**Production setup with Nginx, Docker**

how to adapt your two-microfrontend setup to a production-ready environment, bundling apps with Webpack for production and serving them via Nginx in Docker containers.

**Production Setup Overview**

* Remote app and Host app build production bundles (minified, hashed).
* Docker containers run Nginx servers to serve static files.
* Host app loads remote's federated modules via URLs served by Nginx.

**1. Remote App Production Setup**

**Package.json**

{

  "name": "remote-app",

  "version": "1.0.0",

  "scripts": {

    "build": "webpack --mode production"

  },

  "dependencies": {

    "react": "^18.3.1",

    "react-dom": "^18.3.1"

  },

  "devDependencies": {

    "@babel/core": "^7.21.0",

    "@babel/preset-react": "^7.21.0",

    "babel-loader": "^9.1.2",

    "webpack": "^5.88.0",

    "webpack-cli": "^5.1.2"

  }

}

**webpack.config.js (production mode)**

const path = require('path');

const { ModuleFederationPlugin } = require('webpack').container;

module.exports = {

  entry: './src/index',

  mode: 'production',

  output: {

    path: path.resolve(\_\_dirname, 'dist'),

    publicPath: 'http://localhost:3001/remote/',  // <- Add here

    filename: '[name].[contenthash].js',

    clean: true,

  },

  module: {

    rules: [{ test: /\.jsx?$/, loader: 'babel-loader', exclude: /node\_modules/ }],

  },

  plugins: [

    new ModuleFederationPlugin({

      name: 'remoteApp',

      filename: 'remoteEntry.js', // fixed name for predictable host URL

      exposes: {

        './Widget': './src/Widget',

      },

      shared: {

        react: { singleton: true, requiredVersion: '^18.3.1' },

        'react-dom': { singleton: true, requiredVersion: '^18.3.1' },

      },

    }),

  ],

  resolve: { extensions: ['.js', '.jsx'] },

};

Notes

**Set the correct publicPath on remote app bundles**

In your remote app's webpack.config.js, you correctly set:

output: {

publicPath: '/remote/',

...

}

But for **production and Docker**, you must set the full URL including hostname and port to let the host app load remote chunks correctly.

Change publicPath to:

  output: {

    path: path.resolve(\_\_dirname, 'dist'),

    publicPath: 'http://localhost:3001/remote/',  // <- Add here

    filename: '[name].[contenthash].js',

    clean: true,

  }

**Why?**

* This tells Webpack and Module Federation exactly where all remote chunks are hosted and loaded from.
* Now, when your host app loads remote chunks, URLs will look like http://localhost:3001/remote/433.35930b529ec04b8b1103.js — which is served correctly.

**nginx.conf**

server {

  listen 80;

  server\_name localhost;

  # This location serves files from /usr/share/nginx/html/remote/

  location ^~ /remote/ {

    alias /usr/share/nginx/html/remote/;

    # try\_files tries the file or returns 404

    try\_files $uri $uri/ =404;

  }

  # Caching headers for static resources

  location ~\* \.(js|css|png|jpg|jpeg|gif|ico|svg)$ {

    expires 30d;

    add\_header Cache-Control "public, no-transform";

  }

}

This nginx configuration defines a server block that listens on port 80 and serves requests for the hostname "localhost".​

**Server Block**

* The server { ... } section declares a virtual server configuration for nginx.​
* The listen 80; directive instructs nginx to accept HTTP requests on port 80, the default web port.​
* The server\_name localhost; directive matches requests sent to the "localhost" hostname, ensuring this block only handles those requests.​

**Location Block for /remote/**

* The location ^~ /remote/ { ... } block handles all requests whose URI path begins with /remote/.​
* The ^~ operator gives this block higher priority, matching any path starting with /remote/ before regular expression matches.​
* The alias /usr/share/nginx/html/remote/; directive maps requests to files under that directory. For example, a request for /remote/app.js will serve the file /usr/share/nginx/html/remote/app.js from the file system.​
* The try\_files $uri $uri/ =404; instructs nginx to look for the requested file, then directory, and if neither exists, return a 404 error.​
  + $uri is the requested path.
  + $uri/ checks for a directory if the file isn't found.
  + =404 means that if neither exists, nginx returns a 404 HTTP error.​

