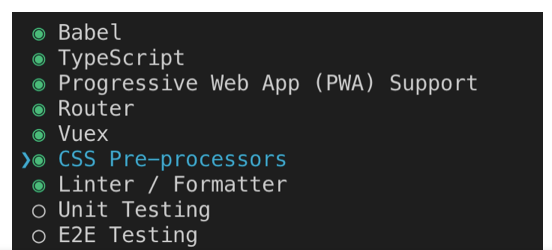
Comenzamos creando de nuevo el proyecto:

vue create mynotificator

Pero en esta ocasión activamos las siguientes características:



La compatibilidad con PWA para que cree el archivo manifest y la compatibilidad con webworkers.

La gestión del router para tener diferentes “páginas” y rutas en nuestra SPA (Single Page Application)

La librería Vuex que gestiona el estado de la aplicación. Es un concepto similar a la varible state de React y permite detectar los cambios que suceden en los datos y optimizar el renderizado de aquellas partes del DOM involucradas.

Un interesante tutorial lo encontramos en <https://www.vuemastery.com/courses/mastering-vuex/intro-to-vuex/>

#### Código fuente

Si ahora analizamos el proyecto que se ha creado veremos algunos cambios:

El archivo main.ts:

import Vue from 'vue';

import App from './App.vue';

import router from './router';

import store from './store';

import './registerServiceWorker';

Vue.config.productionTip = false;

new Vue({

router,

store,

render: (h) => h(App),

}).$mount(‘#app');

Importan un archivo de configuración de rutas que ahora analizaremos y un archivo de gestión del estado.

El archivo de rutas: ./router.ts

export default new Router({

mode: 'history',

base: process.env.BASE\_URL,

routes: [

{

path: '/',

name: 'home',

component: Home,

},

{

path: '/about',

name: 'about',

// route level code-splitting

// this generates a separate chunk (about.[hash].js) for this route

// which is lazy-loaded when the route is visited.

component: () => import(/\* webpackChunkName: "about" \*/ './views/About.vue'),

},

],

});

Define al componente Home para inyectar en caso de la ruta por defecto y al componente About para la ruta /about.

Qué cambia en App.vue:

<template>

<div id="app">

<div id="nav">

<router-link to="/">Home</router-link> |

<router-link to="/about">About</router-link>

</div>

<router-view/>

</div>

</template>

Fundamentalmente hace uso del componente <router-view> donde se inyectarán los componentes que hemos definido en router.ts. También es interesante ver cómo se definen los enlaces con:

<router-link to="/">Home</router-link> |

<router-link to=“/about">About</router-link>

El componente Home.vue es similar al original App.vue que teníamos antes y About es una plantilla simple de HTML.

El archivo store.ts:

import Vue from 'vue';

import Vuex from 'vuex';

Vue.use(Vuex);

export default new Vuex.Store({

state: {

},

mutations: {

},

actions: {

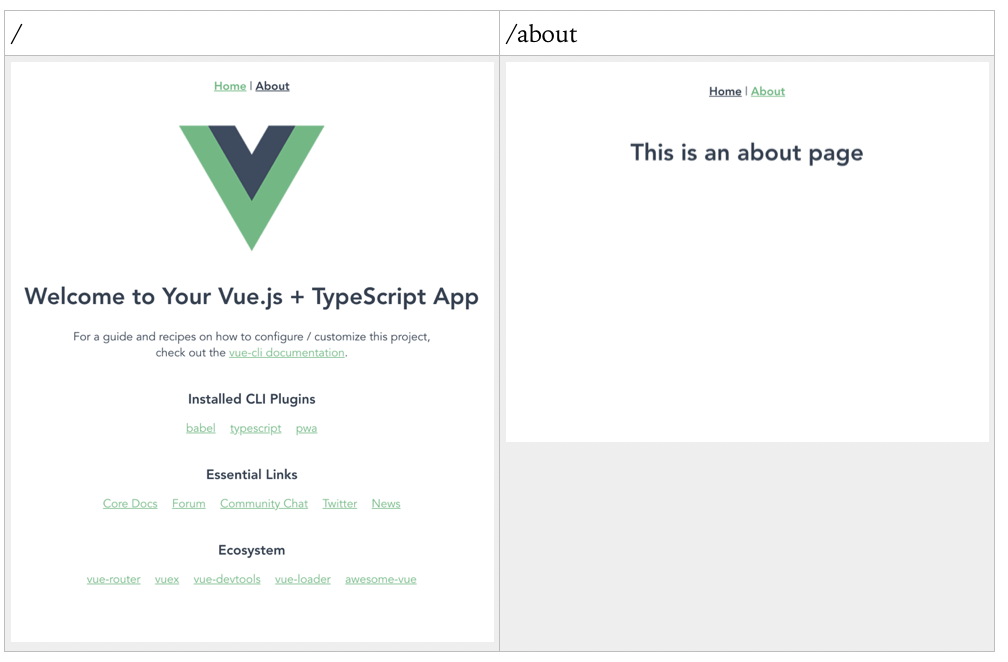
},

});

Define el estado, las mutaciones y acciones que definiremos y explicaremos más adelante más adelante.

Si ejecutamos nuestra aplicación:

npm run serve



Ahora comenzaremos realizando los cambios necesarios para realizar una aplicación de lista de mensajes con imágenes. Para la interfaz gráfica emplearemos el framwork basado en Material (<https://material.io>) Vuetify:

npm install vuetify —save

En el archivo main.ts debemos realizar los siguientes cambios para que el framework se encuentre disponible en todos los componentes.

import Vue from 'vue';

import App from './App.vue';

import router from './router';

import store from './store';

import './registerServiceWorker';

// Vuetify

import Vuetify from 'vuetify';

import 'vuetify/dist/vuetify.min.css';

Vue.config.productionTip = false;

const vuetifyOptions = {};

Vue.use(Vuetify);

new Vue({

router,

store,

render: (h) => h(App),

vuetify: new Vuetify(vuetifyOptions),

}).$mount(‘#app');

El componente principal App.vue contiene el siguiente código:

<template>

<div id="app">

<v-app>

<v-content>

<TabBar />

<router-view/>

</v-content>

</v-app>

</div>

</template>

<script lang="ts">

import { Vue, Component } from 'vue-property-decorator';

import TabBar from './components/TabBar.vue';

@Component({

components: {

TabBar,

},

})

export default class App extends Vue {}

</script>

<style lang="scss">

#app {

font-family: 'Avenir', Helvetica, Arial, sans-serif;

-webkit-font-smoothing: antialiased;

-moz-osx-font-smoothing: grayscale;

text-align: center;

color: #2c3e50;

}

</style>

Como se observa, emplea un componente personalizado TabBar que se encuentra en /src/components/TabBar.vue:

<template>

<v-tabs fixed-tabs background-color="indigo" dark>

<v-tab to="/">Home</v-tab>

<v-tab to="/about">About</v-tab>

</v-tabs>

</template>

<script lang="ts">

import { Component, Vue } from "vue-property-decorator";

@Component

export default class TabBar extends Vue {

}

</script>

Solo añade una barra de navegación de Vuetify.

