AUGUST 26, 2025 / #REACT

How to Build Micro Frontends in React with Vite and Module Federation

Grant Riordan

Micro Frontend Architecture has become increasingly popular in recent years, as teams look to re-use parts of their existing applications in new projects rather than rebuilding everything from scratch.

Micro frontends also allow large teams to share common components – such as headers, footers, and logins – across multiple applications, ensuring consistency and keeping their projects DRY (Don't Repeat Yourself).

In this article, you will learn:

- How to set up a project and folder structure that implements a Micro Frontend (MFE) Infrastructure
- How to use and configure the `@originjs/vite-plugin-federation` to allow Module Federation (MF) usage with Vite projects.

- How to share and consume React components between apps.
- How to run and test your app with shared components locally

Table of Contents

Pre-Requisites

- Node installed on your machine – you can get your download of Node here
- Familiarity with JS / TS and React
- Familiarity with Command Line / Terminal

What Is Module Federation?

Before we go any further, let's talk about Module Federation (MF).

Module Federation is a technique in web development that allows multiple separate builds to work together as a single application. It enables the sharing of code between different, independent applications at runtime, rather than at build time. This means a host application can dynamically load and execute code from a remote application.

At its core, Module Federation uses a **"host"** and **"remote"** architecture. A **host** application is the main app that consumes shared code / components. A **remote** application is the one that exposes code to be consumed.

The remote app specifies **which** parts of its code, or "modules," are available for others to use. The host then references these modules and loads them as needed.

Benefits of Module Federation

Independent Deployment

Module Federation lets teams build and deploy their micro-frontends separately. This eliminates the need for full application redeployments, accelerating development and reducing risk.

Efficient Code Sharing

MF provides a native way to share code and dependencies between micro-frontends. This prevents code duplication, reducing bloat and ensuring consistency.

Performance Gains

By sharing dependencies, Module Federation reduces the overall bundle size and improves initial load times, as each micro-frontend doesn't download its own copy of common libraries.

Scalability and Maintainability

MF enables a scalable architecture by breaking down large applications into smaller, manageable micro-frontends. This makes the codebase easier to maintain and allows teams to work independently.

Analogy

Imagine building an online store. Traditionally, you'd create the entire site – homepage, product pages, cart, user profile – as one big application. A small cart change would require rebuilding and redeploying everything.

With Micro Frontends and Module Federation, the shopping cart can be its own app, built and maintained by a dedicated team. The main site simply imports it, allowing the cart team to release updates independently, speeding up development and improving focus.

This also works for organisations with multiple sites that need a consistent look. Shared components like a header, footer, or product card can be reused across sites with different purposes, such as hiring vehicles or selling furniture, ensuring visual consistency while keeping functionality unique.

Project Structure

You will need to create two projects:

- **host** – this will act as your host application
- **remote** – this will expose the components you want to share

How to Create the Projects

Run the following commands in your terminal to create your root project folder and your two Vite projects:

```
# create micro-frontends directory for two vite projects, and navigate to
mkdir micro-frontends; cd micro-frontends
```

Create a Git Repository (Optional)

Using the below command you can create a Git repository for source control.

```
# initiate git repo
git init
```

```
# create host application vite app
npm create vite@latest host-app

# once submitted the command, select React and press Enter,
Select a framework:
|  ○ Vanilla
|  ○ Vue
|  ● React
|  ○ Preact
|  ○ Lit
|  ○ Svelte
|  ○ Solid
|  ○ Qwik
|  ○ Angular
|  ○ Marko
|  ○ Others

# select Typescript + SWC and again press Enter
Select a variant:
|  ○ TypeScript
|  ● TypeScript + SWC
|  ○ JavaScript
|  ○ JavaScript + SWC
|  ○ React Router v7
```

```
|  ○ React Router v7
|  ○ TanStack Router
|  ○ RedwoodSDK
|  ○ RSC
```

Once this is done, navigate back to the root project folder (`micro-frontends`):

```
# navigate back
cd ../

# create remote-app
npm create vite@latest remote-app

# follow instructions as before to select React, Typescript + SWC
```

You now have your two projects, `host-app` and `remote-app` .

How to Install Dependencies

Open the `micro-frontends` folder in your preferred IDE / Code Editor. In this tutorial I'll be using VS Code.

Tip: You can open the current folder in VS Code via your terminal using the following command `code .` if you've already added `code` to you PATH.

