Microfrontends, las dos caras de la moneda

Ismael Rodríguez González - Arquitecto de aplicaciones

Sergio Alonso Castaño - Arquitecto de soluciones

{codemotion}
Online Tech Conference
- Spanish edition -

20-21-22 abril, 2021

¿Qué es?

- Es un tipo de arquitectura FrontEnd donde tratamos de resolver los mismos problemas que se solucionan con los micro-servicios (romper el monolito!).
- Dividimos la funcionalidad en piezas **más pequeñas e independientes**, que se puedan desarrollar, testear y desplegar por separado.
- El usuario final debe percibirla como una única aplicación.

¿Qué **NO** es?

- No es ninguna tecnología en concreto
- No es un framework, aunque existen soluciones que implementan este patrón como Single-SPA o qiankun, entre otros
- No se trata de un estándar (aún).

En teoría

- Beneficios de un Sistema modular, piezas más pequeñas, cohesionadas y mantenibles.
- Cada microfrontend es una pieza independiente del resto
- Escalable a nivel organizativo, con equipos independientes trabajando en distintas aplicaciones.
- Cada microfrontend se desarrolla con la tecnología más adecuada para su caso de uso.
- Posibilidad de actualizar o modificar microfrontends sin afectar al resto de la aplicación.
- Ciclo de vida independiente (y despliegue por separado).

Los Microfrontends son increíbles

Hagamos todo con microfrontends! o quizá no

atmira

En la practica

Look & feel

Cada microfrontend no puede “hacer la guerra” por su cuenta

- Creación de un catalogo de componentes que los microfrontends consuman

- Utilizar CSS in JS, metodología BEM o anidamiento de estilos con SASS para que los estilos no colisionen.

En la practica

Duplicidad de código

- Crear librerías de funcionalidad transversal
- Catálogo de componentes
- Funcionalidad core expuesta desde el contenedor
- Cuidado con la retrocompatibilidad y versionado

En la practica

Dependencias repetidas

El navegador se tiene que descargar la misma dependencia por cada microfrontend, por ejemplo React.

- Más robusto. Cada microfrontend gestiona sus dependencias y es independiente del resto.
- Mejor rendimiento. Se pueden compartir dependencias comunes usando federación de módulos.

En la practica

Microfrontend Anarchy

Cada micro puede estar escrito en un framework diferente, pero ¿es lo que queremos?

- Compartir código puede ser más difícil o imposible.
- Más costoso de mantener.
- El navegador se tendría que descargar muchas mas dependencias.
- Consenso tecnológico para todos los microfrontends.
- En el futuro se puede decidir incorporar nuevos frameworks, siempre de forma meditada.

atmira

Casos de uso

Migración de grandes aplicaciones monolíticas.

- Al utilizar microfrontends es posible migrar poco a poco la aplicación legacy siguiendo un patrón de “estrangulación”

- En vez de reescribir todo y lanzar la nueva aplicación (big bang) lo que se hace es ir añadiendo partes nuevas que sustituyen a las antiguas, esto se hace progresivamente hasta que no queda nada de la aplicación legacy.

Casos de uso

Simplificar el desarrollo de grandes aplicaciones con muchas partes diferenciadas.

- Reducimos la complejidad de desarrollo de cada microfrontend al acotar su alcance (a costa de aumentar la complejidad a nivel general)

- Aumentamos el desacoplamiento lo que nos lleva a menores posibilidades de colisión entre elementos de la aplicación, más facilidades para evolucionar cada microfrontend por separado, test más sencillos y en general los beneficios inherentes a un sistema menos acoplado.

Casos de uso

La organización de la empresa/proyecto encaja con microfrontends

- Queremos múltiples equipos independientes que trabajen en paralelo.
- Nos interesa dividir la aplicación en “partes” independientes con su propio ciclo de vida y despliegue por separado.
- Queremos una solución tecnológicamente agnóstica y no vincular nuestro proyecto a ningún framework concreto.

