3. Dominio de la capa de presentación

En este capítulo aprenderá a descomponer el frontend en micro aplicaciones para facilitar el cambio y garantizar la autonomía. También aprenderá a diseñar con un enfoque offline-first.

Innovación en la capa de presentación: un zigzag a través del tiempo

Hoy en día, una guerra técnica se libra en torno al uso de frameworks de UI, como React, Vue y Angular. Es bueno tener opciones, y la competencia impulsa la innovación técnica que beneficia a todos, aunque sospecho que nunca pondremos fin a las guerras de frameworks. Sin embargo, debates como el del renderizado en el cliente versus el renderizado en el servidor son más claros. Veamos por qué.

Renderizado en el cliente versus renderizado en el servidor

Todos preferían la experiencia del renderizado en el cliente, pero tenía sus problemas: complicaciones de versionado e instalación, y las limitaciones del hardware y el rendimiento de red de la época. Así que, cuando surgió la World Wide Web, el renderizado volvió al servidor. Sin embargo, la experiencia de usuario se resintió, y esto impulsó la necesidad de la innovación AJAX (Asynchronous JavaScript and XML), que devolvió parte del renderizado al cliente. **A este vaivén entre el renderizado en el cliente y en el servidor lo denominamos péndulo.**

Así que, cuando llegaron las Single-Page Applications (SPAs) y Angular 1, mi reacción inicial fue: «Aquí vamos de nuevo, el péndulo está volviendo al cliente y vamos a experimentar un conjunto similar de problemas». Sin embargo, había pasado el último año luchando con la ejecución de capas de presentación del lado del servidor en la nube, y sabía lo costoso y engorroso que era ofrecer una solución escalable.

Sin embargo, había pasado el último año luchando con la ejecución de capas de presentación del lado del servidor en la nube, y sabía lo costoso y engorroso que era ofrecer una solución escalable.

Supe que todo iba a cambiar en el momento en que me di cuenta de que podía desplegar fácilmente los artefactos de la SPA en un bucket de AWS S3 y servirlos a través de AWS CloudFront, sin necesidad de gestionar ningún servidor. **Con poco o ningún esfuerzo y por centavos, tenía una solución globalmente escalable y tolerante a fallos.**

Renderizado en tiempo de compilación versus en tiempo de ejecución

No es sorprendente, pero no bien tuvimos SPAs renderizando en el cliente, la comunidad de JavaScript se dio cuenta de que también necesitábamos renderizado en el servidor. Necesitábamos un renderizado isomórfico capaz de ejecutarse tanto en el cliente como en el servidor. Parte de una pantalla podía renderizarse en el servidor y el resto podía renderizarse más tarde en el cliente.

La lógica de la capa de presentación permitía que casi cualquier cosa cambiara según las preferencias, roles e historial del usuario actual. **Observé cómo llegaban las métricas de observabilidad y quedó claro que la gran mayoría de las solicitudes producían la misma salida**.

Era un patrón de diseño habitual, pero estaba arraigado en una era de largos plazos de entrega y despliegues infrecuentes. Ahora el contexto ha cambiado. **Podemos desplegar cambios en cualquier momento, muchas veces al día.** No necesitamos hacer una aplicación más compleja y menos eficiente para adaptarse al cambio con lógica basada en datos.

El «Jam» en Jamstack corresponde a JavaScript, API y Markup (Marcado). Se construye sobre los principios de GitOps y las prácticas de despliegue continuo para crear un sistema de gestión de contenidos ligero. También aprovecha el poder del renderizado del lado del servidor en tiempo de compilación para desplegar sitios web y SPAs pre-renderizados que son muy escalables, eficientes y rentables.

Ejecutamos JavaScript en el cliente para animar el renderizado de aquellas secciones de pantalla que necesitan cambiar dinámicamente. Hacemos llamadas asíncronas para recuperar datos que cambian con frecuencia.

Asimismo, cuando cambia contenido relativamente estático, como una publicación de noticias, usamos el lenguaje de marcado Markdown para redactar fácilmente el nuevo contenido. Luego nuestros pipelines de CI/CD compilan y despliegan el cambio.

Podemos ir más lejos y usar nuestra arquitectura event-first para reaccionar a eventos de negocio y automatizar el renderizado y el despliegue de una parte de la aplicación, como una micro aplicación. **La aplicación parece dinámica porque lo es. Pero solo renderizamos el contenido cuando cambia.**

Pero hay más. Pre-renderizar y desplegar SPAs en el borde de la nube tiene un impacto positivo en usuarios cada vez más móviles y frecuentemente desconectados.

Web versus móvil

Construir aplicaciones móviles verdaderamente nativas significa que debemos implementar y mantener aplicaciones separadas para Android e iOS. Un enfoque híbrido solo requiere una única base de código, pero puede no ofrecer a los usuarios la experiencia nativa que desean.

La innovación es nuestro objetivo. La capa de presentación es el campo de batalla donde la innovación se desarrolla. No queremos obstaculizar la innovación imponiendo restricciones innecesarias a la capa de presentación, porque no podemos predecir fácilmente cómo interactuarán los usuarios con los sistemas en el futuro. **Por eso necesitamos una arquitectura para el frontend que facilite el cambio.** Comencemos por ver formas de dividir el frontend.

Rompimiento del monolito en frontend

Necesitamos dividir el monolito frontend, al igual que el backend, para eliminar la fricción que obstaculiza la innovación. El frontend, el backend y la base de datos deben funcionar juntos como una unidad cohesiva.

Esto nos permitirá extender fácilmente el sistema, de arriba a abajo, añadiendo nuevas aplicaciones sin modificar las existentes.

Por subsistema

En el Capítulo 2, *Definición de límites y cesión del control*, alineamos la arquitectura de subsistemas autónomos con los actores, unidades de negocio, capacidades de negocio y ciclo de vida de los datos del sistema.

Cada subsistema debe mantener sus aplicaciones frontend. El frontend de un subsistema tendrá un punto de entrada principal para sus usuarios que brinda acceso a todas sus funcionalidades. **Estas funcionalidades serán aplicaciones frontend desplegables de forma independiente.**

Por actividad del usuario

Dentro de un subsistema, descomponemos el frontend en múltiples aplicaciones según las actividades que realizan los usuarios. **A estas las denominamos micro aplicaciones o micro-apps.**

Por ejemplo, el frontend del subsistema de cliente de nuestro sistema de Entrega de Comida contiene las siguientes micro aplicaciones:

- El punto de entrada principal es una aplicación Jamstack que proporciona material de marketing estático, enlaces de inicio de sesión y registro, y navegación para comenzar a explorar restaurantes y menús.
- El inicio de sesión y el registro son aplicaciones separadas pero relacionadas.
- La búsqueda y exploración de restaurantes y menús es otra aplicación.
- Agregar elementos al carrito y el carrito en sí son micro aplicaciones separadas.

Cada una de estas micro aplicaciones es desplegable de forma independiente y probablemente es propiedad de equipos autónomos distintos. Cada micro aplicación tiene su propio servicio BFF y su propia base de datos.