Once you've opened VS Code, open the terminal and run the following command:

command:

```
cd host-app && npm install -D @originjs/vite-plugin-federation
```

and then run:

```
cd ../remote-app && npm install -D @originjs/vite-plugin-federation
```

Styling

To see styling like my examples, you need to add Tailwind CSS to both your remote and host applications. You can find instructions on how to do so here

How to Configure the Remote App

In order to be able to utilise modules and components from your remote applications, you need to configure your application to expose those modules, and your host app to consume them.

Use the following configurations in your apps to allow your components to be exposed and consumed – don't worry about the components just yet, you'll create them soon.

Host App – `Vite.config.ts`

In the `host` app, open `vite.config.js` and add the following configuration:

```
//host - vote.config.js
import { defineConfig } from "vite";
import react from "@vitejs/plugin-react-swc";
import federation from "@originjs/vite-plugin-federation";


export default defineConfig({
  plugins: [
    react(),
    federation({
      name: "host_app",
      remotes: {
        remote_app: "http://localhost:5001/assets/remoteEntry.js",
      },
      shared: ["react", "react-dom"],
    }),
  ],
  build: {
    modulePreload: false,
    target: "esnext",
    minify: false,
    cssCodeSplit: false,
  },
});
```

Remote App – `Vite.config.ts`

In the `remote` app, open `vite.config.js` and add the following configuration:

```
// remote - vite.config.js
import { defineConfig } from "vite";
import react from "@vitejs/plugin-react-swc";
import federation from "@originjs/vite-plugin-federation";

export default defineConfig({
  plugins: [
    react(),
    federation({
      name: "remote_app",
      filename: "remoteEntry.js",
      exposes: {
        "./Button": "./src/components/Button",
        "./Header": "./src/components/Header",
      },
      shared: ["react", "react-dom"],
    }),
  ],
  build: {
    modulePreload: false,
    target: "esnext",
    minify: false,
    cssCodeSplit: false,
  },
  preview: {
    port: 5001,
    strictPort: true,
    cors: true,
  },
});
```

Explanation of Vite Configuration:

1. `plugins` : array where you tell Vite which **plugins** to use when it:

- Runs your dev server
- Builds your production bundle

Each plugin is basically a little (or big) piece of code that hooks into Vite's build pipeline to add extra features – for example:

- Adding React JSX/TSX support (`@vitejs/plugin-react`)
- Enabling Module Federation (`@originjs/vite-plugin-federation`)

2. `build` : Controls how Vite produces the production build. You don't need to worry too much about this for the purpose of this tutorial.

3. `preview` : Controls how the application is served / previewed:

- Useful in microfrontend setups where a fixed port and CORS enabled are needed so other apps can fetch your remote modules.
- `strictPort: true` ensures predictable networking – avoids "it works on my machine" issues with random ports.

The Vite Module Federation Plugin

You need to configure your Vite Module Federation plugin to inform it what components are to be consumed and exposed. Let's take a look at the

configured properties:

- `name`: The unique identifier for your remote application in a Module Federation setup. This is the name other applications (hosts) will use when they declare your app as a remote.
- `filename`: This is the name of the file that your host will load when it tries to retrieve your exposed modules.
- `exposes`: A mapping of public module names → local file paths. This is how you decide which parts of your code are available to be consumed remotely.

```
exposes: {
  "./Button": "./src/components/Button",
  "./Header": "./src/components/Header"
}
```

The **key** (`"./Button"`) is the **public module name** – the name that other apps (the host) will use when importing the module from your remote.

How It Works:

- **Key** (`"./Button"`) is the exposed identifier other apps can request.
- **Value** (`"./src/components/Button"`) is the actual file path inside your project.

For example, if your remote's `name` is `"remote_app"`, the host can import like this:

```
import Button from "remote_app/Button";
```

Under the hood, `"remote_app"` matches the **remote's** `name` in its `federation()` config and `Button` matches the `"./Button"` key in `exposes`.

- `shared`: dependencies that should be **shared** between the host and remote. It avoids shipping duplicate copies of large libraries (like React), ensuring the host and remote use the same instance – you can read more here about the possible configurations.
- `remotes`: A mapping of remote app names → the URL to their `remoteEntry.js` file. This tells the host where to fetch the exposed modules at runtime.

```
remotes: { remote_app: "http://localhost:5001/assets/remoteEntry.js" }
```

The key `remote_app` must match the `name` in the remote's config.

