UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

Departament de Sistemes Informàtics i Computació

Universitat Politècnica de València

Trabajo Fin de Máster

INTERFAZ PARA LA EJECUCIÓN DE APLICACIONES SOBRE PLATAFORMAS SERVERLESS

Autor: Jordi Peiró Castelló

Tutor: Germán Moltó Martínez

2017-2018

RESUMEN

El auge de las plataformas de tipo Serverless computing, donde el usuario no aprovisiona, gestiona ni escala la infraestructura, está siendo adoptado en numerosos ámbitos. Servicios como AWS Lambda posibilitan la ejecución de funciones en respuesta a eventos codificadas en múltiples lenguajes de programación. En esta línea, SCAR (https://github.com/grycap/scar) permite la definición de funciones en AWS Lambda que ejecuten contenedores a partir de imágenes Docker almacenadas en Docker Hub. Esto permite abrir el campo de serverless computing a la computación científica, que involucra a menudo aplicaciones complejas con múltiples dependencias de librerías externas.

Este TFM plantea la creación de una interfaz de usuario basada en web que permita interactuar con un sistema que ofrece una funcionalidad similar a SCAR pero para entornos on-premises (denominado OSCAR). Este permite la definición y ejecución de funciones orientadas a eventos para el procesamiento de datos basados en ficheros, así como el acceso a resultados de un sistema de almacenamiento orientado a bloques (como Amazon S3). Para ello, será necesario crear una interfaz web que interactué con los API REST de los diferentes servicios de OSCAR para facilitar la interacción del usuario.

Palabras clave: Serverless, SCAR, OpenFaaS, Minio, Vue.js, AWS, AWS S3, AWS Lambda

AGRADECIMIENTOS

Quiero expresar mi más sincero agradecimiento a mi tutor Germán Moltó Martínez por el apoyo prestado, sus buenos consejos y su dedicación a lo largo del desarrollo de todo el proyecto.

A mis padres, mi hermana y mis abuelos por el soporte y apoyo incondicional durante todo este tiempo, tanto en los momentos buenos como en los no tan buenos.

A ti Raquel, por entender la ilusión y el esfuerzo dedicado durante esta etapa de mi vida, la cual hemos compartido y sin tu cariño hubiese sido más difícil de conseguir.

TABLA DE CONTENIDO

TABLA DE ILUSTRACIONES

1. INTRODUCCIÓN

1.1. OBJETIVOS

El objetivo principal de este proyecto es facilitar la interacción de los científicos e investigadores con las tecnologías Serverless. Para ello, se desarrollará una interfaz web que permita desplegar funciones, crear y eliminar buckets y crear y eliminar objetos en cada uno de los buckets.

Las funciones configuradas deben de ejecutarse automáticamente a partir de los eventos generador por la creación de objetos en un determinado bucket de entrada, es decir, cuando el usuario cree un objeto en el bucket de entrada, se desencadenará un evento que llamará a una función. Esta función procesará el objeto y almacenará el resultado en un bucket de salida, al que se tendrá acceso desde la misma interfaz. Por tanto, desde una misma aplicación se podrán crear objetos, que automáticamente serán procesados y visualizar los resultados sin tener que aprovisionar ni configurar ningún tipo de hardware.

1.2. MOTIVACIÓN

El grupo de investigación “Grid y Computación de Altas Prestaciones” de la Universitat Politècnica de València, en adelante GRyCAP, ha desarrollado una herramienta llamada “Serverless Computing for container-based Architectures”[1] (SCAR) la cual permite desplegar de forma transparente contenedores de imágenes Docker en AWS Lambda. Esto permite ejecutar aplicaciones desarrolladas en prácticamente cualquier lenguaje de programación sobre AWS Lambda. En la actualidad en el grupo GRyCAP están desarrollando la herramienta “On-premises Serverless Container-aware ARchitectures”[2] (OSCAR), la cual desplegará un conjunto de servicios sobre un clúster Kubernetes para imitar el modelo de programación ofrecido por SCAR, permitiendo así, ejecutar aplicaciones empaquetadas en contenedores Docker a traves de funciones. OSCAR necesita un interfaz de usuario que permita a los usuarios que estén trabajando sobre él desplegar funciones y acceder a los resultados generados por dichas funciones de forma simple y clara, por tanto, se planteó desarrollar una frontend web que cumpliera con estos requerimientos. Este trabajo se enmarca en una colaboración entre el GRyCAP y EGI [3] para ofrecer soporte de computación serverless en dicha plataforma. EGI es una plataforma de computación federada a lo largo principalmente de Europa que proporciona servicios de computación avanzados para la investigación y la innovación. Por tanto, es de esperar que este desarrollo, liberado como código abierto, sea eventualmente utilizado a nivel Europeo.

2. ESTADO DEL ARTE

La parte computacional del proyecto se realiza mediante funciones FaaS (Functions as a Service), las cuales permiten ejecutar aplicaciones empaquetadas en contenedores Docker sin necesidad de aprovisionar previamente ningún tipo de hardware por parte del usuario. Existe una gran variedad de herramientas que permiten la creación y ejecución de dichas funciones. Por ello es necesario estudiar las alternativas que mas aceptación han tenido por parte de la comunidad. A continuación, se muestra el número de estrellas obtenidas en GitHub en el momento en que se redacta este documento.

Nuclio [4] ★ Star 2,233

Fnproject [5] ★ Star 3,317

Kubeless [6] ★ Star 3,418

Fission [7] ★ Star 3,585

OpenFaaS [8] ★ Star 11,450

El proyecto **nuclio** promete ser extremadamente rápido, llegando a procesar cientos de miles de peticiones http o registros de datos por segundo, esto es debido principalmente al procesamiento en tiempo real y máximo grado de paralelismo. En la web del proyecto señalan como punto diferencial frente a otros proyectos la capacidad para depurar funciones de forma simple, realizar pruebas de regresión y pipelines de CI/CD de varias versiones, siendo esto un factor importante a la hora de mantener el código fuente de las funciones.

Fnproject destaca por la facilidad de uso y la capacidad de importar funciones Lambda de AWS, lo cual es muy importante en caso de que el proyecto que se va a migrar disponga de todas las funciones implementadas en dicho servicio.

Según la comparativa que aparece en la página web del proyecto, **kubeless** es el servicio que más nativo es sobre Kubernetes ya que utiliza directivas k8s directamente. Por esto, no dispone de un servidor adicional de API o de un Gateway adicional.

Fission destaca por su inicio frio en 100 milisegundos. Esto es posible porque mantiene un grupo de contenedores "cálidos" que contienen un pequeño cargador dinámico. Cuando se llama por primera vez a una función, es decir, "iniciada en frío", se elige un contenedor en ejecución y se carga la función. Este conjunto es lo que hace que fission sea tan rápido.

Por último, **OpenFaaS** que es la herramienta elegida para el desarrollo del trabajo, es un proyecto independiente liderado por Alex Ellis y cuenta con una creciente comunidad de desarrolladores ofreciendo soporte. Es importante destacar que tanto OpenFaaS como el resto de las alternativas ofrecen un soporte amplio de lenguajes compatibles,

pero en especial OpenFaaS ya que permite empaquetar cualquier proceso como una función serverless.

3. ENTORNO TECNOLÓGICO

En este apartado se describen todos los componentes que conforman este trabajo. El proyecto se basa principalmente en el desarrollo de una interfaz web para poder desplegar de forma sencilla aplicaciones empaquetadas en contenedores Docker y al mismo tiempo obtener los resultados de las ejecuciones en esta misma aplicación. Para conseguir esta funcionalidad es necesario desplegar un backend que se encargue de gestionar todos los recursos.

Para el despliegue del backend se utiliza Kubernetes [9] como el orquestador de contenedores sobre el que se despliegan OpenFaaS [10] y Minio [11]. La gestión de eventos se realiza mediante la herramienta Serverless Event Gateway [12]. A continuación, se muestra un diagrama y se explica más detalladamente el funcionamiento:

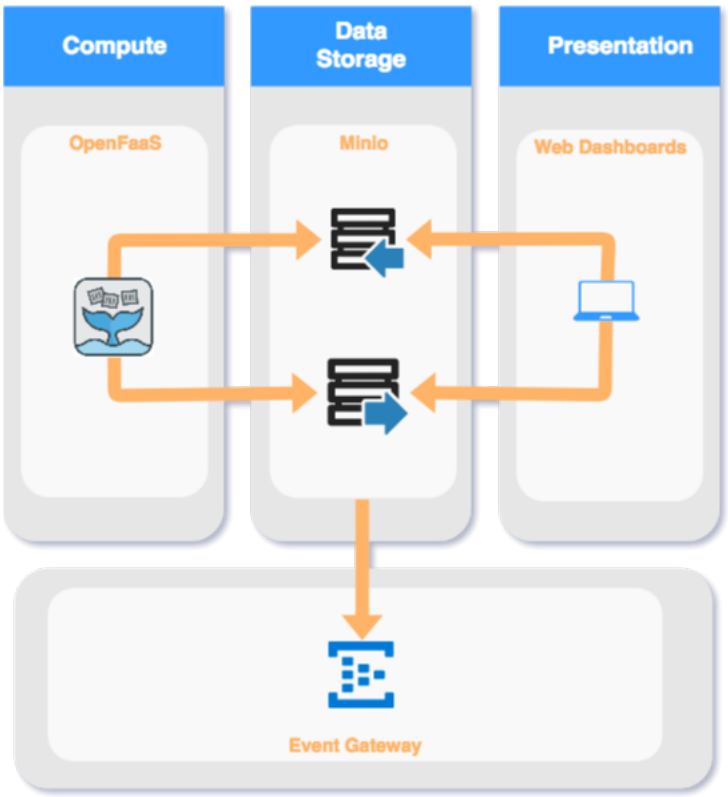

ILUSTRACIÓN 1: DIAGRAMA DE FLUJO DE OSCAR

Tal y como se puede observar en el diagrama anterior, desde el frontend que se ha desarrollado en este proyecto, el usuario crea un objeto dentro del bucket de entrada. Una vez creado dicho objeto Minio emite un evento que es capturado y procesado por Event Gateway. Después de procesar el evento, Event Gateway se encarga de ejecutar

la función asociada a dicho evento, la cual está desarrollada utilizando OpenFaaS. Si la función termina con éxito, crea un nuevo objeto en el bucket de salida de Minio al cual el usuario tiene acceso desde la misma interfaz web.

Llegados a este punto, es importante explicar porqué se ha decidido trabajar sobre Kubernetes, OpenFaaS, Minio y Event Gateway. Tal y como se ha explicado anteriormente OSCAR desplegará un conjunto de servicios sobre un clúster Kubernetes, permitiendo ejecutar aplicaciones empaquetadas en imágenes Docker. Asimismo, ofrecerá una API de acceso muy similar a la de OpenFaaS, por tanto, desarrollar el frontend sobre la API de OpenFaaS facilita una posterior integración con OSCAR. OpenFaaS se puede utilizar para crear funciones desde cero utilizando su interfaz de comandos, en adelante CLI, o desplegar funciones a partir de aplicaciones ya empaquetadas en contenedores. El CLI permite crear aplicaciones auto escalables sobre un sistema Kubernetes generando de forma automática una imagen Docker que se utilizará posteriormente para el despliegue de la función. Minio es un sistema de almacenamiento orientado a bloques similar a Amazon S3, el cual permite crear de forma sencilla y sobre Kubernetes un sistema de almacenamiento distribuido. Minio es compatible con Amazon S3, lo cual es una ventaja importante ya que permite integrar fácilmente herramientas desarrolladas sobre dicho servicio. Por último, Event Gateway permite reaccionar a eventos emitidos por terceras partes, por tanto, lo convierte en una herramienta muy útil a la hora de integrar distintos servicios. OSCAR realizará el despliegue de un conjunto de servicios sobre un clúster Kubernetes por tanto event Gatway es una herramienta muy útil para poder ofrecer reactividad entre los servicios.

3.1. Despliegue del backend

En este apartado se define brevemente cada una de las herramientas utilizadas y se explica la configuración necesaria para desplegar el backend en un entorno local de desarrollo.

3.1.1. KUBERNETES

Kubernetes es una herramienta de código abierto desarrollada para gestionar la automatización del despliegue, ajuste de escala y manejo de aplicaciones empaquetadas en contenedores. Simplificando mucho, un clúster Kubernetes contiene componentes maestros y componentes nodos [13].

