



Nikita Bogachenkov

Posted on Sep 17, 2024 • Edited on Sep 20, 2024



16

Building Micro Frontends with Vite, React, and TypeScript: A Step-by-Step Guide

#react #typescript #microfrontends #vite

In the world of modern web development, the demand for scalable and maintainable applications continues to grow. One way to meet these demands is by breaking down large applications into smaller, independent units through the use of **micro frontends**. By doing so, we can achieve greater flexibility in development and deployment. This also allows teams to work in parallel on different parts of the application, which speeds up the development process.

In this article, we'll explore how to build a micro frontend application using **Vite**, **React**, and **TypeScript**, leveraging **Module Federation**—a powerful feature that simplifies the creation and integration of micro frontends. Additionally, we'll dive into how to organize the sharing of **TypeScript types** and **application state** between different micro frontend applications.

Table of Contents

- [What are Micro Frontends and Module Federation?](#)
- [About the Application](#)
- [Types Sharing](#)
- [Remote Application 1 - UI Components](#)
- [Remote Application 2 - Artist Details](#)

- [Host Application - Artist List](#)
- [State Sharing in Micro Frontends](#)
- [Conclusion](#)

What are Micro Frontends and Module Federation?

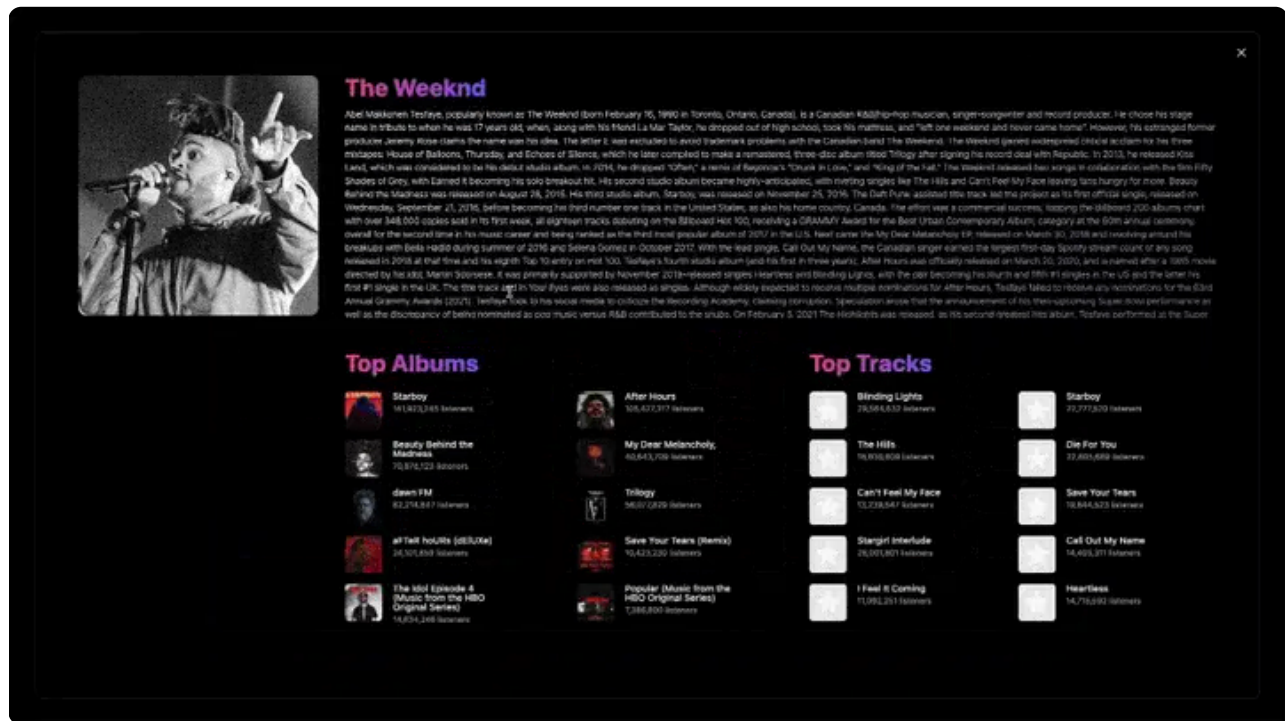
Before diving into the details, let's clarify some key terms — **Micro Frontends** and **Module Federation**.

