Building Tomorrow's Web

Micro Frontend with Module Federation 2.0

Mahendra Hirapra

FE Engineer @EWD

Rohit Ranjan

FE Engineer @EWD

Manas Diyali

FE Engineer @EWD

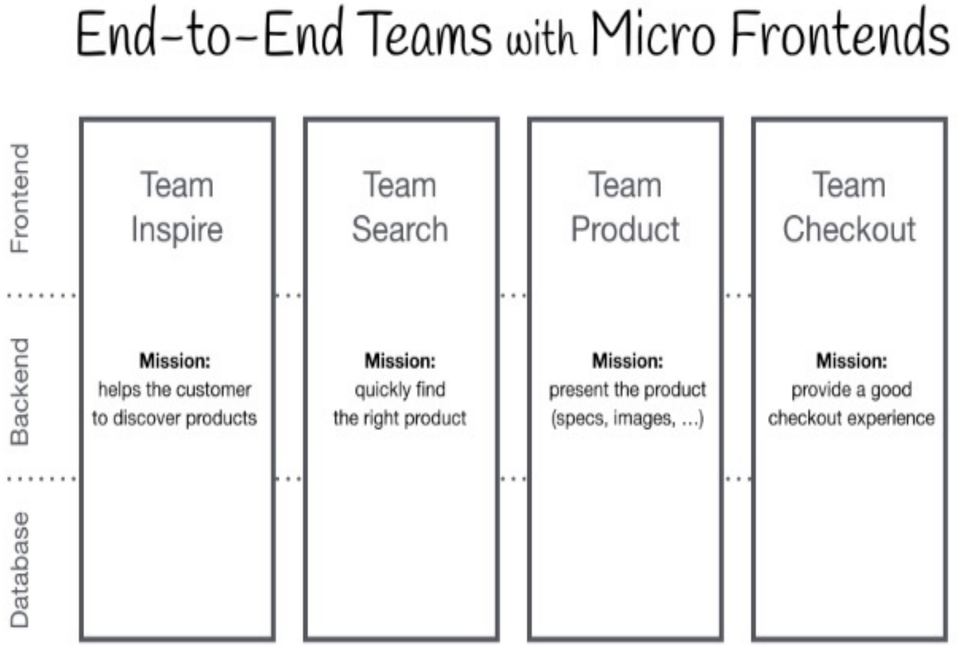
Eurowings Digital

Agenda

1. Introduction 
2. Brief introduction to Micro Frontend
3. Why not Monolith
4. What is module Federation
5. Motivations to choose module federation
6. Hands-on and code walkthrough
7. Q&A

Micro frontend

- It extends the concepts of micro services to the frontend world
- Application is divided into different modules or subdomains, implemented independently.
- Allowing frontend teams, the same level of flexibility and speed that microservices provide to backend teams.



Why not Monolith

- Atrociously complex and slow release cycles
- A monolith builds the entire system every time, even for minor bug fixes and improvements.
- They require a steeper learning curve as a team members needs to learn about system-level complexities.
- Expensive Tech Migrations.

