISPATCHING EVENTS DIRECTLY ON 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. Knowing which DOM element emitted the event is helpful when you have multiple instances of a fragment on one page. Having this element reference avoids the need to create a separate naming or identification scheme yourself.

Parents can also cancel bubbling events on their way up to the window. You can use `event.stopPropagation` on Custom Events the same way you would with a standard

click event. This can be helpful when you want an event to only be processed once. However, the `stopPropagation` mechanism can also be a source of confusion: “Why don’t you see my event on `window`? I’m sure we’re dispatching it correctly.” So be careful with this—especially if more than two parties are involved in the communication.

6.1.4 Publish/Subscribe with the Broadcast Channel API

In the examples so far, we’ve leveraged the DOM for communication. The relatively new *Broadcast Channel API* provides another standards-based way to communicate. It’s a publish/subscribe system which enables communication across tabs, windows, and even iframes from the same domain. The API is pretty simple:

- You can connect to a channel with `new BroadcastChannel("tractor_channel")`.
- Send messages via `channel.postMessage(content)`.
- Receive messages via `channel.onmessage = function(e) { ... }`.

In our case all micro frontends could open a connection to a central channel (like `tractor_channel`) and receive notifications from other micro frontends. Let’s look at a small example.

Listing 6.9 team-checkout.js

```
const channel = new BroadcastChannel("tractor_channel");
const buyButton = document.querySelector("button");
buyButton.addEventListener("click", () => {
  channel.postMessage(
    {type: "checkout:item_added", sku: "fendt"}
  );
});
```




Team Checkout
connects to the central
broadcast channel.

They post an item_added message
when someone clicks the Buy
button. In this example we send an
object, but you can also send plain
strings or other types of data.

Listing 6.10 team-decide.js

```
const channel = new BroadcastChannel("tractor_channel");
channel.onmessage = function(e) {
  if (e.data.type === "checkout:item_added") {
    console.log(`tractor ${e.data.type} added`);
    // -> tractor fendt added
  }
};
```




Team Decide also connects
to the same channel.

They listen to all messages and
create a log entry every time
they receive an item_added.

At the time of writing this book, all browsers except Safari support the Broadcast Channel API.⁴ You can use a polyfill⁵ to retrofit the API into browsers without native support.

⁴ Broadcast Channel API—Browser Support: <https://caniuse.com/#feat=broadcastchannel>.

⁵ Broadcast Channel API—Polyfill: <http://mng.bz/YrWK>.

The biggest benefit of this approach compared to the DOM-based Custom Events is the fact that you can exchange messages across windows. This can come in handy if you need to sync state across multiple tabs or decide to use iframes. You can also use the concept of named channels to explicitly differentiate between team-internal and public communication. In addition to the global `tractor_channel`, Team Checkout could open its own `checkout_channel` for communication between the team's own micro frontends. This team-internal communication may also contain more complex data structures. Having a clear distinction between public and internal messages reduces the risk of unwanted coupling.

6.1.5 When UI communication is a good fit

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

USE SIMPLE PAYLOADS

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

THE NEED FOR INTENSE UI COMMUNICATION CAN BE A SIGN OF BAD BOUNDARIES

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

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

EVENTS VERSUS ASYNCHRONOUS LOADING

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

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

6.2 Other communication mechanisms

So far, we've focused on user interface communication, which happens directly between micro frontends in the browser. However, there are other types of data exchange you have to solve when you build a real application. In the last part of this chapter, we'll discuss how authentication, data fetching, state management, and data replication fit into the micro frontends picture.

6.2.1 Global context and authentication

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

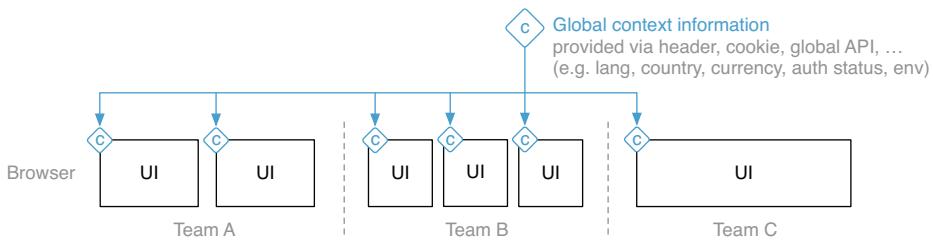


Figure 6.9 You can provide general context information globally to all micro frontends. This puts common tasks like language detection in a central place.

PROVIDING CONTEXT INFORMATION TO ALL MICRO FRONTENDS

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

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

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

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

AUTHENTICATION

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

From a technical integration standpoint, the team that owns the login process becomes the authentication provider for the other teams. It provides a login page or fragment that other teams can use to redirect an unauthenticated user towards. You can use standards like OAuth⁶ or JSON Web Tokens (JWT) to securely provide the authentication status to the teams that need it.

6.2.2 Managing state

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

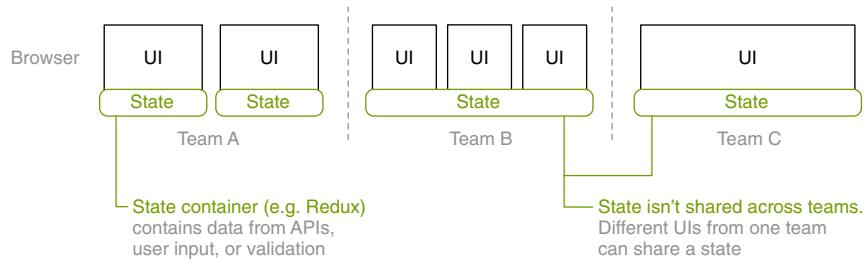


Figure 6.10 Each team has its own user interface state. Sharing state between teams would introduce coupling and make the applications hard to change later on.

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

⁶ See <https://en.wikipedia.org/wiki/OAuth>.

6.2.3 Frontend-backend communication

To do its work, a micro frontend should only talk to the backend infrastructure of its team, as shown in figure 6.11. 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 and test your system, the system from the other team needs to be present. An error in Team B would also affect fragments from Team C.

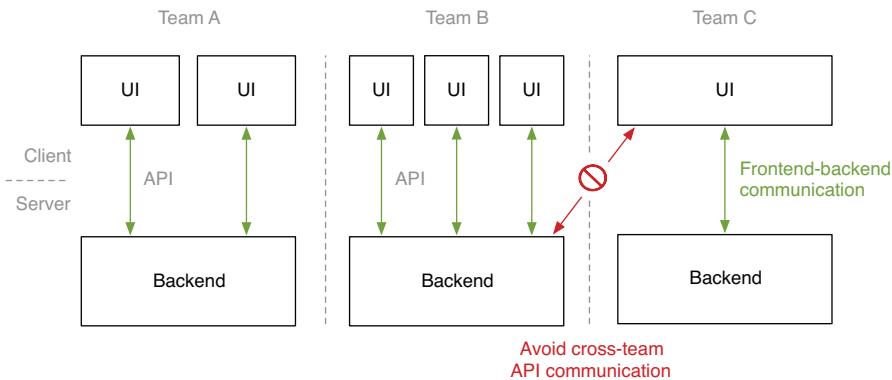


Figure 6.11 API communication should always stay inside team boundaries.

6.2.4 Data replication

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

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

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

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

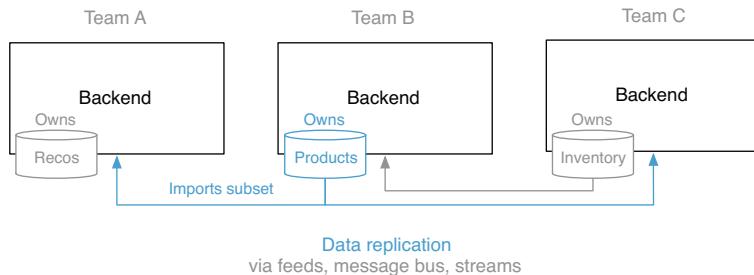


Figure 6.12 Teams can replicate data from other teams to stay independent. This replication increases robustness. If one team goes down, the others can still function.

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

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

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

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

Summary

- Communication between different micro frontends is often necessary at the handover points in your application. When the user moves from one use case to the next, you can handle most communication needs by passing parameters through the URL.

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



Client-side routing and the application shell

This chapter covers

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

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

In chapter 2 we covered the most basic page-integration technique: the plain old link. Later, in chapter 3, 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 (SPAs).

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

- *Hard navigation* describes a page transition where the browser loads the complete HTML for the next page from the server.
- *Soft navigation* refers to a page transition that's entirely client-side rendered, typically by using a client-side router. In this scenario the client fetches its data via an API from the server.

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

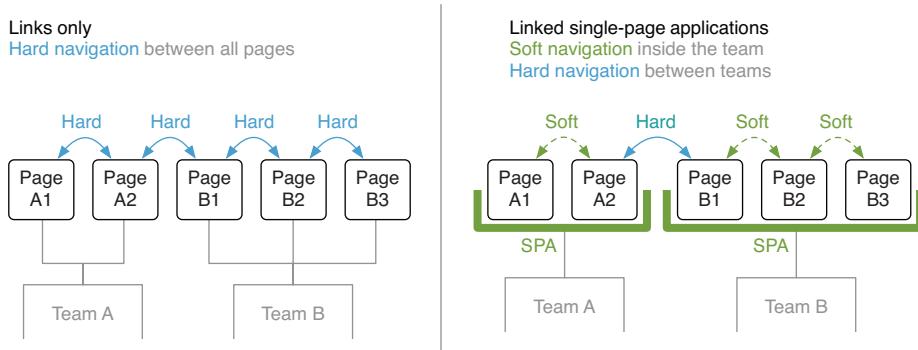


Figure 7.1 Two different approaches to page transitions in a micro frontends context. The “links only” model is simple. Page transitions happen via plain links, which result in a full refresh of the page. Nothing special is needed—Team A must know how to link to the pages of Team B and vice versa. With 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, it's identical to the first approach. The fact that a team uses a SPA for its pages is an implementation detail. As long as it responds correctly to URLs, the other team doesn't have to care.

In these options, the link is the only contract between the teams. There is no other technical requirement or shared code needed to make it work. However, both versions include hard navigations. Whether this is acceptable depends on your use case and especially the number of teams. When your goal is a setup with many teams that are each responsible for only one page, you end up with a lot of hard navigations. Figure 7.2 shows a third option where all page transitions are soft.

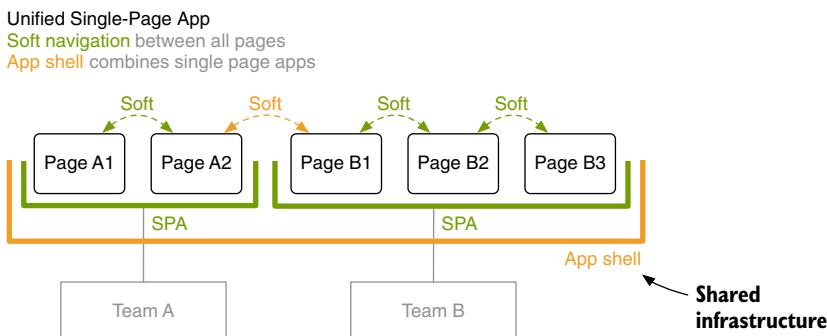


Figure 7.2 The Unified Single Page App approach introduces a central application container. It handles page transitions between the teams. Here all navigations are soft.

To remove hard navigations between the teams, we need to establish a new shared piece of infrastructure: an *application shell*, or *app shell* for short. Its job is to map URLs to the correct team. In this regard, the application shell is similar to the frontend proxy we covered in chapter 3. From a technology perspective, it's different. We don't need a dedicated server like Nginx. The application shell consists of an HTML document and a piece of JavaScript.

In this chapter you'll learn how to combine different SPAs into a unified single-page app using an application shell. We'll build an application shell from scratch. It contains a simple router that we later upgrade to a more sophisticated and maintainable version. At the end of the chapter, we look at the micro frontends meta-framework *single-spa*, which is an out-of-the-box app shell solution.

7.1 App shell with flat routing

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

In a company-wide meeting, they discussed the idea of moving to a full client-rendered user interface. Soft navigation should be possible across all pages, not only inside team boundaries. In a monolithic world, this would be straightforward: use the router of your favorite JavaScript framework—you're done. However, they don't want

to introduce stronger coupling between the teams. Independent deployments and dependency upgrades should continue to be possible to ensure fast iteration. Moving to one shared framework would do the opposite.

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

7.1.1 What's an app shell?

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

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.

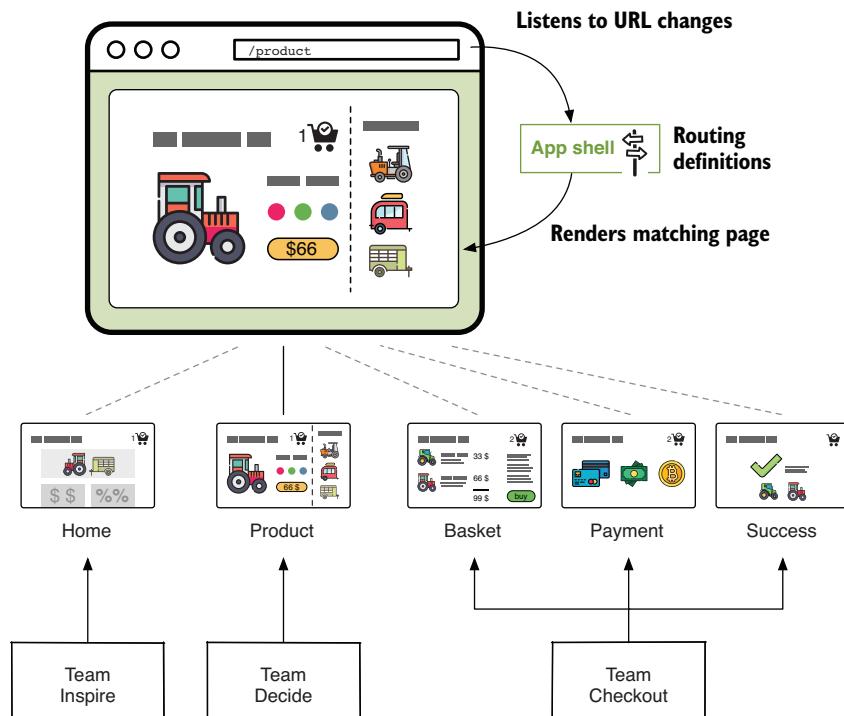


Figure 7.3 The app shell acts as a central client-side router. It watches for URL changes, determines the matching page (micro frontend), and renders it.

7.1.2 Anatomy of the app shell

The four essential parts of a micro frontends app shell are

- 1 Providing a shared HTML document
- 2 Mapping URLs to team pages (client-side routing)
- 3 Rendering the matching page
- 4 (De)initializing the previous/next page on navigation

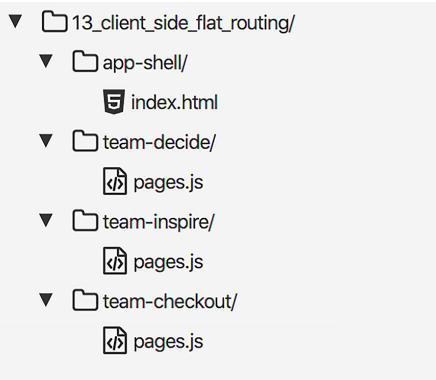


Figure 7.4 The app shell's code is located beside the team's code. It provides a shared HTML document. The teams just deliver page components via JavaScript.

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

```
npm run 13_client_side_flat_routing
```



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

Let's look at the markup in the following listing.

Listing 7.1 `app-shell/index.html`

```

<html>
  <head>
    <title>The Tractor Store</title>
    <script src="https://unpkg.com/history@4.9.0"></script>
    <script src="http://localhost:3001/pages.js" async></script>
    <script src="http://localhost:3002/pages.js" async></script>
    <script src="http://localhost:3003/pages.js" async></script>
  </head>
  
```

A dependency we'll use
in the router code

The application
code for all teams

```
<body>
  <div id="app-content">
    <span>rendered page goes here</span>
  </div>
  <script type="module">
    /* routing code goes here */
  </script>
</body>
</html>
```

Container for the actual page content

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 straightforward. Let's get to the exciting part: the routing.

7.1.3 Client-side routing

There are many ways to build a client-side router. We could use a full-featured existing routing solution like `vue-router`. However, since we want to keep it simple, we'll build our own that's based on the `history` library.¹ This library is a thin wrapper around the browser's History API. Many higher-level routers like `react-router` use it under the hood. Don't worry if you haven't used `history` before. We'll only use two features: `listen` and `push`.

Listing 7.2 app-shell/index.html

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

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

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

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

const appHistory = window.history.createBrowserHistory(); ←
  ↪ Instantiates the history library
```

Maps a URL path to the component name

Looks up a component based on a pathname

Writes the component name into the content container

¹ See <https://github.com/ReactTraining/history>.

```

Registers a history listener that's called every time
the URL changes either through a push/replace call
or by clicking the browser's Back/Forward controls
appHistory.listen(updatePageComponent);
updatePageComponent(window.location); ← Calls the update function once on
                                         start to render the first page

document.addEventListener("click", e => {
  if (e.target.nodeName === "A") {
    const href = e.target.getAttribute("href");
    appHistory.push(href);
    e.preventDefault();
  }
}); ← Registers a global click listener
      that intercepts link clicks, passes
      the target URLs to the history,
      and prevents a hard navigation
...

```

KEEPING URL AND CONTENT IN SYNC

The central piece is the `updatePageComponent(location)` function. It keeps the displayed content in sync with the browser's URL. It's called once on initialization and every time the browser history changes (`appHistory.listen`). The change can be due to a navigation request through the JavaScript API via `appHistory.push()` or when the user clicks the Back or Forward button in the browser. The `updatePageComponent` function looks up the page component that matches the current URL. For now it puts the component name into the `div#app-content` element via `innerHTML`. This way, the browser shows one line of text which contains the matched name. The name acts as a placeholder for us. We'll upgrade this to rendering a real component in a minute.

MAPPING URLs TO COMPONENTS

The `routes` object is a simple pathname (key) to component name (value) mapping. Here is an excerpt from the code you saw before.

Listing 7.3 `app-shell/index.html`

```

...
const routes = {
  "/": "inspire-home",
  "/product/porsche": "decide-product-porsche",
  ...
  "/checkout/pay": "checkout-pay",
  "/checkout/success": "checkout-success"
};
...

```

So every page is a component. The component's name starts with the name of the responsible team. For the URL `/checkout/success`, the app shell should render the `checkout-success` component, which Team Checkout owns.

7.1.4 Rendering pages

The app shell includes a JavaScript file from each team. Let's have a look inside these files. As you might have guessed, we are using Web Components as a neutral

component format. A team exposes their page as a Custom Element. The app shell needs to know the name of this component. It doesn't care what technology the page component uses internally. We use the same approach as discussed in chapter 5, but on a page- and not on a fragment-level. The following code shows Team Inspire's homepage component.

Listing 7.4 team-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>
      <a href="/product/eicher">Eicher</a>
      <a href="/product/fendt">Fendt</a>
    ;
  }
}

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

Annotations for Listing 7.4:

- A callout points to the three links ([Porsche](#), [Eicher](#), [Fendt](#)) with the text "Links to the product page owned by Team Decide".
- A callout points to the final line of code (`window.customElements.define`) with the text "Adds the Custom Element to the global registry".

This is 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's an example for a product page.

Listing 7.5 decide/pages.js

```
class DecideProductPorsche extends HTMLElement {
  connectedCallback() {
    this.innerHTML =
      <a href="/">&lt; home</a> -
      <a href="/checkout/cart">view cart &gt;</a>
      <h1>Porsche-Diesel Master 419</h1>
      
    ;
  }
}

window.customElements.define(
  "decide-product-porsche",
  DecideProductPorsche
);
...
```

Annotations for Listing 7.5:

- A callout points to the first link ([home](#)) with the text "Links to Team Inspire's homepage".
- A callout points to the second link ([view cart](#)) with the text "Links to Team Checkout's cart page".
- A callout points to the final line of code (`window.customElements.define`) with the text "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 doesn't just display the component name.

Listing 7.6 app-shell/index.html

```

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

```

Looks up the component name for the current location

Reference to the existing page component

Instantiates the Custom Element

Replaces the existing component with the new one (disconnectedCallback of the old one and connectedCallback of the new one are triggered)

The preceding 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 Element’s 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 earlier.

Listing 7.7 app-shell/index.html

```

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

```

Adds a click listener to the complete document

Only cares about “a” tags

Extracts the link target from href

Pushes the new URL to the history

Stops the browser from performing a hard navigation

NOTE This is a shortened version of a global click handler. In production, you’d also want to watch for modifier keys to make opening in a new tab possible. You might also want to detect external links. But you get the gist.

This click handler intercepts clicks on links that are rendered by the individual micro frontends. Instead of triggering a full page load, the browser performs a soft navigation:

- The target URL becomes the latest entry in the history stack (`appHistory.push(href)`).

- The `appHistory.listen(updatePageComponent)` callback triggers.
- `updatePageComponent` matches the new URL against the routing table to determine the new component name.
- `updatePageComponent` replaces the existing component with the new one.
- The `disconnectedCallback` of the old component triggers (if implemented).
- The constructor and `connectedCallback` of the new component trigger.

When you start the example code and open `http://localhost:3000/`, you can see this code in action. Click on the links to navigate between the pages. All page transitions are entirely client-side. The app shell document doesn't reload at any time. Figure 7.5 illustrates the links between the pages in the example project.

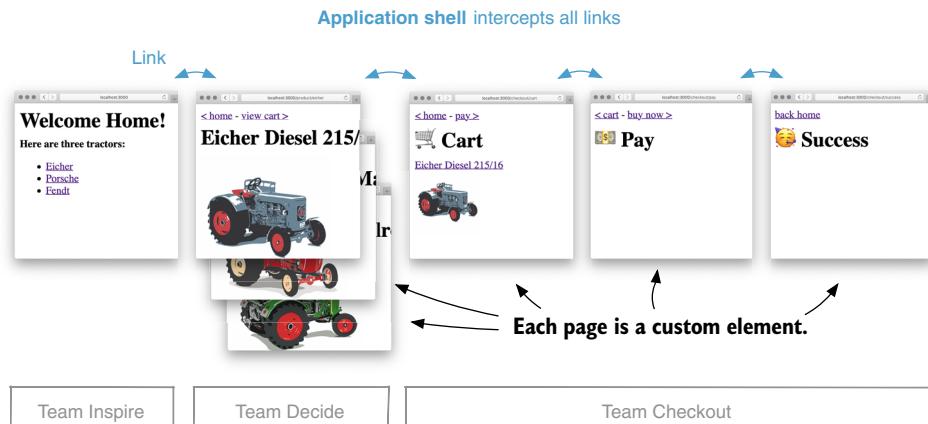


Figure 7.5 The pages in the example project are connected via links. The application shell intercepts these links and performs a soft navigation to the requested page. Teams expose their pages as Custom Elements. On navigation, the app shell replaces the existing page component with the new one.

You should take some time and play around with the code. Add log statements or debugger breakpoints to the app shell and page component code. It gives you a feeling of how our routing code plays together with the (de)initialization of the pages.

7.1.5 Contracts between app shell and teams

Let's take a step back and look at the contracts between the teams and the app shell (see figure 7.6). 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 other team's component names. The application shell encapsulates this information. When a team wants to change the name of a component, it must only update the app shell.

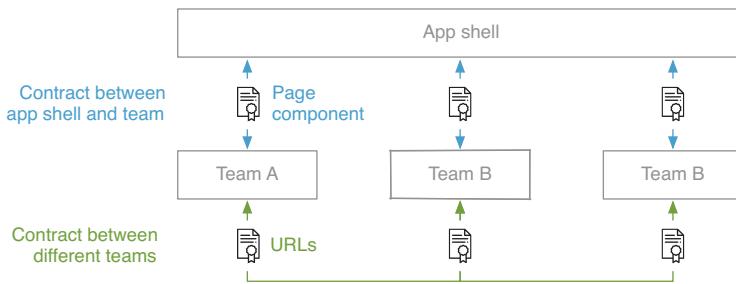


Figure 7.6 Contracts between the systems. Teams need to expose their pages to the app shell in a defined component format (for example, Web Components). A team needs to know the URL of another team if it wants to link to it.

7.2 App shell with two-level routing

The teams are happy with their first app shell prototype. It required less code than expected. However, they already spotted a significant downside. *The flat routing approach requires that the app shell must know all URLs of the application.* When a team wants to change an existing or add a new URL, they also need to adjust and redeploy the app shell. This coupling between the feature teams and the app shell does not feel right. The shell should be a piece of infrastructure that's as neutral as possible. It shouldn't need to know every URL that exists in the application.

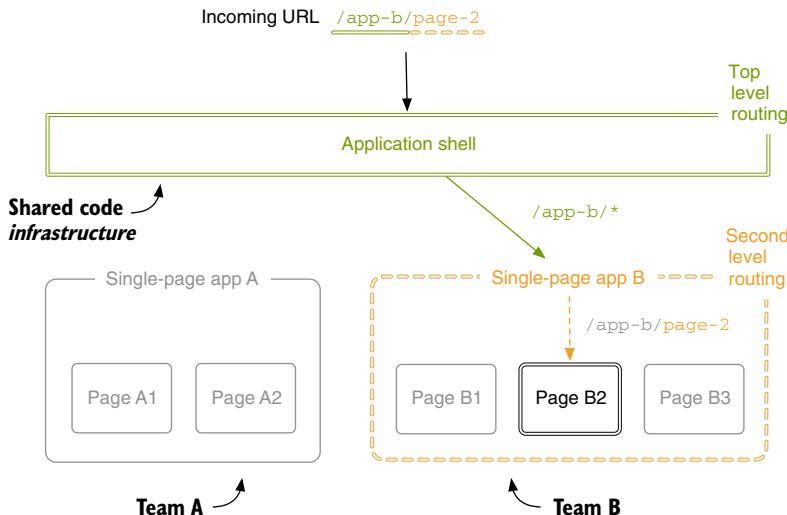


Figure 7.7 Two-level routing. The app shell looks at the first part of the URL to determine which team is responsible (top-level routing). The router of the matched team processes the complete URL to find the correct page inside its single-page application (second-level routing).

The concept of two-level routing circumvents this. Here the app shell only routes between teams. Each team can have its own router that maps the incoming URL to a specific page. It's the same concept you learned in chapter 3, but moved from the web server to JavaScript in the browser.

For this to work, the app shell needs a reliable way to tell which team owns a specific URL. The easiest way to achieve this is by using a prefix. Figure 7.7 illustrates how this works.

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. The top router decides which team is responsible. The actual route definitions move into the responsible team applications. A team can add new URLs inside its application without changing the app shell. They can add it to their router. The app shell only needs to change if you want to introduce a new team or change a team prefix.

Let's go ahead and implement these changes.

7.2.1 Implementing the top-level router

The app shell script can stay the same. We only have to change the routing definitions.

Listing 7.8 app-shell/index.html

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

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

The routes object now maps a URL prefix to a team-level component.

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 more compact 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 can stay the same.

7.2.2 Implementing team-level routing

Let's look inside the `checkout-pages` component to see the second-level routing. This new component takes the role of the `checkout-cart`, `checkout-pay`, and `checkout-success` components from the previous example. Here is Team Checkout's code for handling the pages.

Listing 7.9 `checkout/pages.js`

```
const routes = {  
  "/checkout/cart": () => `<a href="/">&lt; home</a> -  
    <a href="/checkout/pay">pay &gt;</a>  
    <h1> Cart</h1>  
    <a href="/product/eicher">...</a>`,  
  "/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() {  
    this.render(window.location);  
    this.unlisten = window.appHistory.listen(location =>  
      this.render(location)  
    );  
  }  
  render(location) {  
    const route = routes[location.pathname];  
    this.innerHTML = route();  
  }  
  disconnectedCallback() {  
    this.unlisten();  
  }  
}  
  
window.customElements.define("checkout-pages", CheckoutPages);
```

The diagram shows the `checkout/pages.js` code with various annotations:

- Contains all of Team Checkout's routes**: Points to the `routes` object definition.
- Maps the URL of the cart page to a templating function**: Points to the first entry in the `routes` object, which contains a template string starting with `(` and ending with `)`.
- The template for the cart page**: Points to the inner content of the first entry in the `routes` object.
- Looks up the page template via the incoming pathname**: Points to the key in the `routes` object, such as `"/checkout/cart"`.
- Listens to changes in the history and rerenders on change (notice that we are using the `appHistory` instance provided by the app shell)**: Points to the `listen` call in the `connectedCallback` method.
- Triggers when the app shell appends the <checkout-pages> component to the DOM**: Points to the `render` call in the `connectedCallback` method.
- Renders content based on the current location**: Points to the `innerHTML` assignment in the `render` method.
- Responsible for rendering the content**: Points to the `render` method definition.
- Executes the route template and writes the result into `innerHTML`**: Points to the `route()` call in the `render` method.
- Triggers when the app shell removes the component from the DOM and unregisters the before added history listener**: Points to the `unlisten` assignment in the `disconnectedCallback` method.
- Exposes the component as `checkout-pages` to the global Custom Elements registry**: Points to the `customElements.define` call at the bottom of the file.

This code contains the template of all three pages. The `connectedCallback` method triggers when the app shell appends the component to the DOM. It renders the pages based on the current URL. Then it listens for URL changes (`window.appHistory.listen`).

When a location changes, it updates the view accordingly. For simplicity, we use a simple string-based template. In a real application, you'd probably go for a more sophisticated option.

CLEANUP IS KING

It's always good to clean up after you've finished. However, it is extra vital in this micro frontend setup. Running the app shell model is like sharing an apartment with other people. Global variables, forgotten timers, and event listeners may get in the way of other teams or cause memory leaks. These problems are often hard to track down.

It's essential to do proper cleanup when the component isn't in use anymore to avoid issues. That's why in our example the `disconnectedCallback()` removes the history listener via the `unlisten()` function that was returned by the `appHistory.listen()` call.

Be careful when using third-party code. Older jQuery plugins or frameworks like AngularJS (v1) are known for lousy cleanup behavior. However, most modern tools behave well when you unmount them correctly.

That's everything we need to make our two-level routing work. In the first level, the application shell decides which team is responsible. In the second level, the team selects the appropriate page. Take some time and run the example locally for a more in-depth analysis:

```
npm run 14_client_side_two_level_routing
```



7.2.3 What happens on a URL change?

Let's examine what happens when a URL changes. We'll look at three scenarios: first page load, navigation inside team boundaries, and navigation across boundaries.

SCENARIO 1: FIRST VIEW

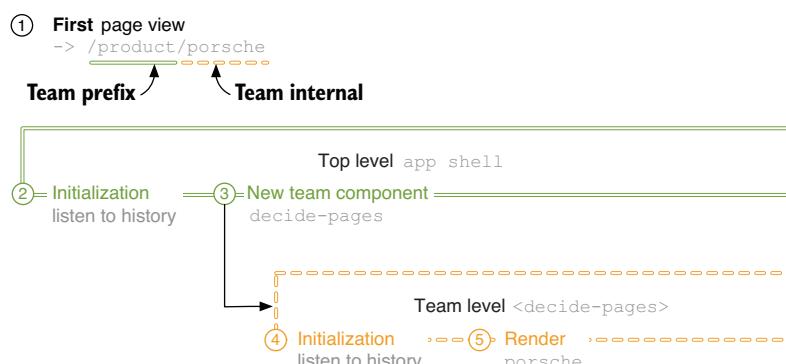


Figure 7.8 First page view in a two-level routing approach. The top-level router looks at the team prefix to determine the responsible team. The team router at the second level looks at the last part of the URL to render the actual page.

Figure 7.8 shows the first page load. Beginning with step 2:

- 2 The app shell code runs first and does everything needed for initialization. It starts watching the URL for changes.
- 3 The current URL starts with the team prefix `/product/`. This prefix maps to Team Decide's `<decide-pages>` element. The app shell inserts this component to the DOM.
- 4 The team-level component initializes itself. It also starts listening to the URL.
- 5 It looks at the current URL and renders the product page for the Porsche tractor.

In short, the app shell picks the team that's responsible for the current URL, and this team renders the page. Both have registered a listener to the URL. In the next scenarios, we'll see the listeners in action.

SCENARIO 2: INSIDE TEAM NAVIGATION

Figure 7.9 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.

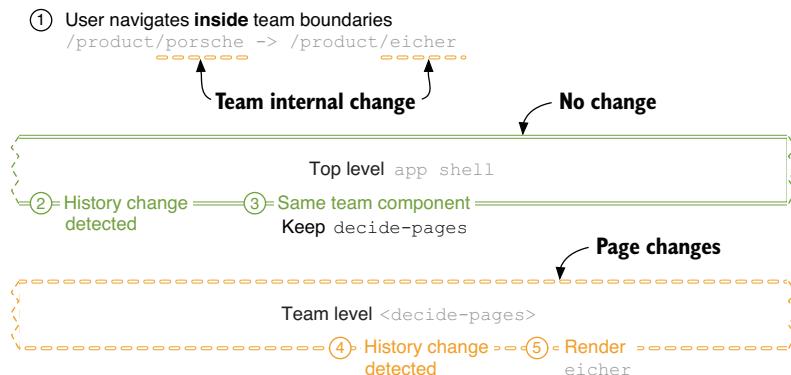


Figure 7.9 When the user navigates to another page controlled by the same team, the app shell has nothing to do. The team-level component needs to update the page according to the URL.

- 2 The app shell detects a history change and notices that the team prefix did not change.
- 3 The team level component can stay the same. The app shell has nothing to do.
- 4 The team component registers the URL change too.
- 5 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 Decide handles the page change on its own.

Now to the exciting part: inter-team navigation.

SCENARIO 3: INTER-TEAM NAVIGATION

When the user moves from the product page to the checkout page, they cross a team boundary. Team Checkout owns the cart page. In figure 7.10 you see how the app shell handles this transition.

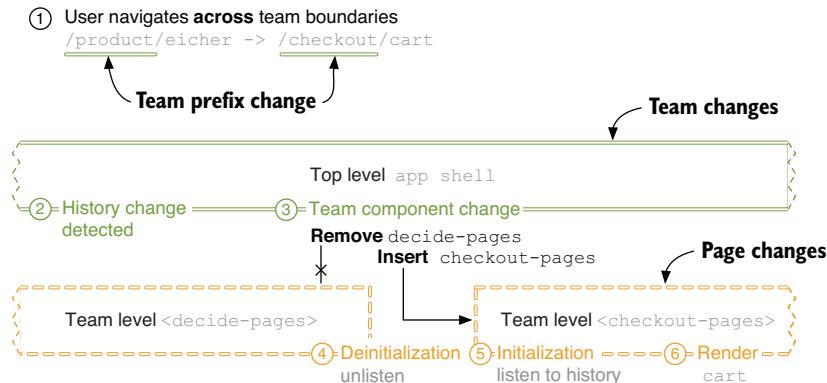


Figure 7.10 On an inter-team navigation, the responsibility changes. The app shell replaces the existing team component with a new one. This new component takes over and is in charge of rendering the page.

- 2 The app shell recognizes a change in history.
- 3 Since the team prefix changed from /product/ to /checkout/, the app shell replaces the existing <decide-pages> component with the new <checkout -pages> component.
- 4 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.
- 5 Team Checkout's component initializes itself and starts listening to the history.
- 6 It renders the cart page.

In this scenario, the app shell swaps the team-level components. It hands over control from one team to another. The team components deal with their initialization and deinitialization.

7.2.4 App shell APIs

You've learned about the app shells most essential tasks:

- Loading the team's application code
- Routing between them based on the URL

Here is a list of additional topics that an app shell might be responsible for:

- Context information (like language, country, tenant)
- Meta-data handling (updating tag, crawler hints, semantic data)

- Authentication
- Polyfills
- Analytics and tag managers
- JavaScript error reporting
- Performance monitoring

Some of these functionalities are not interesting to the application code. Performance monitoring, for example, is often done by adding a script in a central place. The monitoring can work without the application knowing about it.

However, other functionalities can require interaction between the app shell and the applications. The following code shows how a function for tracking events from inside an application might look:

```
window.appShell.analytics({ event: "order_placed" });
```

The app shell can also pass information to the applications. In a web-component-based model it can look like this:

```
<inspire-pages country="CH" language="de"></inspire-pages>
```

It's a good idea to keep this API as lean as possible. Having a stable interface reduces friction. The rollout of breaking API changes comes with inter-team coordination. All teams need to update their code to keep functioning correctly. Figure 7.11 illustrates this contract between the application shell and the team's applications.

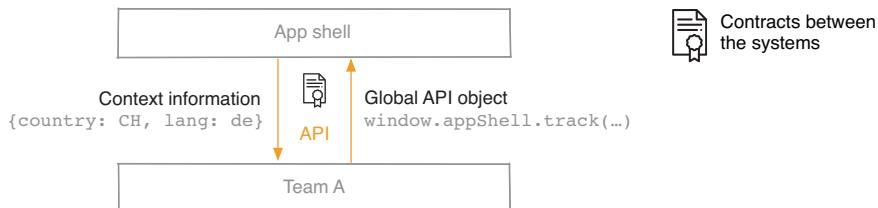


Figure 7.11 Adding shared functionality to the app shell leads to tighter coupling. The API between app shell and team applications acts as a contract between the systems. It should be lean and stable.

Business logic should be in the teams' application code—not in the shared application shell. A good indicator for too-tight coupling is this: *a feature deployment from a team should not require the app shell to change*.

Now we've created a minimal application shell from scratch. Next up, we'll take a quick look into an existing and ready-to-use solution: single-spa.

7.3 A quick look into the single-spa meta-framework

After building and evolving the app shell prototype, Tractor Models, Inc.'s development teams have a pretty good understanding of how the pieces work together. They

know for sure that they want to go with the two-level routing model. However, there are still some features missing. Lazy loading of JavaScript code and proper error handling are two of them.

To avoid reinventing the wheel, they check for existing solutions that fit their needs. They come across single-spa,² which is a popular micro frontends meta-framework. In essence, it's an application shell—similar to the application shell we just built. But it comes with some more advanced features. It has built-in *on-demand loading of application code* and comes with a *broad ecosystem of framework bindings*. You can find examples and helper libraries to hook up a React, Vue.js, Angular, Svelte, or Cycle.js application with few efforts. These make it easy to expose an application in a unified way so that single-spa can interact with them.

The teams want to take their prototype and migrate it to single-spa. To test out the limits, each team chooses another JavaScript framework for their part of the shop. Team Inspire implements the Homepage using Svelte.js, Team Decide renders the product pages using React.js, and Team Checkout opts for the Vue.js framework. Figure 7.12 illustrates this. They don't plan to go to production with this technology mix, but it's an excellent exercise to see how the integration works.

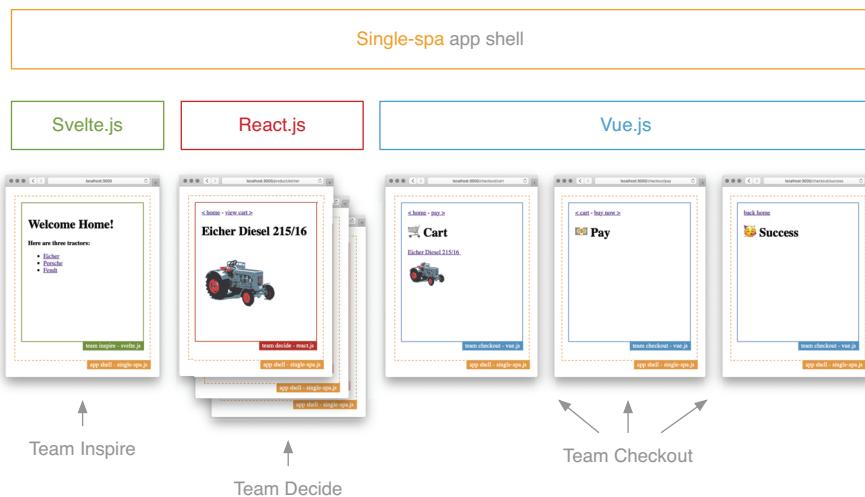


Figure 7.12 Single-spa acts as the application shell which routes between the applications. In our example, all teams have picked a different frontend framework for their application code.

Let's look at how single-spa works.

7.3.1 How single-spa works

TIP You can find the sample code for this task in the `15_single_spa` folder.

² See <https://single-spa.js.org/>.

The basic concepts are the same as in our previous prototype. We have a single HTML file that acts as the starting point. It includes the single-spa JavaScript code and maps URL prefixes to the code of a specific application. The main difference is that it does not use Web Components as the component format. Instead, the teams expose their micro frontends as a JavaScript object that adheres to a specific interface. We'll look at this in a minute. Let's look at the initialization code first.

Listing 7.10 app-shell/index.html

```

<html>
  <head>
    <title>The Tractor Store</title>
    <script src="/single-spa.js"></script> ← Imports the
  </head>
  <body>
    <div id="app-inspire"></div>
    <div id="app-decide"></div>
    <div id="app-checkout"></div> ← Each micro frontend has its own DOM
                                   element which acts as the mount point.

