ASCO DE CODIGO

"Asquerosamente Remediable"

Enrutamientos

Julio 13, 2018

Introduciendo rutas en nuestra aplicación

Dejamos por un momento los componentes y nos centramos en una nueva funcionalidad del framework: el enrutamiento. Una vez que hemos creado nuestros componentes base o comunes (botones, inputs, listados, items, cartas...) y nuestros componentes de negocio (formularios, widgets, buscadores...) puede ser que necesitemos crear componentes vista o página para crear una aplicación completa (un SPA).

Dependiendo del número de vistas de las que se componga mi aplicación, la navegación entre ellas será más o menos fácil. Por ello, cuando una aplicación empieza a crecer en el número de vistas, es buena idea incluir algún mecanismo o herramienta que nos permita escalar este problema de la manera adecuada.

En el post de hoy - y en los sucesivos - veremos las formas en las que podemos incluir un sistema de navegación en nuestra aplicación VueJS de una manera escalable y poco intrusiva. Síganme, por favor:

¿Qué es un sistema de rutas en mi aplicación?

Es un sistema que permite configurar la navegación de nuestra aplicación. Suele componerse de una librería que se encuentra interceptando la ruta que indicamos en nuestro navegador para saber en todo momento a qué estado de la aplicación debe moverse.

Un enrutador nos permite decir, para una url determinada, qué componente renderizar. Está muy basado en los sistemas de Modelo-Vista-Controlador y el diseño de API Rest. Suelen ser muy útiles para gestionar de una manera centralizada el comportamiento y el viaje que debe llevar un usuario por nuestra aplicación. Al final no deja de ser una forma de solucionar las diferentes direcciones web con las que cuenta mi aplicación en un sistema SPA.

¿Y si no necesitamos un sistema de rutas?

En muchas ocasiones, incluimos una librería compleja de gestión de la navegación sin plantearnos siquiera si lo necesitamos. Imaginad que hemos desarrollado una pequeña web donde presentamos nuestro producto, la típica web estática que no presenta más de 7 u 8

páginas diferentes.

Si por un casual, hemos decidido desarrollarla con VueJS, puede ser tentador usar su librería hermana vue-router. Sin embargo, quizá añadamos complejidad sin necesidad.

Y si no añadimos un enrutador ¿Cómo lo hacemos? Podemos preparar nosotros una pequeña solución que permita este dinamismo. Por ejemplo, con algo parecido a esto:

```
const NotFound = { template: '<p>Page not found</p>' };
const Home = { template: '<p>home page</p>' };
const About = { template: '<p>about page</p>' };

const routes = {
  '/': Home,
  '/about': About
};

new  Vue({
  el: '#app',
  data: {
    currentRoute: window.location.pathname
  },
  computed: {
    ViewComponent () {
      return routes[this.currentRoute] || NotFound
    }
  },
  render (h) { return h(this.ViewComponent) }
});
```

Lo que hacemos, en este caso, es apoyarnos en la funcionalidad de propiedades computadas que nos ofrece el framework para conseguir dinamismo. Lo que conseguimos es que cada vez que la variable `currentRoute` cambie, se ejecute la función `ViewComponent` que devuelve el componente que hayamos configurado en nuestro array `routes` . Si la ruta puesta en el navegador no es correcta, renderizamos el componente `NotFound` .

La implementación es bastante sencilla y nos va a permitir la navegación por nuestra aplicación sin hacer mucho más. Si nuestra aplicación empieza a crecer, tenemos que tener en cuenta que una solución como esta es limitada y que deberemos ir pensando en incluir algo más elaborado.

Y si lo necesitamos ¿Cómo empezamos?

Si necesitamos algo más elaborado, puede ser una buena idea que diseñemos nuestra propia librería como hace Juanma en este post, o que usemos una de las librerías con las que ya contamos en la comunidad como es el caso de Director.

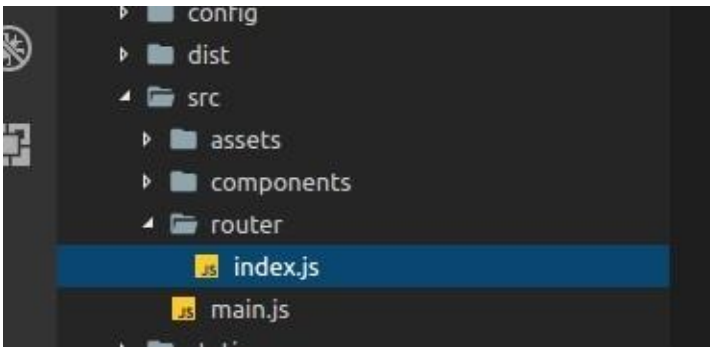
Tenemos también la opción de extender VueJS con una nueva librería llamada vue-router, esta es la opción que vamos a explicar. Es la opción que mejor se adapta al propio framework y que nos va a proporcionar todo lo necesario.

Para usarla tenemos varias formas: si es un proyecto que ya tenemos empezado, por medio de la línea de comandos:

```
$ npm install vue-router --save
```

O creando un proyecto desde 0 con vue-cli, indicando en la configuración del proyecto, que queremos una plantilla con vue-router, como ya explicamos en el post anterior.