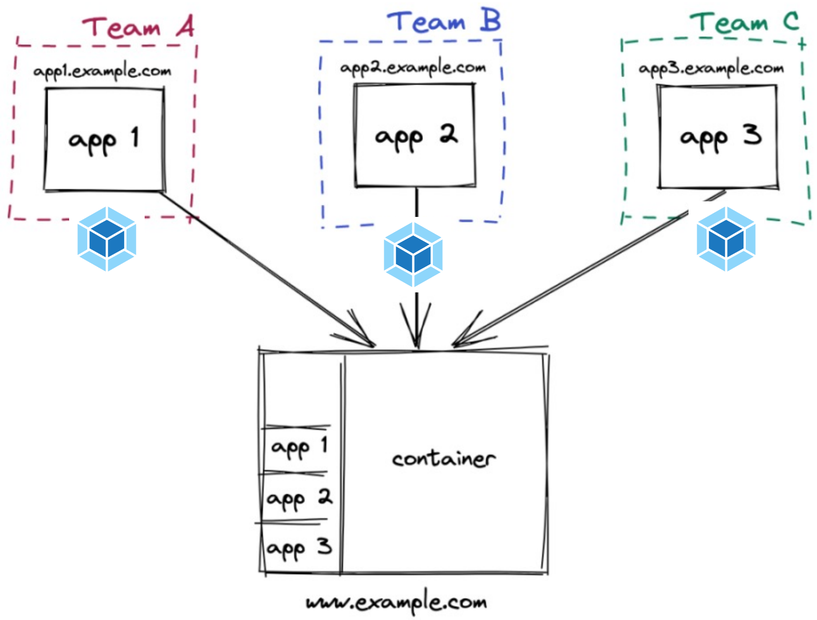
One size doesn’t fit all

Ways to Implement Micro-frontend

- Using Module Federation
- Runtime Integration with I-frame.
- Build-time Integration by importing it as a 3rd party package.
- Using Web Components

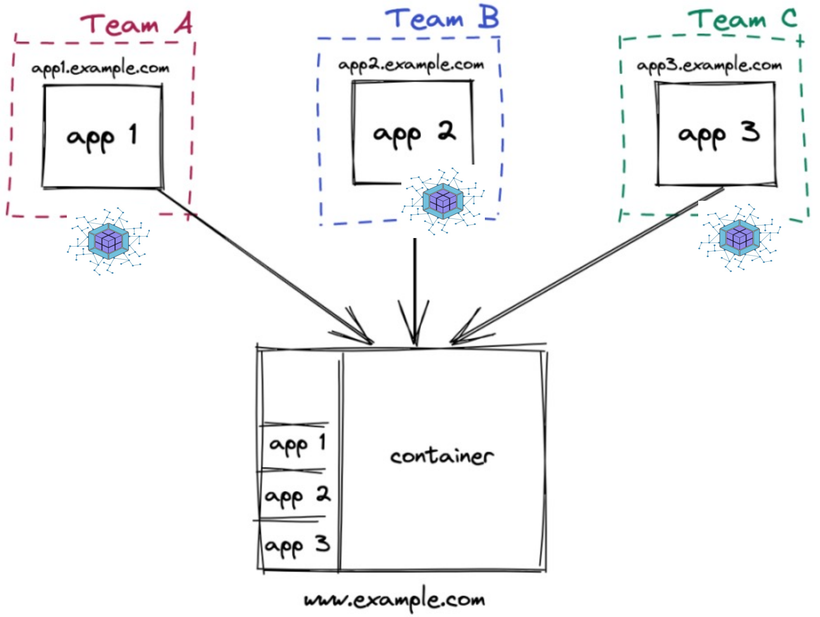
Module Federation

- Module federation is an advanced feature in Webpack that provides a way for a JavaScript application to dynamically load code from another application.
- This feature allows for efficient code sharing and dependency management.
- Powerful tool for building modern web applications
- Module Federation has become an architectural method for building large web applications, similar to microservices in the backend.