Por tipo de dispositivo

Manejar todas las permutaciones de tipos de dispositivo puede afectar nuestra capacidad de cambiar un sistema de forma rápida y sencilla. Dependiendo del sistema, puede que no tenga sentido soportar todas las actividades en todos los tipos de dispositivo.

Algunas actividades pueden estar disponibles únicamente según el tipo de dispositivo.

Hasta cierto punto, podemos hacer que una aplicación sea responsiva según el tipo de dispositivo y ajustar automáticamente el diseño para adaptarse al tamaño de pantalla. Sin embargo, hay límites, y implementar una pantalla responsiva puede ser contraproducente en algunos escenarios. **En estos casos, puede tener sentido crear aplicaciones separadas para distintos tipos de dispositivo.**

Por versión

Cuando realizamos cambios pequeños e incrementales en una aplicación, no necesitamos soportar múltiples versiones. Sin embargo, los feature flags pueden volverse complicados cuando hacemos cambios significativos. En estos casos, es mejor crear una nueva versión de la aplicación y ejecutar ambas versiones en paralelo hasta que la nueva esté completa.

Usamos un rol especial como el único feature flag para limitar el acceso a la nueva versión de la aplicación.

Anatomía de los micro frontends

Para alcanzar nuestro objetivo de crear una arquitectura de software que facilite el cambio, necesitamos dar a los equipos control total del stack completo. Estos equipos autónomos, autosuficientes y full-stack pueden avanzar a su propio ritmo, minimizar el tiempo de entrega y responder a los comentarios de los usuarios a voluntad, precisamente porque no dependen de otros equipos.

El objetivo de un micro frontend es dividir la experiencia de usuario en un conjunto de micro aplicaciones independientes, al tiempo que ofrece a los usuarios una experiencia fluida.
A medida que los usuarios se mueven entre distintas actividades, no debería parecerles que están saltando entre aplicaciones. Debería sentirse como una única aplicación.

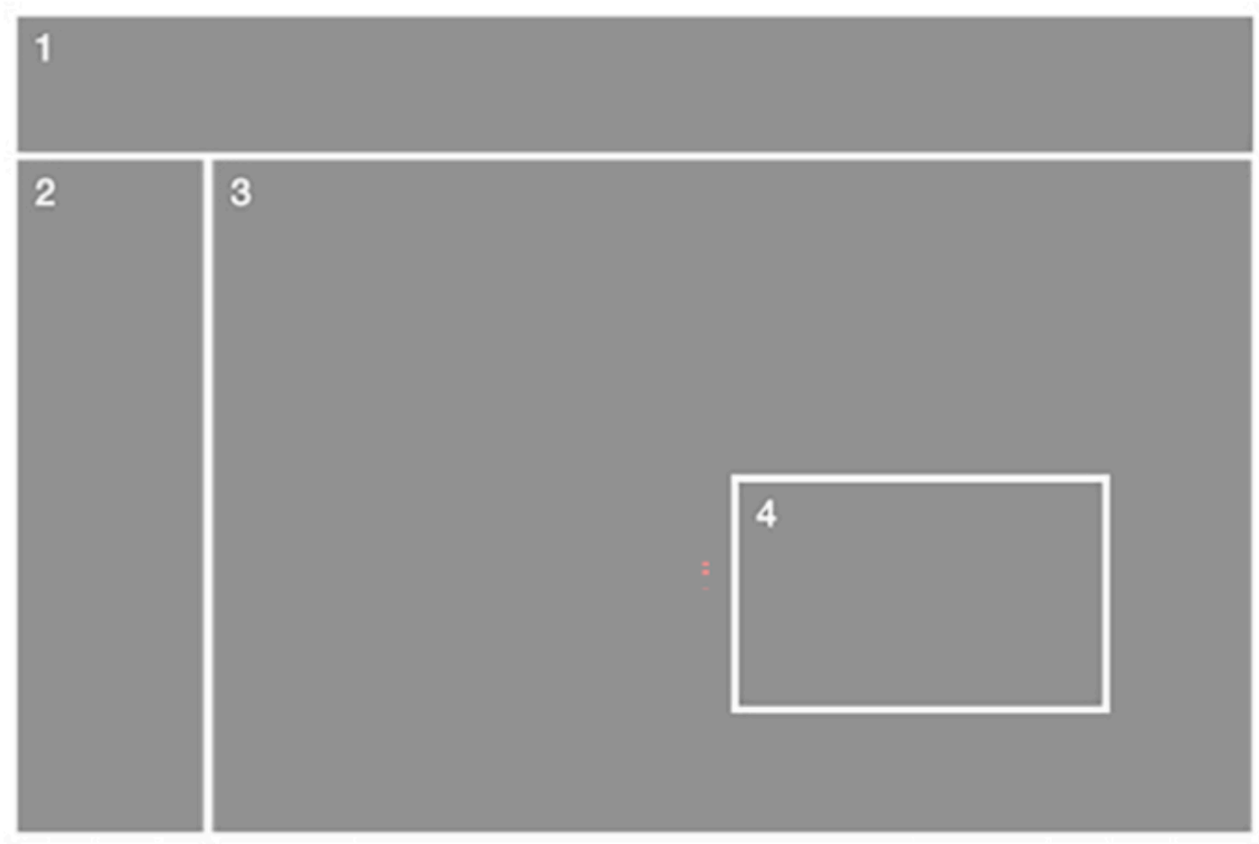

Figure 3.2: Micro frontend

La ilustración anterior muestra el banner (1) y la navegación izquierda (2) como micro aplicaciones separadas. Otra micro aplicación (3) proporciona el contenido de la página, que contiene a su vez otra micro aplicación (4). A medida que el usuario navega por el subsistema, distintas micro aplicaciones proporcionan el contenido de las diferentes pantallas.

El enfoque de micro frontends ofrece los siguientes tres beneficios principales:

- Aísla el cambio: Cada micro aplicación es desplegable de forma independiente. Siguiendo el principio Open-Closed, podemos desplegar una nueva micro-app sin modificar las aplicaciones existentes. **Para cuando necesitemos cambiar y redesplegar una micro-app existente, hemos limitado el alcance del impacto a esa única micro-app.**
- Maximiza el potencial de la carga diferida (lazy loading): La división de código (code splitting) proporciona una solución viable a este problema, pero las micro-apps llevan la carga diferida al primer plano, en lugar de ser una reflexión tardía. Una micro-app individual también puede aprovechar la división de código.
- Admite el uso simultáneo de múltiples frameworks y, más importante aún, múltiples versiones de frameworks: Cada micro-app puede usar un framework diferente, como React o Angular. Sin embargo, **no debemos abusar de esta capacidad**. Esta capacidad también permite actualizar un único framework sin requerir que todas las micro aplicaciones actualicen al mismo tiempo.

Existen muchas alternativas para implementar micro frontends. Podemos ensamblarlos en tiempo de compilación o en tiempo de ejecución, en el cliente o en el servidor; podemos enlazarlos entre sí o conectarlos mediante programación.

Hay una aplicación principal que inicia el sistema, y cada micro-app implementa un ciclo de vida bien definido para que el sistema pueda activarlas y montarlas en ubicaciones bien definidas en pantalla.