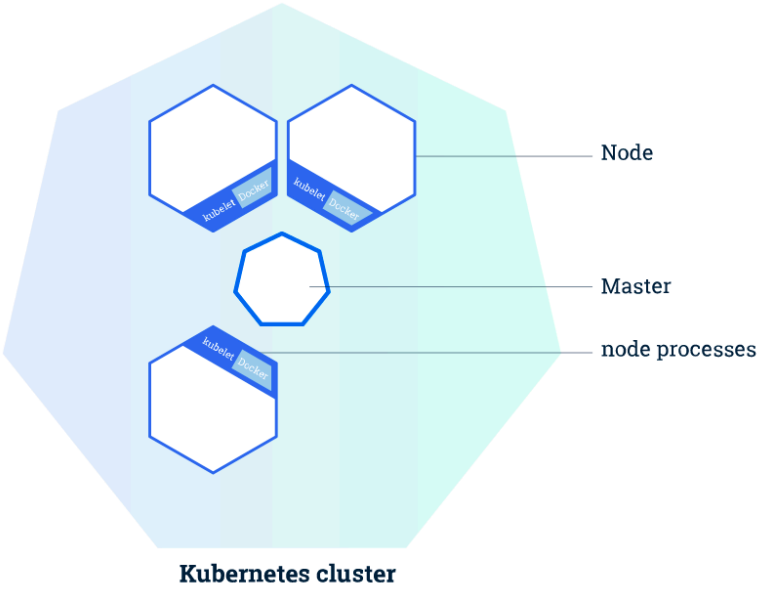

ILUSTRACIÓN 2: ARQUITECTURA DE NODOS DE KUBERNETES

3.1.1.1. Nodos maestros

Los componentes maestros proporcionan el plano de control del grupo, es decir, son los encargados de tomar decisiones globales sobre el clúster y detectan y responden a los eventos del clúster. Por ejemplo, iniciando un nuevo pod en caso de que no se estén cumpliendo las condiciones de un controlador de replicación. Mediante el componente ***kube-apiserver,*** diseñado para escalar horizontalmente, el maestro expone la API de control de Kubernetes. ***Etc*** es el almacén de valores clave constante y altamente disponible utilizado como almacén de backups para todos los datos del clúster. Por otra parte, ***Kube-Scheduler***, es el componente encargado de realizar la asignación de pods recién creados a algún nodo para que los ejecute. El ***kube-controller-manager*** es el componente que ejecuta los controladores dentro del nodo maestro. Controladores:

- Controlador de nodo: responsable de detectar y responder cuando un nodo ha experimentado un fallo de parada.
- Controlador de replicación: se encarga de mantener el número correcto de pods para cada objeto controlador de replicación del sistema.
- Controlador de endpoints: se une a los servicios y módulos.
- Controladores de servicio de cuenta y token: Crea las cuentas predeterminadas y tokens de acceso a la API para cada espacio de nombre nuevo.

El último componente del nodo maestro es el ***cloud-controller-manager*** permite que el código de los proveedores cloud y el núcleo de Kubernetes evolucionen de forma independiente.

3.1.1.2. Nodos "*worker*"

Los componentes de los nodos "*workers*" se ejecutan en cada nodo, manteniendo los pods en ejecución y proporcionando el entorno de tiempo de ejecución de Kubernetes.

El componente ***kubelet*** se asegura de que los contenedores se estén ejecutando en un pod. El ***kubelet*** toma un conjunto de *PodSpecs* que se proporcionan a través de diversos mecanismos y asegura que los contenedores descritos en esas *PodSpecs* se ejecutan y gozan de buena salud. Para finalizar, el componente ***kube-proxy*** es el que se encarga de habilitar la abstracción de servicios de Kubernetes manteniendo las reglas de red en el host y realizando el reenvío de conexión [14].

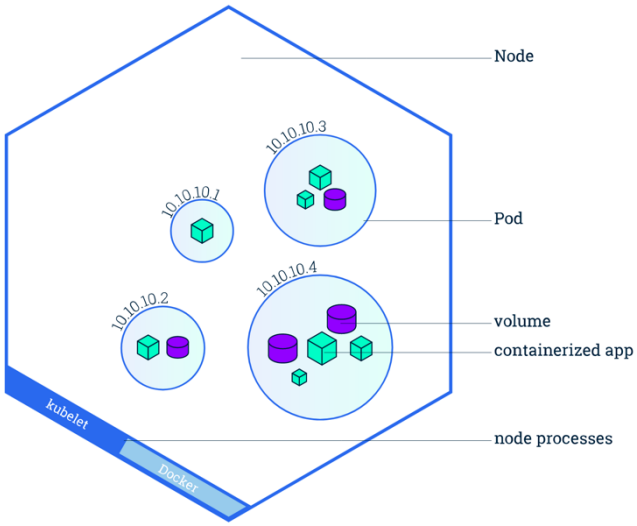

ILUSTRACIÓN 3: NODO WORKER EN KUBERNETES

Kubernetes es una herramienta mucho mas compleja que lo explicado anteriormente, pero es suficiente para que el lector se haga una idea de como se despliegan sobre Kubernetes el resto de las herramientas que se utilizan.

3.1.2. OpenFaaS

OpenFaaS (Functions as a Service) es un framework para construir funciones serverless con Docker y Kubernetes, que cuenta con soporte para métricas. Cualquier proceso puede ser empaquetado como una función lo que hace que sea posible consumir una gama de eventos web sin necesidad de programarlos de forma repetitiva.

Permite instalar funciones desarrolladas en casi cualquier lenguaje de programación para Linux o Windows y paquetes en formato de imagen de Docker/OCI con un solo clic desde su interfaz de usuario. También dispone de un CLI que utiliza ficheros con formato YAML para plantillas y definición de funciones. Por último, y no por eso menos importante, permite auto escalado según aumentan las necesidades.

OpenFaas está disponible para ser utilizado en Kubernetes bajo el proyecto faas-netes,

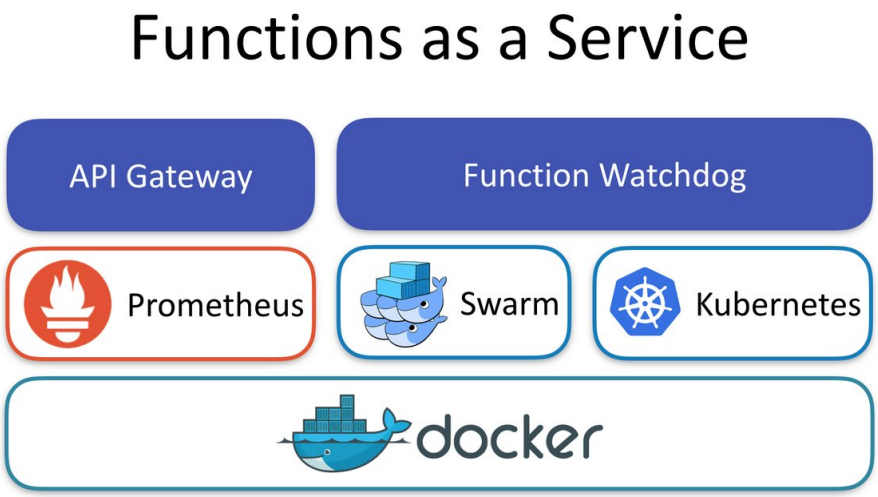

ILUSTRACIÓN 4: ESQUEMA DE FUNCIONAMIENTO DE OPENFAAS

el cual se ha utilizado para el desarrollo de este proyecto. En la ilustración anterior se puede observar la estructura que utiliza OpenFaaS para convertir cualquier desarrollo empaquetado en una imagen Docker en una función serverless.

3.1.3. MINIO

Es un servidor de almacenamiento de objetos compatible con el servicio de almacenamiento en la nube Amazon S3. Es adecuado para almacenar datos no estructurados, perfecto para la aplicación que se esta desarrollando ya que no es necesario almacenar los objetos de forma estructurada. El tamaño máximo de un objeto es de 5TB.

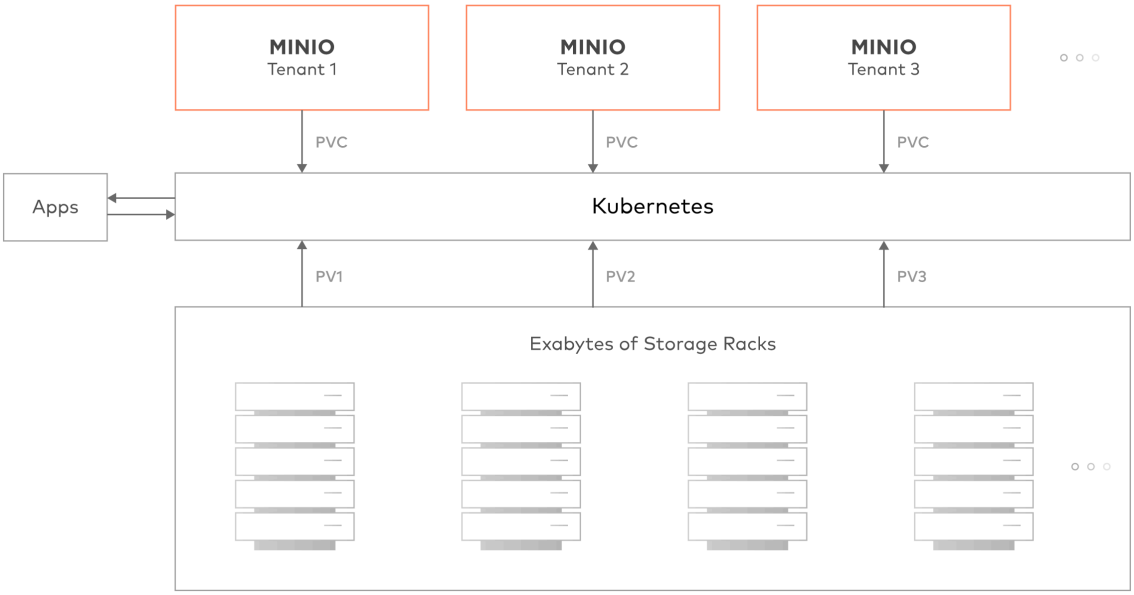

ILUSTRACIÓN 5: ARQUITECTURA DE MINIO SUBRE KUBERNETES [15]

Minio proporciona protección de datos y alta disponibilidad, dentro de un *tenant*, y soporta la mitad del número de fallas de nodo. Kubernetes ofrece una alta disponibilidad entre los tenant y los racks. Permite realizar actualizaciones progresivas en lotes, sin tiempo de inactividad. Kubernetes facilita la implementación y la administración de varios tenants de Minio. Cada tenant puede ejecutar una versión diferente de Minio y actualizar a su propio ritmo. Kubernetes se encarga de manejar la administración de recursos y la programación entre los tenants. A estas características debe de añadirse que, como explican en la propia web del proyecto, la utilización de Minio sobre Kubernetes, permite administrar la configuración y las credenciales de Minio, utilizando Kubernetes ConfigMaps y Secrets, cuando se implementen a través de Helm Chart.

3.1.4. SERVERLESS EVENT GATEWAY

Serverless Event Gateway es un enrutador de eventos. Permite enlazar funciones en endpoints http, capturar y reaccionar frente a cualquier tipo de evento, así como compartir suscripciones de eventos con terceras partes. Es importante destacar la capacidad que ofrece para redirigir eventos automáticamente a servicios externos como Kinesis, Firehose, SQS, etc.

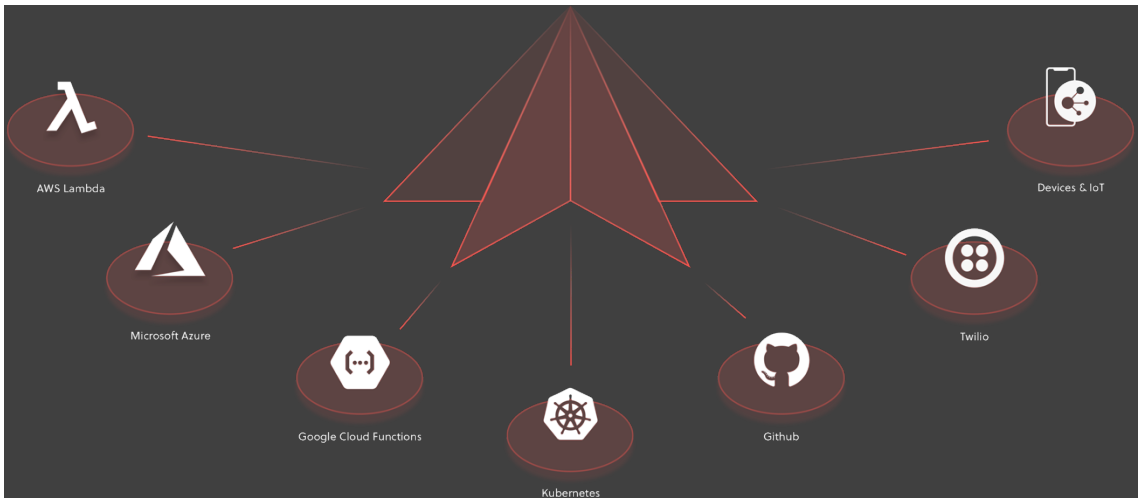

ILUSTRACIÓN 6:SERVICIOS COMPATIBLES CON EVENT GATEWAY

3.1.5. CONFIGURACIÓN Y DESPLIEGUE DEL BACKEND.

Llegados a este punto del documento ya se puede empezar a detallar como se ha realizado el despliegue del backend de la aplicación.

En primer lugar, es importante destacar que, para trabajar en el entorno de desarrollo local, se ha utilizado ***minikube*** como herramienta para ejecutar un clúster Kubernetes de un único nodo. Aunque *minikube* soporta la ejecución en el propio host, para este proyecto se ha ejecutado dentro de una máquina virtual. En concreto se ha utilizado el hipervisor Virtual Box.

Antes de instalar *minikube* es necesario instalar el cliente de línea de comandos ***kubectl***, el cual permite interactuar con el nodo maestro de Kubernetes.