Una vez que tenemos esto, lo siguiente será configurar nuestras rutas e indicar a VueJS que incluya esta configuración en su contexto. Para hacer esto, creamos una carpeta en la raíz del proyecto de esta manera:



Dentro del fichero `index.js` vamos a ir incluyendo toda la configuración de rutas de nuestra aplicación. Dentro de este fichero incluimos las siguientes líneas:

```
import Vue from 'vue';
import Router from 'vue-router';

Vue.use(Router);

export default new Router({});
```

Lo que estamos haciendo es importar tanto la librería de vue como la de `vue-router` . Lo siguiente es extender VueJS por medio de `Vue.use(Router)` . De esta manera, extendemos la funcionalidad con un plugin. Por último, devolvemos la instancia del router que vamos a configurar.

Para terminar de integrar vue-router totalmente, solo nos falta inyectar esta instancia en todos los componentes. Esto lo conseguimos yendo al `fichero main.js` , donde se encuentra el 'setup' inicial de mi aplicación vue. Dentro ponemos lo siguiente:

```
import Vue from 'vue';
import router from './router';
import App from './components/app/app.vue';

new Vue({
    el: '#app',
    router,
    render: h => h(App)
});
```

Lo único que hacemos es inyectar en la instancia principal de nuestra aplicación vue nuestro router para que sea accesible a todo el árbol de componentes.

Si ahora queremos que estos componentes se pinten, `vue-router` cuenta con un componente específico donde se irá incluyendo el componente que la ruta nos indique. En nuestro componente app, hay que añadir el componente `<router-view>` . Lo único que hace este componente es sustituirse por nuestra vista.

Ya está. No necesitamos más fontanería. Ya podemos empezar a configurar rutas.

¿Cómo configuramos rutas?

Dentro de una instancia del router contamos con un parámetro llamado routes que permite configurar nuestras rutas. Por ejemplo, podríamos hacer lo siguiente:

```
// router/index.js
import HomeView from '@/components/views/home-view.vue';

export default new Router({
    routes: [
        { path: '/home', component: HomeView }
    ]
});
```

De esta forma, cuando un usuario ponga en el navegador la ruta `www.mi-spa.com/#/home` , vue renderizará mi componente `HomeView` .

Dentro de este objeto podemos incluir otro nuevo parámetro llamado name. Este parámetro está muy bien para dar un nombre a nuestra ruta. De esta forma los desarrolladores desacoplan la url física del estado al que nos tenemos que dirigir y hace que podamos renombrar rutas muy largas. Por ejemplo:

```
// router/index.js
import HomeView from '@/components/views/home-view.vue';

export default new Router({
    routes: [
        {
            path: 'my/shop/detail/go/www/43544352/app/home',
            name: 'home',
            component: HomeView
        }
    ]
});
```

Ahora podemos usar el nombre para referenciarla sin tener que usar toda la ruta. Es recomendable siempre ponerlo y usar ese nombre dentro de nuestra app.

Cómo decíamos, al final nuestro sistema de rutas está muy pensado para cargar estados o recursos en nuestros componentes de página y por ello, además de rutas estáticas, podemos usar rutas dinámicas. Esto significa que yo puedo indicar que se renderice siempre un componente para una serie de rutas que tienen patrones en común. Por ejemplo:

```
// router/index.js
import ProductDetailView from '@/components/views/product-view.vue';

export default new Router({
    routes: [
        {
            path: 'products/:productId',
            name: 'product-detail',
            component: ProductDetailView
        }
    ]
});
```

Lo que hemos conseguido con esto es que tanto `/products/1234` como `products/3452` nos renderice el mismo componente. Siempre que queramos incluir una parte dinámica a nuestra ruta, tenemos que indicarlo con dos puntos `:` .

Este dinamismo nos puede ser muy útil para obtener productos por un id determinado de servidor dado que la parte dinámica es inyectada dentro de nuestros componentes en el campo `$route.params` .

Con este comportamiento se puede hacer cualquier cosa que se nos ocurra ya que `vue-router` usa la librería `path-to-regexp` para relacionar rutas por medio de expresiones regulares. Si necesitas algo mucho más específico sería bueno que le echases una ojeada.

Puedes preguntarte qué ocurriría si más de una ruta de las que has configurado coincide con la ruta especificada por el usuario. Las rutas son registradas en vue por orden en el array. Por tanto la prioridad será el orden en la que se especificó. En cuanto vue-router encuentra una coincidencia recorriendo el arreglo, ejecuta su renderizado. Tenlo en cuenta.

¿Cómo navegamos a nuevas rutas?

Una vez que hemos configurado nuestras rutas, podemos navegar entre ellas para que el usuario pueda realizar las acciones necesarias. Esta navegación la podemos hacer de dos maneras: de manera semántica por medio del componente `<router-link>` o de manera programática.