La aplicación principal (main app)

La capa de presentación de cada subsistema autónomo tiene un punto de entrada principal para acceder a todas sus funcionalidades. Sin embargo, la aplicación principal no proporciona ninguna funcionalidad específica, más allá de inicializar el entorno de micro frontends. **En esencia, la aplicación principal solo es propietaria del archivo index.html.**

El pipeline de CI/CD realiza las siguientes tareas:

- Crea el almacén de objetos, como un bucket de S3, que contendrá todos los recursos de las distintas micro aplicaciones.
- Crea la Red de Distribución de Contenidos (CDN), como una distribución de AWS CloudFront, que servirá todos los recursos de las distintas micro aplicaciones desde el borde de la nube.
- Sube el archivo index.html y un archivo de manifiesto al bucket. Cubriremos los archivos de manifiesto en la sección *Desplegador de manifiestos*.
- Crea el nombre de dominio que usamos para acceder al archivo index.html, como con un registro de AWS Route 53.

El archivo index

El archivo index.html es liviano y compacto. Inicializa el sistema de carga dinámica de módulos y pone en marcha el proceso de registro de micro aplicaciones.

```
<html>
<head>
<title>My Main</title>
<meta name="importmap-type" content="systemjs-importmap"/>
<script type="systemjs-importmap" src="/importmap.json"/>
<script src="https://cdn.../system.min.js" />
<script src="https://cdn.../amd.min.js" />
<script>System.import('@my/main');</script>
<link ... href="//cdn.../semantic.min.css" />
</head>
<body />
</html>
```

La principal tarea del archivo index.html es cargar la estructura de datos /importmap.json e inicializar el cargador de módulos SystemJS (https://github.com/systemjs/systemjs).

Luego importa la micro aplicación @my/main para iniciar el proceso de registro. Finalmente, carga la hoja de estilos global.

El archivo importmap

La carga diferida es uno de los principales beneficios del enfoque de micro frontends. Cargamos cada micro aplicación justo a tiempo. Esto proporciona una experiencia de usuario más fluida, ya que no gastamos tiempo cargando aplicaciones que no utilizamos.

Sin embargo, por defecto, JavaScript no admite la carga dinámica de módulos. A menos que especifiquemos lo contrario, un bundler, como webpack, incorporará estas dependencias al bundle que crea. **Esto aumenta el tamaño del bundle y duplica estas dependencias en múltiples micro aplicaciones.**

Alternativamente, podemos indicarle al bundler que externalice dependencias específicas. Ahora necesitamos un mecanismo para cargar dinámicamente estos módulos cuando nuestro código los importa. Aquí es donde entran en juego el archivo importmap.json y SystemJS. A continuación se muestra un ejemplo de un archivo importmap:

```
{ "imports": {
    "@my/main": "/my-main/{commit-hash}/my-main.js",
    "@my/main/": "/my-main/{commit-hash}/",
    "@my/app1": "/my-app1/{commit-hash}/my-app1.js",
    "@my/app1/": "/my-app1/{commit-hash}/",
    "@my/shared": "/my-shared/{commit-hash}/my-shared.js",
    "@my/shared/": "/my-shared/{commit-hash}/",
    "react": "https://cdn.jsdelivr.net/...",
    "react-dom": "https://cdn.jsdelivr.net/...",
    "semantic-ui": "https://cdn.jsdelivr.net/...",
    "single-spa": "https://cdn.jsdelivr.net/...",
    }
}
```

Este archivo hace exactamente lo que su nombre indica. Mapea los nombres de importación de los módulos a la ruta donde se encuentran. SystemJS usa este archivo para importar módulos externalizados. Cuando importamos el módulo `@my/main`, SystemJS lo cargará desde nuestro bucket de S3.

Nótese que hay dos entradas para cada una de nuestras micro aplicaciones. Usamos la primera entrada para inicializar una micro aplicación sin cargar el módulo completo. Luego usamos la segunda entrada cuando es momento de cargar una micro aplicación.

Registro de Micro-app

Necesitamos registrar cada micro aplicación con la aplicación principal. El proceso de registro comienza cuando importamos la aplicación principal al final del archivo `index.html`.

```
import { registerApplication, start } from "single-spa";
...
fetch('/apps.json').then(res => res.json())
  .then((apps) => {
    const routes = constructRoutes(apps);
    const applications = constructApplications({
      routes,
      loadApp: ({ name }) => System.import(name),
    });
    const engine = constructLayoutEngine({ routes,
      applications });
    applications.forEach(registerApplication);
    engine.activate();
    start();
  });
```

La aplicación principal recupera y procesa el archivo de manifiesto apps.json. Construye las rutas y registra todas las aplicaciones, y luego llama a start para activar el sistema. También define una función loadApp para cada aplicación.

Esta función cargará de forma diferida el archivo de entrada de una micro aplicación. Esta es la primera entrada de la aplicación en el archivo importmap. Cubriremos el archivo de entrada en breve, en la sección *Micro-app*. A continuación se muestra un ejemplo de un archivo apps.json:

```
{ "routes": [
    {
        "type": "application",
        "name": "@my/nav"
    },
    {
        "type": "route",
        "path": "/things",
        "routes": [{
            "type": "application",
            "name": "@my/app1"
        }]
    },
    ...
]}
```

El motor de diseño single-spa define la estructura del archivo apps.json. El archivo contiene todas las aplicaciones y sus políticas de activación. Por ejemplo, la micro aplicación `@my/app1` se activará cuando un usuario acceda a la ruta /things, y la micro aplicación `@my/nav` siempre estará activa.

Micro-app

Cada aplicación tiene su propio repositorio de código fuente y su propio pipeline de CI/CD. El pipeline de CI/CD realiza las siguientes tareas:

- Prueba todo el código fuente de la micro aplicación.
- Compila el bundle de la micro aplicación con un bundler, como webpack.
- Sube el bundle y un archivo de manifiesto al bucket creado por la aplicación principal. Cubriremos los archivos de manifiesto en la sección *Desplegador de manifiestos*.

Los internos de una micro aplicación son los mismos que los de su framework elegido, como React. **La única diferencia es el punto de entrada. Una micro aplicación tiene un archivo de entrada y un componente raíz.**

El archivo de entrada (entry file)

Cada micro aplicación debe tener un archivo de entrada. Externalizamos este archivo del resto del bundle de la micro aplicación para maximizar la carga diferida.

La aplicación principal cargará de forma diferida este archivo utilizando la función loadApp que definimos en la sección *Registro de micro aplicaciones*.

```
import singleSpaReact from 'single-spa-react';
...
const lifecycles = singleSpaReact({
  ...
  loadRootComponent: () => import('./root.component.js'),
  domElementGetter: () => document
    .getElementById('container'),
  ...
});
export const bootstrap = lifecycles.bootstrap;
export const mount = lifecycles.mount;
export const unmount = lifecycles.unmount;
```

El archivo de entrada define las siguientes funciones del ciclo de vida de la micro aplicación:

- La función `loadRootComponent` carga de forma diferida el bundle de la micro aplicación.
- La función `bootstrap` se llama una vez para resolver el componente raíz.
- La función `domElementGetter` devuelve el elemento donde queremos montar el componente raíz.
- La función `mount` renderiza la aplicación en la ubicación definida por `domElementGetter`.
- La función `unmount` elimina los elementos del DOM que renderizamos durante el montaje.

