AN OPINIONATED INTRODUCTION TO

MICRO-FRONTENDS

with

VUE, REACT, SVELTE

DAMIANO FUSCO

An Opinionated Introduction to Micro-frontends

with React, Vue, and Svelte

Damiano Fusco

This book is available at http://leanpub.com/introduction-to-micro-frontends

This version was published on 2024-11-28

Contents

AN OPINIONATED INTRODUCTION TO MICRO-FRONTENDS

Preface

Micro-frontends are a fascinating architectural approach that enables teams to work on different parts of a frontend application independently. This architectural style can be particularly beneficial in organizations with multiple teams focusing on different features of the same application. In this book, we will explore some of the main concepts, benefits, and downsides of different patterns and architectural designs. We then implement a custom solution using a step-by-step approach, allowing us to make informed choices as we go and better understand the challenges we face.

Goal

The primary aim of this book is to guide you through the process of building a scalable micro-frontend architecture using effective project structure, file organization, naming conventions, state management, and type checking with TypeScript. We will also explore compositional approaches using hooks.

Throughout the chapters, we will develop our project into a robust micro-frontend foundation that is easy to expand and maintain. This foundation will showcase how patterns, conventions, and strategies can lay a solid groundwork, keeping the code organized and uncluttered.

IMPORTANT: Initially, we will write code that allows us to achieve the desired functionality quickly, even if it requires more code. However, we will constantly "rework" it (**refactoring**) to improve efficiency and find solutions that reduce the code footprint or organize it in a way that is clear, easy to expand, and maintain. Therefore, arm yourself with patience and prepare to delve deeply into the iterative process of software development!

About Me

I have been a software developer for over 20 years, transitioning from a full-time musician at the age of 30 to a graphic designer, and then to a web designer as the internet began to proliferate. Over the years, I have worked as a full-stack developer, utilizing technologies such as Microsoft .NET, JavaScript, Node.js, and many others. You can learn more about my journey and experiences on my personal website https://www.damianofusco.com[1] and

[1] https://www.damianofusco.com/

LinkedIn profile https://www.linkedin.com/in/damianofusco/. You can also find me on Twitter (@damianome[2]) and GitHub (github.com/damianof[3]).

Audience

This book is intended for developers ranging from beginners with some experience in **MV*** applications to intermediate developers. The format is akin to a cookbook; however, instead of individual recipes, we'll go through creating a project and continue enhancing, refactoring, and improving it as we advance to more sophisticated chapters to showcase different patterns, architectures, and technologies.

Text Conventions

Throughout this book, I will highlight most terms or names in **bold**. This choice is made to simplify the formatting and focus on content clarity, whether discussing code, directory names, or other key concepts.

[2]https://twitter.com/damianome
[3]https://www.github.com/damianof

Prerequisites

This book assumes that you are familiar with the **terminal** (**command prompt** on Windows), have already worked with the **Node.js** and **NPM** (**Node Package Manager**), know how to install packages, and are familiar with the **package.json** file.

It also assumes you are familiar with **JavaScript**, **HTML**, **CSS** and in particular with **HTML DOM** elements properties and events.

It will also help if you have some preliminary knowledge of **TypeScript**[1] as we won't get into details about the language itself or all of its features but mostly illustrate how to enforce type checking at development time with it.

You will need a text editor like **VS Code** or **Sublime Text**, better if you have extensions/plugins installed that can help specifically for Vue/React/Svelte code. For VS Code for example, you could use extensions like **Volar**[2] (just search for it within the VS code extensions tab). There are also similar extensions for React and Svelte.

[1] https://www.typescriptlang.org
[2] https://marketplace.visualstudio.com/items?itemName=vue.volar

Companion Code

Chapter 1 - Definitions and Architecture Overview

Micro-frontends extend the principles of microservices to the frontend development realm. This approach involves segmenting a frontend application into smaller, more manageable pieces that can be developed, deployed, and managed independently. Each segment, known as a “micro-frontend,” is tasked with handling a distinct feature or component of the user interface and operates autonomously within its domain.

Key Benefits

- **Decentralization and Team Autonomy**: Micro-frontends enable different teams to work independently on their segments of the application without needing to coordinate closely with others. This decentralization reduces bottlenecks and accelerates the speed of development, empowering teams to implement and iterate on features more quickly.
- **Scalability**: As applications grow, managing one giant codebase can become unwieldy. Micro-frontends make scaling more manageable by allowing teams to independently scale parts of the application as needed, rather than scaling the entire application monolithically.
- **Technological Freedom**: Teams can choose the technology stack that best suits their needs, enabling them to adopt new frameworks and tools without the risk of disrupting other parts of the application. This flexibility also facilitates the exploration of cutting-edge technologies.
- **Easier Upgrades and Updates**: Updating older technologies or applying new features can be done incrementally, one micro-frontend at a time. This modular approach reduces the risks associated with deploying changes and allows for continuous improvement without large-scale system downtime.

Common Challenges

- **Integration Complexity**: While micro-frontends can be developed independently, integrating them into a cohesive whole can introduce complexity. Challenges often arise in maintaining consistent styling, behavior, and functionality across the application, requiring robust integration strategies and tools.

- **Performance Concerns**: Incorrect implementations can lead to performance bottlenecks, such as multiple micro-frontends loading the same library multiple times. This redundancy increases the payload size and can lead to less effective code-sharing and increased load times.
- **Operational Complexity**: Each micro-frontend might have its deployment pipeline, monitoring, and performance metrics, potentially increasing the operational overhead. Coordinating these multiple pipelines requires careful management to ensure system stability and efficiency.

Architecture Overview

- **Composition**: Micro-frontends can be composed either in the browser or on the server. Browser-side composition might involve client-side routing and dynamic module loading, enabling on-demand content retrieval and interactive user experiences. Server-side composition might leverage techniques like server-side includes (SSI) or edge-side includes (ESI), which assemble the application at the server level before sending it to the client, improving initial load times and SEO.
- **Communication**: Effective communication strategies are crucial for micro-frontends to function harmoniously. This might involve shared state management through global stores or local storage, event-based communication using custom events or message buses, and API calls orchestrated through a backend for frontend (BFF) layer.
- **Routing**: Routing must be carefully managed to ensure a seamless user experience. Micro-frontends often require sophisticated routing mechanisms that can dynamically load different parts of the application as users navigate, without refreshing the entire page.
- **Deployment**: Each micro-frontend can be independently deployed, which allows for greater flexibility and reduces the risk associated with deploying large changes. Deployment strategies may include using CDNs for static assets, containerization for consistent environments, and automated pipelines for continuous integration and delivery.

Overview of Possible Architecture Solutions

There are many different ways to implement a micro-frontend architecture, such as module federation, service-oriented architecture, source maps, and dynamic loading. Each approach offers unique benefits and comes with its own set of challenges. Here’s a detailed look at the main ones.

