

ChatGPT - React TS Micro-Frontend

show me example for todolist app witch react ts micro-frontend module federation with full example and explain

Building a **Todo List app** using **React TypeScript**, **micro-frontend architecture**, and **Webpack Module Federation** involves breaking the application into smaller independently developed, deployed, and maintained modules. Here's a full example and explanation:

Project Structure

We'll split the application into two micro-frontends:

1. **Shell (Host)**: The main container that loads and orchestrates other micro-frontends.
 2. **Todo Module (Remote)**: A standalone module handling the Todo List functionality.
-

Setup Steps

1. Create Two React Applications

Use `create-react-app` or Vite to initialize the projects, ensuring TypeScript is enabled.

```
bash

npx create-react-app shell --template typescript
npx create-react-app todo-module --template typescript
```

2. Configure Webpack and Module Federation

Install Required Packages

Install Webpack and Module Federation dependencies for both projects:

```
bash
```

```
npm install webpack webpack-cli webpack-dev-server @module-federation/webpack-plugin
-D
```

Shell (Host)

1. Modify `webpack.config.js` in the `shell` project:

javascript

```
const ModuleFederationPlugin = require("@module-federation/webpack-
plugin").ModuleFederationPlugin;
const path = require("path");

module.exports = {
  entry: "./src/index.tsx",
  mode: "development",
  devServer: {
    port: 3001,
    historyApiFallback: true,
  },
  resolve: {
    extensions: [".tsx", ".ts", ".js"],
  },
  output: {
    publicPath: "http://localhost:3001/",
  },
  module: {
    rules: [
      {
        test: /\.tsx?$/,
        loader: "ts-loader",
        exclude: /node_modules/,
      },
    ],
  },
  plugins: [
    new ModuleFederationPlugin({
      name: "shell",
      remotes: {
        todoModule: "todoModule@http://localhost:3002/remoteEntry.js",
      },
      shared: {
```

```

    react: { singleton: true, requiredVersion: "^18.0.0" },
    "react-dom": { singleton: true, requiredVersion: "^18.0.0" },
  },
}),
],
};

```

2. Import and Use Remote Component In `src/App.tsx` of the shell project:

```

tsx

import React from "react";

const TodoList = React.lazy(() => import("todoModule/TodoApp"));

const App: React.FC = () => (
  <React.Suspense fallback={<div>Loading...</div>}>
    <h1>Micro-Frontend Shell</h1>
    <TodoList />
  </React.Suspense>
);

export default App;

```

Todo Module (Remote)

1. Modify `webpack.config.js` in the `todo-module` project:

```

javascript

const ModuleFederationPlugin = require("@module-federation/webpack-
plugin").ModuleFederationPlugin;
const path = require("path");

module.exports = {
  entry: "./src/index.tsx",
  mode: "development",
  devServer: {
    port: 3002,
    historyApiFallback: true,
  },
  resolve: {

```

```

    extensions: [".tsx", ".ts", ".js"],
  },
  output: {
    publicPath: "http://localhost:3002/",
  },
  module: {
    rules: [
      {
        test: /\.tsx?$/,
        loader: "ts-loader",
        exclude: /node_modules/,
      },
    ],
  },
  plugins: [
    new ModuleFederationPlugin({
      name: "todoModule",
      filename: "remoteEntry.js",
      exposes: {
        "./TodoApp": "./src/TodoApp",
      },
      shared: {
        react: { singleton: true, requiredVersion: "^18.0.0" },
        "react-dom": { singleton: true, requiredVersion: "^18.0.0" },
      },
    }),
  ],
};

```

2. Create the Todo Component In `src/TodoApp.tsx` of the `todo-module` project:

```

tsx

import React, { useState } from "react";

const TodoApp: React.FC = () => {
  const [todos, setTodos] = useState<string[]>([]);
  const [input, setInput] = useState("");

  const addTodo = () => {
    if (input.trim()) {
      setTodos([...todos, input]);
      setInput("");
    }
  }
}