**Caching Headers for Static Assets**

* The location ~\* \.(js|css|png|jpg|jpeg|gif|ico|svg)$ { ... } block matches requests for static resource file extensions (JS, CSS, images, etc.) using a case-insensitive regular expression​.
* expires 30d; instructs browsers to consider these resources fresh for 30 days, encouraging caching and reducing repeated requests.​
* add\_header Cache-Control "public, no-transform"; sets caching policy so these files can be cached by public caches and are not transformed during transit.​

**Summary of Behavior**

* Requests to resources under /remote/ are mapped to the specified directory, with missing files resulting in 404 errors.​
* Static resources (JS, CSS, images) receive caching headers for performance optimization.​
* The server only responds to requests for "localhost" on port 80, supporting basic local web application setups.​

This configuration is standard for serving static files efficiently while preventing unauthorized domains from accessing it and ensuring proper error handling and browser-side caching

**Dockerfile**

# Build stage

FROM node:18 AS build

WORKDIR /app

COPY package\*.json ./

RUN npm ci

COPY . .

RUN npm run build

# Production stage with Nginx

FROM nginx:stable-alpine

# Copy built files into /usr/share/nginx/html/remote

COPY --from=build /app/dist /usr/share/nginx/html/remote

# Copy the fixed Nginx config

COPY nginx.conf /etc/nginx/conf.d/default.conf

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]

This Dockerfile sets up a multi-stage build with Node.js to build a frontend application, and then uses an Nginx container to serve the built static files with a custom Nginx configuration.

**Build Stage**

* FROM node:18 AS build: Uses the official Node.js 18 image as the build environment.
* WORKDIR /app: Sets the working directory in the container to /app.
* COPY package\*.json ./: Copies package.json and package-lock.json to the container.
* RUN npm ci: Installs dependencies exactly as specified in the lock file for reproducible builds.
* COPY . .: Copies all source files into the container.
* RUN npm run build: Runs the build script from package.json, typically producing optimized static files in a dist directory.

**Production Stage with Nginx**

* FROM nginx:stable-alpine: Uses the stable Alpine Linux version of the official Nginx image, which is lightweight and suitable for production.
* COPY --from=build /app/dist /usr/share/nginx/html/remote: Copies the built static files from the build stage into Nginx's static file directory under a subdirectory remote.
* COPY nginx.conf /etc/nginx/conf.d/default.conf: Copies a custom Nginx configuration file into the container, replacing the default server block.
* EXPOSE 80: Declares that the container listens on port 80.
* CMD ["nginx", "-g", "daemon off;"]: Starts Nginx in the foreground so the container stays running.

**How it Works Together**

* The first stage compiles the frontend application to static assets.
* The second stage provides a minimal container with Nginx to serve those static files efficiently.
* Custom Nginx config allows routing, caching, or any other optimizations.
* The static files are served from /remote/ as per the earlier provided Nginx config.
* This multi-stage build results in a smaller final image without build dependencies.

This setup is a widely used Docker pattern for deploying frontend apps with a production-ready Nginx server inside a container, ensuring consistent builds and optimized delivery of static assets.

HOST **App**

**package.json**

{

  "name": "host-app",

  "version": "1.0.0",

  "scripts": {

    "build": "webpack --mode production"

  },

  "dependencies": {

    "react": "^18.3.1",

    "react-dom": "^18.3.1"

  },

  "devDependencies": {

    "@babel/core": "^7.21.0",

    "@babel/preset-react": "^7.21.0",

    "babel-loader": "^9.1.2",

    "html-webpack-plugin": "^5.5.0",

    "webpack": "^5.88.0",

    "webpack-cli": "^5.1.2"

  }

}

**.babelrc**

{

"presets": ["@babel/preset-react"]