Module Federation

Definition and Purpose: Module Federation, a feature of Webpack 5, allows JavaScript applications to dynamically load code from another application at runtime. It enables different builds to share dependencies, libraries, or even entire components seamlessly as if they were part of a single cohesive application. This capability is particularly useful in micro-frontend architectures where different parts of a frontend may be developed by separate teams and need to interact or share functionality.

How It Works:

- **Hosts and Remotes**: The application consuming modules from another is termed the "host," while the application exposing modules is known as the "remote."
- **Dynamic Imports**: Leverages dynamic imports to load only necessary pieces of an application, reducing initial load times and improving performance.
- **Shared Dependencies**: Manages shared dependencies intelligently, ensuring that libraries like React or Vue do not get loaded multiple times across micro-frontends if they use the same versions.

Benefits:

- Allows incremental upgrades by decoupling deployments.
- Enables different teams to develop independently with their preferred tools and frameworks.
- Reduces load times by sharing vendor libraries across various parts of the application.

Cons:

- **Complex Configuration**: Requires careful setup, especially concerning dependency management and version alignment.
- **Tight Coupling Risk**: If not properly architected, it can lead to tight coupling between independent micro-frontends.
- **Performance Overheads**: Can introduce performance overheads when resolving dependencies at runtime.
- **Limited Tooling Support**: As a relatively new feature, not all toolchains or frameworks fully support or optimize for Module Federation.

Service-Oriented Architecture (SOA)

Definition and Relevance to Frontend: Traditionally associated with backend services, Service-Oriented Architecture principles can be effectively applied to frontend development. In the context of micro-frontends, SOA involves treating each micro-frontend as a service that provides specific functionality through a well-defined interface.

Key Principles:

- **Loose Coupling**: Each micro-frontend is independent, interacting through defined APIs or events rather than direct code dependencies.
- **Service Contract**: Interfaces between micro-frontends are governed by contracts that specify exposed functionalities and access methods.
- **Reusability**: Aims to design components and utilities that are reusable across different parts of the application.

Benefits:

- Enhances flexibility in development and deployment.
- Improves maintainability by isolating features and concerns.
- Facilitates better testing environments by allowing each service to be tested independently.

Cons:

- **Increased Complexity**: Applying SOA to frontend development increases architectural and operational complexity.
- **Overhead of Remote Calls**: Remote calls between services can introduce latency and degrade user experience.
- **Challenges in Global State Management**: Managing a consistent global state across services can be challenging.
- **Potential for Redundant Development**: Without careful planning, different teams may develop similar functionalities independently.

Import Maps

Definition and Purpose: Import Maps enable web applications to control the URLs from which JavaScript modules are loaded using a simple JSON object. This method simplifies module resolution and allows developers to map import statements to specific URLs directly in the browser, sidestepping the complexity associated with traditional module bundlers or loaders.

How It Works:

- **JSON Configuration**: Developers define a JSON object that maps package names to URLs, which tells the browser exactly where to load modules from when specific import statements are used in the code.
- **Browser-Side Resolution**: The browser interprets this map during the HTML parsing phase, modifying the module loader to understand and resolve the specified import mappings directly.
- **Decoupling from Build Tools**: By handling module resolution in the browser, Import Maps reduce reliance on build-time tooling for module federation and dependency management.

Benefits:

- **Simplifies Dependency Management**: Reduces the need for complex build configurations and tooling by handling dependency resolution natively in the browser.
- **Enhances Security**: Allows developers to explicitly define trusted sources for modules, reducing the risk of loading malicious or unintended code.
- **Improves Performance**: Potentially decreases the amount of JavaScript required to bootstrap applications by eliminating unnecessary module resolution logic and network round-trips that traditional bundlers might incur.
- **Facilitates Agile Development**: Makes it easier to update, version, and deploy modules independently, supporting a more agile and modular development process.

Cons:

- **Limited Browser Support**: Currently, Import Maps are only supported in a few modern browsers, which can limit their use in environments requiring broad compatibility.
- **Lack of Tooling and Ecosystem Support**: As a relatively new technology, the ecosystem around Import Maps is not as mature, which may pose challenges in areas like debugging, testing, and integration with existing tools.

- **Overhead in Management**: While Import Maps reduce build complexity, they require careful management of the mappings themselves, which can become cumbersome as applications grow and dependencies increase.
- **Potential for Misconfiguration**: Errors in the Import Map configuration can lead to failed module resolutions, which might be harder to diagnose and fix compared to traditional server-side or build-time errors.

Dynamic Loading using a Custom Solution

Definition: Dynamic loading refers to the capability of a web application to load JavaScript modules, components, or entire applications on-demand, rather than loading everything upfront. This feature is crucial for improving web performance and is integral to modern web development environments, including micro-frontends.

Implementation:

- **JavaScript Modules**: Uses ES Modules to import functionalities as needed.
- **Code Splitting**: Employs tools like Webpack, Rollup, or Vite to split application bundles into smaller chunks that can be dynamically loaded based on user interactions or other triggers.
- **Lazy Loading**: Ensures components or routes are loaded only when required, significantly reducing initial load time and resource consumption.

Benefits:

- Decreases initial load time by loading only necessary resources.
- Reduces bandwidth usage and enhances user experience.
- Enables scaling of large applications by avoiding monolithic bundle sizes.

Cons:

- **Code Splitting Complexity**: Requires thoughtful planning to balance chunk sizes and load times.
- **Initial Configuration and Maintenance**: Setting up dynamic loading involves complex initial configuration and ongoing maintenance challenges.
- **Flash of Unstyled Content**: May result in a flash of unstyled content if dynamically loaded components render with a delay.
- **Browser Support and Fallbacks**: Not all browsers support dynamic imports and advanced code splitting, requiring additional tooling or fallback mechanisms.

Summary

Comparison of Architecture Solutions

When choosing an architecture solution for micro-frontends, it's essential to consider how each option aligns with the project's specific requirements, such as scalability, maintainability, and ease of development. Here's a detailed comparison of the three main architecture solutions discussed:

Module Federation vs. Import Maps vs. Custom Dynamic Loading

1. **Module Federation**
   - **Pros**: Allows seamless sharing of dependencies and components across different micro-frontends, promoting reusability and reducing redundancy. It supports incremental updates and independent deployment, which can accelerate development cycles and minimize risks.
   - **Cons**: The configuration can be complex, requiring careful management of dependencies and version control. It also poses a risk of creating tightly coupled systems if not architected carefully, potentially complicating future changes.
2. **Import Maps**
   - **Pros**: Simplifies the management of module resolution at runtime directly within the browser, offering an efficient way to handle dependencies without needing extensive build configurations. Enhances the security and integrity of applications by specifying exact locations from where modules are loaded.
   - **Cons**: Currently has limited browser support and requires additional fallback mechanisms for broader compatibility. The maintenance of import maps can become complex as the number of dependencies grows, and misconfiguration can lead to module resolution failures.
3. **Dynamic Loading using a Custom Solution**
   - **Pros**: Offers tailored solutions that are optimized for specific project needs, allowing developers to design loading strategies that best fit their application's requirements. This method can significantly improve the application's load time and user experience by loading only the necessary resources on demand.
   - **Cons**: Requires a more hands-on approach to setup and maintenance, which can increase the complexity of the build process. Developers must also ensure compatibility across browsers and handle potential issues with flash of unstyled content.