Componente raíz (root component)

La funcionalidad real de una micro aplicación comienza en el componente raíz. Esto es análogo al componente App en una aplicación React típica, como el componente App generado por la utilidad Create React App.

```
import React from "react";
const Root = (props) => (<p>{props.name} is mounted!</p>);
export default Root;
```

El componente raíz React típico inicializará varios proveedores de contexto y configurará las reglas de enrutamiento.

Activación de la Micro-app

El enfoque de micro frontend divide la aplicación en un conjunto de micro aplicaciones con carga diferida. Luego cargamos y activamos una micro-app justo a tiempo. Típicamente activamos las micro aplicaciones basándonos en rutas, y también podemos activarlas manualmente. Pero primero, veamos el flujo completo del ciclo de vida.

Ciclo de vida de la micro-app

Como vimos en la sección *El archivo de entrada*, cada micro aplicación proporciona un conjunto de métodos de ciclo de vida que el framework de micro frontend usa para inicializar, montar y desmontar la aplicación.

1. **Al inicio**, la aplicación principal registra cada aplicación en el manifiesto `apps.json` e inicia el framework de micro frontend.
2. Luego el framework verifica la política de activación de cada aplicación.
3. La primera vez que se activa una aplicación, el framework **carga el archivo de entrada** para acceder a las funciones del ciclo de vida de la aplicación. **Luego llama a la función del ciclo de vida bootstrap para cargar el componente raíz.**
4. Cada vez que se activa una aplicación, el framework llama a la función del ciclo de vida `mount` para adjuntar el componente raíz al DOM según lo especificado por la función `domElementGetter`.
5. Cuando una aplicación se desactiva, el framework llama a la función del ciclo de vida `unmount` para eliminarla del DOM.
6. A medida que el usuario navega por el sistema, **el framework verifica todas las políticas de activación y monta y desmonta las aplicaciones según corresponda**.

Algunas micro aplicaciones están siempre activas. Por ejemplo, una micro aplicación de navegación global puede estar siempre disponible para proporcionar una función de menú principal. La mayoría de las micro aplicaciones se activarán cuando el usuario seleccione estos menús. Además, en menor medida, una micro aplicación activará manualmente a otra.

Activación basada en rutas

En una SPA, el enrutamiento es la práctica de renderizar diferentes componentes a medida que cambia la URL. Los frameworks de SPA se enganchan en las interfaces de historial y ubicación del navegador y renderizan los componentes apropiados a medida que el usuario navega por la aplicación.

Estas rutas representan diferentes actividades de usuario. En la sección *Fragmentación del monolito frontend*, **identificamos que es una buena práctica crear una micro aplicación separada por actividad de usuario.**

Las rutas son la forma más común de activar micro aplicaciones. El framework de micro frontend usa rutas de nivel superior para identificar cuándo activar una micro aplicación específica. Luego la micro aplicación usa rutas de subnivel para dirigir el renderizado de sus componentes.

```
const Root = (props) => (
  <Router>
    <Things path="things" {...props} />
    <Thing path="things/:id" {...props} />
  </Router>
);
```

El framework de micro frontend activará esta micro aplicación basándose en la ruta `/things`, tal como está definida en el archivo apps.json. Dentro de esta micro aplicación, la ruta `/things` renderizará el componente `Things` para explorar elementos, y la ruta `/things/:id` renderizará el componente `Thing` para ver un elemento específico.

Activación manual

También es posible tomar el control total y activar manualmente una micro aplicación. **Normalmente haremos esto cuando queremos incrustar una micro aplicación dentro de otra micro aplicación.**

```
<div>
  <Parcel
    config={() => System.import('@my/add-to-cart')}
    {...props, additionalProp } />
</div>
```

La biblioteca `single-spa-react` proporciona el componente `Parcel`. Colocamos este componente en la ubicación donde queremos montar la micro aplicación. Esto significa que las micro aplicaciones incrustadas no necesitan declarar la función `domElementGetter`.

Luego configuramos una función `load` y especificamos el nombre de la micro aplicación que queremos cargar. También podemos pasar propiedades a la micro aplicación incrustada para inicializarla con la información adecuada.

Puntos de montaje

Una micro aplicación se monta en un elemento `dom` específico cuando se activa. A este elemento `dom` lo denominamos punto de montaje. Un punto de montaje representa un contrato entre el propietario del punto de montaje y las micro aplicaciones que lo utilizan.

Por ejemplo, **el propietario controla la ubicación, el ancho y el alto del elemento dom**, y debemos diseñar una micro aplicación para que funcione dentro de esos parámetros.

La navegación es un ejemplo típico de punto de montaje. Por ejemplo, como se muestra en la siguiente captura de pantalla, el menú de navegación izquierdo definirá enlaces para activar micro aplicaciones basadas en rutas y definirá un elemento dom contenedor donde las aplicaciones se montarán.