3.1.5.1. DESPLIEGUE DE OPENFAAS SOBRE KUBERNETES

Como ya se ha indicado previamente, para desplegar OpenFaaS sobre Kubernetes se ha utilizado faas-netes [16]. Faas-netes crea de forma automática dos espacios de nombres en Kubernetes.

Una vez descargado el código fuente desde el repositorio (https://github.com/openfaas/faas-netes) y utilizando la herramienta kubectl, se debe de ejecutar la siguiente instrucción para crear cada uno de los espacios de nombres:

```
kubectl apply –f ./namespaces.yml
```

El primer espacio de nombres creado es "*openfaas"*, utilizado para los servicios de OpenFaaS. El siguiente espacio de nombres que se crea es "*openfaas-fn*" en el cual se desplegarán las funciones. A continuación, se crean todos los servicios, despliegues y mapas de configuración. Para ello es necesario ejecutar la siguiente instrucción:

```
kubectl apply –f ./yaml
```

Según lo explicado en la documentación y corroborado a posteriori, faas-netes permite la actualización de funciones sin *downtime*, es decir, sin que el sistema deje de estar disponible. Para ello se utiliza el mecanismo de replicación. Cuando se utiliza la instrucción "faas-cli deploy –update" se crea un pod nuevo en el espacio de nombres "openfaas-fn" sin eliminar el pod que se utiliza para la ejecución de la versión anterior de la función. Cuando el controlador de nodo detecta que el nuevo pod se ha iniciado correctamente y goza de buen estado de salud, se encarga de terminar el pod con la versión anterior de la función y redirige las peticiones nuevas a dicho pod.

3.1.5.2. DESPLIEGUE DE EVENT GATEWAY

Por facilidad en futuras integraciones con Kubernetes se ha decido trabajar sobre la imagen Docker proporcionada por Serverless, la cual permite desplegar de forma más o

menos sencilla Event Gateway. Para ello se deben exponer los puertos 4000 y 4001 del contenedor donde se esté ejecutando el servicio, ya que el puerto 4001 ofrece una API RESTful JSON para configurar el servicio y el puerto 4000 para realizar la subscripción y emisión de eventos. Ejecutando la siguiente instrucción se consigue desplegar el contenedor con los puertos de configuración y uso expuestos o redirigidos a la maquina host:

```
docker run -p 4000:4000 -p 4001:4001 serverless/event-gateway -dev
```

Como se verá en el siguiente apartado, Minio se puede configurar para publicar eventos vía Webhooks[1], lo que quiere decir que se recibe la información en el momento que sucede. Event Gateway puede ser configurado para suscribirse a los eventos que Minio emita y realizar alguna acción desencadenada por dicho evento, en este caso la acción que desencadenará será la ejecución de una función OpenFaaS que se encargue de procesar el objeto que ha producido el evento. Para que Event Gateway pueda suscribirse a los eventos emitidos por Minio es necesario realizar la siguiente configuración utilizando la API proporcionada para tal efecto, en este caso expuesta en el puerto 4001:

- Crear los tipos de eventos

En primer lugar, se deben crear los tipos de eventos. Estos no son más que una categorización de eventos. Cada evento emitido en un espacio debe tener un tipo de evento registrado previamente.

```
curl --request POST \
    --url http://localhost:4001/v1/spaces/default/eventtypes \
    --header 'content-type: application/json' \
    --data '{
        "name": "ObjectCreated",
        "metadata": {}
    }'
```

- Crear las funciones

[1] Un **webhook** (o **WebHook**), en desarrollo web, es un método de alteración del funcionamiento de una página o aplicación web, con callbacks personalizados. Estos se pueden mantener, modificar y gestionar por terceros; desarrolladores que no tienen por qué estar afiliados a la web o aplicación. El término "webhook" fue inventado por Jeff Lindsay en 2007, a partir del término de programación Hook.

Las funciones son redirigidas a la URL pública del proveedor de funciones serverless, pudiendo ser cualquiera de los proveedores de servicios Cloud o funciones propias desarrolladas, por ejemplo, mediante OpenFaaS.

```
curl --request POST \
    --url http://localhost:4001/v1/spaces/default/functions \
    --header 'content-type: application/json' \
    --data '{
        "functionID": "minio-objectcreated"

        "type": "http",
        "provider": {"url":"'$OPENFAAS_URL'/function/minio-objectcreated"},
        "metadata": {}
    }'
```

- Crear las suscripciones

Una suscripción permite captar los eventos de un tipo en concreto y ejecutar una función asociada a dicho tipo de eventos.

```
curl --request POST \
    --url http://localhost:4001/v1/spaces/default/subscriptions \
    --header 'content-type: application/json' \
    --data '{
        "type": "sync",
        "eventType": "ObjectCreated",
        "functionId": "minio-objectcreated",
        "method": "POST",
        "path": "/minio-objectcreated",
        "metadata": {}
    }'
```

En los ejemplos anteriores, se crea una suscripción para los eventos de tipo “ObjectCreated” y cuando se detecta un evento de este tipo, se ejecuta la función “minio-objectcreated” la cual está ejecutando la función serverless ubicada en la url “$OPENFAAS_URL/function/minio-objectcreated”.

3.1.5.3. Despliegue de Minio.

Posiblemente el despliegue de Minio sobre Kubernetes sea el más complicado de los que se han tenido que hacer en este proyecto ya que se debe crear correctamente un config-map en Kubernetes para configurar la emisión de eventos relacionados con la gestión de los buckets, es decir, los eventos asociados a las acciones Create, Read, Update y Delete, en adelante CRUD, sobre los objetes de cada bucket.

Por no extender demasiado el documento, y no convertirlo en un tutorial del despliegue de Minio sobre Kubernetes, únicamente se van a detallar los aspectos mas importantes dentro de dicho despliegue.

En primer lugar, debe añadirse en el fichero *config.json* de Minio la configuración de los Webhook endpoints. En este caso el objetivo del Webhook es notificar, enviar un evento, a una dirección determinada indicando que se ha realizado una acción sobre un determinado objeto en un bucket de Minio. Para ello basta con indicar el identificador del Webhook, si está habilitado o no, y el endpoint donde se encuentra. En este caso, los endpoints deben apuntar a la API de recepción de eventos de Event Gateway.

```
"webhook": {
    "1": {
        "enable": true,
        "endpoint": "http://192.168.1.235:4000/minio-objectcreated"
    },
    "2": {
        "enable": true,
        "endpoint": "http://192.168.1.235:4000/minio-objectaccessed"
    },
    "3": {
        "enable": true,
        "endpoint": "http://192.168.1.235:4000/minio-objectremoved"
    }
}
```

Una vez desplegado el servicio, es necesario indicarle a Minio qué buckets deben emitir eventos y qué tipo de evento son. Para ello, desde el cliente de línea de comandos de Minio, se debe ejecutar la siguiente instrucción:

```
mc events add myminio/oscar arn:minio:sqs::1:webhook --events put
```

En la instrucción anterior existe un detalle importante que cabe destacar. El número 1 que se ha marcado en color morado, debe coincidir con el identificador del Webhook establecido en el fichero de configuración, de este modo cuando, por ejemplo, se cree un objeto en el bucket “oscar” de la instancia “myminio”, se enviará un evento al endpoint http://192.168.1.235:4000/minio-objectcreated.

Los eventos emitidos por Minio son totalmente compatibles con los que utiliza Amazon S3 [17]. Por tanto, la estructura del mensaje emitido es la misma y cien por cien compatible con las aplicaciones que utilicen dicho servicio.

3.2. Despliegue y desarrollo del Frontend

El frontend de la aplicación se ha desarrollado mediante el framework Vue.js [18]. Como explican en su propia pagina web, Vue.js es un framework progresivo para el desarrollo de interfaces de usuario. La principal diferencia frente a otros frameworks monolíticos es que Vue.js está diseñado desde cero para ser adoptable de forma incremental. El core de Vue.js sólo se centra en la capa de vista y es fácil de integrar con otras librerías o proyectos existentes. Por otra parte, Vue.js también es perfectamente capaz de potenciar Single-Page Applications (SPA) más sofisticadas cuando se utiliza en combinación con herramientas modernas y bibliotecas de soporte. Como el resto de frameworks frontend modernos Vue.js se basa en la creación de componentes reutilizables que son insertados a posteriori en la página a medida que el usuario los solicita tal y como se explicará con mas detalle en siguientes apartados.

Para el desarrollo de la interfaz gráfica de OSCAR se ha utilizado Vuetify [19] juntamente con Vue.js. Vuetify es una librería de componentes reutilizables para Vue.js que aproxima el diseño de las interfaces desarrolladas utilizando el modelo Material Design[20] a Vue.js.

La aplicación se ha desarrollado como una SPA ya que la página no se recarga cada vez que el usuario solicita información nueva, sino que la información obtenida del backend se inyecta al componente que la solicita sin tener que refrescar todo el contenido de la página, agilizando así la navegación entre páginas.

3.2.1. Estructura de la interfaz

ILUSTRACIÓN 7: DASHBOARD DE LA INTERFAZ GRÁFICA DE OSCAR

La interfaz gráfica desarrollada para el control de OSCAR, ha sido implementada como un panel de administración de cualquier backend o servicio web, de manera que el usuario pueda adaptarse rápidamente ya que es una estructura bastante común. Se puede decir que la interfaz se divide en tres componentes principales, la barra de tareas de la parte superior, el menú situado en la parte izquierda de la pantalla y en contenedor principal, que como su propio nombre indica, alberga el contenido de la página web.

3.2.2. ESTRUCTURA DE DIRECTORIOS:

En la ilustración 7 se puede observar el árbol de directorios del proyecto, a continuación, se detalla la finalidad de los mas importantes.

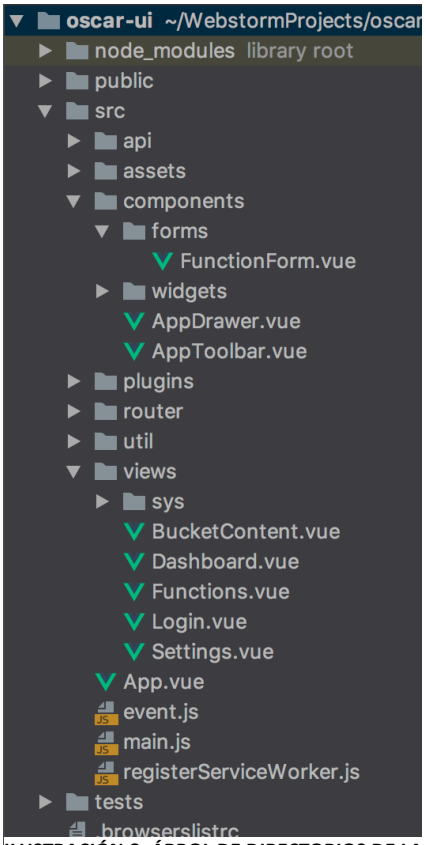

ILUSTRACIÓN 8: ÁRBOL DE DIRECTORIOS DE LA APLICACIÓN

Assets: Almacena los archivos que se importan dentro de los componentes.

Components: Todos los componentes de la aplicación que no son componentes principales de una vista.

Router: En el caso de que se trabaje con Vue Router, en este directorio se almacena un documento JavaScript que contiene todas las rutas de la aplicación.

Utils: Puede almacenara funciones que se utilizan en varios componentes, así como constantes utilizadas a lo largo del proyecto.

Views: En este directorio se almacenan todos los componentes que se renderizan como páginas, es decir, todos los que están enrutados desde el componente Vue Router.

Plugins: En este caso concreto solo se almacena el fichero de configuración de Vuetify, en el cual se importan solamente los componentes a utilizar para evitar sobrecargar la página con componentes que no se utilizan.

3.2.3. COMPONENTES Y VISTAS

Llegados a este punto, es conveniente explicar qué componentes y vistas se han desarrollado a lo largo del proyecto. En Vue.js gracias a *webpack* [21] y al módulo *vue-loader* tanto los componentes como las vistas se pueden desarrollar como ficheros únicos con extensión vue. El siguiente código, obtenido de la pagina de Vue.js [22], muestra un ejemplo de la estructura de los componentes de un solo archivo.

```
<template>
  <div class="example">{{ msg }}</div>
</template>

<script>
export default {
  data () {
    return {
      msg: 'Hello world!'
    }
  }
}
</script>

<style>
.example {
  color: red;
}
</style>
```

El primer bloque contiene los elementos HTML u otros componentes de Vue.js que se van a utilizar dentro del componte que se está definiendo. Contiene la vista. En el bloque <script> se almacena la lógica del componente. Por último, en la sección <style> se almacena el diseño. Como detalle importante, cabe destacar que si se añade el atributo *scoped* a la etiqueta <style>, todos los estilos que se definan dentro del bloque solamente afectarán al componente en cuestión.

3.2.3.1. CONFIGURACIÓN DE LOS ENDPOINTS Y LAS CREDENCIALES.

La motivación principal del proyecto es crear una interfaz que unifique el servicio de almacenamiento por bloques, Minio, y el servicio de creación de funciones serverless, OpenFaaS. Para ello es necesario que el usuario introduzca los endpoints donde ha desplegado dichos servicios y las credenciales de acceso. Se ha creado una pantalla de configuración en la que el usuario puede hacer esto de forma sencilla. A dicha pantalla se accede desde la opción “Settings” del menú principal.