```

```

    }
  };

  return (
    <div>
      <h2>Todo List</h2>
      <input
        value={input}
        onChange={(e) => setInput(e.target.value)}
        placeholder="Enter a todo"
      />
      <button onClick={addTodo}>Add</button>
      <ul>
        {todos.map((todo, index) => (
          <li key={index}>{todo}</li>
        ))}
      </ul>
    </div>
  );
};

export default TodoApp;

```

3. **Export Component** Ensure `TodoApp` is exported properly in `src/index.ts` :

```

tsx

import("./TodoApp");

```

3. Run Applications

1. **Start** `todo-module`

```

bash

npm start

```

2. **Start** `shell`

```

bash

```

Explanation of Key Concepts

1. Module Federation:

- Allows apps to dynamically load components from other applications at runtime.
- Enables shared dependencies between micro-frontends.

2. Micro-Frontend Benefits:

- Independent deployment and scaling.
- Enables parallel development.

3. Webpack Config:

- `remotes` in the host (`shell`) specifies where to fetch remote modules.
- `exposes` in the remote (`todo-module`) specifies which components are shared.

4. React.lazy and Suspense:

- Used to dynamically load and render the remote component.

This example demonstrates a basic integration of micro-frontends with React, TypeScript, and Webpack Module Federation. You can expand this by adding state management, styling, or routing as needed.

show me example for todolist app with react ts vite micro-frontend module federation with full example and explain

To implement a **Todo List app** using **React TypeScript**, **Vite**, and **Module Federation** for a micro-frontend architecture, we'll set up the following:

1. **Shell (Host)**: The main application that integrates and renders remote modules.
 2. **Todo Module (Remote)**: A standalone module that provides the Todo List functionality.
-

Steps to Build the Application

1. Initialize Projects

Create two separate Vite projects for the **Shell** and the **Todo Module**.

```
bash

# Create Shell (Host)
npm create vite@latest shell --template react-ts
cd shell
npm install

# Create Todo Module (Remote)
npm create vite@latest todo-module --template react-ts
cd todo-module
npm install
```

2. Install Required Plugins

Add the **Module Federation Plugin** to both projects:

```
bash

npm install @originjs/vite-plugin-federation --save-dev
```

3. Configure Vite for Module Federation

Shell (Host) Configuration

1. Update `vite.config.ts`:

```
typescript

import { defineConfig } from "vite";
import react from "@vitejs/plugin-react";
```

```
import federation from "@originjs/vite-plugin-federation";

export default defineConfig({
  plugins: [
    react(),
    federation({
      remotes: {
        todoModule: "http://localhost:3001/assets/remoteEntry.js",
      },
      shared: ["react", "react-dom"],
    }),
  ],
  build: {
    target: "esnext",
  },
  server: {
    port: 3000,
  },
});
```

2. Load the remote component in `src/App.tsx` :

```
tsx

import React from "react";

const TodoApp = React.lazy(() => import("todoModule/TodoApp"));

const App: React.FC = () => {
  return (
    <React.Suspense fallback={<div>Loading...</div>}>
      <h1>Shell (Host Application)</h1>
      <TodoApp />
    </React.Suspense>
  );
};

export default App;
```


Todo Module (Remote) Configuration

1. Update vite.config.ts :

typescript

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

export default defineConfig({
  plugins: [
    react(),
    federation({
      name: "todoModule",
      filename: "remoteEntry.js",
      exposes: {
        "./TodoApp": "./src/ToDoApp.tsx",
      },
      shared: ["react", "react-dom"],
    }),
  ],
  build: {
    target: "esnext",
    minify: false,
    cssCodeSplit: false,
  },
  server: {
    port: 3001,
  },
});
```

2. Create a Todo List Component in src/ToDoApp.tsx :

tsx

```
import React, { useState } from "react";

const TodoApp: React.FC = () => {
  const [todos, setTodos] = useState<string[]>([]);
  const [input, setInput] = useState("");

  const addTodo = () => {
```

```

    if (input.trim()) {
      setTodos([...todos, input]);
      setInput("");
    }
  };

  return (
    <div>
      <h2>Todo List (Remote Module)</h2>
      <input
        value={input}
        onChange={(e) => setInput(e.target.value)}
        placeholder="Enter a todo"
      />
      <button onClick={addTodo}>Add</button>
      <ul>
        {todos.map((todo, index) => (
          <li key={index}>{todo}</li>
        ))}
      </ul>
    </div>
  );
};

export default TodoApp;

```

3. Export `TodoApp` in `src/main.tsx`:

typescript

```
import("./TodoApp");
```

4. Run the Applications

1. Start the **Todo Module** (Remote):

bash