  Name of the
  application,
  which makes
  debugging
  easier
    <script type="module">
      singleSpa.registerApplication( ← Registers a micro frontend
        "inspire",
        () => import("http://localhost:3002/pages.min.js"),
        ({ pathname }) => pathname === "/" ← The activity function receives the
                                                location and determines if the micro
                                                frontend should be active or not.
      );
      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(); ← Loading function for
    </script>           the application,
  </body>             which fetches the
</html>               associated JavaScript
                           code when needed

```

← 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 deinitialize itself.

More than one application may be active at the same time. A typical use case for this is global navigation. It can be a dedicated micro frontend that gets mounted at the top and is active on all routes.

JAVASCRIPT MODULES AS THE COMPONENT FORMAT

In contrast to our Web Component-based prototype, single-spa uses a JavaScript interface as the contract between app shell and team application. An application has to provide three asynchronous functions. It looks like this.

Listing 7.11 team-a/pages.js

```
export async function bootstrap() { ... }
export async function mount() { ... }
export async function unmount() { ... }
```

These functions (`bootstrap`, `mount`, `unmount`) are similar to the Custom Elements lifecycle functions (`constructor`, `connectedCallback`, `disconnectedCallback`). Single-spa calls `bootstrap` when a micro frontend becomes active for the first time. It invokes `(un)mount` every time the application is `(de)`activated.

All lifecycle functions are asynchronous. This fact makes lazy loading and data fetching inside an application a lot easier. Single-spa ensures that `mount` is not called before `bootstrap` has completed.

The Custom Elements lifecycle methods are synchronous. Implementing asynchronous initialization with Custom Elements is possible. However, it requires some extra work on top of what the standard specifies.

FRAMEWORK ADAPTERS

Single-spa comes with a list of framework adapters. Their job is to wire the three lifecycle methods to the appropriate framework calls for `(de)`initialization. Let's 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 used Svelte before. It's a simple example.

Listing 7.12 team-inspire/pages.js

```
import singleSpaSvelte from "single-spa-svelte";
import Homepage from "./Homepage.svelte";
```

Imports single-spa's Svelte adapter

Imports the Svelte component for rendering the homepage

```

const svelteLifecycles = singleSpaSvelte({
  component: Homepage,
  domElementGetter: () => document.getElementById("app-inspire")
});

export const { bootstrap, mount, unmount } = svelteLifecycles;

```

Calls the adapter with the root component and a function that retrieves the DOM element to render it in

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 look at the homepage component.

Listing 7.13 `team-inspire/Homepage.svelte`

```

<script>
  function navigate(e) {
    e.preventDefault();
    const href = e.target.getAttribute("href");
    window.history.pushState(null, null, href);
  }
</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>
  <a on:click={navigate} href="/product/porsche">Porsche</a>
  <a on:click={navigate} href="/product/fendt">Fendt</a>
</div>

```

Function that intercepts link clicks by pushing the URL to the history and preventing a reload

Links to Team Decide's product page

In the 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.

Clicking on this product link triggers the termination (`unmount`) of Team Inspire's micro frontend. After that, single-spa loads Team Decide's application and activates it (`mount`). This behavior is similar to the inter-team navigation scenario you saw in the previous section.

RUNNING THE APPLICATION

You can fire up the sample code by running the following command:

```
npm run 15_single_spa
```



It starts four web servers (app shell and three applications) and opens your browser at `http://localhost:3000/`. Have a look at the developer tools when navigating through the shop using the links.

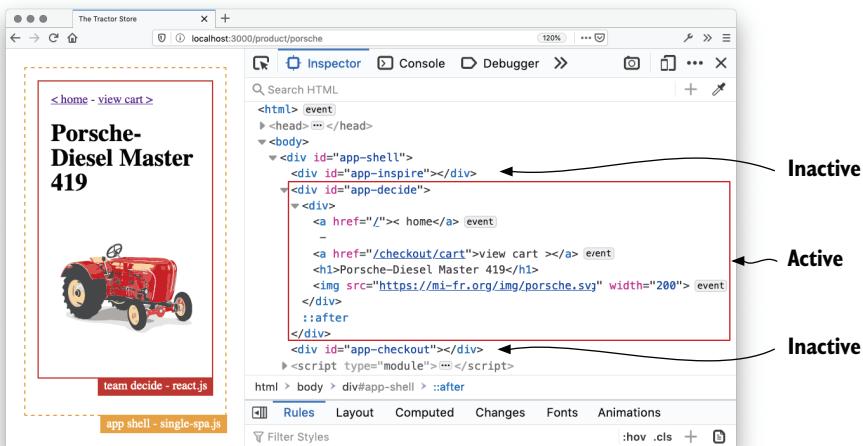


Figure 7.13 With single-spa, each micro frontend has its own DOM node to render its content. In this example, the micro frontend for Team Decide's product page is active. It shows its content inside the `#app-decide` element. The other micro frontends are inactive, and their corresponding DOM elements are empty.

- See how the `#app-inspire`, `#app-decide`, and `#app-checkout` DOM nodes of the app shell come to life. When the user moves from one micro frontend to the next, the content changes. The old micro frontend removes its markup. The new micro frontend fills its DOM node with the new content. You can see this in figure 7.13.
- Open the Network tab and also notice that single-spa loads the JavaScript bundles (`pages.min.js`) as they are needed and not all up front.
- Have a look at the code of Team Decide's and Team Checkout's micro frontends. They both include a framework-level router (`react-router` and `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 `<routerr-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 I 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 called fragments in the last few 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 portals I just mentioned, there are status events, the ability to pass down context information, and ways to deal with errors.

The official documentation³ is an excellent place to start to get deeper into single-spa. They have a lot of good examples showcasing how to use single-spa with different frameworks.

7.4 **The challenges of a unified single-page app**

Now you have a good understanding of what's necessary to build an app shell that connects different single-page apps. The unified model makes it possible for the user to move through the complete app without encountering a hard navigation. All page transitions are client-rendered, which in general results in quicker responses to user interactions.

7.4.1 **Topics you need to think about**

However, the improvement in user experience does not come for free. Here are a couple of topics you need to address when going the unified single-page app route.

SHARED HTML DOCUMENT AND META DATA

The teams have no control over the surrounding HTML document. A micro frontend may only change the content inside of its root DOM node in the body.

You almost 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, the title is often not enough. You want to provide crawlers and preview generators like Facebook or Slack with machine-readable information like canonicals, href langs, schema.org tags, and indexing hints. Some of these might be the same for the complete site. Others are highly specific to one page type.

Coming up with a mechanism to effectively manage meta tags across all micro frontends requires some extra work and complexity. Think of Angular's meta service,⁴ vue-meta,⁵ or react-helmet⁶ but on an app shell level.

³ See <https://single-spa.js.org/docs/getting-started-overview.html>.

⁴ See <https://angular.io/api/platform-browser/Meta>.

⁵ See <https://vue-meta.nuxtjs.org>.

⁶ See <https://github.com/nfl/react-helmet>.

ERROR BOUNDARIES

If the code from different teams runs inside one document, it can sometimes be tricky to find out where an error originated. In the composition approach from the last chapter, we have the same problem. Code from inside a fragment has the potential to cause unwanted behavior on the complete page. However, the unified single-page app model widens the debugging area from page level to the complete application. A forgotten scroll listener from the homepage can introduce a bug on the confirmation page in the checkout. Since these pages are not owned by the same team, it can be hard to make the connection when looking for the error.

In practice, these types of problems are rather rare. Also, error reporting and browser debugging tools have gotten pretty good over the last years. Identifying which JavaScript file caused the error helps in finding the responsible team.

MEMORY MANAGEMENT

Finding memory leaks is more complicated than tracking down a JavaScript error. A common cause of 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 into a bigger problem.

Single-spa has a plugin called `single-spa-leaked-globals` which tries to clean up global variables after a micro frontend is 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 its nature, the single first point of contact. Having a severe error in the app shell can bring down the complete application. That's why your app shell code should be of high quality and well tested. Keeping it focused and lean helps in achieving this.

APP SHELL OWNERSHIP

Similar to the frontend proxy we talked about in chapter 3, the application shell is a critical piece of infrastructure that needs a clear owner. However, once you have a working system, there should not be a high demand for adding features or constantly evolving the application shell itself. If your app shell is lean, it's usually perfectly fine that one of the feature teams takes responsibility for maintaining it. In chapter 13 we'll dive a little deeper into this topic.

COMMUNICATION

Sometimes micro frontend A needs to know something that happened in micro frontend B. The same communication rules we discussed in chapter 6 also apply here:

- Avoid inter-team communication when possible.
- Transport context information via the URL.

- Stick to simple notifications when needed.
- Prefer API communication to your backend.

Don't move state to the app shell. It might sound like a good idea to not load the same information twice from the server. However, misusing the app shell as a state container creates strong coupling between the micro frontends. In the backend world, it's a best practice that microservices don't share a database. One change in a central database table has the potential to break another service. The same applies to micro frontends. Here your state container is equivalent to a database.

BOOT TIME

Code splitting has become best practice in web development. When implementing an app shell, you should consider this as well. In the single-spa example, you saw how the library loads the actual micro frontend code on-demand. It's crucial to think about optimizations to deliver an excellent overall performance.

7.4.2 When does a unified single-page app make sense?

This model plays to its 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 they find something they like. In this case, using a soft navigation makes a noticeable difference in the user experience.

For web applications *where providing a high amount of interactivity is more important than initial page load time*, the unified single-page app approach is a good fit. Sites that require the user to log in before using it and classical back-office applications are prime candidates.

However, as already discussed, this approach does not come for free and introduces a considerable amount of shared complexity. If you want to split your existing single-page application into smaller ones, the unified single-page application approach is not necessarily the way to go. For many use cases, it's totally fine to have a hard navigation between two linked single-page apps.

Imagine a content management application with an area for writing long-form articles and another area to moderate comments. These can be two independent single-page applications. Since a typical user would not always switch from moderating comments to writing an article, it might be perfectly fine to build this as two distinct applications that both include the same header fragment via composition.

Figure 7.14 shows the trade-off between providing the best user experience and having a simple setup with low coupling.

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 few chapters. See the result in figure 7.15.

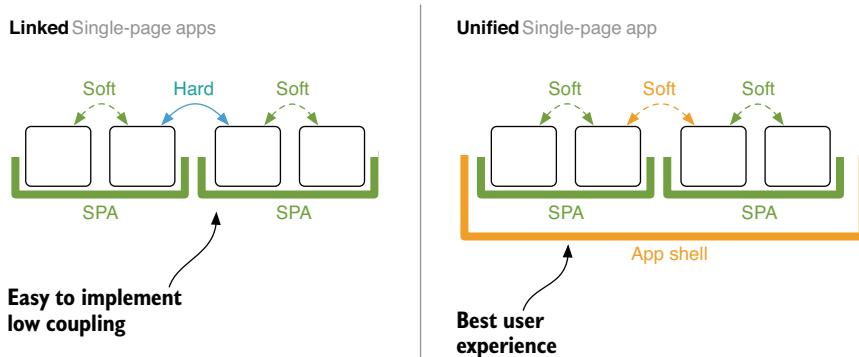


Figure 7.14 Linked single-page apps are easy to build and introduce low coupling. However, they require a hard navigation when moving from one app to the other. The unified single-page app approach solves this and provides a better user experience. But this enhancement does come with some major complexity.

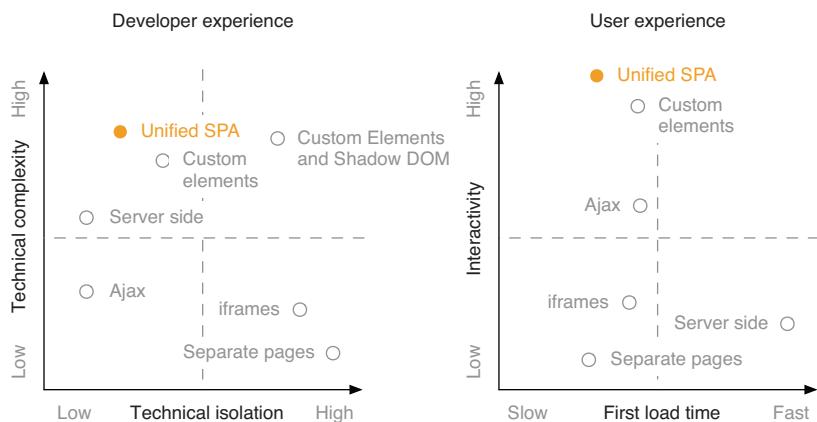


Figure 7.15 Setting up and running a unified single-page app in production is not trivial. Existing libraries like single-spa make it easy to get started. Since all application code lives in the same HTML document, there is no technical isolation. We also have the risk that an error in app A can affect app B. Since a unified single-page application is client-rendered and needs additional app shell code, it has a longer 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.

Summary

- Combining multiple single-page apps requires a shared app shell that handles routing.
- This approach makes it possible to use soft navigations across all pages.

- The app shell is a shared piece of infrastructure and should not contain business logic.
- Deploying a team feature should never require an app shell deployment.
- Having a two-level routing approach where the app shell performs a simple team match and the team's SPA 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.

Composition and universal rendering

This chapter covers

- Employing universal rendering in a micro frontends architecture
- Applying server- and client-side composition in tandem to combine their benefits
- Discovering how to leverage the server-side rendering (SSR) capabilities of modern JavaScript frameworks in a micro frontends context

In the last few chapters, we focused on various integration techniques and discussed their strengths and weaknesses. We grouped them into two categories: server-side and client-side. Integration on the server makes it possible to ship a page that loads quickly and adheres to the principles of progressive enhancement. Client-side integration enables building rich user interfaces where the page can react to user input instantly.

Broad framework support for universal rendering made building applications that run server- and client-side a lot easier for developers. But what do we need to do to integrate multiple universal applications into a big one?

Terminology: Universal, isomorphic, and SSR

The terms *universal rendering*,^a *Isomorphic JavaScript*,^b and *server-side rendering* (SSR) essentially refer to the same concept: Having a single code-base that makes it possible to render and update markup on the server and in the browser. Their meaning or perspective varies in detail. However, in this book, we'll go with the term *universal rendering*.

^a See Michael Jackson, “Universal JavaScript,” *componentDidBlog*, <http://mng.bz/GVvR>

^b See Spike Brehm, “Isomorphic JavaScript: The Future of Web Apps,” *Medium*, <http://mng.bz/zj7X>.

You've already acquired the necessary building blocks. We can combine the client- and server-side composition and routing techniques from the last few chapters to make this happen. Figure 8.1 illustrates how our puzzle pieces fit together.

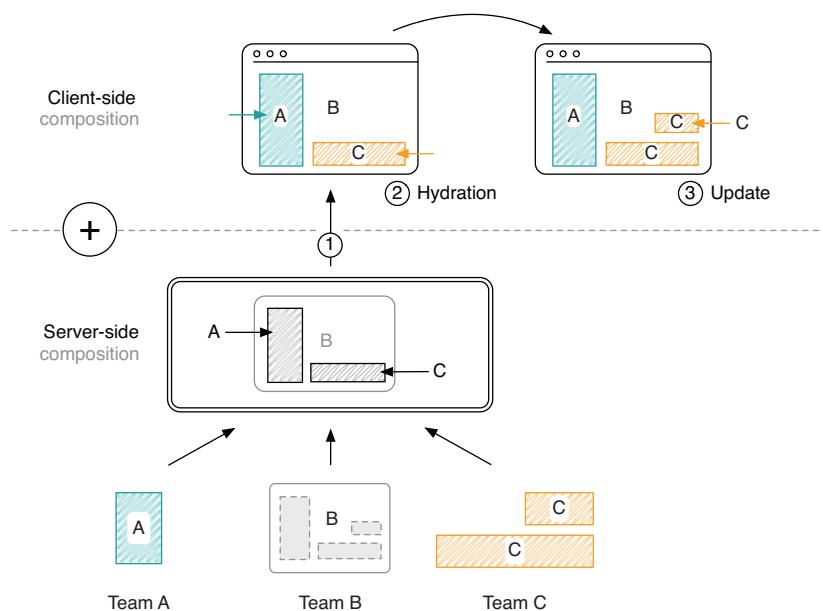


Figure 8.1 Universal composition is the combination of a server- and a client-side composition technique. For the first request, a technique like SSI, ESI, or Podium assembles the markup of all micro frontends server-side. The complete HTML document gets sent to the browser ①. In the browser, each micro frontend hydrates itself and becomes interactive ②. From there on, all user interactions can happen fully client-side. The micro frontends update the markup directly in the browser ③.

NOTE In this chapter, we assume that you’re already familiar with the concept of universal rendering and know what *hydration* is. If not, I recommend reading this blog post for a quick introduction.¹ If you want do dive deeper, you can also check out the book *Isomorphic Web Applications*.²

In this chapter, we’ll upgrade our product detail page. We’ll implement universal rendering for all micro frontends and than apply the required integration techniques to make the site work as a whole.

8.1 Combining server- and client-side composition

Since Team Decide added Team Checkout’s Buy button to the product page, tractor sales skyrocketed. Now hundreds of orders from all over the world arrive every hour. The team behind The Tractor Store was pretty overwhelmed by this success. They had to ramp up their production and logistics capabilities to keep up with the demand. But not everything has been rosy since then. Over recent weeks, the development teams struggled with some serious issues. One day Team Checkout shipped a release of their software that triggered a JavaScript error in all Microsoft Edge browsers. Due to this bug, the Buy button was missing on the page. Sales for that day were down by 34%. This incident showed a significant quality issue, and the team took measures so that this kind of problem wouldn’t strike again.

But this is not the only problem. The product page integrates the Buy button micro frontend using client-side composition via Web Components. The Buy button is not part of the initial markup. Client-side JavaScript renders it. While it loads, the user sees an empty spot where the Buy button will appear after a delay. In local development, this delay is not noticeable. But in the real world, on lower-end smartphones and non-optimal network conditions, it takes a considerable amount of time. Adding new features to the Buy button made this effect even worse. Figure 8.2 shows how the product page looks when JavaScript fails or hasn’t finished loading yet.

The teams decide to switch to a hybrid integration model. Using SSI for server-side composition and also keeping the Web Components composition. This way, the first-page load can be fast, and client-side updating and communication is still possible. Let’s look at this combination.

¹ See Kevin Nguyen, “Universal Javascript in Production—Server/Client Rendering,” *Caffeine Coding*, <http://mng.bz/04jl>.

² Elyse Kolker Gordon, *Isomorphic Web Applications—Universal Development with React*, <http://mng.bz/K2yZ>.

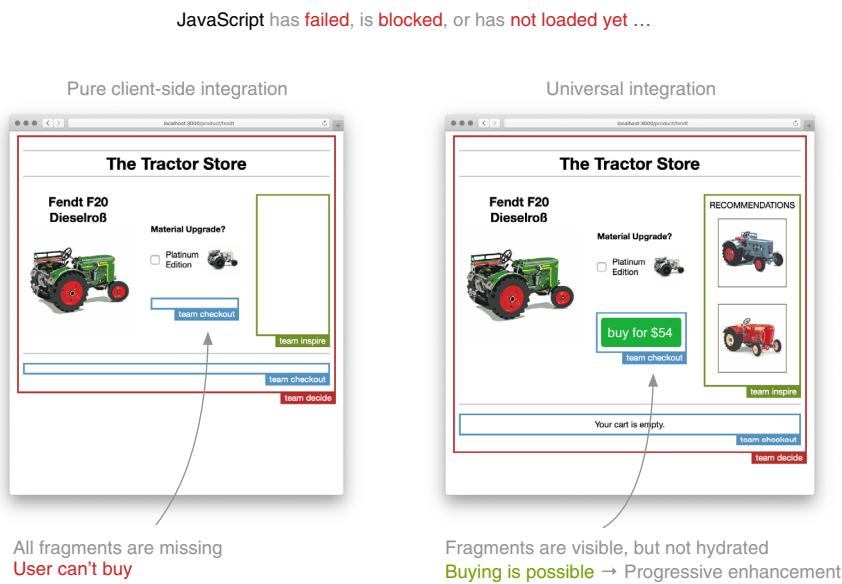


Figure 8.2 Client-side composition requires JavaScript to work. If it fails or takes a long time to load, the included micro frontends are not shown. For the product page, this means that the user can't buy a tractor. Universal composition makes it possible to use progressive enhancement in a micro frontends context. That way, the Buy button can render instantly, and you can make it function even without JavaScript.

8.1.1 SSI and Web Components

In chapter 5, Team Checkout wrapped its Buy button micro frontend into a Custom Element. The browser receives the following HTML markup.

Listing 8.1 team-decide/product/fendt.html

```
...
<checkout-buy sku="fendt"></checkout-buy>
...
```

Since `checkout-buy` is a custom HTML tag, the browser treats it as an empty inline element. At first, the user sees nothing. Client-side JavaScript creates the actual content (a button with a price) and renders it as a child. Then the final DOM structure in the browser looks like this:

```
...
<checkout-buy sku="fendt">
  <button type="button">buy for $54</button>
</checkout-buy>
...
```

It would be great if we could ship the button content already with the initial markup. Sadly, Web Components don't have a standard way to render server-side.³

TIP You can find the sample code for this task in the `16_universal` folder. It essentially combines the example code from `05_ssi` with `08_web_components`.

Since there is no standard way of doing it, we need to be creative. In this example, we will use the SSI technique you learned in chapter 4 for adding server-side composition to the Web Components approach. This way, we prepopulate the Web Components' internal markup. Figure 8.3 shows our folder structure. Team Decide adds an SSI directive as the child to the Buy button's Custom Element.

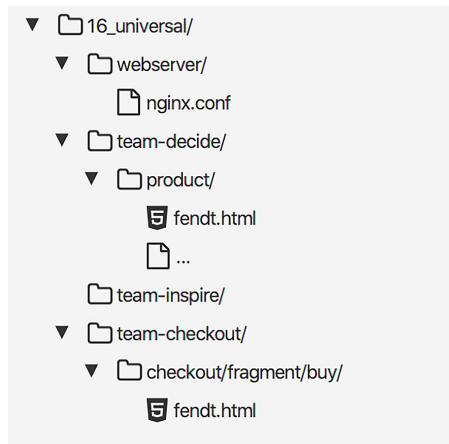


Figure 8.3 Nginx (`webserver/`) acts as the shared frontend proxy and handles the markup composition on the server side. Note that this is an excerpt of the complete folder structure.

Listing 8.2 `team-decide/product/fendt.html`

```

...
<checkout-buy sku="fendt">
  <!--#include virtual="/checkout/fragment/buy/fendt" -->
</checkout-buy>
...
  
```

Client-side Custom Element definition owned by Team Checkout. The associated code runs in the browser and renders/hydrates the micro frontend.

Nginx replaces this SSI directive with the content that's returned by the endpoint specified in `virtual`. Team Checkout owns this endpoint.

The preceding code of Team Decide's product page now combines client- and server-side composition. The Nginx web server replaces the SSI directive with the `<button>` markup, which Team Checkout generates when calling the `/checkout/fragment/buy/fendt` endpoint. Our example simulates this by serving a static HTML file.

Listing 8.3 `team-checkout/fragment/buy/fendt.html`

```
<button type="button">buy for $54</button>
```

³ There are custom solutions available in projects like Skate.js or Andrea Giammarchi's project Heresy. But since the W3C spec defines Shadow DOM as a pure client-side concept, we don't have a web standard to build upon for proper hydration.

In practice, you'd use a library with server-rendering capabilities to dynamically generate a response in a Node.js environment. For a React-based application, you'd call `ReactDOMServer.renderToString(<CheckoutBuy />)` and return its result. Here `<CheckoutBuy />` would be the React-based micro frontend application. The assembled product markup that reaches the browser looks like this:

```
...
<checkout-buy sku="fendt">
  <button type="button">buy for $54</button>
</checkout-buy>
...
```

Nginx replaced the SSI directive with the actual content.

The browser is now able to show the button instantly. The associated Custom Element code runs when the JavaScript finishes loading. It hydrates the micro frontend—making sure that the markup is correct and attaching events for further interaction.

Team Checkout's client-side code for the Buy button looks like this.

Listing 8.4 team-checkout/checkout/static/fragment.js

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

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

window.customElements.define("checkout-buy", CheckoutBuy);
...
```

Renders the markup client-side. This is a “dumb” implementation which replaces all existing markup even if it might already be correct. In a real application, you’d use something more clever and performant like DOM-diffing.

Adding event listeners to be able to react to user input

The code is identical to the examples we used in chapter 5. The component renders its internal markup inside itself and attaches all required event handlers. We again use a simplified implementation here. No client-server code reuse, no DOM diffing. But you get the picture.

When you use something like React, this is the place where you'd call `ReactDOM.hydrate(<CheckoutBuy />, this)`, where `<CheckoutBuy />` is the React application for the button and `this` is the reference to the Custom Element. The call instructs the framework to pick up the existing server-generated markup and hydrate it.

Figure 8.4 shows the complete process we went through, starting with the server-side markup generation at the bottom and ending with the initialization of the Buy button's Custom Element in the DOM.

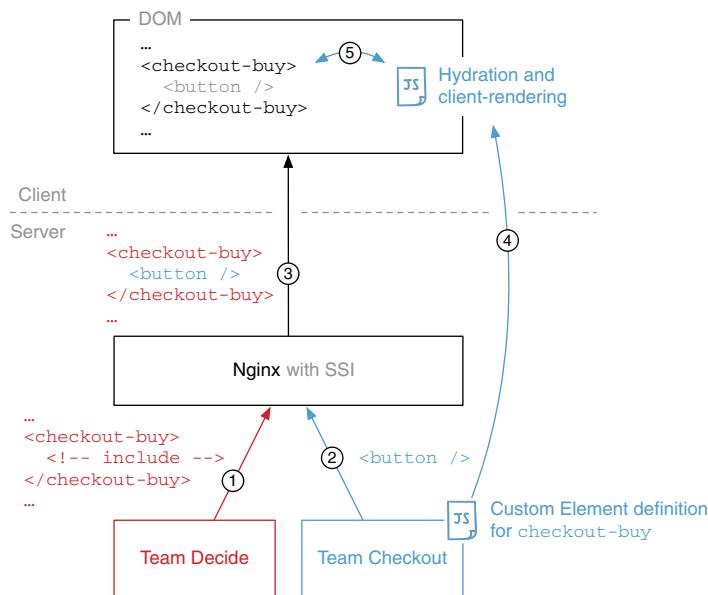
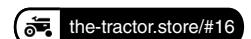


Figure 8.4 Prerendering the contents of a Web Component based micro frontend using SSI. The markup of Team Decide's product page contains a Custom Element for Team Checkout's Buy button. It has an SSI include directive as its content ①. Nginx replaces the include directive with the internal Buy-button markup generated by Team Checkout ②. The browser receives the assembled markup and displays it to the user ③. The browser loads Team Checkout's JavaScript containing the Custom Element definition for the Buy button ④. The Custom Element's initialization code (constructor, connectedCallback) runs. It hydrates the server-generated markup and can react to user input from this point on ⑤.

The integration works. Run the example with `npm run 16_universal` on your machine and open `http://localhost:3000/product/fendt` in your browser to see it working.



- Notice Team Checkout's mini-cart and Team Inspire's recommendation fragment. The integration for these fragments works the same way as for the Buy button.
- Have a look at the server-logs in the console. You can see how Nginx requests the individual SSI fragments needed for the page.
- See how the price on the Buy button updates client-side when you select the platinum edition.

- Clicking the button triggers the checkmark animation and updates the mini-cart.
- Disable JavaScript in your browser to simulate how the page looks when the client-side code fails or isn't loaded yet.

Progressive enhancement

You've noticed that the Buy button now appears even with JavaScript disabled. But clicking it does not perform any action. This is because we are attaching the actual add-to-cart mechanics via JavaScript. But it's straightforward to make it work without JavaScript by wrapping the button inside an HTML form element like this:

```
<form action="/checkout/add-to-cart" method="POST">
  <input type="hidden" name="sku" value="fendt">
  <button type="submit">buy for $54</button>
</form>
```

In the case of failed or pending JavaScript, the browser performs a standard POST to the specified endpoint provided by Team Checkout. After that, Team Checkout redirects the user back to the product page. On that page, the updated mini-cart presents the newly added item.

Building an application with progressive enhancement principles in mind requires a little more thinking and testing than relying on the fact that JavaScript always works. But in practice, it boils down to a handful of patterns you can reuse throughout your application. This way of architecting creates a more robust and failsafe product. It's good to work with the paradigms of the web and not reinvent your ones on top of it.

8.1.2 Contract between the teams

Let's take a quick look at the contract for including a fragment from another team. Here is the definition Team Checkout provides:

- **Buy button**

Custom Element: `<checkout-buy sku=[sku]></...>`

HTML endpoint: `/checkout/fragment/buy/ [sku]`

Since we are combining two integration techniques, the team offering the micro frontend needs to provide both: the Custom Element definition and the SSI endpoint, which delivers the server-side markup. The team using the micro frontend also needs to specify both. In our example Team Decide uses this code:

```
<checkout-buy sku="fendt">
  <!--#include virtual="/checkout/fragment/buy/fendt" -->
</checkout-buy>
```

These three lines include a lot of redundancy. To reduce friction, it's a good idea to establish a project-wide naming schema. This way, tag names and endpoints all look alike, and teams can use a generic template for including a fragment. Figure 8.5 shows how a schema might look.



Figure 8.5 This schema shows how you could generate the universal integration markup in a standardized way. When offering or integrating a fragment, teams need to know three properties: the name of the team that owns it, the name of the micro frontend itself, and the parameters it takes.

8.1.3 Other solutions

This is, of course, not the only way to build a universal integration. Instead of SSI and Web Components, you can also combine other techniques. Integrating server-side with ESI or Podium and adding your client-side initialization on top would also work.

Are you looking for a batteries-included solution? Then you could try the Ara Framework.⁴ Ara is a relatively young micro frontends framework, but it's built with universal rendering in mind. It brings its own SSI-like server-side assembly engine written in Go. Client-side hydration works through custom initialization events. Examples for running a universal React, Vue.js, Angular, or Svelte application exist.

8.2 When does universal composition make sense?

Does your application need to have a fast first-page load? Your user interface should be highly interactive, and your use case requires communication between the different micro frontends. Then there is no way around a universal composition technique like you've seen in this chapter.

8.2.1 Universal rendering with pure server-side composition

But the fact that one team wants to use universal rendering does not mean that you need a client-side composition technique. Let me give you an example.

Team Decide owns the product page and includes a header micro frontend (fragment), which Team Inspire owns. The two applications (product page and header) do not need to communicate with each other. Here a simple server-side composition is sufficient. Both teams can adopt universal rendering inside of their micro frontends if it helps their goal. But they don't have to. If the header has no interactive elements, a pure server rendering is sufficient. They can add client-side rendering later on if their use case changes. The other team does not have to know about it. From an architectural perspective, universal rendering inside a team is a team-internal implementation detail.

⁴ See <https://github.com/ara-framework>.

8.2.2 Increased complexity

Universal composition combines the benefits of server- and client rendering. But it also comes with a cost. Setting up, running, and debugging a universal application is more complicated than having a pure client- or server-side solution. Applying this concept on an architecture level with universal composition doesn't make it easier. Every developer needs to understand how integration on the server and hydration on the client works. Modern web frameworks make building universal applications easier. Adding a new feature is usually not more complicated. But the initial setup of the system and onboarding of new developers takes extra time.

8.2.3 Universal unified single-page app?

Is it possible to combine the application shell model from chapter 7 with universal rendering? Yes, in this chapter, we combined client- and server-side composition techniques to run multiple universal applications in one view. You could also combine client- and server-side routing mechanisms to create a *universal application shell*. However, this is not a trivial undertaking, and I haven't seen production projects that are doing this right now.

The single-spa project plans to add server-side rendering support. But at the time of writing this book, this feature hasn't been implemented yet.⁵

Let's take a look at our beloved comparison chart in figure 8.6 for the last time. As stated before, running a universal composition setup is not trivial and introduces

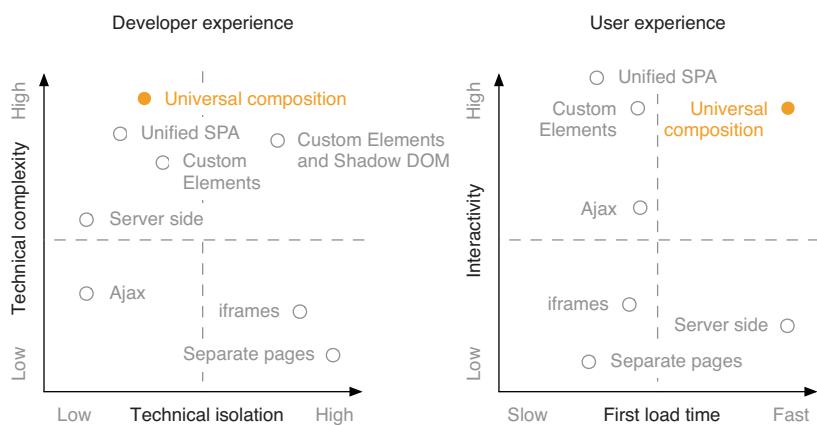


Figure 8.6 To run a micro frontends integration that supports universal rendering for all teams, we need to combine server- and client-side composition techniques. Both have to work together in harmony. This makes this approach quite complex. Regarding user experience, it's the gold standard, since it delivers a fast first-page load while also providing a high amount of interactivity. It also enables developers to build their features using progressive enhancement principals.

⁵ See <https://github.com/CanopyTax/single-spa/issues/103>.

extra complexity. Since it builds on the existing client- and server-side composition techniques, it also does not introduce extra technical isolation. But this approach shines when it comes to user experience. It's possible to achieve the page-load speeds of server-rendered solutions, and it also enables building highly interactive features that directly render in the browser.

To keep this chart readable, I've omitted the theoretical *universal unified SPA* option. It's by far the most complicated approach, but it would rank even higher on the interactivity scale since it eliminates all hard page transitions.

Summary

- Universal rendering combines the benefits of server and client rendering: fast first-page load and quick response to user input. To leverage this potential in a micro frontends project, you need to have a server- and client-side composition solution.
- You can use SSI together with Web Components as a composition pattern.
- Each team must be able to render its micro frontend via an HTTP endpoint on the server and also make it available via JavaScript in the browser. Most modern JavaScript frameworks support this.
- On the first page load, a service like Nginx assembles the markup for all micro frontends and sends it to the browser. In the browser, all micro frontends initialize themselves via JavaScript. From that point on, they can react to user input entirely client-side.
- Currently, there's no web standard to server-render a Web Component. But there are custom solutions to define ShadowDOM declaratively. In our example, we use the regular DOM to prepopulate the Web Components content on the server.
- It's possible to implement a universal application shell to enable client- and server-side routing. However, this approach comes with a lot of complexity.

Which architecture fits my project?

This chapter covers

- Contrasting different micro frontend architectures you can build with the learned integration techniques
- Comparing the benefits and challenges of different high-level architectures
- Figuring out the best architecture and composition technique for your project's needs

In the last seven chapters, you've learned different techniques for integrating user interfaces owned by different teams. We started with simple ones like links, iframes, and Ajax, but also more sophisticated ones like server-side integration, Web Components, and the app shell model. These chapters all ended with a simplified comparison chart indicating how the newly learned technique compares to the previous ones. In this chapter, we'll put all the puzzle pieces together and also make a more in-depth comparison. First, we revisit the terminology and highlight the key advantages of the different techniques and architectures. After that, you'll learn about the Documents-to-Applications Continuum, which can help you decide

if you should go for a server- or client-side integration. This distinction is crucial because it determines which architectures and integration patterns are suitable for your use case. We'll end this chapter with an architecture decision guide. You'll learn how you can make a sound choice based on a handful of questions. These questions will lead you through the different options.

9.1 Revisiting the terminology

When you are setting up a micro frontends project with different teams, everyone must use the same vocabulary. That's why we're taking a step back and sorting the terms you've learned in the previous chapters. We'll start with the basic building blocks: the *integration techniques*. Then we'll look at different *high-level architectures* that you can build with them.

We can group them into two categories: *routing and page transition* and *composition*. Figure 9.1 shows all the integration techniques we've covered in this book.

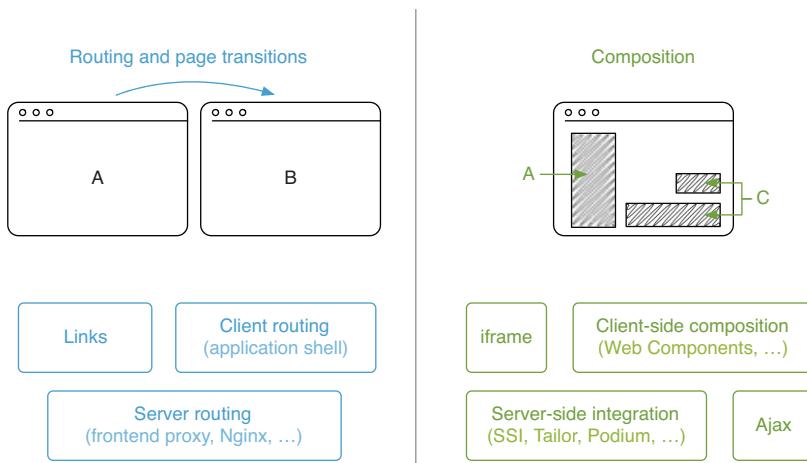


Figure 9.1 The integration techniques required for a micro frontend architecture. On the left, we see two techniques for handling cross-team page transitions. The right side shows a list of methods of composing different user interfaces onto one page.

Let's briefly revisit the techniques. We'll start with routing and page transitions.

9.1.1 Routing and page transitions

When we talk about page transitions as an integration technique, we technically always mean *inter-team page transitions*. How does a user get from a page owned by Team A to a page owned by Team B? From an architectural standpoint, it's not essential to know how a team handles transitions between its own pages. This is an implementation detail.

LINKS

The plain old hyperlink is the most basic form for doing a micro frontends integration. Each team is responsible for a set of pages. Handing over the user to another part of the application is as easy as placing a link to the other team's work. In its cleanest form, no extra coordination is needed. Teams could even host their part of the applications under different domains. We covered the link in chapter 2.

APPLICATION SHELL

Clicking on classical hyperlinks forces the browser to fetch the target markup from a server and then replace the current page with the new one. Having to reload is fine for a lot of use cases. But the evolution of the browser's History API and the rise of single-page app frameworks enabled developers to build entirely client-side page transitions. Its main benefit is the opportunity to render the layout for the target page instantly. That way, the user gets a quick response, even if the content data requested from the server is still pending. Implementing client-side page transitions across team boundaries requires a central piece of JavaScript in the browser. It's typically called the *app shell*. The central app shell acts as a parent application to the single-page applications built by the different teams. It determines which team's application should be active based on the browser's URL. When the URL changes, it passes the responsibility for the page from Team A to Team B. You can find more details on this in chapter 7.

9.1.2 Composition techniques

In practice, you often want to show user interface parts from different teams on one page. A typical example of this is a header or navigation micro frontend. One team builds and owns it. All the other teams integrate it onto their pages. It could also be functionality like the Buy button or mini basket on our product page.

In this book, we've often called an includable micro frontend a *fragment*. To make the integration happen, we need a shared format. The owner of the fragment must provide it in a standardized format. The fragment consumer uses this format to integrate the desired micro frontend on their page.

We can broadly group the composition techniques into two buckets: *server-side integration* and *client-side integration*. We've also included the *iframe* and *Ajax* technique since they are a bit of a hybrid between server and client.

SERVER-SIDE INTEGRATION

Implementing a server-side integration technique makes sense when teams generate their markup server-side. The markup for all fragments of a page gets assembled before it reaches the customer's browser. A central piece of infrastructure like a web server will perform the markup assembly. In chapter 4 we used the SSI technique in Nginx to perform this task. An alternative approach is that the team owning the page fetches the required fragments directly from the other teams. The server-side integration libraries Tailor and Podium work like this.

CLIENT-SIDE INTEGRATION

If teams generate their markup in the browser, you need a client-side integration technique. A solid approach is leveraging the Custom Elements API from the Web Components spec. The API defines (de)initialization hooks. The team that owns the fragment implements them. This way, the integration happens directly through the browser's DOM API. No special libraries or custom JavaScript APIs are required.

An essential part of client-side integration is communication. How can fragment A inform fragment B about an event that might be interesting? Micro frontends can communicate via Custom Events or an event bus/broadcasting solution. We covered this in chapter 6.