Elementos del proyecto

Contenedor

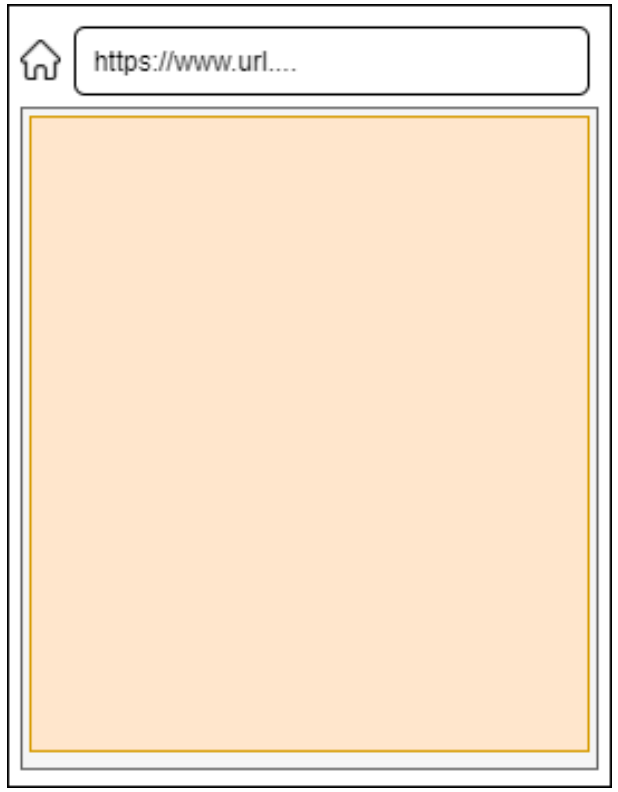
- Gestiona la carga y descarga de los microfrontends.
- Intercepta la navegación y actúa en base a ella.
- Puede proporcionar ciertos métodos transversales como el routing, funcionalidad para compartir estado o elementos de seguridad.

Microfrontends

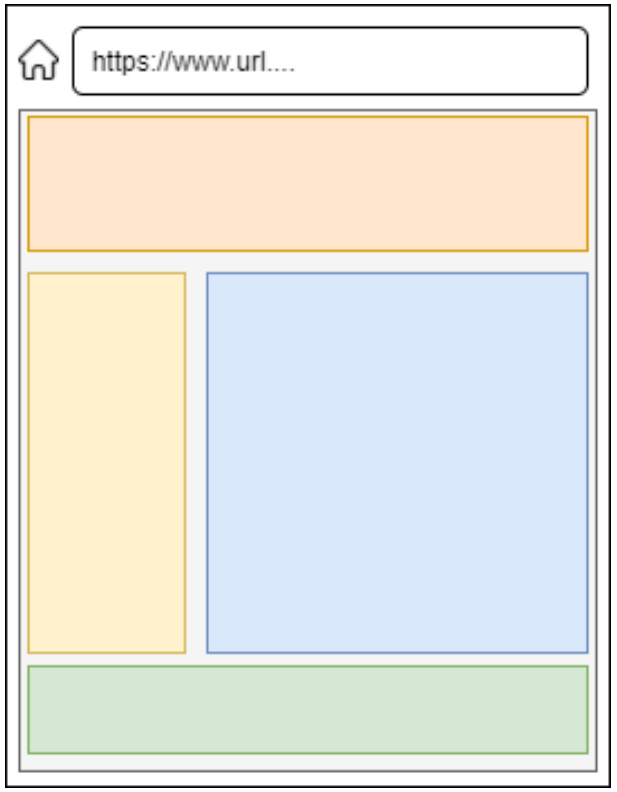
- Cada una de las aplicaciones independientes que componen el proyecto, se pueden organizar de distintas formas.