A **Micro Frontend** is a development approach where an application is split into small, independent modules, each of which can be developed and deployed separately. This allows teams to work on different parts of the application in parallel, speeding up the development process.

In a typical micro frontend architecture, there is a **host application** that integrates and utilizes data from one or more **remote applications**. These remote applications can interact both with the host and with each other.

Module Federation is a feature of Webpack 5 that allows modules from other applications to be dynamically loaded at runtime. This simplifies the creation of micro frontends, as it enables applications to share code without having to rebuild the entire project.

About the Application



Source code

The application is a list of artists fetched from the [Last.fm](https://last.fm/) API. When an artist is selected, a window opens with detailed information about them.

The application's structure will be as follows:

- **Host application:** displays a list of artists.
- **Remote application 1:** shows artist details.
- **Remote application 2:** contains UI components.

I won't go into detail about creating React components, but I will provide links to the source code for each section.

Types Sharing

Source Code

When working with micro frontends, code is dynamically loaded, but there's no

automatic way to retrieve TypeScript types. Since we are using TypeScript in this project and various parts of the application refer to the same types, these types need to be shared between modules.

There are several approaches to achieve this, but in this article, we will focus on creating an npm package for types.

Step 1: Creating an npm Package

Let's start by creating a separate directory and installing the necessary dependencies:

```
mkdir types && cd types
yarn init -y
yarn add @types/react
```

In `package.json`, we'll provide the basic information:

```
{
  "name": "vmf-app-types",
  "version": "1.0.0",
  "description": "Type declarations for the showcase project",
  "main": "index.js",
  "license": "MIT",
  "types": "index.d.ts",
  "registry": "https://registry.npmjs.org/",
  "dependencies": {
    "@types/react": "^18.3.5"
  }
}
```

Step 2: Adding Interfaces

Next, create the `index.d.ts` file where we will define the interfaces for the artists and albums that we will fetch from the [Last.fm](https://last.fm/) API:

```

type ImageSize = "small" | "medium" | "large" | "extralarge" | "mega";

type Images = {
  size: ImageSize;
  "#text": string;
}[]

interface Artist {
  name: string;
  image: Images;
  listeners: number;
  mbid: string;
  url: string;
}

interface MusicEntity {
  url: string;
  image: Images;
  name: string;
  playcount: number;
}

```

Step 3: Modules for React Components

Since we'll be sharing React components between applications, we need to declare modules for them (otherwise, it will make your linter angry). Let's do that now to avoid revisiting this later:

```

// Last.fm interfaces
...

// ArtistDetails app
declare module "artistDetails/ArtistDetails" {
  import React from "react";

  interface ArtistDetailsProps {
    mbid: string;

```

```

    imgUrl: string | null;
  }

  const ArtistDetails: React.FC<ArtistDetailsProps>;
  export default ArtistDetails;
}

// UI app
declare module "ui/components" {
  import React from "react";

  const Wave: React.FC;

  interface ITitleProps extends React.ComponentProps<"h2"> {
    children?: React.ReactNode;
    size?: "xl" | "lg" | "base";
  }

  const Title: React.FC<ITitleProps>;

  export { Wave, Title }
}

```

After that, publish our package to npm:

```

npm login
npm publish

```

Now, we can use this package in each application.