La página About.vue quedará prácticamente como en la versión original, solo incluimos un enlace para ver cómo funciona el enrutamiento en vue:

<template>

<div class="about">

<h1>This is an about page</h1>

<router-link to="/">Home</router-link>

</div>

</template>

Si embargo, Home.vue si cuenta con muchos más cambios:

<template>

<div class="home">

<Items

v-for="todo in todos"

v-bind:key="todo.key"

v-bind:mykey="todo.key"

v-bind:message="todo.message"

v-bind:image="todo.image"

@click.native="select(todo)"

v-bind:class="selected===todo?'selected':''"

/>

<!-- click.native for click event on root component element -->

<v-dialog v-model="showAlert" persistent max-width="290">

<v-card>

<v-card-title class="headline">Warning!</v-card-title>

<v-card-text>{{messageAlert}}</v-card-text>

<v-card-actions>

<v-spacer></v-spacer>

<v-btn color="green darken-1" text @click="showAlert = false">Ok</v-btn>

</v-card-actions>

</v-card>

</v-dialog>

<editItem v-bind:show="showEdit" @submit="itemEdited" v-bind:\_item="selectedToEdit" />

<v-speed-dial

v-model="fab"

bottom

right

direction="top"

open-on-hover

transition="slide-y-reverse-transition"

>

<template v-slot:activator>

<v-btn v-model="fab" color="blue darken-2" dark fab>

<v-icon v-if="fab">close</v-icon>

<v-icon v-else>account\_circle</v-icon>

</v-btn>

</template>

<v-btn fab dark small color="indigo" @click="create()">

<v-icon>add</v-icon>

</v-btn>

<v-btn fab dark small color="red" @click="remove()">

<v-icon>delete</v-icon>

</v-btn>

</v-speed-dial>

</div>

</template>

<script lang="ts">

import { Component, Vue } from "vue-property-decorator";

import Items from "../components/Items.vue"; // @ is an alias to /src

import editItem from "../components/editItem.vue";

import { item } from "../model/item";

@Component({

components: {

Items,

editItem

}

})

export default class Home extends Vue {

todos: item[];

fab = false;

selected: any;

showAlert = false;

messageAlert = "";

showEdit = false;

constructor() {

super();

this.todos = [];

this.selected = null;

}

// computed

get selectedToEdit() {

if (this.selected) {

//return this.selected <- with the element selected is directly binded on dialog

return {

key: this.selected.key,

message: this.selected.message,

image: this.selected.image

};

} else

return {

key: -1,

message: "",

image: ""

};

}

create() {

this.showEdit = true;

}

remove() {

if (this.selected === null) {

this.messageAlert = "No item selected to remove";

this.showAlert = true;

} else {

const index = this.todos.indexOf(this.selected, 0);

console.log(index);

if (index > -1) {

this.todos.splice(index, 1);

}

}

}

select(itemClicked: item) {

if (this.selected === itemClicked) {

this.selected = null;

} else {

this.selected = itemClicked;

}

}

mounted() {

this.todos.push({

key: this.todos.length,

message: "Hello World",

image: ""

});

window.addEventListener("keyup", event => {

// If ESC key was pressed...

if (event.keyCode === 27) {

// try close your dialog

this.showEdit = false;

}

});

}

itemEdited(v: item) {

this.showEdit = false;

if (v.key == -1) {

v.key = this.todos.length;

this.todos.push(v);

} else {

const index = this.todos.indexOf(this.selected, 0);

this.todos[index].message = v.message;

this.todos[index].image = v.image;

}

}

}

</script>

<style>

.v-speed-dial {

position: absolute !important;

}

.selected {

background-color: red !important;

padding: 2px !important;

}

</style>

En la parte de la plantilla html encontramos:

Un componente personalizado Items:

<Items

v-for="todo in todos"

v-bind:key="todo.key"

v-bind:mykey="todo.key"

v-bind:message="todo.message"

v-bind:image="todo.image"

@click.native="select(todo)"

v-bind:class="selected===todo?'selected':''"

/>

El cual itera un array todos. Dado que existirán varios componentes items los diferenciamos por una clave. Y Por otra, parte se le pasan ciertos valores como Props (mykey,message e image). Se asocia el evento click (debemos incluir el modificador native para que haga efecto en un componente raíz) a un método que selecciona el item y se realiza un binding de clases en el cual se comprueba si una variable selected es igual al elemento en cuestión, en caso de afirmativo se aplica una clase selected.

Tenemos el componente de Vuetify:

<v-dialog v-model="showAlert" persistent max-width=“290”>…

Para mostrar mensajes de alerta como modales.

Otro componente personalizado:

<editItem v-bind:show="showEdit" @submit="itemEdited" v-bind:\_item="selectedToEdit" />

Que veremos más adelante, pero que tiene como Props dos variables (show e \_item). Además, tiene el evento submit asociado al método itemEdited.

Por último, cuenta con el componente de Vuetify:

<v-speed-dial

v-model="fab"

bottom

right

direction="top"

open-on-hover

transition="slide-y-reverse-transition"

>

…

Que es un botón para mostrar un menú de acciones.

En la sección de script, lo más destacado es:

La variable computed:

// computed

get selectedToEdit() {

if (this.selected) {

…

La empleamos para realizar una comprobación de si existe un elemento seleccionado y devolverlo, o en caso contrario devolver un elemento vacío.

Como se observa se emplea un modelo de datos asociado a los items, para ello tendremos la carpeta model y en su interior el archivo item.ts:

export interface item {

key: number;

message: string;

image?: string;

}

En cuanto a los componentes personalizados:

/src/components/editItem.vue

<template v-if="show">

<v-dialog v-model="show" persistent max-width="290">

<v-card>

<v-form ref="form" v-model="valid">

<v-text-field

v-model="\_item.message"

:counter="100"

:rules="messageRules"

label="Message"

required

></v-text-field>

<v-text-field v-model="\_item.image" :rules="urlRules" label="Url Image"></v-text-field>

<v-btn :disabled="!valid" color="success" class="mr-4" @click="save">Save</v-btn>

</v-form>

</v-card>

</v-dialog>

</template>

<script lang="ts">

import { Component, Vue, Prop } from "vue-property-decorator";

import { item } from "../model/item";

@Component

export default class editItem extends Vue {

@Prop({ default: false }) show!: boolean;

@Prop() \_item!: item;

valid = false;

messageRules = [

(v:string) => !!v || "Message is required",

(v:string) =>

(v && v.length >= 10 && v.length <= 100) ||

"Name must between 10 and 100 characters"

];

urlRules = [

(v:string) =>

!v ||

((v && /http[s]?\:\/\/.+/.test(v)) ||

"url must be valid: http:// or https://")

];

save() {

if (!this.\_item) {

this.\_item = {

key: -1,

message: "",

image: ""

};

}

this.$emit("submit", this.\_item);

}

}

</script>

<style>

form {

padding: 10px;

}

</style>

Es interesante el empleo de v-model para el tratamiento de formularios y cómo se ha realizado la validación. Otro aspecto a destacar es como se emplea la emisión de eventos para devolver datos al componente padre (en este caso Home.vue). Emitimos el evento submit y devolvemos el item creado o editado para que en Home se trate ese evento (@submit: …) y realice la operación adecuada de actualización del array todos.

Para el compente /src/components/Items.vue:

<template>

<div v-if="message && message!=''">

<v-card class="mx-auto">

<v-img v-if="image && image!=''"

class="white--text"

height="200px"

v-bind:src="image"

>

</v-img>

<v-card-text>{{message}}</v-card-text>

</v-card>

</div>

</template>

<script lang="ts">

import { Component, Vue, Prop } from 'vue-property-decorator';

@Component

export default class Items extends Vue {

@Prop({default: ''}) mykey!: number;

@Prop({default: ''}) image!: string;

@Prop({default: ''}) message!: string;

constructor(){

super();