Elementos del proyecto

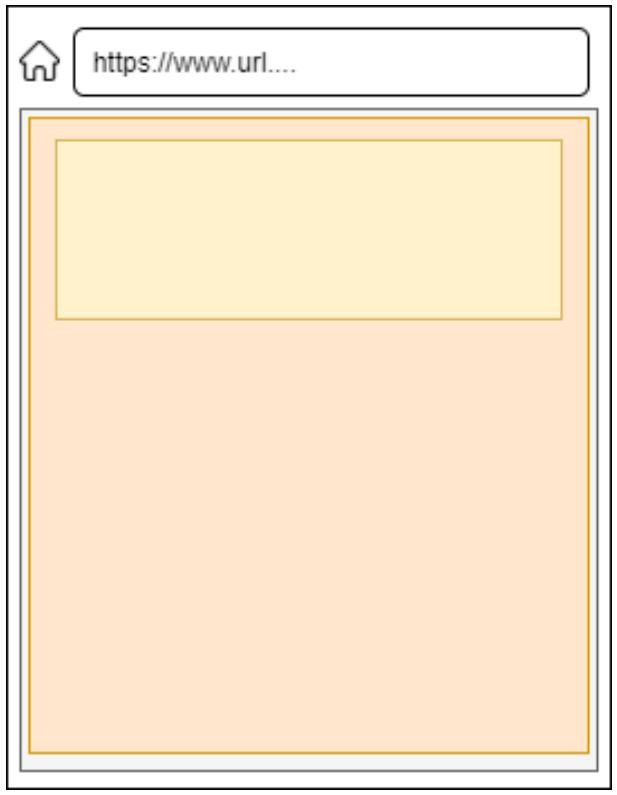
Único por ruta



Composición



Anidamiento



atmira

Conclusiones

¿Cuándo utilizar microfrontends?

- Se trata de un proyecto medio/grande - No matar moscas a cañonazos –
- No dejarse llevar por el hype
- El caso de uso se adapta a este tipo de arquitectura
- Las ventajas superan a los inconvenientes o existe una “key feature”, por ejemplo despliegue independiente, que encaje con el proyecto.
- Estamos dispuestos a lidiar con una mayor complejidad general.
- Decidir estrategia de versionado, granularidad y como se van a compartir los micros o las diferentes “partes”

Construcción

Tipos de solución

Server time

- Se compone el template en servidor con los distintos microfrontends y se devuelve el HTML al navegador.
- El cambio con respecto a un server-render tradicional es más a nivel organizativo que en cuanto a tecnologías.

Build time

- La composición se hace en tiempo de construcción, el contenedor junto a los microfrontends.
- Se pueden de-duplicar dependencias en construcción, pero se niega el despliegue independiente de los microfrontends.

Run Time

- La composición de los micros la hace el contenedor en tiempo de ejecución.
- Se trata de la opción más compleja pero también la más flexible y la que se suele utilizar.

Soluciones RunTime

Iframes

- Buen aislamiento entre microfrontends
- Difícil comunicarse con micros
- Problemas responsive

Web Components

- Buena idea que no acaba de materializarse
- Nivel de uso reducido
- ¿Tecnología muerta?

Ecmascript

- No tiene los problemas de las otras soluciones run-time
- Podemos crear una solución 100% adaptada a nuestras necesidades
- En continua evolución, como ejemplo tenemos la reciente creación de una nueva herramienta que resuelve algunos inconvenientes de un proyecto microfrontend, “Federación de módulos”

Module Federation

Creados por **Zack Jackson** e incorporado a la versión 5 de Webpack

Ejemplos: https://github.com/module-federation/module-federation-examples

Documentación: https://webpack.js.org/concepts/module-federation

Características

- Los módulos federados nos ofrecen un nuevo método para compartir código entre aplicaciones.
- Soluciona los problemas de la duplicidad de dependencias entre aplicaciones.
- Cada aplicación expone componentes que pueden ser consumidos por otras aplicaciones.
- Permite obtener estos componentes en tiempo de ejecución.
- No requiere de cambios en el code-base, es framework-les.
- Afecta a cualquier parte de nuestra aplicación, no solo componentes UI.
- Funciona con cualquier tipo de fichero que Webpack es capaz de procesar, no solo javascript.