Note #1

For each application using this package, you need to add a reference to it in the `vite-env.d.ts` file:

```

/// <reference types="mf-app-types" />

```

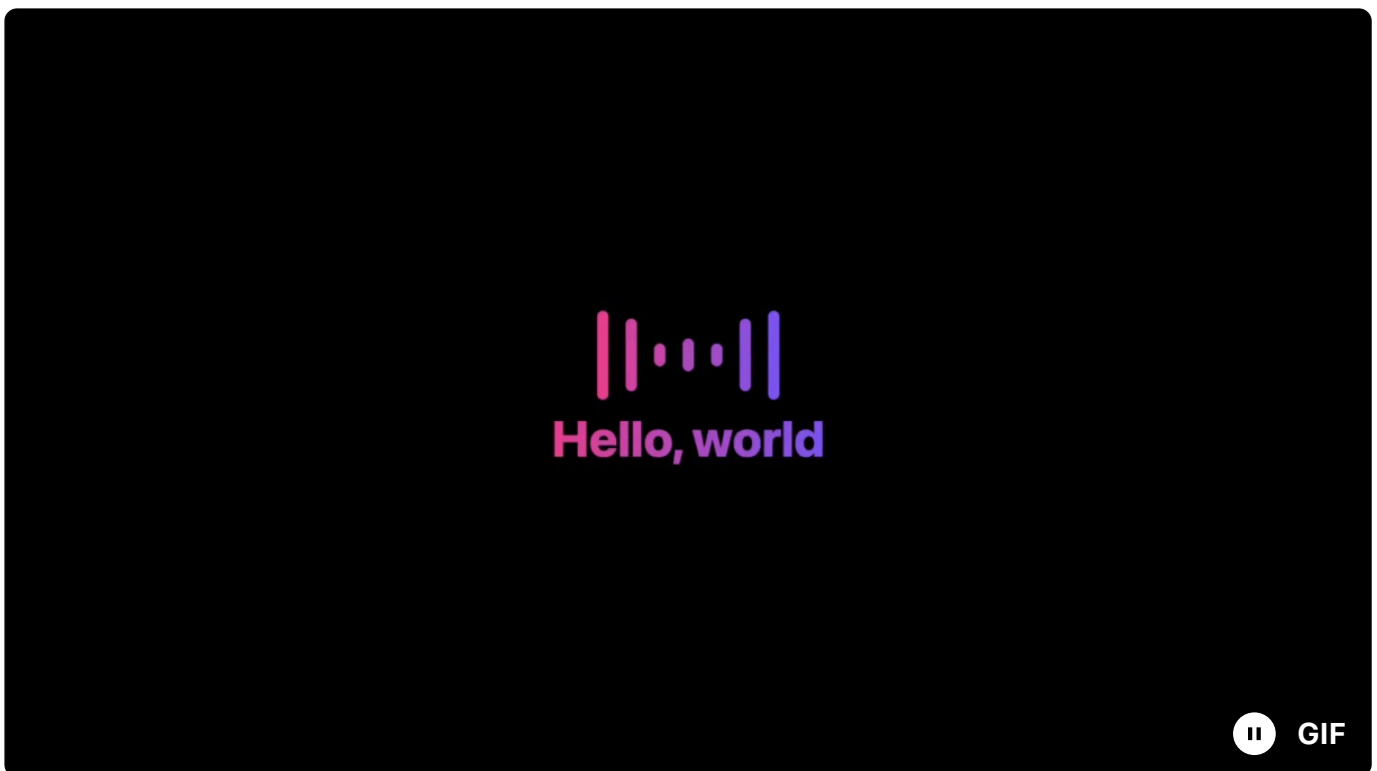
Another approach is to [publish your types under the @types organization](#) on npm.

Note #2

In this case, we've published the package as public. If you need privacy, you can create a private npm package and publish it, for example, via [GitHub Package Registry](#).

Now, we can proceed to building the application.

Remote Application #1 - UI Components



[Source Code](#)

Our UI will consist of two components: a loading indicator (Wave) and a title (Title).