IFRAME

The iframe is the weird but somewhat powerful stepchild of web development. It fell out of favor years ago for various reasons. Using iframes in responsive design doesn't work without JavaScript, and having a lot of iframes on a site is resource-intensive. But its secret superpower is that it provides a high level of technical isolation. In a micro frontends context, this is a desirable feature. This way, faults in micro frontend A can't negatively affect micro frontend B. Communication across iframes is also possible through the `window.postMessage` API. We briefly talked about the iframe in chapter 2.

AJAX

Fetching a snippet of markup from a server endpoint via JavaScript is the technique that enabled the Web 2.0 revolution back in the day. You can also use Ajax as an integration technique for micro frontends. Client-side JavaScript triggers the actual Ajax call to fetch server-side generated HTML. Ajax is a bit of a hybrid approach that does not fit into one of our client- or server-side integration buckets. It is often used in tandem with a server-side integration technique—incrementally updating the markup of an embedded micro frontend. It does not come with a canonical way to handle (de)initialization and communication. Using Web Components together with Ajax for internal updating is also a good fit.

These are the basic integration techniques you've learned so far. Let's zoom out a bit and look at different architectural styles.

9.1.3 *High-level architectures*

One benefit of micro frontends is that teams are free to use the technology that fits their slice of the application best. However, before you start setting up a micro frontends project, all teams need to be on the same page when it comes to the high-level architecture. Are we building static pages that integrate solely via links, or is the goal to create a highly dynamic and tighter-integrated single-page app? You should consciously make this decision together with all teams. Figure 9.2 shows six different architectures.

We'll go through them from top to bottom.

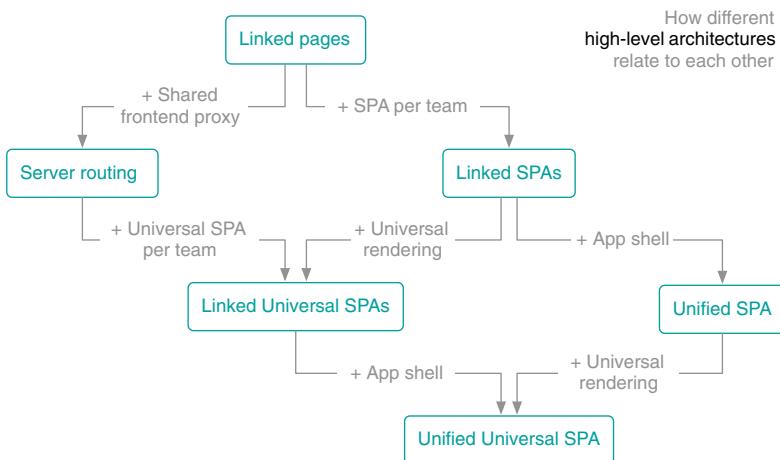


Figure 9.2 Different architectural styles to build a micro frontends project. This chart starts with the simplest form, the linked pages approach, and shows how you can extend this with extra features like single-page applications, universal rendering, or a shared app shell.

LINKED PAGES

This is the most simple architecture. Every team serves its pages as complete server-rendered HTML documents. Clicking a link reloads the complete page and shows the desired content. This hard navigation happens if you are moving between pages from the same team or if you are navigating across team boundaries. The simplicity of this approach is its main benefit: no central infrastructure or shared code is required, debugging is straightforward, and new developers instantly understand what's going on. But from a user experience point of view, there's room for improvement.

SERVER ROUTING

This is identical to the Linked Pages approach, but with the difference that all requests pass through a shared web server or reverse proxy. This server sits in front of the team's applications. It has a set of routing rules to identify which team should handle an incoming request. The routing is often done via URL prefixes associated with a specific team. We talked about this in chapter 3.

LINKED SPAs

To improve the user experience and react to input faster, a team can decide to switch from delivering static server-generated pages to implementing a client-rendered single-page app for the pages they own. This way, all link clicks for pages from this team result in a fast soft navigation. The transitions between team boundaries are still hard navigations. Technically the adoption of a single-page app architecture inside one team can be seen as an implementation detail. As long as linking to a specific page from the outside still works, teams can decide to change their internal architecture.

But the distinction between linked pages and linked single-page apps is essential when we talk about more advanced architectures and suitable integration techniques later.

LINKED UNIVERSAL SPAs

Teams can also decide to adopt universal rendering. The markup for the first request gets rendered on the server. It enables a pretty fast first-page load experience. From there on, the application behaves like a single-page app—incrementally updating the user interface as needed. From a team's point of view, this is a more complicated setup, which requires some additional development skills. But from an architectural view, this approach is identical to the other “linked” architectures. The contract between the teams is still a set of shared URL patterns. A navigation across team boundaries results in a reload of the page. But when implemented well, these reloads should be more seamless compared to a Linked SPA architecture, where the browser needs to execute a bunch of JavaScript before the user can see the content. Chapter 8 discusses universal rendering, its benefits, and its challenges.

UNIFIED SPA

The Unified SPA describes a single-page application composed of other single-page applications. In chapter 7 we introduced this concept. It requires all teams to build their software as a single-page app. These single-page apps are then unified by a parent application, which is often called the *app shell*. The shell typically does not render any user interface. Its job is to listen to changes in the browser’s address bar and pass control from one single-page app to another if necessary. With the Unified SPA architecture, all page transitions are soft navigations. This leads to a snappier and more app-like user interface. However, the app shell is a central piece of code. It introduces a non-trivial amount of coupling and complexity.

UNIFIED UNIVERSAL SPA

When we take the Unified SPA model and introduce universal rendering, we arrive at something we call *Unified Universal SPA*. With this model, each team builds a single-page app with universal rendering capabilities. To make this work, the parent application (app shell) also needs to be universal. It needs to be able to run on the server and the client. This is a pretty challenging architecture. It promises to combine the best of all worlds but comes with the most complexity.

9.2 Comparing complexity

The architecture and the level of integration you choose have a considerable effect on your complexity. This complexity manifests itself in different aspects:

- Initial infrastructure work that’s required to get started.
- Number of moving parts (services, artifacts) that need maintenance.
- Amount of coupling: Which changes require more than one team to become active?
- Developer skill level: What concepts do new developers need to understand?
- Debugging: How easy is it to attribute a bug to a specific team?

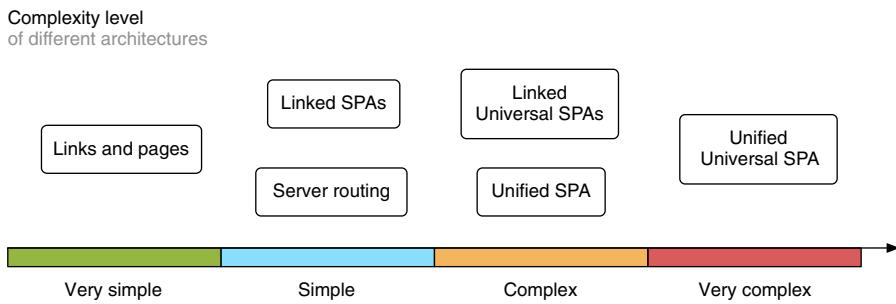


Figure 9.3 Micro frontend architectures sorted by complexity. The links and pages approach is the simplest one to build and run. The complexity rises as you move to more sophisticated architectures. The Unified Universal SPA approach requires a lot of development skills to get it right. You also need shared infrastructure and code to make it happen.

Figure 9.3 sorts these architectures into four complexity groups, starting from very simple and ending with very complex. This is, of course, only general guidance. The real cost associated with an architecture depends on your team's experience and the use case. *As a rule of thumb, you should always opt for the most simple architecture you can responsibly get away with.* Sure, it's nice to have a Unified SPA with no hard page transitions, but does the extra work required to achieve and maintain this justify the potential benefits?

9.2.1 Heterogeneous architectures

In the descriptions so far, we've always assumed that all teams use the same architecture. But you can also mix and match to create a heterogeneous architecture. For a team that builds fast-loading landing pages, the links and pages approach might be sufficient. But for a seamless browsing experience, you want to create a Unified SPA that integrates the team which owns the product list and the team managing the product pages. These architectures can work side-by-side: some teams are doing links and pages, whereas some other teams share an app shell to deliver a Unified SPA. This way, you only increase the complexity in the areas where it's needed.

But having a heterogeneous architecture also has drawbacks:

- There is no go-to architecture for a new team. Teams need to analyze and discuss their use cases beforehand. (This is not necessarily a drawback.)
- Integrating fragments from different teams might get harder. Teams need to deliver their includable micro frontend in a format that works for the page that includes it.

9.3 Are you building a site or an app?

As you've seen throughout the book so far, it makes a significant difference whether you render your markup on the client or server. It's a general question that everyone who's setting up a new web project has to answer. But in a micro frontends context, this decision is essential. It defines which integration techniques are suitable.

In this section, you'll learn about the Documents-to-Applications Continuum. I've found this concept helpful in architecture discussions. It creates an excellent mental model that helps you pick the right tools and techniques for the job. It provides a counterweight to the "Let's use the hot new JavaScript framework!" reflex many developers (me included) have when they're confronted with a greenfield project. After explaining the concept, we'll look at how the high-level architectures fit into this continuum.

9.3.1 The Documents-to-Applications Continuum

What purpose does the project we are building serve? Do people come to our site to *consume content*, or do they want to *use a specific functionality* we provide? For better visualization, it helps to look at extreme examples:

- *Content-centric*—Imagine a simple blog. A user can browse the list of posts and read the complete content on a dedicated article page.
- *Behavior-centric*—Imagine an online drawing application. People can go to the site and draw beautiful sketches with their fingers and export them as an image.

The first one is a prototypical web site where the content is essential. The second one is a pure application. It does not bring any content. It's all about the functionality it provides to the user.

In a non-trivial project, it's typically not that black and white. This is where the Documents-to-Applications Continuum¹ comes in. The idea is that both examples are at different ends on a spectrum, as illustrated in figure 9.4. Positioning your micro frontends project on this scale can help you set the right priorities and select an appropriate high-level architecture.

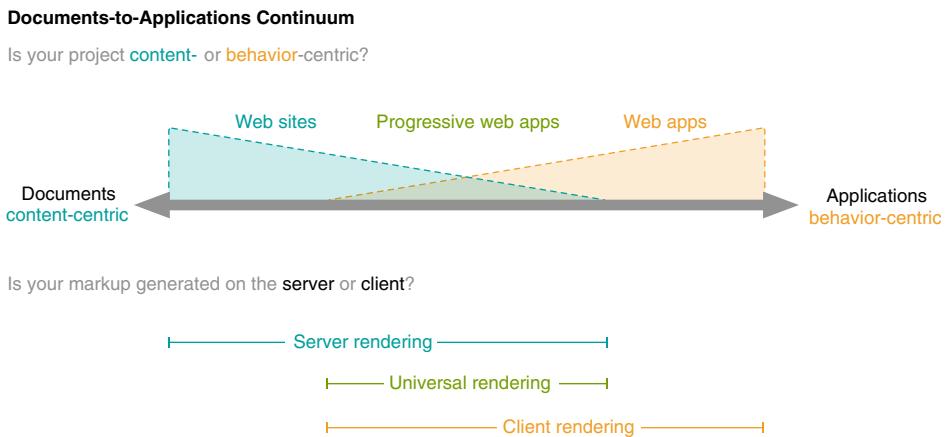


Figure 9.4 The Documents-to-Applications Continuum provides a mental model to help you think about whether your project is more of a web site or a web application. It's a gradual scale and not a black and white decision.

¹ See Aral Balkan, "Sites vs. Apps defined: the Documents-to-Applications Continuum," *Aral Balkan*, <http://mng.bz/90ro>.

Let's look at two examples. Where would amazon.com fit on this scale? They provide a lot of functionality. You can search, sort, and filter through product lists, rate products, manage your returns, or have a live chat with their customer service. But at its core, it's a content-centric site. A good question to ask is, "Would the site still be useful if we stripped away all behavior?" For amazon.com, we can answer this with a definite yes. No doubt, the extra functionality is also important, but without products, the features would be pretty useless. We would put that site somewhere on the left part of the continuum. Starting with server-side composition, with the option to upgrade it to a universal composition, is a safe bet when picking a micro frontends architecture.

Now to our second example. The site CodePen.io lets web developers and designers put together HTML, CSS, and JS to get a live preview in the browser. Developers use the online code editor to sketch out ideas or isolate bugs. CodePen also has an active community of people who showcase their work and share code with others. You can go to the site and discover new exciting techniques by browsing the public catalog. How does CodePen fit on our continuum? It's a harder question to answer because it's strong on both aspects: the online editor (behavior-centric) and the public catalog (document-centric). If we stripped away all behavior, the online editor would vanish. If we removed all content, the catalog would disappear, but the editor would still be there. That's why we'd probably put CodePen in the middle of the spectrum. If we rebuilt CodePen in a micro frontends architecture, we would establish two teams. Team Editor would pick a client-side approach. Team Catalog would likely go the server-side route. This is a good starting point. To decide which micro frontend architecture fits best, we have to go a step further and analyze the use cases. Does one team need to include content from the other? How does the user move through the site?

9.3.2 Server, client, or both

Classifying your product onto the continuum is a good starting point to identify if your templating should live on the server or in the browser. If your product has a strong content focus, server-side rendering should be your first choice. Using progressive enhancement to add functionality should feel natural.

If you're building an application where it's all about interaction and not about content, a purely client-rendered solution will be the best fit. Here the concept of progressive enhancement doesn't help you at all, because there is no enhanceable content to begin with.

For a project that resides in the middle of the spectrum, you need to make a choice. Server- and client-side templating are valid options. But in this area, both have their advantages and disadvantages. If you aren't afraid of the extra complexity, you can also pick both and go with the universal rendering option.

Let's revisit our high-level architectures. Figure 9.5 highlights which of them use server-side, client-side, and universal rendering.

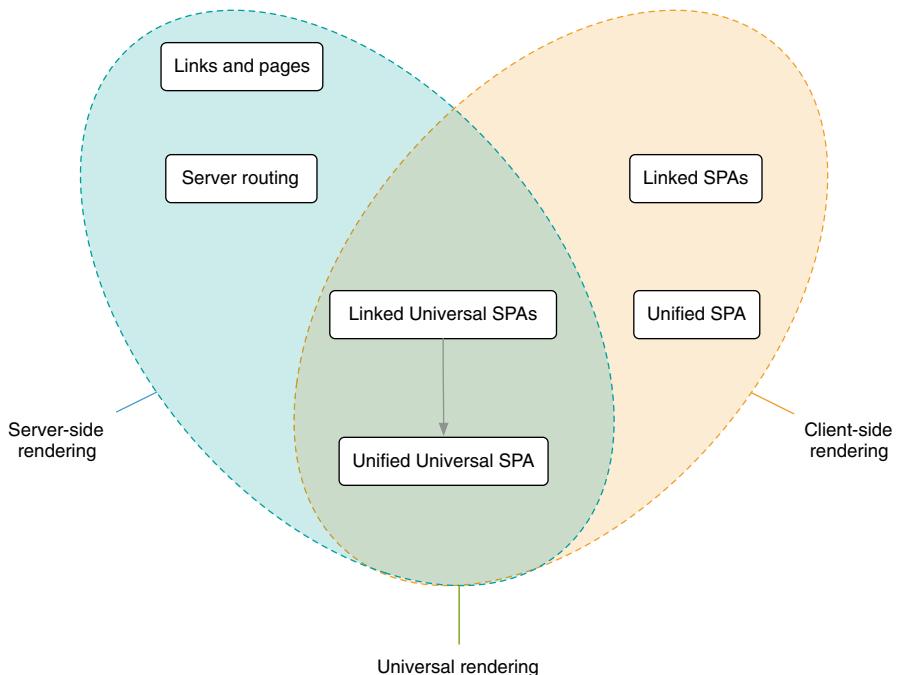


Figure 9.5 Illustrating which architectures feature server-side, client-side, or universal rendering

Make sure the architecture you choose aligns with the nature of your project and the business. Templating and complexity considerations are two significant factors in making a decision. Next up, we'll take another angle on this decision using a decision tree.

9.4 **Picking the right architecture and integration technique**

Now we've sharpened our vocabulary and have a mental model to pinpoint what kind of product we are building. Let's look at a concrete way to determine which architecture and integration your project needs. Figure 9.6 shows a decision tree that helps with this question. It's inspired by Manfred Steyer's work² on creating Angular-based frontend microservices.

Take some time to understand what's going on in this diagram. Follow the lines from top to bottom by answering the questions until you reach your high-level architecture. From there on, you can follow the dotted line to get to the compatible composition technique. If your use case does not require you to have different micro frontends to be active at the same time (fragments or nested micro frontends), you can skip this step.

² See Manfred Steyer, "A Software Architect's Approach Towards Using Angular (And SPAs In General) For Microservices Aka Microfrontends," *Angular Architects*, <https://www.angulararchitects.io/aktuelles/a-software-architects-approach-towards/>.

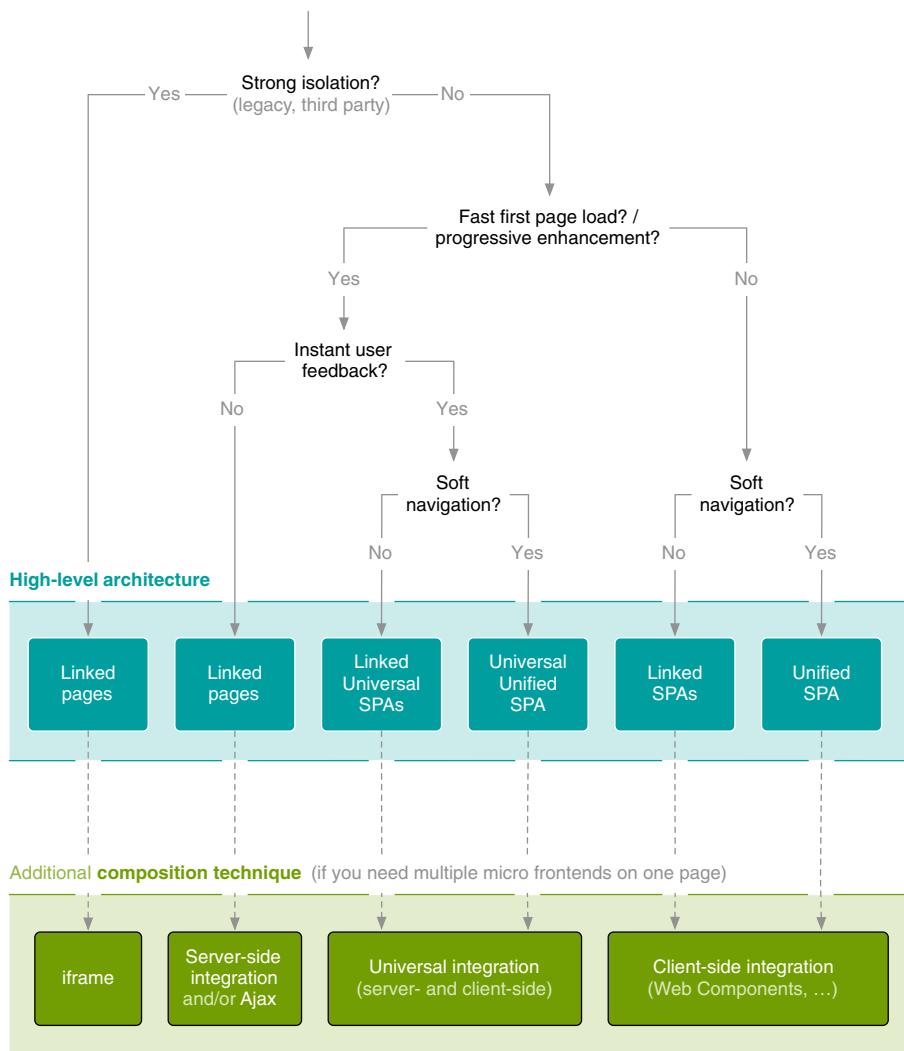


Figure 9.6 The decision tree helps to pick a micro frontends architecture based on your project's requirements. It also shows which kind of composition technique is appropriate for your use case.

Let's look at the questions in this decision tree.

9.4.1 Strong isolation (legacy, third party)

Do you want strong technical isolation between the code of the teams? Of course you do. Why wouldn't you? Isolation and encapsulation generally lead to less unforeseen effects and reduces bugs. But sadly, opting for strong isolation eliminates a lot of other possibilities. So the right question to ask is, Do you *need* strong isolation? This is typically true if you integrate a legacy system that doesn't respect namespacing rules and

requires global state to work correctly. Another reason is security. If you are integrating with an untrusted third party solution or one part of your application has high-security requirements (for example, it handles credit card data), it can be necessary to better shield the micro frontends against each other.

9.4.2 **Fast first-page load/progressive enhancement**

This is a double question. If you need either of these properties, you should follow the *yes* arrow.

Having a fast first-page view is always pleasant, but the importance of this property heavily depends on your business. If you want your site to rank high in search results, first-page load performance is nothing you can ignore. Search engines like Google increasingly favor fast-loading sites in their ranking.³ Even if search ranking is not your primary goal, there are a lot of case studies⁴ that show how better web performance increases business metrics.

We talked about the benefits of progressive enhancement in chapter 3. If you'd locate your project on the middle or left side of the Documents-to-Applications Continuum, I'd highly recommend adopting progressive enhancement practices. You should encourage all developer teams to learn about this approach. For developers that started their web career with frameworks like React or Angular, the concepts might sound strange at first sight. However, architecting features with progressive enhancement in mind and embracing the primitives of the web will lead to more maintainable, easier to understand, and more stable software. If you're on the far right in the continuum and building a pure web application, there is typically no content to enhance. Then progressive enhancement won't help you at all.

9.4.3 **Instant user feedback**

In the previous question, we talked about the first-page load performance. *But how does your site react to further interactions from the user?* The classical “click a link” and “fetch generated markup from the server” works for a lot of cases, primarily when you use Ajax techniques to avoid a full page reload. In this model, the complete templating resides on the server. *This means that at least one server roundtrip is necessary to update the UI in response to a user input.*

If you need to be faster than this, you must adopt client-side rendering. Fetching data will be a bit faster since JSON data is more compact than rendered HTML, but the network latency itself stays the same. However, the most significant advantage of client-side rendering is that it enables us to provide instant feedback. Even if the data the user wants to see is still in transit, you can update the view and show placeholders and skeleton screens.⁵

³ See <http://mng.bz/WP9w>.

⁴ See <https://wpostats.com/>.

⁵ See Luke Wroblewski, “Mobile Design Details: Avoid The Spinner,” *LukeW*, <https://www.lukew.com/ff/entry.asp?1797>.

It also enables you to adopt *optimistic UI patterns*.⁶ With optimistic UI, you try to increase the perceived performance by instantly rendering the result that's most likely. Let's look at a shopping cart example. When a user wants to delete an item, they click on the Delete button. The browser calls the associated API on the server, and when it comes back, the item is deleted from the visual shopping cart list. With optimistic UI, you assume that the delete API call works in most of the cases. That's why you remove the line item directly and don't wait for the API call to return. If this assumption turns out not to be true (item remove failed), you restore the item in the user interface and show an appropriate error message. This technique is powerful, but since you are effectively lying to your user, you should use it with care. These techniques help you render a response to user input instantly, which improves user experience and makes your site feel more app-like.

9.4.4 Soft navigation

In the “instant user feedback” question, we talked about improving the user experience inside a micro frontend. Now let's look at what happens when the user transitions across team boundaries. This question differentiates the *linked* architectures from the *unified* architectures. We talked about this in chapter 7. How important is it that inter-team page transitions are client-side rendered?

Answering this question depends heavily on the team boundaries you establish, the number of teams, and the usage pattern of your application. If you create your team boundaries along with the user's tasks and needs, the user doesn't cross team boundaries that often.

Say you are building a website for a bank. It has two distinct areas developed by two teams: users can check their account balance (Team A), and they can also calculate and request a housing loan (Team B). For a good user experience, it might be essential to provide a high amount of interactivity inside these areas. This is something Team A and Team B can decide independently. But since users seldom switch between balance checking and loan requesting in one session, it might be fine to have a hard navigation between these areas.

Let's pick another example. We are building a call-center application. The agents use the application to manipulate orders (Team A) and make personalized recommendations (Team B). Since the agent switches between these two micro frontends frequently, it might be a good idea to implement soft navigation. It makes using the application faster and positively impacts the agent's workflow.

9.4.5 Multiple micro frontends on one page

If you've answered all questions on your way down the tree and arrived at your high-level architecture, there is one last bonus question: “Do you need composition?” Answering “yes” brings you straight down to the associated composition technique. If

⁶ See Denys Mishunov, “True Lies Of Optimistic User Interfaces,” *Smashing Magazine*, <http://mng.bz/8pdB>.

you are building a pure SPA, you need to integrate client-side. If you opt for server-generated pages, you should use a server-side integration.

Having a composition technique is optional. When we look at our banking example from the previous section, we might not need a composition technique at all. The account area and the housing loan area could be two distinct sections of the site that link to each other.

The most common example of a composition is a header and navigation fragment. Usually, one team owns it, and the others include it on their page.

Summary

- Establishing a shared vocabulary across all teams avoids misunderstandings. Differentiating between transition techniques, composition techniques, and your high-level architecture helps everyone to get a clear picture of what you are building toward.
- The Documents-to-Applications Continuum is a good mental model for identifying whether your project is more content- or behavior-centric. This distinction helps you to make good technology choices.
- There are no right or wrong solutions. Whether a solution fits or not depends on the nature of your project, its usage patterns, the amount of coupling and complexity you are willing to accept, and your team's size and experience level.
- Not all teams need to adopt the same architecture. Some parts of your application might be document-centric; others more behavior-centric. With micro frontends, it's possible to mix and match. But when you need composition, you must find an integration technique that works for all teams.
- Try to pick the simplest architecture that is reasonable for your business.

Part 3

How to be fast, consistent, and effective

Y

You've learned the integration techniques to build a micro frontends application. However, to make your project successful, there are topics beyond integration that you need to address. We touched on the aspects of performance, visual consistency, and team responsibility at various points in the preceding chapters. In this next part of the book, we will cast a brighter light on these complementary aspects.

In chapter 10, we start with a technical topic: asset loading. Loading the right script and style code for the different micro frontends can be surprisingly challenging—especially if you want to adhere to performance best practices without sacrificing team autonomy. In chapter 11, we go deeper into the topic of performance. You'll see common pitfalls and learn strategies to build and maintain a fast-loading site even if the UI comes from different teams. Chapter 12 deals with one of the biggest criticisms of micro frontends: How can we ensure a consistent look and feel for the end user? Design systems are essential in addressing this issue. However, establishing a shared design system so that it doesn't interfere with the autonomy goals of micro frontends is not without challenges. In chapter 13, we talk about the organizational implications of introducing micro frontends. You'll learn how to find sound team boundaries, organize inter-team knowledge transfer, and deal with shared infrastructure. The last chapter covers migrating to micro frontends, tips for local development, and some patterns for effective testing.

10

Asset loading

This chapter covers

- Solving common asset loading challenges in a micro frontends context
- Comparing techniques to deal with cacheability and synchronization when loading assets from different teams
- Deciding what bundling strategy is appropriate: many smaller bundles or fewer large ones
- Understanding how on-demand loading can be effectively used with micro frontends

In the preceding chapters, we covered a lot of different integration techniques. But we always focused on the content—integrating markup on the server and in the client. A topic we only discussed in passing is this: *How to load the assets associated with a micro frontends?* In this chapter, we’ll dive deeper into this significant side topic. There are at least a handful of aspects that you must consider. How can we ensure that teams can deploy a micro frontend and the needed assets on their own? How do you implement cache busting to improve cacheability without introducing tight

coupling? How do you ensure that the loaded CSS and JS always fits the server-generated markup? How coarse or fine-grained should your bundles be? Do you want one big bundle for your application, one per team, or even smaller ones? How can on-demand loading techniques help in reducing the upfront asset data the browser needs to process?

10.1 Asset referencing strategies

We'll start with some techniques to integrate the assets into a page. For simplicity, we will stick to traditional `<link>` and `<script>` tags in the following scenarios. Module loaders like RequireJS¹ (AMD) or CommonJS² are popular and provide programmatic loading functionality. But nowadays, ES Modules are supported in all significant browsers.³ They are a web standard that solves most of our JavaScript loading needs without needing an extra library or a custom module format.

Later in this chapter, we'll talk about bundle granularity. For now, let's assume that every team that provides an includable micro frontend (a fragment) generates one JavaScript and CSS file. The including team must add the references for both files to its page.

10.1.1 Direct referencing

The concept is pretty straightforward. If you want to integrate a micro frontend from another team, you have to add their references to make it work. You can think of the associated assets like adding an `import` at the top of your source code file in languages like Java, C#, or JavaScript.

If you are going the app shell route, it's different. There you have one single HTML document. It's the responsibility of the shared app shell to load the code for all micro frontends. The simplest way is to include all assets from all teams up front. A smarter way would be to load assets just in time when the user needs them. The meta-framework single-spa implements on-demand loading. You can flip back to chapter 7 to see the dynamic `import()`-based JavaScript registration code. We'll talk more about on-demand loading later in this chapter.

Let's go back to The Tractor Store and revisit how we dealt with asset loading in the last chapters. There Team Decide referenced the assets directly. The other teams published the URLs of the associated assets as part of their documentation.

Here is an example from chapter 5. Team Checkout specifies the Custom Element details for their Buy button and the files that contain the associated initialization code and styling:

- *Custom Element*—`<checkout-buy sku="{sku}"></checkout-buy>`
- *Required assets*—`/checkout/fragment.js`, `/checkout/fragment.css`

¹ See <https://requirejs.org>.

² See <http://www.commonjs.org>.

³ See <https://caniuse.com/#feat=es6-module-dynamic-import>.

To ensure fast rendering, it's best practice⁴ to include stylesheets in the `<head>` and script tags asynchronously at the end of the `<body>`. Team Decide directly includes these references in their product page's markup.

Listing 10.1 08_web_components/team-decide/product/porsche.html

```
<html>
  <head>
    <link href="/decide/page.css" rel="stylesheet" />
    <link href="/checkout/fragment.css" rel="stylesheet" /> ← Styles for Team
      Checkout's
      fragments
  </head>
  <body>
    <h1>The Tractor Store</h1>
    <checkout-buy sku="porsche" ></checkout-buy> ← Team Decide
      includes the Buy
      button micro
      frontend of Team
      Checkout. It relies
      on Team Checkout's
      assets to be present.

    <script src="/decide/page.js" async></script>
    <script src="/checkout/fragment.js" async></script> ← Scripts for Team
      Checkout's
      fragments
  </body>
</html>
```

10.1.2 Challenge: Cache-busting and independent deployments

One day CEO Ferdinand walks into Team Decide's office space, laptop under his arm. He grabs a chair, opens his laptop, and points at his screen. “I've read an article about the importance of web performance in e-commerce. I ran a tool called Lighthouse⁵ on our product pages. It measures performance and checks if our site uses best practices. We score 94 points. This score is way better than our competitors! However, Lighthouse shows one piece of advice. We seem to use an inefficient cache policy on static assets.”⁶

The current best practice for performant asset loading is to ship static assets (JavaScript, CSS) in separate files with a *one-year cache header*. This way, you ensure that the browser does not redownload the same file twice. Adding this cache header is not complicated. In most applications, web servers, or CDNs, it's a simple configuration entry. However, you need a cache invalidation strategy. If you've deployed a new CSS file, you want all users to stop using their cached version and download the updated one. An effective invalidation strategy is adding a fingerprint to the filename of the asset. The fingerprint is a checksum based on the contents of the file. A filename could look like this—`fragment.49.css`. The fingerprint only changes when the file is modified.

We call this *cache busting*. Most frontend build tools like Webpack, Parcel, or Rollup support it. They generate fingerprinted filenames at build time and provide a way to

⁴ See Ilya Grigorik, “Analyzing Critical Rendering Path Performance,” <http://mng.bz/rr7e>.

⁵ See <https://developers.google.com/web/tools/lighthouse>.

⁶ See <https://developers.google.com/web/tools/lighthouse/audits/cache-policy>.

use these filenames in your HTML markup. You might already see the issue. Cache busting does not play nice with our distributed micro frontend setup.

In our earlier example, Team Decide needed to know the path to Team Checkout's JavaScript and CSS files. Yes, Team Checkout could update their documentation:

Required assets—/checkout/fragment.a62c71.js, /checkout/fragment.a98749.css

But with the current process, Team Decide would have to manually update these references in their product page's markup every time Team Checkout deploys a new version. In this scenario, a team is not able to deploy without coordinating with another team. This coordination is the kind of coupling we want to avoid. Let's explore some better alternatives.

10.1.3 Referencing via redirect (client)

You can circumvent this problem by using HTTP redirects. The idea is the following:

- 1 Team Decide references Team Checkout's assets as before without fingerprint. The URLs are stable and do not change (e.g., /checkout/fragment.css).
- 2 Team Checkout responds with an HTTP redirect to the fingerprinted file (/checkout/fragment.css → /checkout/static/fragment.a98749.css).

This way, Team Checkout can configure the directly referenced file (/checkout/fragment.css) with a short cache header or set it to no-cache. They can give the fingerprinted file which contains the actual content a long cache lifetime (for example, one year). The benefit is that the registration code can stay the same, and the user only downloads the big asset file when it changes.

In our example, `17_asset_client_redirect`, we've added a redirect configuration and caching header to the team's web servers. You can look at the `serve.json` file in each team's folder. The `mfserve` library picks up this file. In a real application your build tool or bundler would create the fingerprints and redirect rules for you.

Listing 10.2 `team-checkout/serve.json`

```
{
  "redirects": [
    {
      "source": "/checkout/fragment.css",
      "destination": "/checkout/static/fragment.a98749.css"
    },
    ...
  ],
  "headers": [
    {
      "source": "/checkout/static/**",
      "headers": [
        { "key": "Cache-Control", "value": "max-age=31536000000" }
      ]
    }
  ]
}
```

Configuring a redirect from the public asset path to the latest fingerprinted version

Setting a one year cache header to all fingerprinted assets. By default all other resources are served with Cache-Control: no-cache.

Start the application by running `npm run 17_asset_client_redirect`. The network requests for Team Checkout's fragment styles look like this:

```
# Request  
GET /checkout/fragment.css
```

Browser requests Team Checkout's fragment styles

```
# Response (redirect)
HTTP/1.1 301 Moved Permanently
Cache-Control: no-cache
Location: /checkout/static/fragment.a98749.css
```

Team Checkout responds with a 301 redirect to the fingerprinted resource. The redirect is not cacheable.

```
# Request  
GET /checkout/static/fragment.a98749.css
```

Browser requests the fingerprinted resource

```
# Response (actual content)
HTTP/1.1 200 OK
Content-Type: text/css; charset=utf-8
Content-Length: 437
Cache-Control: max-age=31536000000
```

Team Checkout serves the asset file with a one-year cache header

Figure 10.1 shows the example code with the network panel of the browser. You can see that each registration file redirects to the fingerprinted and cacheable version.

The main benefit compared to the direct referencing approach is the decoupling. A team that provides a micro frontend can ship versioned assets with long cache times. They can update their code without having to notify the team that includes it. It's also simple to build, and users only have to redownload an asset if it has changed.

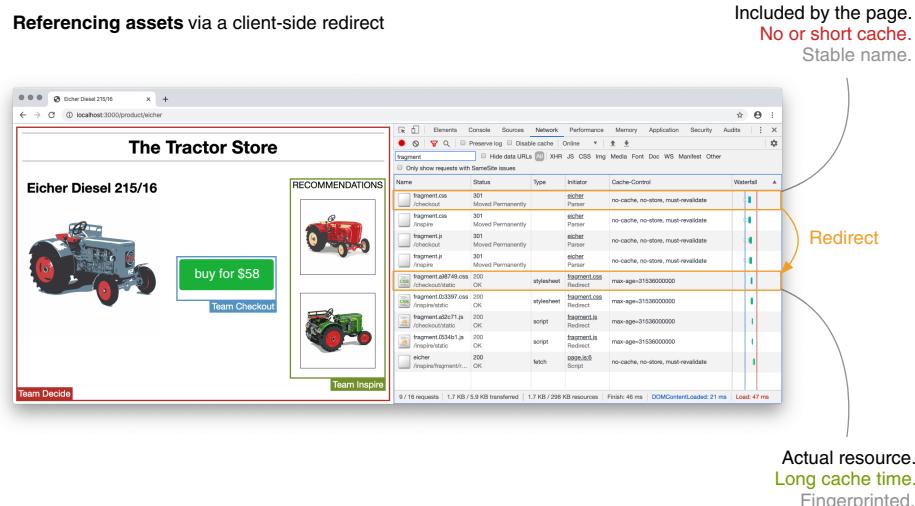


Figure 10.1 Shows the network requests for loading the fragments styles and script. There's a fragment.css and fragment.js for both Team Checkout and Team Inspire. Each resource redirects to the latest version of the file.

NOTE It's possible to achieve similar decoupling and caching results by using Cache-Control: must-revalidate together with the ETag header. But using filename based versioning and long cache headers comes with a few other benefits we'll discuss later.

But there are drawbacks. The browser can't cache the initial resource that returns the redirect. It has to make at least one network request to make sure the redirect still points to the same resource.

A second problem is missing synchronization. The redirect always points to the latest version. When you do rolling deployments, you may have different versions of your software running at the same time. Then you might want to ensure that the CSS, JavaScript, and HTML are all from the same build. We'll address the additional network request first and later talk about synchronization.

10.1.4 Referencing via include (server)

The teams are happy with the improved caching. Ferdinand reruns the Lighthouse test. It shows a 98 score—up 4 points. But a new potential improvement message appears: *Minimize Critical Requests Depth.*⁷

With the redirect approach, we achieve the decoupling and caching benefits at the cost of more network requests. The browser has to make an extra lookup request before it knows the URL of an actual resource. Under poor network conditions, this can introduce a noticeable delay. Let's move this lookup request to the server. Server-to-server communication is much faster. There we talk about latencies on the realm of single-digit milliseconds.

If you are already using a server-side markup integration, we can also use this mechanism to register the assets. The idea is pretty straightforward. At the spot where Team Decide referenced the other team's assets using a link or a script tag, they include a piece of markup that's generated by the respective team.

We'll again use Nginx's SSI feature in our example. You can look back to chapter 4 for more details on how this works.

The product page markup looks like listing 10.3.

NOTE For simplicity, we haven't fingerprinted Team Decide's page.css and page.js files.

Listing 10.3 team-decide/product/eicher.html

```
<html>
  <head>
    <title>Eicher Diesel 215/16</title>
    ...
    <link href="/decide/static/page.css" rel="stylesheet" />
```

⁷ See <https://developers.google.com/web/tools/lighthouse/audits/critical-request-chains>.

```

<!--#include virtual="/checkout/fragment/register_styles" -->
<!--#include virtual="/inspire/fragment/register_styles" -->
</head>
<body>
  <h1>The Tractor Store</h1>
  ...
  <script src="/decide/static/page.js" async></script>
  <!--#include virtual="/checkout/fragment/register_scripts" -->
  <!--#include virtual="/inspire/fragment/register_scripts" -->
</body>
</html>

```

SSI directive will resolve into the link tag markup from the teams

SSI directive will resolve into the script tag markup from the teams

Team Checkout and Team Inspire have to provide the registration endpoints for scripts and styles. These endpoints are now part of the contract between the teams. This example shows the content of one of these registration includes.

Listing 10.4 team-checkout/checkout/fragment/register_styles.html

```
<link href="/checkout/static/fragment.a98749.css" rel="stylesheet" />
```

It's a link tag that points to the fingerprinted asset file. Figure 10.2 illustrates the assembly process. Nginx replaces the directive with the contents of the registration include.

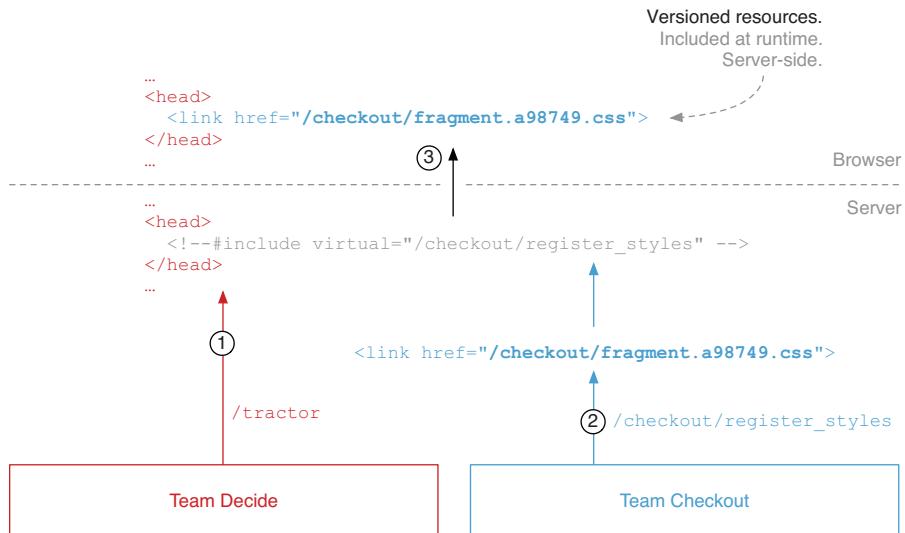


Figure 10.2 Team Decide doesn't reference Team Checkout's assets directly. Instead it adds an SSI include directive which is pointing to Team Checkout's register_style endpoint ①. This endpoint returns HTML markup with the fingerprinted assets. Team Checkout can update the fingerprints on the fly without having to coordinate with Team Decide ②. The browser receives the already assembled markup with the link to the fingerprinted asset ③.