// nothing to do here

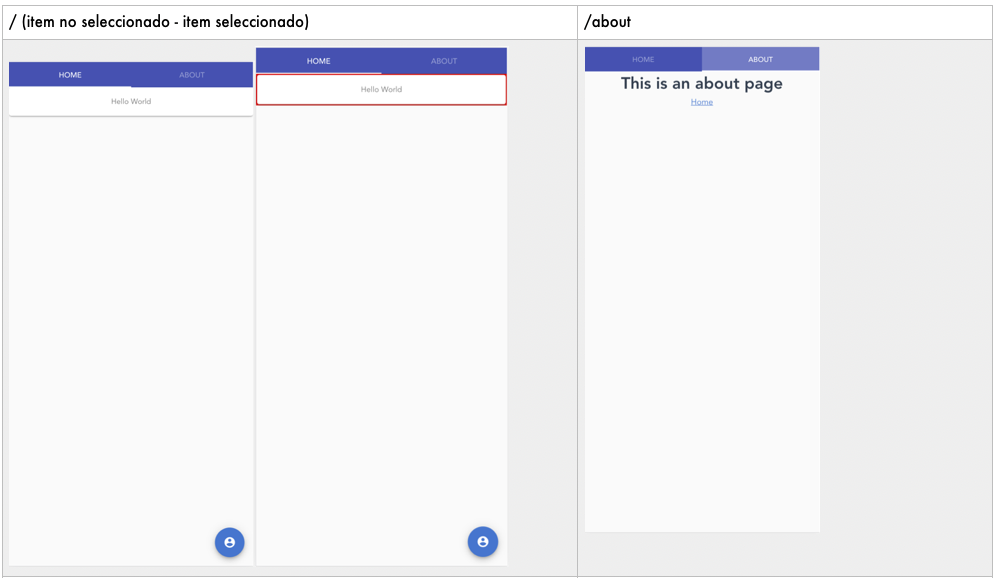
}

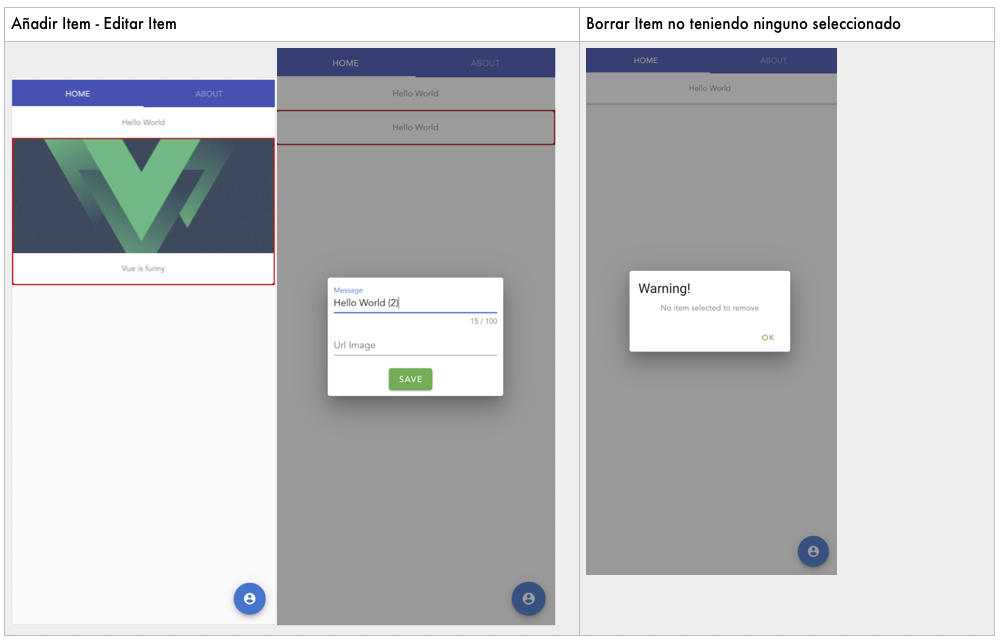
}

</script>

Simplemente es un componente donde se muestra una imagen (en caso necesario) y un mensaje. Este componente presenta la necesidad de significantes mejoras en caso de realizarlo en un proyecto real, como es la presentación de carga de la imagen.

Si realizamos una prueba (npm run serve) tendremos el siguiente resultado:





Como observamos la aplicación es totalmente funcional, pero no realiza lo más importante: no mantiene la lista de items cuando navegamos de una página a otra. Es decir, cada vez que visitamos Home.vue se construye la lista desde 0. Efectivamente, observando el código de Home.vue podemos observar:

constructor() {

super();

this.todos = [];

this.selected = null;

}

¿Y si quisiéramos mantener la lista tanto cuando estamos en Home como cuando navegamos a About?

Existen, sin hablar aún de persistencia de datos, varias soluciones.

Por ejemplo, podríamos declarar el array todos como global, solución funcional que no nos parece nada elegante. Empleando Typescript, de hecho, huimos de estos comportamientos poco aconsejados en la programación estructura y, posteriormente, orientada a objetos.

En este momento es cuando podemos echar en falta los servicios de Angular. Podríamos crear nuestra clase Singleton e incluirla en todos nuestros componentes que la requieran. Veamos una aproximación:

/src/model/Store.ts

import { item } from './item';

export default class Store {

private static instance: Store;

public todos:item[];

public static getInstance(): Store {

if (!Store.instance) {

Store.instance = new Store();

}

return Store.instance;

}

private constructor() {

this.todos=[];

}

}

Ahora solo queda en Home.vue cambiar la referencia de todos a la variable todos del singleton (myStorage):

<template>

<div class="home">

<Items

v-for="todo in myStore.todos"

v-bind:key="todo.key"

v-bind:mykey="todo.key"

v-bind:message="todo.message"

v-bind:image="todo.image"

@click.native="select(todo)"

v-bind:class="selected===todo?'selected':''"

/>

<!-- click.native for click event on root component element -->

<v-dialog v-model="showAlert" persistent max-width="290">

<v-card>

<v-card-title class="headline">Warning!</v-card-title>

<v-card-text>{{messageAlert}}</v-card-text>

<v-card-actions>

<v-spacer></v-spacer>

<v-btn color="green darken-1" text @click="showAlert = false">Ok</v-btn>

</v-card-actions>

</v-card>

</v-dialog>

<editItem v-bind:show="showEdit" @submit="itemEdited" v-bind:\_item="selectedToEdit" />

<v-speed-dial

v-model="fab"

bottom

right

direction="top"

open-on-hover

transition="slide-y-reverse-transition"

>

<template v-slot:activator>

<v-btn v-model="fab" color="blue darken-2" dark fab>

<v-icon v-if="fab">close</v-icon>

<v-icon v-else>account\_circle</v-icon>

</v-btn>

</template>

<v-btn fab dark small color="indigo" @click="create()">

<v-icon>add</v-icon>

</v-btn>

<v-btn fab dark small color="red" @click="remove()">

<v-icon>delete</v-icon>

</v-btn>

</v-speed-dial>

</div>

</template>

<script lang="ts">

import { Component, Vue } from "vue-property-decorator";

import Items from "../components/Items.vue"; // @ is an alias to /src

import editItem from "../components/editItem.vue";

import { item } from "../model/item";

import Store from '../model/Store';

@Component({

components: {

Items,

editItem

}

})

export default class Home extends Vue {

myStore:Store;

fab = false;

selected: any;

showAlert = false;

messageAlert = "";

showEdit = false;

constructor() {

super();

this.myStore=Store.getInstance();

this.selected = null;

}

// computed

get selectedToEdit() {

if (this.selected) {

//return this.selected <- with the element selected is directly binded on dialog

return {

key: this.selected.key,

message: this.selected.message,

image: this.selected.image

};

} else

return {

key: -1,

message: "",

image: ""

};

}

create() {

this.showEdit = true;

}

remove() {

if (this.selected === null) {

this.messageAlert = "No item selected to remove";

this.showAlert = true;

} else {

const index = this.myStore.todos.indexOf(this.selected, 0);

console.log(index);

if (index > -1) {

this.myStore.todos.splice(index, 1);

}

}

}

select(itemClicked: item) {

if (this.selected === itemClicked) {

this.selected = null;

} else {

this.selected = itemClicked;

}

}

mounted() {

window.addEventListener("keyup", event => {

// If ESC key was pressed...

if (event.keyCode === 27) {

// try close your dialog

this.showEdit = false;

}

});

}

itemEdited(v: item) {

this.showEdit = false;

if (v.key == -1) {

v.key = this.myStore.todos.length;

this.myStore.todos.push(v);

} else {

const index = this.myStore.todos.indexOf(this.selected, 0);

this.myStore.todos[index].message = v.message;

this.myStore.todos[index].image = v.image;

}

}

}

</script>

<style>

.v-speed-dial {

position: absolute !important;

}

.selected {

background-color: red !important;

padding: 2px !important;

}

</style>

De esta forma ya tendríamos nuestra aplicación con un pequeño control de estados y manteniendo la lista a pesar de navegar y salir de nuestro componente Home.

Sin embargo, en Vue ya existen mecanismos más avanzados para el control de estado de nuestra aplicación.

### Vuex

Para hablar de Vuex, bien podríamos empeazar hablando de Flux. Un framework para el control de estado de la aplicación que gestiona cuándo se realizan los cambios, los mantiene y determina si se debe renderizar y qué partes del DOM, para ello es clave el concepto de DOM virtual. Es decir, un concepto de estado muy relacionado con React, de hecho, diseñado por Facebook.

Posteriormente han aparecido otras implementaciones, con ciertos cambios de esta filosofía, como son Redux o, para Vue: Vuex.