}

**webpack.config.js**

const path = require('path');

const HtmlWebpackPlugin = require('html-webpack-plugin');

const { ModuleFederationPlugin } = require('webpack').container;

module.exports = {

    entry: './src/index',

    mode: 'production',

    output: {

        publicPath: '/',

        filename: '[name].[contenthash].js',

        path: path.resolve(\_\_dirname, 'dist'),

        clean: true,

    },

    plugins: [

        new ModuleFederationPlugin({

            name: 'hostApp',

            remotes: {

                remoteApp: 'remoteApp@http://localhost:3001/remote/remoteEntry.js',

            },

            shared: {

                react: { singleton: true, requiredVersion: '^18.3.1',eager:true },

                'react-dom': { singleton: true, requiredVersion: '^18.3.1',eager:true },

            },

        }),

        new HtmlWebpackPlugin({ template: './public/index.html' }),

    ],

    module: {

        rules: [

            { test: /\.jsx?$/, loader: 'babel-loader', exclude: /node\_modules/ },

        ],

    },

    resolve: { extensions: ['.js', '.jsx'] },

};

**remoteApp: 'remoteApp@http://localhost:3001/remote/remoteEntry.js', This is nginx server location, not** docker **host or webpack server location**

**src/index.js**

import React, { Suspense, lazy } from 'react';

import ReactDOM from 'react-dom';

const RemoteWidget = lazy(() => import('remoteApp/Widget'));

const App = () => (

  <div>

    <h1>Host App</h1>

    <Suspense fallback={<div>Loading Remote Widget...</div>}>

      <RemoteWidget />

    </Suspense>

  </div>

);

ReactDOM.render(<App />, document.getElementById('root'));

**public/index.html**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8" />

<meta name="viewport" content="width=device-width, initial-scale=1" />

<title>Host App</title>

</head>

<body>

<div id="root"></div>

</body>

</html>

**nginx.conf**

server {

  listen 80;

  server\_name localhost;

  root /usr/share/nginx/html;

  index index.html;

  location / {

    try\_files $uri $uri/ /index.html;

  }

  location ~\* \.(js|css|png|jpg|jpeg|gif|ico|svg)$ {

    expires 30d;

    add\_header Cache-Control "public, no-transform";

  }

}

This nginx configuration sets up a simple server that serves static files from a specified directory and handles client requests with specific caching rules.

**Key Components of the Configuration**

**Listen and Server Name**

* listen 80;: Tells nginx to listen for incoming HTTP requests on port 80, the default port for unencrypted web traffic.​
* server\_name localhost;: Specifies that this server block should handle requests aimed at localhost, which is typically used for local development or testing.​

**Root Directory and Default Page**

* root /usr/share/nginx/html;: Sets the root directory for serving static files, meaning any request not matched by further rules will look for matching files inside /usr/share/nginx/html.
* index index.html;: Defines the default file to serve if a directory is requested, usually index.html.​

**Request Handling**

* location / { try\_files $uri $uri/ /index.html; }: This is a crucial block, especially for Single Page Applications (SPAs).
  + It tries to serve the requested URI ($uri) as a file.
  + If it doesn't find the file, it tries to serve the URI as a directory ($uri/).
  + If neither is found, it falls back to serve /index.html.
  + This setup allows the SPA to handle client-side routing, ensuring the application loads correctly even when navigating directly to nested routes.​

**Static Files Caching**

* location ~\* \.(js|css|png|jpg|jpeg|gif|ico|svg)$ { ... }: This regular expression location block matches static resources based on file extensions.
* expires 30d;: Sets browser cache expiry to 30 days, reducing server load and improving load times for repeat visitors.
* add\_header Cache-Control "public, no-transform";: Adds HTTP headers instructing browsers and intermediate caches that the resources are public and should not be transformed during transit.​

**How it Works Together**

* Requests for static assets like JavaScript, CSS, or images will be cached for 30 days to optimize performance.
* Requests for other resources or client app routes will fallback to index.html, enabling client-side routing with frameworks like React, Angular, or Vue.
* The server responds only for requests targeted at localhost on port 80, making it suitable mainly for local development or containerized environments.​

Overall, this configuration is typical for hosting a Single Page Application with nginx, providing efficient static asset delivery and smooth client-side routing support.

Related

Explain how try\_files falls back to index.html in this config

Show how to serve the app at root URL instead of /remote

Add gzip and Cache-Control optimizations for static assets

Configure long-term asset hashing and cache-busting with Nginx

How to restrict access or add security headers for this server

**Dockerfile**

# Build stage

FROM node:18 AS build

WORKDIR /app

COPY package\*.json ./

RUN npm ci

COPY . .