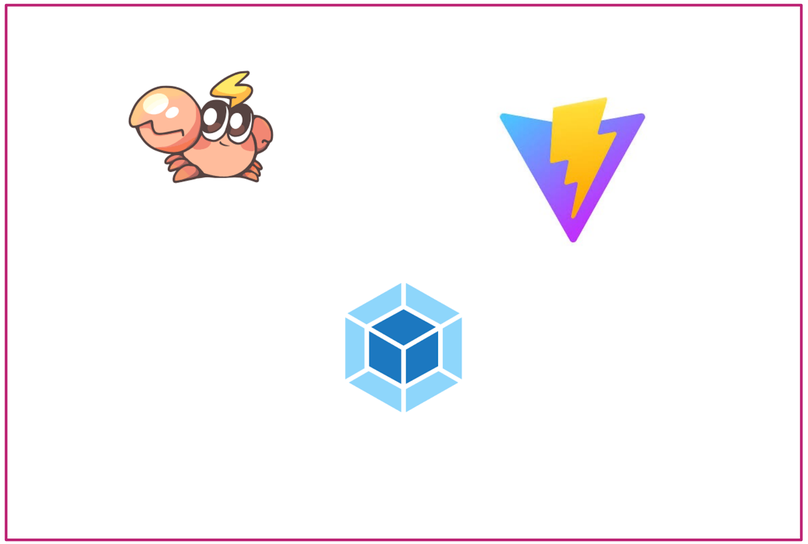
Module Federation 2.0

- Module Federation 2.0 differs from the Module Federation built into **Webpack v5** by providing not only the core features of module export, loading, and dependency sharing but also additional features as well like type hinting, Manifest, Federation Runtime, and Runtime Plugin System
- This feature allows for efficient code sharing and dependency management.



Features

- Federation Runtime is one of the main features of the new version of Module Federation. It supports registering shared dependencies at runtime, dynamically registering and loading remote modules.
- Supports various build tools like Webpack and Rspack.
- Module Federation products also generate types and enjoy type hot reloading.
- Visualization and proxy capabilities provided by Chrome DevTools



Why Module Federation

- Really seamless to integrate
- Easier to migrate from a monolith to modular components.
- Gives you the flexibility to choose build tools and frameworks for different Micro-frontends
- Growing community support .

Breaking From Monolith to Micro-frontend

- **Decompose the Monolith** :

Identify and decompose the monolithic frontend into smaller, cohesive units based on business capabilities.

- **Define Communication Contracts** :

Establish clear communication contracts and APIs between Micro Frontends to ensure seamless integration and a consistent user experience.

- **Use Gateway or Shell** :

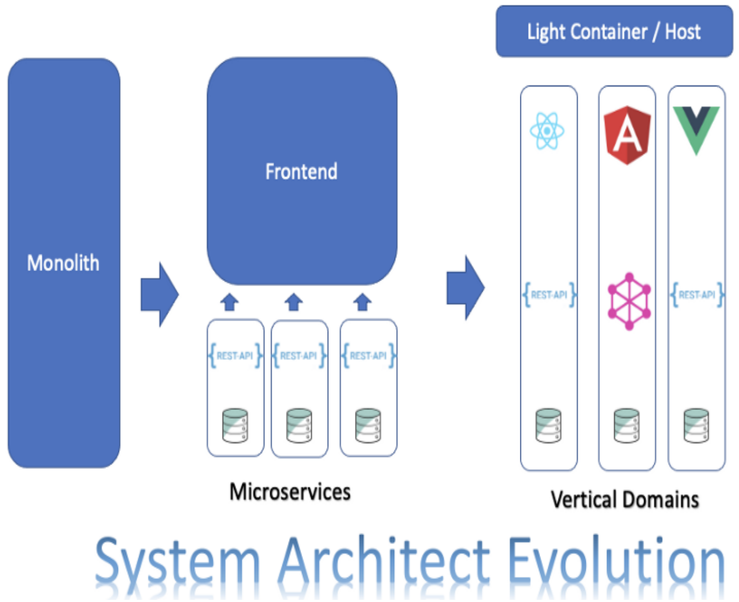
Implement a shell or gateway application that orchestrates the composition of Micro Frontends and manages cross-cutting concerns.

- **Select Technologies Wisely** :

Choose technologies based on the specific requirements of each Micro Frontend. Leverage containerization for deployment consistency.