Si lo hacemos de manera semántica tendríamos que hacerlo de esta manera dentro de nuestros templates:

```
<router-link to="/home">Voy a home</router-link>
```

Esto se renderiza por lo siguiente:

```
<a href="#/home">Voy a home</a>
```

El parámetro to nos acepta un objeto como valor para conseguir ciertas cosas, por ejemplo indicar parámetros. Por ejemplo, con el caso anterior de los productos, yo puedo hacer lo siguiente:

```
<router-link :to="{ name: 'product-detail', params: { productId: 1234 }}">
   Voy a ver el producto 1234
</router-link>
```

Ese objeto se puede definir dentro del JS perfectamente.

Si queremos crear navegaciones en tiempo de lógica del componente, contamos con 3 métodos para poder hacerlo. Yo puedo hacer lo siguiente:

```
$router.push('home');
```

Donde home es la ruta a la que me quiero dirigir. Este método tiene la siguiente firma:

```
$router.push(location, onComplete?, onAbort?);
```

Puede también que queramos navegar una ruta, pero que no queramos que se guarde en el histórico de navegación. En este caso usaríamos el método replace de esta manera:

```
$router.replace('home');
```

En este caso, iremos a la vista home y reemplazará a la ruta en la que estamos actualmente en el histórico.

Por último, contamos con un método llamado go que nos permite decir el número de saltos hacia delante o hacia detrás que queremos dar en el histórico de navegación:

```
$router.go(-1);
```

Esto iría a la ruta anteriormente visitada.

Una de las cosas que me gusta de esta API de navegación es que los nombres no han sido puesto de manera caprichosa. Si nos fijamos en ellos, son un mapeo 1 a 1 de la API History. De manera nativa los navegadores cuentan con `window.history.pushState`, `window.history.replaceState` y `window.history.go`. Esto hace que si hemos usado la API nativa, usarla en vue nos resulte muy intuitivo.

Conclusión

Nunca se sabe de qué manera puede crecer una aplicación, por lo que suele ser difícil de antemano saber si se va a necesitar un sistema como `vue-router` o no. Es por ello que la forma en la que el ecosistema de VueJS nos permite ir incluyendo estas pequeñas funcionalidades, de manera progresiva, me parece todo un acierto para el aprendizaje y para la complejidad de nuestro proyecto.

Lo bueno de un sistema de enrutado como este es que, por lo general, se cuenta con una API muy sencilla, y aprender su mecanismo suele costar poco. La parte más difícil a la hora de desarrollar nuestra aplicación se encuentra en la parte de diseño. La parte donde tenemos que decidir como va a ser el flujo y la experiencia del usuario entre pantallas. Si tenemos claro en qué estado se tiene que encontrar en cada momento nuestro usuario, el resto es pan comido.

En los próximos posts, profundizaremos en la librería y aprenderemos sobre el ciclo de vida que tiene una ruta en nuestro sistema y en cómo sacar partido a sus hooks. Hasta el momento, esto es todo.

Nos leemos :)

Interceptores de navegación entre rutas

Una vez que hemos visto los conceptos básicos de la librería de rutas, es el turno de profundizar en otros conceptos. Dentro de nuestra SPA, nos vendría bien tener algún tipo de mecanismo para saber, en ciertos momentos de una navegación, qué hacer en ciertas situaciones.

La librería de vue-router cuenta en su haber con una funcionalidad para esto llamada Navigation Guards o interceptores de navegación. Estos interceptores son una serie de funciones que nos van a permitir realizar diferentes acciones entre la navegación de una ruta a otra.

Por ejemplo, estos interceptores nos pueden venir bien para hacer ciertas comprobaciones o modificaciones del estado de la aplicación. Nos pueden venir bien para realizar ciertas redirecciones o para abortar una navegación si algo no se encuentra cómo el sistema espera.

vue-router cuenta con varias opciones para incluir estos interceptores que van desde el registro del interceptor de manera global hasta el registro del interceptor de manera local. A lo largo del post vamos a explicar cada uno de ellos y el uso que le podemos dar:

Interceptores globales

`beforeEach`

Dentro de vue-router contamos con la posibilidad de registrar un interceptor que se ejecutará cada vez que se realice un cambio de ruta. Este interceptor se ejecuta de manera global, es decir, para toda las rutas a las que naveguemos, justamente antes de producirse la navegación.

La forma de registrar un interceptor global es con la siguiente sintaxis:

```
const router = new VueRouter({ ... });

router.beforeEach((to, from, next) => {
    // ...
});
```

beforeEach nos inyecta tres parámetros en la función de callback:

- **to**: Es el objeto router con la información de la ruta a la que voy.
- **from**: Es el objeto router con la información de la ruta de la que vengo.
- **next**: Es la función que me permite reanudar la navegación. Es una función que tiene un comportamiento bastante complejo ya que nos permite diferentes datos de entrada. Por ejemplo:
    - Si la ejecutamos sin parámetro (`next()`), la navegación se reanudará hacia la ruta indicada en `to` .
    - Si la ejecutamos con una cadena que nos indique otra ruta (`next('/')`), nos redirigirá a la ruta que le hemos indicado.
    - Si indicamos un valor booleano `false` (`next(false)`), abortará la redirección.
    - Incluso si yo paso la instancia de un error (`next(new Error()`), la navegación se abortará y se inyectará con esta instancia el callback de la función registrada en `onError` . Como veis, muy completito. Siempre debemos ejecutar esta función para que el flujo continúe.