Choosing the Right Solution

The choice between Module Federation, Import Maps, and Custom Dynamic Loading depends on several factors:

- **Project Size and Complexity**: For larger projects with multiple teams, Module Federation might be advantageous for its modular and scalable nature. In contrast, smaller projects might benefit from the straightforwardness of Import Maps or Custom Dynamic Loading.
- **Development and Debugging Needs**: Import Maps simplify dependency management but may not support complex debugging needs. Module Federation allows for easy debugging of shared modules, while Custom Dynamic Loading is best when performance and user experience are priorities.
- **Security and Performance**: Import Maps offer clear security benefits by controlling where modules are loaded from, but managing them can be challenging. Module Federation and Custom Dynamic Loading require careful consideration of performance impacts, particularly regarding how resources are loaded and managed.

Ultimately, the architecture solution should align with the project's long-term goals, considering how it will evolve and scale over time. In many cases, a hybrid approach that combines elements from several solutions might provide the best balance between flexibility, performance, and maintainability.

Chapter 2 - Sample Project Architecture

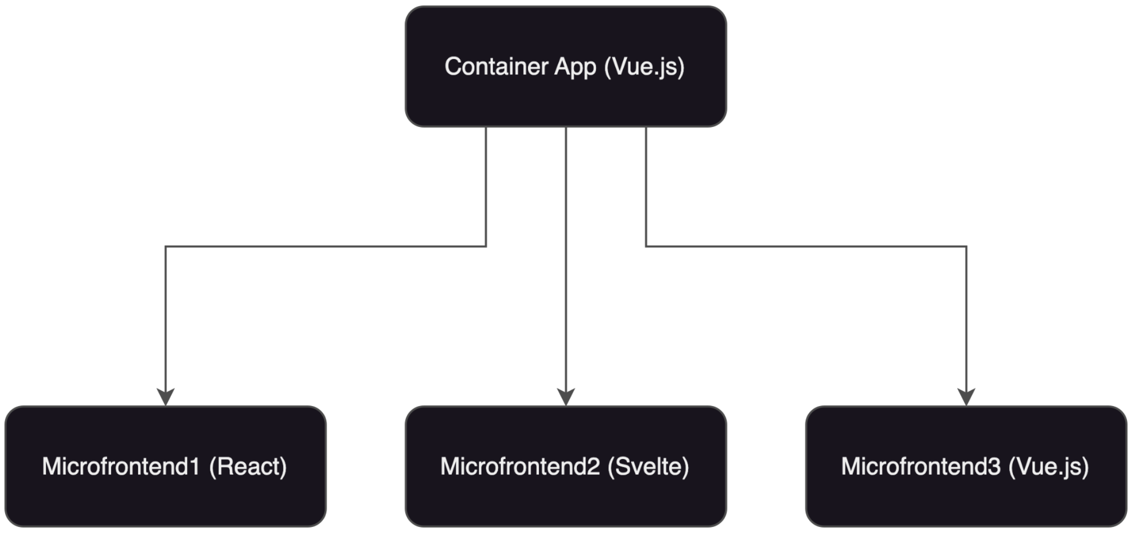
In this chapter, we will outline the architectural foundation of our sample project, following a mix of Custom Dynamic Loading and Service-Oriented Architecture (SOA) patterns. This approach will provide a robust framework for our custom solution, which consists of a container application designed to dynamically load and render three distinct micro-frontends. Each micro-frontend is built with a different modern JavaScript framework, showcasing the flexibility and scalability of the micro-frontend architecture.

NOTE: Later in Chapter 12 we'll outline a sample solution using import maps instead.

We will build the following apps:

- Container App (container-app)
- Microfrontend1 (built with React)
- Microfrontend2 (built with Svelte)
- Microfrontend3 (built with Vue)

And this is a very high-level diagram of the overall architecture:



The Container App (container-app)

The container app serves as the orchestrator for the entire application. It is responsible for managing the lifecycle of each micro-frontend, including their loading, rendering, and unmounting processes. This app will be developed using Vue.js, chosen for its reactivity and comprehensive tooling which facilitate easy integration of dynamic modules. The container app's primary responsibilities include:

- **Dynamic Module Loading**: It uses JavaScript's dynamic import feature to load micro-frontends on demand.
- **Routing**: Handles client-side routing, which directs users to different parts of the application without reloading the page.
- **State Management**: Manages any shared state that might be needed across different micro-frontends, such as user authentication data.

Microfrontend1 (Built with React)

Microfrontend1 is a React-based application that demonstrates how a popular library like React can be utilized within a micro-frontend architecture. This module will focus on showcasing interactive UI components that leverage React's state and lifecycle features. Key characteristics include:

- **Scoped Interaction**: It handles user interactions within its scope, maintaining a self-contained environment for its state and logic.
- **Independence**: Deployed independently, it encapsulates all necessary dependencies that don't conflict with the container app or other micro-frontends.

Microfrontend2 (Built with Svelte)

Microfrontend2 utilizes Svelte, a compiler-based framework that excels in building high-performance UI components with less boilerplate code. This module will illustrate how Svelte's innovative approach to compiling away the framework can be advantageous in a micro-frontend setup, particularly for achieving faster load times and smoother transitions. It will:

- **Compile-time Magic**: Leverage Svelte's compile-time enhancements to reduce runtime overhead.
- **Reactivity**: Implement Svelte's reactive programming model to handle state changes efficiently.

Microfrontend3 (Built with Vue)

The third micro-frontend is built using Vue.js, providing a cohesive experience by utilizing Vue's ecosystem for managing both the local component state and interactions with the global state managed by the container app. This setup will demonstrate:

- **Single File Components**: Use Vue's Single File Components for clear separation of template, script, and style.
- **Ecosystem Integration**: Integrate with Vue's ecosystem tools like Vuex for state management and Vue Router for component-level routing within the micro-frontend.

Integration Strategy

The architecture uses a combination of Webpack and Module Federation for JavaScript code, ensuring each micro-frontend is a standalone application that can be developed, tested, and deployed independently:

- **Module Federation**: Allows sharing of libraries and components across different micro-frontends without reloading them multiple times in the browser.
- **Isolation and Namespace**: Each micro-frontend will maintain its namespace, avoiding global conflicts and encouraging better maintainability.