```
npm run dev
```

It will be available at `http://localhost:3001`.

2. Start the **Shell (Host)**:

```
bash
```

```
npm run dev
```

It will be available at `http://localhost:3000`.

Explanation

1. Vite and Module Federation:

- Vite provides fast build times, and the `@originjs/vite-plugin-federation` plugin enables dynamic loading of remote modules.
- The `remotes` configuration in the **Shell** specifies where to load the remote app.
- The `exposes` configuration in the **Todo Module** specifies which components are shared.

2. Lazy Loading with `React.lazy`:

- Dynamically imports the remote component, improving initial load times.

3. Micro-Frontend Benefits:

- Independent teams can develop, test, and deploy their micro-frontends.
 - Modules can be updated or scaled independently.
-

Full Example

Now, your **Shell** integrates the **Todo Module** at runtime. When you open

`http://localhost:3000`, the **Shell** app loads the Todo List component from

`http://localhost:3001`.

You can expand this architecture by adding more micro-frontends or implementing shared state management (e.g., Zustand, Redux).

show me this example with integrate other module like counter app

To enhance the **Todo List app** with another **Counter module**, we will create a second micro-frontend for the counter functionality and integrate it into the **Shell**. Here's how the setup will look:

Final Structure

- **Shell (Host):** Integrates the **Todo Module** and **Counter Module**.
 - **Todo Module (Remote):** Provides the Todo List functionality.
 - **Counter Module (Remote):** Provides the Counter functionality.
-

Steps to Add the Counter Module

1. Create the Counter Module

1. Create a new Vite project for the **Counter Module**:

```
bash

npx create-vite@latest counter-module --template react-ts
cd counter-module
npm install
```

2. Install the **Module Federation Plugin**:

```
bash

npm install @originjs/vite-plugin-federation --save-dev
```

3. Configure **Counter Module** in `vite.config.ts`:

```
typescript
```

```

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

export default defineConfig({
  plugins: [
    react(),
    federation({
      name: "counterModule",
      filename: "remoteEntry.js",
      exposes: {
        "./CounterApp": "./src/CounterApp.tsx",
      },
      shared: ["react", "react-dom"],
    }),
  ],
  build: {
    target: "esnext",
    minify: false,
    cssCodeSplit: false,
  },
  server: {
    port: 3002,
  },
});

```

4. Create a Counter Component in `src/CounterApp.tsx` :

tsx

```

import React, { useState } from "react";

const CounterApp: React.FC = () => {
  const [count, setCount] = useState(0);

  return (
    <div>
      <h2>Counter Module</h2>
      <p>Current Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <button onClick={() => setCount(count - 1)}>Decrement</button>
    </div>
  );
}

```

```
);  
};  
  
export default CounterApp;
```

5. Export the **CounterApp** in `src/main.tsx`:

```
typescript  
  
import("./CounterApp");
```

6. Start the **Counter Module**:

```
bash  
  
npm run dev
```

It will be available at `http://localhost:3002`.

2. Update Shell to Integrate Both Modules

1. Update `vite.config.ts` in the **Shell** project to include both remotes:

```
typescript  
  
import { defineConfig } from "vite";  
import react from "@vitejs/plugin-react";  
import federation from "@originjs/vite-plugin-federation";  
  
export default defineConfig({  
  plugins: [  
    react(),  
    federation({  
      remotes: {  
        todoModule: "http://localhost:3001/assets/remoteEntry.js",  
        counterModule: "http://localhost:3002/assets/remoteEntry.js",  
      },  
      shared: ["react", "react-dom"],  
    }),  
  ],  
  build: {
```

```
    target: "esnext",
  },
  server: {
    port: 3000,
  },
});
```

2. Update `src/App.tsx` in the **Shell** to render both the **Todo Module** and the **Counter Module**:

```
tsx

import React from "react";

const TodoApp = React.lazy(() => import("todoModule/ToDoApp"));
const CounterApp = React.lazy(() => import("counterModule/CounterApp"));

const App: React.FC = () => {
  return (
    <div>
      <h1>Shell Application</h1>
      <React.Suspense fallback=<div>Loading Todo Module...</div>>
        <TodoApp />
      </React.Suspense>
      <React.Suspense fallback=<div>Loading Counter Module...</div>>
        <CounterApp />
      </React.Suspense>
    </div>
  );
};

export default App;
```

3. Start the **Shell**:

```
bash

npm run dev
```

It will be available at `http://localhost:3000`.

Final Architecture

1. Shell (Host):

- Loads and renders `TodoApp` and `CounterApp` dynamically.
- Acts as the central orchestrator.

2. Todo Module:

- Available at `http://localhost:3001`.
- Provides the `TodoApp` component.

3. Counter Module:

- Available at `http://localhost:3002`.
 - Provides the `CounterApp` component.
-

How It Works

- The **Shell** dynamically imports and renders components from both remotes using **React.lazy** and **Webpack Module Federation**.
 - Micro-frontends are decoupled, independently deployed, and loaded at runtime.
 - Shared dependencies (`react` and `react-dom`) ensure no duplication.
-

Run and Test

1. Start the **Todo Module** (`npm run dev` in the `todo-module` directory).
 2. Start the **Counter Module** (`npm run dev` in the `counter-module` directory).
 3. Start the **Shell** (`npm run dev` in the `shell` directory).
 4. Open `http://localhost:3000` to see the **Shell** integrating both modules.
-

Demo Output

You should see:

1. **Todo List:** A fully functional Todo List from the Todo Module.
2. **Counter:** A working Counter App from the Counter Module.

Each module works independently, but they are seamlessly integrated into the **Shell**.

what is mean Export the CounterApp in src/main.tsx:

When we say "**Export the CounterApp in src/main.tsx**", it means making the CounterApp component available for other applications (like the Shell) to import dynamically via **Module Federation**.

In **Module Federation**, a remote application must explicitly "expose" the components or modules it wants to share. The `exposes` configuration in the `vite.config.ts` specifies which files or components are made available, and those files must export something.

How It Works

1. In the **Counter Module**, we configure `vite.config.ts` to expose `./src/CounterApp.tsx`:

```
typescript

federation({
  exposes: {
    './CounterApp': './src/CounterApp.tsx',
  },
});
```

This tells Module Federation that the CounterApp component, located at `src/CounterApp.tsx`, is available for other applications to use.

2. The `src/main.tsx` file in a typical React application is where the app's root component is rendered, but for the **Counter Module**, it needs to bootstrap Module Federation.

What to Do in `src/main.tsx`

The `src/main.tsx` must initialize the app properly, ensuring that Module Federation works correctly. Specifically, if we use the `./CounterApp` file, we just need to **import** it to register the module federation runtime.

Here's an example of `src/main.tsx` for the **Counter Module**:

typescript

```
// Import the component that is being exposed  
import "./CounterApp";
```

Why This Is Needed

- **Module Federation Registration:** The `vite-plugin-federation` plugin dynamically registers any file imported in the entry point (`src/main.tsx`). By importing `CounterApp`, we make it available to the Module Federation runtime.
- **Tree Shaking:** Without explicitly importing the exposed file, modern bundlers like Vite might exclude it during the build process.

Key Points

- The exposed file (`CounterApp.tsx`) must be explicitly imported or referenced in the app's entry point (`src/main.tsx`).
- If `src/main.tsx` already contains rendering logic (e.g., rendering a `root`), this additional import does not interfere with it.

For instance:

typescript

```
import React from "react";  
import ReactDOM from "react-dom/client";  
import App from "./App";
```

```
// Import the exposed CounterApp for Module Federation  
import "./CounterApp";  
  
ReactDOM.createRoot(document.getElementById("root")).render(  
  <React.StrictMode>  
    <App />  
  </React.StrictMode>  
);
```

This ensures the `CounterApp` is properly registered as a remote module for other apps (like the Shell) to use.