Puedo incluir dentro de mi aplicación todos los interceptores globales que yo necesite. Por ejemplo, puedo tener esto:

```
const router = new VueRouter({ ... });

router.beforeEach((to, from, next) => {
    // ...
});

router.beforeEach((to, from, next) => {
    // ...
});
```

El orden de ejecución es FIFO (El primero en registrarse es el primero en ejecutarse).

Los interceptores globales nos pueden venir muy bien para comprobar algún estado global de la aplicación. Por ejemplo, puede venirnos muy bien para comprobar si un usuario en particular va a poder tener acceso a una determinada parte de la aplicación o no.

```
function existToken() {
    return !!localStorage.token;
}

router.beforeEach((to, from, next) => {
    if (to.path != '/login' && existToken()) {
        next();
    } else {
        next('login');
    }
});
```

El ejemplo es un caso muy simplificado de control de acceso en la parte privada de una aplicación. Lo que hace es comprobar si la aplicación va a navegar a una ruta diferente de login. De ser así, comprueba que tenga un token de sesión con la API, lo que significa que se tiene acceso. Si lo tiene, continúa con la navegación normal. De no ser así, redirige a la vista de login.

Será uno de los interceptores que más usemos.

`afterEach`

Este interceptor se ejecuta después de que todos los interceptores de los componentes se hayan ejecutado. Su sintaxis es esta:

```
router.afterEach((to, from) => { // ... });
```

Como podemos apreciar, en este caso no se inyecta la función `next` por lo que no será posible influir en la navegación.

Es un interceptor que vamos a usar poco y que nos puede venir bien para depurar las rutas de navegación.

Interceptores locales a una ruta

En ocasiones, puede que necesitamos influir en la navegación de una sola ruta y no en cada una de ellas. Cuando necesitamos un uso más específico en una ruta, podemos registrar una función en tiempo de configuración.

Por ejemplo, podemos hacer esto:

```
const router = new VueRouter({
    routes: [
        {
            path: '/login',
            component: LoginView,
            beforeEnter: (to, from, next) => {
                delete localStorage.token;
                next();
            }
        }
    ]
});
```

Lo que hacemos es limpiar la sesión del usuario antes de navegar a login. De esta manera nunca se nos olvidará quitar privilegios al usuario si se ha hecho un logout.

Este interceptor puede sernos útil para evitar ir a rutas intermedias en un proceso de compra. Por ejemplo, podemos evitar mostrar la vista detalle si el usuario no ha seleccionado antes en una vista previa ningún producto.

Interceptores locales a un componente

Podemos localizar todavía más cuándo interceptar la navegación. Podemos incluir interceptores a nivel de un componente. Dependiendo del contexto de navegación en el que se encuentre un componente, podremos hacer unas acciones u otras.

Este mecanismo interno en un componente es muy importante porque la librería `vue-router` cachea los componentes y evita que ciertos hooks - como los de creación o destrucción - no sean ejecutados siempre. Por tanto, estos interceptores nos podrán ayudar porque tienen acceso a la instancia interna del propio componente.

Si yo quisiera interceptar comportamientos de navegación para un componente, lo haría de la siguiente forma:

```
const CartSummary = {
    template: `...`,
    beforeRouteEnter (to, from, next) { },
    beforeRouteUpdate (to, from, next) { },
    beforeRouteLeave (to, from, next) { }
};
```

Estudiemos cada uno de ellos:

`beforeRouteEnter`

En este interceptor entramos cuando la navegación ha sido confirmada, pero todavía no se ha creado, ni renderizado el componente. Nos puede venir bien usarlo para saber si el componente tiene algún comportamiento de navegación especial antes de pintarse.

Si nosotros queremos influenciar en el estado de un componente, tenemos que esperar a que sea creado. Para conseguir esto, podemos hacerlo pasándole una función a `next()` de la siguiente manera:

```
const CartSummary = {
    template: `...`,
    beforeRouteEnter (to, from, next) {
        next(vm => vm.products = []);
    }
};
```

Este interceptor, es un buen sitio para iniciar el estado de un componente con datos de un servicio externo. Por ejemplo, voy a traer los productos de mi backend y a cargarlos:

```
const CartSummary = {
    template: `...`,
    beforeRouteEnter (to, from, next) {
        axios.get('/products', (res) => {
            next(vm => vm.products = res.data);
        });
    }
};
```

Como los interceptores detienen la navegación hasta que se ejecuta la función `next()` , nos vienen bien para gestionar este asincronismo.

`beforeRouteUpdate`

Se ejecuta cuando la ruta de navegación cambia y el componente está siendo reutilizado para la siguiente ruta o por la misma. Por ejemplo, imaginemos que la ruta ha cambiado en la parte dinámica, en sus parámetros.