Cuando creamos el proyecto seleccionamos compatibilidad con Vuex y el cliente nos creó un archivo /store.ts:

import Vue from 'vue';

import Vuex from 'vuex';

Vue.use(Vuex);

export default new Vuex.Store({

state: {

},

mutations: {

},

actions: {

},

});

Y en el archivo main.ts podemos ver cómo lo importa a la hora de crear la App:

new Vue({

router,

store,

render: (h) => h(App),

vuetify: new Vuetify(vuetifyOptions),

}).$mount('#app');

¿Cómo funciona Vuex?

El objeto state son, tal como se vio en React, los datos de los cuales queremos tener control de estado. Sería equivalente a la función data() de nuestros componentes.

state: {

todos: Array<item>()

},

Para hacer referencia a una de las variables debemos emplear:

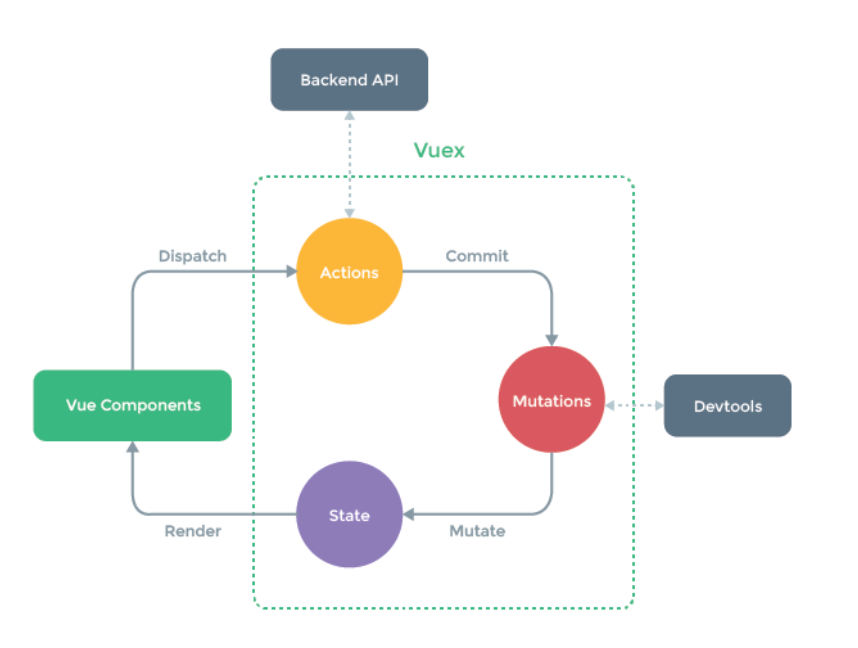
this.$store.state.todos.length;

Las mutaciones son los mecanismos que nos permiten cambiar el valor de las variables del estado. Nunca se deben modificar directamente, dado que las mutaciones permiten ir registrando los cambios. La ejecución de las mutaciones es sincronía y para ejecutarlas debemos hacerlo mediante la sintaxis:

this.$store.commit(“nombre\_de\_mutación” [,{valor\_de\_paso}] );

En cambio, las actions pueden ser asíncronas, es decir devolver un Promise y no deberían acceder directamente al estadio sino emplear mutations.

this.$store.dispatch(“nombre\_action” [, {valores\_de\_paso}).then …



En nuestro caso, no debemos realizar peticiones a APIs remotas, por lo que emplearemos las mutaciones directamente, aunque definimos una action para ver cómo se realizaría.

/store.ts

import Vue from 'vue';

import Vuex from 'vuex';

import { item } from './model/item';

Vue.use(Vuex);

export default new Vuex.Store({

state: {

todos: Array<item>()

},

mutations: {

addItem(context,n:item){

context.todos.push(n);

},

editItem(context, n) {

const index = context.todos.indexOf(n.selected, 0);

if ( index >= 0 ) {

context.todos[index].message = n.item.message;

context.todos[index].image = n.item.image;

}

},

removeItem(context, n ) {

const index = context.todos.indexOf(n, 0);

if (index > -1) {

context.todos.splice(index, 1);

}

}

},

actions: {

addItemAsync(context, n:item){

return new Promise((resolve,reject)=>{

setTimeout(()=>{

context.commit('addItem', n);

resolve();

},2000)

})

}

},

});

Ahora modificamos Home.vue para que en vez de emplear el singleton que programamos, utilice Vuex:

<template>

<div class="home">

<Items

v-for="todo in ToDos"

v-bind:key="todo.key"

v-bind:mykey="todo.key"

v-bind:message="todo.message"

v-bind:image="todo.image"

@click.native="select(todo)"

v-bind:class="selected===todo?'selected':''"

/>

<!-- click.native for click event on root component element -->

<v-dialog v-model="showAlert" persistent max-width="290">

<v-card>

<v-card-title class="headline">Warning!</v-card-title>

<v-card-text>{{messageAlert}}</v-card-text>

<v-card-actions>

<v-spacer></v-spacer>

<v-btn color="green darken-1" text @click="showAlert = false">Ok</v-btn>

</v-card-actions>

</v-card>

</v-dialog>

<editItem v-bind:show="showEdit" @submit="itemEdited" v-bind:\_item="selectedToEdit" />

<v-speed-dial

v-model="fab"

bottom

right

direction="top"

open-on-hover

transition="slide-y-reverse-transition"

>

<template v-slot:activator>

<v-btn v-model="fab" color="blue darken-2" dark fab>

<v-icon v-if="fab">close</v-icon>

<v-icon v-else>account\_circle</v-icon>

</v-btn>

</template>

<v-btn fab dark small color="indigo" @click="create()">

<v-icon>add</v-icon>

</v-btn>

<v-btn fab dark small color="red" @click="remove()">

<v-icon>delete</v-icon>

</v-btn>

</v-speed-dial>

</div>

</template>

<script lang="ts">

import { Component, Vue } from "vue-property-decorator";

import Items from "../components/Items.vue"; // @ is an alias to /src

import editItem from "../components/editItem.vue";

import { item } from "../model/item";

import store from "../store"; //<-- VUEX

@Component({

components: {

Items,

editItem

}

})

export default class Home extends Vue {

fab = false;

selected: any;

showAlert = false;

messageAlert = "";

showEdit = false;

constructor() {

super();

this.selected = null;

}

// computed

get selectedToEdit() {

if (this.selected) {

//return this.selected <- with the element selected is directly binded on dialog

return {

key: this.selected.key,

message: this.selected.message,

image: this.selected.image

};

} else

return {

key: -1,

message: "",

image: ""

};

}

create() {

this.showEdit = true;

}

remove() {

if (this.selected === null) {

this.messageAlert = "No item selected to remove";

this.showAlert = true;

} else {

this.$store.commit("removeItem",this.selected);

}

}

select(itemClicked: item) {

if (this.selected === itemClicked) {

this.selected = null;

} else {

this.selected = itemClicked;

}

}

mounted() {  this.selected=null;

window.addEventListener("keyup", event => {

// If ESC key was pressed...

if (event.keyCode === 27) {

// try close your dialog

this.showEdit = false;

}

});

}

itemEdited(v: item) {

this.showEdit = false;

if (v.key == -1) {

v.key = this.ToDos.length;

/\*\* Using Vuex \*/

this.$store.commit("addItem",v);

} else {

/\*\* Using Vuex \*/

this.$store.commit("editItem",{selected:this.selected,item:v});

}

}

//Getter for Vuex

get ToDos():item[]{

return this.$store.state.todos;

}

}

</script>

<style>

.v-speed-dial {

position: absolute !important;

}

.selected {

background-color: red !important;

padding: 2px !important;

}

</style>

No solo tenemos controlado de mejor forma el estado, sino que el código queda más ordenado y limpio.