The markup that reaches the browser already contains the resolved include. The browser can instantly start to download the asset. If it's present in the disk cache, the browser can use its local copy without having to make another network request or revalidation. Start the example by running `npm run 18_asset_registration_include`. Figure 10.3 shows how a first page visit looks in the browser.

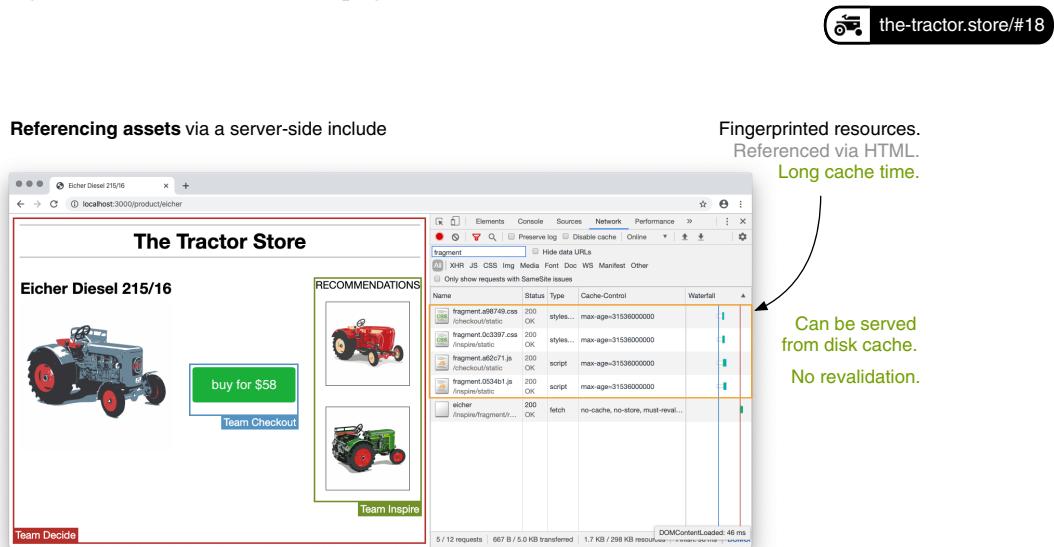


Figure 10.3 Shows the browser's Network tab with the fragment assets required for this page. The HTML directly links to the fingerprinted files from the other teams. The assets are cacheable for a long time.

This approach provides good decoupling. Teams can change their asset URLs without having to notify another team. Because we don't need a client-side redirect or a revalidation request, this is also a perfect solution from a web performance standpoint.

If you don't already have a server-side integration mechanism in place, this approach requires some extra work and shared infrastructure.

10.1.5 Challenge: Synchronizing markup and asset versions

Registering the assets server-side improved the Lighthouse score significantly. Team Decide's product page now scores the full 100 points. The developers and CEO Ferdinand are delighted. They rolled out the change to their production servers.

A week later, Noah, Team Checkout's DevOps guy, recognizes a strange pattern while going through the server logs. From time to time, the application servers report a 404 error on one of their fingerprinted asset files. It seems like the browser is requesting a file which the application does not know. First, he suspects a bug, but after closer examination, he realizes that these issues appear at times when Team Checkout deploys a new version of their software.

After consulting his teammates, Noah is pretty confident about the cause of these issues: *rolling deployments*. To deal with the high amount of traffic, Team Checkout runs 10 instances of their application. The application contains everything from database communication to rendering markup and shipping the assets. The team uses Kubernetes for automatic deployments. During a deployment, Kubernetes incrementally replaces the old applications with new instances. It does this step by step: creating a new instance, waiting until it's operational, redirecting traffic to it, and then killing an old instance. Kubernetes repeats this process until all 10 applications are updated. A full deployment can take a few minutes. During this time, old and new versions of the application run side by side. This running side by side is the cause of the 404 issues.

NOTE The problem gets even worse when you use *canary deployments*. With canary deployments, you roll out the new version to a small percentage of your instances and monitor them for a while. If the new instances perform well, all instances will be updated. If they have performance issues, the team rolls back the deployment. With canary deployments, old and new versions run side by side for a much longer time. The risk of inconsistencies increases.

A load balancer routes an incoming request randomly to one of the 10 application servers to distribute the work load evenly. Imagine the freshly deployed instance serves the registration fragment (`/checkout/register_styles`) but the actual asset request (`/checkout/static/fragment.[new-fingerprint].css`) reaches an old instance which only knows the file with an old fingerprint. This scenario results in a 404 error and an unhappy user who's looking at a page with an unstyled Buy button. Figure 10.4 illustrates this.

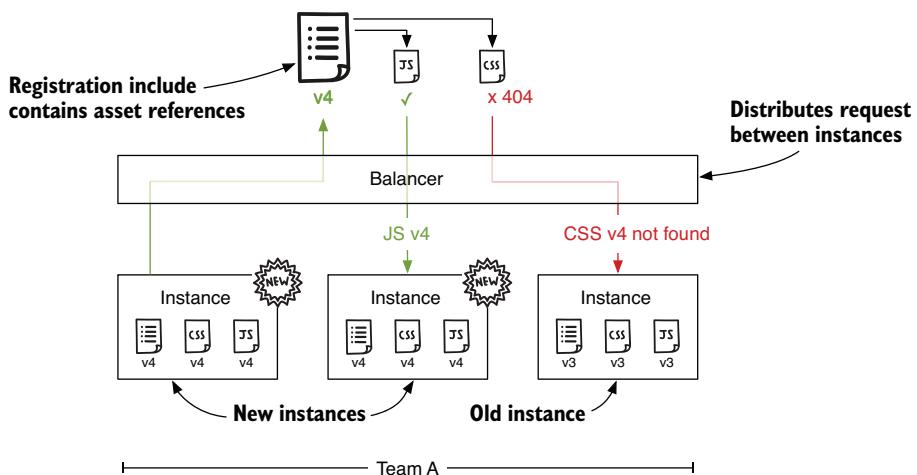


Figure 10.4 Using fingerprinted asset references can lead to issues during rolling deployments. When the registration include comes from an application server with a new version and the actual asset request reaches an old instance that doesn't know this file, the browser receives a 404 error.

There are two quick fixes to avoid this issue:

- 1 *Enable sticky sessions* in the load balancer to ensure that all requests from one user go to the same application server.⁸
- 2 *Serve all assets from a CDN*. Teams push new assets to the CDN before an application deployment. The CDN contains new and old assets.

These fixes reduce the likelihood of the previously described error, but they aren't perfect mitigations. *Sticky sessions are not a guarantee*. When an application server goes down due to a fault or redeployment, users must switch to another application.

The CDN solution also doesn't solve all problems. Not only do you need to ensure that all asset files are present, you also have to guarantee that the fragment markup is compatible with the loaded JavaScript and CSS files. If you ship the new markup with a fancy Christmas teaser, but an old stylesheet is loaded that doesn't contain the associated styles, the site will not look Christmasy but broken. We have to find a way to ensure that markup and asset references always fit together. Figure 10.5 illustrates how this mismatch can happen.

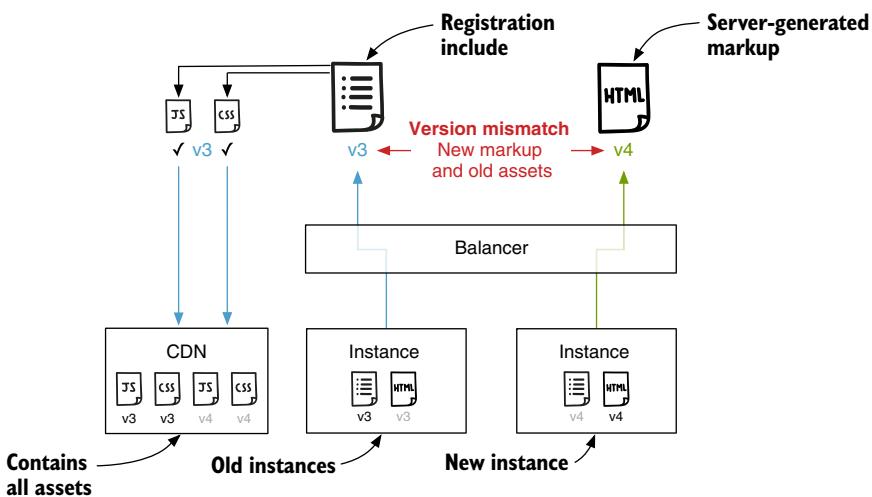


Figure 10.5 In this diagram we use a CDN that contains old and new assets. The CDN ensures that fingerprinted asset requests can always be resolved. However, since the registration include and the actual server-generated markup are retrieved in two separate requests, a version mismatch can occur. Here the old instance (v3) serves the registration include but the actual content is generated by a new version of the application (v4). This results in a version mismatch which may lead to errors in the browser.

NOTE Synchronization is primarily an issue for server-generated markup. When you run a fully client-rendered application, the HTML template is part of the JavaScript file. If you are using a CSS-in-JS solution, the styles are most

⁸ See Zhimin Wen, "Sticky Sessions in Kubernetes," Medium, <http://mng.bz/VgKW>.

likely also part of the JavaScript bundle. Otherwise, you can ship the script and link tag via the same registration fragment to ensure that they are compatible with each other.

10.1.6 Inlining

The easiest way to ensure synchronization is to embed the tags into the markup of the fragment itself. Let's say Team Checkout generates the Buy button markup server-side. Then they could ship the link and style tags directly into the requests that respond with the button markup. It could look like this.

Listing 10.5 team-checkout/fragment/buy-button.html

```
<link href="/checkout/static/fragment.a98749.css" rel="stylesheet" />
<button>buy now</button>
<script src="/checkout/static/fragment.a62c71.js" async></script>
```

Inlining works but comes with a few issues:

- *Redundant link/script tags*—If you have a page that includes the Buy button five times, you will get five identical link and script tags. If the resources are cacheable, browsers are smart and download the files only once.
- *More JavaScript execution*—Even though the browser would download the JavaScript once, it would execute it again for every script tag. Double execution may introduce unforeseen issues and a higher CPU load.
- *Works for server-side integration only*—Since the style and script references are part of the server-generated markup, this solution won't work for client- or universal-rendered micro frontends.

If these trade-offs are acceptable for you, inlining could be a viable and easy-to-build option.

10.1.7 Integrated solutions (Tailor, Podium, ...)

Most micro frontend libraries come with a solution to deal with assets. In chapter 4 we introduced Tailor and Podium. Let's see how they handle JavaScript and CSS.

TAILOR'S ASSET HANDLING

Zalando's Tailor transfers the asset references via an HTTP header. A team can specify the associated assets to a piece of server-generated markup via a Link entry. A response can look like this:

```
$ curl -v http://.../checkout/fragment/buy-button
HTTP/1.1 200 OK
Link: </checkout/static/fragment.a98749.css>; rel="stylesheet",
      </checkout/static/fragment.a62c71.js>; rel="fragment-script"
Content-Type: text/html
Connection: keep-alive
<button>buy now</button>
```

**List of required
CSS and JS files**

**The HTML
content**

Because references and markup are in the same request, we have no synchronization issues. The Tailor service assembles the page and keeps track of all references. In the final markup, it creates a `link` tag for all unique CSS files and loads the JavaScript via the `require.js` module loader.

PODIUM'S ASSET HANDLING

With Podium, a team defines its asset references in a `manifest.json` file. The manifest also contains a version number. Team Checkout's manifest for the Buy button could look like this:

```
$ curl http://.../checkout/fragment/buy-button/manifest.json
{
  "name": "buy-button",
  "version": "4",
  "content": "/checkout/fragment/buy-button",
  "css": [
    { value: "/checkout/static/fragment.a98749.css" }
  ],
  "js": [
    { value: "/checkout/static/fragment.a62c71.js" }
  ]
}
```

The annotations explain the structure of the manifest.json file:

- Deployed version of the software.** Typically a build number or commit hash. Points to the `version` field.
- Endpoint that returns the markup** Points to the `content` field.
- List of associated assets** Points to the `css` and `js` fields.

Team Decide would use Podium's layout library and provide it with the `manifest.json` URLs for all micro frontends the product page needs. On startup, Podium downloads all manifest files to determine the endpoints for the content. These endpoints respond with plain HTML:

```
$ curl -v http://.../checkout/fragment/buy-button/
HTTP/1.1 200 OK
Content-Type: text/html
Connection: keep-alive
podlet-version: 4
<button>buy now</button>
```

The annotations explain the response headers and content:

- Version number of the application** Points to the `podlet-version` header.
- The HTML content** Points to the `<button>buy now</button>` element.

The response also includes a `podlet-version` header. It does not indicate the version of the Podium library. It is a string that uniquely identifies the deployed version of the software. The owner of the fragment (or podlet) has to set it explicitly. It can be a build number or a commit hash. In our example, the version number is "4". It's the same number you find in the `manifest.json` in the previous code.

Every time Podium fetches the HTML content, it compares the `podlet-version` header to the version number in its cached `manifest.json` file. If the versions match, it can use the assets files specified in the current manifest. A difference in version numbers indicates that the owner of the fragment has deployed a new software version. Podium will redownload the `manifest.json` to get a link to the updated assets.

Listing 10.6 team-decide/server.js

```

...
const buyButton = layout.client.register({
  name: 'buy-button',
  uri: 'http://.../checkout/fragment/buy-button/manifest.json'
});

app.get("/product/eicher", async (req, res) => {
  const button = await buyButton.fetch(res.locals.podium);
  console.log(button);
  console.log(button.css);
  console.log(button.js);
  res.send(`<h1>Eicher<h1>${button}`);
});

```

The preceding code shows how Team Decide registers Team Checkout's Buy button micro frontend and fetches the content. Podium performs the synchronization and manifest updating under the hood. Team Decide waits for the promise (`buyButton.fetch`) to resolve and receives the HTML and the assets in one object (`button`). This object contains the HTML as well as the associated asset reference. Team Decide can use it to construct its page markup.

10.1.8 Quick summary

Now you've seen a couple of strategies to pull the required assets for all included micro frontends into your page. As with the markup integration strategies, there are no right or wrong solutions. It depends on your use case. How important is performance and caching? Do you need perfect synchronization, or is it practical to write your CSS and JS in a backward-compatible manner? For the projects I've worked on, we mostly used the server-side generated registration include approach, and have had few issues so far. We accepted the fact that markup and assets might get out of sync for brief periods of time. Table 10.1 summarizes the loading methods and lists their features.

Table 10.1 Properties of registration strategies

Method	Independent deployments	Caching and performance	Guaranteed synchronization
Direct	no	bad	no
Redirect (client)	yes	ok	no
Include (server)	yes	good	no
Inlining	yes	bad	yes
Integrated (Tailor, Podium,)	yes	good	yes

Whichever concrete solution you choose, it's essential to define a uniform way that all teams use. A producer of a micro frontend must be able to count on the fact that the page owner references their assets correctly. They also need to be able to update their assets without manually notifying other teams. Table 10.2 shows the technical contracts between a micro frontend owner and user. What does Team A have to know about Team B's micro frontend to use it?

Table 10.2 Contract for loading required assets

Method	Inter-team contract	Example
Direct	Asset file URLs	/checkout/fragment.js /checkout/fragment.css
Redirect (client)	Asset file URLs	/checkout/fragment.js /checkout/fragment.css
Include (server)	Endpoints with registration markup	/checkout/register_scripts /checkout/register_styles
Inlining	None (<i>only for server markup</i>)	
Tailor	HTTP-Header	Link: <fragment.css>; <fragment.js>
Podium	manifest.json	/checkout/manifest.json

10.2 Bundle granularity

We've talked about how to load the assets for your micro frontends. Now let's look at the files themselves. What granularity should the asset files have? One file per micro frontend, one file per team, or even a single huge file for the complete project?

10.2.1 HTTP/2

Best practices change over time. Some years ago, it was crucial to load as few resources as possible to keep the number of network requests down. Bundling up everything in one file and combining multiple images into one (spriting) was widespread. A couple of years later, tools like Google PageSpeed heavily rewarded inlining the CSS for the viewport into the HTML to ship everything needed for the first render in only a few TCP packets.

With the introduction of HTTP/2, these best practices became bad practices. The protocol reduced the overhead cost of loading multiple resources from the same domain. Its built-in multiplexing and server push features removed the need to manually inline assets into the page, which reduces complexity in the application and is also great for cacheability.

These HTTP/2 features come in handy when you are building a micro-frontends-style application.

10.2.2 All-in-one bundle

In 2014 I worked on my first project with vertically organized teams. Back then, we had lengthy discussions about the necessity of building an overarching asset bundling process. This bundling service would collect the scripts and styles from all teams to combine them into one single file for delivery. Luckily we decided not to introduce such a central service, but I know of other projects that did that.

A central asset bundler introduces a significant amount of coupling and friction. Someone needs to build and maintain that service. Deployments have to be synchronized between the asset service and the applications to ensure that the markup always matches the delivered assets. Today it's an anti-pattern to deliver all-in-one bundles for most use cases:

- The cost of shipping a lot of unused code outweighs the gains of using fewer requests.
- The chance of cache invalidation is high. The complete bundle needs to be redownloaded even if only a small part has changed.

But even today, a central bundler can provide one valuable feature: *elimination of redundant code*. When two teams use the same JavaScript library or button styling, the central service could remove one instance of it to make the bundle smaller. In the next chapter we'll discuss options for how to remove redundancy without introducing a shared service.

10.2.3 Team bundles

In our example application, each team has one page and fragment bundle. For the product page, Team Decide loads their page bundle. If they want to include Team Checkout's Buy button micro frontend, they would also need to add Team Checkout's fragment bundle.

Since HTTP/2 makes additional requests very cheap but not free, you should still use bundling inside your team and not confront the browser with your raw component and dependency tree. In the projects I worked on, the bundle-per-team approach has proven itself a reasonable trade-off between bundle size, overfetching, and reusability across pages.

But as always, it depends on your use case. If one team provides a fragment that requires a lot of CSS code, but only one niche page uses it, it might make sense to create a separate bundle for it.

10.2.4 Page and fragment bundles

The *one bundle per micro frontend* approach takes granularity one step further. There every fragment or page has its script and style bundle. You can think of this as adding an import statement to the top of your file before you can use the actual component in your code.

This more fine-grained way of bundling ensures that you only download code the customer needs on the page. But depending on your page structure and the number of fragments you include, it could lead to quite a few assets that need to be loaded.

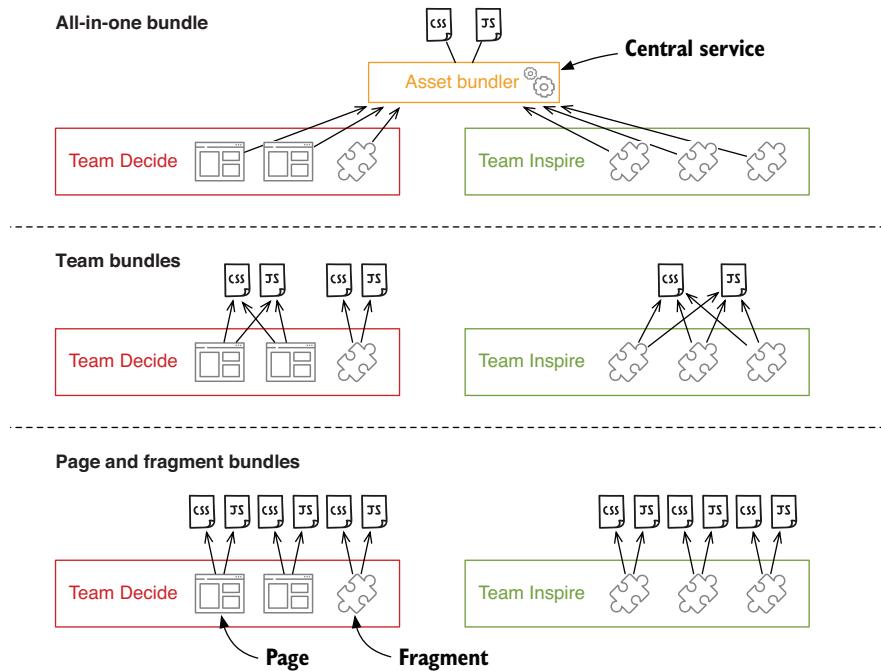


Figure 10.6 Different asset bundle granularities

Figure 10.6 illustrates the three bundling strategies. Pick the strategy that fits your needs best.

10.3 On-demand loading

Picking the bundle granularity is essential because it affects the contracts between the teams. *But not all code has to live directly in this bundle.*

A team can adopt techniques like code splitting inside of their bundle to further improve the loading behavior, reducing initial download size and fetching parts of the code when the user needs them.

Say Team Checkout's Buy button would open a fancy layer that requires a bunch of JavaScript. The team could take the layer code out of the initial fragment bundle and fetch it when the user hovers over the button.

10.3.1 Proxy micro frontends

But we could reduce the bundle size even further. Say your asset file contains the code for five different micro frontends, which are rarely used together on one page. Instead of putting the code directly into the file, you can set up proxy components that fetch the real code when it's needed the first time. If you are using Custom Elements, the code could look like this.

Listing 10.7 team-checkout/static/fragment.js

```
class CheckoutBuyProxy extends HTMLElement {  
    constructor() {  
        import("./real-buy-button.js").then(...); ← Dynamically loads the real  
implementation of the Buy button  
when needed for the first time  
    }  
}  
window.customElements.define("checkout-buy", CheckoutBuyProxy);
```

WARNING Proxying a Custom Element is more complex than this example. It's currently not possible to update a Custom Element definition solely by registering a new class, and the lifecycle methods are synchronous. But we can't go more in-depth into Web Component land in this book. You'll find resources for doing this properly on the internet.

Shipping micro frontend proxies in your asset bundle can reduce initial download, which is good.

10.3.2 Lazy loading CSS

If you are using plain CSS, lazy loading is not that easy because there is no native browser support to split and load CSS files dynamically. But many CSS-in-JS solutions, CSS Modules, and most bundlers come with mechanisms to enable lazy loading for CSS without a bunch of manual work.

These are standard performance optimization techniques you would also use in a monolithic frontend. In a micro frontend architecture, each team can adopt them for their part of the system.

Summary

- Teams must be able to update their assets without having to coordinate with other teams.
- The asset paths must be part of the contract between the teams. A team that uses a micro frontend needs to add the associated assets.
- There are different ways to communicate the asset URLs: via documentation, through a redirect or registration include, via HTTP headers, or via a machine-readable manifest file.
- If you render on the server, you need to ensure that the JavaScript and CSS files match the version of the generated markup. For pure client-side rendering, this is less of an issue since the template is part of the JavaScript itself.
- The development teams must implement performance optimization techniques like on-demand loading inside their applications. Try to avoid overarchitecting optimizations like a shared asset bundling service. They introduce extra coupling and complexity.

11

Performance is key

This chapter covers

- Examining how to measure performance when multiple micro frontends exist on one page
- How to find regressions and bottlenecks and attribute them to the right team
- Typical performance drawbacks that are consequential to the micro frontends architecture
- Reducing the amount of required JavaScript by sharing larger vendor libraries across teams
- Implementing library sharing without compromising team independence

In 2014 my colleague Jens handed me an article¹ written by a company that implemented a vertical style architecture. Back then, the term *micro frontends* didn't exist. Being a frontend developer who takes pride in delivering fast user experiences, my

¹ S. Kraus, G. Steinacker, O. Wegner. "Teile und Herrsche: Kleine Systeme für große Architekturen," OBJEKTspektrum 05/2013 (German), <http://mng.bz/xWDg>.

first gut reaction to this idea was rejection—strong rejection. “*Five teams that all roll their own frontend? This sounds like a lot of overhead. The result will surely be inefficient and slow.*”

Today, when I introduce micro frontends to developers, I often get a similar reaction. They understand the concept and its benefits, but sacrificing performance for increased development speed can be hard to swallow. Having worked in micro frontend projects over the last years, my initial worries faded quickly. This does not mean that my concerns were unfounded or magically resolved themselves. *Autonomy inherently comes with the cost of accepting redundancy.* But I learned to focus on the bottlenecks that have a real impact for our users instead of reflexively fighting code duplication.

The micro frontend projects we built all outperformed the monolith they replaced. This resulted in faster responses, less code shipped to the browser, and better overall load times. One factor these projects had in common was that architecting for excellent performance was a top priority from the start and not an afterthought. Another significant benefit I experienced while working with micro frontends was that the architecture made it easier to optimize the user experience in the places where it made the most significant difference. But more on this later.

In this chapter, you’ll learn how to address performance in your micro frontends project. We’ll start with the “definition of fast.” What does “performant” mean for the different parts of your project? Measuring performance and acting upon the results is tricky when the frontend includes code from different teams. I’ll show you some strategies that have proven valuable when architecting for excellent performance. At the end of the chapter, you’ll learn how to keep your JavaScript overhead to a healthy minimum, avoiding large redundant framework downloads while still being able to deploy autonomously.

11.1 Architecting for performance

Early on in the project, Finn, Tractor Models, Inc.’s lead architect, arranged a meeting with developers from all three teams. Together they defined some performance guidelines that act as the default for all pages of the shop. They decided that the total weight of a page should never exceed 1MB of data. The viewport of a page must render in one second under good conditions and three seconds under 3G network conditions.

11.1.1 Different teams, different metrics

They arrived at these values by looking at their competitors’ websites. The team knows that excellent performance is vital for e-commerce. Users enjoy sites that feel fast. They spend more time browsing and have a higher chance of actually buying a tractor. But what does *feel fast* actually mean? Figure 11.1 illustrates this. Different parts of the site have different performance requirements:

- A user that opens the home page for the first time mainly cares about seeing the content without waiting.
- On the product page, the main image (hero image) is most important and should be one of the first items to load.

- When the user enters the checkout process, it's all about interaction—trusting the system while entering personal data. For that, the software must react reliably and swiftly.

The meaning of **good performance** depends on the **task at hand**.

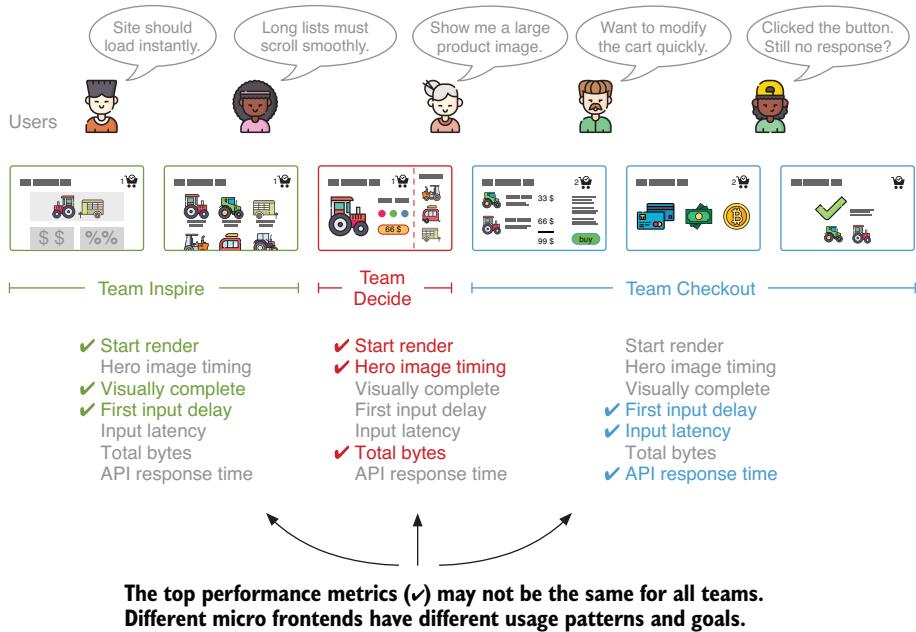


Figure 11.1 The metrics a team should optimize to depend on their use case. The performance expectations of the homepage are not the same as the performance goals for the checkout process.

Having some overarching rules that act as a performance baseline is good. You can see them as basic hygienic requirements. But if you want to optimize further, the metrics a team should focus on will vary depending on the context the user is in. Each team must understand the performance requirements of its subdomain and pick their own metrics.

11.1.2 Multi-team performance budgets

Picking a metric and defining a concrete limit is also called a *performance budget*.² A performance budget is a perfect tool to establish a performance-oriented culture inside a team. The mechanism is simple:

- Your team defines a concrete budget for a specific metric. Say, your site should never be bigger than 1MB.

² See Tim Kadlec, “Setting a performance budget,” *Tim Kadlec*, <http://mng.bz/VgAW>.

- *You continuously measure this metric to ensure your site stays in budget.* Lighthouse CI, sitespeed.io, Speedcurve, Calibre, Google Analytics, and so on are useful tools for that.
- *If a new feature breaks your site's budget, the development stops.* Developers investigate the cause of the degradation. Then the complete team, including product managers, discusses options to get back into budget: rolling back the change, implementing an optimization, or even removing another feature from the page.

Budgets are a powerful mechanism for fostering performance discussions on a regular basis. But how can budgets work for a site with micro frontends from multiple teams? Should Team A stop their development because Team B's micro frontend slows down the page? Maybe!

You can address this in different ways:

- *Dividing the budget to all micro frontends.* This is the analytical approach. This would mean that a page containing five micro frontends could, for example, use 500KB for the content of the page itself and grant each included micro frontend a budget of 100KB. Adding everything up will get us to our 1MB ($500\text{KB} + 5 * 100\text{KB}$) size budget. In theory, this works, and for metrics like bytes and server response times, it's possible to measure and sum up the pieces like that. But for metrics like load time, lighthouse score, or time to interactive, it's not that linear.
- *Page owners are responsible.* This is the social approach. Here budgets are always on a page level. A page owner is in charge of staying in that budget. In our example, Team Decide would be responsible for the product page. The team's goal is to provide the user with the best experience possible. Should an included micro frontend use an unreasonable amount of resources, Team Decide contacts the owner, explains, and discusses options. You can view an includable micro frontend as a guest that tries to be as well-behaved as possible.

We've had good experiences with the latter approach. It avoids getting into the weeds with too fine-grained budgets: Should a recommendation strip consume 100KB, or is 150KB more reasonable? The responsibility is very clear. When the product page is slow, it's Team Decide that needs to become active. Yes, the team might not have caused the issue, but it's their task to get back to a performant state by finding the cause of the problem and informing the right team.

For the page-owning team, this might feel cumbersome at first. But in practice, this has worked well for us. No team that provides an includable micro frontend wants to be the "slow kid" that's holding everyone back. Teams started to measure the performance of their micro frontends in isolation to detect regressions before they go to production.

11.1.3 Attributing slowdowns

Team Decide installed a large dashboard screen in their office area. It shows the performance of their system with live updating charts and big green numbers. One day the team came back from lunch and noticed that the average load time of the product page's main image tripled. Before, the product images rendered in around 300 ms;

now it takes nearly a second. The team checked their last commits but didn't find any suspicious change that could have caused such an issue. They checked the site in the browser. It didn't look broken.

They figured that an included micro frontend from another team might be responsible. Since they use server-side integration, this slowdown can be due to a service that has issues producing its micro frontend's markup (see chapter 4). They opened up the centralized metrics system where they could see the response times of every endpoint in the platform. None of the endpoints that are used to assemble the product page's markup showed anomalies.

Then they checked their web performance monitoring tool. This tool opens the product page on a regular basis with a real web browser, recording a video of the process and also storing the browser's network graph. With this tool, the team was able to compare the product page from before lunch with the current slower version. This before-and-after comparison highlighted the real issue. Before, the user loaded four images—Team Decide's big hero image and three images from the recommendation strip. Now the network graph shows 13 images. With this information and a bit of digging, it became apparent that the recommendation strip was the cause of the slowdown.

It turned out that Team Inspire implemented a carousel feature for their recommendations. Now users can tap on a small arrow to see more matching products. But this simple carousel implementation did not feature any lazy loading. Even though the user only sees three recommendation images at a time, all images from the carousel load up-front. Jeremy, Team Decide's product owner, walked over to Team Inspire's office space and explained the issue. Team Inspire rolled back their carousel feature. They implemented lazy loading for the images and reintroduced the optimized version the next day.

OBSERVABILITY

Debugging a distributed system is a challenging task. The root of a problem is not always visible. Investing in proper monitoring will make finding issues much more manageable. If you integrate markup on the server, it's crucial to know how long the different parts of the page take to produce. Having a central view with the deployments of all teams can help to correlate a measured effect to a specific change in the system.

Monitoring the code that runs in the browser can be tricky. The software from all teams has to share bandwidth, memory, and CPU resources. Having video recordings, network graphs, and metrics you can compare over time is a vital first step. Implementing unique team prefixes for all resources the browser loads also helps (see chapter 3). This way, the ownership of a file that looks suspicious is always evident.

ISOLATION

A popular debugging technique is isolating the issue. Imagine you've written a piece of software and notice a mysterious bug that you can't explain. A good strategy for finding the root cause is to comment out parts of your code and check if the bug is still present.

You can apply the same approach to a micro frontends website. The “Block URL” feature in the Network tab of your favorite browser is your friend. Block the scripts or styles from a specific team. This way, you can test your site without the code of Team B or Team C and measure if the performance degradation or error still exists.

11.1.4 Performance benefits

I often talk about performance challenges the micro frontends concept introduces. But this approach comes with a couple of positive properties this approach.

BUILT-IN CODE SPLITTING

With the move to HTTP/2, it has become a best practice to split an application’s JavaScript and CSS code into smaller pieces. We talked about this in chapter 10. Delivering the code into smaller chunks (per team, per micro frontend) and not into one monolithic blob has benefits:

- *Cacheability*—Browsers only need to redownload the parts of the code that changed. Not everything. Micro frontends are often used in conjunction with continuous delivery, where teams deploy to production several times a day.
- *Fewer long-running tasks*—The browser’s main thread becomes unresponsive as it processes a JavaScript file. Loading multiple smaller files gives it more room to breathe and accept user input in-between processing the JavaScript resources.
- *On-demand loading*—Since the assets are often grouped by team or micro frontend, it’s easy to include only the code a page needs or implement route-based loading as we saw in the single-spa example. The user doesn’t have to download the code for the cart page when they visit the homepage.

These benefits are by no means exclusive to micro frontends. You can also achieve these optimizations in a well-architected monolithic frontend. But the way you think about and develop features in a micro frontends project naturally guides you toward this structure.

OPTIMIZING FOR THE USE CASE

Developers working in a micro frontends team have a much narrower scope. A team focuses on one specific set of use cases to help the customer. It’s in the interest of the team to optimize this use case as much as possible.

Let me give you an example. Imagine Team Inspire is responsible for displaying promotional teasers in different areas of the shop. These images or videos are often large in size and have a considerable performance impact. Since the team controls the complete process from teaser creation, uploading, and delivery, it’s easy for them to experiment with new file formats like WebP, AV1, or H.265 to speed up teaser loading.

They don’t have to think about what a format switch would mean for product images or user uploaded review videos. This focus on teasers allows them to move quicker. No big meetings with everyone who has an opinion about image or video formats. No grand rollout plans. No big business case calculations. No compromises. The team has everything it needs to improve its teasers.

After the experiment, Team Inspire shares their learnings with the other teams. They're helping the other teams to not fall into the same traps when they try something similar.

Being able to have this kind of focus and control is the biggest strength of a micro frontends architecture. Not only can it lead to better web performance of the individual pieces, but it improves quality and increases user focus.

EASIER CHANGES

The narrower scope makes it possible for a developer to know every aspect of the software—something that is not possible in sizeable monolithic frontend projects. Have you ever deleted an old dependency that isn't in use anymore, only to realize two days later that you broke an obscure marketing page you didn't even know existed? I definitely have.

Having clearly isolated micro frontends reduces the risk of cleaning up dramatically. This makes it easier to lose old cruft and evolve the software.

11.2 *Reduce, reuse... vendor libraries*

The most discussed performance optimization topic for micro frontends is how to deal with libraries that are the same across teams. Downloading the same code twice triggers the Pavlovian reflex for all frontend developers: "This is inefficient, and we must avoid it!" But let's take a step back and question this reflex. We'll take a more analytical look at the topic of redundant code.

11.2.1 *Cost of autonomy*

Tractor Models, Inc.'s three development teams all chose to go with the same JavaScript framework to build their frontend. Before they started the project, lead architect Finn and the teams discussed three different options:

- 1 *Alignment*—Everyone uses the same framework.
- 2 *No constraints*—Every team can choose the framework they want.
- 3 *Some constraints*—Free choice, as long as the framework has specific properties, like having a runtime that's smaller than 10 KB.

Each option has its benefits and drawbacks. They went for the *everyone uses the same framework* option for two reasons. First, teams can help each other because they are all familiar with the same stack. Second, recruiting is easier: developers switching teams get up to speed quickly, and human resources can use the same job profile for all teams.

Although all teams start with the same tech stack, lead architect Finn emphasized that this decision is not set in stone. It should be possible for a new team to pick another stack if there are good reasons. The same goes for version upgrades or migration to a newer, better framework in the future. Teams must maintain their autonomy. All integration techniques and architecture-level artifacts have to be technology agnostic.

They use the JavaScript framework to generate server-side markup. However, for interactive features, the framework also needs to run in the browser. Each team has its own git repository and a dedicated deployment pipeline. The team's JavaScript bundler builds an optimized asset file that's self-contained. It includes everything that the

team uses. If teams use the same dependency, the client will download it multiple times. We can optimize this by providing the large framework code as a separate download from a central place. See the example in figure 11.2.

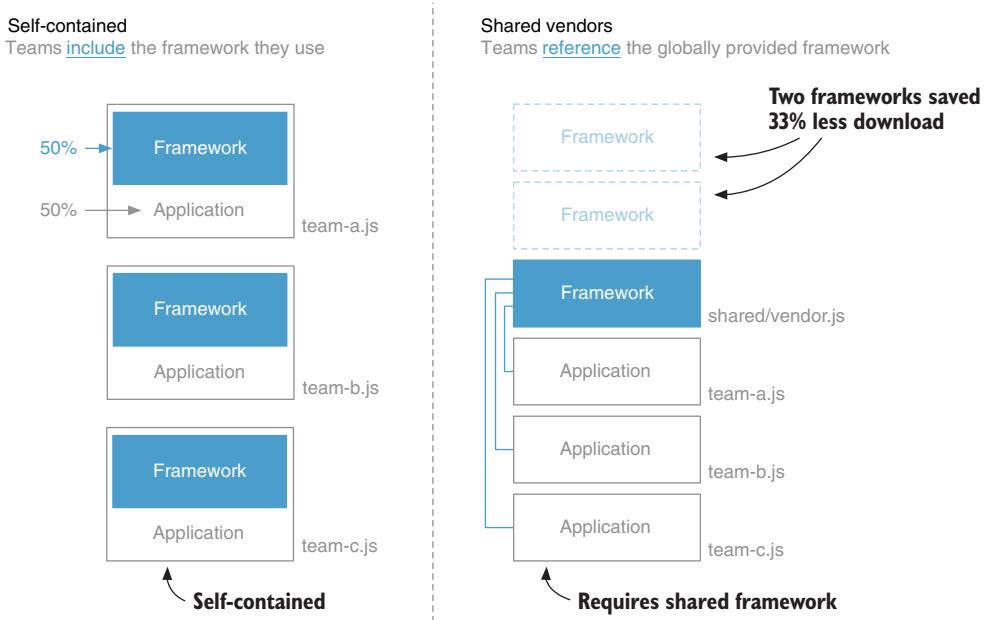


Figure 11.2 A team's JavaScript should be **self-contained**. It should be able to function on its own. That's why bundling all dependencies and vendor libraries is the easiest option (left side). When all teams use the same framework, it could be a worthwhile optimization to host the framework code in a central place (right side). This reduces the amount of network traffic and lowers memory footprint and CPU usage on the user's device.

The figure shows three teams using the same framework. In our case, the framework code makes up for 50% of the team's bundle size. Removing the framework from the team bundles and providing it from a central place decreases the JavaScript size by 33%. The user saves two framework downloads. This sounds like a good optimization, but before we go ahead and build this, we should look at real numbers and the project's demands.