Conclusion

The architecture described in this chapter provides a robust framework for developing a scalable and maintainable micro-frontend application. By leveraging different frameworks and technologies, we demonstrate the flexibility of the micro-frontend approach, catering to varied technical needs and preferences. This setup not only facilitates independent development and deployment but also ensures that the overall application remains cohesive and performant.

Chapter 3 - Microfrontend1 Project (React)

***IMPORTANT**: This chapter assumes that you already have installed a recent version of **Node.js** on your computer. If you do not have it yet, you can download it here: https://nodejs.org/en/download/*

Creating this application can be approached in various ways. Here, we will leverage TypeScript and need to set up a project with a build/transpile process that allows us to make changes and verify them in real time. While you could manually create this project, install all required npm packages, and create each file individually, it is much more efficient to use **vite**[1]

Directory Structure

Before we proceed creating the individual apps, let's create a root directory called `introduction-to-micro-frontends-project`. Inside this directory we will create the individual microfrontends and the container-app. In the end, your directory structure should look similar to this:

Create Project Wizard

To set up the project, within the directory `introduction-to-micro-frontends-project`, open your terminal and execute the following Node.js command:

```
npm init vite@latest
```

If you do not have already installed the package `create-vite@latest`[2], you will be prompted to install it. In this case, type **y** and then press **enter** to proceed:

[1] https://vitejs.dev
[2] https://www.npmjs.com/package/create-vite

```
Need to install the following packages:
  create-vite@latest
Ok to proceed? (y)
```

The create-vite wizard will then start and ask for the name of the project. The default is `vite-project`, so change this to **microfrontend1** and press enter:

```
? Project name: › microfrontend1
```

Next, you will be asked to select a framework. Use the keyboard arrows to scroll down the list and stop at **React**, then press enter:

```
? Select a framework: › - Use arrow-keys. Return to submit.
    Vanilla
    Vue
▯   React
    Preact
    Lit
    Svelte
    Solid
    Qwik
    Others
```

Then, you will be asked which "variant" you want to use. Scroll down to **TypeScript** and press enter:

```
? Select a variant: › - Use arrow-keys. Return to submit.
▯   TypeScript
    TypeScript + SWC
    JavaScript
    JavaScript + SWC
    Remix ↗️
```

This will create a folder named **microfrontend1**, which is also the name of our project. At the end, it should display a message similar to this:

```
Scaffolding project in /local-path-to/microfrontend1...

Done. Now run:

  cd microfrontend1
  npm install
  npm run dev
```

The first command navigates to the newly created sub-directory called **microfrontend1**, the second one installs all npm dependencies, and the third one serves the app locally. You'll see a message similar to this displayed:

```
  VITE v5.2.8  ready in 239 ms

  □  Local:   http://localhost:5173/
  □  Network: use --host to expose
  □  press h + enter to show help
```

From your web browser, navigate to http://localhost:5173[3], and you'll see the application's homepage rendered.

Stop the application by pressing **CTRL + C**.

Proceed to delete the file `public/vite.svg` as it will not be needed. Also, move the file `react.svg` from `src/assets/` to the `public/` directory.

Then open the file `index.html` and modify the icon `<link>` like this:

```
<link rel="icon" type="image/svg+xml" href="/react.svg" />
```

and the like this:

```
<title>Microfront1</title>
```

***NOTE**: This `index.html` will be served only when we run the microfrontend1 app standalone, not from within the container-app.*

Replace the `App.tsx` code with this:

[3] http://localhost:5173/

```
// file: App.tsx
import Counter from './components/Counter'

function App() {
  return (
    <>
      <Counter/>
    </>
  )
}

export default App
```

Add a new file at the path `microfrontend1/src/components/Counter.tsx` with this code:

```
// file: introduction-to-micro-frontends-project/microfrontend1/src/components/Count\
er.tsx
import { useState } from 'react'

const buttonStyle = {
  display: 'flex',
  alignItems: 'center',
  borderRadius: '8px',
  padding: '0.6em 1.2em',
  background: '#ffffff',
  color: '#000000',
  cursor: 'pointer',
}

export default function Counter() {
  const [count, setCount] = useState(0)

  const onClick = () => {
    console.log('react: counter onClick')
    setCount((count) => count + 1)
  }

  return (
    <button className="react-button" style={buttonStyle} onClick={onClick}>
      <img src="http://localhost:5001/react.svg" style={{width: '1.5rem'}}/>
      <span style={{marginLeft: '0.25rem'}}>Microfrontend1: Count {count}</span>
    </button>
  )
}
```

Replace the code inside src/style.css with this:

```css
:root {
  font-family: Inter, system-ui, Avenir, Helvetica, Arial, sans-serif;
  line-height: 1.5;
  font-weight: 400;

  color-scheme: light dark;
  color: rgba(255, 255, 255, 0.87);
  background-color: #242424;

  font-synthesis: none;
  text-rendering: optimizeLegibility;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
}
```

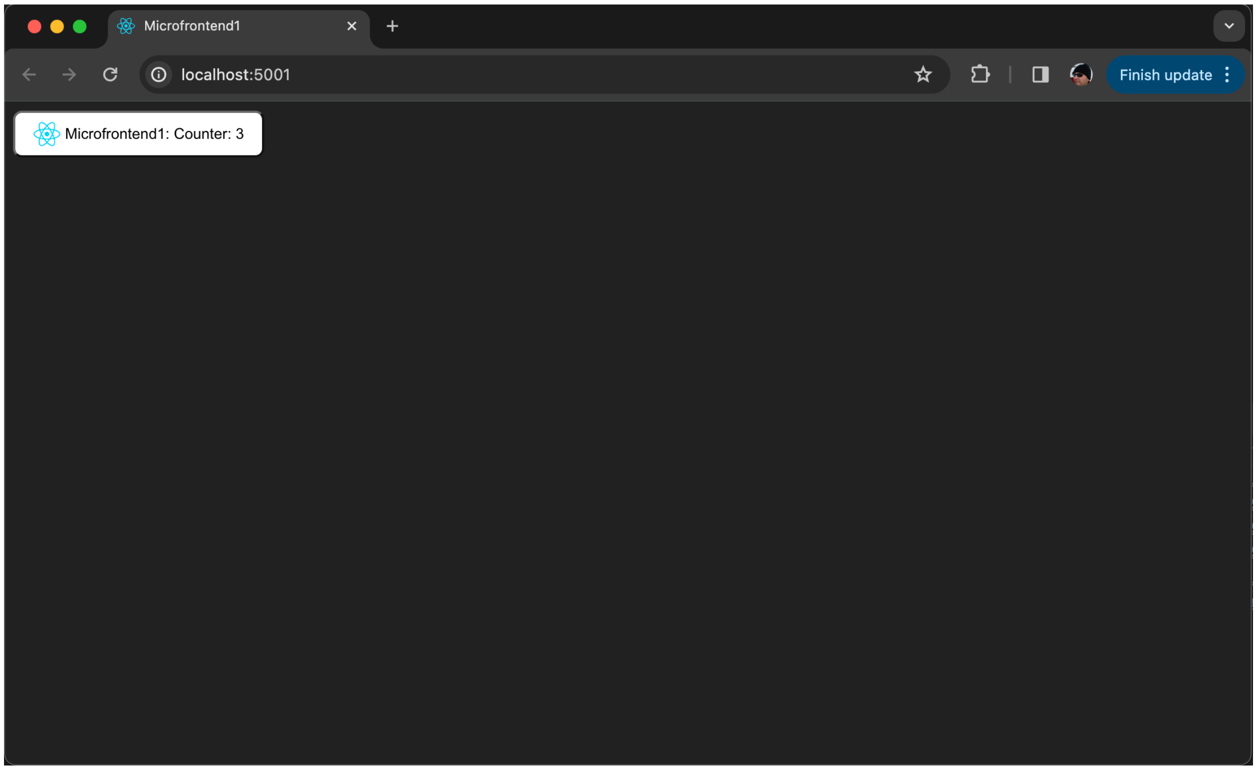
Finally, adjust the `vite.config.ts` file to customize the output directories and ensure the application runs on port 5001:

```ts
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

// This is a module app (to be consumed by the host app)
// This microfrontend uses the React framework

const port = 5001

// https://vitejs.dev/config/
export default defineConfig({
  server: {
    port: port
  },
  plugins: [react()],
  build: {
    outDir: './microfrontend1',
    cssCodeSplit: false,
    sourcemap: false,
    minify: false,
    rollupOptions: {
      output: {
        entryFileNames: `assets/[name].js`,
        chunkFileNames: `assets/[name].js`,
        assetFileNames: `assets/[name].[ext]`
      }
    }
  }
})
```