Module Federation

Configuración App con módulos federados

- Todo se configura dentro de Webpack
- Elementos de la configuración:
  - Name: Nombre con el que identificar nuestra configuración.
  - Library: Tipo de librería que creará webpack (commonJS, SystemJS…).
  - Filename: Fichero que creará Webpack con nuestros módulos a federar.
  - Remotes: Objeto con los módulos que mi App consume
  - Exposes: Objeto con los módulos que mi App expone
  - Shared: Objeto con las dependencias a compartir.

```
const Header = React.lazy(() => import("mf-nav/Header"));
const Footer = React.lazy(() => import("mf-nav/Footer"));
```

```
new ModuleFederationPlugin({
  name: "home",
  library: { type :"var", name:"home"},
  filename: "remoteEntry.js",
  remotes: {
    "mf-nav": "nav@http://localhost:8081/remoteEntry.js",
  },
  exposes: {},
  shared: {
    ...deps,
    nav: {
      singleton: true
    },
    react: {
      singleton: true,
      requiredVersion: deps.react,
    },
    "react-dom": {
      singleton: true,
      requiredVersion: deps["react-dom"],
    },
  },
}),
```

```
new ModuleFederationPlugin({
  name: "nav",
  filename: "remoteEntry.js",
  library: { type :"var", name:"nav"},
  remotes: {},
  exposes: {
    "./Header": "./src/Header",
    "./Footer": "./src/Footer"
  },
  shared: {
    ...deps,
    react: {
      eager:true,
      singleton: true,
      requiredVersion: deps.react,
    },
    "react-dom": {
      eager:true,
      singleton: true,
      requiredVersion: deps["react-dom"],
    },
  },
}),
```

atmira

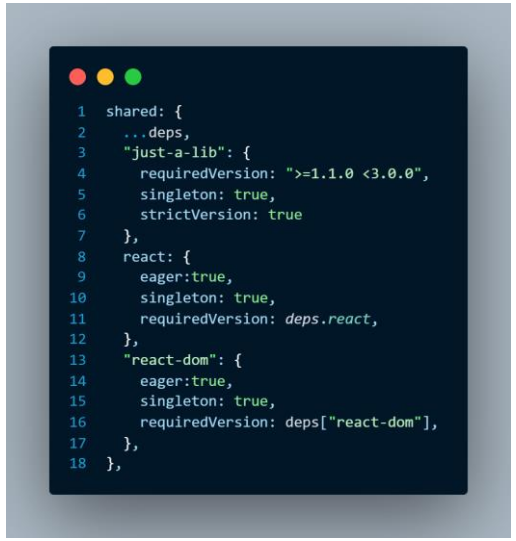
Module Federation

Evitar la duplicidad de librerías

- “Module Federation” nos permite compartir librerías entre todos nuestros módulos
- Con esto soluciona el problema de que cada MFE tenga sus propias dependencias, muchas de ellas duplicadas.
- Usa Semver para escoger que librería será la que usará en su contexto de ejecución global
- Permite que un módulo federado no se ejecute si una librería no cumple con los requisitos de versión marcados.
- Toda esta configuración irá dentro del objeto shared del plugin de “Module Federation”

Casos de uso

- Dos aplicaciones usan librerías compatibles entre si.
- Dos aplicaciones usan librerías no compatibles entre si.
- Dos aplicaciones usan librerías no compatibles entre si, configuradas como singleton.
- Dos aplicaciones usan librerías no compatibles entre si, configuradas con comprobación de versiones.
- Dos aplicaciones usan librerías no compatibles entre si, pero aceptando un rango de versiones.



```
shared: {
  ...deps,
  "just-a-lib": {
    requiredVersion: ">=1.1.0 <3.0.0",
    singleton: true,
    strictVersion: true
  },
  react: {
    eager:true,
    singleton: true,
    requiredVersion: deps.react,
  },
  "react-dom": {
    eager:true,
    singleton: true,
    requiredVersion: deps["react-dom"],
  },
},
```

atmira

Module Federation

Compartiendo librerías compatibles entre si

- Configuramos los shared de todos nuestros módulos federados con versiones de librerías compatibles entre si