11.2.2 Pick small

The amount of overhead obviously depends on the framework and other libraries you choose. Going with a large framework like Angular will increase the need to centralize vendor code.³ Even though the major big frameworks have gained a lot of popularity, you can see a trend toward adopting smaller libraries and frameworks.

³ If you are building an Angular project, you should check out Manfred Steyer's work on Angular, micro frontends, and reducing Angular bundle size at <https://www.softwarearchitekt.at/blog/>.

Picking a stack like Preact, hyperapp, lit-html, or Stencil will reduce framework overhead. Tools like Svelte go even a step further. They don't have vendor runtime code at all. The source code gets transpiled to native DOM operations. This way, your JavaScript bundle grows proportionally with the features you build. No fixed costs are introduced by the framework.

No worries, we won't go into the "What's the best framework?" discussion. Comparing a batteries-included framework like Angular to the small templating library like lit-html is an apples-to-oranges comparison. However, since the individual micro frontends are smaller in scope, it can be a viable option not to pick an almighty framework that has you covered for everything the future can bring. It might be worth going with a leaner option that's better tailored to your use case. If your bundle includes little vendor code, the overhead of loading it multiple times diminishes.

The other factor you should take into consideration is the team boundaries. How much composition does a typical page require? If you don't use composition at all and every team manages its own set of pages, there is no overhead when loading a page. You only have the disadvantage that vendor libraries aren't cached between the pages of different teams. But the importance of minimizing redundancy increases with the number of different teams that run a micro frontend on one page. Figure 11.3 shows a rough calculation that gives you an idea of how much code we are talking about.

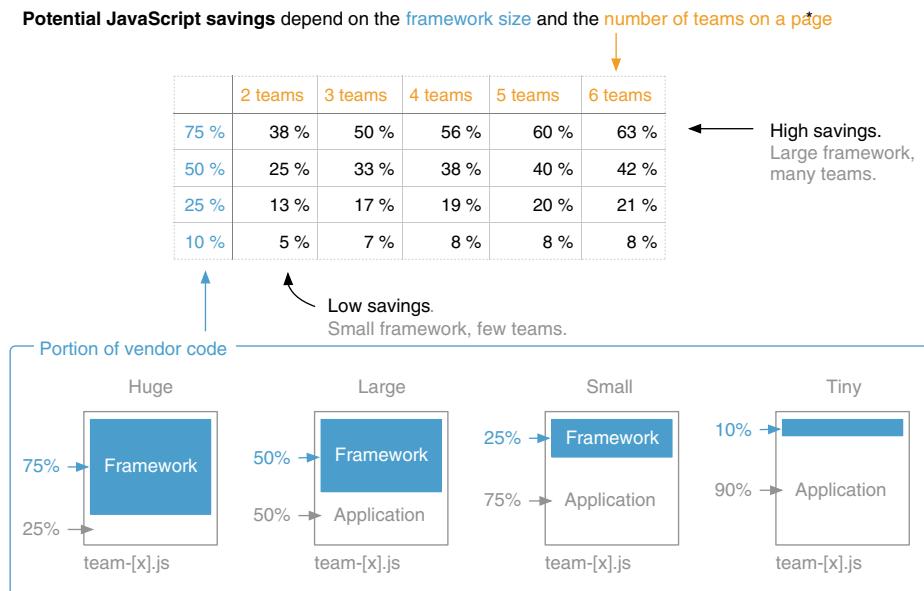


Figure 11.3 The potential savings depend on the portion of vendor code the teams include and the number of teams that are active on one page. Using small dependencies reduces the overhead noticeably. If multiple teams include larger libraries, you can save a lot of JavaScript by centralizing vendor code.

Now we have rough numbers to qualify the overhead in bytes. *It's still essential to measure the real performance implications for your use case and target audience.* Intelligent on-demand loading and good code splitting can make a more significant difference than shaving an extra 25 KB off your JavaScript bundle.

11.2.3 One global version

The teams at Tractor Models, Inc. decided that centralizing their framework code is an optimization worthwhile pursuing. They wanted to start with the most straightforward implementation possible. When we can assume that all teams are on the same version of one framework, we can use a low-tech solution:

- 1 Including the framework as a global script tag.
- 2 Excluding the framework from the team bundles and referencing it once from a global location.

The associated HTML code can look like this.

Listing 11.1 team-decide/index.html

```
<body>
  ...
  <script src="/shared/react.16.11.0.min.js"></script>
  <script src="/shared/react-dom.16.11.0.min.js"></script>

  <script src="/decide/static/bundle.js" async></script>
  <script src="/inspire/static/bundle.js" async></script>
  <script src="/checkout/static/bundle.js" async></script>
</body>
```

The React script tags attach their code to the window object. Teams can call it via `window.React` or `window.ReactDOM`. All bundlers provide an option to mark a library as “globally available.” Webpack calls this concept externals. This removes the code from the bundle and replaces it with a reference to a given variable. The configuration for Webpack can look like this.

Listing 11.2 team-decide/webpack.config.js

```
const webpack = require("webpack");

module.exports = {
  externals: {
    react: 'React',
    'react-dom': 'ReactDOM'
  }
};
```

Voilà! That's it. We've eliminated the redundant framework code.

But we've created a new central artifact (/shared/...) which someone must maintain. Tractor Models, Inc. decided not to instantiate a dedicated platform team. Instead, one of the feature teams should take over responsibility. Team Checkout volunteers to do the job. This team now ensures that the files get deployed to the correct location and coordinates version upgrades with their neighbor teams.

11.2.4 Versioned vendor bundles

The centralization worked well and improved performance measurably. Keeping the React version up-to-date was also not a big issue for Team Checkout. Every time a new React version came out, they informed all teams they needed to test their software against it, deployed the new files to the shared folder two days later, and ensured that the markup references the new script.

But when React 17, the next major version, was announced, it became complicated. It included breaking changes requiring the teams to restructure parts of their existing software.

Team Checkout and Team Decide made the required changes to their codebase in the first week after the announcement. However, they were not able to deploy their migration because Team Inspire wasn't ready. This team was in the middle of a major rewrite of their recommendation algorithms to bring personalized product suggestions to the next level. Moving to the next version of React at the same time was not an option for them. This task had to wait until the algorithm update shipped. So the other teams had no choice but to park their changes in a git branch and wait for the other team.

Three weeks later, Team Inspire managed to get their React migration done—paving the way to finally ship the new framework. The teams agreed on a day and time for the deployment of all software systems and the updated React library. Otherwise, the functionality of the site might be faulty if the central framework did not match with the application code.

This is what's often referred to as a *lock-step deployment*. If you're operating on a smaller scale, it might be fine to do such a manually orchestrated deployment from time to time. It gets extra fun when one team discovers that they need to roll back to the previous version because they found a severe bug. Then we have a *lock-step rollback*. These kinds of activities are exhausting and unsatisfactory. Furthermore, it contradicts the *autonomous deployments* paradigm of micro frontends.

A solution to this problem is to move away from one central framework to a versioned approach. Figure 11.4 shows a deployment process where two teams upgrade from Vue.js 2 to Vue.js 3:

- 1 Before the migration, both teams reference version 2.
- 2 Vue.js 3 gets published as a shared library.
- 3 Team B migrates first and deploys their software, which now references the new framework.
- 4 Team A migrates as well. Now both teams are on version 3. The old version 2 is not referenced any more.

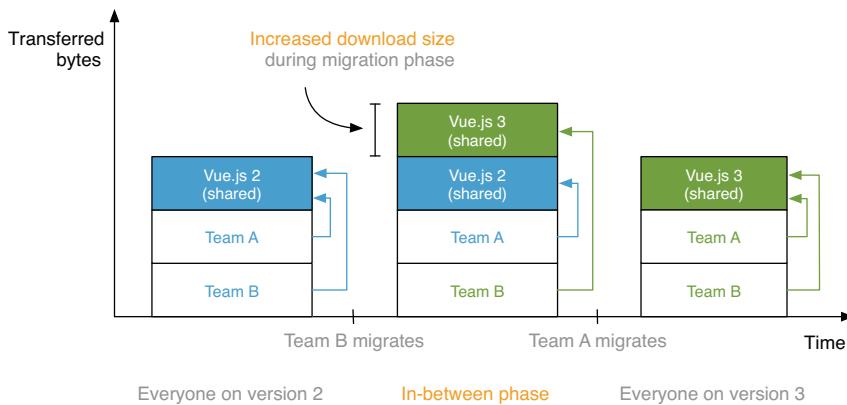


Figure 11.4 Illustrates a framework upgrade process. Team A and B both reference Vue.js v2 from a central location. The framework is loaded once. Now Team B migrates to Vue.js v3. Now the user has to download two Vue.js libraries (v2 and v3). In the last step, Team A also migrates to the latest version. At this point, both teams use Vue.js v3 and the user must only download one version.

With this approach, both teams can upgrade at their own pace. They control which version of the library their code references. Even a rollback is possible without having to coordinate with another team. The only drawback is that the total download size increases during the migration phase.

There are a lot of different ways of achieving this. Let's explore some possible solutions.

WEBPACK DLLPLUGIN

TIP You can find the sample code for this in the `19_shared_vendor_webpack_dll` folder.

The Webpack bundler enjoys great popularity. It includes a tool called `DllPlugin`.⁴ It strangely gets its name from the dynamic link library concept Windows users are familiar with. The plugin works in two steps:

- 1 You can *create a versioned bundle* with the shared dependencies. The plugin generates the JavaScript, which you can host statically, and a manifest file. Think of the manifest as the table of contents for the vendor bundle.
- 2 You provide this manifest to the teams (e.g., via an NPM package). The team's Webpack configuration reads that manifest, omits all listed vendor libraries from its own build, and adds references to the versioned libraries of the central vendor bundle.

⁴ See <https://webpack.js.org/plugins/dll-plugin/>.

Let's look at the structure of the sample project in figure 11.5.

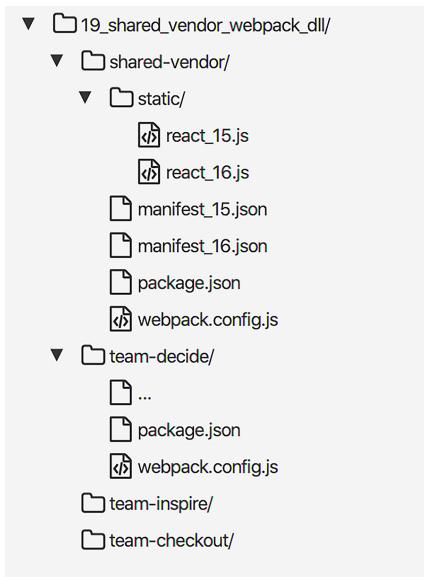


Figure 11.5 Folder structure of the Webpack DllPlugin example project. We've introduced a `shared-vendor/` folder that sits beside the teams. It's the project that generates the shared vendor bundles (`static/`) using the DllPlugin. It also includes the `manifest_[x].json` files for each version. The teams use Webpack for packaging their application code. You can find the configuration in `webpack.config.js`.

The `shared-vendor/` folder contains the JavaScript and manifest code for versions 15 and 16.

CREATING THE VERSIONED BUNDLE

We'll go through the essential pieces required to make this happen. Here is an excerpt from the vendor bundles `package.json`.

Listing 11.3 `shared-vendor/package.json`

```
{
  "name": "shared-vendor",
  "version": "16.12.0",
  "dependencies": {
    "react": "^16.12.0",
    "react-dom": "^16.12.0"
  },
  ...
}
```

Specifying the dependencies and their version

Here is the Webpack code for generating JavaScript and manifest.

Listing 11.4 `shared-vendor/webpack.config.js`

```
const path = require("path");
const webpack = require("webpack");
```

```

module.exports = {
  ...
  entry: { react: ["react", "react-dom"] }, ←
  output: {
    filename: "[name]_16.js",
    path: path.resolve(__dirname, "./static"),
    library: "[name]_[hash]"
  },
  plugins: [
    new webpack.DllPlugin({
      context: __dirname,
      name: "[name]_[hash]",
      path: path.resolve(__dirname, "manifest_16.json")
    })
  ]
};

```

List of dependencies to include in the vendor bundle. Here one bundle called react gets created. It contains the code of react and react-dom.

Configuring location and name for the JavaScript code

Adding the DllPlugin and specifying where to write the manifest file

USING THE VERSIONED BUNDLE

The teams must have access to the desired manifest at build time. Publishing the shared-vendor project as an NPM module is one option to do this. Here is the package.json for a team.

Listing 11.5 team-decide/package.json

```
{
  "name": "team-decide",
  "dependencies": {
    ...
    "react": "^16.12.0",
    "react-dom": "^16.12.0",
    "shared-vendor": "file:../shared-vendor"
  },
  ...
}
```

Specifying the framework dependencies

Referencing the shared-vendor package. We use the file: syntax to make it happen locally. In the real project we would publish it as a properly named and versioned package like this: @the-tractor-store/shared-vendor@16.12.0.

The Webpack configuration of the team looks like this.

Listing 11.6 team-decide/webpack.config.js

```

const webpack = require("webpack");
const path = require("path");

module.exports = {
  entry: "./src/page.jsx",
  output: {
    ...
    publicPath: "/static/",
    filename: "decide.js"
  },
  plugins: [

```

Entry point of team decides application

Configuring where the generated files should go

```

new webpack.DllReferencePlugin({
  context: path.join(__dirname),
  manifest: require("shared-vendor/manifest_16.json"),
  sourceType: "var"
})
]
...
};

}

```

Adding the `DllReferencePlugin` and pointing it to the `manifest_[x].json` of the `shared-vendor` package

This is a pretty standard Webpack configuration. Adding the `DllReferencePlugin` is the special part. It performs the magic of omitting the code of all vendor libraries specified in the `manifest.json` and replacing it with references to the central bundle.

Are you curious about the manifest's content? Let's have a look inside.

Listing 11.7 `shared-vendor/manifest_16.json`

```

{
  "name": "react_a00e3596104ad95690e8", ← Unique internal name. Ensures that
  "content": { different DLLs can exist on one page.
    "./node_modules/react/index.js": {
      "id": 0,
      "buildMeta": { "providedExports": true }
    },
    "./node_modules/object-assign/index.js": {
      "id": 1,
      "buildMeta": { "providedExports": true }
    },
    ...
  }
}

```

List of node modules that the bundle contains

The bundle also contains the dependencies of the dependencies.

The last step in our process is adjusting the script tags in the HTML to ensure that the bundles load in the correct order.

Listing 11.8 `team-decide/index.html`

```

<html>
  ...
  <body>
    <decide-product-page></decide-product-page>
    <script src="http://localhost:3000/static/react_15.js"></script>
    <script src="http://localhost:3000/static/react_16.js"></script>
    <script src="http://localhost:3001/static/decide.js" async></script>
    <script src="http://localhost:3002/static/inspire.js" async></script>
    <script src="http://localhost:3003/static/checkout.js" async></script>
  </body>
</html>

```

Including the bundle for both React versions

It's crucial that the vendor bundles execute before the team's code. Run the example locally (`npm run 19_shared_vendor_webpack_dll`) and look at the output at `http://localhost:3001/product/fendt`. Figure 11.6 shows the result.

You can see that the micro frontends run on different React versions.



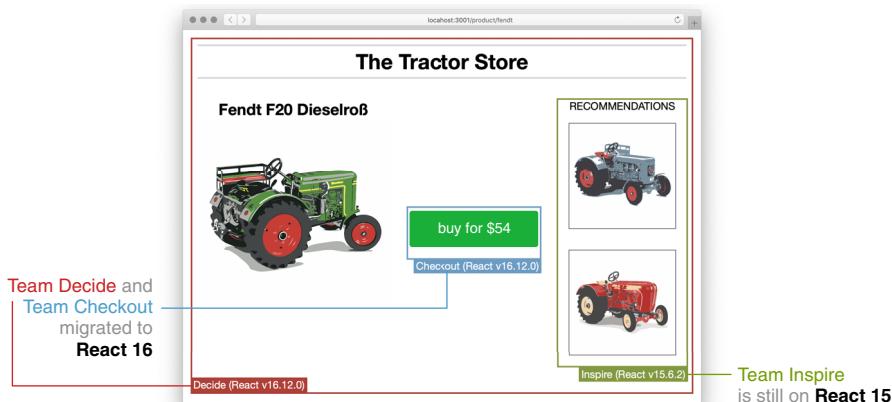


Figure 11.6 Team Decide’s and Team Checkout’s micro frontends run on React 16. Team Inspire still uses version 15. The different versions can coexist on the same page.

The DllPlugin has some benefits compared to the “one global version” approach:

- Safe way to globally provide *different versions of the same library*.
- A vendor bundle can contain *more than one library*.
- manifest.json is a *machine-readable and distributable documentation* of the vendor bundle.
- Works in *all* browsers.

But there are some drawbacks:

- *No on-demand or dynamic loading of vendor assets*. The vendor bundle has to be loaded before the application code that relies on it. The application code does not automatically pull in the vendor bundle it needs.
- *All teams must use Webpack*. The vendor bundle uses Webpack’s internal module loading and referencing code.

NOTE At the time of writing this book, there’s a lot of work going on to improve Webpack’s code sharing abilities across projects. Webpack 5 will introduce a technique called *module federation*⁵ that addresses many micro frontend requirements.

Let’s explore a third option that’s based on JavaScript’s new ES Modules standard.

CENTRAL ES MODULES (ROLLUP.JS)

Nowadays, relevant browsers (except Internet Explorer 11)⁶ support JavaScript’s native modules system with the `import/export` syntax. This opens up new possibilities for sharing dependencies without needing a specific bundler.

⁵ See Zack Jackson, “Webpack 5 Module Federation: A game-changer in JavaScript architecture,” *inDepth.dev*, <http://mng.bz/Z285>.

⁶ See <https://caniuse.com/#feat=es6-module>.

TIP You can find the sample code for this in the `20_shared_vendor_rollup_absolute_imports` folder.

Let's take a quick look at the capabilities of the `import` mechanism. The spec calls the dependency string, a *module specifier*. Here is a list of different specifier types:

- Relative path (starts with a dot)

```
import Button from "./Button.js"
```

- Absolute path (starts with a slash)

```
import Button from "/my/project/Button.js"
```

- Bare specifier (simple string)

```
import React from "react"
```

- URL (starts with a protocol)

```
import React from "https://my.cdn/react.js"
```

TIP If you want to learn more about ES modules, I recommend this resource⁷ as a starting point.

In this example, we'll use the last option: the absolute URL. The concept of this example is the same as in the previous Webpack case:

- We have a shared-vendor project that creates versioned bundles containing `react` and `react-dom`. But the files are now standard ES modules.
- We adjust the team projects to reference the vendor bundle by using an absolute URL.

In production, the code that runs in the browser works like this.

Listing 11.9 `shared-vendor/static/react_16.js`

```
export default [...react implementation...];
```

Listing 11.10 `team-decide/static/decide.js`

```
import React from "http://localhost:3000/static/react_16.js";
```

The central React JavaScript file is in ES module format, and the teams point to it via a URL.

You could ship this code without using a bundler at all. For our example, we use `rollup.js`⁸ to ship `react` and `react-dom` in one bundle file and build and optimize our team's code for production. `Rollup.js` recognizes absolute URL dependencies

⁷ *JavaScript for impatient programmers* by Dr. Axel Rauschmayer, https://exploringjs.com/impatient-js/ch_modules.html.

⁸ See <https://rollupjs.org/>.

(`http://..`) and leaves them untouched. This is something that isn't yet possible with Webpack.

We won't go through the full code but will highlight the significant parts. Figure 11.7 shows the folder structure of the sample code.

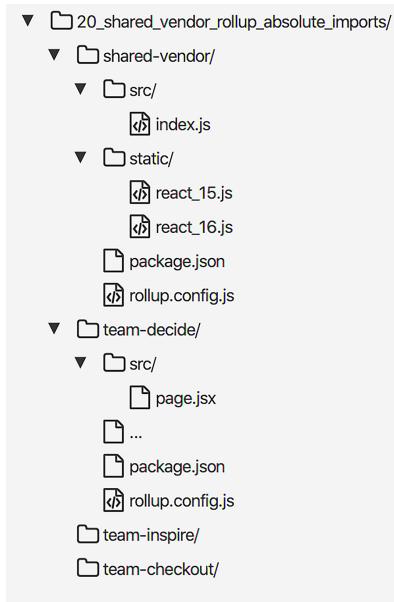


Figure 11.7 The shared-vendor project creates versioned bundles in ES module format using rollup.js. The other teams also use rollup.js

CREATING THE VERSIONED BUNDLE

Rollup's configuration is straightforward. We define input and advise it to write the bundle as an ES module (esm) to the static/ folder.

Listing 11.11 shared-vendor/rollup.config.js

```

...
export default {
  input: "src/index.js",
  output: {
    file: "static/react_16.js",
    format: "esm"
  },
  plugins: [...]
};
  
```

Input file specifies what should go into the vendor bundle

Output defines the target location of the bundle and sets its format

The `src/index.js` imports `react` and `react-dom` and exposes them as both a default and named export. This way, Rollup will create one bundle which contains both libraries.

Listing 11.12 shared-vendor/src/index.js

```
export { default } from "react";
export { default as ReactDOM } from "react-dom";
```

That's everything we need to do to create the vendor bundle. As with the earlier example, the generated file will be available at `http://localhost:3000/static/react_16.js`. Let's look at how we configure the team's React applications to use this bundle.

USING THE VERSIONED BUNDLE

The team's rollup configuration is basically the same as the one we saw before: configuring input, output, and setting the format. It includes a few plugins to deal with JSX, Babel, and CSS, but these are all straight from the official documentation.

Listing 11.13 shared-vendor/src/index.js

```
export default {
  input: "src/page.jsx",
  output: {
    file: "static/decide.js",
    format: "esm"
  },
  plugins: [...]
};
```

Let's have a look inside the input file `src/page.jsx`. To use the globally provided vendor bundle, we need to set our imports accordingly. In a traditional React application, you would use a *bare specifier* like this:

```
import React from "react"
```

The bundler then searches for `react` in your `node_modules` and includes it. In our case, we can specify the absolute URL:

```
import React from "http://localhost:3000/static/react_16.js";
```

Rollup.js will treat this as an external resource. Since all components in a React application need to import `react`, it's a little cumbersome to always write the absolute URL to the versioned file. In the example, I've used Rollup's alias feature⁹ to configure this in a central place. This way, the application code can stay as is, and Rollup replaces all instances of `react` with the absolute URL on build.

The absolute URL approach has two significant benefits:

- 1** *It's standards-based.* Asset sharing is an architecture decision that affects all teams. Changing it later on in the project will produce a non-trivial amount of work. Relying on standards makes future changes in tooling or libraries much more manageable. Want to switch your bundler? No problem, as long as it supports ES modules.

⁹ See <http://mng.bz/RAYD>.

- 2 Dynamic loading of required vendor bundles.** The DllPlugin requires you to load the vendor files before the application code synchronously. With ES modules, the application code requests the vendor bundle(s) it needs. If it's already downloaded because another micro frontend requested the same module, it reuses the existing one.

The dynamic loading makes the integration code a lot simpler. Here is Team Decide's HTML file.

Listing 11.14 team-decide/index.html

```
<html>
  ...
<body>
  <decide-product-page></decide-product-page>
  <script src="http://localhost:3001/static/
    decide.js" type="module" async></script>
  <script src="http://localhost:3002/static/
    inspire.js" type="module" async></script>
  <script src="http://localhost:3003/static/
    checkout.js" type="module" async></script>
</body>
</html>
```

The HTML must only reference the JavaScript files from the teams. They download central bundles when needed.



Start the example locally by running `npm run 20_shared_vendor_rollup_absolute_imports`. In the first view, it looks exactly like the previous example. Two teams use React 16. One team is still on React 15. Opening up the developer tools shows a difference. In the Network tab, you see that the three application bundles load first (small

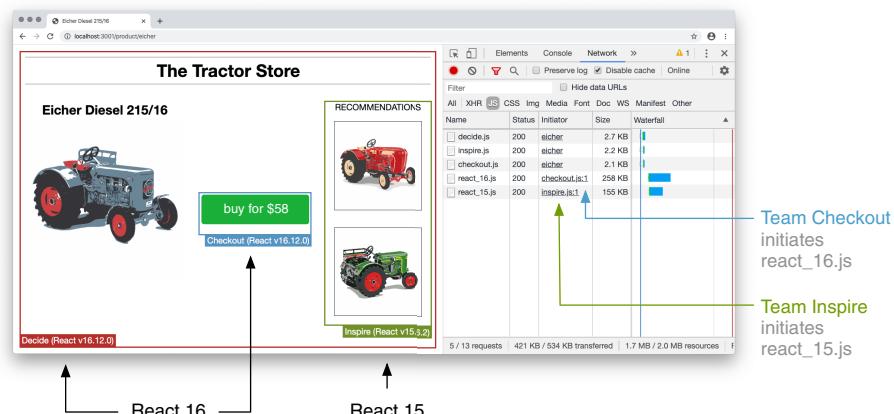


Figure 11.8 Different framework versions on one page by using ES modules. The Network tab shows which team initiated the download of a specific vendor bundle. Team Decide and Team Checkout both reference `react_16.js`. Team Checkout was the first to request it. Team Inspire references the `react_15.js` bundle.

parallel downloads). Then they request their associated vendor bundle (large parallel downloads). You can see this in figure 11.8. The *Initiator* column shows the team which first requested the bundle.

IMPORT-MAPS

In the previous example, we used Rollup's alias plugin to make our life easier. It saved us the hassle of using an absolute URL in all files that require `react`. Let's look at import-maps. Import-maps is a proposed web standard¹⁰ that can simplify our loading process even further. It provides a declarative way to map bare specifiers to absolute URLs. An import-map looks like this:

```
<script type="importmap">
  {
    "imports": {
      "vue": "https://my.cdn/vue@2.6.10/vue.js",
      "vue@next": "https://my.cdn/vue@3.0.0-beta/vue.js"
    }
  }
</script>
```

The definitions from the import-map apply globally. Teams can reference the current version of `Vue.js` by importing `vue`. They don't need to know the URL of the shared bundle. The following example illustrates that:

```
<!-- Team A -->
<script type="module">
  import Vue from "vue";
  console.log(Vue.version);
  // -> 2.6.10
</script>

<!-- Team B -->
<script type="module">
  import Vue from "vue@next";
  console.log(Vue.version);
  // -> 3.0.0-beta
</script>
```

MORE ABOUT IMPORT-MAPS

Import-maps are a promising solution but not an official standard yet. Right now, the preceding code only works in Chrome when you've activated a feature flag.

If you want to use them today, I recommend having a look at `SystemJS`.¹¹ SystemJS maintainer and single-spa developer Joel Denning has published a video series¹² on using import-maps and SystemJS with micro frontends.

¹⁰ See <https://github.com/WICG/import-maps>.

¹¹ See <http://mng.bz/2X89>.

¹² See “What are Microfrontends?” <http://mng.bz/1zWy>.

Podium developer Trygve Li has written an introduction to using import-maps in a micro frontend context.¹³ He also authored a rollup plugin¹⁴ that works similarly to our alias approach but takes an import-map as an input.

11.2.5 Don't share business code

Extracting large pieces of vendor code is a powerful technique. You've learned a couple of ways to achieve it. But you should be careful of what you extract.

It's tempting to share snippets of code every team uses, like currency formatting, debugging functions, or API clients. But since this is business code and has a tendency to change over time, you should avoid that.

Having a similar piece of code in the codebase of multiple teams feels wasteful. However, sharing code creates coupling that you shouldn't underestimate. Someone has to be responsible for maintaining it. Changes to shared code have to be well thought-out and appropriately documented. Don't be afraid of copying and pasting snippets of code from other teams. It can save you a lot of hassle.

If you're confident that it's a good idea to share a specific piece of code with other teams, you should instead do it as an NPM package that teams include at build time. Try to avoid runtime dependencies. They increase complexity and make your application harder to test.

In the next chapter, we'll talk about code that's often shared in micro frontends projects: the design system.

Summary

- Performance budgets are an excellent tool to foster performance discussions on a regular basis. They also form a shared baseline that all team members can agree upon.
- Having some project-wide performance targets is valuable. If teams want to optimize further, they might pick different metrics because they work on different use cases. The performance requirements for the homepage are not the same as the requirements for the checkout process.
- Measuring performance is tricky when micro frontends from multiple teams exist on one site. Having clear responsibilities helps. The owner of a page can also be responsible for the overall page performance. If another team's micro frontend slows down the page, the page owner informs that team to fix the issue.
- It makes sense to measure the performance characteristics of a micro frontend in isolation to detect regressions and anomalies.
- Micro frontend teams have a narrower scope that they are responsible for. This makes it easier for them to optimize performance in the places where it has the most significant effect on the user.

¹³ See <http://mng.bz/PApg>.

¹⁴ See <http://mng.bz/JyXP>.

- The size of your JavaScript framework and the number of teams on a page have an impact on performance. Because teams have a smaller scope, it might be a viable solution to pick a lighter framework. This eliminates the need for vendor code centralization.
- You can improve performance by extracting large libraries from the team's application bundles and serving them from a central place.
- Sharing assets introduces extra complexity and requires maintenance.
- You should measure the real impact of redundant JavaScript code for your use case and target audience.
- Forcing all teams to run the same version of a framework can become complicated for major version upgrades. Teams have to deploy in lock-step to avoid breaking the page.
- Allowing teams to upgrade dependencies at their own pace is an important feature and can save a lot of discussion. You can achieve this by implementing versioned asset files that can work side-by-side. Use Webpack's DllPlugin or native ES modules to implement this.
- Only centralize generic vendor code. Sharing business code introduces coupling, reduces autonomy, and can lead to problems in the future.

12

User interface and design system

This chapter covers

- Examining how a design system can help deliver a consistent experience to your users
- Developing a design system and how it can affect the autonomy of the micro frontends teams
- Technical challenges when building a pattern library that should be technology-agnostic
- Distinguishing if a component should go into the central pattern library or stay under a product team's control

In a micro frontend architecture, every team builds its part of the frontend. A team can plan, build, and ship new features without talking to its neighbors. But how do you deliver a consistent look and feel for the user? The different frontends should use the same color palette, typography, and grid layout. These measures ensure that the website does not look weird. But it typically doesn't stop there. There's also button styling, spacing rules, breakpoint definitions to support a variety of screen sizes, and a lot more.

Classical architecture discussions often dismiss these topics as unimportant. You hear sentences like, “We’ll find a way to make it pretty afterward.” However, in a distributed architecture like this, it’s essential to have a proper plan for managing your design from the start. Throughout the book, you’ve learned techniques to avoid sharing code and keep teams as decoupled as possible. When it comes to design, it’s not that easy. If you don’t want to alienate your users, you need *a system to share your design building blocks with all teams*. A design system enables them to build interfaces that have a similar look. However, a design system also introduces coupling because every team has to be compatible with it.

In the micro frontends projects I’ve worked on, planning and setting up a shared design system was always among the first and most important tasks. How to integrate a design system into the team’s code is a much-discussed topic. It has direct implications on how teams build their frontend features. Changing these architecture decisions afterward is expensive because all user-facing features rely upon it.

In this chapter, we’ll briefly introduce the concept of a design system, discuss how to organize effective development, and look at a variety of technical integration options and their trade-offs.

12.1 Why a design system?

Creating an overarching design that different teams can use is far from specific to micro frontends. The term *design system* has become popular in software development in recent years. It provides a way to systematically tackle design in an era of growing web applications that must work on a multitude of devices.

A design system contains *design tokens* (fonts, colors, icons …), *reusable interface components* (buttons, form elements …), more *advanced patterns* (tooltips, layers …), and most importantly a *well-explained set of rules* on how to use these individual pieces together. Figure 12.1 shows some design system examples.¹

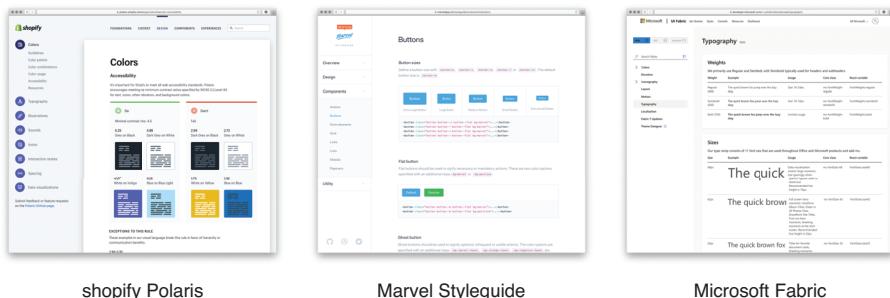


Figure 12.1 A lot of companies have published their design systems on the web. You can use them in your project or leverage them as a source of inspiration when creating your own.

¹ See <https://designsystemsrepo.com/design-systems/>.

Two other terms often also come up in this context: *pattern library* and *(living) style guide*. They mean the same thing: a way to modularize the complexity of the web with a component-based system. However, they have a slightly different focus.

The term *pattern library* describes a set of concrete building blocks developers can use. It is a library that contains tangible components like buttons and form inputs. It focuses more on the components than on the documentation aspect. You can say that a pattern library is a subset of a design system.

Style guide is a traditional term from the design world. Before the internet, a style guide in the form of a well-crafted stack of paper describing all design rules for a company's corporate identity. The "living" prefix transferred this concept to the digital age, where the illustrated components use the real code. In this chapter, we'll use the term *design system* when we talk about the broader concept and use *pattern library* when it comes to the technical integration with the team's applications.

In this book, we won't discuss how to build a design system. You can find a lot of excellent blog posts,² books,³ and even hands-on checklists⁴ to get deeper into this topic. Instead, we'll focus on the design system aspects that are crucial to get right for running a successful micro frontends architecture.

12.1.1 Purpose and role

In a micro frontends project, all features the product teams create are directly targeted to the end user. These features make the user's life more enjoyable and thus create value for the company. A centralized design system does not fit into this model.

No user signs up for Microsoft Office 365 because they think that Microsoft's Fluent UI Design System is the best. But there's no question that the existence of the design system makes Office a more usable product. People who are familiar with using Word have an easier time understanding PowerPoint or Excel because all teams use the same UI paradigms and components.

A design system has an indirect effect that manifests itself through the product teams. The goal of a design system team should never be to create the most beautiful, best documented, or most versatile design system on the market. The objective of a design system team should be supporting the product teams as best as possible. *A design system is a product that serves other products.*

12.1.2 Benefits

A sound design system can help product development by providing these benefits:

- *Consistency*—Making user interfaces from different teams "feel familiar" to the user.

² See Vitaly Friedman, "Taking The Pattern Library To The Next Level," *Smashing Magazine*, <http://mng.bz/wB7W>.

³ See *Design Systems*, by Alla Kholmatova, <http://mng.bz/qM7E>.

⁴ See <https://designsystemchecklist.com>.

- *Shared language*—A design system forces you to create a shared vocabulary that all involved parties understand. Proper naming is never easy, but having consistent names for your components and patterns improves communication across teams and avoids misunderstandings.
- *Development speed*—Having clear guidance and the necessary UI components to build a new feature makes the developer's life easier.
- *Scaling*—The value of a design system increases by the number of teams using it. New teams have a solid foundation they can build upon. No redundant discussions on "if we should use a custom select box or not." Hopefully, the authors of the design system have documented this decision before.

The benefits of a design system are mid- and long-term. Creating a robust system will take a considerable amount of time. However, if your project is of a particular size, these efforts will pay off quickly. It will also save you a lot of unsatisfactory design consolidations and eliminate chaotic redesign projects.

12.2 Central design system versus autonomous teams

Now you know the basics of a design system and its benefits. Let's look at some aspects that are important in a micro frontends architecture. One question that's frequently asked is if it's indispensable to build your own design system.

12.2.1 Do I need my own design system?

Creating a design system is not an easy or cheap task. If you are building an internal product where branding is not an important aspect, it's perfectly fine to go with an off-the-shelf solution. Projects like Twitter Bootstrap,⁵ Google's Material Design,⁶ Semantic UI,⁷ or Blueprint⁸ are great candidates. They all bring a set of generic components developers can adapt for their use case.

But you shouldn't choose a library by its appearance alone. They have different technical architectures that introduce constraints into your project. Some integrate solely via CSS classes (Bootstrap, Semantic UI), dictate a specific frontend framework (Blueprint), or provide a set of framework options (Material Design). Later in this chapter, we'll dive deeper into the possible integrations and their pros and cons.

If your product should convey a unique style and must be in line with your companies branding, it's a good idea to develop your own design system from scratch. Such a system also enables you to incorporate components that are unique to your business domain. In e-commerce, you want to have a price component that defines how reductions, sales, or the base price should render. When you are building a messaging application, you will want to include primitives like user avatars or chat bubbles.

⁵ See <https://getbootstrap.com>.

⁶ See <https://material.io/develop/web/>.

⁷ See <https://semantic-ui.com>.

⁸ See <https://blueprintjs.com>.

12.2.2 Process, not project

Having your own design system has some real benefits. We, as developers, like to focus on its technical aspects. Creating a set of usable components for all teams sounds like a worthwhile project. But in an otherwise distributed organizational structure, a design system also introduces an important social aspect. A former co-worker of mine likes to describe the design system as...

... the campfire around which people from different teams and with different professions gather regularly.

Dennis Reimann

This quote highlights the fact that a design system is never a finished product. It's better to think of it as a process. A design system should be a living and evolving piece of infrastructure. The usable components and formalized design rules are the result of discussions between user experience (UX) and design experts as well as developers and product owners from the teams. It should be the single source of truth when it comes to design questions. Figure 12.2 illustrates this.

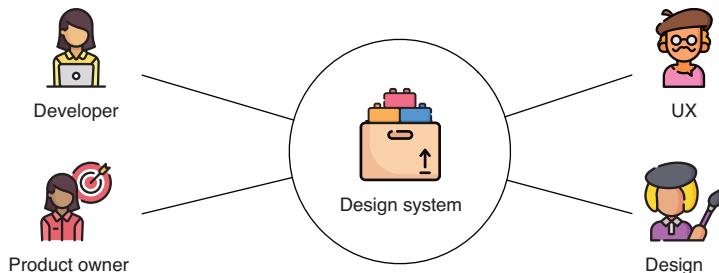


Figure 12.2 A good design system is a place where all key design decisions get documented. It's constantly refined to best meet the needs of its users.

12.2.3 Ensure sustained budget and responsibility

It's important to set appropriate expectations in management. The bulk of the design system work will be in the first few months, but the work doesn't stop then. New use cases arise, and teams develop new and more sophisticated features. You need free space to adjust and grow the design system accordingly. It's crucial to have a sustainable budget dedicated to doing this work:

- Extending components
- Questioning existing patterns
- Refactoring areas
- Refining the documentation
- Fixing inconsistencies

I've seen projects with thoroughly crafted design systems that worked pretty well in the beginning. But when there is nobody who feels responsible or can maintain and evolve the system, it starts getting out of date. Teams work around existing patterns. They modify components with custom override styles to adjust them to their needs. Some components get extended several times and grow in complexity. Documentation becomes out of date.

From this point, it usually gets worse pretty quickly. That's what the community calls a *zombie style guide*.⁹ Don't let your design system join the zombie army. Rebuilding and replacing a design system is expensive. The micro frontends architecture optimizes for feature development speed inside team boundaries (vertical). Introducing substantial changes across teams (horizontal) requires a lot of coordination, creates friction, and can impair development for a considerable time.

Make sure to establish proper conditions in the first place. Having a dedicated budget and strong responsibility is vital.

12.2.4 Get buy-in from the teams

Getting the green light from management is an essential precondition, but it's even more important to have a healthy relationship with the product teams. They are the users of your design system. They are your customers. Take time to explain the design system and its concepts to them.

THE FIRST SPRINTS

Learn about their development roadmap and discuss wireframes to identify the components that are required first. A transparent development process helps the product teams to know when needed parts are ready. Publishing documentation, examples, and changelogs supports this.

In the early stages of a new project, the design system team is usually the bottleneck. There's a lot of technical setup to do. The team needs to build essential components for typography and interactions. Giving the design system team a head-start of a couple weeks is something we've had good experiences with. This way, the product teams can use the pattern library from the start. Nobody needs to wait or use temporary solutions that cause trouble later on.

ACCEPTANCE

Even though all teams know about the benefits of the design system, it's often tempting to work around it. Imagine Team Decide wants to ship a new product review feature. To build it, they need a new rating-star icon and a new, smaller heading style. The team is already under pressure because the lead developer broke their arm in a sporting accident a week ago. To keep the schedule, it would save time to add the icon directly to Team Decide's application code. They could take the standard heading and

⁹ Tweet by @jina: "zombie style guides — style guides that aren't maintained and part of your process. they die and rot. they eat your brains," <https://twitter.com/jina/status/638850299172667392>.

just overwrite it with a smaller font size. Yes, other teams wouldn't be able to use these components in their features. "But that's not important right now." These moves would save Team Decide time. Bringing the change into the central pattern library requires more work than building it locally.