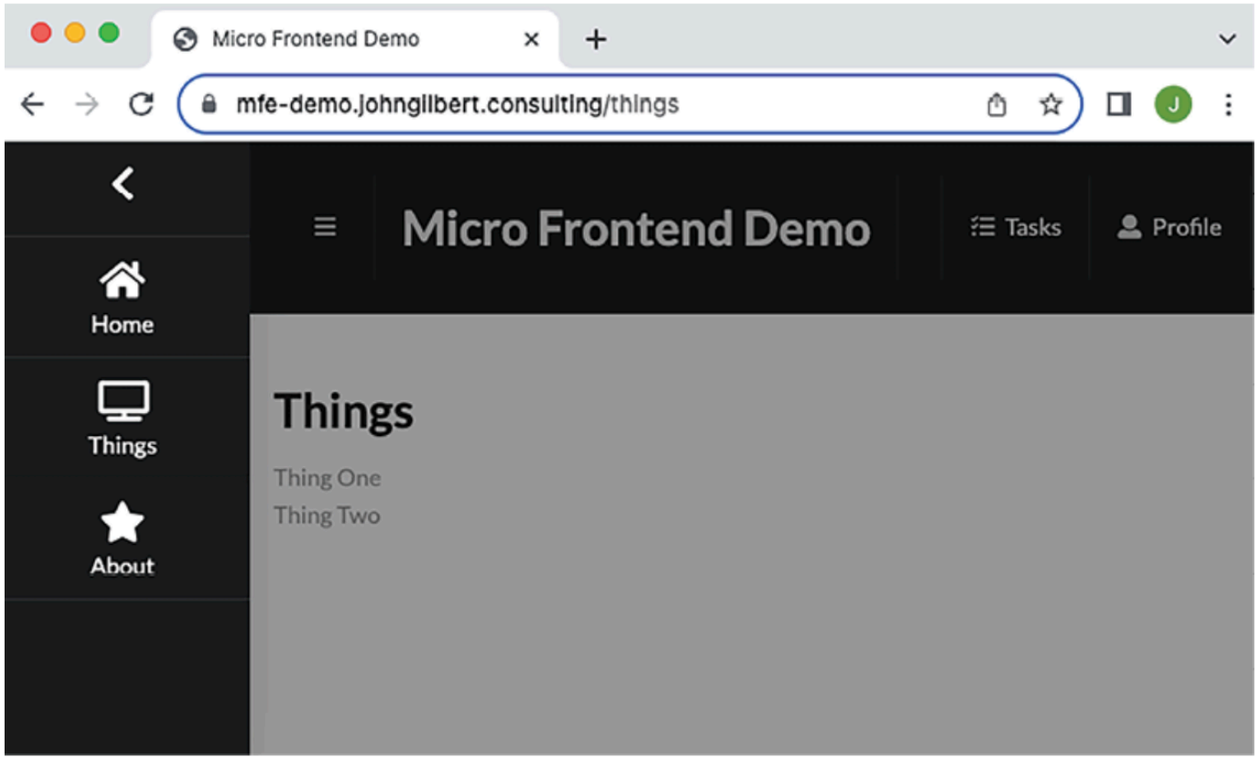

Figure 3.3: Mount point demo

Para una aplicación grande, preferimos generar la navegación dinámicamente basándonos en metadatos. Esto nos permite añadir y eliminar micro aplicaciones sin realizar cambios en el código fuente de la navegación.

```
export const LeftNavMenuItems = () => {
  const { items } = useMountPoint('left-nav');
  return items.map((item) => <Menu.Item key={item.key}
        content={item.content} icon={item.icon}
        as={Link} to={item.to} />);
};
export const MainContainer = ({ children }) =>
  <div id='container'>{children}</div>;
```

El Hook `useMountPoint` recupera un archivo `/mount-points.json` y devuelve los metadatos solicitados. Este componente itera sobre los metadatos y renderiza menús con enlaces que enrutarán a las micro aplicaciones. También definimos el elemento contenedor donde pretendemos montar las micro aplicaciones.

```
"mount-points": {
    "left-nav": [{
        "key": "thing",
        "content": "Things",
        "icon": "tv",
        "to": "/things"
    }, ...]
  }
```

El código del punto de montaje dicta el contenido de los metadatos para un punto de montaje específico. En este ejemplo, el punto de montaje espera una clave única, el contenido e icono a mostrar, y la ruta a la que navegará el enlace.

Cuando un usuario hace clic en un enlace de menú, el framework activará una micro aplicación siguiendo el proceso que describimos en la sección *Activación de micro aplicaciones*.

Podemos ir más lejos con los puntos de montaje y agregar reglas condicionales para controlar aún más la navegación. Por ejemplo:

- Podemos filtrar enlaces según el rol del usuario o el tipo de dispositivo.
- Podemos deshabilitar menús según el estado de las entidades del dominio.

Añadiríamos estos criterios a los metadatos, como una lista de roles válidos, agentes de usuario o indicadores de estado.

Desplegador de manifiestos

No queremos arriesgarnos a actualizar manualmente estos archivos de manifiesto maestros a medida que añadimos y cambiamos micro aplicaciones. En cambio, cada aplicación define su propio archivo de manifiesto que contiene los fragmentos que usamos para generar los archivos de manifiesto principales. A continuación se muestra un ejemplo de un archivo de manifiesto `mfe.json`:

```
{
  "orgName": "my",
  "projectName": "app1",
  "importmap": { ... },
  "apps": { ... },
  "mount-points": { ... }
}
```

Cada archivo debe tener los campos `orgName` y `projectName` que **identifican de forma única** la micro aplicación, más los elementos `importmap` y `apps`. El elemento `mount-points` es opcional.

1. El pipeline de CI/CD de una micro aplicación sube su archivo de manifiesto `mfe.json` al bucket de S3.
2. Esto a su vez activa el servicio `manifest-deployer`.
3. Este servicio construye los archivos de manifiesto maestros a partir de todos los archivos de manifiesto individuales y los almacena en el bucket de S3.
4. El servicio desplegador aprovecha el mismo framework de procesamiento de flujos que cubrimos en el Capítulo 4, *Confianza en los hechos y consistencia eventual*.

Comunicación entre aplicaciones

Ahora que hemos dividido el sistema en micro aplicaciones, necesitamos mecanismos que permitan que las aplicaciones se comuniquen entre sí. Por ejemplo, una aplicación puede lanzar otra, o múltiples aplicaciones en una pantalla pueden necesitar actualizarse juntas a medida que cambia el contexto.

En la sección *Fragmentación del monolito frontend*, identificamos varias buenas prácticas que podemos usar para descomponer un sistema en micro aplicaciones. **Minimizamos la necesidad de comunicación entre aplicaciones cuando seguimos estas buenas prácticas**, porque cada actividad de usuario es autónoma.

Si encontramos que dos micro aplicaciones necesitan comunicarse de ida y vuelta, esto puede ser una señal de que deberíamos fusionarlas. Pero existen muchos escenarios no cíclicos donde una aplicación interactúa con otra.

Una aplicación puede pasar información a otra aplicación usando **parámetros de ruta**. Por ejemplo, una aplicación puede proporcionar únicamente una función de búsqueda. Desde la lista de resultados de búsqueda, podemos querer lanzar diversas actividades de usuario.

Implementaríamos cada una de estas actividades de usuario como una micro aplicación separada. Luego pasamos información identificadora como parámetro, como el ID del producto seleccionado, para que la micro aplicación pueda recuperar la información apropiada. **Lo mismo aplica para la activación manual de una micro aplicación y el paso de propiedades.**

También podemos crear micro aplicaciones que actúen como biblioteca compartida. Exportan funciones y las importamos dinámicamente usando importmap. **Esto garantiza que solo exista una única instancia de la biblioteca.** Esto nos permite almacenar en caché datos comunes dentro de estos módulos y acceder a ellos a través de las funciones exportadas.

También podemos usar un bus de eventos para intercambiar información con un acoplamiento mínimo. Por ejemplo, una aplicación puede registrar un escuchador de eventos usando `window.addEventListener('my-evt', ...)`. Luego, otras aplicaciones pueden emitir información a medida que su estado cambia usando `window.dispatchEvent(...)`.

También es posible compartir información de estado entre micro aplicaciones con bibliotecas como Redux, pero esto limita la independencia de las micro aplicaciones. **Cada micro aplicación debe mantener su propio almacén de estado.** Debemos limitar cualquier almacenamiento compartido o global a un ámbito muy específico que viva en una micro aplicación compartida con una interfaz claramente definida y exportada.

Diseño con enfoque offline-first

Para aplicaciones con contenido dinámico, el proceso de hacer el contenido disponible sin conexión es más complejo. Además, **es más difícil predecir cuándo la conectividad estará no disponible**. Es útil pensar en este problema en términos del teorema CAP, que nos muestra que en caso de una partición del sistema, debemos elegir entre consistencia y disponibilidad. No podemos tener ambas.