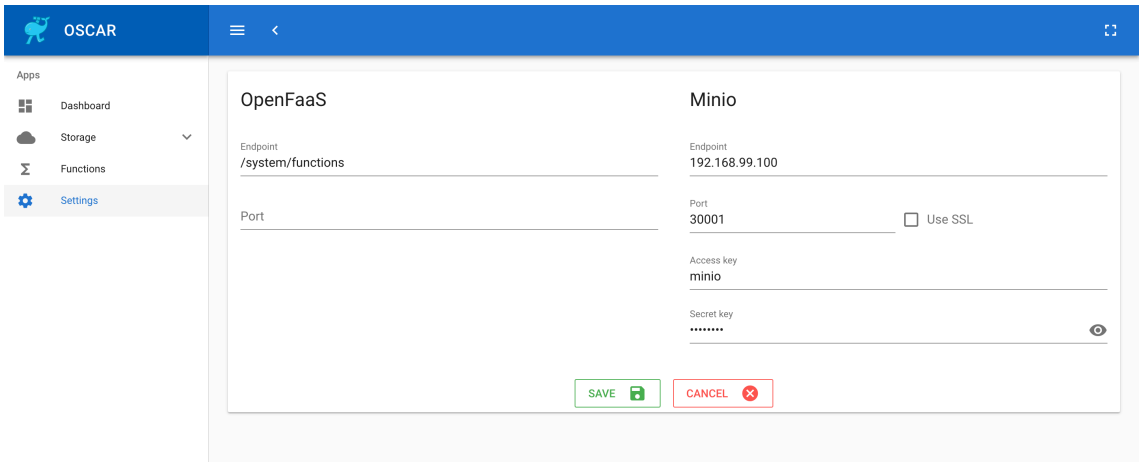

ILUSTRACIÓN 9: PANTALLA DE CONFIGURACIÓN DE ACCESOS

Como se puede observar en la ilustración anterior, la pantalla se divide en dos secciones bien diferenciadas, OpenFaaS y Minio. OpenFaaS requiere menor configuración ya que ofrece una API de acceso libre a la que se puede acceder solamente configurando la dirección y el puerto. En cambio, el SDK de Minio para JavaScript requiere autenticación, por esto, es necesario introducir la dirección y el puerto, si requiere o no navegación segura y por último las credenciales de acceso al servicio.

En Vue.js se puede compartir información entre componentes de varios modos. Cuando se trata de pasar información de un componente padre a un componente hijo, se puede hacer de forma sencilla mediante las propiedades del componente hijo. Es decir, en el componente hijo se declaran como propiedades los objetos, variables o arrays que se deban recibir en el momento que se instancia dicho componente hijo. A continuación, se muestra como se han declarado en el componente hijo, en este caso la vista "Settings", las propiedades que debe recibir del componente padre.

```
<script>

export default {
  props: {
    minio: {},
    openFaaS: {}
  },
```

Como es necesario que varios de los componentes de la aplicación compartan la configuración y credenciales de acceso, se ha decidido declarar en la función data() de la instancia principal de Vue, (App.vue) los objetos que almacenarán dicha información, de modo que esta información puede transmitirse de forma fácil de componentes padres a componentes hijos. Pero ¿Cómo se pasa la información de los componentes hijos a los componentes padres? Para transmitir la información ascendentemente es posible utilizar Vuex [23] aunque no está recomendado para aplicaciones que tengan

que gestionar poco estado como es el caso de este proyecto y, por tanto, se decidió no utilizarlo para no añadir complejidad innecesaria al proyecto. La única alternativa que quedaba era utilizar un bus de eventos para transmitir información entre componentes de forma ascendente o de forma horizontal, es decir, de componentes hijos a componentes padres o entre componentes hermanos.

En la instancia principal de Vue.js se crea un manejador de eventos. Dicho manejador de eventos está suscrito a los eventos personalizados que se emiten desde todos los componentes de la aplicación. En la ilustración 10 se puede ver un diagrama del ciclo de vida de un componente en Vue.js. Los métodos aparecen de color rojo y con un borde también de color rojo. Estos métodos se llaman cada vez que un componente realiza una transición de un estado al siguiente, por ejemplo, el método `created` se ejecuta cuando la creación del componente ha finalizado. Es importante tener en cuenta el ciclo de vida de un componente dado que, para que la instancia principal de Vue.js, que también es un componente, pueda suscribirse a los eventos emitidos por otros componentes, ésta debe estar ya creada. Por tanto, la suscripción a los eventos se crea desde el método "*created()*".

```
{
        name: 'APP_SHOW_SNACKBAR',
        callback: (data) => { this.onShowSnackbar(data) })
    },
    {
        name: 'MINIO_RECONNECT',
        callback: () => { this.createMinioClient() })
    }
```

Array de objetos AppEvents.

```
  created () {
    AppEvents.forEach(item => {
      this.$on(item.name, item.callback)
    })
    window.getApp = this
  },
```

La porción anterior de código muestra el método "created()" de la instancia principal de Vue.js. Como se puede observar dentro del método se itera sobre los objetos que conforman el array de eventos y se añaden las suscripciones. De esta manera, cuando un componente emita un evento, la instancia principal reaccionará frente a él ejecutando el callback asociado.

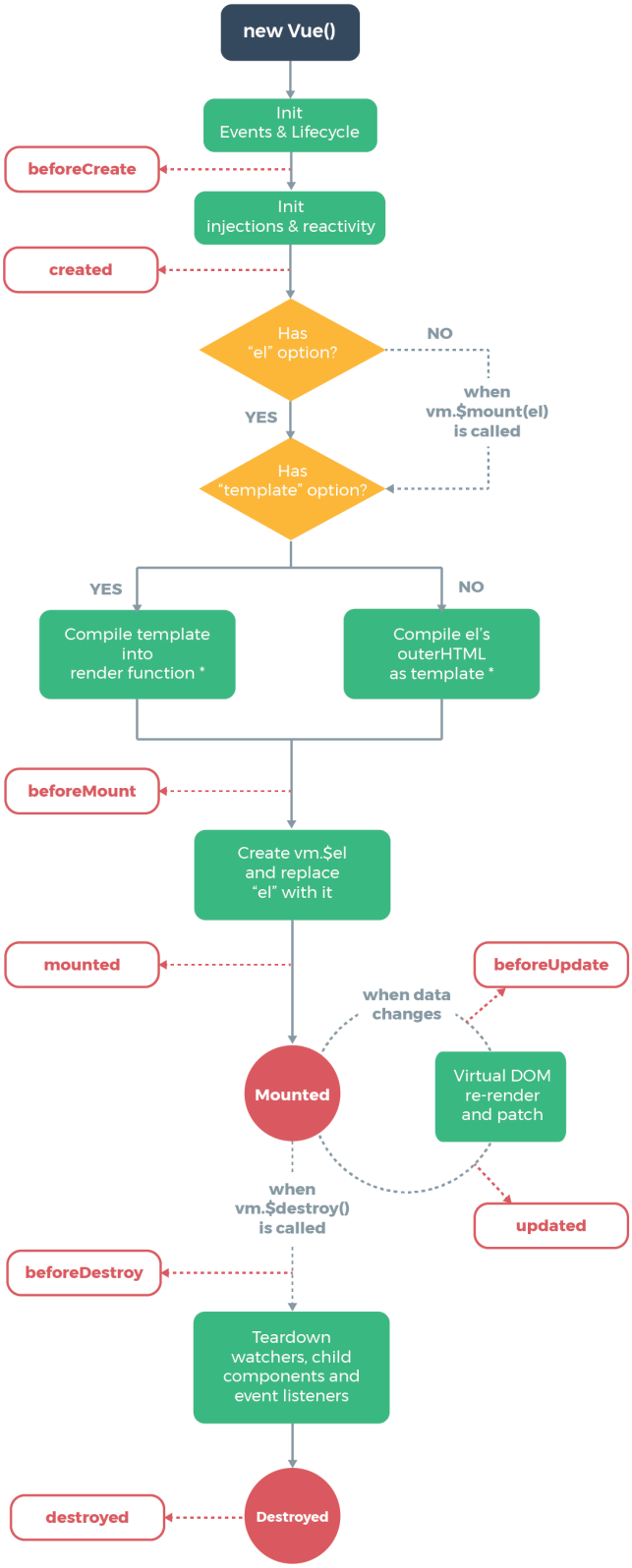

ILUSTRACIÓN 10: CICLO DE VIDA DE UN COMPONENTE [24]

Una vez explicada la comunicación entre componentes solamente queda detallar cómo la instancia principal reacciona a la actualización de credenciales de ambos servicios. El componente “Settings” recibe una copia de las credenciales almacenadas en la instancia principal mediante la utilización de sus propiedades. Cuando el usuario desea modificar las credenciales de acceso a alguno de los servicios, se desencadena un evento personalizado que comunica los valores nuevos a la instancia principal para que vuelva a crear el cliente de Minio o actualice el puerto y el endpoint de acceso a OpenFaaS. El siguiente código muestra los métodos ejecutados por las llamadas asíncronas del evento `'MINIO_RECONNECT'` y del evento `'APP_SHOW_SNACKBAR'`.

```
methods: {
  onShowSnackbar (data) {
    this.snackbar.text = data.text
    this.snackbar.color = data.color
    this.snackbar.showBucketContent = true
  },
  createMinioClient () {
    this.minioClient = null
    this.minioClient = new Client({
      endPoint: this.minio.endpoint,
      port: this.minio.port,
      useSSL: this.minio.useSSL,
      accessKey: this.minio.accessKey,
      secretKey: this.minio.secretKey
    })
  }
},
```

El evento `'MINIO_RECONNECT'` vuelve a crear el objeto que actúa como cliente de Minio. Dicho objeto permite a Vue.js acceder a la API de Minio para interactuar con el servicio de almacenamiento. Por otra parte, el evento `'APP_SHOW_SNACKBAR'` permite al sistema mostrar notificaciones al usuario utilizando el componente *snackbar* de Vuetify. A continuación, se muestra una imagen del *snackbar* indicando al usuario que las credenciales se han actualizado con éxito.

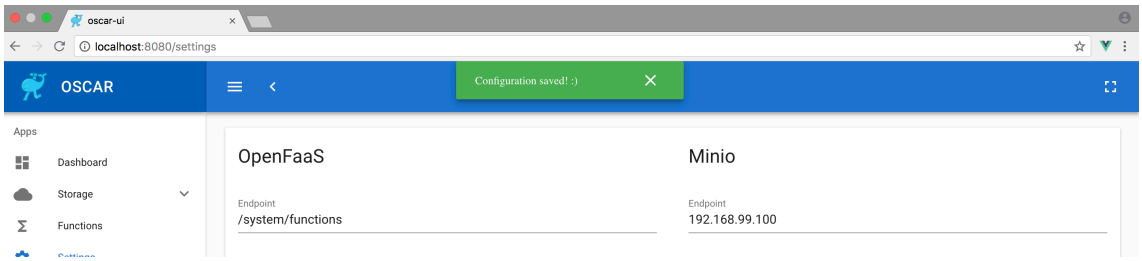

ILUSTRACIÓN 11: NOTIFICACIONES DE OSCAR-UI

3.2.3.2. GESTIÓN DE OBJETOS Y BUCKETS

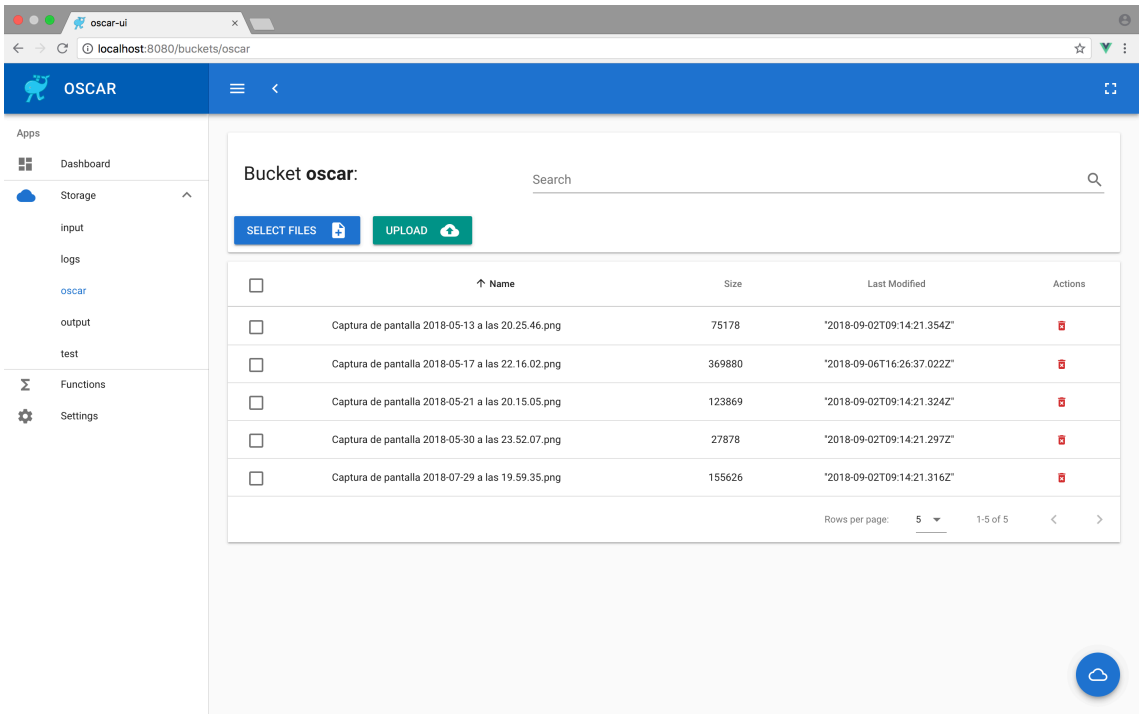

ILUSTRACIÓN 12: VENTANA DE GESTIÓN DE OBJETOS Y BUCKETS

La ilustración 12 muestra la interfaz de gestión de objetos y buckets. En primer lugar, es importante mencionar que el usuario no solo puede acceder a los buckets del menú y listar sus objetos, sino que el usuario puede crear y eliminar buckets de forma dinámica. Los buckets creados dinámicamente se añaden automáticamente al menú para que estén en todo momento a disposición del usuario.