Run the app again using `npm run dev`, and it will now be served on port 5001:

```
VITE v5.2.8  ready in 117 ms

  ➜  Local:   http://localhost:5001/
  ➜  Network: use --host to expose
  ➜  press h + enter to show help
```

From your web browser, navigate to http://localhost:5001[4], and the updated interface should be displayed as shown in the screenshot.



Microfrontend1: Counter: 3

[4]http://localhost:5001/

Chapter 3 Recap

In Chapter 3, we focused on setting up and configuring Microfrontend1 using React. This chapter was pivotal in demonstrating how React can be effectively utilized in a microfrontend architecture, ensuring that it integrates smoothly with the broader application ecosystem.

Key Developments

1. **Project Setup**: We initiated Microfrontend1 using Vite, selecting React as the framework due to its widespread popularity and robust ecosystem. The setup process involved:

- Utilizing `npm init vite@latest` to scaffold a new React project.
- Choosing TypeScript during the Vite setup to enhance the project with strong typing and modern JavaScript features.

2. **Configuring the Development Environment**: Adjustments to the project's configuration were made to optimize the development and build processes specifically for a React microfrontend:

- Modifying the `vite.config.ts` to handle React-specific settings, including JSX transformation and fast refresh capabilities.
- Structuring the project directory to clearly separate concerns, such as placing components, assets, and styles in designated folders.

3. **Developing the React Application**: The core of the chapter was focused on building a functional React application that could stand alone for development and testing but also be integrated within the container app:

- Implementing essential React components and utilizing hooks for state management to demonstrate typical React patterns within a microfrontend.
- Showing how React's component-based architecture is ideally suited for encapsulated features within microfrontends.

Integration with the Container App

A significant part of the chapter detailed how to ensure that Microfrontend1 could be dynamically loaded and managed by the container app:

- Exploring strategies for exposing the React app's lifecycle methods to the container app, allowing for seamless integration and interaction.
- Discussing how to handle the mounting and unmounting processes within the container to prevent memory leaks and ensure clean transitions.

Testing and Validation

We implemented scripts and commands to build and serve Microfrontend1, verifying its functionality both as a standalone app and as part of the larger microfrontend system:

- Running the React app independently to ensure it functions correctly and meets the design requirements.
- Integrating the built React app within the container app and testing the dynamic loading process, validating the entire flow from loading to unmounting.

Conclusion

Chapter 3 provided a comprehensive guide to developing a React-based microfrontend, highlighting React's strengths in a modular and decoupled architectural context. By the end of this chapter, Microfrontend1 was fully operational, capable of independent function as well as integration with the overarching container application, thus enhancing the modular nature of our project.

Chapter 4 - Microfrontend2 Project (Svelte)

For Microfrontend2 we will use the **svelte**[1] framework.

Create Project Wizard

To set up the project, within the directory `introduction-to-micro-frontends-project`, open your terminal and execute the following Node.js command:

```
npm init vite@latest
```

The create-vite wizard will begin and prompt you for the project name. The default is `vite-project`, so change this to **microfrontend2** and press enter:

```
? Project name: › microfrontend2
```

Next, you will be asked to select a framework. Use the keyboard arrows to navigate to **Svelte** and press enter:

```
? Select a framework: › - Use arrow-keys. Return to submit.
    Vanilla
    Vue
    React
    Preact
    Lit
▯   Svelte
    Solid
    Qwik
    Others
```

The wizard will then ask which variant you want to use. Scroll down to **TypeScript** and press enter:

[1] https://svelte.dev/

```
? Select a variant: › - Use arrow-keys. Return to submit.
▯   TypeScript
    JavaScript
    SvelteKit ↗️
```

This setup creates a folder named **microfrontend2**, which corresponds to the name of our project. Upon completion, you should see a message like this:

```
Scaffolding project in /local-path-to/microfrontend2...

Done. Now run:

  cd microfrontend2
  npm install
  npm run dev
```

The first command navigates to the sub-directory called **microfrontend2**, the second installs all npm dependencies, and the third serves the app locally. You should see a message displayed like this:

```
  VITE v5.2.8  ready in 239 ms

  ▯  Local:   http://localhost:5173/
  ▯  Network: use --host to expose
  ▯  press h + enter to show help
```

From your web browser, navigate to http://localhost:5173[2] to see the application's homepage rendered.

Stop the application by pressing **CTRL + C**.

Proceed to delete the file `public/vite.svg` as it will not be needed. Also, move the `svelte.svg` file from `src/assets/` to the `public/` directory.