En este caso, los usuarios prefieren la disponibilidad sobre la consistencia. Quieren que la aplicación siga funcionando y que resuelva las cosas cuando se restablezca la conexión. **Esto significa que los usuarios móviles están impulsando la necesidad de sistemas eventualmente consistentes.**

Diseñaremos la capa de presentación para trabajar con un enfoque offline-first. Almacenará datos en caché localmente y será completamente funcional cuando esté desconectada.

Transparencia

La transparencia es el primer imperativo para las aplicaciones offline-first. Necesitamos dar a los usuarios visibilidad del estado de la aplicación para que puedan tomar decisiones informadas.

Indicadores de estado

En primer lugar, una aplicación necesita proporcionar al usuario algún tipo de indicador de estado de conectividad. Esto se suma al indicador proporcionado por el dispositivo, porque una aplicación puede perder conectividad solo con su backend, como durante una interrupción del servicio.

A continuación, puede ser útil proporcionar información sobre la caché, como la versión del código de la aplicación y la hora de la sincronización de datos más reciente. En algunos casos, puede ser útil proporcionar una **fecha de referencia (es decir, una marca de tiempo) para cada registro**. El indicador también debe **mostrar cuándo la sincronización está en progreso**.

Bandeja de salida (Outbox)

Las cosas se ponen más interesantes cuando el usuario edita contenido dinámico sin conexión. **Necesitamos encolar los cambios y enviarlos al backend cuando tengamos conexión.** Debemos dejar esto claro a los usuarios. Necesitan entender que su trabajo está incompleto, y puede que necesiten intervenir en el peor de los casos.

Bandeja de entrada (Inbox)

Una vez que tenemos conexión y la sincronización está completa, puede haber ciertas actualizaciones que sea importante destacar para el usuario. Por ejemplo, puede haber cambios relacionados con los datos que el usuario tenía en la bandeja de salida. Esto puede ser meramente informativo, pero en el peor de los casos, el usuario puede necesitar reconciliar cambios si hubo conflictos con cambios realizados por otros usuarios.

Es importante dar control a los usuarios. Pueden iniciar una sincronización de caché para aumentar su confianza en los datos. O pueden iniciar una sincronización cuando saben que van a quedarse sin conexión.

Caché local

Una aplicación offline-first mantiene una caché local en el dispositivo del cliente para que la aplicación siga siendo usable cuando hay conectividad limitada o nula.

- Necesitamos almacenar en caché el código de las micro-apps,
- necesitamos almacenar en caché los datos más recientes,
- y necesitamos estar preparados para almacenar en caché nuestras escrituras.

Almacenamiento en caché del código

Tradicionalmente, instalamos una aplicación descargándola desde una tienda de aplicaciones y permitimos que se actualice automáticamente cuando hay cambios disponibles. La web, por otro lado, tradicionalmente descargaba los recursos justo a tiempo, lo que significaba que una aplicación web solo estaba disponible cuando estaba en línea. **Las cabeceras Cache-Control permiten a un navegador almacenar contenido en caché para mejorar el rendimiento y minimizar la transferencia de datos, pero un navegador aún producirá errores con el contenido en caché cuando no hay conectividad.**

Una PWA es una alternativa más reciente que se esfuerza por proporcionar los beneficios de ambos enfoques. Una PWA es una aplicación web normal que se ejecuta en el navegador, pero descarga progresivamente y almacena en caché el código de la aplicación y eventualmente ejecuta toda la aplicación desde el código en caché, como si la hubiéramos instalado completamente desde el inicio.

Convertimos nuestro micro frontend en una PWA añadiendo un archivo `manifest` y un `ServiceWorker` a la micro-app principal. Todas las demás micro-apps se beneficiarán automáticamente.

```
{
  "short_name": "My Apps",
  "name": "My Apps",
  "background_color": "#000000",
  "display": "standalone",
  "start_url": ".",
  "icons": [{
    "src": "/public/logo.png",
    "type": "image/png",
    "sizes": "144x144"
  }]
}
```

Un enlace al archivo manifest en el archivo index.html le indica al navegador que habilite ciertas funcionalidades, como la posibilidad de instalar la aplicación como un ícono en el escritorio o la pantalla de inicio.

Nótese que los navegadores pueden ser quisquillosos con estas configuraciones. Por ejemplo, el tamaño del ícono especificado debe coincidir con el tamaño real del archivo especificado, de lo contrario estas funcionalidades no se habilitarán.

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker
```

```
    .register('/service-worker.js');
}
```

Este código vive en el archivo de entrada de la aplicación principal. Primero, verificamos si el navegador admite service workers. En otras palabras, añadiremos progresivamente estas funcionalidades a nuestro micro frontend para ayudar a garantizar que pueda ejecutarse en la más amplia gama de navegadores posible. Luego registramos la ruta al archivo `service-worker.js`.

La instancia de `ServiceWorker` se ejecuta en un hilo separado y actúa como proxy para las solicitudes de red. Intercepta las solicitudes, detecta si hay conectividad y da a la aplicación la oportunidad de utilizar la API de Caché del navegador (https://developer.mozilla.org/en-US/docs/Web/API/Cache), que es distinta de la caché HTTP del navegador.

Veamos las estrategias de caché que usaremos:

- **La estrategia Cache-First** siempre verifica primero la caché. Si el recurso no está en la caché, recurrimos a la red y almacenamos la respuesta en la caché para la próxima vez. Usaremos esta estrategia para recursos que cambian con poca frecuencia.
- **La estrategia Network-First** siempre verifica primero la red. Almacena la respuesta más reciente en la caché para usarla como respaldo cuando no tenemos conectividad. Usaremos esta estrategia para recursos que cambian con más frecuencia. Nótese que cuando el `ServiceWorker` realiza estas llamadas de red, observará la caché HTTP del navegador, que está controlada por las cabeceras cache-control.
- **La estrategia Stale-While-Revalidate** es un término medio. Usa los recursos en caché, pero los actualiza desde la red en segundo plano para la próxima vez. Usaremos esta estrategia cuando queramos priorizar la velocidad sobre la actualidad de los datos.

Ahora necesitamos implementar el service worker. Este código puede ser tedioso y es en gran parte código repetitivo, así que usaremos un conjunto de recetas y utilidades de la biblioteca Workbox (https://developer.chrome.com/docs/workbox) para simplificar el código.