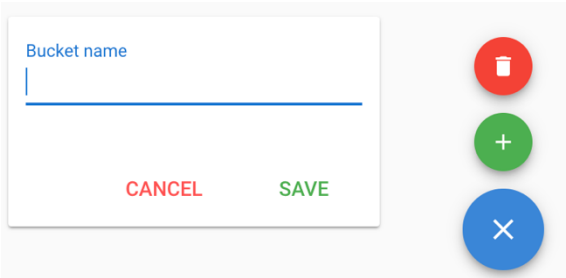

ILUSTRACIÓN 13: GESTIÓN DE BUCKETS

La ilustración de la derecha muestra el menú de gestión de buckets. Para desplegar dicho menú se utiliza un botón flotante, con una nube como icono, ubicado en la parte inferior derecha de la pantalla. Cuando el usuario clica sobre dicho botón, se despliega el menú que permite crear o eliminar buckets de Minio. Como se ha mencionado anteriormente Minio ofrece un cliente JavaScript que permite interactuar con el servidor de forma bastante cómoda.

```
createBucket (name) {
      this.minioCreateBucket(name).then(() => {
        window.getApp.$emit('APP_SHOW_SNACKBAR', {
          text: `Bucket ${name} has been successfully created`,
          color: 'success'
        })
```

```
        window.getApp.$emit('REFRESH_BUCKETS_LIST')
      }).catch((err) => {
        window.getApp.$emit('APP_SHOW_SNACKBAR', {
          text: err.message,
          color: 'error'
        })
      }).finally(() => {
        this.menu = false
        this.newBucketName = ''
      })
    },
    minioCreateBucket (name) {
      return new Promise((resolve, reject) => {
        this.minioClient.makeBucket(name, function (err) {
          if (err) {
            reject(err)
          }
          resolve(true)
        })
      })
    },
    removeBucket (name) {
      this.minioRemoveBucket(name).then(() => {
        window.getApp.$emit('APP_SHOW_SNACKBAR', {
          text: `Bucket ${name} has been successfully created`,
          color: 'success'
        })
        window.getApp.$emit('REFRESH_BUCKETS_LIST')
      }).catch((err) => {
        window.getApp.$emit('APP_SHOW_SNACKBAR', {
          text: err.message,
          color: 'error'
        })
      })
    },
    minioRemoveBucket (name) {
      return new Promise((resolve, reject) => {
        this.minioClient.removeBucket(name, function (err) {
          if (err) {
            reject(err)
          }
          resolve(true)
        })
      })
    }
  }
```

El código anterior muestra los métodos utilizados para crear y eliminar buckets de forma asíncrona y utilizando promesas de JavaScript. “Una promesa es un objeto que

representa la terminación o el fracaso eventual de una operación asíncrona" [25]. Por tanto, el uso de promesas asegura la terminación o el fracaso de una operación asíncrona. Para este proyecto y para muchos otros en los que se trabaja en comunicaciones asíncronas con servicios de terceros, es importante asegurar que en el momento que terminen, ya sea con éxito o con fracaso, las llamadas a dichos servicios, se pueda reaccionar; bien inyectando en algún componente la información obtenida o simplemente mostrando un mensaje de éxito o error al usuario.

En el caso de la creación y eliminación de buckets, el usuario envía una petición al servidor de Minio, el cual responde en caso afirmativo con un mensaje de éxito o por el contrario con un mensaje de error. El usuario debe saber si su bucket se ha creado o eliminado de forma correcta. La utilización de promesas permite enviar la petición del usuario al servidor e informarle con la respuesta obtenida. Si la petición de creación de un bucket termina exitosamente el usuario verá una notificación informándole de que el bucket se ha creado satisfactoriamente y éste se añadirá a la lista de buckets disponibles. En caso contrario, el usuario recibirá una notificación informando del motivo por el cual fracasó la operación. El funcionamiento de la eliminación de buckets es muy similar, pero en lugar de añadir el bucket a la lista de los disponibles, se elimina el bucket en cuestión.

Dado que el cliente de Minio no permite la utilización de promesas de forma nativa, se decidió crear un método que devolviera una promesa de la petición al servidor. De esta forma se asegura la terminación de la petición, permitiendo así, reaccionar cuando esta haya finalizado. El siguiente código muestra un ejemplo de creación de promesas sobre la petición al servicio.

```
return new Promise((resolve, reject) => {
        this.minioClient.makeBucket(name, function (err) {
          if (err) {
            reject(err)
          }
          resolve(true)
        })
      })
```

A continuación, se muestra una captura de pantalla que permite explicar la funcionalidad que ofrece la interfaz grafica de OSCAR sobre los objetos de un bucket.

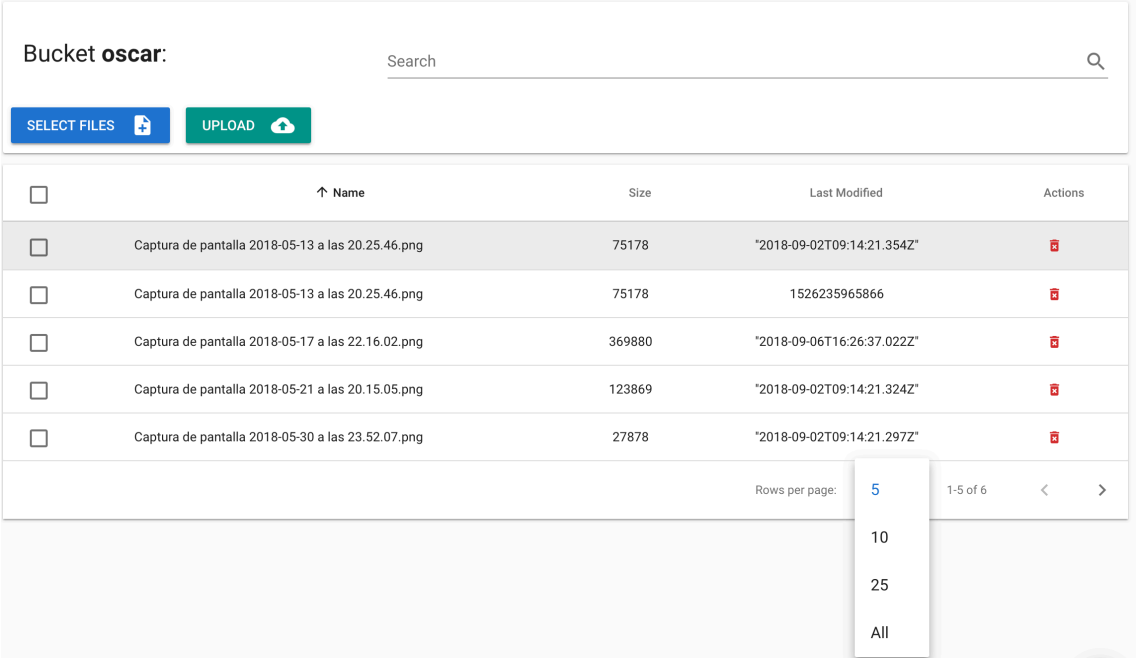

ILUSTRACIÓN 14: FILTRO Y PAGINACIÓN DE OBJETOS DENTRO DE UN BUCKET

En la parte superior izquierda se puede observar el nombre del bucket en el que se está trabajando. A su derecha está ubicado el buscador de objetos, el cual permite al usuario filtrar objetos dentro del grid. Dicho grid muestra de forma paginada todos los objetos del bucket. El número de objetos por página se puede modificar mediante el desplegable habilitado para ello.

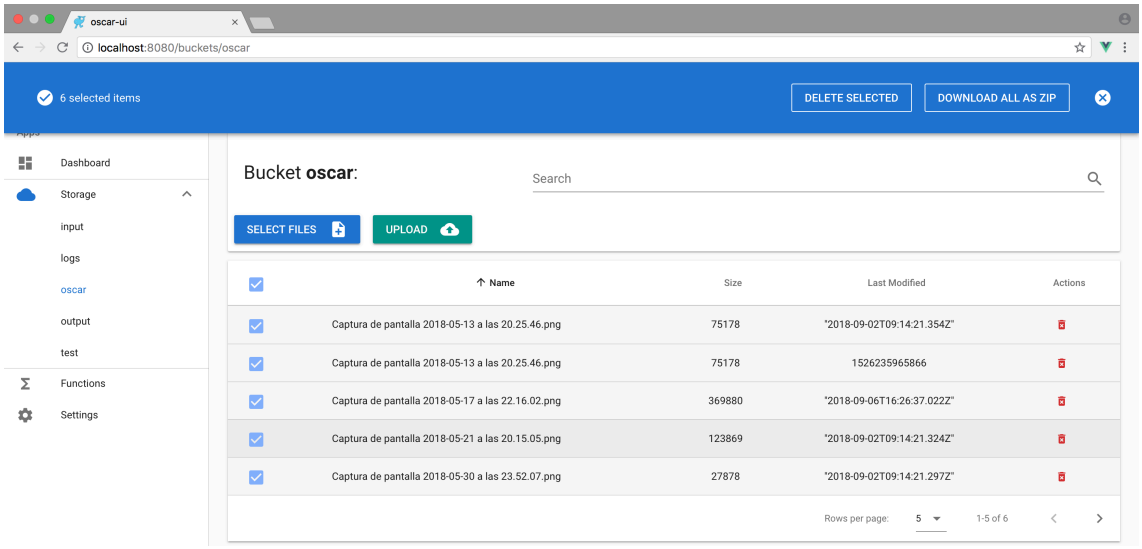

ILUSTRACIÓN 15: ELIMINACIÓN Y DESCARGA DE OBJETOS

La ilustración 15 permite ver como se pueden eliminar o descargar varios objetos en un mismo archivo comprimido en formato *zip*. También es posible eliminar objetos de forma individual clicando sobre el icono de la papelera ubicado en el campo “Acciones” del grid. Desde esta misma ventana, también es posible crear objetos dentro del bucket.

Para ello se habilita un botón de selección de archivos y otro para confirmar la subida de archivos.

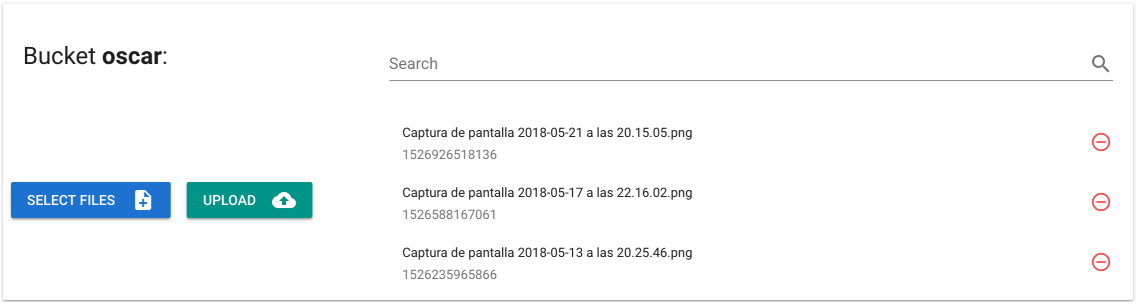

ILUSTRACIÓN 16: SELECCIÓN DE ARCHIVOS Y CREACIÓN DE OBJETOS

Como se puede ver en la anterior imagen, cuando se seleccionan los archivos que se quieren subir, aparecen listados a la derecha de los botones de selección y subida de archivos. Cada uno de los archivos seleccionados, se puede eliminar clicando sobre el icono habilitado para ello a la derecha del nombre del archivo.