First, let's create the UI application:

```
yarn create vite ui --template react-ts
cd ui
yarn
```

Then, install the necessary dependencies:

```
yarn add clsx tailwind-merge
yarn add -D tailwindcss postcss autoprefixer @originjs/vite-plugin-feder
```

The components will be located in **src/components** and exported from **index.tsx**:

```
export { default as Title } from './Title';
export { default as Wave } from './Wave';
```

Now, configure Module Federation in `vite.config.ts` using vite-plugin-federation:

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'
import ModuleFederationPlugin from "@originjs/vite-plugin-federation";

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [
    react(),
    ModuleFederationPlugin({
      name: "ui",
      filename: "remoteEntry.js",
      exposes: {
        "./components": "./src/components"
      },
      shared: ["react", "react-dom"],
    })
  ]
})
```



```
    }),  
  ],  
  build: {  
    target: "esnext",  
    minify: false,  
    cssCodeSplit: false  
  }  
})
```

Let's go over each property of the plugin:

1. `name` - The name of the remote application.
2. `filename` - The entry file for the remote application (default is `remoteEntry.js`). We will later reference this file to load the remote application.
3. `exposes` - A list of components we want to expose from this module. In this case, we're specifying `components/index.tsx`, which, in turn, exports our `Wave` and `Title` components.
4. `shared` - The dependencies and libraries we want to share between our applications. For example, if two applications use the same library, it doesn't make sense to load it multiple times in the browser. Instead, we can define these dependencies and share them across both applications.

You can also fine-tune module sharing by specifying the library version, path, and other parameters. More information can be found [here](#).

I recommend being cautious with shared libraries because it can impact tree-shaking. For instance, if you share a large icon library just to use one icon, or a UI library for one component (like a button), this can significantly increase the bundle size, potentially by several megabytes.

After configuring module federation, we can build our UI application and run it on port 3002. Note that the port must be explicitly set, as we will later reference

this port when linking to the application.


package.json:

```
"preview": "vite preview --port 3002 --strictPort"
```

```
yarn build  
yarn preview
```





















Now it's time to create the next remote application — the artist details window.

Remote Application #2 - Artist Details



The Weeknd

Abel Makkonen Tesfaye, popularly known as The Weeknd (born February 16, 1990 in Toronto, Ontario, Canada), is a Canadian R&B/hip-hop musician, singer-songwriter and record producer. He chose his stage name in tribute to when he was 17 years old, when, along with his friend LaMar Taylor, he dropped out of high school, took his mattress, and "left one weekend and never came home". However, his estranged former producer Jeremy Rose claims the name was his idea. The letter E was excluded to avoid trademark problems with the Canadian band The Weeknd. The Weeknd gained widespread critical acclaim for his three mixtapes: House of Balloons, Thursday, and Echoes of Silence, which he later compiled to make a remastered, three-disc album titled Trilogy after signing his record deal with Republic. In 2013, he released Kiss Land, which was considered to be his debut studio album. In 2014, he dropped "Often," a remix of Beyoncé's "Drunk in Love," and "King of the Fall." The Weeknd released two songs in collaboration with the film Fifty Shades of Grey, with Earned it becoming his solo breakout hit. His second studio album became highly-anticipated, with riveting singles like The Hills and Can't Feel My Face leaving fans hungry for more. Beauty Behind the Madness was released on August 28, 2015. His third studio album, Starboy, was released on November 25, 2016. The Daft Punk-assisted title track led the project as its first official single, released on Wednesday, September 21, 2016, before becoming his third number one track in the United States, as also his home country, Canada. The effort was a commercial success, topping the Billboard 200 albums chart with over 348,000 copies sold in its first week, all eighteen tracks debuting on the Billboard Hot 100, receiving a GRAMMY Award for the Best Urban Contemporary Album, category at the 60th annual ceremony, overall for the second time in his music career and being ranked as the third most popular album of 2017 in the U.S. Next came the My Dear Melancholy EP, released on March 30, 2018 and revolving around his breakups with Bella Hadid during summer of 2016 and Selena Gomez in October 2017. With the lead single, Call Out My Name, the Canadian singer earned the largest first-day Spotify stream count of any song released in 2018 at that time and his eighth Top 10 entry on Hot 100. Tesfaye's fourth studio album (and his first in three years), After Hours was officially released on March 20, 2020, and is named after a 1985 movie directed by his idol, Martin Scorsese. It

Top Albums		Top Tracks	
 Starboy 142,360,158 listeners	 After Hours 106,731,002 listeners	 Blinding Lights 29,625,935 listeners	 Starboy 22,880,211 listeners
 Beauty Behind the Madness 71,084,304 listeners	 My Dear Melancholy 40,763,968 listeners	 The Hills 16,886,969 listeners	 Die For You 22,814,673 listeners
 dawn FM 82,488,627 listeners	 Trilogy 56,222,959 listeners	 Can't Feel My Face 13,274,005 listeners	 Save Your Tears 19,897,350 listeners
 aTelt hoJiIs (dEUXe) 24,171,929 listeners	 Save Your Tears (Remix) 10,437,862 listeners	 Stargirl Interlude 26,105,213 listeners	 Call Out My Name 14,449,311 listeners
 The Idol Episode 4 (Music from the HBO Original Series) 14,635,018 listeners	 Popular (Music from the HBO Original Series) 7,433,957 listeners	 I Feel It Coming 11,118,891 listeners	 Heartless 14,751,018 listeners

[Source Code](#)

This application will expose the `ArtistDetails` component. This component accepts the artist's `id` and a link to their photo as props, loads data about the

artist from the API, and displays their information, including the top 10 albums and tracks.

Just like with the UI app, we'll start by creating a new application using Vite:

```
yarn create vite artist-details --template react-ts
```

Then, install the dependencies:

```
yarn add axios swr  
yarn add -D tailwindcss postcss autoprefixer @originjs/vite-plugin-feder
```

Let's configure module federation right away:

vite.config.ts

```
ModuleFederationPlugin({  
  name: "artistDetails",  
  filename: "remoteEntry.js",  
  exposes: {  
    "./ArtistDetails": "./src/components/ArtistDetails"  
  },  
  remotes: {  
    ui: "http://localhost:3002/assets/remoteEntry.js"  
  },  
  shared: ["react", "react-dom", "swr"],  
}),
```

You'll notice a new field here — `remotes`. In this field, we specify the link to our remote UI application (this is where port 3002 comes in handy).

Then, we can use the UI components as if they were from a normal library:

```
import { Title, Wave } from "ui/components";
```

Now, let's add the `ArtistDetails` component:

```
import React from "react";
import Info from "@components/Info";
import TopList from "@components/TopList";

interface IArtistDetailsProps {
  mbid: string;
  imgUrl: string;
}

const ArtistDetails: React.FC<IArtistDetailsProps> = ({ mbid, imgUrl }) :
  return (
    // render artist details
  );
};

export default ArtistDetails;
```

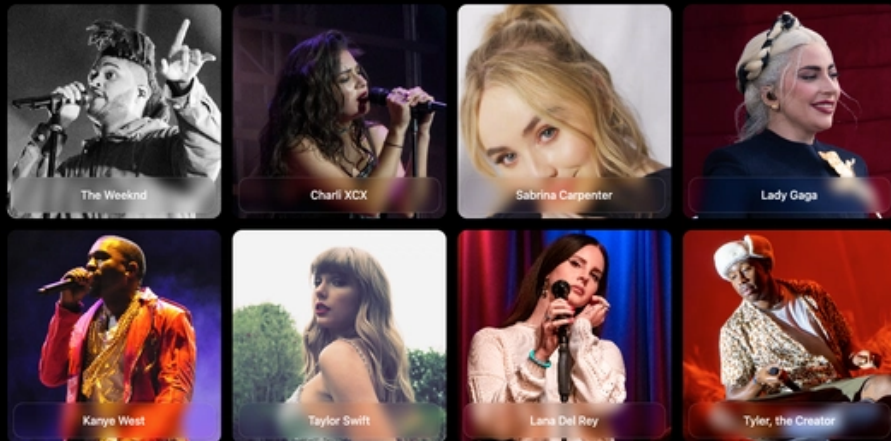
We will also build this application and run it on port 3001:

```
vite preview --port 3001 --strictPort
```

Now, it's time to bring everything together, for which we'll need the host application.

Host Application - Artist List

Last.fm Top Artists



[Source Code](#)

The creation of this application is no different from the previous ones: we still use the Vite CLI:

```
yarn create vite artist-details --template react-ts
```

Next, we install the dependencies, but this time we'll add [@nextui](#) and [framer-motion](#):

```
yarn add axios swr @nextui-org/react framer-motion  
yarn add -D tailwindcss postcss autoprefixer @originjs/vite-plugin-feder
```

Configure Module Federation:

```
ModuleFederationPlugin({  
  name: "artistList",  
  remotes: {
```

```

    artistDetails: "http://localhost:3001/assets/remoteEntry.js",
    ui: "http://localhost:3002/assets/remoteEntry.js"
  },
  shared: ["react", "react-dom", "axios", "swr"]
})

```

As you can see, the `shared` field is no longer necessary here. We simply use our remote applications.

In this application, we will load a list of top artists from [Last.fm](https://last.fm) and display it as cards. When clicking on a card, we save the selected artist's `id` and image URL:

```

const [selectedArtistMbid, setSelectedArtistMbid] = useState<string | nu
const [selectedArtistImgUrl, setSelectedArtistImgUrl] = useState<string

```

Then, we open a modal with the `ArtistDetails` component, which we previously exposed from the remote `artistDetails` application:

```

import React, { useEffect, useState } from "react";
import useSWR from "swr";
import { Spacer } from "@nextui-org/react";
import { LayoutGroup, motion } from "framer-motion";

import { fetcher } from "@utils/fetcher";
import { filterByPhoto } from "@utils/filterByPhoto";

import ArtistCard from "@components/ArtistCard";
import ArtistsLayout from "@components/ArtistsLayout";
import Modal from "@components/Modal";

import { Wave, Title } from "ui/components";
import ArtistDetails from "artistDetails/ArtistDetails";

interface IArtistListProps {
  children?: React.ReactNode;
}

```

```

}

export interface ArtistsResponse {
  artists: {
    artist: Artist[];
  };
}

const ArtistList: React.FC<IArtistListProps> = () => {
  const [selectedArtistMbid, setSelectedArtistMbid] = useState<string | null>
  (null);
  const [selectedArtistImgUrl, setSelectedArtistImgUrl] = useState<
    string | null
  >(null);

  const close = () => setSelectedArtistMbid(null);

  const { data, isLoading } = useSWR<ArtistsResponse>(
    "/?method=chart.gettopartists&format=json&limit=11",
    fetcher
  );

  useEffect(() => {
    const closeOnEscapePressed = (e: KeyboardEvent) => {
      if (e.key === "Escape") {
        close();
      }
    };
    window.addEventListener("keydown", closeOnEscapePressed);
    return () => window.removeEventListener("keydown", closeOnEscapePres
  }, []);

  const onArtistPress = (mbid: string, imgUrl: string) => {
    setSelectedArtistMbid(mbid);
    setSelectedArtistImgUrl(imgUrl);
  };

  if (isLoading) return <Wave />;

```

```

if (!data) return <Title>Sorry, no data found</Title>;

return (
  <LayoutGroup>
    <Title className="mx-auto" size="xl">
      Last.fm Top Artists
    </Title>
    <Spacer y={6} />
    <ArtistsLayout>
      {data.artists.artist.filter(filterByPhoto).map((a) => (
        <motion.div key={a.mbid} layoutId={a.mbid}>
          <ArtistCard artist={a} onPress={onArtistPress} />
        </motion.div>
      ))}
    </ArtistsLayout>
    {selectedArtistMbid && (
      <Modal layoutId={selectedArtistMbid} onClose={close}>
        <ArtistDetails
          mbid={selectedArtistMbid}
          imgUrl={selectedArtistImgUrl}
        />
      </Modal>
    )}
  </LayoutGroup>
);
};