```
import {...} from 'workbox-recipes';
import {registerRoute} from 'workbox-routing';
import {NetworkFirst} from 'workbox-strategies';
// cache first recipes
imageCache();
googleFontsCache();
// stale while revalidate recipes
staticResourceCache();
// network first recipes
pageCache();
registerRoute(
  /.*\.json/,
  new NetworkFirst({
    cacheName: 'json',
  }),
);
// catch recipe
offlineFallback({ pageFallback: 'index.html' });
```

Este ejemplo usa múltiples recetas y estrategias de caché. Las recetas de imágenes y fuentes usan la estrategia cache-first. La receta de recursos estáticos usa la estrategia stale-while-revalidate para scripts y estilos. La receta de páginas usa la estrategia network-first para las navegaciones de página.

Configuramos nuestra propia caché network-first para nuestros archivos `importmap.json`, `apps.json` y `mount-points.json`. Y finalmente, cuando todo lo demás falla, enviamos a los usuarios a la página de inicio.

Una vez que un usuario accede a la página de inicio, habremos almacenado en caché el esqueleto de la aplicación, como el encabezado, pie de página, barras laterales, navegación y contenido de la página de inicio, por lo que siempre tendremos algo a lo que recurrir. A medida que el usuario se mueve por el sistema, almacenaremos en caché más y más micro-apps, para que el usuario pueda acceder a más funcionalidades sin conexión.

Existen muchas opciones avanzadas que podemos aprovechar para almacenar en caché las apps de forma más agresiva. Por ejemplo, **podemos hacer que webpack configure una precaché** y podemos utilizar la información de nuestros archivos `importmap.json`, `apps.json` y `mount-points.json` para precalentar la caché de una sola vez según los permisos del usuario y el agente de usuario del dispositivo actual.

Almacenamiento en caché de lecturas de datos

Para acceder a los datos, necesitamos realizar llamadas remotas a los servicios BFF. Pero no tendremos acceso a estos servicios cuando estemos sin conexión, **por lo que necesitamos mantener una caché local que almacene datos en el dispositivo del usuario.**

Usaremos una combinación de las estrategias de caché `Network-First` y `Stale-While-Revalidate` para nuestros datos, ya que cambian dinámicamente.

```
import {NetworkFirst} from 'workbox-strategies';
import {ExpirationPlugin} from 'workbox-expiration';
registerRoute(
```

```
  ({ request }) => request.url.includes('/api-')),
  new NetworkFirst({
    cacheName: 'api-gets',
    plugins: [new ExpirationPlugin({
        maxAgeSeconds: 45 * 24 * 60 * 60, // days
    })],
  }),
  'GET'
);
```

En este ejemplo, configuramos una caché `NetworkFirst` para todas las solicitudes GET a nuestros servicios BFF. En el Capítulo 9, *Ejecución en múltiples regiones*, veremos cómo configuramos todos los endpoints de nuestros servicios BFF con el prefijo /api-.

También configuramos esta caché para que expire después de 45 días, para que no crezca sin límite. Este ejemplo básico almacenará en caché cualquier dato con el que el usuario esté trabajando actualmente.

Para estar completamente preparados para el modo sin conexión, también necesitamos obtener proactivamente ciertos tipos de datos. Por ejemplo, necesitamos almacenar en caché datos de referencia que la UI necesita para poblar listas desplegables. **Podemos precargar estos datos al inicio de la aplicación y usar una estrategia `Stale-While-Revalidate`,** para que actualicemos estos datos en segundo plano.

También es posible que queramos diferentes períodos de expiración para diferentes servicios. Además, podemos combinar esto con el uso de cabeceras cache-control en las respuestas individuales del servicio para ajustar el comportamiento y aprovechar también la caché HTTP del navegador. Además, podemos usar bibliotecas como `react-query` (https://www.npmjs.com/package/@tanstack/react-query) para mantener una caché en memoria a nivel de aplicación para optimizar completamente la capacidad de respuesta de la UX.

Almacenamiento en caché de escrituras de datos

Si la aplicación permite a los usuarios crear o modificar datos, entonces necesitamos realizar llamadas remotas a los servicios BFF para persistir los cambios. Pero no tendremos acceso a estos servicios cuando estemos sin conexión, **por lo que necesitamos almacenar en caché los cambios del usuario localmente y enviar los cambios a los servicios remotos cuando haya conexión disponible.**

También necesitamos proporcionar al usuario consistencia de sesión. En otras palabras, necesitamos garantizar la consistencia de lectura de escrituras propias (read-your-writes). Cuando estamos sin conexión, no podemos escribir cambios en el servicio BFF, por lo que tampoco podemos leer los cambios desde el servicio BFF.

Comencemos almacenando una copia borrador de cualquier cambio que esté realizando el usuario.

```
import {useForm} from "react-hook-form";
import useFormPersist from 'react-hook-form-persist'
const { ..., watch, setValue} = useForm();
useFormPersist("myForm", {
  watch, setValue, storage: localStorage});
```

En este ejemplo, el hook `useFormPersist` usa la función watch para suscribirse a los cambios de entrada y escribir los valores en el almacenamiento local. Luego puede usar la función `setValue` para restaurar los valores desde el almacenamiento local. **Cuando el usuario envía el formulario, los valores se eliminan del almacenamiento.** Esto nos permite recuperar el trabajo en borrador del usuario si hay una interrupción.

A continuación, necesitamos actualizar la caché local cuando el usuario envía un cambio.

```
const qc = useQueryClient();
useMutation({
  mutationFn: ...,
  onSuccess: (data) => {
    qc.setQueryData(['order', data.id], data)
    qc.setQueriesData(['orders'], (prev) =>
      prev.map((d) => (d.id === data.id ? data : d)))
}});
```

Registramos una función `onSuccess` que actualizará la caché con los datos mutados después de realizar una escritura.

- Primero, actualizamos la caché de detalles del pedido con la nueva versión de los datos.
- Luego encontramos la versión anterior en la caché de la lista de pedidos y la reemplazamos con la nueva versión.

Para una mutación de inserción, añadiríamos los datos a la caché de pedidos, y para una mutación de eliminación, los eliminaríamos. **Esto provocará que la UI se vuelva a renderizar con los datos modificados para que podamos leer nuestras propias escrituras.**

A continuación, necesitamos implementar una bandeja de salida que encolará nuestras mutaciones cuando estemos sin conexión y las ejecutará en segundo plano cuando recuperemos la conectividad.

```
import {BackgroundSyncPlugin} from 'workbox-background-sync';
const bgSyncPlugin =
```