Como dijimos, los componentes se encuentran cacheados y no ejecutan sus hooks de creación y destrucción. Como este interceptor tiene acceso a la instancia del componente, puede ser un buen momento para manipular el estado según las necesidades nuevas.

```
const CartSummary = {
    data() {
        return {
            products: null,
        };
    },
    template: `...`,
    beforeRouteUpdate (to, from, next) {
        this.products = [];
    }
};
```

Sin callbacks ni artificios. Directamente accediendo a la instancia porque el componente ya se encuentra instanciado. Nada de fontanería.

`beforeRouteLeave`

Por último, contamos con este interceptor que se ejecuta antes de cambiar de ruta y sabiendo que el componente no va a ser utilizado.

Como se ejecuta antes de ir a la nueva navegación, tiene acceso al estado del componente. Es muy utilizado para evitar navegaciones involuntarias y sin querer. Imagínate que el usuario ha dado sin querer a salir de la compra y tiene todo relleno.

Podemos usar ese interceptor para poner un popup e indicar si el usuario está de acuerdo con no guardar los cambios realizados.

```
const CartSummary = {
    template: `...`,
    beforeRouteLeave (to, from, next) {
        this.popup()
            .then(next)
            .catch(() => next(false);
    }
};
```

¿Cuál es el flujo de ejecución de estos Guards?

Como parece un poco confuso cuando se ejecuta cada uno de los interceptores. Os pongo una guía del flujo que se sigue:

1. La navegación es activada.
2. Se llama a todos los beforeRouterLeave que se hayan registrado y que no van a ser reutilizados en la siguiente ruta a la que voy.
3. Se llama a todos los interceptores globales beforeEach.
4. Se llama a todos los `beforeRouteUpdate` de los componentes que van a ser reutilizados en la siguiente ruta a la que voy.
5. Se llama a `beforeEnter` que hemos configurado en la ruta a la que voy.
6. Se resuelven toda la asincronía de componentes de esa ruta.
7. Se llama a beforeRouteEnter de los componentes que van a estar activos.
8. Se da la navegación como confirmada.
9. Se llama al interceptor `afterEach` .
10. Se llama a los callbacks pasados a next in beforeRouteEnter.

11. Y vuelta a empezar cuando se lanza una nueva navegación.

Conclusión

Una de las cosas malas de usar frameworks y librerías de terceros es que te tienes que adaptar a lo que ellos entienden por un buen momento para que tu enganches tu funcionalidad. Quizá para muchos estos interceptores cumplan con sus exigencias, quizá para otros esto sea un impedimento.

Lo bueno es que VueJS ha pensado en ello y nos da bastantes opciones para poder redirigir o abortar navegaciones o incluso hacer acciones de borrado, actualización o creación de estados influidos por la navegación que se nos pide.

El problema de los interceptores suele ser el de siempre: el de tener funcionalidad ejecutándose de una manera asíncrona, lo que hace más difícil depurar si no somos metódicos y no tenemos en cuenta cuando incluirlos y cuando no.

Me he encontrado muchas veces interceptores dios que hacen más de lo que deben o que incluso que no respetan la regla de única responsabilidad. Tengamos en cuenta que este uso debería ser un recursos limitado y muy justificado en nuestro contexto.

Conceptos avanzados

Con lo aprendido hasta ahora sobre `vue-router` , podríamos cubrir gran parte de la funcionalidad necesaria para un buen número de aplicaciones.

Sin embargo, cuando nos enfrentamos a aplicaciones más grandes, tener en cuenta otras posibilidades nos puede ayudar en términos de reutilización y buenas prácticas.

El posts de hoy está dedicado a estudiar todos aquellos conceptos avanzados de vue-router que pueden ayudarnos a mejorar y a darnos mayor versatilidad cuando desarrollamos en un proyecto con vue.

Para conseguir esto, nos centraremos en el anidamiento de rutas, el paso de propiedades a un ComponentView, la inclusión de meta información y el cambio de comportamiento del scroll de nuestra web. Vayamos con ello:

Anidar rutas

Durante esta serie hemos hecho ejemplos con rutas bastantes simples. Los sistemas de navegación de los ejemplos siempre han sido de una ruta a otra pero, ¿qué ocurre cuando

dentro de nuestro sistema existe una navegación entre subrutas?

Imaginemos por un momento, que tenemos que desarrollar una aplicación bancaria y que estamos implementando la funcionalidad de transferencias entre cuentas. Nuestra aplicación podría contar, por ejemplo, con un proceso dividido en 3 pantallas:

- Una pantalla para configurar los datos de la transferencia: En esta pantalla, el usuario indica el importe y el destinatario al que realizar la transferencia.
- Una pantalla para mostrar el detalle de la transferencia: Nos puede ser muy útil para mostrar el estado actual de la cuenta, el día en que se hará la transferencia y el cálculo de las comisiones que conlleva la operación.
- Y una pantalla de confirmación de la transferencia realizada: Se le muestra al usuario cuando toda la operación de transferencia fue correctamente.

Para realizar esto, podríamos diseñar 3 vistas que se corresponderían con estas rutas:

```
/transfer/config
/transfer/detail
/transfer/confirm
```

Como vemos, la propia funcionalidad, me lleva a tener un subenrutado o anidamiento de rutas.