The value is the full URL to the remote's entry file (served in dev or deployed in prod). Remember we set `strictPort:true` earlier – this is why. We need

to ensure we're pointing at the correct domain & port.

Key Constraints On Exported Modules

Naming constraints and rules

The key in `exposes` (`"./Button"`):

- Must start with `./` (per Module Federation spec).
- Must be unique within the remote.
- Is case-sensitive.
- This is the **public module path** the host will request.
- Doesn't have to match the filename, but matching is a good convention for ease of reading

The file you point to (`"./src/components/Button"`):

- Can export default, named exports, or both.
- The host can import default or named exports, same as any ES module:

```
// Default export
import MyButton from 'remote_app/Button';

// Named export
import { SpecialButton } from 'remote_app/Button';
```

The import name:

- Completely up to the host developer when importing a **default** export.
- Must match exactly for **named** exports.

How to Create the Remote Components

Ok, so you've created your project structure, and you've setup your `vite.config.ts` file to allow for exposing and consuming your shared assets. Next you will create the remote components.

Button Component

Let's say you want to create a button component which will be shared across all your host applications, as you want to keep consistency. You can do this as below:

Navigate to the `remote-app` folder and create a new file called `Button.tsx` in `src/components` this will ensure it matches the configured federation plugin.

```
// remote - ./src/components/Button.tsx
import React from "react";

interface ButtonProps {
  text: string;
  onClick?: () => void;
```

```
}

const Button: React.FC<ButtonProps> = ({ text, onClick }) => {
  return (
    <button
      onClick={onClick}
      className="px-4 py-2 bg-green-500 text-white rounded hover:bg-green
    >
      {text}
    </button>
  );
};

export default Button;
```

You now have a re-usable `Button` component which has some base styling but allows for configuration of what the button does by passing in a `onClick()` argument.

Header Component

Sticking with the theme of consistency, you want to create a `<header/>` component which you can use on all your organisation's websites, ensuring a themed appearance on all applications.

Like before, create a `Header.tsx` file within `src/components/`, and paste in the following code:

```
// remote - .src/components/Header.tsx
```

```
import React from "react";

const Header: React.FC = () => {
  return (
    <header className="bg-gray-800 text-white p-4">
      <h1 className="text-2xl">Remote App Header</h1>
      <p className="text-white">Hi, Grant</p>
    </header>
  );
};

export default Header;
```

I've kept it simple, as this tutorial is for proof of concept purposes rather than aesthetic / real-world components.

How to Consume the Remote Components Within the Host

You have your remote components created, so next you need to get them into your host application and begin using them. This is quite simple now that you've already setup your `vite.config.ts`.

You **could** import the components into your `App.tsx`, but this is not best practice as it can bloat your App.tsx (entry point component). I've opted to create a `RemoteWrapperComponent` which pulls in the remote components and handles the business logic.

`RemoteComponentWrapper`:

Create a file called `RemoteComponentWrapper.tsx` in `src/components`, pasting the following code:

```
// host - ./src/components/RemoteComponentWrapper.tsx
import React, { Suspense } from "react";

const RemoteHeader = React.lazy(() => import("remote_app/Header"));
const RemoteButton = React.lazy(() => import("remote_app/Button"));

const LoadingSpinner = () => (
  <div className="flex justify-center p-4">
    <div className="animate-spin rounded-full h-8 w-8 border-b-2 border-g
  </div>
);

export const RemoteComponentWrapper = () => {
  return (
    <div className="p-4">
      <Suspense fallback={<LoadingSpinner />}>
        <RemoteHeader />
      </Suspense>

      <div className="mt-4">
        <Suspense fallback={<LoadingSpinner />}>
          <RemoteButton
            text="Remote Button"
            onClick={() =>
              alert(
                "Well done you've imported the MF remote component succes
              )
            }
          />
        </Suspense>
      </div>
    </div>
  );
```

```
};
```

This component acts as a wrapper, handling some very simple business logic such as loading your remote components, handling a loading spinner whilst waiting for the remote, and passing your `onClick` event to the remote button.

Why Use `React.lazy()`?