export default ArtistList;

```

Now our artist list can not only display data but also dynamically load information about each artist from a remote application. All of this happens on the fly, without the need to refresh the entire application. Thus, we have created a fully functional application consisting of several independent micro frontends, each of which handles its own task and can be reused in other projects.

Voila! 🎉

State Sharing in Micro Frontends

Sharing state between micro frontends can be challenging, as each micro frontend operates independently. In the previous example, we used props to pass data between micro frontends. However, depending on your application architecture and requirements, other state-sharing methods might be useful. Let's explore several approaches and how they can be applied in the context of module federation and micro frontends.

1. LocalStorage

`localStorage` can be handy for sharing data between micro frontends if you need to persist state between sessions. Each micro frontend can read and write to `localStorage`, allowing them to maintain common data.

2. Custom Events

Using `CustomEvent` to share data between micro frontends allows you to dispatch events with data and subscribe to these events in other micro frontends.

Usage Example:

```
// Dispatch an event in one micro frontend
const event = new CustomEvent(
  'artistSelected',
  {
    detail: {
      mbid: '123',
      imgUrl: 'http://example.com/image.jpg'
    }
  }
);
```

```

window.dispatchEvent(event);

// Subscribe to the event in another micro frontend
window.addEventListener('artistSelected', (event: CustomEvent) => {
  const { mbid, imgUrl } = event.detail;
  // handle artist selection...
});

```

This method enables decoupled communication between micro frontends, but managing a large number of events can make the code more complex.

3. Message Bus

A Message Bus is a messaging system that allows different micro frontends to exchange data. You can use libraries like `postal.js` or `eventbus` for this purpose.

Example using `postal.js`:

```

// Publish a message in one micro frontend
postal.publish({
  channel: 'artist',
  topic: 'selected',
  data: {
    mbid: '123',
    imgUrl: 'http://example.com/image.jpg'
  }
});

// Subscribe to the message in another micro frontend
postal.subscribe({
  channel: 'artist',
  topic: 'selected',
  callback: (data) => {
    const { mbid, imgUrl } = data;
    // handle artist selection...
  }
});

```

```
});
```

The Message Bus provides flexibility in managing messages, but it may require additional setup and maintenance.

4. React Context

React Context can be used to pass data within a single application or between micro frontends if they share the same execution context. This is convenient if the micro frontends are embedded in a single application and operate within the same component tree.

Usage Example:

```
// Create a context
const ArtistContext = React.createContext(null);

// Context provider
export const ArtistProvider: React.FC = ({ children }) => {
  const [artist, setArtist] = React.useState({ mbid: '', imgUrl: '' });

  return (
    <ArtistContext.Provider value={{ artist, setArtist }}>
      {children}
    </ArtistContext.Provider>
  )
}

// Use the context in another component
import ArtistContext from "shared/ArtistContext";
const ArtistDetails: React.FC = () => {
  const { artist } = React.useContext(ArtistContext);
  // render artist details...
}
```

React Context is effective for sharing state within a single application, but for

micro frontends loaded via module federation, it may be less practical.

5. State Managers

Modern state managers like Redux, Zustand, or Recoil provide powerful state management capabilities. They allow centralized state management and can be used to share data between micro frontends.

Example using Zustand:

```
import create from 'zustand';

// Create the store
export const useArtistStore = create((set) => ({
  artist: {
    mbid: '',
    imgUrl: ''
  },
  setArtist: (artist) => set({ artist })
}));

// Set state in one micro frontend
import { useArtistStore } from "shared/stores";
const setArtist = useStore((state) => state.setArtist);
setArtist({ mbid: '123', imgUrl: 'http://example.com/image.jpg' });