Then, open the file `index.html` and modify the icon `<link>` like this:

```
<link rel="icon" type="image/svg+xml" href="/svelte.svg" />
```

and the like this:

[2] http://localhost:5173/

```
<title>Microfront2</title>
```

NOTE: *This `index.html` will be served only when running the Microfrontend2 app in standalone mode, not from within the container-app.*

Replace the `App.svelte` code with this:

```
<script lang="ts">
  import Counter from './lib/Counter.svelte'
</script>

<main>
  <Counter />
</main>
```

Replace the `lib/Counter.svelte` code with this:

```
<script lang="ts">
  let count: number = 0

  const onClick = () => {
    //console.log('svelte: counter onClick')
    count += 1
  }
</script>

<button class="svelte-button" on:click={onClick}>
  <img src="http://localhost:5002/svelte.svg" alt="Svelte Logo" style="width:1rem" />
  <span>Microfrontend2: Count {count}</span>
</button>

<style>
.svelte-button {
  display: flex;
  align-items: center;
  border-radius: 8px;
  padding: 0.6em 1.2em;
  background: #ffffff;
  color: #000000;
  cursor: pointer;
}
.svelte-button *:not(:first-child) {
  margin-left: 0.25rem;
}
</style>
```

Replace the `src/app.css` with this:

```
:root {
  font-family: Inter, system-ui, Avenir, Helvetica, Arial, sans-serif;
  line-height: 1.5;
  font-weight: 400;

  color-scheme: light dark;
  color: rgba(255, 255, 255, 0.87);
  background-color: #242424;

  font-synthesis: none;
  text-rendering: optimizeLegibility;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
}
```

Finally, adjust the `vite.config.ts` file to customize the output directories and ensure the application runs on port 5002:

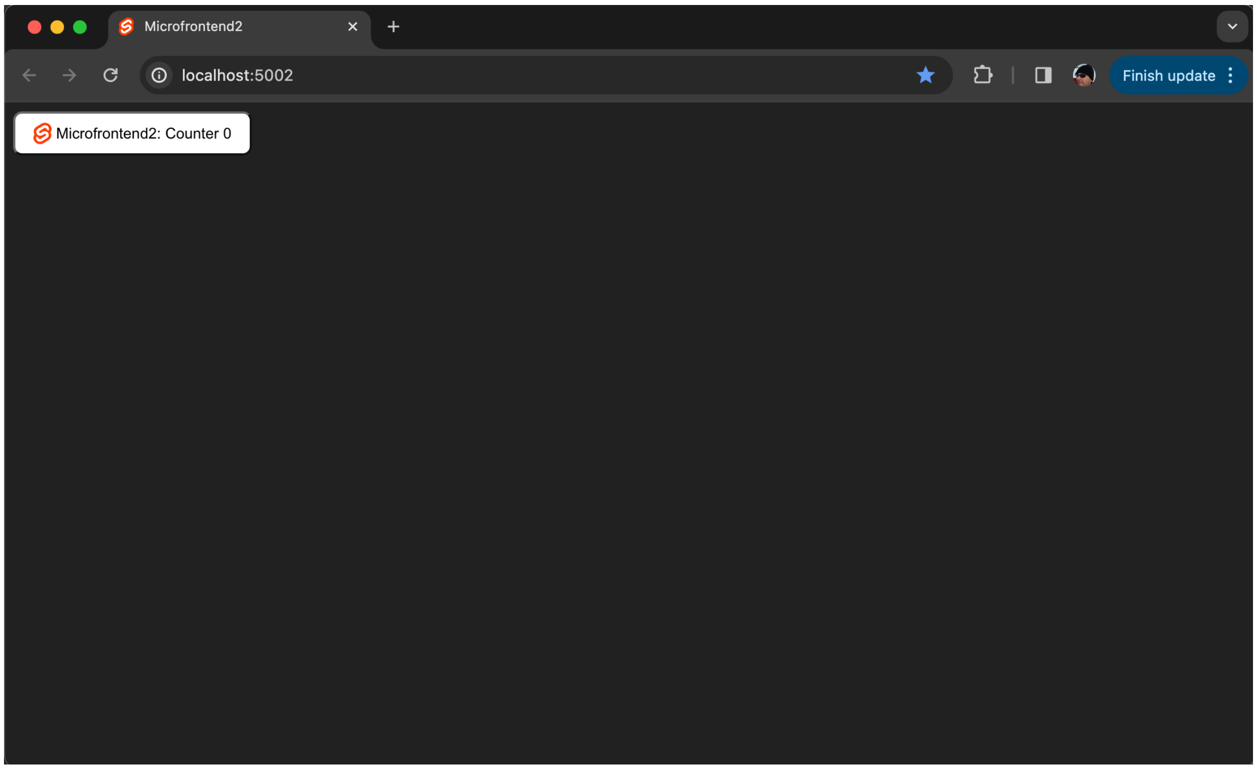
```
import { defineConfig } from 'vite'
import { svelte } from '@sveltejs/vite-plugin-svelte'

// This is a module app (to be consumed by the host app)
// This microfrontend uses the Svelte framework

const port = 5002

// https://vitejs.dev/config/
export default defineConfig({
  server: {
    port: port
  },
  plugins: [svelte()],
  build: {
    outDir: './microfrontend2',
    cssCodeSplit: false,
    sourcemap: false,
    minify: false,
    rollupOptions: {
      output: {
        entryFileNames: `assets/[name].js`,
        chunkFileNames: `assets/[name].js`,
        assetFileNames: `assets/[name].[ext]`
      }
    }
  }
})
```

Run the app again using `npm run dev`, and it will now be served on port 5002:

```
VITE v5.2.8  ready in 117 ms

  ▯  Local:   http://localhost:5002/
  ▯  Network: use --host to expose
  ▯  press h + enter to show help
```

From your web browser, navigate to http://localhost:5002[3], and the updated interface should be displayed as shown in the screenshot.

[3] http://localhost:5002/


Microfrontend2
localhost:5002
Finish update
Microfrontend2: Counter 0

Chapter 4 Recap

In Chapter 4, we embarked on creating and configuring Microfrontend2, utilizing the Svelte framework. This chapter was instrumental in demonstrating Svelte's unique capabilities and integrating them into our microfrontend architecture.

Key Developments

1. **Project Setup**: We initiated Microfrontend2 using Vite, selecting Svelte as the framework due to its innovative approach to building reactive user interfaces with a compile-time framework architecture. The setup process included:

- Utilizing `npm init vite@latest` to scaffold a new Svelte project.
- Choosing TypeScript during the Vite setup to enhance the project with strong typing alongside Svelte's reactivity model.

2. **Configuring the Development Environment**: Adjustments to the project's build and development settings were made to optimize the Svelte microfrontend:

- Customizing `vite.config.ts` to handle Svelte's specific compilation needs and to ensure efficient bundling and reloading.
- Organizing the project directory to clearly separate concerns, such as by keeping scripts, styles, and Svelte components in dedicated folders.

3. **Developing the Svelte Application**: We focused on implementing a functional and responsive Svelte application that could operate independently or as part of the container app:

- Building key features using Svelte's reactive stores for state management and single-file components for the UI.
- Demonstrating Svelte's straightforward reactivity concepts, which make it ideal for dynamic content updates within microfrontends.