The import with `React.lazy` isn't required for Module Federation components – it's more of a best practice for React apps when the remote module is:

- Loaded asynchronously at runtime, which Module Federation remotes almost always are
- You want React to handle the loading state and code-splitting gracefully – shown using the `Suspense` component

`React.lazy` + `<Suspense>` gives React a built-in way to pause rendering until the component is ready. Without it, you'd need manual loading state handling.

It also keeps your components looking "normal." With React.Lazy, `<RemoteHeader/>` is just another component in your JSX.

Without it, you'd need something like:

```
const [Header, setHeader] = useState(null);

useEffect(() => {
  import("remote_app/Header").then(m => setHeader(() => m.default));
}, []);

return Header ? <Header /> : <LoadingSpinner />;
```

...which is messier and repeats for every remote component.

How to Handle TypeScript Errors

Inside your `RemoteWrapperComponent` you're going to see the following error on your `Button` and `Header` imports.

> *Cannot find module 'remote_app/Button' or its corresponding type declarations.ts (2307)*

You get this error because the remote modules are not defined with types, so both your remote and your host doesn't know what this imported component is, nor does it know its structure (a key part to TypeScript development).

To fix this you will need to provide your host app with custom types.

Add a Type Declaration File

A type declaration file (if you're unaware of them) has a `.d.ts` suffix.

Within your host app, create a file in `src/types` called `remote-app.d.ts`. Naming the file in this way lets us know the declarations within are related to the *remote-app*. This is useful especially when consuming multiple remotes.

Copy and paste the following declarations into your `remote-app.d.ts` file:

```
// host - ./src/types/remote.d.ts
declare module "remote_app/Button" {
  const Button: React.FC<{
    text: string;
    onClick?: () => void;
  }>;
  export default Button;
}

declare module "remote_app/Header" {
  const Header: React.FC;
  export default Header;
}
```

Now if you return to your `RemoteWrapperComponent` your errors should be gone. If they aren't, restart your IDE (in VS Code you can open your command palette and select `Restart Typescript Server`).

How to Add `RemoteWrapperComponent` to App.tsx

App.tsx

Import the `RemoteWrapperComponent` into App.tsx.

I've removed all the boilerplate code and replaced with some basic styling to allow us to easily see what is the host, and what are the remote components.

Copy and paste the following code into your host `App.tsx`:

```
// host - ./src/App.tsx
import viteLogo from "/vite.svg";
import "./App.css";
import "./index.css";
import { RemoteComponentWrapper } from "./components/RemoteComponentWrapp

function App() {
  return (
    <>
      <div className="px-6 border-2">
        <div className="flex justify-center items-center">
          <img src={viteLogo} alt="Example" />
        </div>
        <h1 className="text-2xl">Host Application</h1>
        <p>
          {" "}
          Welcome to the Host application, below are the components pulle
          the remote application
        </p>
        <RemoteComponentWrapper />
      </div>
    </>
  );
}

export default App;
```

How to Serve the Remote App and Run Your Host

Due to how Vite works, you need to build the application before you preview / serve the application.

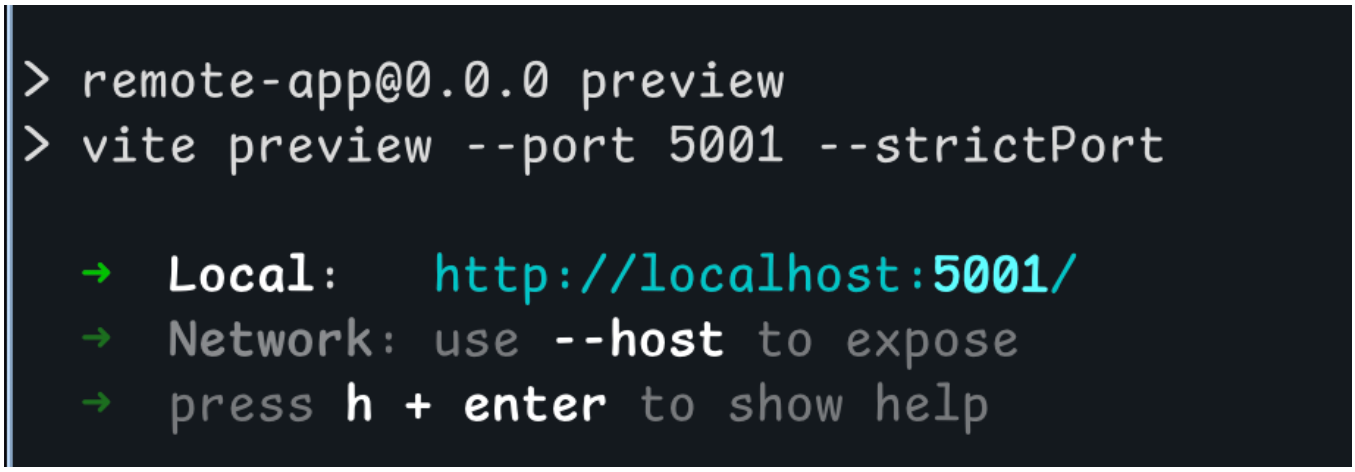
Make sure that your `package.json` file's scripts block looks like this:

```
# remote-app
"scripts": {
    "dev": "vite",
    "build": "tsc -b && vite build",
    "lint": "eslint .",
    "preview": "vite preview --port 5001 --strictPort",
    "serve": "npm run build && npm run preview"
  },

# host-app
"scripts": {
    "dev": "vite",
    "build": "tsc -b && vite build",
    "lint": "eslint .",
    "preview": "vite preview --port 5000 --strictPort",
    "serve": "npm run build && npm run preview"
  },
```