// Read state in another micro frontend
import { useArtistStore } from "shared/stores";
const artist = useStore((state) => state.artist);
```


Choosing the right method depends on your application architecture and the interaction requirements between micro frontends. It's important to evaluate each method based on the specific needs of your project.

Conclusion

In conclusion, micro frontends allow teams to break down monolithic applications into smaller, manageable parts, each of which can be independently developed and deployed. By leveraging Vite's fast build tools and React's component-based architecture, along with Webpack's Module Federation, developers can seamlessly integrate and share code across different parts of their application. Whether you're sharing state via props, localStorage, or more advanced patterns like a message bus, micro frontends offer tremendous flexibility. As micro frontends continue to grow in popularity, mastering these techniques will help you build scalable, maintainable, and performant web applications.

I hope this guide has been helpful and inspiring for your own micro frontend projects.

 Coherence PROMOTED

 Coherence

Authenticated AI Chat in your app. 15 Lines of code.

```
<!-- Add this script tag to your HTML -->
<script src="https://app.withcoherence.com/sdk/coherence-sdk.js"></script>
<script>
// Initialize when page loads
window.addEventListener('DOMContentLoaded', async () => {
  await window.Coherence.init({
    oauth: {
      enabled: true,
      appId: '0b034654-121e-4d39-a096-fa20f04e33f4',
      getToken: async () => {
        const user = firebase.auth().currentUser;
        return await user?.getIdToken();
      }
    }
  });
});
</script>
<!-- Add a chat button anywhere in your app -->
<button onclick="window.Coherence.show()">🗨️ Chat with AI</button>
```

What were my 5 most interesting site visits last week?


Here are the top 5 interesting visits from last week:

Apex Marketing

- Intent signal: 🔥
- Industry: B2B Marketing Automation
- Employee count: 150-200
- Location: Austin, TX
- Visit details: June 24, 2:14pm — 8 minutes on site
- Funnel: Home → Pricing → Enterprise → Case studies

Why this is interesting:

- Spend 3+ minutes on pricing page
- Downloaded ROI calculator whitepaper

📎 Write a message... 

Authenticated AI Chat, Just 15 Lines of Code

Multi-modal streaming chat (including charts) with your existing backend data. Choose from 10+ models from all leading providers. Total control and visibility.

[Learn more](#)

Top comments (1)



Kásio Eduardo · Jul 17



nice article tnks for your time!

[Code of Conduct](#) · [Report abuse](#)



Sonar

PROMOTED



The advertisement banner features the Sonar and SonarQube logos at the top left. The main text reads "Explore the coding personalities of the leading LLMs". To the right, a blue robotic arm holds a green square with a lightbulb icon. At the bottom left, another blue robotic arm holds a pink square with a padlock icon. A blue button with the text "Read now" is positioned at the bottom right. The entire banner is framed by a light blue border.

[Explore the coding personalities of leading LLMs](#)

Sonar's new report on leading LLMs explores the critical tradeoffs between performance and security. Explore the distinct coding personalities of models like OpenAI's GPT-4o and Claude Sonnet 4 to determine the best AI strategy for your team.

[Read now](#)



Nikita Bogachenkov

Frontend developer working with React, TypeScript, and a bunch of other cool technologies. I love building sleek, user-friendly interfaces and have a passion for learning new tools

EDUCATION

Self-taught

WORK

Frontend Developer | Fullstack Developer

JOINED

Sep 13, 2024

More from [Nikita Bogachenkov](#)

Dive into Next.js Parallel & Intercepting Routing: Enhancing UX

[#nextjs](#) [#webdev](#) [#react](#) [#learning](#)

Dive into Next.js App Router: Building Dynamic, Nested, and Static Pages

#nextjs #webdev #react #learning

Dive into Next.js Server Actions: Simplifying Data Fetching & Mutations

#nextjs #react #webdev #frontend

ZeroBounce PROMOTED

...

[Improve Email Deliverability with ZeroBounce](#)

Achieve 99%+ email accuracy and transform your campaigns with ZeroBounce.

Sign Up