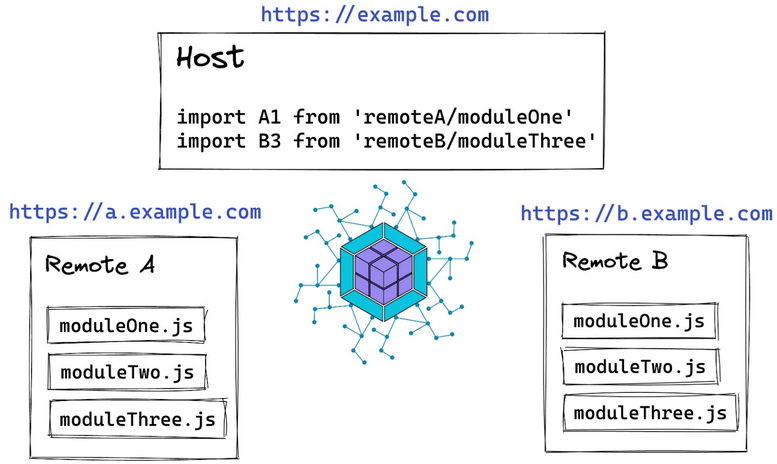
- **Continuous Integration and Deployment** :

Implement CI/CD pipelines for each Micro Frontend to support independent testing, deployment, and release cycles.



Architecture Blocks

1. Exposed Module (Remote): Exposes specific modules for other projects
2. Consumption Module (Host Remote Import): Consumes remote modules
3. Shared Module/Dependency: Shares dependencies across modules



Eurowings

Steps:

1. Identify the component to be separated.
2. Start with the setup of producer component.
3. Add the producer configuration for module federation, exposing the separated module.
4. After producer is up and running , start with the configuration of consumer to integrate the producer i.e micro-frontend module, along with shared libraries.

Hands On

Code Snippet Producer and Consure with 3rd Part Module Sharing

Producer

```ts
import { defineConfig } from '@rsbuild/core';
import { pluginReact } from '@rsbuild/plugin-react';
import { ModuleFederationPlugin } from '@module-federation/enhanced/rspack';
import { dependencies } from './package.json';

export default defineConfig({
  server: {
    port: 3001,
  },
  dev: {
    // It is necessary to configure assetPrefix, and in the production environment, you need to configure output.assetPrefix
    assetPrefix: 'http://localhost:3001',
  },
  tools: {
    Pieces: Comment | Pieces: Explain
    rspack: (config, { appendPlugins }) => {
      // You need to set a unique value that is not equal to other applications
      config.output!.uniqueName = 'remote_component';
      appendPlugins([
        new ModuleFederationPlugin({
          name: 'remote_component',
          exposes: {
            './playButton': './src/components/PlayerControlButton.tsx',
          },
          shared: {
            react: {
              singleton: true,
              requiredVersion: dependencies['react'],
            },
            'react-dom': {
              singleton: true,
              requiredVersion: dependencies['react-dom'],
            },
            '@mui/material': {
              singleton: true,
              requiredVersion: dependencies['@mui/material'],
            },
            '@mui/icons-material': {
              singleton: true,
              requiredVersion: dependencies['@mui/icons-material'],
            },
            '@mui/system': {
              singleton: true,
              requiredVersion: dependencies['@mui/system'],
            },
          },
        }),
      ]);
    },
  },
  plugins: [pluginReact()],
});
```

Consumer

```ts
import { dependencies } from './package.json';

export default defineConfig({
  html: {
    template: './index.html',
  },
  source: {
    entry: {
      index: './src/main.tsx',
    },
  },
  tools: {
    Pieces: Comment | Pieces: Explain
    rspack: (config, { appendPlugins }) => {
      appendPlugins([
        new ModuleFederationPlugin({
          name: 'federation_consumer',
          remotes: {
            remote_component:
              'remote_component@http://localhost:3001/mf-manifest.json',
          },
          shared: {
            react: {
              singleton: true,
              requiredVersion: dependencies['react'],
            },
            'react-dom': {
              singleton: true,
              requiredVersion: dependencies['react-dom'],
            },
            '@mui/material': {
              singleton: true,
              requiredVersion: dependencies['@mui/material'],
            },      You, 12 hours ago • Uncommitted changes
            '@mui/icons-material': {
              singleton: true,
              requiredVersion: dependencies['@mui/icons-material'],
            },
            '@mui/system': {
              singleton: true,
              requiredVersion: dependencies['@mui/system'],
            }
          },
        }),
      ]);
    },
  },
  plugins: [pluginReact()],
});
```

Module federation Bridge

The MF bridge is a utility function provided by Module Federation to help users load application-level modules through Module Federation. it allows proper routing collaboration between applications.