Llegados a este punto solo queda explicar los detalles de la implementación del componente de gestión de objetos.

```
fetchData () {
    this.getBucketFiles(this.bucketName)
    .then(fileList => {
        this.files = fileList
    })
    .catch(err => {
        window.getApp.$emit('APP_SHOW_SNACKBAR', {
        text: err.message,
        color: 'error'
        })
    })
},
getBucketFiles (name) {
    return new Promise((resolve, reject) => {
    let files = []
    let stream = this.minioClient.listObjects(name, '', true)
    stream.on('data', function (obj) {
        files.push(obj)
    })
    stream.on('error', function (err) {
        reject(err)
    })
    stream.on('end', function (e) {
        resolve(files)
    })
    })
},
```

Como se puede ver en el anterior código fuente, la obtención de objetos desde el servidor se realiza de forma asíncrona mediante el uso de promesas tal y como se ha explicado en el punto anterior. Es importante mencionar como se utilizan los objetos obtenidos en formato json para ser mostrados como archivos dentro del grid. Como se puede ver en el siguiente código fuente, para crear el grid se utiliza el componente data-table de la librería Vuetify.

```
<v-data-table

    v-model="selected"
    :headers="headers"
    :items="files"
    class="elevation-1"
    item-key="name"
    :search="search"
    :pagination.sync="pagination"
    select-all
>
    <template slot="headers" slot-scope="props">
    <tr>
        <th>
        <v-checkbox
            :input-value="props.all"
            :indeterminate="props.indeterminate"
            primary
            hide-details
            @click.native="toggleAll"
        ></v-checkbox>
        </th>
        <th
        v-for="header in props.headers"
        :key="header.text"
        :class="['column sortable', pagination.descending ? 'desc' : 'asc',
header.value === pagination.sortBy ? 'active' : '']"
        @click="changeSort(header.value)"
        >
        <v-icon small>arrow_upward</v-icon>
        {{ header.text }}
        </th>
    </tr>
    </template>
    <template slot="items" slot-scope="props">
    <tr :active="props.selected">
        <td @click="props.selected = !props.selected">
        <v-checkbox
            :input-value="props.selected"
            primary
            hide-details
        ></v-checkbox>
```

```
        </td>
        <td class="text-xs-left">{{ props.item.name }}</td>
        <td class="text-xs-center">{{ props.item.size }}</td>
        <td class="text-xs-center">{{ props.item.lastModified }}</td>
        <td class="justify-center layout px-0">
        <v-icon small @click="removeFile(props.item)" color="red darken-
2">delete_forever</v-icon>
        </td>
    </tr>
    </template>
    <template slot="no-data">
    <v-alert :value="true" color="error" icon="warning">
        Sorry, there are no files to display in this bucket :(
    </v-alert>
    </template>
    <v-alert slot="no-results" :value="true" color="error" icon="warning">
    Your search for "{{ search }}" found no results.
    </v-alert>
</v-data-table>
```

La propiedad ítems del componente es la encargada de recibir los archivos que se desean mostrar en el data-table. Dichos archivos se deben pasar al componente como un array de objetos JSON. El array de objetos se rellena cuando el servidor de Minio responde con éxito a la petición enviada para obtener el listado de todos los objetos del bucket. La respuesta contiene un array de objetos JSON con mucha más información de la requerida para listar los objetos en el grid. En el grid únicamente se mostrarán los valores indicados en la etiqueta <table> del código HTML de la plantilla. El componente facilita la iteración dentro del *array* ya que de forma automática introduce un *<tr>* y tantos *<td>* como se definan en la plantilla para cada objeto, a continuación, se muestra la creación de una celda de la tabla. `<td class="text-xs-left">{{ props.item.name }}</td>`. En este caso se muestra la propiedad *"name"* del archivo en la iteración actual. Para crear las cabeceras de la tabla, se itera sobre el array *"headers"* el cual contiene el nombre de todas las celdas que forman parte de la cabecera de la tabla. Vue.js permite iterar sobre un array utilizando la directiva *v-for* como se muestra a continuación.

```
<th
v-for="header in props.headers"
:key="header.text"
:class="['column sortable', pagination.descending ? 'desc' : 'asc',
header.value === pagination.sortBy ? 'active' : '']"
@click="changeSort(header.value)"
>
    <v-icon small>arrow_upward</v-icon>
    {{ header.text }}
</th>
```

El cliente de Minio para JavaScript ofrece dos métodos para insertar objetos en el servidor, el primero de ellos `minioClient.putObject()` ofrece la posibilidad de subir un objeto desde un buffer o una cadena de texto, el segundo método que ofrece es `minioClient.fputObject()`, el cual ofrece la posibilidad crear objetos a partir de ficheros, pero con la peculiaridad que requiere acceso al sistema de ficheros y esto desde un navegador web no está permitido. Por tanto, para este proyecto no se puede utilizar ninguno de los métodos anteriores. Es por esto por lo que haciendo uso de la compatibilidad con Amazon S3, se optó por utilizar el SDK que ofrece dicho servicio, el cual permite insertar ficheros en un bucket desde el navegador web. La creación de objetos dentro de un bucket se gestiona desde el componente InputFile.vue, el cual se muestra dentro de la vista *"/buckets/nombreDelBucket"*. A dicho componente se le ha de pasar como propiedad un objeto con la información referente al servidor de Minio para poder realizar la conexión desde el SDK de Amazon S3.

```
minioUpload (file) {
    AWS.config.s3 = { s3BucketEndpoint: false }
    let s3Client = new AWS.S3({
        apiVersion: '2006-03-01',
        params: {Bucket: this.bucketName},
        endpoint: this.minio.endpoint,
        accessKeyId: this.minio.accesskey,
        secretAccessKey: this.minio.secretkey,
        s3ForcePathStyle: true,
        signatureVersion: 'v4',
        logger: window.console
    })

    let params = {
        Bucket: this.bucketName,
        Key: file.name,
        Body: file
    }
    this.files.find((f) => {
        if (f.name === file.name) {
            f.showUploading = true
        }
    })
    return s3Client.putObject(params).promise()
}
```

Como demuestra el código anterior, el SDK de Amazon S3 permite trabajar con promesas JavaScript de forma nativa. El método *minioUpload()* devuelve una promesa que se resolverá cuando se termine la creación de objetos en el servidor o en caso que no tenga éxito, devolverá un código de error.

3.2.3.3. GESTIÓN DE FUNCIONES SERVERLESS

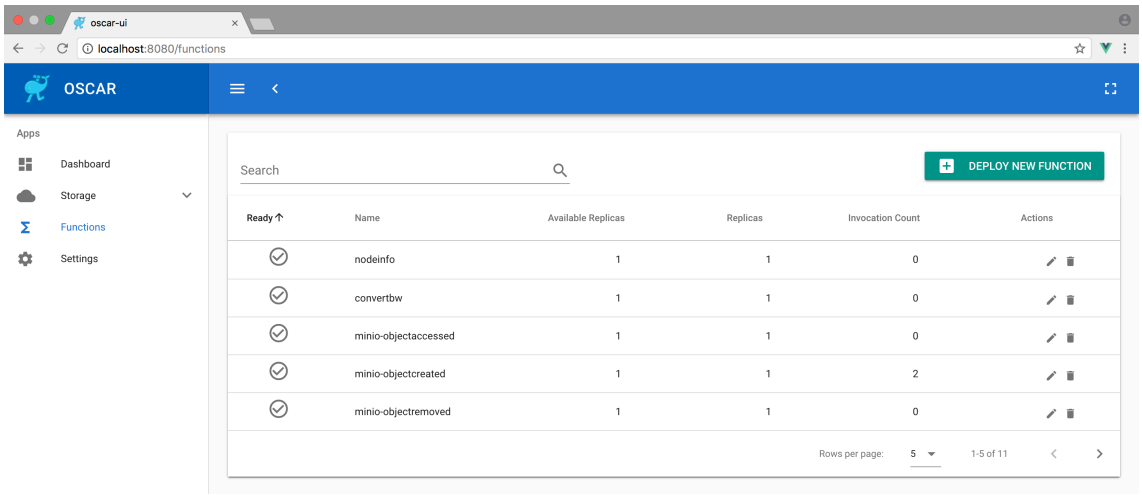

ILUSTRACIÓN 17: PANTALLA DE GESTIÓN DE FUNCIONES

Para facilitar la familiarización del usuario con el entorno, se decidió que tanto el mantenimiento de buckets y objetos como el de funciones serverless tuvieran un aspecto similar. En el caso de la pantalla de gestión de funciones se muestra un data-table paginado con las funciones disponibles en OpenFaas, así como el estado de la función. En el caso de que la función esté preparada para ser invocada, se muestra un check indicando que está activa. Por el contrario, si la función no está preparada el icono mostrado es una señal de prohibición que indica que la función no puede ser invocada. El grid muestra el número de réplicas y de réplicas activas por cada función, el registro de invocaciones y por último las acciones que se pueden realizar sobre la función, siendo estas las de edición y/o borrado.

El listado de funciones disponibles se obtiene desde la API de OpenFaaS, a la que se ataca mediante la librería *Axios* [26], recomendada en la documentación oficial de Vue.js.

```
loadFunctions() {
    axios.get(this.openFaaS.endpoint)
        .then((response) => {
            // handle success
            this.functions = response.data.map((func) => {
                return {
                    name: func.name,
                    envProcess: func.envProcess,
                    image: func.image,
                    availableReplicas: func.availableReplicas,
```

```
                    replicas: func.replicas,
                    invocationCount: func.invocationCount,
                    ready: (Number(func.availableReplicas) > 0)
                }
            })
            this.loading = false
        })
        .catch((error) => {
            // handle error
            window.getApp.$emit('APP_SHOW_SNACKBAR',{text:
error.response.data, color: 'error' })
        })
}
```

Axios es una librería bastante simple de utilizar, pero al mismo tiempo es muy potente ya que ofrece una interfaz basada en los métodos HTTP, es decir, GET, PUT, POST y DELETE entre otros, además cada petición devuelve una promesa asegurando así la finalización de cada llamada. Como muestra el código anterior para obtener el listado de funciones se utiliza el método get(), al que se le pasa como parámetro la URL que expone el API de OpenFaaS. Llegados a este punto, es importante mencionar que, para el desarrollo en local y por seguridad, el navegador impide realizar las peticiones de origen cruzado.

"El Intercambio de Recursos de Origen Cruzado (CORS) es un mecanismo que utiliza encabezados adicionales HTTP para permitir que un user agent obtenga permiso para acceder a recursos seleccionados desde un servidor, en un origen distinto (dominio), al que pertenece. Un agente crea una petición HTTP de origen cruzado cuando solicita un recurso desde un dominio distinto, un protocolo o un puerto diferente al del documento que lo generó."[27]

Cuando se despliega el servidor de desarrollo se ejecuta en http://localhost:8080 y las peticiones al servidor de OpenFaaS, que está desplegado en el clúster Kubernetes, se encuentra en la dirección http://192.168.99.100:31112. Por tanto, para el navegador se está produciendo un intercambio de recursos de origen cruzado y bloquea la petición. La solución al problema durante el tiempo de desarrollo es bastante fácil, ya que basta con crear un proxy inverso en el servidor web para que redirija las peticiones hechas a una determinada ruta dentro del mismo dominio del servidor web al servidor de OpenFaaS. A continuación, se muestra el código del servidor proxy.

```
// vue.config.js
```

```
module.exports = {
    devServer: {
      proxy: {
        '/system': {
          target: 'http://192.168.99.100:31112/',
          ws: true,
          changeOrigin: true
        }
      }
    }
  }
```

La configuración anterior, indica que todas las peticiones que apunten a http://localhost:8080/system/* se redirijan automáticamente a la dirección de OpenFaaS, que en este caso es http://192.168.99.100:31112/.

Una vez solucionado el problema con el control de acceso HTTP, queda por mostrar cómo se editan, eliminan y crean funciones serverless desde la interfaz de OSCAR. Si el usuario desea eliminar una función basta con que haga clic sobre el icono de la papelera del grid. En este caso se pide una confirmación y si el usuario confirma la acción, se procede con la eliminación. Para ello se realiza una petición mediante Axios al API de OpenFaaS, pero ahora utilizando el método *delete()*. Como peculiaridad, hay que destacar que, para indicar a OpenFaaS la función que se desea eliminar, es necesario pasar un objeto data con el nombre de la función. Esto se puede hacer en Axios pasando el objeto *data* como segundo parámetro del método *delete()*.

```
deleteFunction(func) {
    const index = this.functions.indexOf(func)
    if (confirm('Are you sure you want to delete this function?')) {
        axios.delete(this.openFaaS.endpoint, {
            data: {
                functionName: func.name
            }
        })
            .then((response) => {
                // handle success
                this.functions.splice(index, 1)
                window.getApp.$emit('APP_SHOW_SNACKBAR', { text: `Function
${func.name} was deleted`, color: 'success' })
            })
            .catch((error) => {
                // handle error
```

```
                window.getApp.$emit('APP_SHOW_SNACKBAR', { text:
error.response.data, color: 'error' })
            })
    }
}
```

Como se puede ver en el código fuente anterior, si la promesa se cumple, se elimina la función del data-table y se muestra una notificación al usuario. En caso contrario se muestra una notificación con el error obtenido por parte del servidor.

Cuando el usuario decide modificar una función pulsando sobre el icono del lápiz de la celda de acciones del grid, se muestra una ventana emergente con la configuración que ya tiene almacenada la función. El campo "Docker image" corresponde a la imagen Docker que debe obtener OpenFaaS para crear la función. Es importante mencionar que la imagen Docker debe estar almacenada en un repositorio publico de Docker Hub para que OpenFaaS pueda acceder a ella. El siguiente campo no es editable, ya que el nombre de la función es el identificador interno y no puede ser modificado.