Package.json y webpack config

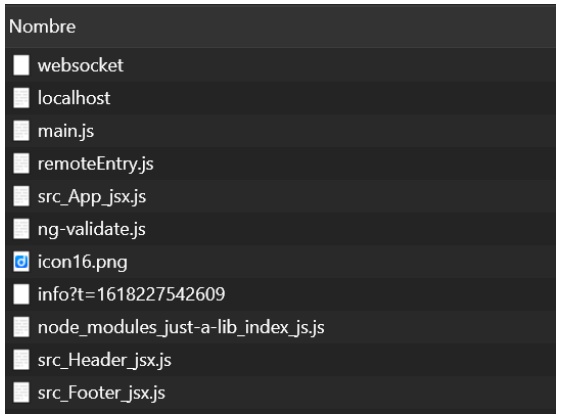
- Dentro de los package.json de las dos aplicaciones de ejemplo tenemos la misma versión de “just-a-lib”*.
- Dentro de nuestras configuraciones de webpack configuramos los shared a la versión que tenga el propio package.json.

App Shell

App Nav

Ejecución

- Vemos que cada aplicación tiene la misma versión, pero que el navegador solo se baja una sola vez la librería.



*just-a-lib: Librería de ejemplo que nos devuelve su propia versión

atmira

Module Federation

Compartiendo librerías no compatibles entre si

- Configuramos los shared de todos nuestros módulos federados con versiones de librerías no compatibles entre si

Package.json y webpack config

- Dentro de los package.json de las dos aplicaciones de ejemplo tenemos distintas versiones de “just-a-lib”*.
- Dentro de nuestras configuraciones de webpack configuramos los shared a la versión que tenga el propio package.json.

App Shell

```
"dependencies": {
  "just-a-lib": "~1.0.0",
  "react": "^17.0.2",
  "react-dom": "^17.0.2"
}
```

```
"just-a-lib": {
  requiredVersion: deps["just-a-lib"]
},
```

App Nav

```
"dependencies": {
  "just-a-lib": "^1.1.0",
  "react": "^17.0.2",
  "react-dom": "^17.0.2"
}
```

```
"just-a-lib": {
  requiredVersion: deps["just-a-lib"]
},
```

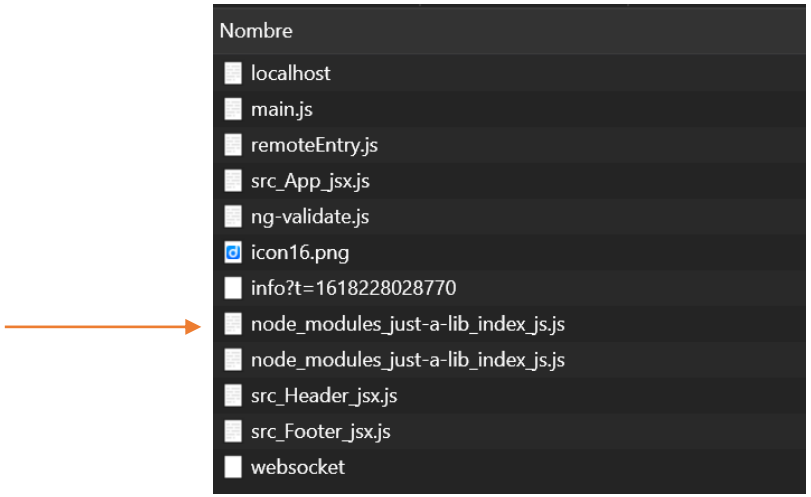
Ejecución

- Vemos que cada aplicación tiene su propia versión y que el navegador se baja cada versión de la librería.

HEADER lib version 1.1.0!

Hi there, I'm React from React.
Component lib version: 1.0.1

Footer!



*just-a-lib: Librería de ejemplo que nos devuelve su propia versión

atmira

Module Federation

Compartiendo librerías no compatibles entre si con singleton

- Configuramos los shared de todos nuestros módulos federados con versiones de librerías no compatibles entre si y con el flag a singleton a true, con ello obligamos a webpack a que solo exista una versión de la librería en el contexto de ejecución.