The main benefit of micro frontends is to empower teams to move fast by eliminating dependencies and waiting for other people. A central design system will get in the way. Having discussions around reusability and consistency doesn't help the product team's primary mission. Make sure everyone understands this conflict of interest and recognizes the importance of the design system. Find a way to spot technical debt, and don't let it build up.

COMMUNICATION

Establishing proper communication channels between the design system and the product teams is a crucial factor for success. There are a lot of ways to do this. It doesn't have to be regular in-person meetings. Being creative and coming up with lightweight solutions makes the process leaner and can build up acceptance.

We've experimented with a concept called *opening hours*. The design team offers dedicated time slots. Product teams can come in and discuss wireframes for upcoming features. They don't need to schedule a meeting. The goal is to identify changes for the design system in an early phase.

However, the method we found most effective is to directly involve people from all teams in the development process itself. Next up, we'll see how this can work.

12.2.5 Development process: Central versus federated

There's no single way to organize the development of a design system. Up until now, we implicitly talked about an organizational form that's called the *central model*. We have a dedicated team that plans and builds the design system and distributes it to the product teams to use. But there's another approach that's gaining popularity and fits well into our autonomous-teams architecture: the *federated model*.¹⁰ Figure 12.3 shows both models side by side.

THE CENTRAL MODEL

In the central model, we have a clear division of labor. A group of developers, designers, and UX specialists plan and build the design system. To know what they should build, they talk to the product teams. The design team has a pretty good overview of the complete system, can spot inconsistencies quickly, and works efficiently.

The product teams are only users of the pattern library. They make requests to the design system team and wait until their components are ready. The central team has the potential to become a bottleneck. When product teams request more changes than the design team can implement, it gets ugly. Teams have to delay their schedule or start working around the design system.

¹⁰ See Nathan Curtis, "Team Models for Scaling a Design System," *Medium*, <http://mng.bz/7XBg>.

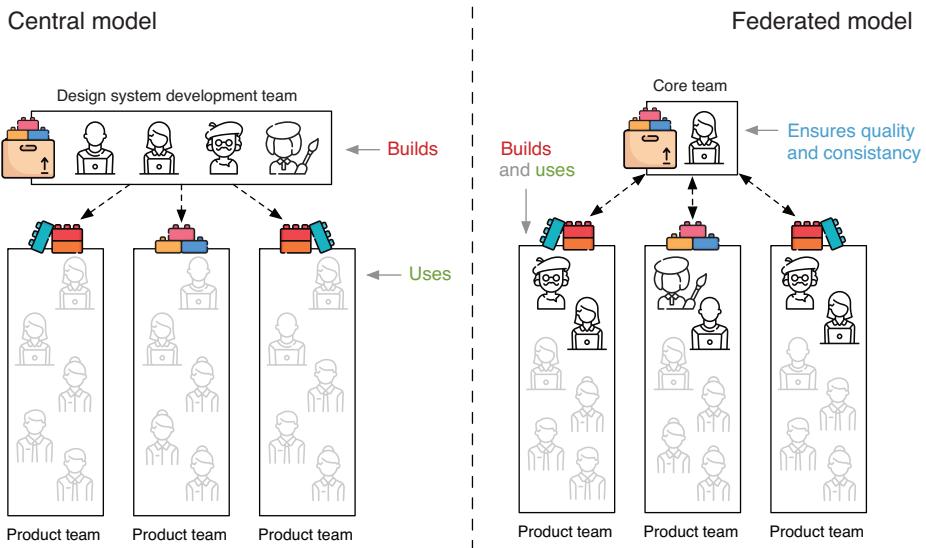


Figure 12.3 Two approaches for organizing the development of a design system. In the central model, a dedicated design team develops the system, and the teams use it. The federated model blurs the line between the design system and the product team. The members of the product teams contribute to the system and drive development.

THE FEDERATED MODEL

The federated model changes this. Designers and UX specialists move into the product teams. There's no real central team anymore. Yes, we still need someone who stewards the design system and has an eye on quality and consistency. However, the product teams now drive the development of the design system themselves. When a product team needs a new component, they design it, build it, and publish it to the design system for everyone to use.

This model gives the team a lot more freedom and autonomy. But since the design system is a shared project, it's crucial to properly communicate changes to others. Running this model requires some skill and experience. Its most significant benefit is that UX experts and designers now work in the product teams. They bring new perspectives to the development team and can help to improve the product directly. This quote from Nathan Curtis¹¹ brings it to the point:

We need our best designers on our most important products to work out what the system is and spread it out to everyone else. Without quitting their day jobs on product teams.

Nathan Curtis

¹¹ A series of blog posts on design systems. It's a goldmine. You should read them all. :) <https://medium.com/@nathanacurtis>

12.2.6 Development phases

You might ask yourself what's the best model for your project. It's hard to give a general answer to this question. But I'll share what worked for us.

First of all, the two models aren't mutually exclusive. They blend very well. You don't have to pick one of the extremes. Running a design system with a strong central team does not mean that you can't take contributions from a product team. Figure 12.4 shows the Central-to-Federated Continuum.

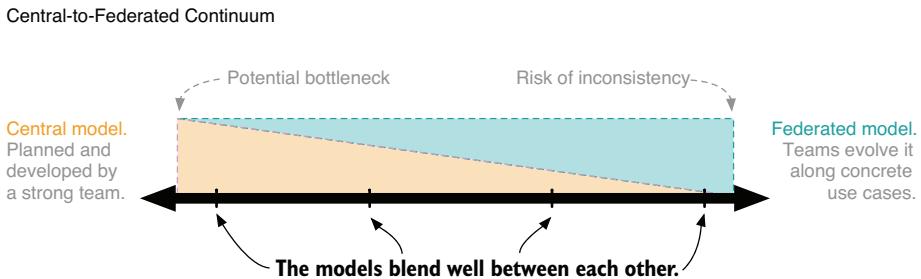


Figure 12.4 Central vs. federated is not a binary decision. The models work well together. The scale at the bottom shows the spectrum. You can run the central model with some federated aspects (left side). It's also possible to run the federated model in combination with some central planning and development (right side).

In our projects, I could observe two phases of design system development: the ramp-up phase (phase 1) and the production phase (phase 2). Figure 12.5 shows how these phases vary in focus.

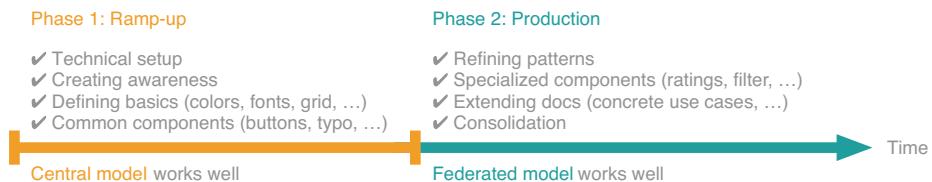


Figure 12.5 What model fits best can depend on the development phase your project is in.

When starting a new project, we've had good experiences with the central model. It's an efficient way to get a new design system off the ground. In this ramp-up phase, there's a lot of work to do—setting up pipelines and tools, making initial decisions, and creating the first set of standard components. Having a dedicated team that has no other responsibilities is valuable in this phase.

When the dust has settled, and teams start to get productive, we slowly move toward the federated model. This way, we ensure that the real use cases drive the development. We encourage frontend developers from the product teams to learn about the design system and contribute. Developers and designers from the design system team move toward the product teams. In this transition phase, it's common for people to divide their time between two teams. A designer might spend 50% of their time on the design system and 50% on the product team. These percentages make planning easier. They can gradually shift over time.

12.3 Runtime versus build-time integration

You've learned a lot about the organizational aspects. Let's see how we can technically integrate a pattern library with the team's applications. First, we'll talk about different strategies for rolling out changes.

Imagine you've changed the color of your button component in the central pattern library. What needs to happen so that the user can see it?

You can find two deployment approaches that people use: *runtime integration* and distribution as *versioned packages*. Figure 12.6 shows both of them side by side.

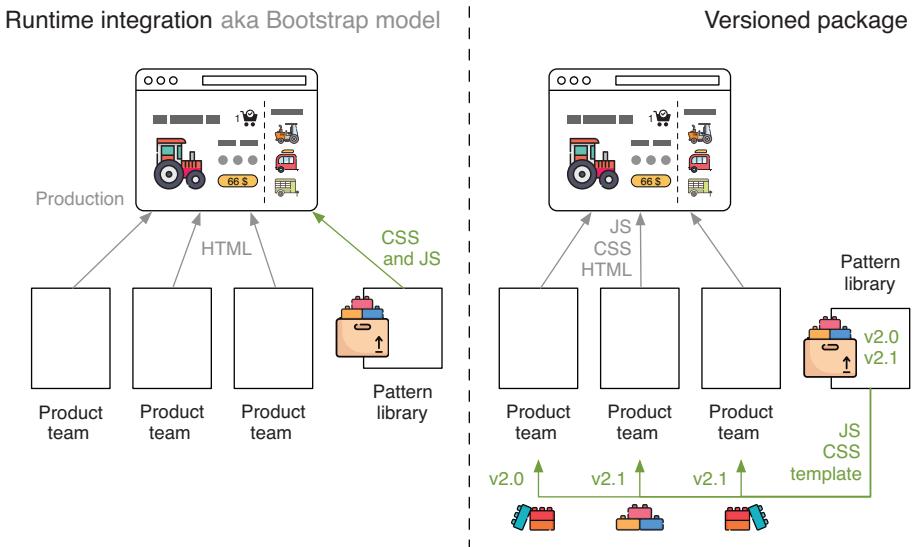


Figure 12.6 In the Bootstrap model (left), the pattern library deploys its artifacts (JS, CSS, and possibly images) directly to production. Changes are instantly visible and distributed across teams. With versioned packages (right), the pattern library offers the components as a package (for example, NPM) that teams can pull into their application. Teams control when to update to the latest version.

12.3.1 Runtime integration

Twitter Bootstrap is the most famous example of a *runtime integration*. The concept is simple. Teams embed a link to a global CSS file that's maintained by the design system team. They can style their page by applying CSS classes to the markup. The same goes

for the micro frontends embedded on that page. The CSS classes are globally available. Here's a code sample that shows how to embed and use global styles.

Listing 12.1 /team-decide/product/porsche.html

```
<link rel="stylesheet" href="/shared/pattern-library.css"> ← Integrating the pattern library styles  
<button class="btn btn-call-to-action">Buy a tractor</button> ← Using the styles via CSS classes
```

The runtime model is not exclusive to pure styling. If you use client-side rendering, it's also possible to provide components that encapsulate styling and internal markup. Here's an example of doing it via Web Components.

Listing 12.2 /team-decide/product/porsche.html

```
<script src="/shared/pattern-library.js"></script>  
<tractor-store-price reduction="10%" value="$66"> ← Integrating the pattern library script which contains Web Component definitions  
Using the price component as markup. It renders the appropriate styled markup inside its ShadowDOM.
```

Setting up a pattern library using a runtime integration is pretty straightforward. It's simple to develop and easy to use. Another benefit is that the design system team can roll out changes instantly.

However, this model has some considerable coupling and autonomy disadvantages:

- *Testing in isolation*—Micro frontends should be self-contained. With runtime integration, a team's user interface doesn't work in isolation. For it to function, it's necessary to include the pattern library's styles and scripts. Since the design system team can change these assets at any time, a product team can't ensure that its user-interface looks right and is working correctly. They would have to run their automated test-suite on every pattern library change.
- *Single point of failure*—With runtime integration, the pattern library becomes a mission-critical part of the system. An error that slips through can bring down the complete project, since all teams rely on it.
- *No tree shaking or deprecations*—Since the design system team cannot know which components a specific page uses, it's common practice to include the styling code for all of them in one big CSS file. This file tends to only grow in size because there's no safe way to ensure that an old component is unused. When you go for the JavaScript components option, you can at least use on-demand loading strategies to avoid loading unneeded script code.
- *Breaking changes*—There's no structured way to handle breaking changes. If the team wants to refactor the button component in a significant way, they need to

create a new one (for example, `.btn_v2`) and delete the old one when everyone has updated their markup.

- *Versioning and scoping*—It's hard to establish proper versioning in this model. There's also no easy way to prevent leaking styles between different micro frontends.

The lack of versioning is particularly critical. It means that all teams must use the latest version of the pattern library. You can't restructure or upgrade the pattern library in a meaningful way. It would require close coordination and simultaneous deployments from all teams. The fear of introducing friction incentivizes the design system team to shy away from making the necessary steps forward. Let's look at a more flexible model to distribute a pattern library.

12.3.2 Versioned package

In the versioned model, the pattern library is not a runtime system. Instead, the design team distributes it as a package that contains all components. The LEGO™ metaphor works well here. You can think of it as a big box of bricks. The product teams can grab one of these boxes and take the required bricks out of it. Together with their own special bricks, they can build features for the customer.

Listing 12.3 /team-decide/static/product.jsx

```
import { Price, Button } from "@the-tractor-store/pattern-library";           ←

function ProductPage() {
  return <div>
    <Price reduction="10%" value="$66" />
    <Button type="call-to-action">Buy a tractor</Button>
    ...
  </div>;
}
```

Importing the required components
from the pattern library package

Using the components to
build the product page

INDEPENDENT UPGRADES

The design system team can iterate on the pattern library. They produce new revisions of the LEGO™ box regularly. An updated revision might include new kinds of bricks or an updated surface finish. But teams don't have to upgrade instantly. They can upgrade at their own pace. An older revision might not look as sweet as the new one, but it still works fine.

DON'T SHIP UNUSED CODE

With this approach, each team generates its own CSS file. A bundler like Webpack includes only the pattern library components that a team uses. So if the pattern library still includes an old component, but no team requires it, the browser won't have to download its code. This mechanism leads to quite small CSS files.

SELF-CONTAINED

You can instruct your bundler to prefix all CSS classes automatically. This way, it's possible to achieve proper scoping, and a page can contain micro frontends that use different versions of the pattern library.

Let's look at an example. Imagine Team Decide owns the product page, which displays a price and a button. It also includes a micro frontend from Team Inspire, which also shows a button:

- 1 Team Decide uses pattern library version 4 in their applications.
- 2 The design system team releases a new iteration (v5) in which the buttons have a new and rounder style.
- 3 Team Inspire immediately upgrades to this version and deploys its application.
- 4 Team Decide has other work to do. They'll update tomorrow.

Here is what the generated production code might look like.

Listing 12.4 /team-decide/dist/product.css

```
/* based on pattern library v4 */
.decide_price {...}
.decide_button { border-radius: 2px; }           ← The old button styling
.decide [...] {}                                from pattern library v4
```

Listing 12.5 /team-inspire/dist/reco.css

```
/* based on pattern library v5 */
.inspire_button { border-radius: 10px; }           ← The new more rounded button
.inspire [...] {}                                styling from pattern library v5
```

Listing 12.6 https://the-tractor.store/product/porsche

```
<div>
  <span class="decide_price">only $66 (10% off)</span>
  <button class="decide_button">Buy a tractor</button>           ← Team Decide's
  <aside>                                         button references
    <button class="inspire_button">Show recommendations</button>   its own CSS class.
  </aside>
</div>                                              ← Team Inspire's button also
                                                       references its own CSS class.
```

The two buttons on the page have different appearances. Team Inspire's button is already on the new rounder style, whereas Team Decide's button is still on the old style. Being able to use different versions side by side is an essential step for independent deployments. With this approach, a micro frontend is fully self-contained and doesn't rely on styles from other teams. The product teams are in control of upgrading their pattern library and testing the changes before they deploy them.

THE DRAWBACKS

This approach has a lot of advantages compared to the runtime integration. But it also has some drawbacks:

- *Redundancy*—When teams use the same component, the user has to download the associated code multiple times. You can see this in the preceding example. We have two versions of the button styling. This redundancy is typically not a big problem. Since the bundler only includes components that are in use, and no team uses all components at once, the total CSS file size is usually much smaller compared to the global Bootstrap model.
- *Slower rollouts*—Changes in the pattern library take longer to be visible in production. The design system team cannot push new updates. They can provide a new version and inform all the teams. The changes are only visible when all teams have updated and deployed their application. It might be necessary to encourage teams to deploy faster to rollout a critical design system bugfix quickly.
- *Eventual consistency*—Most graphic designers are not comfortable with the idea that a page can contain different versions of the same component. However, when teams update on a regular schedule, this is not a pressing issue. Pro tip: In my last project, we created a dashboard that shows which team is using which version of the pattern library. Merely showing this information in an aggregated view leads to faster upgrades by the teams. On another note: Take a look at figure 12.7. It shows different generations of Amazon's buttons. These were all active at the same time in different areas of the site. The fact that Amazon does it should not be an excuse to discard consistency, but having temporal inconsistencies can be perfectly fine.

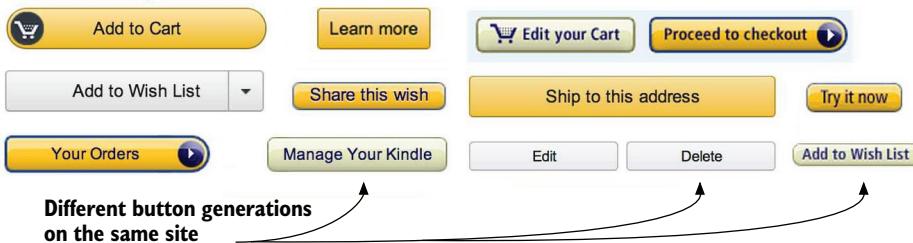


Figure 12.7 Screenshot of different Amazon button styles from different parts of the site

12.4 Pattern library artifacts: Generic versus specific

Now let's take a closer look at the technology of the pattern library. Its output has to be compatible with the technology stack of the teams. There's no gold standard for shipping reusable components. Different options exist, and they all have their benefits and drawbacks. Some don't support server-side rendering, some require a specific JavaScript framework, and others support styling but not templating.

12.4.1 Choose your component format

User interface components consist of three parts:

- 1 *Styling* in the form of CSS code.
- 2 *Templating* to generate the components' internal HTML markup based on the provided inputs. You can execute the templates on the server and/or client, depending on its format.
- 3 *Behavior* (optional) for components the user can interact with, like tooltips or modals. They require client-side JavaScript to work.

Let's explore our options. Look at figure 12.8 and take some time to get an overview. The diagram shows different formats a pattern library can produce.

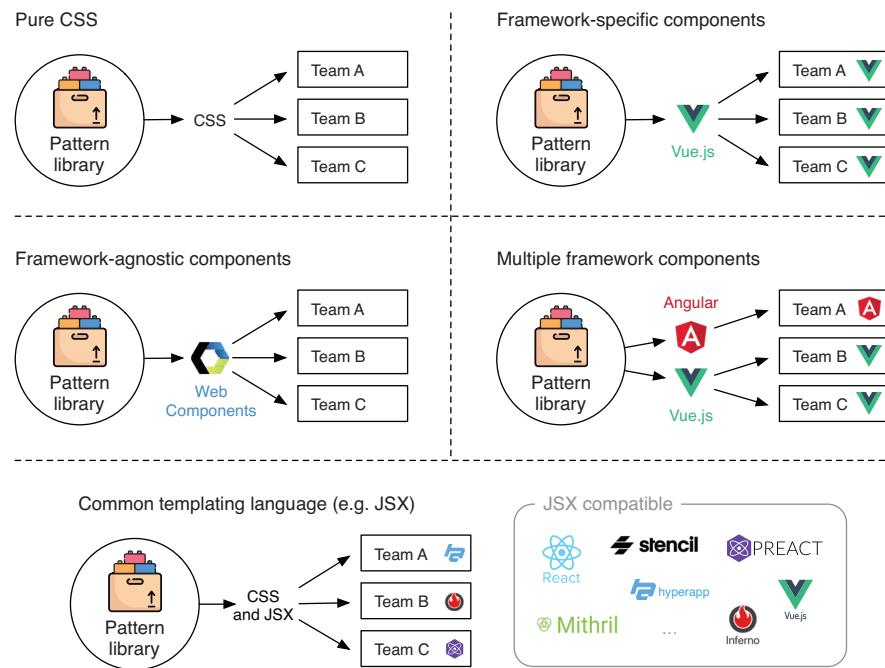


Figure 12.8 Different artifacts a pattern library can produce. Some output formats have technical implications for the team. When the pattern library only exports Vue.js components, all teams need to use Vue.js to be compatible.

We'll go through the diagram line by line.

PURE CSS

The pattern library provides its component styling via CSS classes. Twitter Bootstrap is the role model in this category. Teams need to craft the components markup according to the pattern library's documentation:

■ Benefits

- Easy to implement.
- Works server- and client-side.
- Compatible with all tech stacks that can generate HTML.

■ Drawbacks

- Styling only.
- Teams need to know the internal markup.
- Changing the component markup is hard.

FRAMEWORK-SPECIFIC COMPONENTS

The pattern library uses the component format of one specific framework. An open-source example is Vuetify,¹² a component library designed for Vue.js. This model requires all teams to use the chosen JavaScript framework. The component formats of the popular frameworks have been pretty stable—even across major versions. This format stability means that teams have to use the same framework but aren’t required to run the same version:

■ Benefits

- Easy to implement.
- Works server- and client-side.
- Components integrate seamlessly with the team’s code.
- Components can use the full feature-set of the framework.

■ Drawbacks

- All teams must use the same framework.

FRAMEWORK-AGNOSTIC COMPONENTS

Web Components integrate well with all modern frameworks.¹³ You can also use them on plain old HTML pages. Have a look at the Duet Design System¹⁴ as a useful reference. The developers built it using Stencil.¹⁵ In contrast to the “pure CSS” approach, Web Components also encapsulate templating and behavior:

■ Benefits

- Supported by all browsers
- Future-proof (web standard)
- Compatible with plain HTML and frameworks

¹² See <https://vuetifyjs.com/>.

¹³ See <https://custom-elements-everywhere.com/>.

¹⁴ See <https://www.duetds.com/>.

¹⁵ Stencil is a toolchain for building reusable, scalable Design Systems. See <https://stenciljs.com/>.

■ Drawbacks

- Only works client-side¹⁶
- JavaScript required (makes progressive enhancement hard)

MULTIPLE FRAMEWORK COMPONENTS

The model is related to framework-specific components. But instead of supporting one framework, the pattern library exports its components in different formats. Providing more than one format requires extra work because you will need to implement the framework-specific parts multiple times. However, the concepts, component list, and the CSS styling stay the same.

Google's Material Design is a large-scale example of this. The design system itself defines styling, markup documentation, and scripts. Projects like Material UI (React) or Angular Material take the “generic” design system and transform it into a framework-specific format:

■ Benefits

- Works server- and client-side.
- Components integrate seamlessly with the team's code.
- Components can use the full feature set of the framework.

■ Drawbacks

- More work required.

COMMON TEMPLATING LANGUAGE (E.G. JSX)

It doesn't have to be a specific component format. You can also ship HTML templates and styling (for example, via CSS Modules). Many JavaScript libraries and frameworks support the JSX templating format. This way, it's possible to write the HTML template once and use it in a Hyperapp, Inferno, Preact, or React application.

The lifecycle methods and event handling in these frameworks are not the same. This difference means that you can't include behavior. Components have to be stateless. But if your design system mainly includes essential UI components, this is not an issue.

Have a look at X-DASH¹⁷ from *The Financial Times* to see a real world example of this method. We are using the JSX approach in newer projects and are happy with its trade-offs:

■ Benefits

- Works server- and client-side.
- Supports all frameworks compatible with the templating language.

¹⁶ Yes, there are ways to render them on the server. But there's no standardized templating in the current web components spec (<https://github.com/whatwg/html/issues/2254>). That's why all current solutions require a lot of hacking and fiddling.

¹⁷ See <https://financial-times.github.io/x-dash/>.

■ Drawbacks

- You can't include behavior.
- Implementations might vary and speak different "dialects."

NOTE You can use this model with any templating language. But be aware that implementations aren't always 100% compatible with each other. For example, we had significant issues with using handlebar templates across languages like Scala, Python, and JavaScript. Be confident that your model works and its limitations are well understood. Create technical spikes to verify it before you roll it out company-wide.

12.4.2 There will be change

As I said before, there's no clear winner. The right choice for your project or company depends on your needs.

However, if you've made a choice, you should communicate the contract between the pattern library and the teams. Does your integration rely on a framework component format, is it based on a specific DOM structure that's documented somewhere, or do teams need to support a dedicated templating language?

This decision impacts the team's autonomy long-term. Switching to another model later is costly and cumbersome.

BE OPEN FOR CHANGE

A good option for being open to future trends and technologies is to have the "multiple framework components" model in mind from the beginning, even if you decide to start with Vue.js components. If your concepts are stable and you architect your CSS in a reusable way, it will be easier to add new output formats like Web Components, Angular, or Snowcone.js¹⁸ later on.

KEEP IT SIMPLE

Another tip is to keep the central components as dumb as possible. Try to keep the behavioral aspects to a minimum.

Let's take a navigational tree component as an example. It's a vertical list of links you can expand to see its nested links. The pattern library could provide a fully-fledged component which includes functions like expand/collapse and text search, and has hooks for lazy loading subtrees. But getting all of these aspects right and fulfilling every team's needs is challenging.

You could also go the other route and let the pattern library only provide the building blocks and states this tree component can have: expanded/collapsed items, active state, and position of a search box. With this approach, the teams have more work to do because they need to build the mechanics (toggle, search, ...) themselves,

¹⁸ This might be the hot new thing in a couple of years. You never know :)

but they also have much more flexibility. They could decide to pick an open-source tree library that fits their needs and feed it with the styled building blocks from the pattern library. Focusing on the visual aspects makes your pattern library more flexible and reduces feedback loops with the teams.

Finding the right balance between centralized and distributed is not always easy. In the next section, we'll dig a little deeper into this question.

12.5 What goes into the central pattern library?

Having all user interface elements visible and documented in the central pattern library is valuable. The central documentation makes it easy to get an overview. But sharing a component comes with costs.

12.5.1 The costs of sharing components

Changing a component in a team's application code is much easier than changing a component in the central pattern library because central components

- *Live in another project.* You have to publish a new version to see the change in the team's code.
- *Might be used by other teams.* You need to think about the possible consequences of these teams.
- *Must conform to higher quality standards.* You want to ensure that even people from outside your team understand a component's capability and the reasoning behind it.
- *Might require code review.* Depending on your design system development process, you might instantiate a dual control principle to guarantee a high standard.

These aspects make changing a component in the central pattern library much harder than directly changing it in your own code. Putting all components into the pattern library would slow down the development. That's why you need to consciously decide what goes into the central pattern library and what should better be local to a team.

12.5.2 Central or local?

In a lot of cases, the decision whether a component should be *central for all* or *local for one team* is easy to make:

- The definition of the sale color should, of course, be global. The same goes for an icon set or the styling of an input field.
- Advanced patterns like a payment options box or the concrete layout of the product page should be controlled by the respective teams.

But there's a middle-ground where these decisions are not that clear. Is the filter navigation or a product tile a central component? Let's look at some vectors that help you make your decisions.

COMPONENT COMPLEXITY

The atomic design methodology¹⁹ is quite popular. It uses the chemistry metaphor of atoms, molecules, and organisms to sort components by their complexity. This metaphor also highlights the fact that larger components are a composition of smaller ones. Figure 12.9 shows the atomic design categories from lowest complexity (design tokens) to highest complexity (features and pages).



Figure 12.9 The atomic design methodology organizes the design system by complexity. The central pattern library should include the basic building blocks (tokens, atoms, molecules). More sophisticated components (organisms, features, entire pages) should be under a team's control. The middle ground around molecules and organisms is fuzzy.

This scale maps well to our central vs. local question. A good rule of thumb is to share simple components and put complex ones under team control.

But this model is fuzzy in the middle of the scale. Developers like to argue about whether a particular component is a molecule or an organism, but these discussions are almost always theoretical and fruitless. Comparing code complexity is not the only important factor.

REUSE VALUE

The reusability of a component is a reliable indicator. Components that different teams need might go into the central pattern library, even if they are not simple. Patterns like accordions or carousels are good examples here.

However, you should be careful with this rule. The focus of larger components might change over time. Here is an example:

Team Inspire uses the product tile component for their recommendations. Team Decide has built a wishlist feature. They use the same product tile on their wishlist page. The central component worked great at the start, but over time both teams come up with conflicting requirements. Team Inspire wants the component to be more compact to fit more tiles in a recommendation slot. Team Decide needs to add more functions and product details to it. Moving the component out of the central pattern library and letting each team work on their own version of it might be an option. Another alternative can be to reduce the product tile component to its essence. The new component could provide dedicated slots where teams can add functionality if they need to.

These conflicts are natural because nobody can foresee the future. It's essential to reevaluate your decisions regularly and be open to revising them.

¹⁹ See Brad Frost, “Extending Atomic Design,” *Brad Frost*, <https://bradfrost.com/blog/post/extending-atomic-design/>.

DOMAIN SPECIFIC

Domain-specific components are good candidates to be team-owned. To identify this, you can ask the question: “*Which team has the interest in changing this component, and why?*” When there’s a team that frequently updates a component to improve its business, it’s a reliable indicator that this component should be local to that team.

A filter navigation is a good example here. At first sight, a list of filters looks pretty simple. However, when you have a product team with the mission to “make finding products easier,” this team will want to change this component frequently. They want to test different variants, collect feedback, and improve the component. Making this team jump through hoops by centralizing the component will slow them down and block innovation.

TRUST IN TEAMS

These three properties (complexity, reusability, and being domain-specific) can give you a good idea of where a component should live. Don’t be afraid to give up central control and let teams own and evolve specific components.

But it’s not just about control. If a component is not part of the global pattern library, it’s hard for designers to keep an overview. Next up, we’ll look at the concept of *local pattern libraries* to mitigate this.

12.5.3 Central and local pattern libraries

It’s not a hard rule that you must have a single design system. There’s the concept of *tiered design systems*²⁰ that perfectly fits into our micro frontend architecture. The idea is to have a central pattern library that defines the basics, and other pattern libraries that build on it and add their use case-specific components. In our case, each team can have its own local pattern library. Figure 12.10 illustrates this.

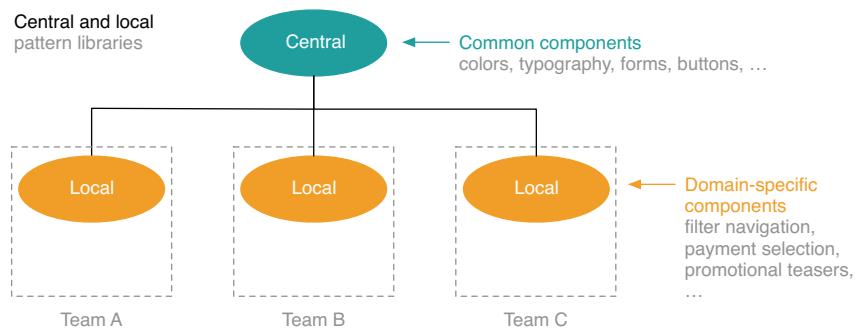


Figure 12.10 A two-tiered pattern library approach. Each micro frontend team has its own local pattern library where it develops its domain-specific components.

²⁰ See Nathan Curtis, “Design System Tiers,” Medium, <https://medium.com/eightshapes-llc/design-system-tiers-2c827b67eae1>.

A team can only use the components from its own local pattern library. But all teams can browse the component catalog of the other teams. This visibility is a good starting point for spotting cross-team inconsistencies. It's also a solid basis to start a “central vs. local” discussion.

The central and local pattern libraries could also use the same tool to develop and generate the design system documentation site. Popular tools for this are Storybook,²¹ Pattern Lab,²² and UIengine.²³ Using the same tool has the advantage that moving a component from the central to the local pattern library (or the other way around) is as easy as moving a component folder.

Now you've learned a lot of aspects that can help you when implementing a design system in a micro frontends project. In the next chapter, we'll broaden our view and look at other organizational implications this architecture introduces to your company.

Summary

- Every micro frontend team develops its own user interface. A central design system that all teams can use helps to deliver a consistent user experience across all micro frontends.
- A shared design system introduces coupling between the teams. All teams must work with the system and be compatible with its technology.
- All visible features the product teams produce rely on the design system. Changing the technical architecture of the design system afterward is cumbersome and costly.
- The design system exists to help the product teams ship features and be more consistent. It does not create value on its own.
- Developing a design system is a continuous process. Ensure that it's maintained properly and doesn't get out of date. Don't let it become a zombie design system.
- The design system can become a bottleneck when teams request more changes than the central team can handle. Product teams might need to wait and delay features.
- You can develop the design system in a federated way. Developers and designers from each team contribute to the system. A small core team ensures quality and has an eye on consistency. This model can scale and works well with the micro frontend principles.
- The central and federated development models are not mutually exclusive. You can move between them.
- There are two ways to integrate the pattern library into your project: runtime integration and via versioned packages.

²¹ See <https://storybook.js.org>.

²² See <https://patternlab.io>.

²³ See <https://uiengine.uix.space>. (This is the tool we are using in most projects.)

- With runtime integration, the design system team deploys directly to production, and all product teams must use the latest version. This model has considerable drawbacks for team autonomy, since teams cannot ensure that their software is always working correctly.
- Distribution via versioned packages enables the product teams to upgrade the pattern library at their own pace. A team's micro frontend can be self-contained because it doesn't rely on external dependencies at runtime.
- There are different formats (CSS only, framework-specific ...) a pattern library can publish their components in. The format has technical implications for the team. Some don't work server-side; others require all teams to use the same JavaScript framework.
- Frontend tools and libraries change over time. Try to build your design system in a way that makes adapting to changes easy.
- Sharing components across teams is not free of cost because they require a higher-quality standard. The central pattern library should include basic building blocks. More complex and domain-specific components should be local to the team that needs it.
- A product team can have its own local pattern library. It presents all the components this team owns. This is a good way for designers and developers to get an overview, spot inconsistencies, and start discussions.

13

Teams and boundaries

This chapter covers

- Structuring your teams to maximize the benefits of the micro frontends architecture
- Fostering a healthy amount of knowledge-sharing between the teams
- Identifying common crosscutting concerns and highlighting different strategies to address them
- Illustrating the challenges a diverse technology landscape can introduce
- Helping new teams to get up and running quickly

Throughout this book, we've focused on the technical aspects of micro frontends. You learned techniques to integrate independent user interfaces that form a greater whole. We talked about strategies to mitigate architecture-inherent issues like performance and providing a seamless user interface. But why are we doing all this?

Yes, there are some technical benefits that come with this architecture. Smaller software projects are simpler to build, test, understand, and rebuild than a mono-

lith. Being able to use different tech stacks in different areas of the product can also be a valuable asset.

However, *the most significant benefits our composable frontend architecture unlocks are the organizational ones*. It makes it possible to parallelize development. Properties like having real team ownership and local decision making can lead to faster innovations.

You might have noticed that I've used the word *team* a lot in this book—it occurs 1,723 times if I've counted correctly. This is not by accident or lack of creativity. It would have been perfectly fine to use words like *micro frontend application* or *software system* in most cases to understand the described techniques. *But it's not about the software. It's about the people designing and building it.*

I've talked to a lot of smart people who successfully introduced a micro frontends architecture in their company. In all cases, the motivation to go down this road was the organizational and not the technical benefits—setting up individual and robust teams and empowering them to build and improve a specific area of the product.

That's what we'll talk about in this chapter. What organizational and cultural changes should you make to leverage the full potential of this model? How can you address cross-cutting concerns without reinventing the wheel in each team? Lastly we'll look at the topic of technology diversity. How much freedom should a team have to pick their stack? Let's start with a bit of theory.

13.1 Aligning systems and teams

If you have ever explored the concept of microservices before, you've probably come across *Conway's Law*.¹ In the 1960s, computer programmer Melvin Conway formulated the hypothesis that the communication structures of an organization are reflected in the technical systems they create.

This means that if you let *one* team build a product, it will likely produce a more monolithic system. If you give the same task to *four* teams, they'll probably come up with a more modular solution.

The importance of keeping the structure of the organization and its technical systems in sync has been well researched² and understood in modern software development. Here's a quote from the book *Organizational Patterns of Agile Software Development*, published in 2004:

If the parts of an organization (e.g., teams, departments, or subdivisions) do not closely reflect the essential parts of the product [...], then the project will be in trouble... Therefore: Make sure the organization is compatible with the product architecture.

James O. Coplien and Neil Harrison

¹ See https://en.wikipedia.org/wiki/Conway%27s_law.

² See Alan D. MacCormack, et al., "Exploring the Duality between Product and Organizational Architectures: A Test of the Mirroring Hypothesis," <http://mng.bz/aRyj>.

For a micro frontend architecture, this means that the team boundaries should align with the boundaries of the vertical applications that form the product. Figure 13.1 illustrates this.

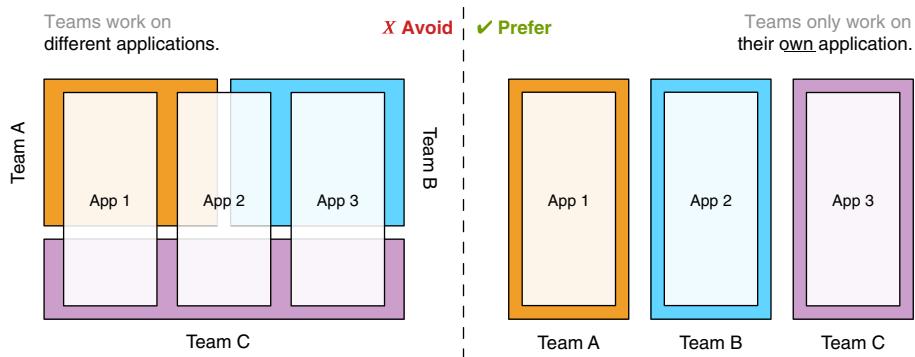


Figure 13.1 Team structure and software structure should align. Having one team working on multiple applications, or even worse, multiple teams working on the same application, can create issues. An architecture where one team owns one application will likely be more effective.

13.1.1 Identifying team boundaries

Ok, understood! We should keep team and software structure aligned. But how do we find out what structure is beneficial for the product we want to create? How do we identify sound boundaries? Here are three methods that can help you.

DOMAIN-DRIVEN DESIGN (DDD)

Domain-driven design is a popular approach for structuring software. It acknowledges the fact that it's hard to create a consistent model for a project of a specific size. It provides patterns to handle this complexity by creating smaller sub-models that have an explicit relationship with each other.

DDD provides a set of concepts and tools to identify and isolate areas in your project. It introduces the idea of analyzing the language of different experts and departments in a company: *ubiquitous language*.

By analyzing differences in vocabulary, it's possible to identify *bounded contexts*, one of DDD's core concepts. You can see a bounded context as a group of business processes that are related to each other. A checkout process could be viewed as a bounded context. It consists of different sub-topics like delivery and payment which are closely related to each other. We won't go into more detail on DDD in this book. Still, there's a lot of great content³ you can check out if you want to learn about it. A bounded context is an excellent candidate to become its own micro frontend application and team.

³ See tag *domain driven design*, <http://mng.bz/6Ql6>, and *Domain-Driven Design*, by Eric Evans, 2003, Addison-Wesley Professional..

USER-CENTERED DESIGN

Let's set aside our IT glasses and put on our product management scarf for a minute. A critical task in product design is to pinpoint user needs. In day-to-day business, it's easy to get lost in optimizing our current products.

If we want a sustainable relationship with our customers, it's essential to understand their real motivations. What do they want when they come to us? How can we make their life easier?

Techniques like *design thinking*⁴ or *jobs to be done*⁵ provide solid mental models to reason about a user's motivation. A famous quote⁶ from Theodore Levitt highlights the difference between our current offers and the users' needs:

People don't want to buy a quarter-inch drill. They want a quarter-inch hole!

Theodore Levitt

Modeling your teams and systems around your customers needs can be a valid choice. It gives the teams a clear goal that's focused on what matters most: your user.

In the example of Tractor Models, Inc., we've structured the teams and systems along the typical buying process of our customers. A customer goes through different phases like "browsing the site for interesting products" (Team Inspire), "considering if a specific product would be a good choice" (Team Decide), and finally "doing everything that's necessary to acquire the desired product" (Team Checkout). The customer has different needs in these three phases and the individual teams can specialize in addressing them.

You can apply these phases and user needs to other business areas. Let's look at another business. You have a company that sells Internet of Things devices like smart bulbs and sensors. Here you might have a "Which devices do I need?" phase, followed by a "How do I set it up?" phase. In the third phase, everything is running and the user wants to interact with the devices—checking measurements or switching the light. These three phases are good candidates to structure your software around. They don't have too much overlap and the user has very different needs in them.