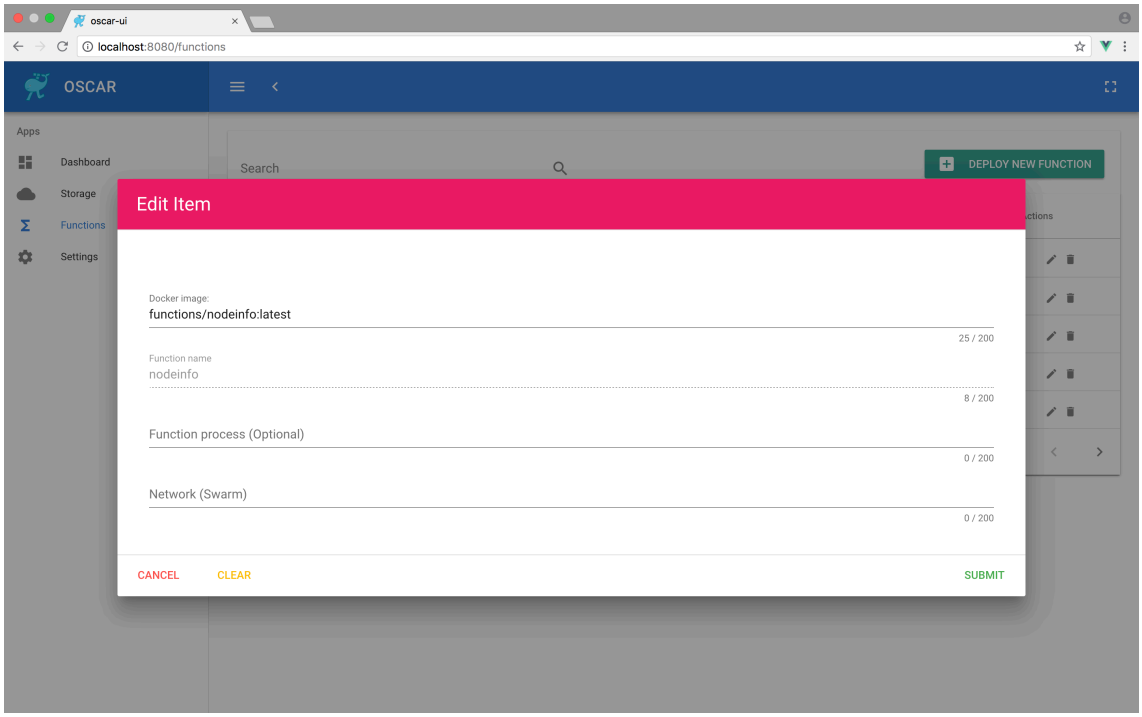

ILUSTRACIÓN 18: MODIFICACIÓN DE FUNCIONES

El tercer campo de la ventana de modificación es "Function Process", aunque es opcional puede que en ocasiones sea necesario rellenarlo ya que es el comando que ejecutará el contenedor Docker al ser desplegado. Por último, el campo "Network" solo será necesario para despliegues sobre Docker Swarm. OSCAR estará desplegado sobre Kubernetes, por tanto, este campo no debería de utilizarse.

Siguiendo la filosofía de desarrollo de Vue.js, se creó un componente que alberga el formulario de edición/creación de funciones. Con esto se consigue reutilizar el componente para los dos escenarios posibles. Aunque la información a rellenar sea la misma en ambos casos, la ventana de edición de funciones se visualiza con la barra de título de color fucsia y en el caso que se desee desarrollar una función nueva, se visualiza la barra de título de color azul.

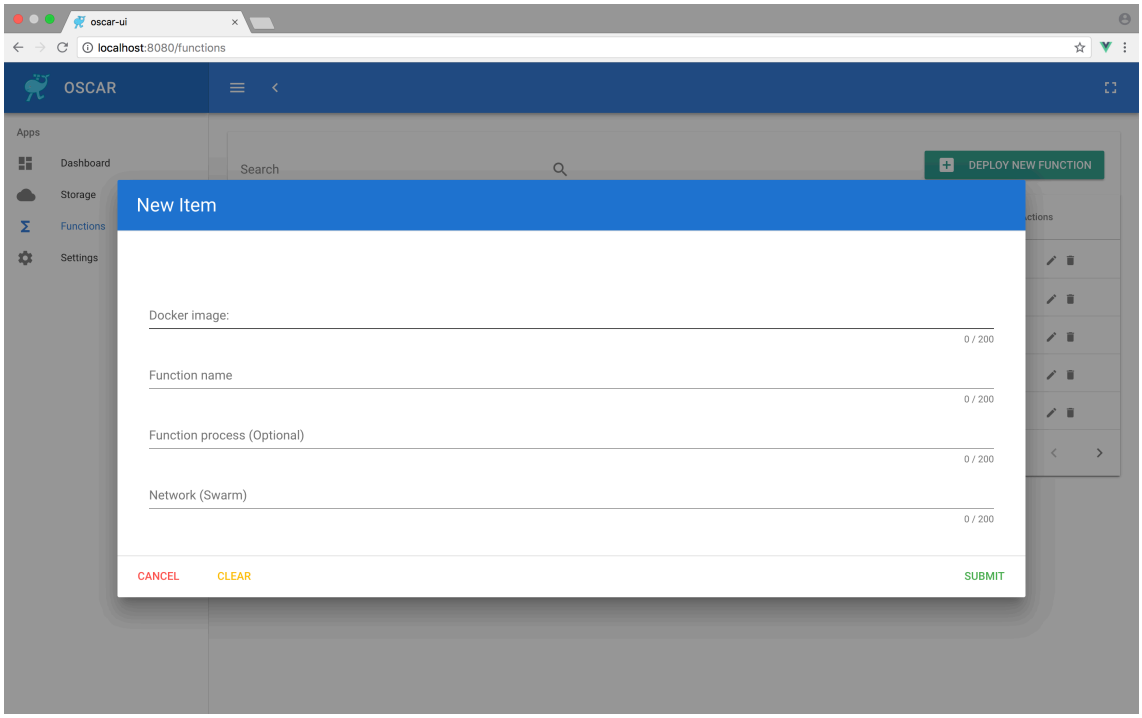

ILUSTRACIÓN 19: DESARROLLO DE NUEVAS FUNCIONES

A nivel de peticiones al servidor, la principal diferencia entre ambas pantallas es el método utilizado en Axios para hacer la llamada. Cuando se está editando una función el método http que se debe utilizar es PUT y en el caso de creación de una función nueva es el método POST. A continuación, se muestran ambos métodos:

```
newFunction() {
    axios.post(this.openFaaS.endpoint, {
        service: this.form.name,
        network: this.form.network,
        image: this.form.image,
        envProcess: this.form.process
    }).then((response) => {
        // handle success
        window.getApp.$emit('APP_SHOW_SNACKBAR', { text: `Function
${this.form.name} has been deployed`, color: 'success' })
        this.dialog = false
        this.clear()
        this.updateFunctionsGrid()
    }).catch((error, data) => {
```

```
        // handle error
        window.getApp.$emit('APP_SHOW_SNACKBAR', { text: error.response.data,
color: 'error' })
    }).then(() => {
        this.progress.active = false
    })
},
editFunction() {
    axios.put(this.openFaaS.endpoint, {
        service: this.form.name,
        network: this.form.network,
        image: this.form.image,
        envProcess: this.form.process
    }).then((response) => {
        // handle success
        window.getApp.$emit('APP_SHOW_SNACKBAR', { text: `Function
${this.form.name} has been updated`, color: 'success' })
        this.dialog = false
        this.clear()
        this.updateFunctionsGrid()
    }).catch((error, data) => {
        // handle error
        window.getApp.$emit('APP_SHOW_SNACKBAR', { text: error.response.data,
color: 'error' })
    }).then(() => {
        this.progress.active = false
    })
},
updateFunctionsGrid() {
    window.getApp.$emit('FUNC_GET_FUNCTIONS_LIST')
}
```

Como en el caso de eliminación de funciones, es necesario pasar como segundo parámetro del método un objeto con la información de la función que se desea crear o modificar. Por último, es necesario comunicar al componente padre que se ha insertado o modificado una función y debe actualizar la lista. Por tanto, es necesario emitir un evento desde el componente hijo para que el padre actualice el grid con la información nueva. El siguiente código muestra cómo el componente padre se suscribe al evento `'FUNC_GET_FUNCTIONS_LIST'` para detectar cuando el formulario de modificación e inserción de funciones le indique que se ha hecho una modificación en el servidor y debe actualizar la lista de funciones.

```
created: function () {
    window.getApp.$on('FUNC_GET_FUNCTIONS_LIST', () => {
      this.loadFunctions()
    })
  },
```

3.2.3.4. DASHBOARD

Por el momento, el dashboard únicamente muestra dos widgets: uno con el número de buckets disponibles en Minio y otro con el número de funciones disponibles en OpenFaaS.

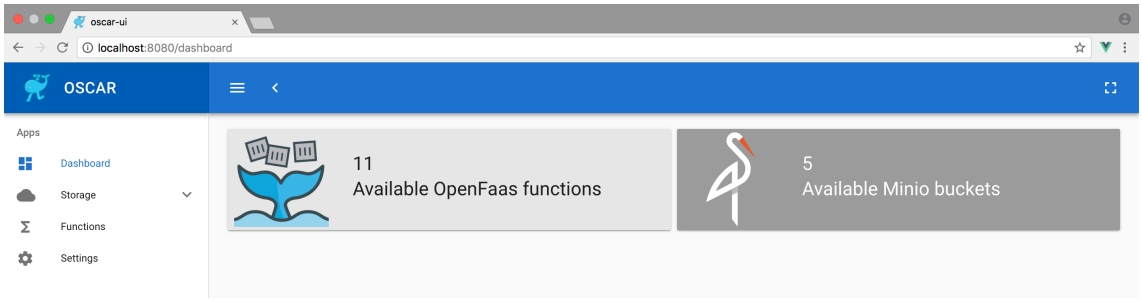

ILUSTRACIÓN 20: DASHBOARD DE LA APLICACIÓN

3.2.3.5. TOOLBAR

El componente toolbar de Vuetify ofrece la posibilidad de abrir la aplicación en modo de pantalla completa y de mostrar u ocultar el menú lateral de la aplicación, así como reducir su tamaño para mostrar únicamente los iconos como se puede observar en las siguientes imágenes.

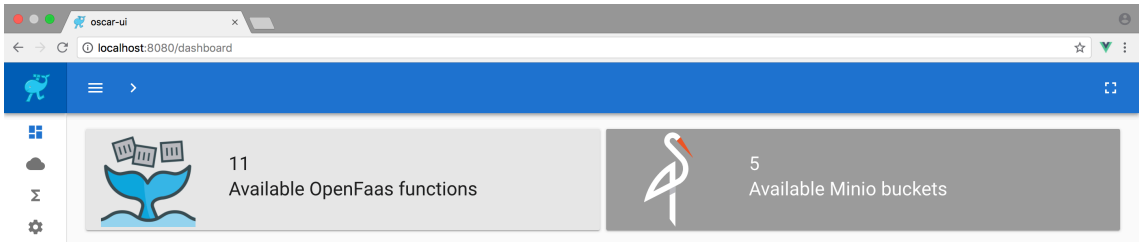

ILUSTRACIÓN 21: MENÚ MINIMIZADO

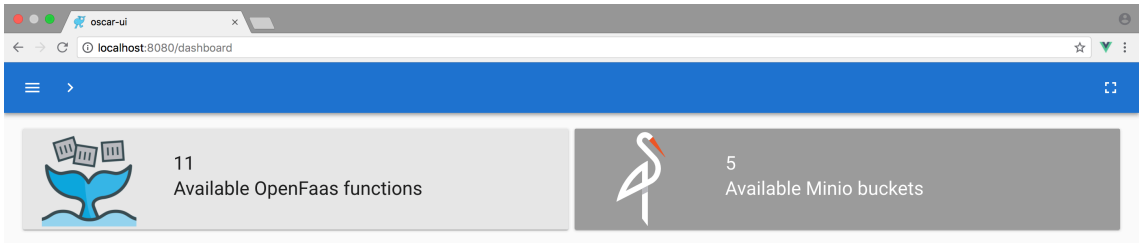

ILUSTRACIÓN 22: MENÚ OCULTO

3.2.3.6. COMPONENTE APP-DRAWER

El componente app-drawer de Vuetify permite crear un menú a la parte izquierda la pantalla. Este componente alberga la responsabilidad de enlazar las páginas dentro de la aplicación. En otras palabras, es el encargado de la navegación dentro de la interfaz. Es importante destacar que para permitir la navegación entre páginas se ha utilizado el componente vue-router, el cual permite a vue.js crear rutas para simular una navegación entre páginas, creando en el usuario la percepción de navegación. En cambio vue.js realmente no cambia de página, sino que inyecta la vista asociada a la ruta dentro del componente `<router-view>`. En Vue.js existen rutas dinámicas las cuales permiten utilizar comodines para mostrar una información u otra dependiendo del valor del comodín. En este caso se utilizan rutas dinámicas para listar los objetos contenidos dentro de un bucket. El resto de las rutas de la aplicación son rutas fijas, es decir, no permiten la utilización de comodines.