Why use Module federation Bridge

Bridge is mainly used to solve two problems:

- Cross-application framework (React, Vue) loading and rendering
- Support for loading modules with routes (routes can work together properly)

Hands On

Federation Runtime

Federation Runtime is one of the main features of the new version of Module Federation.
It supports registering shared dependencies at runtime, dynamically registering and loading remote modules.

API

```
// You can use the runtime to load modules without depending on the build plugin
// When not using the build plugin, shared dependencies cannot be automatically configured
import { init, loadRemote } from '@module-federation/enhanced/runtime';

init({
  name: '@demo/app-main',
  remotes: [
    {
      name: "@demo/app1",
      // mf-manifest.json is a file type generated in the new version of Module Federation build tools, providir
      // Preloading depends on the use of the mf-manifest.json file type
      entry: "http://localhost:3005/mf-manifest.json",
      alias: "app1"
    },
    {
      name: "@demo/app2",
      entry: "http://localhost:3006/remoteEntry.js",
      alias: "app2"
    },
  ],
});

// Load using alias
loadRemote<{add: (...args: Array<number>)=> number }>("app2/util").then((md)=>{
  md.add(1,2,3);
});
```

Use case

- Can be used independently of build plugins and can directly use pure runtime for module loading.
- No need to install excessive additional dependencies and build configurations, just declare the module name and the path of the exported module to complete the module export.

Hands On

Challenges and Considerations

- **Identifying modules / setting up domain boundaries:**

- **Communication between Micro Frontends** :
Coordinating communication and ensuring a seamless user experience across Micro Frontends can be challenging. Well-defined APIs and communication contracts are crucial.

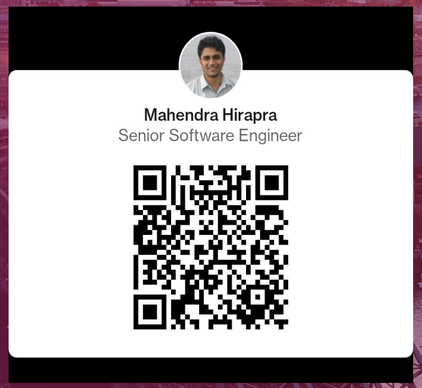
- **Initial Setup and Learning curve**:
Setting up a Micro Frontends architecture requires careful planning and may involve overcoming initial challenges related to integration and tooling.

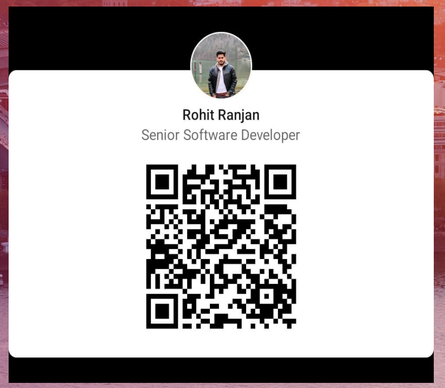
- **Cross-Cutting Concerns** :
Managing cross-cutting concerns, such as authentication, routing, and global state, requires thoughtful design and coordination among Micro Frontends.

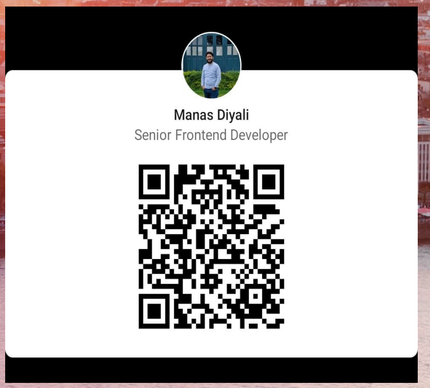
THANK YOU

Q&A

Jobs@EWD







Eurowings Digital