Integration with the Container App

A significant portion of the chapter detailed the process for preparing Microfrontend2 to be dynamically managed by the container app:

- Exposing necessary lifecycle functions, such as mount and unmount, to allow the container app to initiate or terminate the Svelte app as required.
- Discussing best practices for ensuring the Svelte app cleanly integrates with and detaches from the DOM to maintain performance and prevent resource leaks.

Testing and Validation

We set up commands to build and serve Microfrontend2, checking its functionality both as a standalone app and when integrated with the main system:

- Independently running the Svelte app to verify that it meets functional and performance benchmarks.
- Loading the built Svelte app within the container app and testing the entire lifecycle from dynamic loading to unmounting.

Conclusion

Chapter 4 effectively guided the development of a robust Svelte-based microfrontend, demonstrating how to leverage Svelte's compilation strategy and reactive updates in a microfrontend setup. By the end of the chapter, Microfrontend2 was fully prepared for seamless integration into the larger microfrontend ecosystem, ensuring that it could be dynamically loaded, displayed, and managed within the container application.

Chapter 5 - Microfrontend3 Project (Vue)

For the Microfrontend3 we will use the **vue**[1] framework.

Create Project Wizard

To set up the project, within the directory `introduction-to-micro-frontends-project`, open your terminal and execute the following Node.js command:

```
npm init vite@latest
```

The create-vite wizard will begin and prompt you for the project name. The default is `vite-project`, so change this to **microfrontend3** and press enter:

```
? Project name: › microfrontend3
```

Next, you will be asked to select a framework. Use the keyboard arrows to navigate to **Vue** and press enter:

```
? Select a framework: › - Use arrow-keys. Return to submit.
    Vanilla
▯   Vue
    React
    Preact
    Lit
    Svelte
    Solid
    Qwik
    Others
```

The wizard will then ask which variant you want to use. Scroll down to **TypeScript** and press enter:

[1] https://vuejs.org/

```
? Select a variant: › - Use arrow-keys. Return to submit.
▯   TypeScript
    JavaScript
    Customize with create-vue ↗️
    Nuxt ↗️
```

This setup creates a folder named **microfrontend3**, which corresponds to the name of our project. Upon completion, you should see a message like this:

```
Scaffolding project in /local-path-to/microfrontend3...

Done. Now run:

  cd microfrontend3
  npm install
  npm run dev
```

The first command navigates to the sub-directory called **microfrontend3**, the second installs all npm dependencies, and the third serves the app locally. You should see a message displayed like this:

```
  VITE v5.2.8  ready in 239 ms

  ▯  Local:   http://localhost:5173/
  ▯  Network: use --host to expose
  ▯  press h + enter to show help
```

From your web browser, navigate to http://localhost:5173[2] to see the application's homepage rendered.

Stop the application by pressing **CTRL + C**.

Proceed to delete the file`public/vite.svg` as it will not be needed. Also, move the `vue.svg` file from `src/assets/` to the `public/` directory.

Then, open the file `index.html` and modify the icon `<link>` like this:

```
<link rel="icon" type="image/svg+xml" href="/vue.svg" />
```

and the like this:

[2] http://localhost:5173/

```
<title>Microfrontend3</title>
```

***NOTE**: This `index.html` will be served only when running the Microfrontend2 app in standalone mode, not from within the container-app.*

Replace the `App.vue` code with this:

```
<script setup lang="ts">
import Counter from './components/Counter.vue'
</script>

<template>
  <Counter />
</template>
```

Create a new file at the path `src/components/Counter.vue` with this code:

```
<script setup lang="ts">
import { ref } from 'vue'

const count = ref(0)

const onClick = () => {
  count.value += 1
  console.log('vue: counter onClick')
}
</script>
<template>
  <button class="vue-button" @click="onClick">
    <img style="width:1rem" src="http://localhost:5003/vue.svg" class="logo vue" alt\
="Vue logo" />
    <span>microfrontend3: Count {{ count }}</span>
  </button>
</template>

<style>
.vue-button {
  display: flex;
  align-items: center;
  border-radius: 8px;
  padding: 0.6em 1.2em;
  background: #ffffff;
  color: #000000;
  cursor: pointer;
}
.vue-button *:not(:first-child) {
  margin-left: 0.25rem;
}
</style>
```

Replace the content of the file `src/style.css` with this:

```
:root {
  font-family: Inter, system-ui, Avenir, Helvetica, Arial, sans-serif;
  line-height: 1.5;
  font-weight: 400;

  color-scheme: light dark;
  color: rgba(255, 255, 255, 0.87);
  background-color: #242424;

  font-synthesis: none;
  text-rendering: optimizeLegibility;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
}
```

Finally, adjust the `vite.config.ts` file to customize the output directories and ensure the application runs on port 5003:

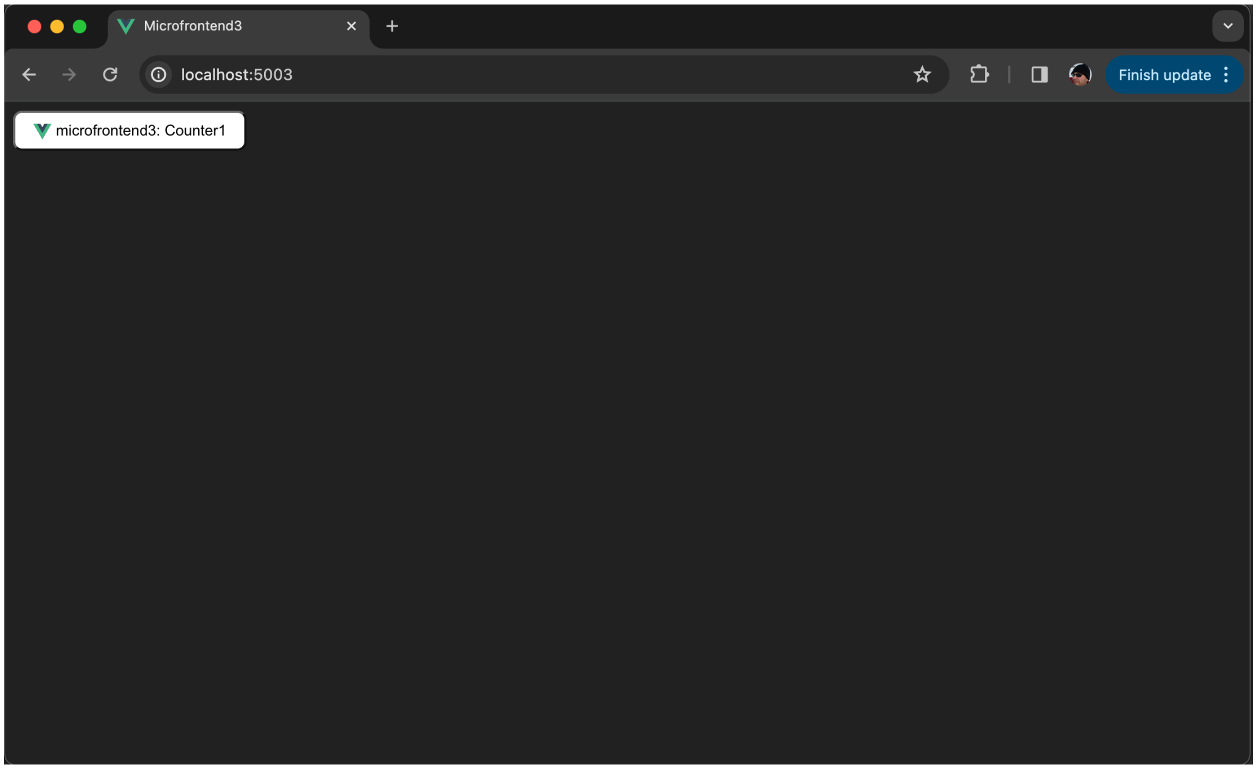
```
import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'

// This is a module app (to be consumed by the host app)
// This microfrontend uses the Vue framework

const port = 5003

// https://vitejs.dev/config/
export default defineConfig({
  server: {
    port: port
  },
  plugins: [
    vue()
  ],
  build: {
    outDir: './microfrontend3',
    cssCodeSplit: false,
    sourcemap: false,
    minify: false,
    rollupOptions: {
      output: {
        entryFileNames: `assets/[name].js`,
        chunkFileNames: `assets/[name].js`,
        assetFileNames: `assets/[name].[ext]`
      }
    }
  }
})
```