Estos procesos suelen conllevar una interfaz parecida en cada vista para favorecer la navegación al usuario. Por ejemplo, las 3 vistas van a compartir unas cabeceras y un componente de navegación que nos indique en qué paso de la realización de la transferencia nos encontramos.

Para desarrollar esto, podemos hacerlo de 3 maneras diferentes:

1. Creando un HTML diferente para cada una de las vistas.
2. Extrayendo todo el HTML común y creando componentes que se utilizasen en las 3 vistas.
3. Creando una 'layout' que contenga toda la parte común a las 3 vistas e ir rellenando de manera dinámica la diferencia entre vistas.

La opción 1 la descartamos porque no respeta los principios de reutilización. Si necesito hacer un cambio en cualquier momento, tengo que cambiar 3 vistas.

La opción 2 es bastante buena, pero no llega a ser lo suficientemente reutilizable; Sí, estamos reutilizando componentes comunes, pero aún hay partes que pueden cambiar en el tiempo y que me hagan trabajar más de la cuenta, manteniendo código idéntico.

La opción 3 es la mejor porque respeta los principios de reutilización ya que toda la parte común se encuentra encapsulada en un componente padre que orquesta a 3 componentes hijos.

La idea es hacer algo como esto:

Desarrollar un componente común que se llame `TransferView` que haga de componente contenedor, y 3 componentes hijos, más específicos, llamados `ConfigView` , `DetailView` y `ConfirmView` .

Aunque con vue tendríamos soporte para desarrollar esto, vue-router nos da una funcionalidad que se apoya en un sistema de rutas anidadas. Yo podría configurar mi ruta de la siguiente manera:

```
const TransferView = {
    template: `
        <div>
            <h2>Cabecera func. transferencia</h2>
            <router-view></router-view>
        </div>
    `
};

const ConfigView = {
    template: '<h3>Paso configuración</h3>'
};

const DetailView = {
    template: '<h3>Paso detalle</h3>'
};

const ConfirmView = {
    template: '<h3>Paso confirmación</h3>'
};

const router = new VueRouter({
    routes: [
        {
            path: '/transfer',
            component: TransferView,
            children: [
                { path: 'config', component: ConfigView },
                { path: 'detail', component: DetailView },
                { path: 'confirm', component: ConfirmView }
            ]
        }
    ]
});
```

Lo que hacemos es incluir un nuevo parámetro en la ruta llamado children. En este campo, podemos configurar todas las rutas anidadas que necesitemos. En nuestro caso 3.

Lo último que hemos hecho es añadir el componente `router-view` en nuestro componente padre `TransferView` . De esta manera vue-router sabe en qué parte tiene que renderizar el componente hijo.

Este anidamiento puede ser todo lo profundo que queramos, simplemente tenemos que incluir el parámetro `children` , configurar las rutas con su componente hijo que queramos e indicar en el componente padre dónde pintarlo por medio de `router-view` .

El anidamiento suele utilizarse también para crear el layout principal de nuestra aplicación. Como todas las vistas contarán con el header, el menú y el footer, lo que se hace es crear un componente `AppView` que contiene estos elementos y un componente `router-view`

donde se renderizará la parte de la vista más específica.

Pasar propiedades a un ComponentView

Aunque un `ComponentView` es un componente muy específico dentro de nuestra aplicación, y es bastante improbable que pueda ser reutilizado por estar tan acoplado con el negocio, es buena idea usar las mismas buenas prácticas que usamos en componentes más específicos.

Cuando un componente de vista hace uso de parámetros para su correcto funcionamiento, es necesario que no lo acoplemos ,dentro de lo posible, a `vue-router` y que usemos la funcionalidad de propiedades de entrada especificada por vue.

Por tanto, intentemos evitar este tipo de accesos:

```
const ProductDetailView = {
  template: '<label>ProductId {{ $route.params.productId }}</label>'
};

const router = new VueRouter({
  routes: [{
    path: '/products/:id',
    component: ProductDetailView
  }]
});
```

Por el siguiente:

```
const ProductDetailView = {
  props: ['productId'],
  template: '<label>ProductId {{ productId }}</label>'
};

const router = new VueRouter({
  routes: [{
    path: '/products/:id',
    component: ProductDetailView,
    props: true
  }]
});
```

Lo que hemos hecho es decir a la ruta que active el paso de parámetros por medio de propiedades al componente. De esta forma el componente está más desacoplado y nos ayuda a su reutilización y a ser probado de manera aislada.

El atributo props nos permite varios valores según nuestras necesidades. Por ejemplo, puedo pasar un objeto para 'mapear' manualmente las propiedades con los parámetros. O incluso, puede que la ruta no cuente con parámetros, pero sí con propiedades que hay que iniciar. Este caso nos puede ser útil.

Un caso puede ser este.

```
const router = new VueRouter({
  routes: [
    {
      path: '/promotion/from-newsletter',
      component: PromotionView,
      props: { newsletterPopup: false }
    }
  ]
});
```

Hacemos que el popup del componente no se muestre al principio al navegar a esta ruta. No recibimos parámetros de la ruta, pero si pasamos propiedades al componente.