Package.json y webpack config

- Dentro de los package.json de las dos aplicaciones de ejemplo tenemos distintas versiones de “just-a-lib”*.
- Dentro de nuestras configuraciones de webpack configuramos los shared a la versión que tenga el propio package.json y singleton a true.

App Shell

App Nav

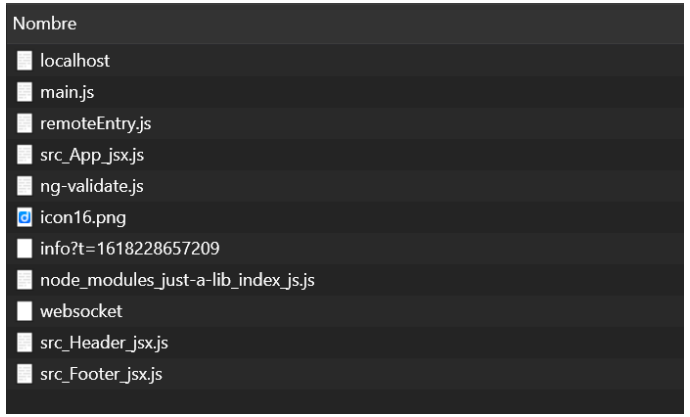
Ejecución

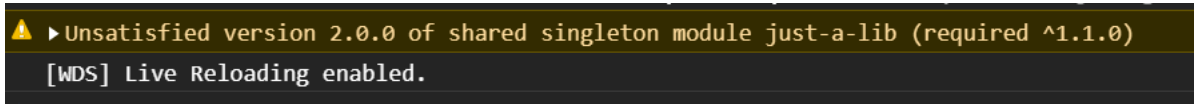
- Todas las aplicaciones que compartan la librería usarán la misma versión, la mas alta disponible, en este caso la 2.0.0, pero tendremos un mensaje de advertencia con la discrepancia de versión.

HEADER lib version 2.0.0!

Hi there, I'm React from React.
Component lib version: 2.0.0

Footer!





*just-a-lib: Librería de ejemplo que nos devuelve su propia versión

atmira

Module Federation

Compartiendo librerías no compatibles entre si con versión estricta

- Configuramos los shared de todos nuestros módulos federados con versiones de librerías no compatibles entre si y con el flag a singleton a true y con check estricto de la versión, con ello obligamos a webpack a que solo exista una versión de la librería en el contexto de ejecución y que solo se ejecute cuando las versiones sean compatibles entre si.

Package.json y webpack config

- Dentro de los package.json de las dos aplicaciones de ejemplo tenemos distintas versiones de “just-a-lib”*.
- Dentro de nuestras configuraciones de webpack configuramos los shared a la versión que tenga el propio package.json, singleton a true y strictVersion a true.

App Shell

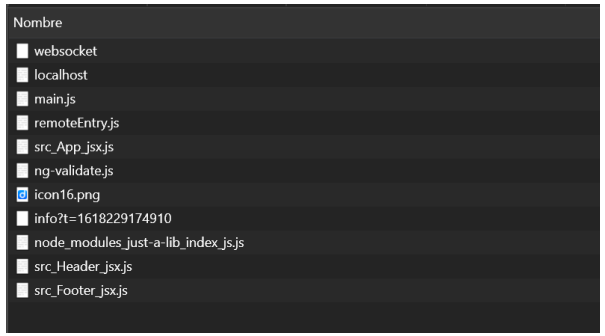
App Nav

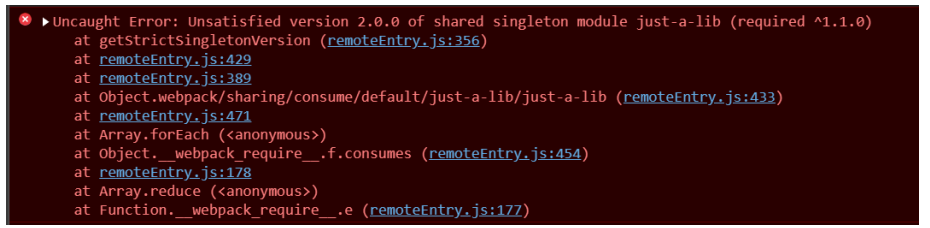
Ejecución

- El módulo de cabecera fallará en su ejecución ya que el chequeo de versión evita que se ejecute al no ser su versión de la librería con la que está disponible en quien lo consume.