In your terminal run:

```
cd ./remote-app

# run a build, and run / load the app on port 5000
npm run serve
```

You should see something like this:



Now you can run the host application by doing the same:

```
# change to the host-app in another terminal
cd ../host-app

# build and run the application
npm run serve
```

If you open the `localhost:5000` you should now see your host application with the remote components:

Next, if you click the button, you can see that it shows the message you configured from within the `RemoteWrapperComponent`:

The True Power of Micro Frontends

The true power of micro frontends lies in their ability to update the remote components, without the need to rebuild the host. To fully demonstrate this, keep the host running, and update the `Button` component on your `remote-app`.

Let's update the remote components. Use the below code to update both the `Button` and `Header` components:

```tsx
// remote - ./src/components/Header.tsx
import React from "react";

const Header: React.FC = () => {
  return (
    <header className="bg-gray-800 text-white p-4">
      <h1 className="text-2xl">Updated Remote App Header</h1>
      <p className="text-white">Hi, Grant</p>
    </header>
  );
};

export default Header;

// remote - ./src/components/Button.tsx
import React from "react";

interface ButtonProps {
  text: string;
  onClick?: () => void;
}
```



```tsx
onClick }) => {
      className="px-4 py-2 bg-red-500 text-white rounded hover:bg-red-600
    >
      {text}
    </button>
  );
};

export default Button;
```

Once you've updated the remote components, run the following command in your `remote-app` folder:

```
npm run serve
```

Next, refresh your host app in the browser and you will see the updated app:

The update to the remote components is visible immediately, without restarting or rebuilding the host app. This highlights a key benefit of micro frontends: shared components are fetched from their own server via the `remote.js` file, enabling independent updates.

Final Thoughts

You've successfully built and deployed a micro frontend architecture – congrats! This basic implementation demonstrates the true power of Module Federation and the ability to update shared components without

needing to rebuild and redeploy the entire host application.

This independence can dramatically accelerate development cycles and empower teams to work more autonomously.

I hope you've learned something from this article, and as always for more tutorials and discussions, connect with me on twitter/x.

Grant Riordan

Read more posts.

If you read this far, thank the author to show them you care.

Say Thanks

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers. Get started

freeCodeCamp is a donor-supported tax-exempt 501(c)(3) charity organization (United States Federal Tax Identification Number: 82-0779546)

Our mission: to help people learn to code for free. We accomplish this by creating thousands of videos, articles, and interactive coding lessons - all freely

Trending Books and Handbooks

REST APIs

Clean Code

TypeScript

JavaScript

AI Chatbots

Command Line

GraphQL APIs

available to the public.

Donations to freeCodeCamp go toward our education initiatives, and help pay for servers, services, and staff.

You can make a tax-deductible donation here.

CSS Transforms

Access Control

REST API Design

PHP

Java

Linux

React

CI/CD

Docker

Golang

Python

Node.js

Todo APIs

JavaScript Classes

Front-End Libraries

Express and Node.js

Python Code Examples

Clustering in Python

Software Architecture

Programming Fundamentals

Coding Career Preparation

Full-Stack Developer Guide

Python for JavaScript Devs

Mobile App

Our Charity