```
  new BackgroundSyncPlugin('outbox', {
  maxRetentionTime: 24 * 60, // retry for max hours
});
registerRoute(
  ({ request }) => request.url.includes('/api-')),
  new NetworkOnly({
    plugins: [bgSyncPlugin],
  }),
  'PUT',
);
```

En este ejemplo, usamos la sincronización en segundo plano proporcionada por la biblioteca `Workbox`. Añadimos este código a nuestro service worker.

- Primero, inicializamos el `BackgroundSyncPlugin` que reintentará transacciones en segundo plano hasta por 24 horas.
- Luego registramos el plugin para solicitudes `PUT` a nuestros servicios BFF (es decir, URLs que comienzan con /api-).
- También registraremos para solicitudes `DELETE`.

El plugin almacena las solicitudes en IndexedDB (https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API). Así, **podemos usar estos datos para proporcionar una vista de la bandeja de salida al usuario.**

Actualizaciones en vivo (live updates)

Entendemos que los datos en caché pueden volverse obsoletos cuando una aplicación se queda sin conexión. Esta es una concesión que estamos dispuestos a hacer para poder seguir trabajando cuando perdemos conectividad.

Implementaremos una solución de actualizaciones en vivo que capitaliza nuestra arquitectura event-first. Veamos dos implementaciones alternativas del servicio de actualizaciones en vivo. Primero veremos la implementación con WebSocket y luego la alternativa con long-polling.

WebSocket

Nuestro sistema está compuesto por servicios autónomos. Estos servicios publican eventos de dominio en un hub de eventos a medida que cambia su estado. Una micro aplicación interactúa con un conjunto específico de entidades de dominio y las mantiene en una caché local. Para mantener esta caché actualizada, queremos que el frontend se suscriba a los eventos de dominio, para que podamos actualizar la caché local en tiempo casi real cuando estamos en línea.

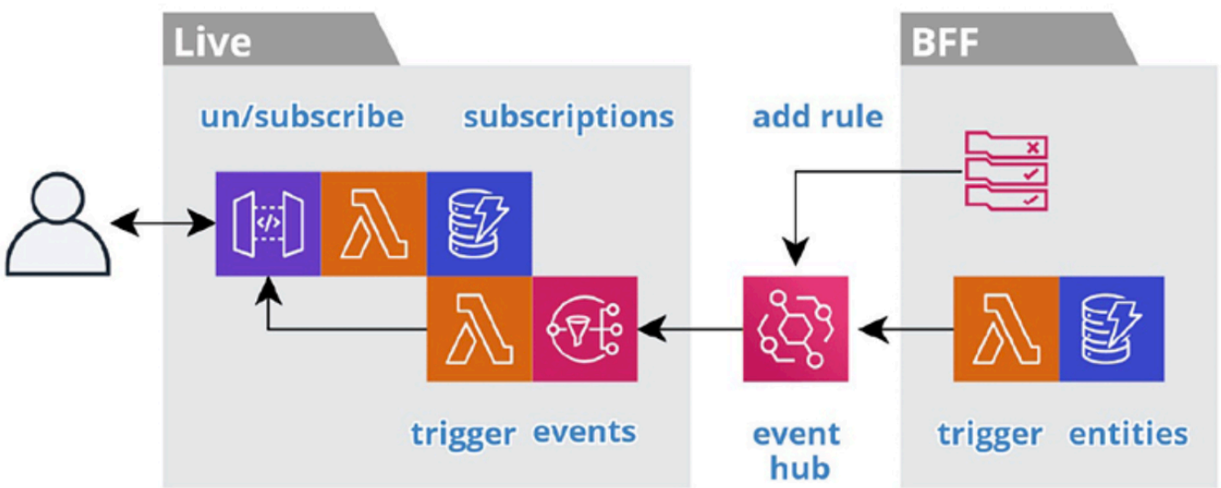

Figure 3.4: Live updates – WebSocket

El servicio de actualizaciones en vivo define un topic de eventos, usando AWS SNS. Los servicios BFF de las distintas micro aplicaciones añaden reglas al hub de eventos para los eventos de dominio específicos que desean enrutar al topic de eventos.

Una micro aplicación compartida inicia una conexión con el servicio de actualizaciones en vivo para crear el WebSocket del usuario, y la URL de callback del WebSocket se almacena en la base de datos de suscripciones.

Las micro aplicaciones individuales pueden añadir suscripciones específicas para controlar los eventos que recibirá el usuario.

Una función trigger reacciona a los eventos a medida que llegan al topic de eventos. La función busca suscripciones en la base de datos y reenvía los eventos a las URLs de callback de los usuarios.

La micro-app compartida recibe los eventos a través del WebSocket en el cliente y pone los eventos a disposición de otras micro aplicaciones a través de un escuchador de eventos de ventana, como discutimos en la sección *Comunicación entre aplicaciones*.

Desde aquí, las micro aplicaciones individuales toman el control. Por ejemplo, una micro aplicación basada en React puede aprovechar react-query para la gestión de estado y persistir el estado en la caché local.

La micro-app compartida reenvía (es decir, despacha) los eventos de dominio a las micro-apps suscritas, que actualizan el estado de su caché y, en consecuencia, actualizan la pantalla con los nuevos valores.

Long polling

La implementación con WebSocket funciona bien cuando una aplicación está en línea, pero no soporta directamente una aplicación cuando está sin conexión. La Figura 3.5 describe una implementación alternativa que usa AWS SQS para almacenar en búfer los eventos cuando un usuario está sin conexión:

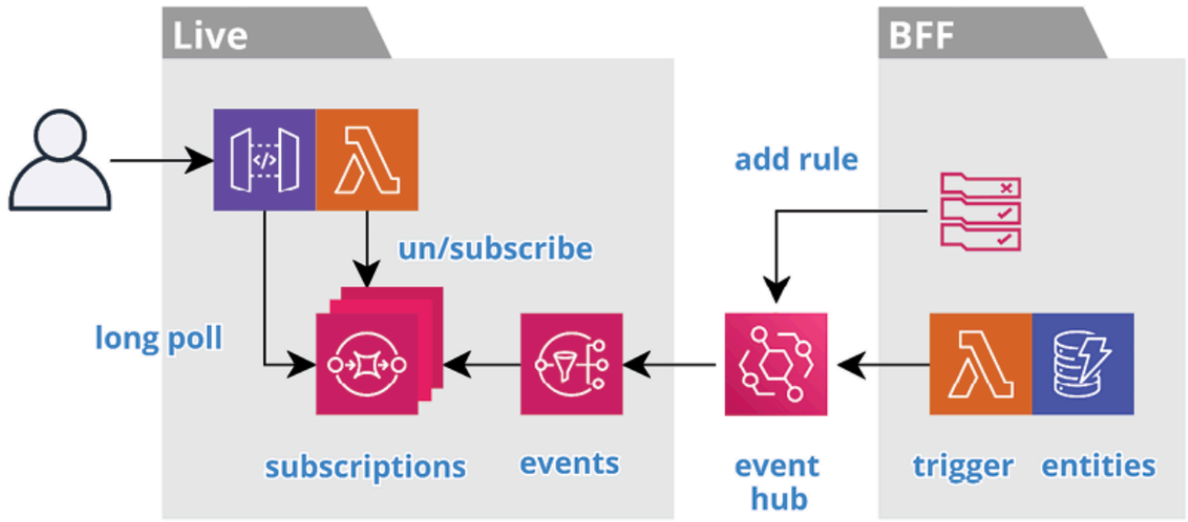

Figure 3.5: Live updates – long polling

El lado derecho de la Figura 3.5 es idéntico a la Figura 3.4. La diferencia comienza después de que el topic de eventos recibe los eventos. La micro aplicación compartida se suscribe al servicio de actualizaciones en vivo y crea una cola SQS dedicada para el usuario actual y registra la cola con el topic de eventos.

La micro aplicación compartida realiza el long polling para el usuario y esperará hasta el máximo de 20 segundos de SQS. Cualquier evento que llegue a la cola durante ese intervalo fluirá inmediatamente a través de la conexión en tiempo casi real.

Desde aquí, la micro aplicación compartida pone los eventos a disposición de las otras micro aplicaciones a través de escuchadores de eventos de ventana, y las micro aplicaciones toman el control, tal como en la implementación con WebSocket.

Cuando el usuario se queda sin conexión, la aplicación compartida deja de hacer polling. Mientras tanto, los eventos se acumulan para el usuario en sus colas dedicadas. Cuando se restablece la conexión, el long polling se reanudará y los eventos almacenados en búfer fluirán y actualizarán la caché.