Fallback Header
Hi there, I'm React from React.
Component lib version: 2.0.0

Footer!





*just-a-lib: Librería de ejemplo que nos devuelve su propia versión

atmira

Module Federation

Compartiendo librerías con rango de versión

- Configuramos los shared de todos nuestros módulos federados con versiones de librerías no compatibles entre si

Package.json y webpack config

- Dentro de los package.json de las dos aplicaciones de ejemplo tenemos distintas versiones de “just-a-lib”*.
- Dentro de nuestras configuraciones de webpack configuramos los share, pero con un rango de versiones aceptadas.

App Shell

```
"dependencies": {
  "just-a-lib": "^2.0.0",
  "react": "^17.0.2",
  "react-dom": "^17.0.2"
}
```

```
"just-a-lib": {
  requiredVersion: deps["just-a-lib"],
  singleton: true,
  strictVersion: true
},
```

App Nav

```
"dependencies": {
  "just-a-lib": "^1.0.0",
  "react": "^17.0.2",
  "react-dom": "^17.0.2"
}
```

```
"just-a-lib": {
  requiredVersion: ">=1.1.0 <3.0.0",
  singleton: true,
  strictVersion: true
},
```

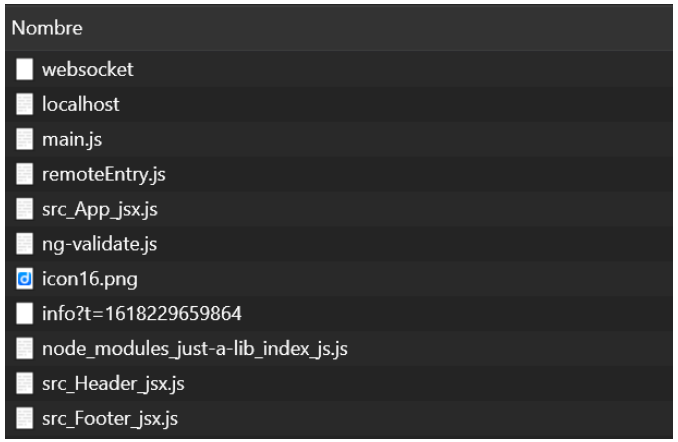
Ejecución

- Vemos que las dos aplicaciones comparten la misma versión, la más alta disponible que sea compatible con ambas aplicaciones y que además solo es descargada una vez.

HEADER lib version 2.0.0!

Hi there, I'm React from React.
Component lib version: 2.0.0

Footer!



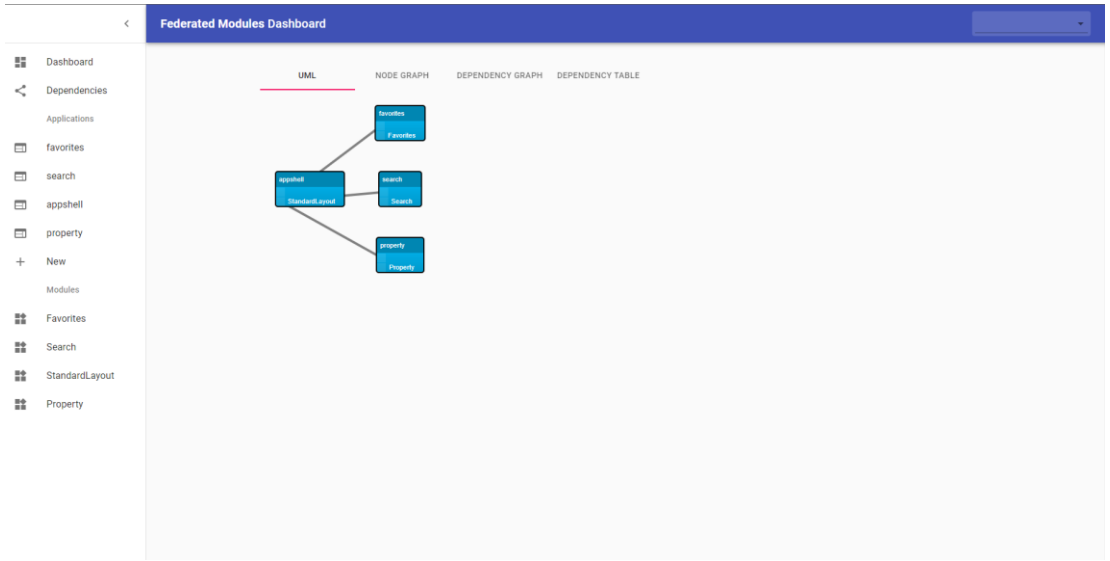
*just-a-lib: Librería de ejemplo que nos devuelve su propia versión