Para terminar nuestra aplicación vamos a instalar una librería que os permitirá de forma automática hacer persistente nuestro state de Vuex, almacenándolo en localstorage.

npm install vuex-persist

Y modificamos el archivo /store.ts:

import Vue from 'vue';

import Vuex from 'vuex';

import VuexPersist from 'vuex-persist';

import { item } from './model/item';

const vuexPersist = new VuexPersist({

key: 'my-app', // key used in localstorage

storage: localStorage //Change localStorage with sessionStorage to use that other storage system

})

Vue.use(Vuex);

export default new Vuex.Store({

state: {

todos: Array<item>()

},

mutations: {

addItem(context:any,n:item){

context.todos.push(n);

},

editItem(context:any, n) {

const index = context.todos.indexOf(n.selected, 0);

if ( index >= 0 ) {

context.todos[index].message = n.item.message;

context.todos[index].image = n.item.image;

}

},

removeItem(context:any, n:item ) {

const index = context.todos.indexOf(n, 0);

if (index > -1) {

context.todos.splice(index, 1);

}

}

},

actions: {

addItemAsync(context:any, n:item){

return new Promise((resolve,reject)=>{

setTimeout(()=>{

context.commit('addItem', n);

resolve();

},2000)

})

}

},

plugins: [vuexPersist.plugin] // <- IMPORTANT

});

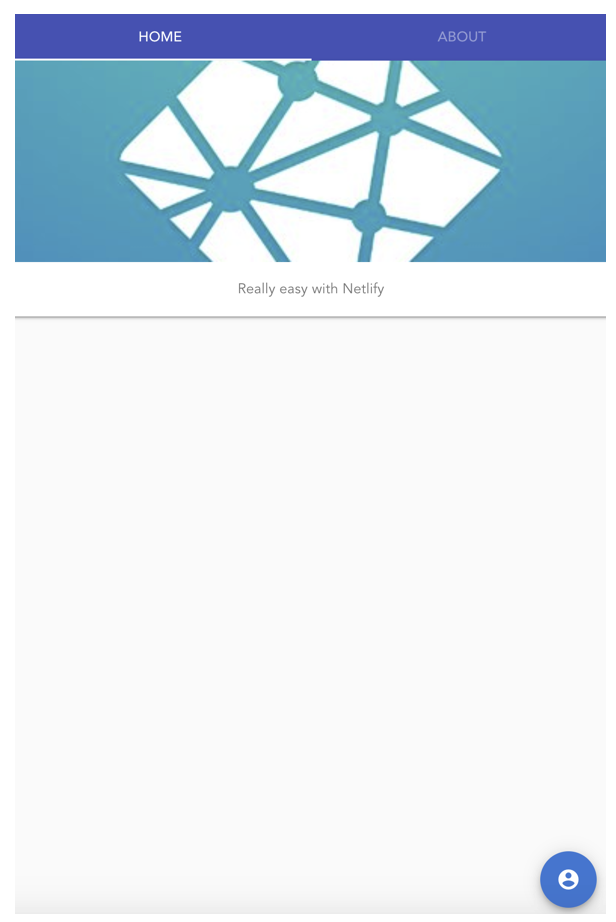
De esta forma nuestra aplicación mantendrá en local nuestra propia lista de items, a pesar de que cerremos el navegador.

### Despliegue

Para esta pequeña aplicación emplearemos el servicio de despliegue Netfily (<https://www.netlify.com/> ). Tras crear nuestra cuenta nos aparece directamente el menú para conectar con Git o directamente arrastrar nuestra carpeta. Antes de nada, generaremos nuestra aplicación de producción:

ng serve build

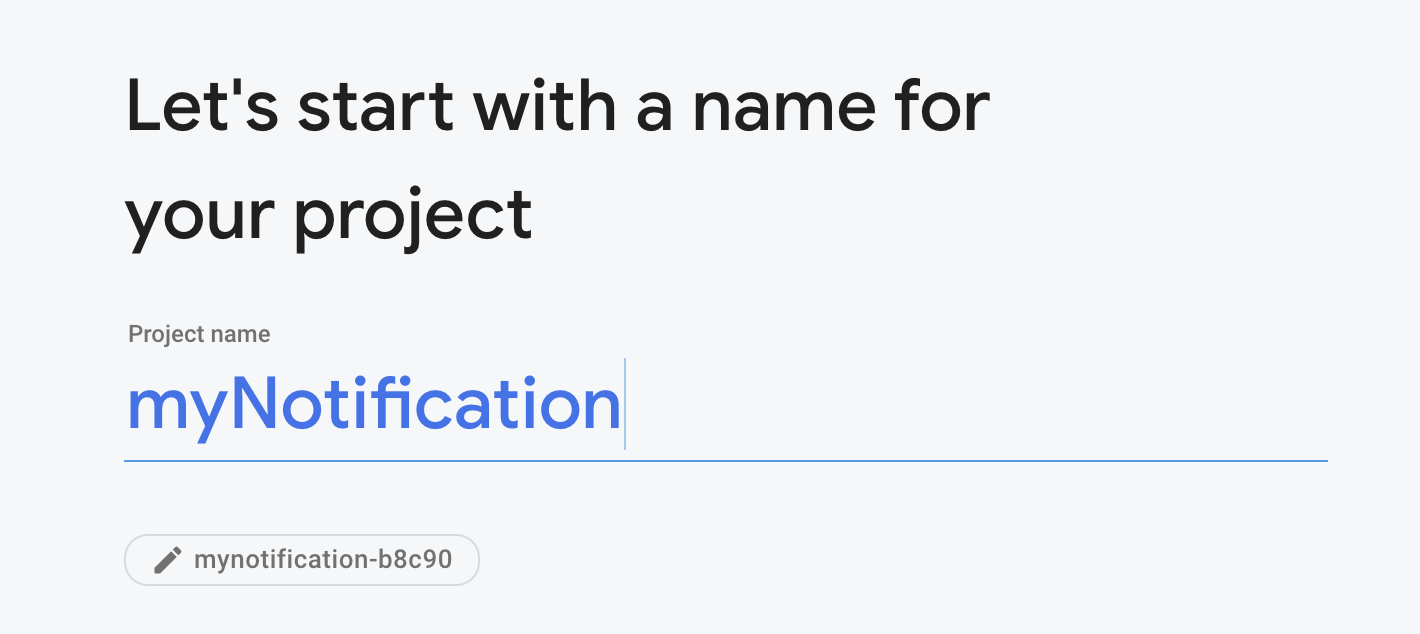
Tenemos la carpeta /dist lista para desplegar. Arrastramos la carpeta a la web de netfily, y listo: <https://flamboyant-turing-6dd8ec.netlify.com/>



### Push Notification

Vamos a añadir una funcionalidad bastante útil en aplicaciones en producción: las notificaciones. Para ello empezaremos el servicio que ofrece Firebase.

Accedemos a la consola de firebase y creamos un proyecto nuevo (myNotificator) y registramos una aplicación web:



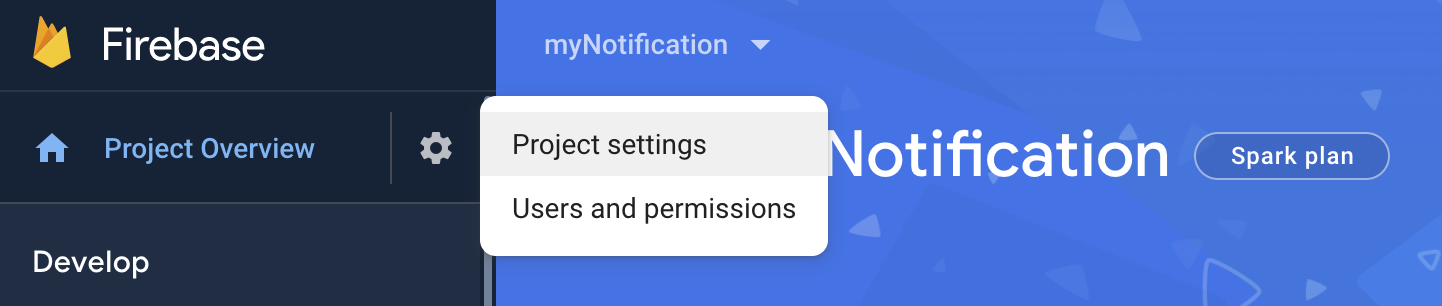
Instalamos la librería de firebase

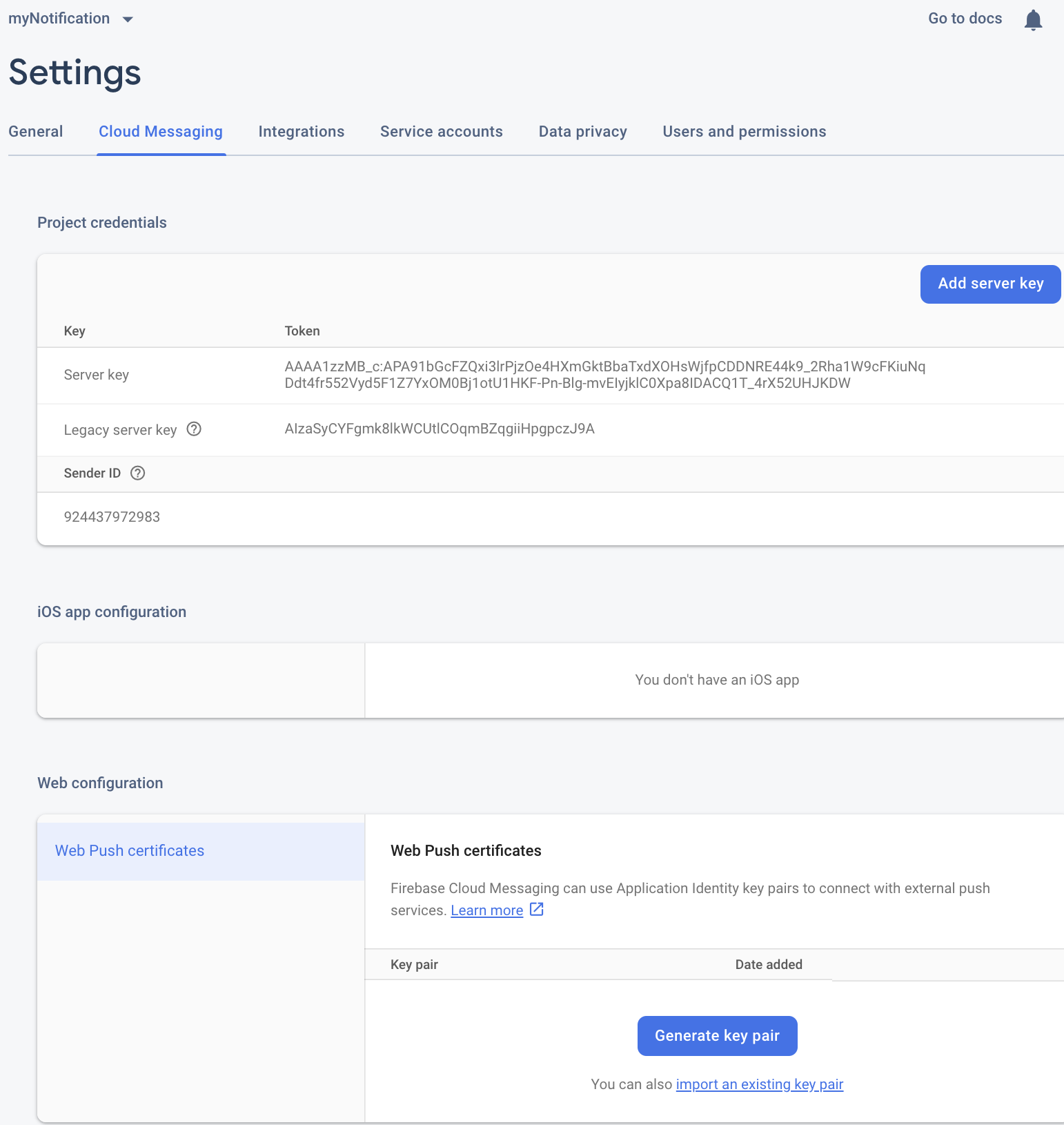
npm install firebase --save

Y las herramientas de despliegue (si deseamos desplegarlo en firebase):

npm install -g firebase-tools

Ahora en la consola de Firebase, en la configuración del proyecto vamos a crear un par de claves Web Push certificates.





Copiamos el Sender ID en el archivo /public/manitest.json:

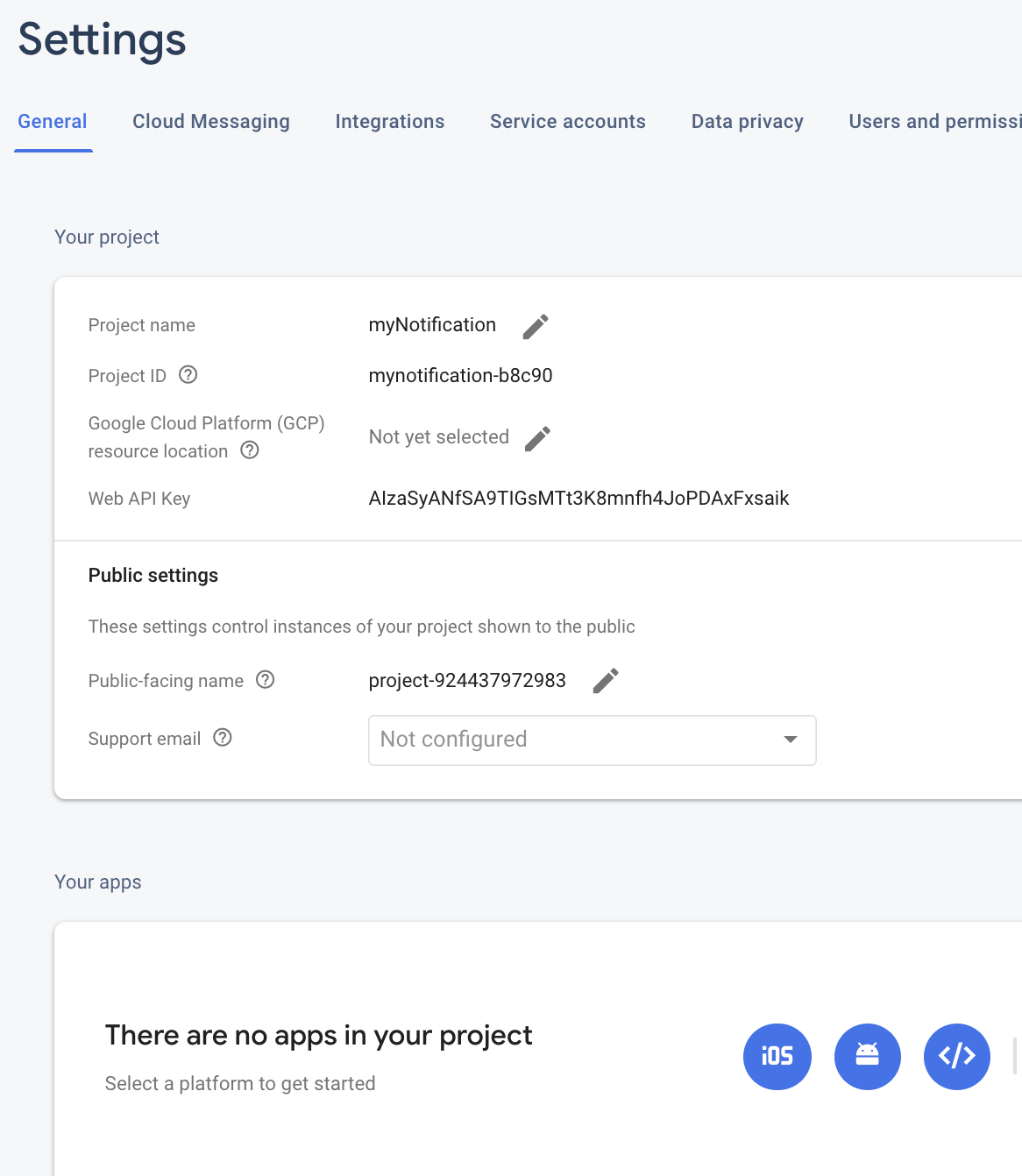
./public/manitest.json

{

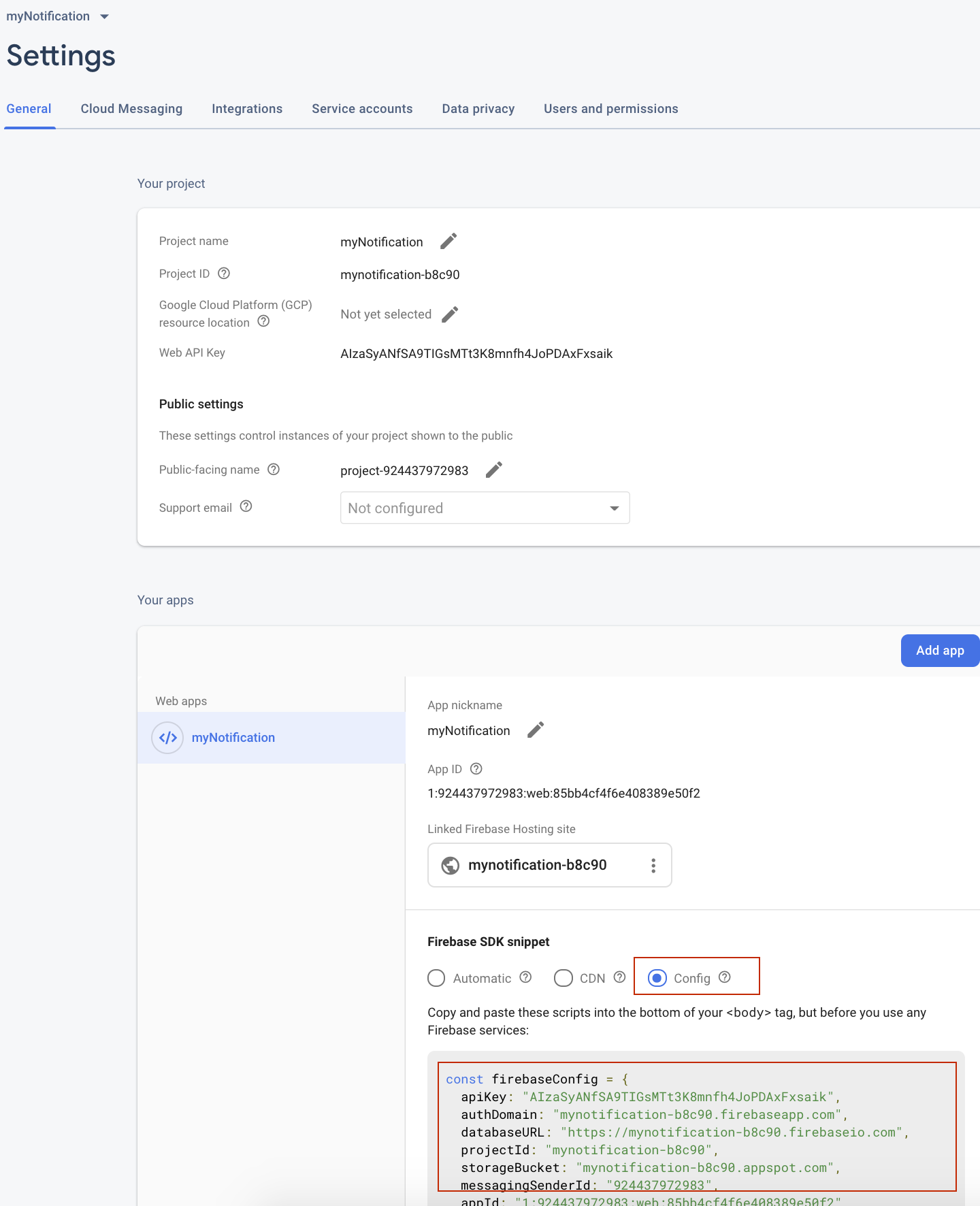
...,

"gcm\_sender\_id": "924437972983"

}



Tras darle nombre (myNotification), avanzamos los pasos hasta volver a la consola. En la configuración de nuestro proyecto copiamos las credenciales:



Creamos un archivo en la carpeta src llamado config.js. Modificamos el archivo config.js para que contenga el siguiente código:

//This file is used as config in main.ts and firebase.messaging-sw.js

const config = {

apiKey: "AIzaSyANfSA9TIGsMTt3K8mnfh4JoPDAxFxsaik",

authDomain: "mynotification-b8c90.firebaseapp.com",

databaseURL: "https://mynotification-b8c90.firebaseio.com",

projectId: "mynotification-b8c90",

storageBucket: "mynotification-b8c90.appspot.com",

messagingSenderId: "924437972983",

appId: "1:924437972983:web:85bb4cf4f6e408389e50f2",

serverKey:

"AAAA1zzMB\_c:APA91bGcFZQxi3lrPjzOe4HXmGktBbaTxdXOHsWjfpCDDNRE44k9\_2Rha1W9cFKiuNqDdt4fr552Vyd5F1Z7YxOM0Bj1otU1HKF-Pn-Blg-mvEIyjklC0Xpa8IDACQ1T\_4rX52UHJKDW",

pairKey:

"BDMwTNRSRGOwt2yLC0ClKVgaawHK3KxAwUJo-hm04vzFKtHm8loSnR\_f3D1TlH-HQH8Un9vv-SxTjrA1pIbLEdg"

};

try { //exports for importing into main.ts.

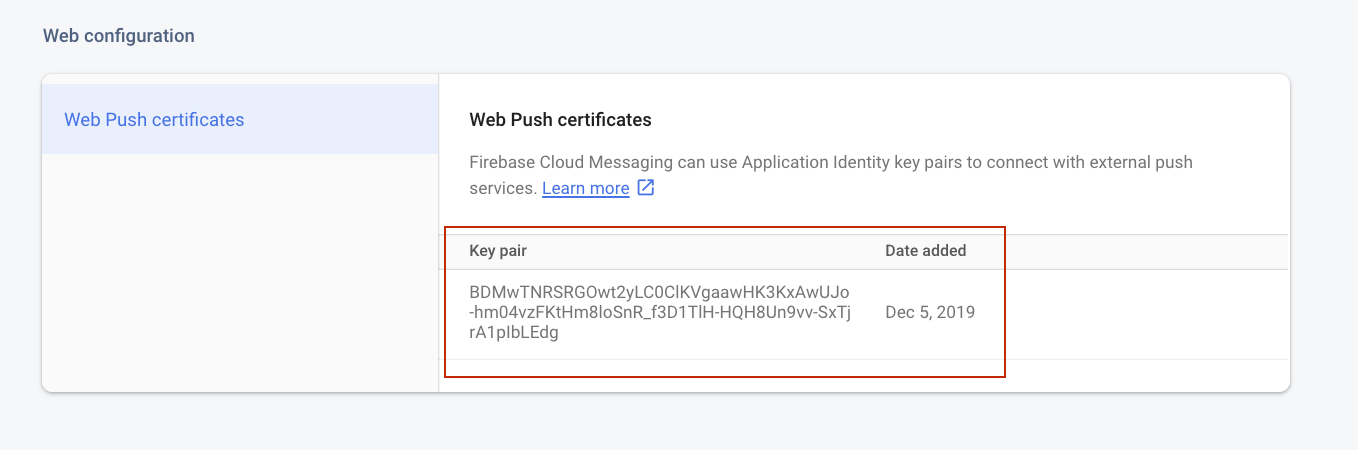
//Becareful, export no defined in importScript in firebase.messaging-sw.js

if ("undefined" != typeof module) module.exports = config;