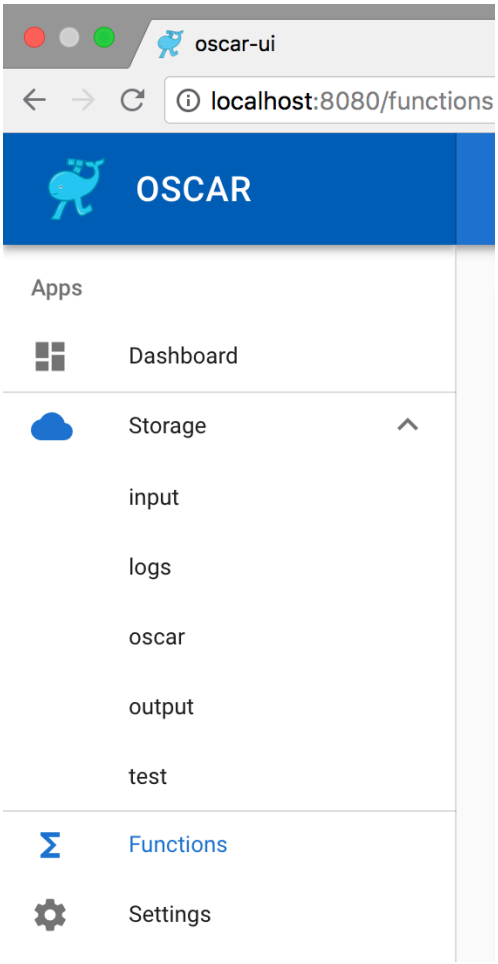

ILUSTRACIÓN 23: MENÚ DE LA APLICACIÓN

El código siguiente muestra las rutas accesibles desde el menú de la aplicación.

```
{
    path: '/',
    meta: {
      public: false
    },
    name: 'Root',
    redirect: {
      name: 'Dashboard'
    }
  },
  {
    path: '/dashboard',
    meta: {
```

```
      public: false
    },
    name: 'Dashboard',
    component: () => import(
      /* webpackChunkName: "routes" */
      /* webpackMode: "lazy" */
      `@/views/Dashboard.vue`
    )
  },
  {
    path: '/functions',
    meta: {
      public: false
    },
    name: 'Functions',
    component: () => import(
      /* webpackChunkName: "routes" */
      /* webpackMode: "lazy" */
      `@/views/Functions.vue`
    )
  },
  {
    path: '/settings',
    meta: {
      public: false
    },
    name: 'Settings',
    component: () => import(
      /* webpackChunkName: "routes" */
      /* webpackMode: "lazy" */
      `@/views/Settings.vue`
    ),
    props: true
  },
  {
    path: '/buckets/:bucketName',
    meta: {
      public: false
    },
    name: 'BucketContent',
    component: () => import(
      /* webpackChunkName: "routes" */
      /* webpackMode: "lazy" */
      `@/views/BucketContent.vue`
    ),
    props: true
  }
```

Para definir una ruta en vue-router, es necesario establecer el path al que responderá, el nombre y la ubicación del componente o vista a la que hace referencia. También es posible definir metadatos dentro del objeto meta. En este caso se utiliza para indicar si la dirección es pública o solo para usuarios autenticados.

3.2.3.7. LOGIN

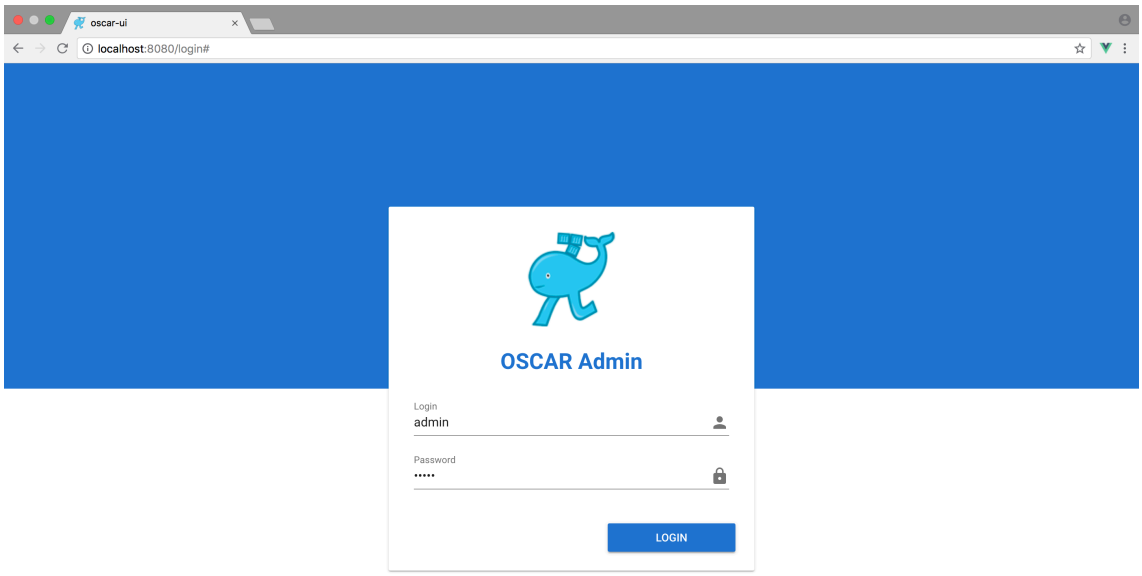

ILUSTRACIÓN 24: LOGIN OSCAR

La captura anterior muestra la pantalla de identificación de la aplicación. En estos momentos solamente acepta como usuario “admin” y como contraseña “admin” dado que no se ha dotado a la aplicación de un servicio de autenticación. Se espera poder integrar en el futuro un esquema de autenticación externo proporcionado por Amazon Cognito[28].

4. CONCLUSIONES Y TRABAJOS FUTUROS

En este proyecto se ha desarrollado una interfaz gráfica para acercar a científicos e investigadores de todas las áreas a la computación serverless, permitiendo así, crear funciones que ejecuten aplicaciones empaquetadas en imágenes Docker sin tener que aprovisionar previamente la infraestructura necesitada para la ejecución del programa.

Como herramientas principales se ha utilizado Kubernetes como orquestador de contenedores sobre el que se han desplegado OpenFaaS y Minio, permitiendo de esta forma el auto escalado de las funciones y el almacenamiento persistente de archivos en Minio. Para unificar ambos servicios se ha utilizado la herramienta Serverless Event Gateway, la cual permite crear suscripciones a eventos y reaccionar a ellos ejecutando

funciones serverless. En este caso, Minio envía una notificación indicando que en cierto bucket se ha creado un archivo. Cuando Event Gateway capta el evento, ejecuta la función serverless asociada. De esta forma es posible reaccionar a la creación de un objeto en un bucket de Minio.

La interfaz se ha desarrollado utilizando Vue.js. Pensando en la facilidad de uso y lo extendido que está el uso de Material Design, se decidió utilizar la librería de componentes Vuetify la cual acerca la filosofía Material Design a Vue.js. Desde la interfaz desarrollada, el usuario puede crear, editar y eliminar funciones de OpenFaaS de forma muy sencilla, así como también puede crear y eliminar buckets de Minio. Siguiendo con la funcionalidad, el usuario puede desarrollar las cuatro funciones básicas CRUD sobre objetos dentro de cada bucket.

Hay que destacar que el proyecto ha sido desarrollado usando técnicas de desarrollo de software modernas y el código fuente ha sido gestionado a través de un repositorio de GitHub que, eventualmente será abierto a la comunidad mediante licencia Apache 2.0, una vez haya sido integrado en la plataforma OSCAR.

La creación de una interfaz de usuario web fácil de utilizar posibilitará la adopción de las tecnologías de serverless computing por parte de usuarios científicos no experimentados, con el objetivo de que puedan interactuar con una plataforma de computación avanzada por medio de interfaces bien conocidas (navegador web y ficheros a procesar).

Como trabajo pendiente o para futuras versiones del software se podría implementar un servicio de autenticación para que realmente se dispusiera de una aplicación multiusuario y pudiera ser utilizada por distintas personas al mismo tiempo sin intervenir uno el trabajo del otro. Es importante destacar que esta funcionalidad no supone una limitación al desarrollo actual, pues la interfaz está pensada para desplegarse de forma dinámica sobre una infraestructura virtualizada específicamente desplegada para un usuario, por lo que no es de esperar un uso *multi-tenant* de la misma.

Se había pensado desarrollar un sistema de notificaciones dentro de la aplicación y por correo electrónico que permitiera enviar/mostrar una notificación al usuario indicando que cierta función ha finalizado y el nombre del archivo de salida que ha generado. Por otra parte, también seria conveniente dotar al dashboard de mas funcionalidad. Por ejemplo, añadir un gráfico de barras con las invocaciones realizadas a cada función. Otro con la cantidad de recursos utilizados por cada función y un indicador del número de funciones en ejecución. Por último, se podría añadir un diagrama que indicara el espacio en disco utilizado por cada bucket de Minio.

5. BIBLIOGRAFÍA

[1] A. Pérez, G. Moltó, M. Caballer, and A. Calatrava, "Serverless computing for container-based architectures," *Futur. Gener. Comput. Syst.*, vol. 83, pp. 50–59, 2018.

[2] "GitHub - grycap/oscar: On-premises Serverless Container-aware ARchitectures." [Online]. Available: https://github.com/grycap/oscar. [Accessed: 14-Sep-2018].

[3] "EGI | EGI Advanced Computing Services for Research." [Online]. Available: https://www.egi.eu/. [Accessed: 14-Sep-2018].

[4] "GitHub - nuclio/nuclio: High-Performance Serverless event and data processing platform." [Online]. Available: https://github.com/nuclio/nuclio. [Accessed: 14-Sep-2018].

[5] "GitHub - fnproject/fn: The container native, cloud agnostic serverless platform." [Online]. Available: https://github.com/fnproject/fn. [Accessed: 14-Sep-2018].

[6] "GitHub - kubeless/kubeless: Kubernetes Native Serverless Framework." [Online]. Available: https://github.com/kubeless/kubeless. [Accessed: 14-Sep-2018].

[7] "GitHub - fission/fission: Fast Serverless Functions for Kubernetes." [Online]. Available: https://github.com/fission/fission. [Accessed: 14-Sep-2018].

[8] "GitHub - openfaas/faas: OpenFaaS - Serverless Functions Made Simple for Docker & Kubernetes." [Online]. Available: https://github.com/openfaas/faas. [Accessed: 14-Sep-2018].

[9] "Production-Grade Container Orchestration." [Online]. Available: https://kubernetes.io/. [Accessed: 14-Sep-2018].

[10] "Home | OpenFaaS - Serverless Functions Made Simple." [Online]. Available: https://www.openfaas.com/. [Accessed: 14-Sep-2018].

[11] "Minio: Private cloud storage." [Online]. Available: https://minio.io/. [Accessed: 14-Sep-2018].

[12] "Event Gateway." [Online]. Available: https://serverless.com/event-gateway/. [Accessed: 14-Sep-2018].

[13] "Using kubectl to Create a Deployment - Kubernetes." [Online]. Available: https://kubernetes.io/docs/tutorials/kubernetes-basics/deploy-app/deploy-intro/. [Accessed: 14-Sep-2018].

[14] "Viewing Pods and Nodes - Kubernetes." [Online]. Available: https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/. [Accessed: 14-Sep-2018].

[15] “Minio Kubernetes.” [Online]. Available: https://www.minio.io/kubernetes.html. [Accessed: 14-Sep-2018].

[16] OpenFaaS, “GitHub - openfaas/faas-netes: Serverless Kubernetes with OpenFaaS (Functions as a Service).” [Online]. Available: https://github.com/openfaas/faas-netes. [Accessed: 14-Sep-2018].

[17] Inc. Amazon Web Services, “Estructura de mensaje de evento - Amazon Simple Storage Service.” [Online]. Available: https://docs.aws.amazon.com/es_es/AmazonS3/latest/dev/notification-content-structure.html. [Accessed: 14-Sep-2018].

[18] You Evan, “Introduction — Vue.js.” [Online]. Available: https://vuejs.org/v2/guide/. [Accessed: 14-Sep-2018].

[19] V. LLC, “Vue.js Material Component Framework — Vuetify.js.” [Online]. Available: https://vuetifyjs.com/en/. [Accessed: 14-Sep-2018].

[20] “Design - Material Design.” [Online]. Available: https://material.io/design/. [Accessed: 14-Sep-2018].

[21] “webpack.” [Online]. Available: https://webpack.js.org/concepts/. [Accessed: 14-Sep-2018].

[22] “Introduction | Vue Loader.” [Online]. Available: https://vue-loader.vuejs.org/. [Accessed: 14-Sep-2018].

[23] “What is Vuex? | Vuex.” [Online]. Available: https://vuex.vuejs.org/. [Accessed: 14-Sep-2018].

[24] “The Vue Instance — Vue.js.” [Online]. Available: https://vuejs.org/v2/guide/instance.html#Lifecycle-Diagram. [Accessed: 14-Sep-2018].

[25] “Usar promesas - JavaScript | MDN.” [Online]. Available: https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Usar_promesas. [Accessed: 14-Sep-2018].

[26] “GitHub - axios/axios: Promise based HTTP client for the browser and node.js.” [Online]. Available: https://github.com/axios/axios. [Accessed: 14-Sep-2018].

[27] “Control de acceso HTTP (CORS) - HTTP | MDN.” [Online]. Available: https://developer.mozilla.org/es/docs/Web/HTTP/Access_control_CORS. [Accessed: 14-Sep-2018].

[28] “AWS | Gestión de identidades y autenticación de usuario en la nube.” [Online]. Available: https://aws.amazon.com/es/cognito/. [Accessed: 14-Sep-2018].