EXAMINING EXISTING PAGE STRUCTURES

A more hands-on method for identifying boundaries is to look at the page structure of your current project. This method works when you already have a functioning business model. Print out all page types on a piece of paper. Gather a group of experienced colleagues and group the pages by using your intuitions.

In most cases, a page represents a specific use case or task your user needs to do. Looking at pages is not a perfect solution. Some pages might have more than one purpose. Print a copy of your page and use scissors to cut out parts from it. These cutouts

⁴ See https://en.wikipedia.org/wiki/Design_thinking.

⁵ See Clayton Johnson, "The Jobs to be Done" Theory of Innovation," *Harvard Business Review*, <http://mng.bz/oP7v>.

⁶ See Clayton M. Christensen, et al., "What Customers Want from Your Products, <https://hbswk.hbs.edu/item/what-customers-want-from-your-products>.

are candidates to become fragments. This method is an excellent entry to start more in-depth discussions.

If you've established groups, you can try to verify your hypotheses by looking at analytics data you've gathered in the past. Do the usage patterns align with your page groups?

Now that we have an idea of how to structure the teams, let's talk about who should be on the teams.

13.1.2 Team depth

The integration techniques described in this book are all frontend-related. But micro frontends is not an architecture limited to the frontend—on the contrary. It unfolds its full potential when it covers the complete stack. Figure 13.2 shows different depths of integration and their potential benefits.

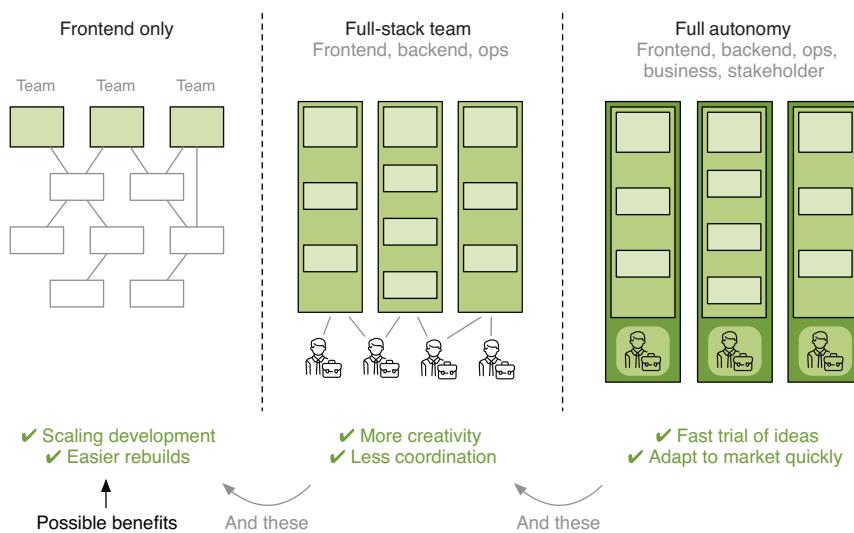


Figure 13.2 A micro frontend team can be limited to the frontend (left). However, when you add more disciplines like backend and operations to the team (middle), it becomes easier to ship features end-to-end. An ideal team also includes business experts and stakeholders (right). Then it's able to make all its decisions locally to create customer value.

Let's look more closely at the three approaches described in the diagram.

FRONTEND ONLY

In this model, you have a backend, be it monolithic or microservices-style. The vertical micro frontend teams sit on top of this backend. In case of a microservices architecture, each frontend might have its own *backend for frontends (BFF)*⁷ to communicate with the services.

⁷ See Sam Newman, “Backends For Frontends,” <https://samnewman.io/patterns/architectural/bff/>.

This approach has some real benefits compared to, for example, a monolithic single-page application:

- *Scaling development*—Here the Two-Pizza Team Rule we talked about in chapter 1 comes into play. Assuming you've come up with good boundaries, it's more efficient to have three teams with five developers, each working on a dedicated piece of software, than to have a 15-person team working on a large code base. It's easier for developers to understand the part of the system they're responsible for. When you've established a micro frontends architecture with three teams, all patterns are in place to create a fourth team that develops an entirely new part of the application. The other three teams can go on with their regular business. Integrating the new micro frontend with the existing application is a small amount of work.
- *Easier rebuilds*—Modernizing an existing micro frontend is a more straightforward task. You don't have to think about the complete application. You can upgrade and rebuild team-by-team. No all-hands-on-deck, big-bang migrations.

FULL-STACK TEAM

In this model, we make our micro frontends teams go beyond the frontend-backend line. Each team includes developers from the frontend, backend, operations, or data science. We form a cross-functional team that combines competences from database to user interface. Here are the benefits of the full-stack approach:

- *More creativity*—Cross-functional teams combine people with different backgrounds who provide different perspectives on a problem. This diversity can lead to better and more creative solutions.⁸
- *Less coordination*—The most significant benefit of the end-to-end team model is that it *reduces waiting time*. All features that can be accomplished inside team boundaries don't require other teams to become active. This autonomy eliminates the need for organizing meetings with other teams, formalizing requirements, and global ticket prioritization.

Moving to this deeply decoupled model introduces some unique challenges: How do teams share data in the backend when there are no shared services? This is typically solved by accepting data redundancy and asynchronously replicating data from other systems. In chapter 6 we talked about different technical solutions to architect this.

FULL AUTONOMY

We can take this one step further by also including domain experts and business people in the team. In most companies, these people typically work in departments like legal, marketing, risk, customer support, logistics, controlling, and so on. These departments specify requirements that the "IT people" must implement. Breaking up this traditional boundary and moving these experts closer to the development teams

⁸ See https://en.wikipedia.org/wiki/Cross-functional_team#Effects.

is not an easy task. It's a slow transition that must be encouraged by the top of the organization.

Having expertise like marketing, legal, or customer support directly available in a development team can unlock further benefits:

- *Fast trial of ideas*—Moving from a formal requirements and prioritization process to a basis where you can exchange ideas at eye level can improve your product. Here's a small-scale example: In our last project, the team developing the checkout system invited people from the call center to its end-of-sprint meeting. A developer presented the new voucher system. A call center employee interrupted and described the fact that older customers are often stressed by the minimum order value—especially if their shopping cart total is only slightly below it. In this meeting, they came up with the idea to make the minimum value constraint more tolerant: communicating a minimum of \$20 but enforcing only \$18. This trivial software change had a measurable effect on customer satisfaction: fewer support calls and a more generous company image. Ideas and changes like this can make a big difference.
- *Adapt to market quickly*—The digital services landscape and your user's expectations can change fast. New forms of payment methods, integrations with social platforms, and communications channels emerge. When all people that are necessary for strategic decisions work on the same team, you can move quicker.

A general rule of thumb is that extending your vertical teams deeper into the organization will likely increase the speed and quality of these teams' work. If you want to get deeper into this topic, the *Agile Fluency Model* is a good starting point. The model describes four fluency zones an agile team can reach.⁹ A team in the first zone (*Focusing*) leverages basic agile practices like scrum to improve its work. Running a micro frontends architecture with *full-stack teams* aligns with the second stage: *Delivering*. The *full autonomy* approach maps to the third agile fluency stage: *Optimizing*.

A development team can decide to adopt the micro frontends architecture for technical reasons. But extending this model to the entire development team or even an organization is a significant management task. Let's briefly talk about the cultural changes that come with it.

13.1.3 Cultural change

The vertical architecture plays well with having a user-focused culture. Every team delivers to the customer directly.

This mentality is often already part of the DNA of startups. That's why the proposed vertical team structure might feel like a natural way of growing a startup.

Large traditional organizations have a harder time moving to a more vertical architecture. They often think in short-term projects rather than long-term products. Also, the concept of *ownership* plays a vital role in running this architecture successfully.

⁹ See <https://www.agilefluency.org>.

You want teams to identify with the product they create and make it more valuable for the user. Teams should be empowered to make decisions, conduct experiments, and learn from failures. Hierarchies and department structures might get in the way. I think having an open culture based on agile values¹⁰ is a prerequisite for getting the most out of a micro frontend-style architecture.

13.2 Sharing knowledge

The cross-functional team structure optimizes communication along with a business domain (vertical). This model is good because it helps to focus on the user, but it also introduces challenges: *How do you avoid reinventing the wheel in every team?*

Ok, granted—the majority of the work in these teams is not the same. Developers building a fast-loading product list are faced with other challenges than the developers who are architecting a registration form that has to work in all countries around the world.

But there are aspects that all teams share. What are the right strategies to automatically test the software? What's a good way to handle state inside my application? “I've encountered a strange issue. I wonder if anyone else has this problem?”

Let me give you a real-world example of insufficient cross-team communication. In my last project, we've been developing an e-commerce shop with five teams staffed from three software companies. Half a year into the project, a co-worker from one of the other companies gave a talk on debugging Node.js performance at a conference in Hamburg. I attended his talk because I was curious. On stage, he referred to a mysterious problem he'd been tracking down for the past few weeks. He was convinced that it had to be something in his team's application code. The behavior he described was instantly familiar to me because I'd encountered something very similar in our team's application. After the talk, we spoke and shared our findings. It became apparent that it had to be an issue with the hosting infrastructure our applications ran on.

But the fact that we had to meet at a public conference to figure this out was a little disturbing. We could have saved each other a considerable amount of time and headaches if we had spoken to each other earlier.

13.2.1 Community of practice

In the early '90s, the concept of a *community of practice (CoP)*¹¹ was formulated. It describes ways to spread knowledge across teams. A CoP is a group of people that share a craft or profession. In our example, all people doing frontend work could be part of the same CoP. These groups create their communication channel to exchange information on a specific technology, ask for help, or share learnings.

Spotify is famous for its agile and team-focused organization structure. They also organize in end-to-end teams. They've institutionalized communities of practice

¹⁰ See <https://agilemanifesto.org/>.

¹¹ See https://en.wikipedia.org/wiki/Community_of_practice.

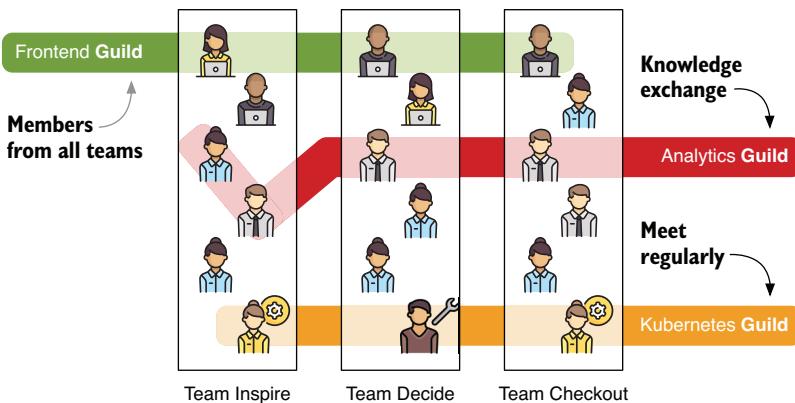


Figure 13.3 A guild creates a room for people from different teams that share an interest or profession. Their primary goal is to exchange knowledge.

called *guilds*.¹² There, like-minded people from different parts of the organization can exchange knowledge. Figure 13.3 shows some example guilds that form horizontal channels across our otherwise vertical organization structure.

Guilds typically have a dedicated communications channel like a Slack group. All guild members meet regularly. In our projects, short guild meetings happen (bi)weekly via video call to discuss recent issues. From time to time, the guild also organizes longer in-person workshops for diving deeper into a specific topic.

Typical guilds in our projects are frontend, backend, UX/design, analytics, infrastructure, data-science, coaching, security, and macro-architecture.

13.2.2 Learning and enabling

Having a cross-functional team that's able to handle the complete stack flawlessly sounds terrific on paper. In practice, it is rarely possible to assemble a team that is capable of developing customer features while also mastering all non-functional requirements such as performance, security, or testing. Being faced with these expectations as a team can be intimidating.

In some areas, technical developments like *cloud hosting* play in our favor. A team can offload tasks like “managing real hardware” to a cloud provider. These services make a *You build it; you run it!* approach more realistic.

But that's not possible for all topics. Learning and improving is an integral part of forming a cross-functional team. Having each team identify and formalize its strengths and weaknesses can speed up the learning process.

CoPs can play a central role in education. A developer from Team A who has experience in analytics can teach other guild members and help them to level up their

¹² See Darja Smite, et al., “Spotify Guilds,” ACM, March 2020, <http://mng.bz/4Awv>.

skills. For some topics, it might also be a good option to hire an external mentor to support a guild.

13.2.3 Present your work

Another way to exchange information is by presenting your team's work. This way, teams have an idea of what the others are working on. This can be in the form of a real on-stage presentation where all teams show what they've accomplished and learned in the last month. But it can also be in the form of a small internal blog post.

Since all teams work in different areas, these presentations usually don't have much immediate value for the others. However, they enable moments like, "Wait, hasn't Team Inspire also built a feature with Apache Spark? Let's talk to them first." Rituals like these can also strengthen group cohesion and avoid an "us vs. them" mentality.

13.3 Cross-cutting concerns

Let's dig a little deeper into these cross-cutting concerns. Yes, rituals like guilds can help to spread knowledge, but let's look at some concrete examples of how to address common topics.

13.3.1 Central infrastructure

Some cross-cutting concerns require dedicated infrastructure. Examples of this are your version control system, continuous delivery pipeline, analytics, dashboards, monitoring, error tracking, hosting setup, and shared services like a load balancer. Each team could make these choices on its own. However, these are all general topics most professional software projects need. Having each team figure out a solution might not be the best use of their time.

Establishing a shared set of infrastructure all teams can use is a good idea. There are different ways to organize this.

SOFTWARE AS A SERVICE (SaaS)

For commodity products, it's often easiest to use an off-the-shelf product like Amazon's AWS for infrastructure and GitLab for version control and pipelines. Using standardized services does not introduce inter-team coupling. All teams communicate directly with the provider of the service. Make sure that each team has its sub-account or an explicit namespace to avoid conflicts. If a team has a strong need to switch to another provider, there should be no technical hurdles.

Sometimes going for a SaaS solution is not an option. The reasons for this might be the price or lack of functionality. If you need to run an infrastructure component yourself that all teams can use, you have two options: having it owned by one of the product teams or introducing a dedicated infrastructure team.

OWNED BY ONE PRODUCT TEAM

In this model, the product teams take responsibility for the self-hosted central services. Team A might be responsible for setting up, running, and maintaining the shared load balancer. Team B might run a private NPM registry that all teams can use.

Having clear responsibility is essential to ensure that these services receive the attention and care they need. By spreading the services among the teams, the burden that each team has to carry is reduced. This model works well if the number of self-hosted services is not too high and the services themselves are easy to maintain.

WARNING Share generic infrastructure components only. Avoid sharing business logic this way, because it creates coupling and undermines team autonomy.

CENTRAL INFRASTRUCTURE TEAM

If the preceding methods don't work for you, there's always the option to create a dedicated infrastructure team that takes responsibility for all shared infrastructure aspects. But this pure infrastructure team does not fit well into our otherwise vertical and customer-centric architecture. It has the potential to become a bottleneck that hinders feature development.

13.3.2 Specialized component team

Sometimes there is neither a managed service nor an open source solution that fulfills our need. This is where the concept of *component teams* comes in.¹³ Spotify calls these teams *infrastructure squads*.¹⁴

Say different product teams need to talk to a legacy ERP system that does not support modern APIs. Having a dedicated team that develops a service or an abstraction library might save the product teams a lot of time. Another example of a component team is the *central design system team* we talked about in the previous chapter.

Component teams don't provide direct value. Their goal is to enable the product teams to move faster. Since the introduction of a component team creates friction and inter-team dependencies, its use should be carefully considered. These two questions can help with deciding if you should use a component team or not:

- Is the service it provides *required by many teams*?
- Does building the service *require specialized technical expertise* that is not present in the product teams?

If you can answer one, or better both questions with a clear yes, it might be worth thinking about a component team.

13.3.3 Global agreements and conventions

Not all cross-cutting concerns manifest themselves in a shared service or library. Often an agreement to which all teams adhere is enough. For topics like currency formatting, internationalization, search engine optimization, or language detection, it's often perfectly fine to have central documentation.

¹³ See “Organizing by Feature or Component,” <https://www.scaledagileframework.com/features-and-components/>.

¹⁴ See <http://mng.bz/XPRp>.

All teams agree upon the described canonical way and implement it in their applications. Yes, this may result in redundant code, but for topics that are not critical or don't change frequently, it's often the most effective way.

13.4 Technology diversity

The micro frontends architecture enables each team to pick and change their technology stack. We've already discussed the benefits this introduces. But just because you *can* does not mean you *must* use a diverse technology stack.

Having to hand-pick a technology stack can also be a burden. Let's talk about some techniques to make these decisions easier.

13.4.1 Toolbox and defaults

The *toolbox* idea explicitly limits the technology choices by providing a list of vetted options. It's a project- or company-wide piece of documentation that may live in the wiki. The content of the toolbox might read like this: Java or Scala are the backend programming languages of choice. PostgreSQL is our go-to for relational databases. You should stick to Webpack for your frontend builds.

The toolbox should be guidance and not a set of laws. If teams have reasons to deviate from the norm, they should have the possibility to do so. For most teams, the toolbox is a source of sensible default options. Need an end-to-end testing framework? Let's open the toolbox and see what has worked for other teams.

Since technology is evolving, the toolbox has to be a living document that's updated regularly by adding new technologies that have proven valuable or deprecating existing ones that went out of date.

13.4.2 Frontend blueprint

When a new team starts fresh, it has to do a lot of setup work, creating its basic application, build process, and other tedious tasks that are necessary before it can become productive.

We've been using the concept of a *shared frontend blueprint* to ease this pain. The blueprint is an example project that includes all significant aspects a micro frontend application needs. We can divide these aspects into two groups: technical and project-specific.

TECHNICAL ASPECTS

- Directory structure
- Testing (unit, end-to-end)
- Linting and formatting rules
- Code formatting rules
- API communication
- Performance best practices (optimizing assets)
- Build tool configuration

These general topics are necessary to have, but they are not that interesting. Most major JavaScript frameworks have a scaffolding tool that generates an example project for you. But a stock frontend setup will not be sufficient for a team to get going.

PROJECT-SPECIFIC ASPECTS

Your frontend needs to integrate with the other teams and must adhere to the high-level architecture guidelines. A new frontend must also cover *project-specific* aspects. That's why our frontend blueprint also includes

- Composition examples
 - Including another micro frontend
 - Providing an includable micro frontend
- Communication examples
 - Team prefixing for CSS and URLs
 - Template for documenting your micro frontends
 - Integration with the central pattern library
 - Setup for the local pattern library
 - Wiring for shared services like error tracking or analytics
 - CI/CD pipeline

New teams will copy the blueprint over to their project and adjust it to their needs. Building on the existing work reduces setup time noticeably. But for us, the blueprint has another, even more, important role. It's the reference implementation for the macro architecture decisions.

It includes running examples of integration patterns and communication strategies. This example code helps all developers to understand high-level topics by seeing them in action in a real application.

MAKE IT OPTIONAL

Teams are not forced to use the blueprint as is, or even at all. They are free to adapt it to their needs. It's explicitly not a shared production code base. The frontend applications are based on a copy. Making a change to the blueprint will not affect the existing frontends. Developers communicate improvements to the blueprint via the frontend guild. If a specific improvement is valuable for a team, that team can look at the change and apply it manually.

13.4.3 Don't fear the copy

As you've seen with the blueprint, it's often a good idea to copy and paste from other applications. For everyday tasks, it's an easy solution that ensures team autonomy down the road. Copying the 15-line currency formatting algorithm from your neighbor team is a good example. This algorithm is not set in stone, but it's easy to understand and unlikely to change monthly.

We, as developers, have a trained tendency to spot and eliminate duplications. But this elimination is not free—especially when you try to centralize across teams.

Maintaining a shared library that six teams depend on is not a trivial job and will come with a lot of discussions, waiting, and headaches.

For bigger use cases, the pain a duplication introduces might be greater. Our poster-child example is the central pattern library. You don't want to copy and paste it on every change. There might be other pieces of code you want to share, like a library that makes talking to a legacy system easier. Sharing these as a versioned library might be fine, but it should always be a conscious decision and come with the right amount of dedication. In discussions, I found this quote helpful to create the right mindset:

Only do it if you are willing to run it as a successful (internal) open source project.

Don't underestimate the organizational overhead a shared library introduces.

13.4.4 The value of similarity

In our projects, teams often picked similar programming languages and frameworks to build their applications.

Using the same technologies as your neighbors has advantages. It makes sharing best practices more accessible. Developers that want to switch teams can get up and running quickly. The ability to browse other teams' Git repositories and see how they've solved a particular task is also valuable.

Artifacts like the toolbox and the blueprint can help in forming a shared technical direction. Finding the right balance between similarity and freedom is never easy. Technical arguments often drive discussions around this. But taking the business, or even better, the user perspective can help to maintain focus. Will using Haskell instead of Scala improve the product in a noticeable way?

Summary

- Running a successful micro frontends architecture is not a technical decision. The team structure should align with the software systems to be most effective.
- There are different ways to identify team and system boundaries. Domain-driven design provides tools like analyzing expert language to identify groups of functionality. Bounded contexts are good candidates for a micro frontend team.
- Organizing your teams around user needs can be a good model. Techniques like design thinking and jobs-to-be-done can help to isolate these use cases.
- The existing page structure of your site might already be a good indicator of team boundaries. The question "What purpose does this page serve?" can lead you to groups of functionalities.
- Using micro frontends only on the frontend has technical benefits like parallelizing work and easier rebuilds. Rolling it out to the complete development stack or extending it further also to include stakeholders and business experts can unlock further benefits, like faster development and a better customer focus. The vertical team structure aligns well with the upper stages of the Agile Fluency Model.

- The vertical architecture optimizes for delivering features inside a team's scope. Introducing horizontal groups like communities of practice or guilds helps to spread knowledge.
- It's often more efficient or necessary to run a shared infrastructure. Leveraging SaaS solutions like AWS can be a good option that doesn't introduce inter-team coupling. Sometimes the SaaS model doesn't fit, and you need to self-host. You can distribute the responsibility for the infrastructure components across the product teams. Introducing a dedicated infrastructure team is an alternative, but does not fit well into a vertical architecture.
- Since micro frontends are decoupled, each team can choose its technology stack freely. Methods like a shared toolbox or a central blueprint can help to form a common technology direction that ensures room for innovation and experimentation when needed.

Migration, local development, and testing

This chapter covers

- Migrating a monolithic application to a micro frontends architecture
- Setting up a local development environment and examining techniques like micro frontend mocks to ensure independence
- Implementing automated testing in a micro frontends architecture

Micro frontends is not the first architecture for most companies. It's something you migrate to because the old architecture has trouble keeping up with new demands like increasing team size or high demand for features.

If you are a fresh startup that needs to grow quickly, it might be a good idea to start with micro frontends from scratch. However, most larger companies use micro frontends to replace a functioning but slow or unmaintainable monolith. If you find yourself in the latter camp, this chapter will help you by highlighting some good migration strategies.

In the second part of this chapter, we'll take a closer look at the developers' day-to-day life in a micro frontends project. A team only works on its slice of the complete application. Developing a feature locally without seeing it integrated with the rest of the software will feel strange at first. You'll learn techniques and tricks that make developing and testing easier.

14.1 **Migration**

Migrating a non-trivial project from one architecture to another is a scary and often costly task. You can take different roads, which all have their benefits and drawbacks. On the following pages, we'll discuss three ways to move to a micro frontends architecture. This chapter will not be the definitive guide for software migrations. Lots of publications describe the essential parts you should think about when migrating a large project. Instead, we will focus on the micro frontend-specific aspects.

Having a somewhat realistic idea of the complexity and effort a migration takes is vital to set expectations and calculate costs. But when your team doesn't have experience with the target architecture, it's hard to come up with reasonable estimates. Playing around with the technology in a sandbox project helps to reduce the fuzziness. The examples in this book can be a good starting point for these experiments.

Micro frontends' user-interface integration techniques are a valuable asset for incremental migrations. The micro frontends paradigm and its frontend integration techniques lend themselves well to building and integrating a proof of concept and even verifying it in your production application. Before we go into the migration strategies, let's have a closer look at this proof of concept idea.

14.1.1 **Proof of concept and building a lighthouse**

You can adopt micro frontends by building a single feature as its own end-to-end system and integrating it into your existing application. Figure 14.1 shows a two-part diagram that illustrates this.

A REAL WORLD EXAMPLE

Let's look at a concrete example. The company Miniature Farming Industries, one of Tractor Model, Inc's rivals, has a monolithic e-commerce shop that doesn't perform well. They consider moving to a micro frontends architecture. To test out the waters and avoid losing a lot of time, they decide to develop one of their already planned features as a micro frontends application.

Miniature Farming Industries forms a new team dedicated to building this new feature: the wishlist. The core user-facing aspect is *the wishlist overview page*, where the users can see and manage their favorite products. Also, a user should be able to add products to the wishlist by clicking a small heart icon button on a product tile. The new team builds and owns both the wishlist page and the add-to-wishlist button.

The wishlist page should have the same header and footer as the other pages of the shop. Since the new team doesn't want to duplicate the header and footer, they

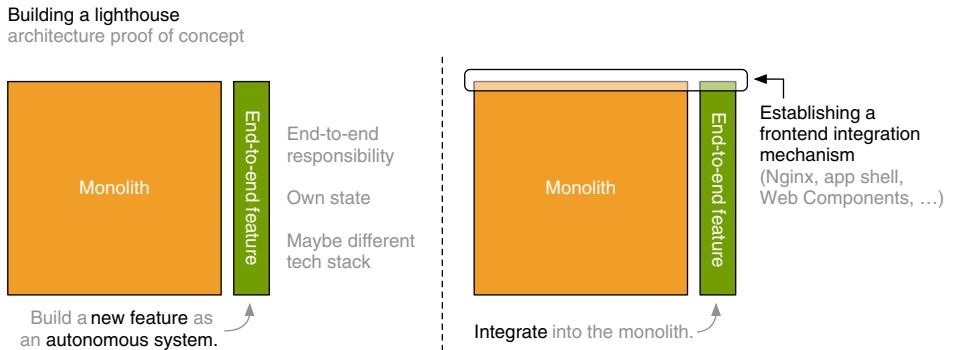


Figure 14.1 To try out the micro frontends architecture, you build a new feature as a dedicated application that has its own state but also includes the associated user interface. It's decoupled from the existing monolith. That's why the team responsible for this new feature can build it based on a new technology stack if it wants to (left). A frontend integration mechanism is established to integrate this new application with the monolith (right). The frontend integration can be as simple as using hyperlinks between both applications, but depending on your architecture choice, it might also be the introduction of a frontend proxy or application shell.

decide to include it from the existing application as a fragment. To make this possible, the team working on the monolith has to provide the header and footer as standalone micro frontends. In reverse, the wishlist team provides the *add-to-wishlist button* as a fragment for the monolith to include in every product tile.

The teams must establish a shared integration technique. They go with a server-side composition using SSI. Therefore they install an Nginx server as a frontend proxy that sits in front of both applications. This server has two tasks: routing and composition. All requests starting with /wishlist get routed to the new application; all others hit the monolith. The web server also handles composition. It replaces the header/footer SSI directives of the wishlist page with the actual markup from the monolith.

That's everything required to make the integration work. Ok, not quite. The teams also needed to work on some other relevant topics. The frontend developers refactored the CSS code of both systems to ensure that the old and the new application don't over-style each other. The backend developers had to build an import for necessary product data like image, name, and price. The data import is necessary to ensure that the new system has its own data store and doesn't depend on the monolith at runtime.

THE ROLE MODEL

If everything goes as planned, this first vertical system can act as a lighthouse for your migration project. We've established a frontend integration mechanism that new systems can use. The "proof of concept" can become the role model other teams can follow to build new features.

14.1.2 Strategy #1: Slice-by-slice

The first migration strategy, *slice-by-slice*, is a natural progression from our earlier proof-of-concept. Figure 14.2 shows a monolith that's migrated to a three-team micro frontend architecture.

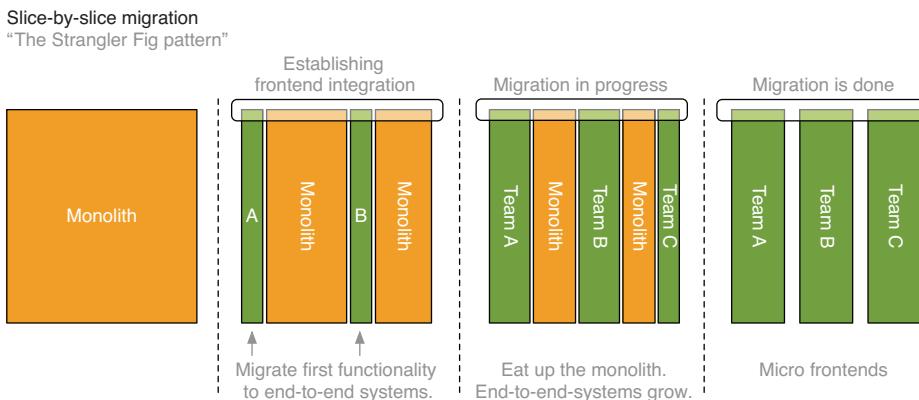


Figure 14.2 Migrating a monolithic application (left) to a three-team micro frontends architecture (right). In this diagram, we create three new applications (Team A-C), which take over functionality from the monolith step by step until the monolith has vanished (middle). Like in the previous example, we start with establishing the required frontend integration mechanism that handles the routing and composition of the different applications.

HOW IT WORKS

First of all, we need a shared plan for what the final team boundaries should look like. Which team owns which feature? We talked about methods to identify these boundaries in the previous chapter. After these decisions, the teams can go ahead, set up their new applications, and start migrating functionality from the monolith into their micro frontend application. They migrate the system feature by feature. The first feature to extract might be product reviews. One team moves the feature over to their application, from user interface to the database.

The teams establish a frontend integration mechanism that handles routing and composition. After migrating a feature, the team replaces the associated user interface in the monolith with the new micro frontend's UI. Then they tackle the next feature.

The teams repeat this process until the monolith has vanished. This migration follows the *Strangler Fig Pattern*.¹ This pattern describes how a new application gradually replaces the existing one. During the migration phase, both applications are still in business.

BENEFITS AND CHALLENGES

The main benefit of this incremental migration approach is that it introduces little risk. The newly created software goes into production regularly. There's no big-bang

¹ See <http://mng.bz/yymy>.

moment when switching from the old to the new system. The system is always in a working state. Even if you decide to cancel the migration project in the middle of the process, you have a functioning application. All software that's written goes to production quickly.

Projects where legacy code works together with newly created systems are often called *brownfield projects*.² This term is in contrast to *greenfield projects*, where you build a new system from scratch on a “clean sheet” without caring about the existing architecture.

Compared to a greenfield project, our incremental approach requires more thought, understanding of the existing system, and coordination. Extracting features from the monolith does not mean that you have to remove them. However, you will at least have to adapt the monolith’s user interface along the process to play nice with the new micro frontends. Depending on the software quality, the CSS code and lack of proper scoping are often the most significant tasks that you face. Web Components and Shadow DOM can be of help. Revisit section 5.2 for more details on this.

14.1.3 Strategy #2: Frontend first

The frontend-first approach follows a similar pattern, but avoids mixing the old and the new frontend code. Not having to care about the “old frontend code” can make your life easier, especially when you are planning to do a frontend facelift along the way. Figure 14.3 shows the migration process.

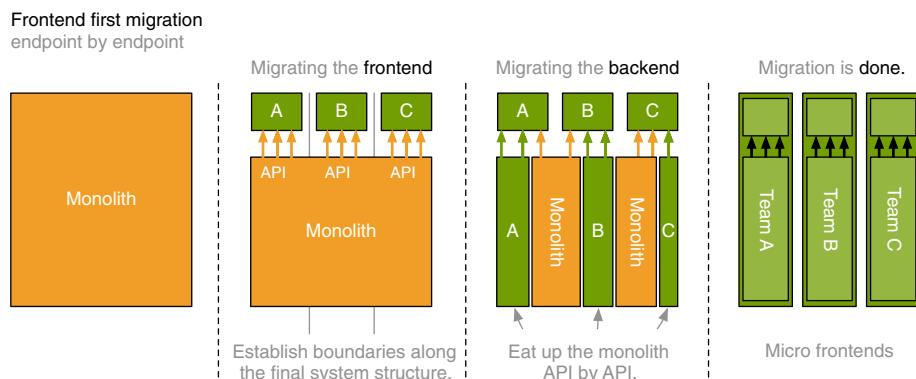


Figure 14.3 We start with a monolith (left). The migration has two phases. In the first phase, we replace the frontend of the monolith with three new frontend applications, which are each owned by one team. The frontends communicate via APIs with the old monolith. In the second phase, we migrate the backend with the slice-by-slice approach, migrating each API endpoint into the new backend application of the responsible team. After the backend migrations, we’ve reached our goal: a vertically sliced application (right).

² See <http://mng.bz/aRno>.

HOW IT WORKS

Here the migration is a two-phase process. We start with the frontend. It's rebuilt to fit into the desired vertical structure. You need to plan team boundaries and responsibilities ahead of time. Each team builds its own part of the frontend. Teams integrate their user interface via the known routing and composition techniques. The new frontends receive their data from the old monolith. In this process, new APIs are added to the monolith to serve the data needs of the frontend applications.

In the second phase, we start splitting up the backend. The APIs we've implemented in the previous step define the boundaries and guide the way for the backend. Each team creates a backend application that's able to replace the monolith APIs its frontend relies on. In this phase, we can again apply the slice-by-slice pattern. The teams replace API after API until the monolith isn't required any more.

Now we've reached our desired state. The monolith has vanished, and each team owns a system that reaches from frontend to backend.

BENEFITS AND CHALLENGES

As I said before, the most significant benefit with the frontend-first approach is that we don't have a phase where the old and new frontend code mixes. There are no issues with leaking styles or unexpected side effects because we create a clean, new frontend landscape in one step. If your frontend does not contain too much business logic and complexity, this approach also has the benefit of delivering fast results.

We had good experiences with this approach. However, it has two disadvantages that you should consider.

The required frontend and backend work will not be distributed evenly. The first phase is more frontend-heavy, and in the second phase, the backend work dominates. You can counteract this by overlapping the phases or, even better, encourage your teams to work cross-functionally.

The second aspect you should keep in mind is that visible progress in this model is non-linear. From an outsider's or the management's perspective, the first phase, rebuilding the frontend, will introduce a lot of improvements. Even if you don't build new features, the use of modern technology or the introduction of a new design will make the site feel faster and fresher. The second phase will, at best, not introduce any visible change to the user at all. This lack of visual progress might not be a problem, but you should manage expectations accordingly.

14.1.4 **Strategy #3: Greenfield and big bang**

The greenfield and big bang approach is the easiest from a conceptual standpoint. The old system stays as is, and you build a new system in a clean environment in parallel: a greenfield project. When the new system is ready, we switch over to the new system: the big bang. Figure 14.4 illustrates this approach.

Greenfield migration

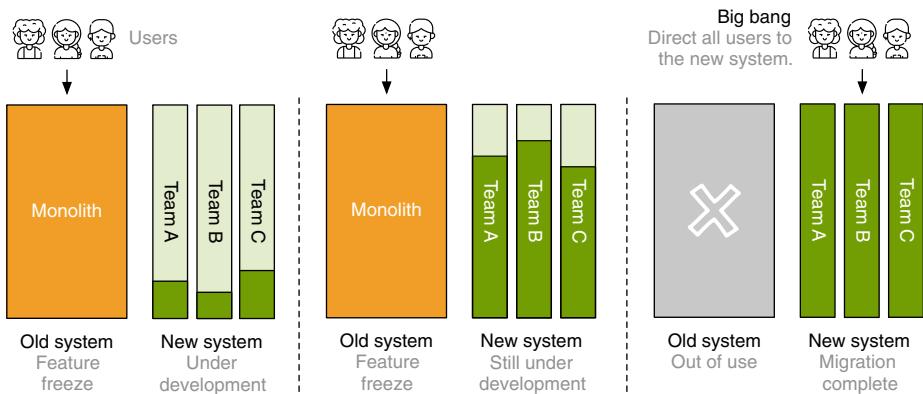


Figure 14.4 We set up our new team structure and system architecture beside the existing monolith (left). The new and old systems don't share anything. During the development phase, all incoming traffic still arrives at the monolith (middle). When the teams finish building the new system, we route the incoming traffic to the new system and retire the old one. The old and the new systems don't mix at any time. Users are either using the old or the new system.

HOW IT WORKS

We make a plan for how the new system should look and set it up in a new environment that's separate from the existing monolith. The development of the old system is often halted to avoid extending the migration phase. The teams start building their slices of the system. When all teams are finished implementing the features that are necessary for production, we route the incoming traffic to the new system and retire the old one. The old and the new systems don't mix at any time. Users are either using the old or the new system.

BENEFITS AND CHALLENGES

The main benefit of a greenfield approach is the fact that we can start fresh and don't have to deal with legacy code. The clean slate makes it easy to adopt techniques like continuous delivery or introduce a new design system that can be free of hacks and compromises. Because teams can focus on building the new architecture and don't have to wrestle with the legacy system, development will be faster.

We've used this migration strategy in different projects. It's attractive when it's hard to adapt the existing monolith during the migration process. This inflexibility may be the case when the monolith relies on proprietary technology that you can't change, or when it's on a very long deployment cycle that would slow down your development.

But as the big bang in the title implies, there's a considerable amount of risk associated with this approach. The teams develop the new system over a long period without receiving real user feedback. Verifying that the system works in production is extremely valuable. Consider moving users to your new system as early as possible. Having actual users reduces risk and increases confidence in the system you're building. Concepts like releasing it as a beta version or testing it in smaller markets can be of help.

Now you've seen a couple of strategies for getting from monolith to micro frontends. There's no golden way, and it always depends on the system you have and the goals you want to reach with the new architecture. But leveraging frontend integration techniques to gradually replace the old monolith with new micro frontend applications is a powerful tool that you should consider.

14.2 Local development

Now we'll leave the architecture level and zoom into the day-to-day life of a developer working in a micro frontends project. Running and developing a classical monolith is pretty straightforward. You can check out one source code repository, which contains everything required to start the complete application on your local machine. Everything should work, and you can try the application in your browser from start to finish. With a distributed architecture like micro frontends, this gets more complicated.

14.2.1 Don't run another team's code

Each team has its source code repository, and teams may have different tech stacks. Yes, it might be possible for a developer to not only have their team's repository checked out but also pull an up-to-date copy of the other team's source code regularly. While this might work, it can become cumbersome very quickly. Having to know about the development environment of other teams introduces friction.

What do you do if the other team has a bug that prevents their application from starting? Has Team B upgraded to the latest version of Node.js or are they still on the old one? You shouldn't have to care about these kinds of problems to do your job. You should be able to focus on the code your team owns. So, let's talk about how we can develop without running other people's code.

But what about monorepos ?

When you read about micro frontends on the web, the term *monorepo*^a sometimes appears as a solution for local development. Monorepo describes a concept where the code of independent applications or libraries live in one version control repository. A monorepo makes it easy to download and update multiple projects at once and manage shared dependencies.

If you see micro frontends purely as a set of integration techniques for one team to modularize its frontend, the monorepo approach is reasonable. However, if you want to take advantage of the organizational benefits of multiple independent teams that can work side by side without close coordination, the monorepo is an anti-pattern. The team's applications should be independent and shouldn't share code or a deployment pipeline. Separate repositories guard against unwanted inter-team dependencies.

^a See <https://en.wikipedia.org/wiki/Monorepo>.

14.2.2 Mocking fragments

TIP You can find the sample code for this chapter in the `21_local_development` folder.

Ok, so if I can't run the code from other teams, how can I develop? On a page level, the answer is simple: replace other teams' fragments with mock versions of them. Let's look at Team Decide's product page.

Go into the sample code and run the following command:

```
npm run 21_local_development
```



Open up `http://localhost:3001/product/porsche` to see the product page in local development mode. Figure 14.5 shows the result.

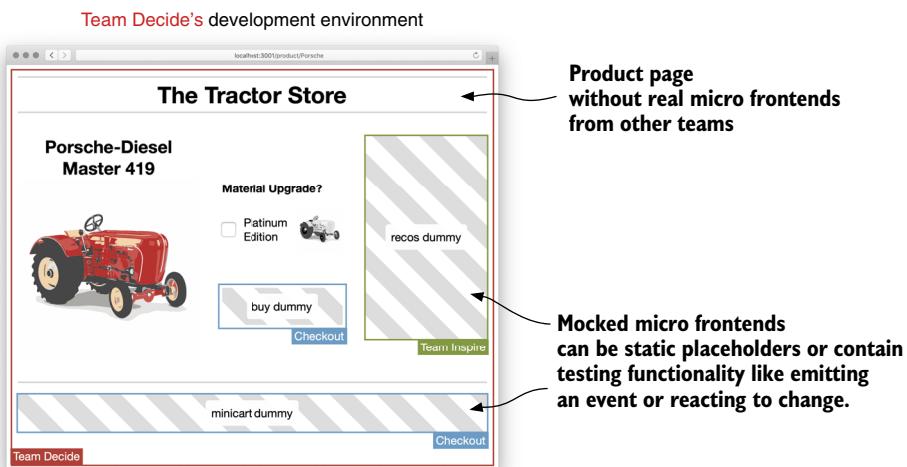


Figure 14.5 Team Decide's product page in local development mode. The fragments from the other teams are replaced by simple mock micro frontends.