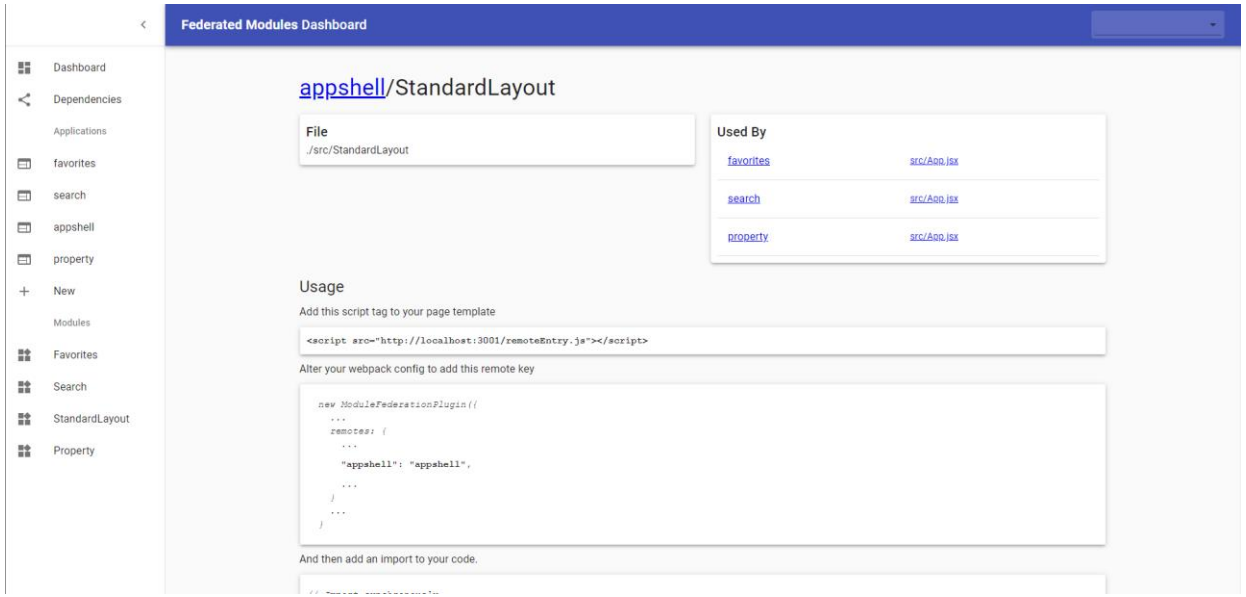
atmira

Module Federation

Gobierno de nuestros Microfrontend

- Con el paso del tiempo y el crecimiento de nuestra App se puede llegar a no saber que expone nuestra App o que consume nuestra App.
- Los creadores de “Module Federation” nos proveen de un Dashboard para gestionar el gobierno de todos nuestros MFE.
- Dentro de este Dashboard podremos ver un gráfico con todos los MFE que componen nuestra aplicación.





atmira

Muchas gracias

Presentación y Código disponible en nuestro Github: http://www.github.com/atmiraio/module-federation-demo