Puedo pasar también una función. Nos puede ayudar a tomar decisiones sobre el valor que quiero incluir en la propiedad del componente dependiendo de la ruta. Por ejemplo:

```
const router = new VueRouter({
    routes: [
        {
            path: '/search',
            component: SearchView,
            props: (route) => ({ query: route.query.q })
        }
    ]
});
```

En este caso usamos la función para incluir la query realizada en la url como propiedad del componente `SearchView` .

Incluir meta-información de una ruta

Puede darse el caso que necesitemos incluir información específica para una ruta. Puede sernos útil marcar ciertas rutas para influir en su comportamiento. Esto se puede hacer incluyendo nuevos campos en el atributo `meta` del objeto `route` de la siguiente manera:

```
const router = new VueRouter({
    routes: [
        {
            path: '/login',
            component: LoginView,
            meta: {
                isPublic: true
            }
        }
    ]
});
```

De esta manera estamos marcando que la ruta login es pública para cualquier usuario. Hay que tener cuidado con esta meta información porque es heredada de padres a hijos por lo que tengámoslo en cuenta si pensamos que algo está yendo mal.

Esta información es inyectada a los componentes dentro de `$route.matched` y es accesible tanto dentro de los componentes como de los interceptores de navegación. Por ejemplo, podemos combinar este campo con el interceptor `beforeEach` :

```
router.beforeEach((to, from, next) => {
    if (!to.matched.some(record => record.meta.isPublic) && !auth.loggedIn()) {
        next({
           path: '/login', query: { redirect: to.fullPath }
        });
    } else {
        next();
    }
});
```

Si la ruta no es pública, ni el usuario tiene autorización en el sistema, se devuelve al usuario a la pantalla de login. Esto se ejecutará para todas las rutas a las que naveguemos.

Poner nuestro sistema de rutas en modo History de HTML5

Un SPA suele contar con un sistema de rutas precedido por una almohadilla. De esta forma, el navegador sabe que no tiene que resolver la ruta contra un servidor, ni hace una recarga de la página, sino que intenta solucionar la ruta a nivel interno de navegador. De esta manera, la librería de rutas intercepta el nuevo valor y actúa según su configuración.

Los que ya hemos trabajado con otros SPAs sabemos que las rutas suelen tener esta apariencia:

```
https://my-web.com/#/app/login
```

Evitar una URL como esta puede deberse a 3 razones:

- Puede que desde negocio se quiera trabajar en un sistema de rutas usable para que el usuario pueda recordarlas o guardarlas en favoritos.
- Puede que no nos interese como desarrolladores dar pistas al usuario del tipo de herramientas que usamos para nuestro desarrollo (Cuando en una web se ve un hash de este tipo es indicativo de SPA).
- Puede que tengamos problemas de SEO al no poder acceder a ciertas rutas.

Sería bastante bueno que nuestra librería pudiese entender correctamente la siguiente ruta:

```
https://my-web.com/app/login
```

Esto es posible en vue gracias a nuestra librería de rutas `vue-router` . Yo puedo configurarlo de esta manera para que se comporte de forma nativa a como lo hace el navegador:

```
const router = new VueRouter({
    mode: 'history',
    routes: [...]
});
```

De esta forma, tenemos el comportamiento esperado. Lo malo de esto es que ahora todas las rutas harán una petición a nuestro servidor de aplicaciones y que dependiendo de la herramienta que usemos, el error será gestionado de una u otra manera, pudiendo romper el funcionamiento correcto de nuestra aplicación.

Este mecanismo se puede configurar de diferentes formas. Por ejemplo, si nuestra aplicación es servida por un Apache, debemos registrar la siguiente regla en su configuración:

```
<IfModule mod_rewrite.c>
  RewriteEngine On
  RewriteBase /
  RewriteRule ^index\.html$ - [L]
  RewriteCond %{REQUEST_FILENAME} !-f
  RewriteCond %{REQUEST_FILENAME} !-d
  RewriteRule . /index.html [L]
</IfModule>
```

Esta regla nos redirige a index.html cada vez que no encuentra la ruta especificada. De esta forma, la aplicación sabrá reiniciarse correctamente.

Si usamos un NGINX, lo deberíamos hacer así:

```
location / {
    try_files $uri $uri/ /index.html;
}
```

Si queremos que el propio NodeJS nos gestione esto, contamos con un middleware de Express que nos permite configurar este redireccionamiento a nivel de servidor: connect-history-api-fallback.

El problema que seguimos teniendo con esto es que si la ruta no existe se nos seguirá redirigiendo a index.html no dando información al usuario de que esa ruta no existe. Para solucionar esto podemos registrar una ruta genérica en vue-router que siempre se ejecutará cuando ninguna otra regla haya conseguido relacionarse:

```
const router = new VueRouter({
    mode: 'history',
    routes: [{
        path: '*',
        component: NotFoundView
    }]
});
```

Para cualquier ruta (wildcard *), mostramos el componente `NotFoundView` . De esta forma siempre damos una información adecuada al usuario.