We see the product page, but the fragments from the other teams (recommendations, Buy button, and mini-cart) got replaced with mock versions of these micro frontends. But the page itself is working as expected. You can toggle the platinum option and the product image updates accordingly.

The product page you are seeing does not include any code from other teams. In development mode, Team Decide omits the script and style tags from the other teams' fragment definitions. Not loading these files would lead to empty blocks where the fragments should be.

To improve this, Team Decide created its simple mock implementations for the three fragments. You can find the associated code in `team-decide/static/mock-fragments.(css|js)`. Since we are using Custom Elements for integration, it's pretty easy to mock the fragments. Here is the code for one mock.

Listing 14.1 `team-decide/static/mock-fragments.js`

```
...
class CheckoutMinicart extends HTMLElement {
  connectedCallback() {
    this.innerHTML = `<div>minicart dummy</div>`;
  }
}
window.customElements.define("checkout-minicart", CheckoutMinicart);
...
```

This code is a pretty simple mock that just shows a text. But if you expect a fragment to throw an event, you can get more sophisticated and, for example, add a button that triggers the event.

NOTE The example uses client-side rendering, but the concepts are also applicable for a server-generated application. Instead of replacing Custom Element definitions, you'd route the fragment's HTTP request to an endpoint that returns mock markup.

Using mock fragments instead of pulling in real components will make development more straightforward and more robust. You only have to fire up your application, and if something breaks, you can be sure that it's the fault of your code.

Each team that provides a fragment should document its interface. The interface lists the parameters it understands and the events it can emit. The fragment documentation can be the basis for creating your local mock.

WARNING If you find yourself in a situation that requires building a lot of sophisticated mocks to develop and test your software, you might have issues with your team boundaries. Make sure the responsibility for one use case is not spread across different teams.

14.2.3 *Fragments in isolation*

Let's see what developing a fragment looks like. Keep the sample application running and open your browser at `http://localhost:3003/sandbox` to find Team Checkout's sandbox page, which shows both of their fragments. Team Inspire has a similar page running on port 3002. Figure 14.6 shows both sandbox pages.

DEVELOPMENT PAGE

The sandbox page acts as the development environment for fragments. It's an empty page (in this case with a stripy background) that contains a team's fragments. The page itself also includes basic global styles like root font definitions and some CSS

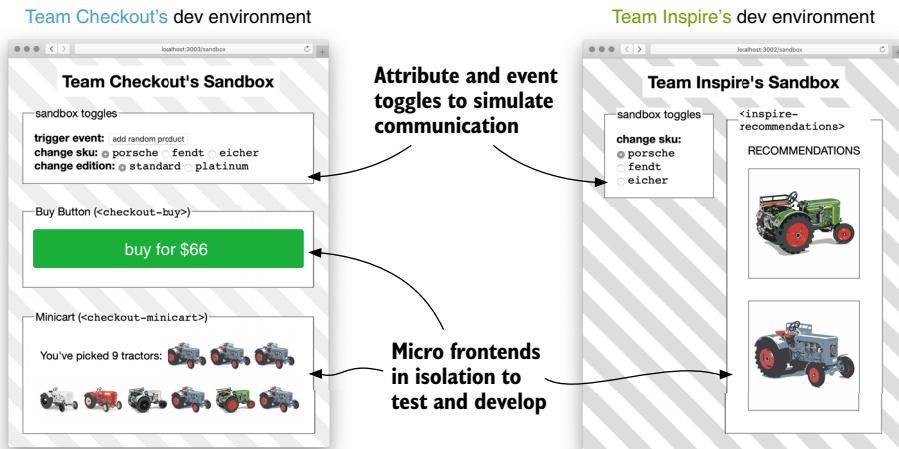


Figure 14.6 Each team has its own sandbox page where it can develop and test fragments in isolation. The sandbox page also contains some toggles to simulate communication.

resets, since you don't want every fragment to redefine these styles itself. Tools like Podium create such a page out of the box,³ but building this page from scratch is also not complicated. You'd also use your favorite live-reload or hot-code-replacement solution here to make development more enjoyable.

SIMULATING INTERACTIONS

Now we have an environment to develop our fragments in, but how do you test communication across micro frontends? You might have noticed the “sandbox toggles” section at the top of our pages. It contains a set of actions our fragments can react to.

You can, for example, use the “change sku” control to switch from one tractor to another. Changing the option will toggle the associated `sku` attribute of the Buy button fragment, which should then update its price accordingly. In the example, the toggle mechanics are a few lines of plain JavaScript in the sandbox file.

The mini cart also updates itself when someone clicks the Buy button. You can test this fragment-to-fragment communication on the sandbox page. Click the button, and the product will appear in the mini-cart. The mini-cart listens to the `checkout:item_added` event on the window, just as it would on a fully integrated page. The sandbox page also has a dedicated `add random product` button that triggers such an event.

INDEPENDENCE THROUGH MOCKS

Working with mocks can give you a lot of independence and reduces inter-team friction. When tests fail, you can be sure that your own code caused the issue. It can't be the fault of another team's script, because they aren't even included. This approach

³ See https://podium-lib.io/docs/podlet/local_development.

makes your integration pipeline run reliably. Investing some effort in good mocks will make your life easier and can save a lot of time down the road.

14.2.4 Pulling other teams micro frontends from staging or production

But in some cases, mocking is not sufficient. If you are trying to reproduce a mysterious bug, you might want to test with the real code.

If you’re doing client-side rendering, this can be easy. You don’t have to check out and build the other team’s code from scratch. Point the associated script and style tags to your staging or production environment and fetch the code for the other teams’ fragments from there. Now you can debug how your local code plays with the released code from the others.

Single-spa even goes a step further. They’ve built a tool called `single-spa-inspector` that lets you do it the other way around.⁴ You can open up a production page in the browser, and the inspector makes it possible to replace the released version of your code with your local development code. Single-spa uses `import-maps` to do the trick.

Pulling in fragments from a remote server is also possible with server-side rendering. There you’d advise your HTML assembly mechanism to fetch the markup for some routes directly from production. If you’re using Nginx and SSI, you can achieve this by changing the upstream configuration for the other teams to the production server but keeping your upstream pointing to localhost.

14.3 Testing

Automated testing has become the centerpiece of modern software development. Having good test coverage reduces the need for manual testing and enables you to adopt techniques like continuous delivery.

How does testing look in a micro frontends project? It’s not so different from testing in a monolithic project. Every team tests its application on different levels. They’ll have a bunch of fast-running unit and service tests and a couple of browser-based end-to-end tests.

You probably know about the testing pyramid.⁵ It describes that tests with a low level of integration (for example, unit tests) are cheap to write and run quickly. Tests with a high level of integration run slowly and are expensive to maintain. Figure 14.7 shows a variant of the classic testing pyramid.

In a micro frontends project, we can split the topmost category (UI or end-to-end tests) into two parts:

- 1 *Isolation (most tests)*—A team should perform the largest part of their user interface tests in an isolated environment without the code from other teams. These tests would run against a version of the software with mocked fragments. The

⁴ See <https://single-spa.js.org/docs/devtools>.

⁵ See Martin Fowler, “TestPyramid,” <https://martinfowler.com/bliki/TestPyramid.html>.

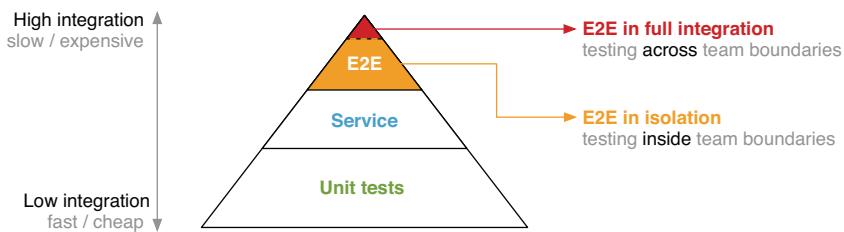


Figure 14.7 The testing pyramid shows that low-level tests are fast and cheap (bottom). Tests with a high level of integration, like browser-based end-to-end tests, are slow and expensive to maintain. In a micro frontends project, we can split the end-to-end test category (top) into types of tests: those that only run on one team's UI and those that run across team boundaries.

team's own fragments are tested in an isolated environment (sandbox), as shown in the previous section.

- 2 *Full integration (very few tests)*—Even if every team tests its fragments and pages accurately, there is a possibility of errors at the user interface boundaries. You should test critical transition points in full integration.

Full integration tests are hard to write because they require knowledge about the markup structure from at least two teams. We didn't have good experiences with introducing an overarching integration test-suite that runs against the complete software. All our attempts ended in brittle solutions with lots of false positives. Also, the question "Who owns the overarching integration tests?" is hard to answer if you don't want to introduce a horizontal testing team.

Instead, we go for a distributed approach. Every team can decide to test across the borders of their direct neighbors. Team Checkout could test if its Buy button micro frontend works when it's integrated on Team Decide's product page. Team Decide might check if Team Inspire's recommendation fragment is not empty.

Summary

- You can use the micro frontend's user interface integration techniques to test out this architecture with your existing project. These techniques also enable gradual migrations, where new micro frontends replace the old monolith user interface slice by slice.
- Replacing a current system slice by slice introduces low risk, because you have a working application at all times. However, mixing the new frontends with the monolith's frontend can be challenging due to leaking styles. Using Shadow DOM for the new micro frontends can help.
- If mixing user interfaces with the monolith doesn't work, the frontend-first or a greenfield approach are good alternatives, but they come with a higher risk.
- It's a good idea to disable code from other teams in your local development and testing environment. Eliminating foreign code reduces complexity and makes

the environment more stable. Creating simple mock micro frontends helps to get a more realistic impression of the layout.

- Mock micro frontends can be static placeholders, but they can also include simple functionality like emitting events.
- You can develop fragments on a dedicated sandbox page. It shows the fragment in isolation. This sandbox page can also contain some custom user interface to test communication (trigger events) or simulate changes in the environment (for example, change SKU).
- Nearly all your tests should run against your own team's code. Test in isolation where possible. In some cases, it might be necessary to test across team boundaries. A central testing team can be responsible for this. Another solution is that teams test the integration point to the neighboring teams themselves.

index

Numerics

404 error 180–181

A

absolute paths 206
absolute URLs 73, 206, 208, 210
activityFn 137
ad hoc server 26, 30, 122
add random product button 261
add-to-wishlist button 252–253
addEventListener 92
Agile Fluency Model 242
Ajax, composition via 42–51, 159
 benefits of 48
 declarative loading with h-include 47–48
 drawbacks of 49
 when to use 50
Akamai 73
Alignment option 196
alt attribute 73
Amazon Web Services (AWS) 245
AMD module 75
Android 19
Angular 14, 85, 135, 140, 153, 167, 197–198
Angular Elements 93
Angular Material 229
@angular/router 119
AngularJS (v1) 131
anonymous function 46
app shell
 anatomy of 122–123

APIs 133–134
defining 121
flat routing 120–127
 client-side routing 123–124
 contracts between app shell and teams 127
 page rendering 124–127
ownership of 141
two-level routing 128–134
 cleanup 131
 implementing 129
 implementing team-level routing 130–131
 URL changes 131–133
app.get method 79
appHistory.listen 124, 127
appHistory.push() function 124
application shell. *See* app shell
appShell.setTitle() method 140
Ara Framework 153
architecture
 choosing 165–169
 fast first-page load/progressive enhancement 167
 instant user feedback 167–168
 multiple micro frontends on one page 168–169
 soft navigation 168
 strong isolation 166–167
 comparing complexity 161–162
 composition techniques 158–159
 Ajax 159
 client-side integration 159
 iframe 159
 server-side integration 158

architecture (*continued*)

- heterogeneous architectures 162
- high-level architectures 159–161
 - linked pages 160
 - linked SPAs 160–161
 - linked universal SPAs 161
 - server routing 160
 - unified SPAs 161
 - unified universal SPA 161
- routing and page transitions 157–158
- sites vs. apps 162–165
 - Documents-to-Applications Continuum 163–164
 - server- vs. client-side rendering 164–165
- architecture-inherent issues 236
- architecture-level artifacts 196
- asset loading
 - asset referencing strategies 174–186
 - cache-busting and independent deployments 175–176
 - direct referencing 174–175
 - inlining 183
 - Podium 184–185
 - referencing via include 178–180
 - referencing via redirect 176–178
 - synchronizing markup and asset versions 180–183
 - Zalando Tailor 183–184
 - bundle granularity 186–188
 - all-in-one bundle 187
 - HTTP/2 186
 - page and fragment bundles 187–188
 - team bundles 187
 - on-demand loading 188–189
 - lazy loading CSS 189
 - proxy micro frontends 188–189
 - asynchronous loading 49, 112
 - attachShadow 93
 - attributeChangedCallback 92, 102
 - authentication 114
 - autonomous deployments 200
 - autonomy 16–17
 - cost of 196–197
 - full 241–242
 - self-contained fragments and pages 16
 - shared nothing architecture 17
 - technical overhead 17
 - AWS (Amazon Web Services) 245

B

- Babel 14, 208
- backend for frontends (BFF) 240

bare specifier 206, 208

- behavior-centric 163
- BFF (backend for frontends) 240
- Block URL feature 195
- blueprints 25
- boot time, unified SPAs 142
- bootstrap function 137
- bounded contexts 238
- Broadcast Channel API 111–112
- brownfield projects 255
- bundle granularity 186–188
 - all-in-one bundle 187
 - HTTP/2 186
 - page and fragment bundles 187–188
 - team bundles 187

C

-
- cache busting 174–176
 - cache invalidation strategy 175, 187
 - Cache-Control 176, 178
 - cacheability 173, 195
 - Calibre 193
 - canary deployments 181
 - CDN (content delivery network) 175, 182, 185
 - central design system team 246
 - central infrastructure team 246
 - central model 219
 - Central-to-Federated Continuum 221
 - change event 105
 - channel.postMessage 111
 - checkout-buy attributes 87, 108, 110
 - checkout-buy element 88–89, 103, 148
 - checkout-cart component 130
 - checkout-minicart 110
 - checkout-pages component 129–130
 - checkout-pay component 130
 - checkout-success component 124, 129–130
 - checkout:item_added event 104, 106, 108–109, 261
 - client-side composition
 - combining with server-side 147–153
 - contract between teams 152
 - SSI and Web Components 148–152
 - style isolation using Shadow DOM 93–96
 - creating shadow root 93–94
 - scoping styles 94–96
 - when to use 96
 - when to use 97–98
 - wrapping micro frontends using Web Components 86–93, 96–98
 - benefits of 96–97
 - Custom Elements 88–91

- drawbacks of 97
- parametrization via attributes 91–92
- process for 87–92
- Web Components as container format 88
- wrapping framework in Web
 - Components 92–93
- client-side integration 158
- client-side rendering 97, 164–165, 262
- client-side routing
 - APIs 133–134
 - app shell with flat routing 120–127
 - anatomy of app shell 122–123
 - client-side routing 123–124
 - contracts between app shell and teams 127
 - defining app shell 121
 - keeping URL and content in sync 124
 - mapping URLs to components 124
 - page rendering 124–127
 - app shell with two-level routing 128–134
 - cleanup 131
 - implementing team-level routing 130–131
 - implementing top-level router 129
 - URL changes 131–133
 - single-spa meta-framework 134–140
 - framework adapters 137–138
 - JavaScript modules as component format 137
 - navigating between micro frontends 138
 - nesting micro frontends 139–140
 - running application 139
 - unified SPAs 140–143
 - app shell ownership 141
 - boot time 142
 - communication 141–142
 - error boundaries 141
 - memory management 141
 - shared HTML document and meta data 140
 - single point of failure 141
 - when to use 142–143
 - closed mode 94
 - cloud hosting 244
 - code splitting 142, 188, 195, 199
 - CodePen.io site 164
 - common design system 12
 - CommonJS 174
 - communication
 - authentication 114
 - data replication 115–116
 - frontend-backend communication 115
 - global context 113–114
 - managing state 114
 - unified SPAs 141–142
 - user interface communication 100–112
 - fragment to fragment 107–111
 - fragment to parent 104–107
 - parent to fragment 101–104
 - publishing/subscribing with Broadcast Channel API 111–112
 - when to use 112
 - compatible composition technique 165
 - composition 11, 158–159
 - client-side 159
 - style isolation using Shadow DOM 93–96
 - when to use 97–98
 - wrapping micro frontends using Web Components 86–93, 96–98
 - server-side 158
 - benefits of 82–83
 - choosing a solution 81–82
 - drawbacks of 83
 - markup assembly performance 69–72
 - unreliable fragments 64–69
 - via Edge Side Includes 73
 - via Nginx and Server-Side Includes 60–64
 - via Podium 75–81
 - via Zalando Tailor 73–75
 - when to use 83–84
 - universal rendering and combining server- and client-side composition 147–153
 - when to use 153–155
 - via Ajax 42–51, 159
 - benefits of 48
 - declarative loading with h-include 47–48
 - drawbacks of 49
 - namespacing styles and scripts 45–47
 - process for 43–44
 - when to use 50
 - via iframe 33–36, 159
 - benefits of 35
 - drawbacks of 35–36
 - process for 34–35
 - composition technique 36
 - connectedCallback 92, 102, 125–127, 130, 137, 151
 - consistency 18, 215, 226
 - constructor method 92, 127, 137, 151
 - content delivery network (CDN) 175, 182, 185
 - content-centric 163
 - context information 113
 - contracts
 - between app shell and teams 127
 - combining with server- with client-side composition 152
 - page transitions via links 28–29
 - universal rendering 152

cookies 47
 CoP (community of practice) 243–244
 creativity 241
 critical path 64
 cross-functional teams 6–7, 244
 cross-team communication 100
 CSS (cascading style sheets)
 lazy loading 189
 pattern library 227–228
 CSS Modules 46, 189
 CSS-in-JS solutions 46, 96, 182, 189
 Custom Elements 49, 88, 91, 98, 125–126, 130,
 137, 148, 174, 188–189, 260
 defining 89–90
 using 90–91
 Custom Events 47, 49, 108, 112, 159
 emitting 105
 listening for 106–107
 customElements.define 89
 CustomEvent constructor 105
 CustomEvents API 105
 Cycle.js 135

D

data replication 115–116
 DAZN 19, 21
 decision making, local 15–16
 declarative loading 47–48
 decoupling 180
 deferred loading 71
 design system
 autonomous teams vs. 216–222
 benefits of 215–216
 buy-in from teams 218–219
 acceptance 218–219
 communication 219
 early stages 218
 central vs. federated process 219–220
 central model 219
 federated model 220
 development phases 221–222
 off-the-shelf vs. developing your own 216
 pattern library 226–234
 central and local 233–234
 central vs. local 231–233
 change 230–231
 component format 227–230
 costs of sharing components 231
 as process 217
 purpose and role of 215
 runtime integration vs. versioned
 packages 222–226

 runtime integration 222–224
 versioned packages 224–226
 sustained budget and responsibility 217–218
 disconnectedCallback() 92, 126–127, 131, 137
 DllPlugin 209
 DllReferencePlugin 204
 DLLs 204
 document flow 48
 Documents-to-Applications Continuum 163–164
 domain-driven design (DDD) 238
 Duet Design System 228
 dynamic route configuration 56–57

E

element.dispatchEvent 110
 Elm language 15
 error boundaries 141
 error handling, flexible 48
 ES Modules 174, 207, 209
 ES6 class 89
 ESI (Edge Side Includes), server-side composition via 73
 fallbacks 73
 time to first byte 73
 timeouts 73
 ETag header 178
 events
 asynchronous loading vs. 112
 Custom Events
 emitting 105
 listening for 106–107
 dispatching directly on window 110–111
 event bus via browser events 109–110

F

fail_timeout option 68
 fallbacks, server-side composition
 via Edge Side Includes 73
 via Nginx and Server-Side Includes 67–69
 via Podium 79–81
 via Zalando Tailor 74
 federated model 219–220
 fetch() function 47–48, 78
 findComponentName 129
 Fluent UI Design System 215
 fragment.css file 43, 90, 93–94, 177
 fragment.js file 90, 109, 177
 fragments 9–10, 158
 bundle granularity 187–188
 in isolation 260–262
 development page 260–261

independence through mocks 261–262
simulating interactions 261
integrating using SSI 63
mocking 259–262
self-contained 16
unreliable 64–69
 fallback content 68–69
 flaky fragments 65–66
 integrating Near You fragment 66–67
 timeouts and fallbacks 67–68
user interface communication
 fragment to fragment 107–111
 fragment to parent 104–107
 parent to fragment 101–104
framework adapters, single-spa meta-framework 137–138
framework-agnostic components 228–229
framework-specific components 228
frontend first migration approach 255–256
 benefits and challenges of 256
 process for 256
frontend integration 10–11
 communication 11
 composition 11
 routing and page transitions 10–11
frontend-backend communication 115
full-stack teams 241

G

github-elements 88
global context 113–114
greenfield and big bang migration approach 256–258
 benefits and challenges of 257–258
 process for 257

H

h-include library 47–48
heterogeneity 18
heterogeneous architectures 162
high-level architectures 159–161, 165
 linked pages 160
 linked SPAs 160–161
 linked universal SPAs 161
 server routing 160
 unified SPAs 161
 unified universal SPA 161
history library 123
Homepage.svelte component 138
HTTP/2, bundle granularity 186

I

iframe
 composition via 33–36, 159
 benefits of 35
 drawbacks of 35–36
 process for 34–35
 when to use 36
IIFE (immediately invoked function expression) 46
import-maps 210–211, 262
import() function 136, 174
include directive 61, 71
Inlining 183
innerHTML 92, 103, 124, 130
isolation
 fragments in 260–262
 development page 260–261
 independence through mocks 261–262
 simulating interactions 261
isolating JavaScript 46–47
isolating styles 45–46
isolating styles using Shadow DOM 93–96
 creating shadow root 93–94
 scoping styles 94–96
missing from Ajax 49
slowdowns 194–195
strong 166–167
item_added event 107

J

JavaScript
 isolating 46–47
single-spa meta-framework 137

L

layout library 75–76
lazy loading 189
legacy systems 14–15
lifecycle methods 97, 229
Light DOM 96
Link header 75
link tag 44–45, 174, 178–179, 183–184
linked pages 160
linked single-page applications (SPAs) 119, 160–161
linked universal single-page applications (SPAs) 161
links
 page transitions via 27–33
 benefits of 32

- links (*continued*)
 - contract between teams 28–29
 - data ownership 28
 - dealing with changing URLs 32
 - drawbacks of 32–33
 - process for 29–32
 - product page markup 30–31
 - starting applications 31–32
 - styles 31
 - when to use 33
- listen feature 123
- lit-html 198
- live-reload 261
- loading
 - asset
 - asset referencing strategies 174–186
 - bundle granularity 186–188
 - on-demand loading 188–189
 - asynchronous 49, 112
 - declarative 47–48
 - deferred 71
 - lazy 189
 - on-demand 188–189
 - parallel 69
- loadingFn 136
- local development 258–262
 - fragments in isolation 260–262
 - development page 260–261
 - independence through mocks 261–262
 - simulating interactions 261
- mocking fragments 259–260
- not running other team's code 258
- pulling other teams' micro frontends from staging or production 262
- lock-step deployment 200
- loose coupling 23, 32, 34
-
- M**
- manifest.json file 76, 78–79, 81, 184, 201, 204–205
- markup
 - assembly performance 69–72
 - deferred loading 71
 - nested fragments 70–71
 - parallel loading 69
 - time to first byte and streaming 71–72
 - fragment 43–44
 - product page 30–31
 - synchronizing markup and asset versions 180–183
- Material Design 216, 229
- Material UI (React) 229
- max_fails option 68
- maxwait attribute 73
- memory management, unified SPAs 141
- mfserve library 176
- micro frontends 4–12
 - downsides of 17–19
 - consistency 18
 - heterogeneity 18
 - more frontend code 19
 - redundancy 17–18
 - frontend 7–10
 - fragments 9–10
 - page ownership 8–9
 - frontend integration 10–11
 - communication 11
 - composition 11
 - routing and page transitions 10–11
 - nesting 139–140
 - problems solved by 12–17
 - adapting to new technology 14–16
 - autonomy 16–17
 - avoiding frontend monolith 13–14
 - optimization for feature development 12–13
 - productivity vs. overhead 20
 - organizational complexity 20
 - setup 20
 - proxy 188–189
 - shared topics 11–12
 - design systems 12
 - sharing knowledge 12
 - web performance 12
 - software systems and teams 4–7
 - cross-functional teams 6–7
 - team missions 6
 - when not to use 20–21
 - when to use 19–21
 - medium-to-large projects 19
 - on the web 19–20
 - who uses 21
- @microfrontends/serve package 27
- migration 252–258
 - frontend first approach 255–256
 - benefits and challenges of 256
 - process for 256
 - greenfield and big bang approach 256–258
 - benefits and challenges of 257–258
 - process for 257
 - proof of concept 252–253
 - real world example 252–253
 - role model 253
 - slice-by-slice approach 254–255
 - benefits and challenges of 254–255
 - process for 254

module specifier 206
MongoDB database 24
monolith
 avoiding frontend monolith 13–14
 native monolith 19–20
monorepos 258
mount function 137–138
multiple framework components 229–230
MutationObserver 49

N

namespacing 45–47
 resources 55
 scripts 46–47
 styles 45–46
navigate click handler 138
Near You fragment 66–68
nested fragments 70–71
nesting micro frontends 139–140
Nginx
 installing locally 53
 server-side composition via 60–64
 better load times 63–64
 process for 61–63
 server-side routing via 51–58
 infrastructure ownership 57–58
 namespacing resources 55
 process for 53–55
 route configuration methods 56–57
 when to use 58
No constraints option 196
no-cache setting 176
node_modules 208
node-tailor package 74
Node.js library 26, 74–75, 78
NPM package 201, 203
npm run script 66, 180, 204, 209

O

OAuth standard 114
on-demand loading 135, 174, 188–189, 195, 199, 205
 lazy loading CSS 189
 proxy micro frontends 188–189
one-year cache header 175
open mode 94
opening hours concept 219
optimization
 for feature development 12–13
 for use cases 195–196

overhead
 performance 35
 productivity vs. 20
 organizational complexity 20
 setup 20
 technical overhead 17
ownership
 of app shell 141
 of data 28
 of infrastructure 57–58
 by one product team 245–246
ownership concept 47, 242

P

page rendering
 app shell with flat routing 124–127
 linking between micro frontends 126–127
page transitions 10–11
 via links 27–33, 158
 benefits of 32
 contract between teams 28–29
 data ownership 28
 dealing with changing URLs 32
 drawbacks of 32–33
 process for 29–32
 product page markup 30–31
 starting applications 31–32
 styles 31
pages
 bundle granularity 187–188
 composition 11
 examining existing page structures to identify team boundaries 239–240
 linked 160
 ownership of 8–9
 self-contained 16
 parallel loading 69
 parametrization, via attributes 91–92
pattern library 215, 226–234
 central and local 233–234
 central vs. local 231–233
 component complexity 232
 domain specific 233
 reuse value 232
 trust in teams 233
 change 230–231
 being open for change 230
 keep it simple 230–231
 component format 227–230
 common templating language 229–230
 framework-agnostic components 228–229
 framework-specific components 228

pattern library (*continued*)
 multiple framework components 229
 pure CSS 227–228
 costs of sharing components 231
 Pavlovian reflex 196
 performance
 architecting for 191–196
 attributing slowdowns 193–195
 benefits of micro frontends 195–196
 different teams, different metrics 191–192
 multi-team performance budgets 192–193
 vendor libraries 196–211
 cost of autonomy 196–197
 one global version 199–200
 pick small 197–199
 sharing business code 211
 versioned vendor bundles 200–211

platinum option 102

Podium
 asset loading 184–185
 server-side composition via 75–81
 architecture 76–77
 fallbacks and timeouts 79–81
 Implementation 77–79
 Podlet manifest 75–76

@podium/* libraries 81
@podium/layout 76
@podium/podlet 76
Podlet manifest 75–76
podlets 75–76, 78, 81, 184
productivity, overhead vs. 20
 organizational complexity 20
 setup 20
progressive enhancement 48, 83, 145, 152, 167
Prototype.js file 14
proxy micro frontends 188–189
proxy_read_timeout property 67
push feature 123

R

react 206–208, 210
react-dom 206–207
react-router 123, 139
ReactDOM.hydrate 150
ReactDOMServer.renderToString 150
recos 78–79, 81
redundancy 17–18, 31, 183, 187, 191, 226
referencing
 via include (server) 178–180
 via redirect (client) 176–178
register_style endpoint 179
registration file 177

request forwarding 62
require.js module loader 184
RequireJS 174
REST API 20
reusable interface components 214
robustness, of links 32
rolling deployments 181
rollup.js (Central ES modules) 205–210
 creating versioned bundle 207–208
 using versioned bundle 208–210
routes object 124, 129
routing 10–11
 client-side
 APIs 133–134
 app shell 158
 app shell with flat routing 120–127
 app shell with two-level routing 128–134
 single-spa meta-framework 134–140
 unified SPAs 140–143
 server-side, via Nginx 51–58
 infrastructure ownership 57–58
 namespacing resources 55
 process for 53–55
 route configuration methods 56–57
 when to use 58
runtime integration 222–224

S

SaaS (Software as a Service) 245
scoped attribute 45
Scoped CSS 45
scoping styles 94–96
script tag 46, 75, 174, 178, 183, 199, 204, 259
scripts
 isolating JavaScript 46–47
 no lifecycle for 49
search engines
 composition via Ajax 48
 iframes and 35–36
security features 35
Semantic UI 216
SEO (search engine optimization) 35–36
server rendered pages 37
server requests, composition via Ajax 49
server routing 160
server-side composition 11, 253
 benefits of 82–83
 choosing a solution 81–82
 combining with client-side 147–153
 contract between teams 152
 SSI and Web Components 148–152
 drawbacks of 83

- markup assembly performance 69–72
 - deferred loading 71
 - nested fragments 70–71
 - parallel loading 69
 - time to first byte and streaming 71–72
 - universal rendering with pure 153
 - unreliable fragments 64–69
 - fallback content 68–69
 - flaky fragments 65–66
 - integrating Near You fragment 66–67
 - timeouts and fallbacks 67–68
 - via Edge Side Includes 73
 - fallbacks 73
 - time to first byte 73
 - timeouts 73
 - via Nginx and Server-Side Includes 60–64
 - better load times 63–64
 - process for 61–63
 - via Podium 75–81
 - architecture 76–77
 - fallbacks and timeouts 79–81
 - Implementation 77–79
 - Podlet manifest 75–76
 - via Zalando Tailor 73–75
 - asset handling 75
 - fallbacks and timeouts 74
 - time to first byte and streaming 75
 - when to use 83–84
 - Server-Side Includes. *See SSI*
 - server-side integration 97, 158, 183
 - server-side rendering (SSR) 145–146, 262
 - server-side rendering, client-side vs. 164–165
 - server-side routing, via Nginx 51–58
 - infrastructure ownership 57–58
 - namespacing resources 55
 - process for 53–55
 - route configuration methods 56–57
 - when to use 58
 - session storage 47
 - Shadow DOM 88, 96, 149, 255
 - style isolation using 93–96
 - creating shadow root 93–94
 - scoping styles 94–96
 - when to use 96
 - shadowRoot property 93–95
 - shared frontend blueprint 247
 - shared integration technique 253
 - shared nothing architecture 17
 - shared topics 11–12
 - design systems 12
 - sharing knowledge 12
 - web performance 12
 - shared-vendor folder 202–203, 206
 - simulating interactions 261
 - single point of failure 223
 - single-page applications. *See SPAs*
 - single-spa meta-framework 11, 120, 134–140
 - framework adapters 137–138
 - JavaScript modules as component format 137
 - navigating between micro frontends 138
 - nesting micro frontends 139–140
 - running application 139
 - single-spa-inspector 262
 - single-spa-leaked-globals plugin 141
 - single-spa-svelte 138
 - single-spa.js library 136
 - singleSpa.registerApplication function 136
 - singleSpaSvelte function 138
 - sitespeed.io 193
 - Skate.js 149
 - skeleton screens 167
 - slice-by-slice migration approach 254–255
 - benefits and challenges of 254–255
 - process for 254
 - slowdowns 193–195
 - isolation 194–195
 - observability 194
 - soft navigation 168
 - Some constraints option 196
 - spa meta-framework 174
 - SPAs (single-page applications)
 - linked 160–161
 - linked universal 161
 - single-spa meta-framework 134–140
 - framework adapters 137–138
 - JavaScript modules as component format 137
 - navigating between micro frontends 138
 - nesting micro frontends 139–140
 - running application 139
 - unified 140–143, 161
 - app shell ownership 141
 - boot time 142
 - communication 141–142
 - error boundaries 141
 - memory management 141
 - shared HTML document and meta data 140
 - single point of failure 141
 - when to use 142–143
 - universal unified 154–155, 161
 - specialist teams 6
 - specialized component team 246
 - SSI (Server-Side Includes)
 - server-side composition via 60–64
 - better load times 63–64
 - process for 61–63
 - universal rendering 148–152

SSR (server-side rendering) 145–146, 262
 state management 114
 sticky sessions 182
 Strangler Fig Pattern 254
 streaming templates
 via Nginx and Server-Side Includes 71–72
 via Zalando Tailor 75
 stub parameter 68–69
 style guide 215
 styles
 isolating 45–46, 93–96
 links 31
 namespacing 45–46
 synchronization 178, 182
 SystemJS 210

T

Tailor 72–75, 79, 81, 83, 158, 186
 teams
 architecting for performance
 different teams, different metrics
 191–192
 multi-team performance budgets 192–193
 bundle granularity 187
 central vs. local pattern libraries 233
 contract between teams 28–29
 cross-cutting concerns 245–247
 central infrastructure 245–246
 global agreements and conventions
 246–247
 specialized component team 246
 cross-functional 6–7
 cultural change 242–243
 depth of 240–242
 frontend only 240–241
 full autonomy 241–242
 full-stack team 241
 design system vs. 216–222
 identifying boundaries 238–240
 domain-driven design 238
 existing page structures 239–240
 user-centered design 239
 missions 6
 multiple frontends per team 20
 not running other team's code 258
 pulling other teams' micro frontends from
 staging or production 262
 sharing knowledge 243–245
 community of practice 243–244
 learning and enabling 244–245
 presenting your work 245
 software systems and 4–7

technology diversity 247–249
 frontend blueprint 247–248
 not fearing the copy 248–249
 toolbox and defaults 247
 value of similarity 249
 technology diversity 247–249
 frontend blueprint 247–248
 making optional 248
 project-specific aspects 248
 not fearing the copy 248–249
 toolbox and defaults 247
 value of similarity 249
 templating language 229–230
 test-suite 263
 testing 262–263
 tiered design systems 233
 time to first byte (TTFB) 69
 time to first byte, server-side composition
 via Edge Side Includes 73
 via Nginx and Server-Side Includes 71–72
 via Zalando Tailor 75
 timeouts, server-side composition
 via Edge Side Includes 73
 via Nginx and Server-Side Includes 67–68
 via Podium 79–81
 via Zalando Tailor 74
 Tractor Store website 24–27
 example code 25–27
 directory structure 26
 installing dependencies 26
 Node.js 26
 starting examples 26–27
 freedom of choosing technology 24–25
 independent deploys 25
 TTFB (time to first byte) 69

U

UI communication 100
 UIengine 234
 unavoidable globals 47
 unidirectional dataflow 104
 unified single-page apps (SPAs) 140–143, 161
 app shell ownership 141
 boot time 142
 communication 141–142
 error boundaries 141
 memory management 141
 shared HTML document and meta data
 140
 single point of failure 141
 when to use 142–143

universal application shell 154
universal rendering
 combining server- and client-side composition 147–153
 contract between teams 152
 SSI and Web Components 148–152
when to use 153–155
 increased complexity 154
 universal rendering with pure server-side composition 153
universal unified SPAs 154–155
universal unified single-page apps (SPAs) 154–155, 161
unlisten() function 131
unmount function 137–138
URLs
 changes 131–133
 first page view 132
 inside team navigation 132
 inter-team navigation 133
 contract between teams 28–29
 dealing with changing 32
 keeping in sync wth content 124
 mapping to components 124
 route configuration 56
usage pattern 168
user feedback, instant 167–168
user interface communication 100–112
 fragment to fragment 107–111
 dispatching events directly on window 110–111
 event bus via browser events 109–110
 fragment to parent 104–107
 emitting Custom Events 105
 listening for Custom Events 106–107
 parent to fragment 101–104
 platinum option 102
 updating on attribute change 102–104
publishing/subscribing with Broadcast Channel API 111–112
when to use 112
 bad boundaries 112
 events vs. asynchronous loading 112
 simple payloads 112
user interface, design system for
 as process 217
 autonomous teams vs. 216–222
 benefits of 215–216
 buy-in from teams 218–219
 central vs. federated process 219–220
 development phases 221–222
 off-the-shelf vs. developing your own 216

pattern library 226–234
purpose and role of 215
runtime integration vs. versioned packages 222–226
sustained budget and responsibility 217–218
user-centered design 239
user-focused culture 242
user-interface integration techniques 252

V

vendor libraries 196–211
 cost of autonomy 196–197
 one global version 199–200
 pick small 197–199
 sharing business code 211
versioned vendor bundles 200–211
 Central ES modules (rollup.js) 205–210
 import-maps 210–211
 Webpack DllPlugin 201–205
versioned bundles 201, 206
versioned packages 224–226
 avoiding shipping unused code 224
 drawbacks of 226
 independent upgrades 224
 self-contained 225
vue-router 119, 123, 139
Vue.js 14, 85, 93, 135, 153, 200
@vue/web-component-wrapper package 93
Vuetify 228

W

Web Components 86–93, 96–98, 159
 as container format 88
 benefits of 96–97
 Custom Elements 88
 defining 89–90
 using 90–91
 drawbacks of 97
 parametrization via attributes 91–92
 universal rendering 148–152
 wrapping framework 92–93
 wrapping micro frontends 87–92
Webpack DllPlugin 201–205
 creating versioned bundle 202–203
 using versioned bundle 203–205
window object 46, 110, 199
window.customElements.define function 89
window.dispatchEvent 110
window.history.pushState 138
window.postMessage API 159

window.React 199
window.ReactDOM 199
wrapping micro frontends, using Web Components 86–93, 96–98
benefits of 96–97
defining Custom Elements 89–90
drawbacks of 97
parametrization via attributes 91–92
process for 87–92
using Custom Elements 90–91
Web Components and Custom Elements 88
Web Components as container format 88

wrapping framework in Web Components 92–93

Z

Zalando Tailor
asset loading 183–184
server-side composition via 73–75
asset handling 75
fallbacks and timeouts 74
time to first byte and streaming 75
zombie style guide 218

Micro Frontends IN ACTION

Michael Geers

Micro frontends deliver the same flexibility and maintainability to browser-based applications that microservices provide for backend systems. You design your project as a set of standalone components that include their own interfaces, logic, and storage. Then you develop these mini-applications independently and compose them in the browser.

Micro Frontends in Action teaches you to apply the microservices approach to the frontend. You'll start with the core micro frontend design ideas. Then, you'll build an e-commerce application, working through practical issues like server-side and client-side composition, routing, and maintaining a consistent look and feel. Finally, you'll explore team workflow patterns that maximize the benefit of developing application components independently.

What's Inside

- Create a unified frontend from independent applications
- Combine JavaScript code from multiple frameworks
- Browser and server-side composition and routing
- Implement effective dev teams and project workflow

For web developers, software architects, and team leaders.

Michael Geers is a software developer specializing in building user interfaces.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
www.manning.com/books/micro-frontends-in-action

“You can base your Micro Frontends directly on the examples in the book!”

—Barnaby Norman
Frontend Developer

“Expert practical advice for robust frontend development.

The final piece of the micro puzzle!”

—Adail Retamal
Cubic Transportation Systems

“A very concise and logically structured introduction to implementing a micro frontends team infrastructure.”

—Tanya Wilke, Sanlam

“A definitive guide for applying micro frontends architecture to any existing UI technology.”

—Karthikeyaran Rajendran
Johnson Controls India



ISBN: 978-1-61729-687-1



9 781617 296871



MANNING

\$49.99 / Can \$65.99 [INCLUDING eBook]