} catch (err) {}

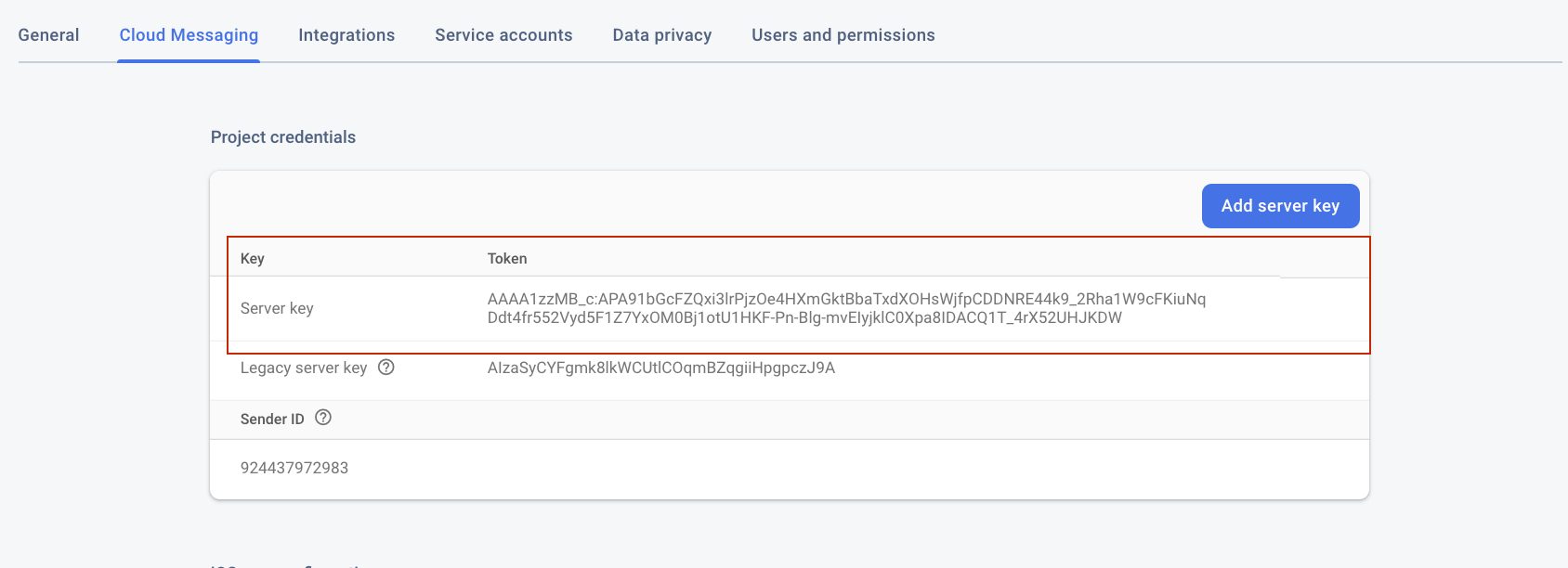
En este mismo archivo, config.js, debemos cambiar dos valores fundamentales.

Para ello, primero debemos ir a la consola de firebase. En la pestaña de Cloud Mesagging copiamos la clave pública del Web Push certificates:



Este valor lo fijamos a la propiedad pairKey.

Desde la misma ventana de la consola de firebase copiamos la server key:



Y, de nuevo, actualizamos su valor en la variable oportuna de config.js:

En esta versión de la aplicación el archivo main.ts contiene el siguiente código:

import Vue from 'vue';

import App from './App.vue';

import router from './router';

import store from './store';

import './registerServiceWorker';

// Vuetify

import Vuetify from 'vuetify';

import 'vuetify/dist/vuetify.min.css';

// Custom Stencil Component

import { applyPolyfills, defineCustomElements } from 'histencil/loader';

/\*\* VUE NOTIFICATION \*/

import \* as firebase from 'firebase';

import axios from 'axios';

import \* as config from './config';

firebase.initializeApp(config);

try {

const messaging = firebase.messaging();

Vue.prototype.$messaging = firebase.messaging();

// tslint:disable-next-line: max-line-length

messaging.usePublicVapidKey(

config.pairKey

); // 1. Generate a new key pair

messaging

.requestPermission()

.then(() => firebase.messaging().getToken())

.then((token:any) => {

axios

.post(

'https://iid.googleapis.com/iid/v1/' + token + '/rel/topics/all',

{},

{

headers: {

// key-server from console

Authorization:

'key='+config.serverKey

},

},

)

.then((r: any) => {

console.log('Subscribed to all');

new Vue({

router,

store,

render: (h) => h(App),

vuetify: new Vuetify(vuetifyOptions),

}).$mount('#app');

})

.catch((err: any) => {

console.log(err);

});

})

.catch(function(err:any) {

console.log('Unable to get permission to notify.', err);

});

} catch (eer) {

// Error with API and browser version

console.log(eer);

}

Vue.config.productionTip = false;

const vuetifyOptions = {};

Vue.use(Vuetify);

// Binding stencil component to window

applyPolyfills().then(() => {

defineCustomElements(window);

});

Además de solicitar permisos para mostrar notificaciones, subscribimos a la aplicación al topic all. Los mensajes pueden enviarse a un dispositivo, a un grupo o, como en este caso, a los usuarios que estén subscritos a determinados temas. Para subscribirnos a un tema debemos contar con el token que lo extraemos de la solicitud getToken() y la clave de servidor que se consigue en la consola de firebase. Algo más abajo aparece una captura del lugar exacto.

Para que nuestra aplicación pueda recibir el evento de recepción en segundo plano debemos crear un service worker, en nuestro caso en ./public/firebase-messaging-sw.js:

//./public/firebase-messaging-sw.js

// [START initialize\_firebase\_in\_sw]

// Give the service worker access to Firebase Messaging.

// Note that you can only use Firebase Messaging here, other Firebase libraries

// are not available in the service worker.

importScripts('https://www.gstatic.com/firebasejs/7.5.2/firebase-app.js');

importScripts('https://www.gstatic.com/firebasejs/7.5.2/firebase-messaging.js');

importScripts('config.js');

// Initialize the Firebase app in the service worker by passing in the messagingSenderId.

firebase.initializeApp(config);

// Retrieve an instance of Firebase Messaging so that it can handle background messages.

const messaging = firebase.messaging();

// [END initialize\_firebase\_in\_sw]

/\*messaging.setBackgroundMessageHandler((payload)=>{

console.log('[firebase-messaging-sw.js] Received background message ');

console.log(payload)

// Customize notification here

var notificationTitle = 'Background Message Title';

var notificationOptions = {

body: 'Background Message body.',

icon: '/firebase-logo.png'

};

return self.registration.showNotification(notificationTitle,

notificationOptions);

});\*/

*Nota: es importante que la versión que se importa de firebase coincida con la versión que existe en el proyecto: ver package.json. En nuestro caso 7.5.2*

Como es el único service worker que tenemos en nuestra aplicación lo vamos a registrar modificando el archivo /src/registerServiceWorker.ts:

if (process.env.NODE\_ENV === 'production') {

register(`${process.env.BASE\_URL}firebase-messaging-sw.js`, {

ready() {

console.log(

…

Ya controlamos los mensajes en segundo plano, ahora vamos a gestionar los recibidos en primer plano (cuando la aplicación está abierta). Para ello modificamos App.vue para mostrar mensajes cuando se reciba el evento:

<template>

<div id="app">

<v-app>

<v-content>

<TabBar />

<router-view />

<v-snackbar v-model="snackbar" bottom color="success" vertical left :timeout="timeout">

<h3>{{title}}</h3>

<p>{{message}}</p>

<v-avatar color="primary" size="50">

<img v-if="img" :src="img" alt="avatar" />

<span v-else>VJ</span>

</v-avatar>

<v-btn dark text @click="snackbar = false">Close</v-btn>

</v-snackbar>

</v-content>

</v-app>

</div>

</template>

<script lang="ts">

import { Vue, Component } from "vue-property-decorator";

import TabBar from "./components/TabBar.vue";

import \* as firebase from "firebase";

@Component({

components: {

TabBar

}

})

export default class App extends Vue {

snackbar = false;

title = null;

message = null;

img = null;

timeout = 5000;

created() {

try{

firebase.messaging().onMessage(payload => {

console.log("Message received On first plane. ");

this.showMessage(

payload.notification.title,

payload.notification.body,

payload.notification.icon

);

});

}catch(err){

console.log(err)

}

}

showMessage(title:any, message:any, img:any) {

this.title = title;

this.message = message;

this.img = img;

this.snackbar = true;

}

}

</script>

<style lang="scss">

#app {

font-family: "Avenir", Helvetica, Arial, sans-serif;

-webkit-font-smoothing: antialiased;

-moz-osx-font-smoothing: grayscale;

text-align: center;

color: #2c3e50;

}

</style>

Cosas importantes:

Cada vez que ejecutamos npm run-script build generamos la carpeta dist con la aplicación compilada y copiamos el contenido que haya en la carpeta public. Por ello, es importante repasar su contenido: iconos, serviceworker, manifiesto…

Si se observa el archivo package.json, el comando build está asociado :

vue-cli-service build && cp -rv src/config.js dist/config.js

Es decir, compilado y posterior copia del archivo config.js a la carpeta raíz de la aplicación para que el archivo del serviceworker (firebase-messaging-sw.js) tenga acceso a él. Habría que diseñar un sistema que proteja este archivo, dado que contiene información sensible.

Será la carpeta dist la que emplearemos para su despliegue.