Run the app again using `npm run dev`, and it will now be served on port 5003:

```
VITE v5.2.8  ready in 117 ms

  ➜  Local:   http://localhost:5003/
  ➜  Network: use --host to expose
  ➜  press h + enter to show help
```

From your web browser, navigate to http://localhost:5003[3], and the updated interface should be displayed as shown in the screenshot.

[3]http://localhost:5003/


Microfrontend3
localhost:5003
Finish update
microfrontend3: Counter1

Chapter 5 Recap

In Chapter 5, we explored the creation and configuration of Microfrontend3, a Vue.js application. This chapter was crucial for demonstrating how Vue could be effectively used in a microfrontend architecture, complementing the other microfrontends built with React and Svelte.

Key Developments

1. **Project Setup**: We initiated the project using Vite, which is particularly well-suited for Vue projects due to its fast cold starts and hot module replacement. The setup process involved:

- Utilizing `npm init vite@latest` to scaffold a new Vue project.
- Selecting Vue as the framework and TypeScript as the programming language during the Vite setup, ensuring that the project benefits from strong typing and modern JavaScript features.

2. **Configuring the Development Environment**: Adjustments were made to the project's configuration to optimize the development and build processes specifically for a Vue microfrontend:

- Tweaking the `vite.config.ts` to properly handle Vue files.
- Ensuring that the project structure is organized in a way that separates concerns and enhances maintainability, such as by placing components and assets in dedicated directories.

3. **Developing the Vue Application**: The core of the chapter was focused on building a functional Vue application that could stand alone for development and testing but also be integrated within the container app:

- Implementing a simple yet functional Vue application using single-file components (SFCs).
- Demonstrating the use of Vue's reactive system within a microfrontend context to handle state and UI rendering effectively.

Integration with the Container App

A significant part of the chapter was dedicated to ensuring that Microfrontend3 could be dynamically loaded and controlled by the container app:

- Detailing how to expose the Vue microfrontend's lifecycle hooks (like mount and unmount) to the container app, allowing for seamless integration and interaction.
- Discussing strategies for ensuring that the Vue app cleans up after itself through proper unmounting to prevent memory leaks and maintain performance.

Testing and Validation

We implemented scripts and commands to build and serve Microfrontend3, verifying its functionality both as a standalone app and as part of the larger microfrontend system:

- Running the Vue app independently to ensure it functions correctly and meets the design requirements.
- Integrating the built Vue app within the container app and testing the dynamic loading process, validating the entire flow from loading to unmounting.

Conclusion

Chapter 5 not only provided a practical guide to developing a Vue-based microfrontend but also highlighted the versatility and robustness of Vue in a microfrontend architecture. By the end of this chapter, we established a solid foundation for Microfrontend3, ensuring that it is ready for integration with the overarching container application, thus paving the way for a cohesive and scalable microfrontend ecosystem.

Chapter 6 - Setting Up The Container App

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Create Project Wizard

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Chapter 6 Recap

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Key Steps Undertaken

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Integration and Testing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Chapter 7 - Types and Utils

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

MicroFrontend Interface

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Defining the MicroFrontend Interface

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Additional Utility: loadScript

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

MicroFrontendLoader Utility

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Container-app Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

App.vue changes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Chapter 7 Recap

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Key Developments

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Integration and Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Testing and Validation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Chapter 8 - Getting the Microfrontends Ready

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Retaining Standalone Functionality

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Implementing the MicroFrontend Interface

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Modifying the Build Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Step-by-Step Modifications

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Microfrontend1 (React app)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Implementing the MicroFrontend Interface

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Modifying the Vite Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Additional Script Commands

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Microfrontend2 (Svelte app)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Implementing the MicroFrontend Interface

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Modifying the Vite Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Additional Script Commands

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Microfrontend3 (Vue app)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Implementing the MicroFrontend Interface

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Modifying the Vite Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Additional Script Commands

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Adding a global package.json

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Chapter 8 Recap

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Key Steps Undertaken

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Integration with the Container App

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Chapter 9 - CSS Styles

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Hybrid Approach with Scoped CSS and Shared Core Styles

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

1. Scoped CSS for Component-Specific Styles

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

2. Shared Core Styles

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

3. Integration in Build Process

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

4. Testing and Validation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Benefits of This Approach

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Root Styles Project

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Additional Script Commands

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Chapter 9 Recap

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Key Challenges and Solutions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Implementation and Testing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Chapter 10 - Structuring the Layout

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Structuring the Initial Container-App Layout

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Chapter 10 Recap

Chapter 11 - Postbox - intermodule communication

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Create Project Wizard

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Install additional npm dependencies

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Add vite.config.ts

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Update package.json commands

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

tsconfig.json file

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Remove obsolete files

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Postbox types

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Postbox Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Types Declaration file

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Build the Postbox library

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Install Postbox in container-app and microfrontends

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Update vote-env.d.ts

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Import Postbox into the container-app and each microfrontend entry files

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Test it

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Chapter 11 Recap

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Chapter 12 - Sample outline using import maps (chapter content in progress)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Benefits of Import Maps

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Project Setup

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Configuring Import Maps

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Developing Micro-frontends

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/introduction-to-micro-frontends.

Integration and Communication

Deployment and Scaling