RUN npm run build

# Serve with Nginx

FROM nginx:stable-alpine

COPY --from=build /app/dist /usr/share/nginx/html

COPY nginx.conf /etc/nginx/conf.d/default.conf

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]

This Dockerfile uses a multi-stage build to produce an optimized container for a Node.js frontend application served by Nginx.

**Build Stage**

* Base image: node:18 which includes Node.js runtime and npm package manager.
* Working directory: /app inside the container.
* Dependency install: Copies package.json and package-lock.json, then runs npm ci for clean, reproducible installs.
* Source copy: Copies all app source files into the container.
* Build command: Runs npm run build to produce an optimized static build of the frontend, typically output in a dist directory.

**Production Stage**

* Base image: nginx:stable-alpine, a lightweight Nginx server image on Alpine Linux.
* Copy build artifacts: Copies the static files from /app/dist in the build stage to Nginx's default static directory /usr/share/nginx/html.
* Custom config: Copies a user-defined Nginx config file nginx.conf into the container to override the server block settings.
* Expose port: Declares port 80 for HTTP traffic.
* Start command: Runs Nginx in the foreground (daemon off) so the container stays active.

**Benefits and Workflow**

* The multi-stage build ensures the final production image contains only the static files and the minimal Nginx runtime, reducing image size and attack surface.
* The build tools and Node environment are excluded from the final image.
* Separating build and runtime concerns increases security and performance.
* The custom Nginx config allows fine-tuned routing, caching, and other server behaviors tailored to the frontend app.

This is a common best practice for deploying modern JavaScript frontend applications such as those built with React, Vue, or Angular inside Docker containers using Nginx to serve static content efficiently and securely

**Docker Compose (optional orchestration)**

Docker-compose.yml

services:

  remote:

    build:

      context: ./remote

    container\_name: remote\_app

    ports:

      - "3001:80"

    restart: unless-stopped

  host:

    build:

      context: ./host

    container\_name: host\_app

    ports:

      - "8080:80"

    restart: unless-stopped

    depends\_on:

      - remote

This Docker Compose configuration defines two services named remote and host. Here's the detailed explanation:

**Service: remote**

* Build context: The Dockerfile and related files are located in the ./remote directory.
* Container name: The container will be named remote\_app for easier identification.
* Ports: The container's port 80 (standard HTTP) is mapped to port 3001 on the host machine, allowing access via localhost:3001.
* Restart policy: The container will restart automatically unless stopped explicitly, improving resilience.

**Service: host**

* Build context: The Dockerfile here is under the ./host directory.
* Container name: Named host\_app.
* Ports: Maps container port 80 to host port 8080, accessible at localhost:8080.
* Restart policy: Same as remote, it will restart unless stopped.
* Dependencies: The depends\_on field specifies that host service depends on the remote service. This means Docker Compose will start the remote service first before starting host, ensuring proper startup order and availability of the remote service to host.

**How It Works Together**

* Running docker-compose up will build and run both services.
* remote runs independently and serves its function on port 3001.
* host service waits until remote service is up and running before starting.
* This is typical in microservices or front + backend deployments where one service depends on another being available.

This configuration, by defining separate build contexts and containers for each service with explicit dependency, allows modular, maintainable, and orchestrated deployments for multi-component applications with Docker Compose

**4. Build and Run**

# Build remote app image

cd remote-app

npm i

docker build -t remote-app-prod .

# Build host app image

cd ../host-app

npm i

docker build -t host-app-prod .

# Run containers with docker-compose if using

**docker-compose up –build**

**Summary**

* Both apps are Webpack-built for production, output to /dist
* Nginx serves static builds in Docker containers
* Module Federation loads remote components via URLs through Nginx
* Docker Compose orchestrates both containers locally

**Verify inside the container**

* Run shell inside your remote app container:

bash

docker exec -it <container\_name> sh

* Then:

bash

ls /usr/share/nginx/html/remote

Confirm that remoteEntry.js (or remoteEntry.[contenthash].js if you use hashes) exists inside.