Cambiar el comportamiento del scroll

Otro 'daño colateral' de usar el modo histórico de HTML5 y no el comportamiento de URLs por defecto de un SPA, es que podemos gestionar en qué parte del scroll queremos colocar nuestra nueva vista; Cuando naveguemos podemos preferir que la nueva vista siempre se sitúe en su parte superior o que mantenga el scroll de la ruta de la que procedemos.

En el objeto router contamos con otro parámetro para gestionar esto:

```
const router = new VueRouter({
    routes: [...],
    scrollBehavior (to, from, savedPosition) {
        return { x: 0, y: 0 }
    }
});
```

Esta función `scrollBehavior` nos permite acceder a los datos de la ruta de la que vengo y de la que voy. Además, se cuenta con un parámetro opcional que solo es inyectado si el cambio de ruta es provocado por los botones de navegación del propio navegador.

En el ejemplo de arriba, hemos indicado que scroll siempre se sitúe en la parte superior e izquierda de la página. Puede darse el caso que queramos simular una navegación hasta un 'anchor' determinado. Esto lo podríamos conseguir así:

```
scrollBehavior (to, from, savedPosition) {
    if (to.hash) {
        return {
            selector: to.hash
        }
    }
}
```

Esta función tiene acceso a los datos de la meta información de una ruta, por lo que el comportamiento del scroll es tan configurable como nosotros necesitemos.

> Nota: Esta funcionalidad, como decíamos, solo funciona con el modo 'history' activo.

Todo junto

Ahora vamos a juntar todos estos conocimiento y a hacer un pequeño ejemplo. Vamos a implementar del todo la funcionalidad de transferencias de apartados anteriores.

Lo primero que hacemos es crear el servidor de NodeJS que servirá nuestra aplicación:

```
const express = require('express');
const history = require('connect-history-api-fallback');
const app = express();

app.use(history());
app.use(express.static('public'));

const server = app.listen(3001, "localhost", () => {
    const host = server.address().address;
    const port = server.address().port;
    console.log('Running at http://' + host + ':' + port);
});
```

Hemos incluido el fallback del histórico ya que nuestro ejemplo estará configurado con modo histórico de HTML5, el modo que ponía las URLs 'bonitas', recuerda.

Lo siguiente es desarrollar los componentes y la configuración de rutas:

```
const TransferView = {
    template: `
        <div>
            <h2>Transferencia nacional</h2>
            <router-view></router-view>
        </div>
    `
};

const ConfigView = {
    data() {
        return {
            amount: 0,
            iban: 'ES6712345678987612 32'
        }
    },
    template: `
        <div>
            <h3>Configurar transferencia</h3>
            <form @submit.prevent="showDetail">
                <label for="amount">Cantidad</label>
                <input id="amount" type="text" v-model="amount"/>
                <label for="iban">IBAN</label>
                <input id="iban" type="text" v-model="iban"/>
                <button>Realizar transferencia</button>
            </form>
        </div>
    `,
    methods: {
        showDetail() {
            this.$router.push({
                name: 'detail',
```

```
                params: { amount: this.amount, iban: this.iban }
            });
        }
    }
};

const DetailView = {
    props: ['amount', 'iban'],
    template: `
        <div>
            <h3>Detalle de la transferencia que va a realizar</h3>
            <p>Cantidad: {{ amount }} €</p>
            <p>IBAN: {{ iban }}</p>
            <router-link :to="{ name: 'confirm' }">Confirmar</router-link>
        </div>
    `
};

const ConfirmView = {
    template: `
        <div>
            <h3>Transferencia realizada correctamente</h3>
            <router-link to="/">Volver</router-link>
        </div>
    `
};

const NotFoundView = {
    template: `
        <div>
            <router-link :to="{ name: 'config' }">Empezar transferencia</router-link>
        </div>
    `
};

const routes = [
    {
        path: '/transfer',
        component: TransferView,
        children: [
            { path: 'config', name: 'config', component: ConfigView },
            { path: 'detail', name: 'detail', component: DetailView, props: true },
            { path: 'confirm', name: 'confirm', component: ConfirmView },
        ]
    },
    { path: '*', name: 'not-found', component: NotFoundView }
];


const router = new VueRouter({
    mode: 'history',
    routes,
    scrollBehavior(to, from, savedPosition) {
```

```
        return { x: 0, y: 0 }
    }
});

const app = new Vue({ router }).$mount('#app');
```

Si vemos el ejemplo con detalle, veremos que se incluye la funcionalidad de history mode HTML5, el anidamiento de rutas, el paso de parámetros como propiedad y el control del scroll.

Conclusión

A lo largo de estos 3 últimos posts hemos hecho un repaso a todo lo que puede aportarnos una librería como `vue-router` . No es una librería innovadora, ni lo necesita ser, simplemente es una opción que se integra perfectamente con el ecosistema de vue para la gestión de rutas.

Terminado este capítulo, entraremos en una nueva fase de la serie donde estudiaremos la gestión del estado en una aplicación. Estamos muy cerca de contar con todas las piezas necesarias para poder hacer una aplicación completa, robusta y escalable con el ecosistema de vue.

Por ahora la experiencia está mereciendo la pena y lo aprendido es coherente con lo que muchos fronts han demandado a lo largo de los años a un framework JavaScript.